Васильев А.Н.

ПРОГРАММИРОВАНИЕ НА JavaScript

В ПРИМЕРАХ И ЗАДАЧАХ

- Основы веб-разработки с «нуля»
- Концепция и основные принципы ООП
- От сценариев и инструкций к элементам управления и событиям
- Наглядные примеры и их разбор с пояснениями автора
- Подходит для самостоятельного обучения

РОССИЙСКИЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Васильев А.Н.

ПРОГРАММИРОВАНИЕ НА JavaScript

В ПРИМЕРАХ И ЗАДАЧАХ

Москва
2017

УДК 004.43
ББК 32.973.26-018.2
В19

Васильев, Алексей Николаевич.

В19 JavaScript в примерах и задачах / Алексей Васильев. – Москва : Издательство «Э», 2017. – 720 с. – (Российский компьютерный бестселлер).

ISBN 978-5-699-95459-9

Простой и интересный самоучитель по JavaScript, наиболее популярному сегодня языку программирования во всем мире. Полный спектр сведений о языке JavaScript с примерами и разбором задач от автора учебников-бестселлеров по языкам программирования Алексея Васильева. С помощью этой книги освоить язык JavaScript сможет каждый желающий – от новичка до специалиста.

УДК 004.43
ББК 32.973.26-018.2

ISBN 978-5-699-95459-9

ОГЛАВЛЕНИЕ

ЧАСТЬ III. ИСПОЛЬЗОВАНИЕ JAVASCRIPT

Вступление
КНИГА О JAVASCRIPT

Видел чудеса техники, но такого...

из к/ф «Иван Васильевич меняет профессию»

Вниманию читателя предлагается книга о JavaScript. На сегодня JavaScript является одним из наиболее популярных языков программирования. Причем не просто программирования, а веб-программирования. Вместе с тем язык JavaScript в некоторых аспектах особенный. Поэтому, прежде чем приступить к его изучению, имеет смысл прокомментировать общее положение дел.

Как правило, JavaScript упоминают как *сценарный* язык. Другими словами, обычно на JavaScript создаются *сценарии*. Сценарий — это та же программа. Различие между программой и сценарием в том, под управлением какой среды они выполняются. Обычная программа чаще всего выполняется под управлением операционной системы. Сценарий выполняется под управлением *браузера*. Браузер, в свою очередь, представляет собой специальную программу, с помощью которой просматриваются веб-документы.

> **НА ЗАМЕТКУ**
>
> Браузеров существует достаточно много. Наиболее популярными (на момент написания книги) являются *Internet Explorer*, *Mozilla Firefox*, *Opera* и *Google Chrome*. Среди браузеров существует достаточно сильная конкуренция. Поэтому лидеры списка время от времени меняются, появляются новые браузеры, а уже существующие уходят со сцены. В общем, вопрос выбора браузера непростой и достаточно динамичный в плане предпочтений пользователей.

Указанное различие между программой и сценарием во многом техническое. Тем не менее оно имеет последствия. Собственно, об этих последствиях и поговорим далее.

НА ЗАМЕТКУ

То, что описывается далее, имеет некоторый привкус «технических подробностей». Эти подробности важные, но не критичные. Поэтому, даже если что-то покажется малопонятным, отчаиваться не стоит. Данный материал скорее для тех, кому интересно знать больше, чем необходимо.

Сценарии и программы

Меня не проведешь. Приемы сыщиков я вижу на пять футов вглубь.

из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Что сценарий, что программа представляют собой набор инструкций, которые следует выполнить. Вопрос только в том, кто или что данные инструкции будет выполнять. Мы уже знаем, что программа выполняется под управлением операционной системы. Как это происходит? Достаточно просто и прозаично. Общая последовательность действий при создании и выполнении программы выглядит примерно следующим образом.

- Сначала набирается программный код. Это как раз та последовательность команд, которая должна быть выполнена. Программный код набирается в соответствии с правилами языка программирования, на котором пишется программа. Языков программирования очень много — например, C++, C#, Java или Python (этот перечень языков программирования далеко не полный). У каждого из них свои правила составления кодов.
- После того как код набран и сохранен, его следует выполнить. Проблема в том, что набранный нами код понятен для нас, но непонятен для компьютера. Необходима программа-посредник, которая переведет понятный для человека код в команды, понятные для компьютера (или, точнее, операционной системы, установленной на компьютере). В качестве такого посредника выступает или *компилятор*, или *интерпретатор*. Для каждого языка программирования предназначен свой персональный компилятор или интерпретатор.
- Если используется программа-компилятор, то исходный программный код компилируется и в результате получается набор

команд, готовых к выполнению операционной системой. Фактически при компиляции на основе набора инструкций, написанных на каком-то определенном языке программирования, создается набор команд машинного или квазимашинного уровня. Эти команды и выполняются при запуске программы на выполнение.

- Принципиальное отличие интерпретатора от компилятора состоит в том, что интерпретатор выполняет трансляцию исходного кода в исполняемый код, так сказать, в процессе выполнения, инструкция за инструкцией. Если компилятором конвертируется вся программа, а затем начинается ее выполнение, то интерпретатор конвертирует и исполняет сначала одну инструкцию, затем вторую, третью и так далее. То есть конвертирование программного кода происходит по мере необходимости.

Описанная последовательность действий достаточно условная, но вместе с тем показательная. Она дает ответ на вопрос, что же нам на самом деле нужно, чтобы написать программу на том или ином языке программирования. Во всяком случае, становится понятно, что без компилятора (или интерпретатора — все зависит от конкретного языка программирования) не обойтись.

В общем и целом получается так: имеется некоторый язык программирования, на котором мы собираемся писать программные коды. Язык программирования — это в широком смысле набор правил. Поэтому формально программный код можно написать, имея под рукой лишь обычный текстовый редактор. Но затем нам понадобится программа, способная не просто «понять» этот код, но и преобразовать его в форму, приемлемую для восприятия операционной системой. Здесь на сцену выходит компилятор или интерпретатор. Компиляторы и интерпретаторы предназначены для компилирования и интерпретации программ. Для этого они создаются фирмами-разработчиками. Это их основная задача. Что здесь важно? Важно то, что написание кода и его компиляция/интерпретация разнесены во времени и в пространстве. Другими словами, создавая программный код, мы можем особо не заботиться, как затем его компилировать или интерпретировать.

Еще один важный момент связан непосредственно с компиляторами и интерпретаторами. Это программы, которые *устанавливаются* на компьютер. Проще говоря, если мы хотим писать (и запускать) программы на языке C++, то нам необходимо будет установить компилятор для данного языка (и для данной операционной системы).

ДЕТАЛИ

Откровенно говоря, очень редко компилятор устанавливается отдельно. Обычно устанавливается *интегрированная среда разработки* (обычно упоминается как *IDE*, что является сокращением от английского *Integrated Development Environment*), в состав которой входит не только компилятор, но и редактор кодов, встроенный отладчик и многие другие полезные утилиты. Со средой разработки написание программы превращается в процесс комфортный и где-то даже приятный. Но в данном случае это не важно. Даже если используется среда разработки, компилятор все равно устанавливается и используется.

Все сказанное, напомним, относится к программам. Теперь вспомним о сценариях. Что принципиально меняется при написании сценариев? На самом деле, мало что. Как и в случае с программой, при написании сценария в соответствии с правилами языка (в данном случае имеется в виду язык JavaScript) набирается программный код. Код будет выполняться при выполнении сценария. Нас интересует создание сценариев на языке JavaScript. Сценарии на языке JavaScript *интерпретируются*. Но интерпретация выполняется не в явном виде, а, как отмечалось, средствами браузера. Беды в том нет, но есть определенное неудобство, связанное с необходимостью «инкапсуляции» сценария в веб-документ. Дело в том, что при интерпретации программы, с одной стороны, имеется программный код, а с другой — интерпретатор. Программный код интерпретируется интерпретатором. А вот если речь идет о сценарии, то этот сценарий должен быть включен в веб-документ. При открытии веб-документа браузером сценарий выполняется встроенными средствами браузера.

НА ЗАМЕТКУ

Здесь мы имеем в виду разработку с помощью JavaScript клиентских веб-приложений. То есть речь идет о приложениях, которые выполняются на компьютере клиента. С серверными приложениями ситуация несколько иная.

Фактически интерпретатор для сценария «спрятан» в браузере. Выполнить единственно сценарий при таком подходе проблематично. Сценарий приходится выполнять в контексте работы с веб-документом. Само по себе это не хорошо и не плохо. Но с методологической точки зрения при изучении языка JavaScript ситуация не самая луч-

шая, поскольку в «компании» с JavaScript-кодом сценария оказывается еще и код веб-документа. Причина в том, что основное назначение браузеров — работа с веб-документами. Выполнение сценариев браузерами — в известном смысле дополнительная функция.

НА ЗАМЕТКУ

Великий немецкий философ *Иммануил Кант* (1724–1804) утверждал, что к человеку должно относиться только как к цели, но не как к средству (*категорический императив Канта*). В рамках данной философской парадигмы, пожалуй, можно утверждать, что, тогда как программа для интерпретатора является целью, сценарий для браузера скорее является средством.

ДЕТАЛИ

Ситуация со сценариями (в плане выполнения кода под управлением не операционной системы, а другой программы — в данном случае браузера) не является уникальной. Например, существуют *макросы*. Скажем, при работе с приложением Excel из пакета Microsoft Office на языке VBA можно создавать программные коды, выполняемые под управлением приложения Excel. Это и есть макросы.

Здесь имеет смысл отметить, что при создании веб-документов используется специальный язык, который называется *языком гипертекстовой разметки*, или *HTML* (сокращение от английского *HyperText Markup Language*). Язык HTML не является предметом книги, но не упомянуть его совсем не удастся. Покоряясь неизбежному, предадимся рассуждениям о языке HTML.

Веб-документы и язык HTML

> Он начинает новую жизнь, дайте ему возможность вспомнить все лучшее.
>
> *из к/ф «Покровские ворота»*

Концепция языка гипертекстовой разметки HTML достаточно проста. В стандартный (обычный) текст добавляются специальные коды, которые принято называть *тегами* или *дескрипторами*. Данные коды представляют собой инструкции для браузера. Инструкции касаются способа отображения текста, помеченного дескрипторами. Поэто-

му на браузер в некотором приближении можно смотреть как на программу для просмотра текстов с гипертекстовой разметкой.

ДЕТАЛИ

Откровенно говоря, ситуация не такая простая, как может показаться на первый взгляд. Один и тот же документ с гипертекстовой разметкой в разных браузерах может отображаться по-разному. Более того, есть дескрипторы, которые «понимаются» одними браузерами и совершенно «не признаются» другими. Подход, базирующийся на создании HTML-документов, ориентированных на работу с одним определенным типом браузера, не проходит, поскольку концептуально веб-документы предназначены для использования в Интернете, что автоматически сводит на нет попытку ограничить пользователей документа браузером одного типа. То есть даже на уровне стандартного (не использующего сценарии) HTML-документа возникают неожиданные моменты. С другой стороны, для каждой проблемы обычно имеется более или менее удачное решение.

Создать HTML-документ достаточно просто. Из всех возможных простых способов мы здесь рассмотрим самый простой и наименее ресурсозатратный. Для этого понадобится простенький текстовый редактор вроде Notepad (*Блокнот*) и, естественно, браузер.

НА ЗАМЕТКУ

Браузер в принципе подойдет любой, но мы будем ориентироваться на группу лидеров: Internet Explorer, Mozilla Firefox, Google Chrome и Opera.

Сначала в текстовом редакторе необходимо набрать код документа. Под словом «код» имеется в виду HTML-код. Чтобы не быть голословными, поступим следующим образом: создадим пустой текстовый документ. Этот документ необходимо открыть и в окне текстового редактора ввести код, представленный в листинге В.1 (назначение инструкций кода поясняется позже).

Листинг В.1. Код гипертекстовой разметки документа

```
<!DOCTYPE HTML>
<html>
  <head>
```

```
    <title>
      Омар Хайям. Рубаи
    </title>
  </head>
  <body>
    <h3>Рубаи</h3>
    Чтоб мудро жизнь прожить, знать надобно немало,<br>
    Два важных правила запомни для начала:<br>
    Ты лучше голодай, чем что попало есть,<br>
    И лучше будь один, чем вместе с кем попало.
    <hr>
    <b>Омар Хайям</b>
  </body>
</html>
```

Далее необходимо сохранить изменения в документе, закрыть его и изменить расширение txt на html (или htm). Собственно, все: мы создали HTML-документ. Этот документ можно открыть с помощью браузера. Также можно его открыть с помощью текстового редактора. В последнем случае увидим содержимое документа, то есть его HTML-код. На рис. В.1 показан документ с HTML-кодом из листинга В.1, открытый в текстовом редакторе.

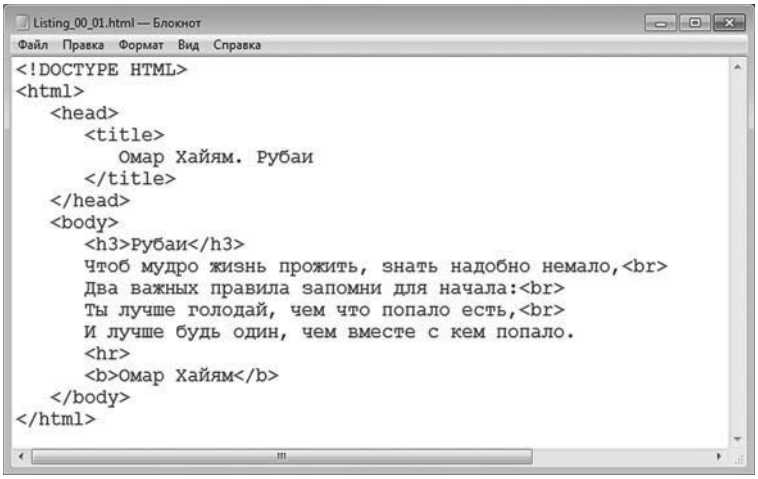

Рис. В.1. Документ с HTML-кодом открыт в текстовом редакторе

Если мы откроем документ с помощью браузера, получим несколько иной результат. На рис. В.2 показано, как будет выглядеть созданный нами документ в окне браузера Mozilla Firefox.

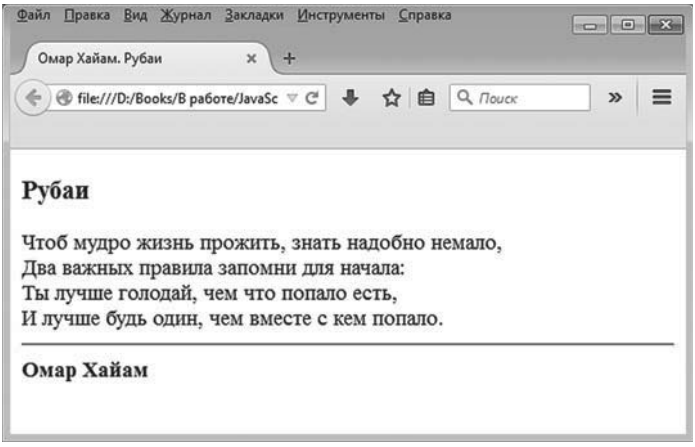

Рис. В.2. Документ открыт в окне браузера Mozilla Firefox

Для сравнения на рис. В.3 этот же документ открыт браузером Internet Explorer.

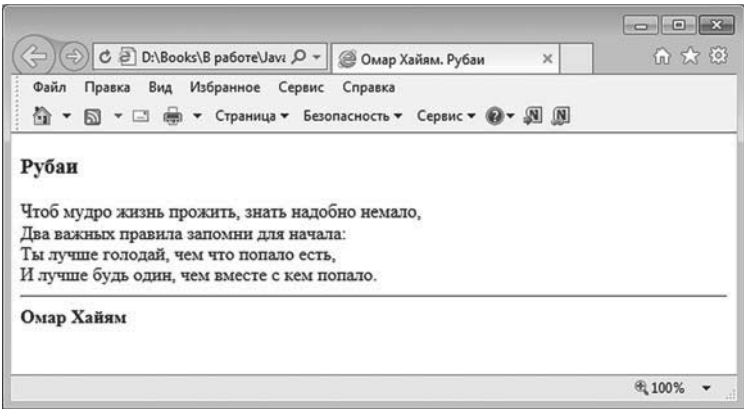

Рис. В.3. Документ открыт в окне браузера Internet Explorer

Также желающие могут взглянуть на рис. В.4 и рис. В.5, на которых показан документ с HTML-кодом, открытый соответственно с помощью браузеров Google Chrome и Opera (разработчик — компания Opera Software).

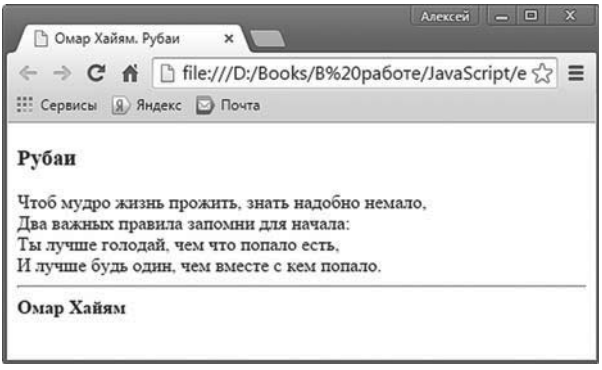

Рис. В.4. Документ открыт в окне браузера Google Chrome

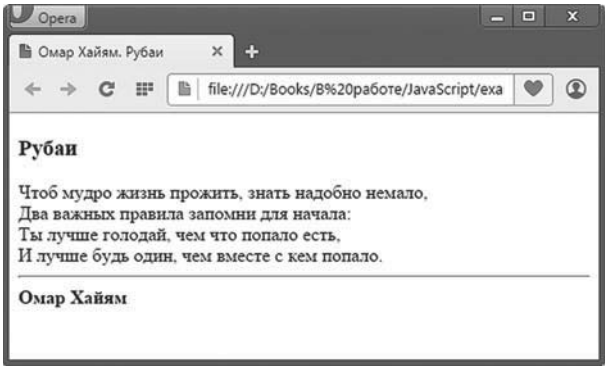

Рис. В.5. Документ открыт в окне браузера Opera

В принципе способ отображения документа разными браузерами идентичен. Так и должно быть. Теперь кратко обсудим назначение инструкций, использованных в HTML-коде документа.

> **НА ЗАМЕТКУ**
>
> В книге информация относительно HTML-кодировки обычно дается непосредственно при объяснении примеров или приводится в специальных врезках.

Сначала сделаем несколько общих замечаний относительно HTML-кодов. Во-первых, текст, не помеченный дескрипторами, отображается как обычно, то есть как текст. Во-вторых, дескрипторы бывают парными (таких большинство) и непарными. Дескриптор — это определенное ключевое слово, заключенное в угловые скобки. Если де-

скриптор парный, то второй (закрывающий) дескриптор отличается от первого (открывающего) дескриптора наличием обратной косой черты / перед ключевым словом в угловых скобках.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Дескриптор имеет вид <дескриптор>. Это открывающий дескриптор. Закрывающий дескриптор (если такой имеется) будет выглядеть как </дескриптор>. Например, пара дескрипторов и определяют фрагмент текста, который в браузере будет отображаться жирным шрифтом. Здесь речь идет о парном дескрипторе. Примеры непарных дескрипторов:

- дескриптор
 является инструкцией разрыва строки (в месте размещения данного дескриптора выполняется разрыв текстовой строки и осуществляется переход к новой строке для отображения текста);
- дескриптор <hr> является инструкцией отображения горизонтальной линии (по умолчанию на ширину окна браузера).

В-третьих, у HTML-документа должна быть определенная структура, которая формируется, как несложно догадаться, с помощью дескрипторов.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Откровенно говоря, браузеры очень демократичны в отношении правил оформления HTML-документа. Даже если взять самый обычный текстовый файл, добавить туда некоторые дескрипторы (например, выделить часть текста дескрипторами и), сохранить файл с расширением html и затем открыть его в браузере, браузер отобразит документ с учетом наличия в нем дескрипторов.

Вместе с тем существуют определенные правила создания документов с гипертекстовой разметкой, и их стоит придерживаться.

Использованный нами код содержит как парные, так и непарные дескрипторы. Начинается он с дескриптора <!DOCTYPE HTML>, являющегося стандартным началом HTML-документа, указывающим браузеру, с какого типа данными предстоит иметь дело. Весь фактический HTML-код размещается между дескрипторами <html> (открывает код документа) и </html> (закрывает код документа). Между дескрипторами <html> и </html> размещается несколько блоков кода (в данном случае блоков два). Первый блок выделен дескрипторами <head> и </head>.

Это заглавный блок документа. В данном блоке размещается важная информация о документе. В нашем примере в заглавном блоке между дескрипторами <title> и </title> указан текст Омар Хайям. Рубаи, который станет рабочим названием документа (не путать с названием файла!).

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Рабочее название документа, указанное между дескрипторами <title> и </title>, в рабочей области документа не отображается. Оно отображается на вкладке браузера, когда в нем открыт документ.

Второй блок выделен дескрипторами <body> и </body>. Этот блок — основное тело документа. В данном блоке фактически размещается код и текст, предназначенный для отображения в рабочей области документа. Что мы находим внутри блока? Внутри мы находим инструкцию <h3>Рубаи</h3>. В соответствии с ней текст Рубаи будет выделен как заголовок третьего уровня. О том, что речь идет о заголовке именно третьего уровня, свидетельствуют дескрипторы <h3> и </h3>.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Несложно догадаться, что если существует заголовок третьего уровня, то есть заголовок второго уровня (выделяется дескрипторами <h2> и </h2>) и заголовок первого уровня (выделяется дескрипторами <h1> и </h1>). Как выделять заголовок, определяется браузером. Обычно речь идет о применении увеличенного размера шрифта и выделении жирным стилем.

Далее следует текст, в котором встречаются непарный дескриптор
. В месте размещения данного дескриптора выполняется перенос строки. Непарный дескриптор <hr> используется для отображения в документе горизонтальной линии. При этом переход к новой строке после данной линии выполняется автоматически. Наконец, дескрипторы и используются для применения к тексту жирного шрифта.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

В написании дескрипторов можно использовать как большие, так и маленькие буквы. Например, вместо кода
 можем использовать код
. Данное замечание относится и к прочим дескрипторам.

Добавление сценария в веб-документ

— Что за вздор. Как вам это в голову взбрело?
— Да не взбрело бы, но факты, как говорится, упрямая вещь.

из к/ф «Чародеи»

Веб-документ может содержать не только дескрипторы, определяющие внешний вид документа при отображении его в браузере, но и особый программный код — *сценарий*. Сценарий в веб-документе помещается в специальный блок. Блок выделяется дескрипторами <script> и </script>. Блоков со сценариями в веб-документе может быть несколько, и находиться они могут в разных местах документа. Например, блок со сценарием разрешается размещать в заглавном блоке документа (выделен дескрипторами <head> и </head>). А можно блок со сценарием разместить в блоке основного тела документа (выделяется дескрипторами <body> и </body>). Именно так мы и будем поступать в дальнейшем — код сценариев будет содержаться внутри <body>-блока.

ДЕТАЛИ

В целом место размещения блока со сценарием имеет значение: сценарий выполняется по мере загрузки документа. Если сценарий находится в заглавном блоке веб-документа, то его выполнение начнется практически сразу при загрузке документа в браузер. Если сценарий находится в блоке основного тела документа, то выполняется он по мере загрузки блока с кодом. Но нас пока что такие детали интересовать не будут.

Прежде чем рассмотреть код веб-документа с инкапсулированным в него сценарием (очень простым), отметим еще одно немаловажное обстоятельство. Дело в том, что при добавлении блока сценария в веб-документ желательно указать, на каком языке программирования написан сценарий. В общем-то такая ситуация стандартна для HTML-документа: во многих случаях дескрипторы блоков содержат некоторые дополнительные настройки, влияющие на способ отображения того или иного элемента или фрагмента текста. Дополнительные настройки выполняются присваиванием значений *параметрам* или *атрибутам*. Значения атрибутам присваиваются внутри открывающего дескриптора (внутри угловых скобок после имени деструктора). Вся конструкция имеет вид <дескриптор атрибут=значение>. Здесь де-

скриптор обозначает имя дескриптора, атрибут обозначает имя атрибута, а значение (обычно заключается в двойные кавычки), соответственно, является значением атрибута.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Например, для создания гипертекстовой ссылки в документе используется парный дескриптор <a> (закрывающий дескриптор). У дескриптора есть атрибут href. Значение данного атрибута определяет адрес страницы для выполнения перехода при щелчке на гиперссылке. Более конкретно, инструкция Сайт автора книги добавляет в документ гиперссылку с текстом Сайт автора книги, щелчок на которой приводит к переходу по адресу www.vasilev.kiev.ua.

Для <script>-дескриптора желательно указать значение "text/javascript" для атрибута type. Фактически данное значение указывает, что сценарий написан на языке JavaScript. Шаблон включения блока со сценарием в веб-документ следующий:

```
<script type="text/javascript">
  // код сценария
</script>
```

Вся эта конструкция будет размещаться внутри <body>-блока, а непосредственно код сценария находится там, где размещен комментарий // код сценария.

НА ЗАМЕТКУ

Две обратных косых черты // в программном коде сценария являются началом *комментария*. Все, что находится справа от двойной косой черты, при выполнении сценария игнорируется.

Также стоит отметить, что, если не указать инструкцию type="text/javascript" в дескрипторе <script>, сценарий все равно будет выполняться. Дело в том, что по умолчанию, если явно не указан язык, на котором написан сценарий, предполагается язык JavaScript. Но все же лучше не полагаться на случай и указывать язык сценария в явном виде.

Как иллюстрацию к использованию сценариев рассмотрим небольшой пример, в котором вывод текста в диалоговое окно выполняется с помощью сценария, добавленного в HTML-код веб-документа.

Рассмотрим листинг В.2, в котором представлен соответствующий HTML-код, в том числе внутри этого кода имеется небольшой сценарий.

Листинг В.2. Документ со сценарием

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>
      Сценарий JavaScript
    </title>
  </head>
  <body>
    <h3>Документ со сценарием JavaScript</h3>
    При загрузке этого документа выполняется сценарий...<br>
    <b>Результат выполнения сценария:</b>
    <hr>
    <!-- Начало блока сценария -->
    <script type="text/javascript">
      // Команда вывода текста в окно документа:
      document.write("Сценарий на языке JavaScript")
    </script>
    <!-- Завершение блока сценария -->
    <hr>
    <b>Выполнение сценария завершено.</b>
  </body>
</html>
```

На рис. В.6 показано окно текстового редактора, в котором открыт соответствующий веб-документ, так что мы можем видеть, как в реальности выглядит документ с HTML-кодом и вставленным в него блоком сценария.

Прежде чем посмотреть, как будет выглядеть этот же самый документ в окне браузера, кратко проанализируем код документа — нас интересует прежде всего код сценария.

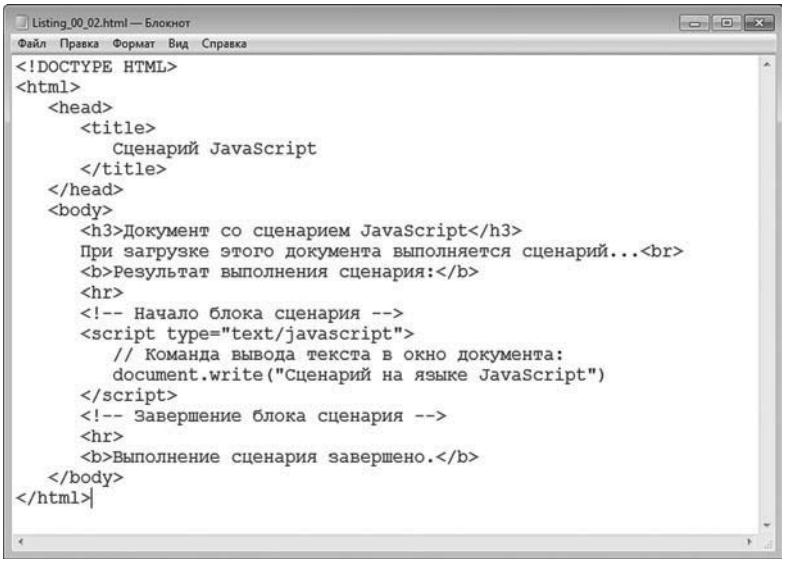

Рис. В.6. Веб-документ со сценарием открыт в текстовом редакторе

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

На всякий случай напомним назначение основных дескрипторов в HTML-коде. Итак, документ стандартно начинается инструкцией <!DOCTYPE HTML>, идентифицирующей принадлежность документа к семейству HTML. Весь код документа помещен между дескрипторами <html> и </html>. Внутри <head>-блока между дескрипторами <title> и </title> указывается рабочее название документа (отображается во вкладке браузера при отображении документа). То, что отображается в области документа, содержится в <body>-блоке. В частности, там имеется заголовок 3-го уровня (между дескрипторами <h3> и </h3>), обычный текст и текст, выделенный жирным шрифтом. Текст для выделения жирным шрифтом заключается между дескрипторами и . Для перехода к новой строке использована инструкция
. Инструкцией <hr> отображается горизонтальная линия (на ширину всей рабочей области окна браузера).

Программный код сценария размещен между дескрипторами <script> и </script>. Открывающий дескриптор содержит инструкцию type="text/javascript", определяющую язык программирования, на котором написан сценарий.

Также в HTML-коде представлены *комментарии*, которые выделяют блок со сценарием. В HTML комментарий начинается инструкцией <!-- и заканчивается инструкцией -->. Все, что находится между инструкциями <!-- и -->, при открытии веб-документа в окне браузера не отображается.

Внутри <script>-блока, если не считать комментарий (строка, которая начинается с двойной косой черты //), всего одна команда document.write("Сценарий на языке JavaScript"). Как несложно догадаться, данной командой в окно браузера выводится текст, указанный в двойных кавычках. Читателю, не знакомому с принципами объектно-ориентированного программирования, команда может показаться странной. Хотя на самом деле команда самая обычная: из объекта document вызывается метод write().

> **НА ЗАМЕТКУ**
>
> Далее приводится небольшое формальное объяснение назначения команды и смысла входящих в нее идентификаторов. Более детально объекты и методы разбираются в основной части книги. Здесь же даются лишь минимальные сведения, необходимые для общего понимания принципов выполнения программного кода.

Объект document отождествляется с веб-документом, открытым в окне браузера (документ, в котором размещен код сценария). С данным объектом можно выполнять некоторые действия. В частности, среди таких действий есть операция «отобразить в документе текст». Реализуется она методом write(). Чтобы отобразить в документе текст, необходимо вызвать метод write(), передав ему аргументом текст, предназначенный для отображения. Но метод существует не сам по себе, а только в привязке к объекту, с которым или по отношению к которому выполняются действия. Метод вызывается из объекта. Последнее означает, что при вызове метода должен быть явно указан и объект, из которого метод вызывается. Объект указывается первым, после чего через точку следует имя метода (используется так называемый *точечный синтаксис*). Во всем остальном метод похож на обычную функцию или процедуру: именованный блок программного кода, который вызывается через имя.

Здесь речь идет об объекте document и методе write(). Команда вызова метода write() из объекта document выглядит как document.write(). Только в круглых скобках нужно указать текст для отображения в документе. В результате получается команда document.write("Сценарий на языке JavaScript"). Помимо нее, в программном коде сценария имеется еще комментарий — он находится одной строкой выше команды вывода текста в документ и начинается с двойной косой черты. Назначение комментария — в пояснении смысла инструкций программного кода.

НА ЗАМЕТКУ

Обратите внимание, что комментарии в HTML-коде и в коде сценария выполняются по-разному. Комментарий для HTML-кода заключается между инструкциями <!-- и -->. Такие комментарии используются в HTML-коде, но не используются внутри программного кода сценария.

В сценарии комментарий начинается с двойной косой черты //. Вне сценария такие комментарии неприменимы.

Если загрузить веб-документ в окно браузера, получим результат, как, например, на рис. В.7. Там показан документ, открытый в окне браузера Internet Explorer.

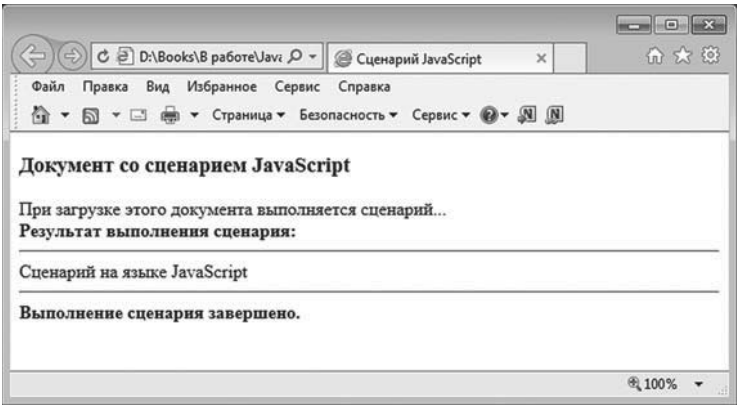

Рис. В.7. Документ со сценарием открыт в браузере Internet Explorer

Аналогичным образом все выглядит и при использовании других браузеров. Для сравнения на рис. В8 показано окно браузера Mozilla Firefox с открытым в нем документом со сценарием. Окно браузера Google Chrome с открытым документом показано на рис. В.9. Как будет выглядеть документ в окне браузера Opera, показано на рис. В.10.

Во всех перечисленных случаях результат примерно одинаков, с поправкой на незначительные декоративные детали. Важно также подчеркнуть, что результат выполнения сценария каждый раз отображается между двумя горизонтальными линиями. Если из всего текста в рабочей области браузера исключить содержимое, обязанное своим существованием HTML-коду документа (за исключением кода сценария), то результат выполнения исключительно сценария будет следующим.

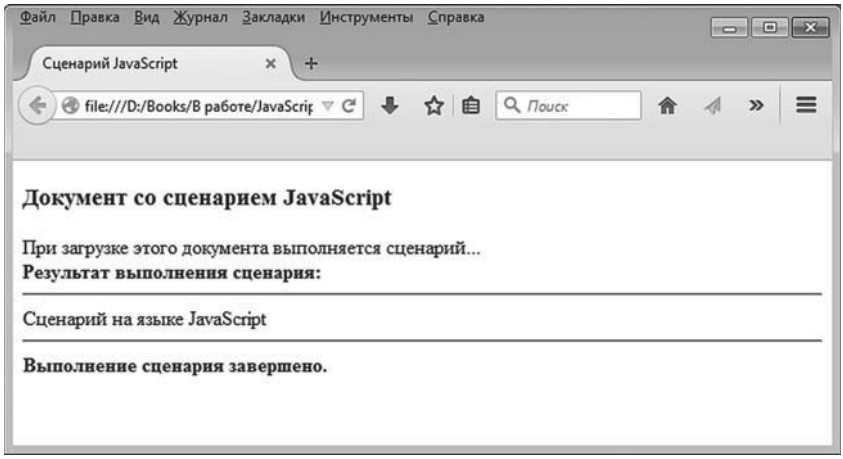

Рис. В.8. Документ со сценарием открыт в браузере Mozilla Firefox

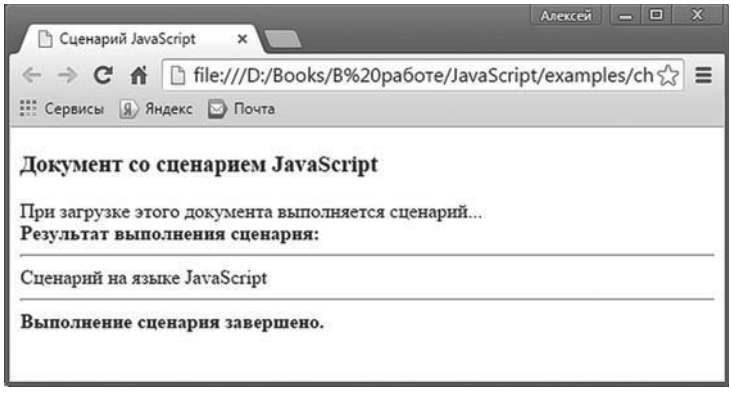

Рис. В.9. Документ со сценарием открыт в браузере Google Chrome

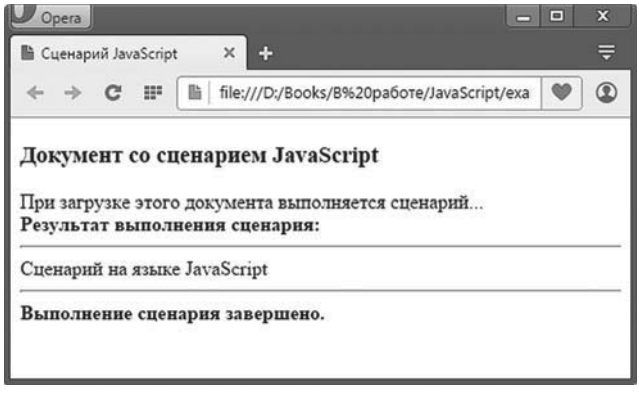

Рис. В.10. Документ со сценарием открыт в браузере Opera

Результат выполнения сценария (из листинга В.2)

```
Сценарий на языке JavaScript
```

Еще раз подчеркнем, что здесь речь идет о той части содержимого вебдокумента, которая отображается при выполнении сценария.

Концепция книги

> Ален ноби, ностра алис! Что означает — ежели один человек построил, другой завсегда разобрать может.
>
> *из к/ф «Формула любви»*

Как отмечалось выше, эта книга — о языке программирования JavaScript. Но здесь имеется некоторая проблема методологического характера. Не будет преувеличением сказать, что язык JavaScript достаточно специфичный — не в плане синтаксиса (хотя во многом он оригинальный), но в плане прикладного применения языка. Как отмечалось ранее, язык используется для написания сценариев, которые выполняются под управлением браузера.

НА ЗАМЕТКУ

Современные тенденции таковы, что область применимости языка JavaScript постоянно расширяется. На данный момент написание браузерных сценариев — далеко не единственное поле применения языка JavaScript. По-видимому, такой тренд сохранится и в будущем. Мы же ограничимся изучением языка JavaScript как такового, а также познакомимся с практической стороной применения JavaScript для написания сценариев.

Отделить собственно язык JavaScript от вопросов его прикладного применения не очень просто. Другими словами, обсуждая JavaScript, приходится постоянно делать рефрен в сторону прикладного аспекта сценарного программирования. Реализация программного взаимодействия с браузером подразумевает владение основами объектной модели документа. То есть речь фактически идет об объектно-ориентированном программировании. Причем в JavaScript объектно-ориентированный подход реализован специфически.

НА ЗАМЕТКУ

Будет много сюрпризов для тех, кто знаком с языками программирования C++, C# или Java.

И хотя ситуацию с реализацией объектно-ориентированной парадигмы в языке JavaScript нельзя назвать сложной, простой ее тоже не назовешь. В результате получается, что для иллюстрации работы сценариев JavaScript нужно использовать браузер, а использование браузера подразумевает неплохое (на уровне объектной модели документа) владение основами языка. Чтобы разорвать этот замкнутый круг, мы поступим просто и прагматично. Сначала сведем «взаимодействие» сценария с браузером к минимуму и сконцентрируемся исключительно на особенностях языка программирования JavaScript. После того как вершины JavaScript будут «взяты», мы переключимся на вопросы прикладного использования языка JavaScript для написания сценариев.

Структура книги соответствует подходу, задекларированному выше. Книга состоит из трех частей. В первой части обсуждаются основные синтаксические конструкции языка JavaScript. Во второй части описываются методы объектно-ориентированного программирования в JavaScript. Третья часть посвящена написанию сценариев для работы с веб-документами.

НА ЗАМЕТКУ

В третьей части книги рассматривается *объектная модель документа* — набор средств языка JavaScript, предназначенных для работы с веб-документом.

В первой и второй частях книги браузер будет использоваться как средство вывода информации сценарием. Что касается задач и примеров, рассматриваемых в первых двух частях книги, то они имеют достаточно общий характер (встречаются в том числе и математические задачи).

НА ЗАМЕТКУ

Сказанное не означает, что читателю необходимо иметь специальную математическую подготовку. Во-первых, примеры рассматриваются простые. А во-вторых, в книге приводится полная ин-

формация, необходимая для понимания сути задачи и методов ее решения.

Такой подход представляется оправданным, поскольку позволяет показать гибкость и эффективность языка JavaScript как такового. С методической точки зрения также имеются плюсы: по крайней мере, у читателя вырабатывается четкое представление о том, что такое язык JavaScript, во-первых, и как этот язык может использоваться для написания сценариев в браузерах, во-вторых.

ДЕТАЛИ

В прикладной плоскости проблема совместного использования языка JavaScript и браузера связана с тем, что в веб-документе присутствует как HTML-код, так и непосредственно код сценария (даже если он включен в веб-документ не в явном виде, а через файл сценария). Хотя в принципе HTML-код и JavaScript-код легко различить, все равно остается некоторая эстетическая незавершенность, особенно если веб-документ большой и нетривиальный. Ведь как сценарий, так и HTML-код влияют на вид и функциональность веб-документа. Отделить последствия выполнения сценария от результатов настроек посредством HTML-кода бывает непросто.

Очевидно, что, даже сведя к минимуму взаимодействие с браузером, совсем избежать наличия HTML-кода не удастся. Для первой и второй частей книги мы будем использовать определенный шаблонный веб-документ с минимально необходимым HTML-кодом. Основная же часть документа будет представлена сценарием, выполняемым при загрузке документа в браузер. Таким образом, в первой и второй частях книги нам практически не придется отвлекаться на обсуждение HTML-кодов (разве что в минимальных объемах). Тем не менее, когда это все же придется делать, по ходу изложения будет приводиться справка относительно используемых HTML-инструкций.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Для лучшего восприятия информация относительно HTML-кодов в основной части книги выносится в специальные блоки.

В первой и второй частях книги анализируются в основном сценарии на языке JavaScript и, соответственно, интерес представляет резуль-

тат выполнения сценариев. Результат выполнения сценариев отображается в окне браузера в текстовом виде. В книге результаты выполнения сценариев приводятся в специальных текстовых блоках (как это было сделано ранее, в предыдущем разделе, при обсуждении результата выполнения сценария).

В третьей части книги используется несколько иной подход. Поскольку там обсуждается применение сценариев на практике, то интерес представляет не сам сценарий, а то, как он отражается на всем веб-документе. Соответственно, для объяснения результатов нам понадобится все окно браузера. Другой подход здесь вряд ли уместен, особенно если учесть, что речь может (и будет) идти не только о тексте, но и о графических компонентах (таких, например, как кнопки или поля ввода).

Для лучшего усвоения материала каждая глава содержит в конце краткое *Резюме*, в котором выделяются основные наиболее важные моменты, обсуждаемые в главе.

Тестирование сценариев и программное обеспечение

— Вот по этому поводу первый тост.
— Сейчас запишу.
— Потом запишешь.

из к/ф «Кавказская пленница»

Книгу по программированию (любую) мало прочитать. С ней нужно работать. Последнее предполагает изучение, анализ и проработку программных кодов из книги или иных кодов, близких по духу и смыслу. Как бы там ни было, чтобы научиться программировать, разумно, вооружившись минимальными теоретическими познаниями, приступить к написанию кодов. Для этого необходимы программные средства разработки, или, проще говоря, программное обеспечение.

Обычно для написания программ на том или ином языке устанавливают специальную среду разработки для этого языка. В принципе нечто похожее на среды разработки существует и для языка JavaScript. Однако сценарий, написанный на JavaScript, все равно рано или поздно инкапсулируется в веб-документ. Только при использовании сред разработки проекты получаются достаточно громоздкие. Проще все

сделать самому вручную. Помимо этого, настройка среды разработки может оказаться задачей не самой простой. Поэтому далее мы опишем способ создания и тестирования сценариев, который не подразумевает использование какого-либо специализированного программного обеспечения. Нам понадобится только браузер и текстовый редактор.

НА ЗАМЕТКУ

В принципе подойдет любой браузер, поддерживающий язык JavaScript. Все наиболее популярные браузеры язык JavaScript поддерживают.

Для большей конкретики процесс работы со сценарием проиллюстрируем на примере браузера Mozilla Firefox и текстового редактора Notepad. Браузер и редактор могут быть другими, но принципы работы с ними те же.

ДЕТАЛИ

Выполнение сценариев считается занятием потенциально опасным. Поэтому в настройках браузеров имеются опции, позволяющие заблокировать выполнение сценариев. Для работы со сценариями JavaScript в браузере должен быть установлен режим, разрешающий выполнение сценариев. Более детальную информацию о настройках браузера обычно можно найти в его справочной системе.

Идея состоит в том, чтобы открыть один и тот же документ одновременно в браузере и текстовом редакторе. Далее схема действий такая.

- Вносим изменения в код в окне текстового редактора.
- Сохраняем внесенные изменения в текстовом редакторе.
- Анализируем изменения в окне браузера. Для этого в браузере щелкаем по кнопке **Reload (Обновить текущую страницу)**, выполняя тем самым перезагрузку открытого документа.

На рис. В.11 показано окно браузера Mozilla Firefox с открытым пустым документом, а на фоне окна браузера этот же документ открыт в текстовом редакторе Notepad.

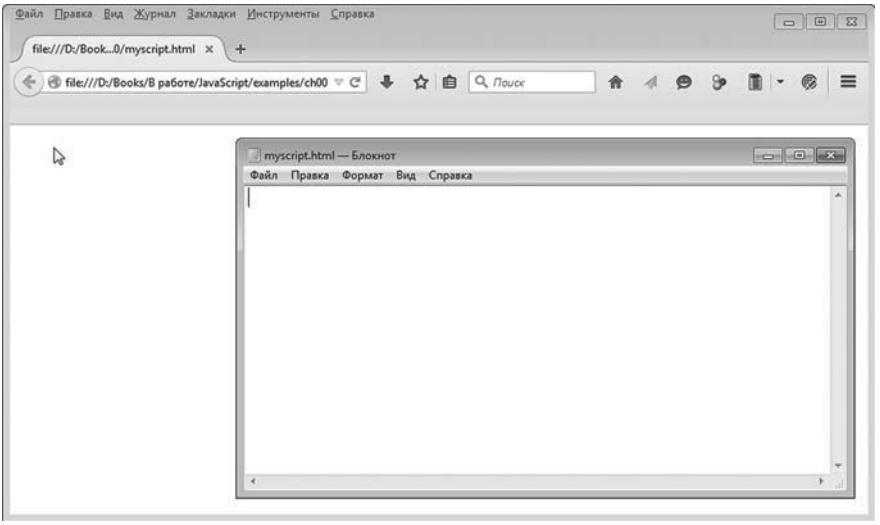

Рис. В.11. На фоне окна браузера Mozilla Firefox открыто окно текстового редактора с пустым веб-документом

ДЕТАЛИ

Предварительно следует создать новый текстовый файл myscript.txt, заменив в нем расширение с .txt на .html. В итоге получаем пустой (не содержит ничего) файл myscript.html. Этот файл открываем браузером, и этот же файл открываем в текстовом редакторе (при этом файл остается открытым и в браузере).

На следующем этапе вводим в текстовый файл код, представленный в листинге В.3.

Листинг В.3. Код для веб-документа

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>Файл myscript.html</title>
  </head>
<body>
  <h3>Тестируем сценарий</h3><hr>
</body>
</html>
```

После того как код введен в окне текстового редактора, изменения следует сохранить (в меню **Файл** текстового редактора выбирается команда **Сохранить**). Процесс сохранения введенного в текстовом редакторе кода показан на рис. В.12.

При этом в браузере изменения в документ в силу еще не вступили. Чтобы увидеть эффект от введения в документ кода, в окне браузера щелкаем по пиктограмме обновления открытого документа, как показано на рис. В.13.

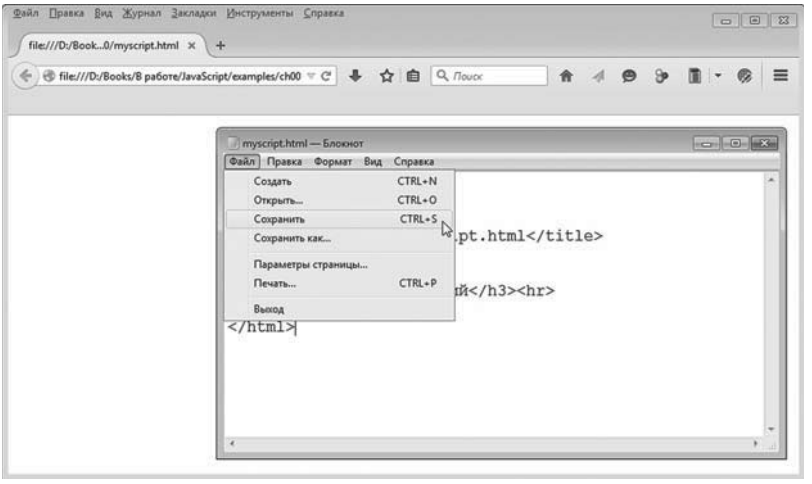

Рис. В.12. Сохранение в текстовом редакторе введенного кода

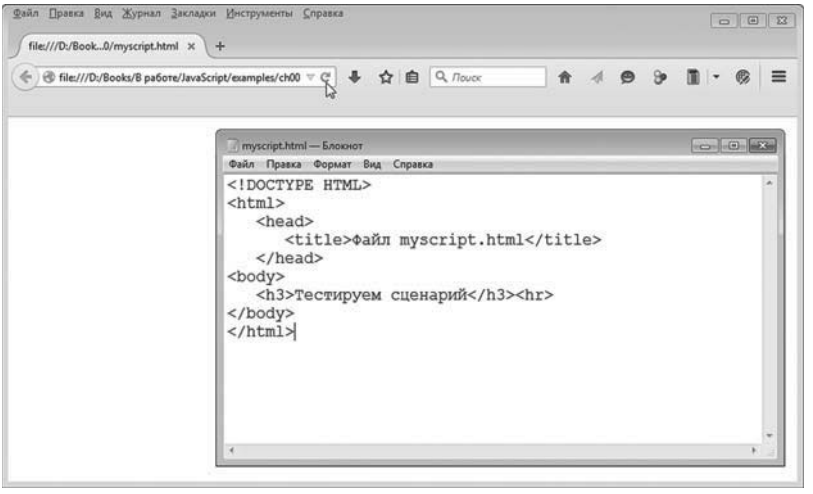

Рис. В.13. Обновление открытого документа после внесения в него кода

НА ЗАМЕТКУ

Для обновления документа в окне можно щелкнуть правой кнопкой мыши на корешке вкладки документа и в раскрывшемся контекстном меню выбрать команду **Обновить вкладку**.

Изменения в браузере вступят в силу. Результат показан на рис. В.14.

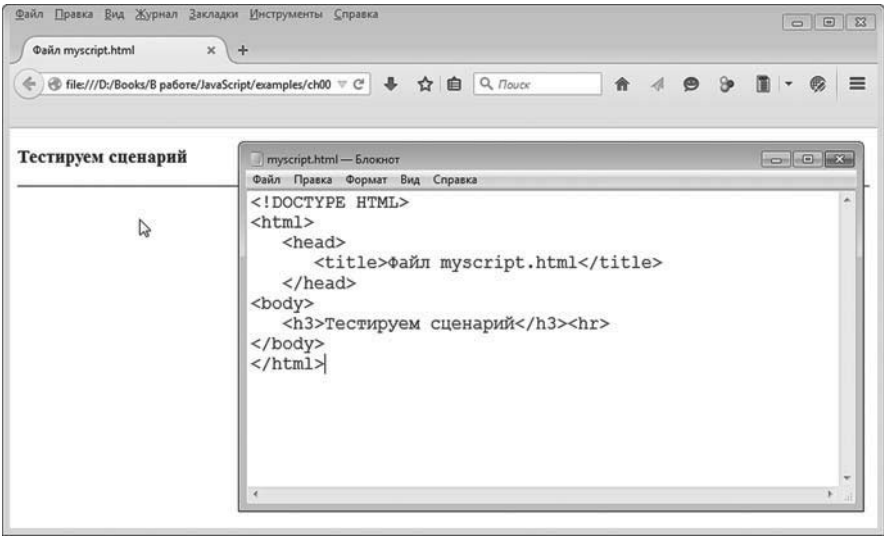

Рис. В.14. Внесенные в документ изменения вступили в силу в окне браузера

Если в последующем нам понадобится отредактировать документ, повторяем описанные выше действия. Например, мы хотим добавить в веб-документ сценарий. Для этого в окне текстового редактора добавляем несколько строк кода. Весь новый, измененный код представлен в листинге В.4.

Листинг В.4. Измененный код для веб-документа

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>Файл myscript.html</title>
  </head>
<body>
```

```
  <h3>Тестируем сценарий</h3><hr>
  <script type="text/javascript">
    document.write("Изучаем JavaScript")
  </script>
</body>
</html>
```

Изменения в коде документа незначительные.

НА ЗАМЕТКУ

По сравнению с листингом В.3 в листинге В.4 появился <script>-блок с инструкцией document.write("Изучаем JavaScript"). Как следствие в области документа выводится текстовое сообщение Изучаем JavaScript.

Вносим изменения в текстовый документ и сохраняем файл в окне текстового редактора (рис. В.15).

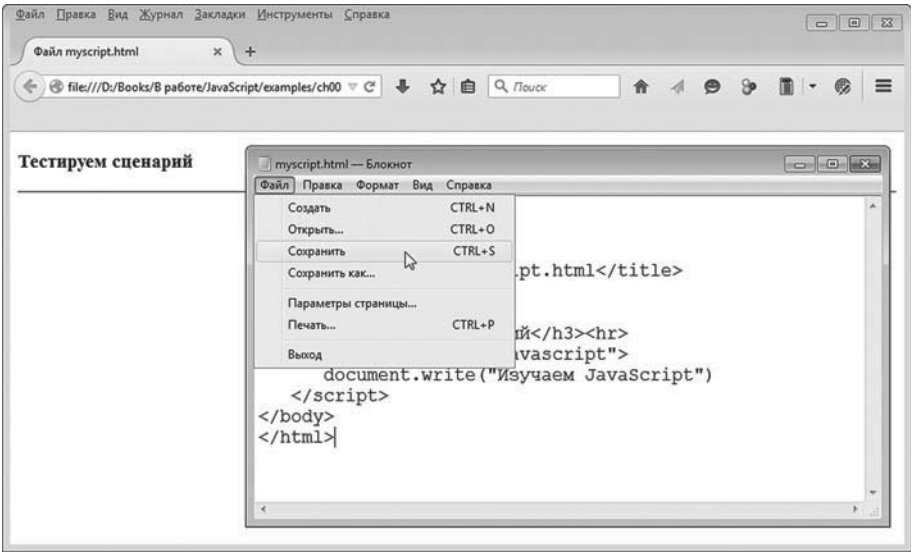

Рис. В.15. После внесения изменений в программный код выполняем сохранение документа

Далее обновляем содержимое окна браузера, как показано на рис. В.16.

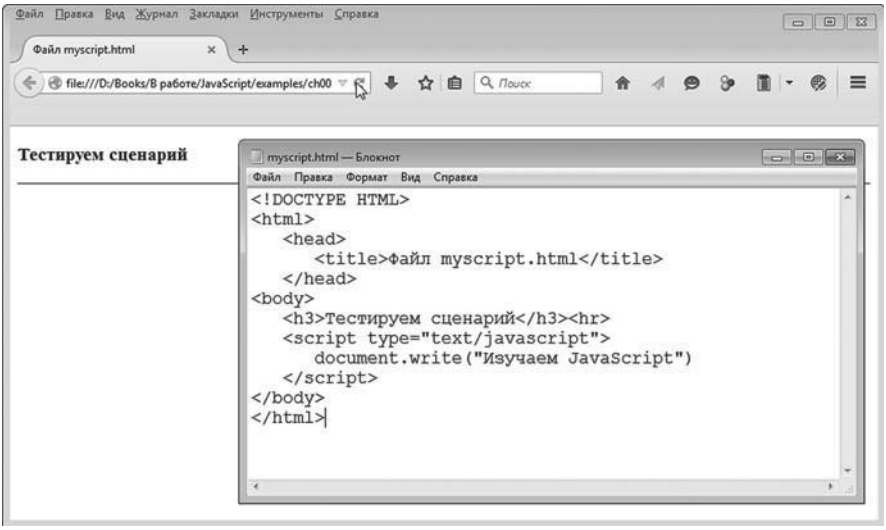

Рис. В.16. Обновление содержимого окна браузера

Результат представлен на рис. В.17.

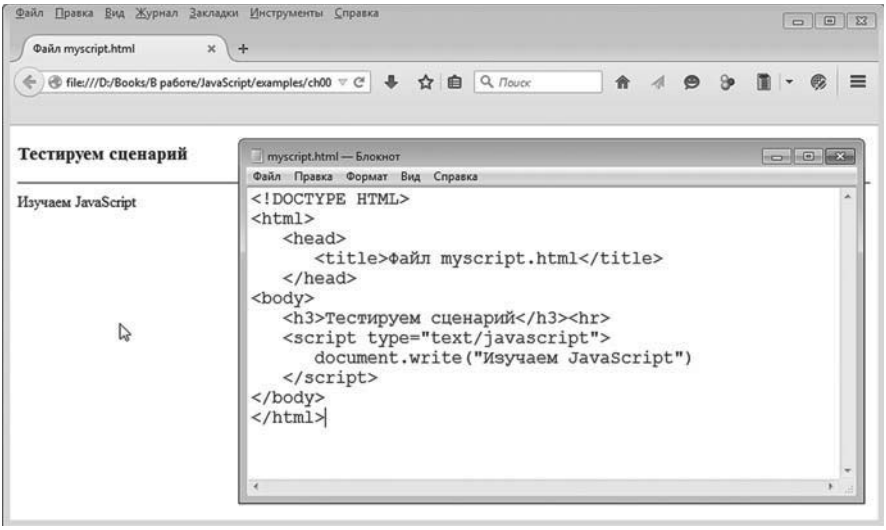

Рис. В.17. Документ в окне браузера после внесения изменений и обновления страницы

В принципе данный подход вполне приемлем для создания и тестирования программных кодов. Вместе с тем у него есть существенный недостаток. Дело в том, что набор и редактирование кода выполня-

ются в текстовом редакторе. Это означает, что ни «подсветки» кода, ни автоматической проверки его на корректность нет. Как нет и автоматического заполнения ввода (когда при вводе команды выпадает список подстановки или появляется подсказка по вводимой инструкции). Ничего особо страшного здесь нет, но и приятного тоже мало.

> **НА ЗАМЕТКУ**
>
> Желающие могут поискать более «изысканный» редактор для набора и работы со сценариями JavaScript и HTML-кодами. Благо недостатка в них нет. Правда, многое зависит от используемой операционной системы, да и программные продукты изменяются достаточно интенсивно. Так что данная почетная миссия всецело перекладывается на плечи читателя — здесь вопрос с редакторами обсуждать не будем.

В некоторых браузерах имеются довольно неплохие встроенные средства разработки сценариев. Скажем, в браузере Mozilla Firefox в меню **Инструменты** есть подменю **Веб-разработка**. Среди команд данного подменю можно видеть (рис. В.18) команду **Простой редактор JavaScript**.

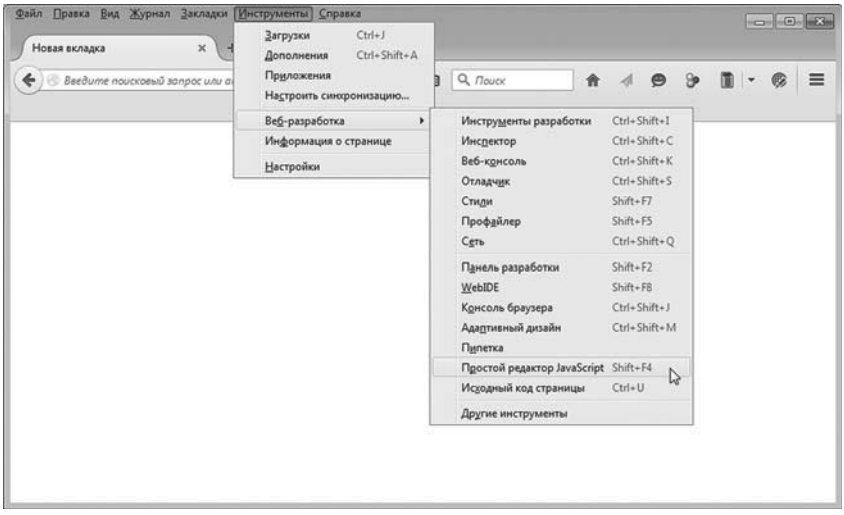

Рис. В.18. Отображение встроенного редактора кодов JavaScript

После выбора указанной команды открывается окно встроенного редактора кодов JavaScript, представленное на рис. В.19.

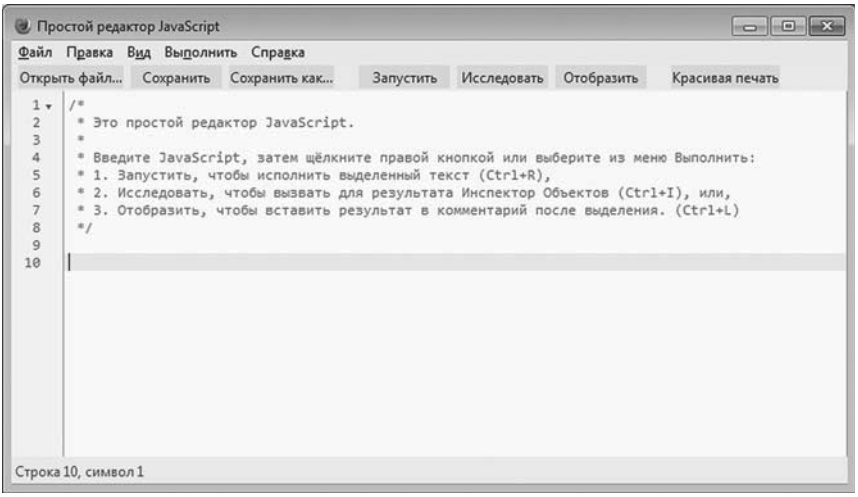

Рис. В.19. Окно встроенного редактора кодов JavaScript

НА ЗАМЕТКУ

Мы уже знаем, что комментарий в JavaScript начинается с двойной косой черты //. Это однострочный комментарий. Еще есть многострочные комментарии. Многострочный комментарий начинается с символов /* и заканчивается символами */. Все, что находится между символами /* и */, является комментарием и при выполнении кода игнорируется.

При отображении окна редактора JavaScript в нем появляется текст, который, как несложно заметить, начинается символами /* в первой строке и завершается символами */ в последней строке (см. рис. В.19). Таким образом, текст, отображаемый в окне редактора кодов, является комментарием JavaScript.

В окно редактора кодов можно ввести программный код сценария и запустить его на выполнение, щелкнув по пиктограмме **Запустить**, как показано на рис. В.20.

В данном случае код сценария состоит всего из одной команды `alert("Вас приветствует Firefox!")`. При выполнении команды отображается диалоговое окно с текстовым сообщением Вас приветствует Firefox! и кнопкой **ОК**. Окно показано на рис. В.21.

Учитывая, что в JavaScript существует возможность отобразить диалоговое окно с полем ввода, в сценарии можно организовать процесс ввода и вывода информации через диалоговые окна. Одна-

ко в этом случае браузер все равно должен быть запущен, да и диалоговые окна, откровенно говоря, не очень удобные. Поэтому мы предпочтем данному подходу метод вставки сценария в HTML-код веб-документа.

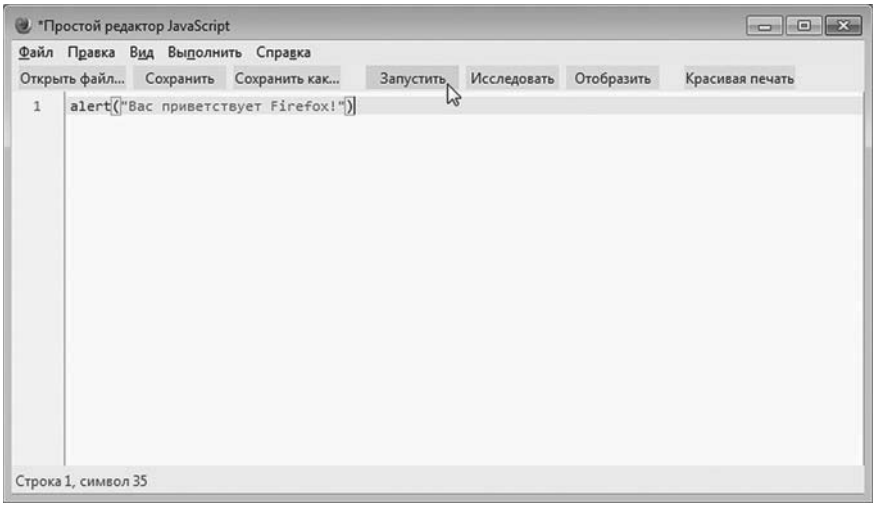

Рис. В.20. Программный код JavaScript перед запуском в окне редактора кодов

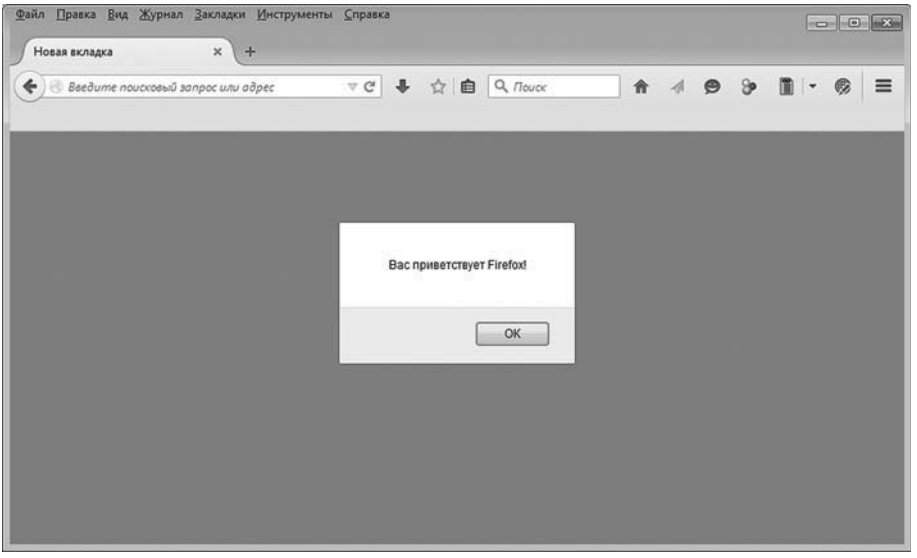

Рис. В.21. В окне браузера появляется диалоговое окно с приветствием

Сценарии можно сохранять в отдельных файлах с расширением .js.

НА ЗАМЕТКУ

Далее подразумевается, что используется операционная система Windows.

Например, создаем текстовый файл с названием ShowWindow.txt, вводим туда команду WScript.echo("Изучаем JavaScript!"), сохраняем файл, закрываем его и меняем расширение на .js (получается файл ShowWindow.js). Если попытаться открыть этот файл, появится диалоговое окно, как на рис. В.22.

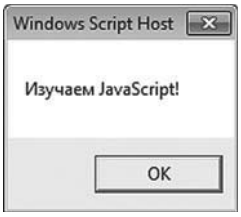

Рис. В.22. Диалоговое окно отображается при открытии файла со сценарием

В данном случае окно браузера не отображается — только диалоговое окно.

ДЕТАЛИ

Если быть откровенным, то единственная команда сценария WScript.echo("Изучаем JavaScript!") не совсем относится к JavaScript. Здесь используется библиотека MSDN, в которой описан объект WScript, а у этого объекта есть метод echo(), позволяющий отображать окно с сообщением. Весь код выполняется под управлением специального компонента Windows, который называется Windows Script Host и предназначен для выполнения сценариев на таких языках, как, например, JavaScript или VBScript. Сценарий в файле с расширением .js выполняется этим компонентом. То есть в данном случае речь идет о «специфическом» сценарии, «понятном» только для операционной системы Windows. И хотя теоретически мы можем добавить в сценарий совершенно корректные с точки зрения JavaScript команды, а объект WScript использовать только для вывода информации, это все равно будет Windows-ориентированное программирование. В принципе допустимо, но не универсально.

Вывод из всего сказанного следует простой: при работе с JavaScript существует достаточно широкое окно возможностей. Читатель при

желании легко (хочется в это верить) найдет для себя приемлемый путь для реализации сценариев на языке JavaScript. Мы же пойдем, может быть, не самым эффективным, зато самым надежным путем: будем вставлять блоки со сценариями в HTML-коды и тестировать результат в окне браузера.

НА ЗАМЕТКУ

По умолчанию (если явно не указан тип браузера) имеется в виду, что коды сценариев тестировались в браузере Mozilla Firefox.

Обратная связь

— Высокие, высокие отношения!
— Нормальные для духовных людей.

из к/ф «Покровские ворота»

Свои замечания, предложения и просто мысли по поводу книги читатели могут сообщить автору по электронной почте vasilev@univ.kiev.ua или alex@vasilev.kiev.ua. На сайте www.vasilev.kiev.ua обычно представлена краткая информация о книге. Там же (нередко по просьбе читателей) размещаются некоторые дополнительные сведения — в основном речь идет о программных кодах примеров. В любом случае важно и интересно знать мнение читателей о книге: что понравилось и, самое главное, что не удалось. Критический взгляд со стороны полезен в любом деле, а в таком, как написание книг, — вдвойне. Правда, в силу очевидных и объективных причин всем приславшим письма ответить не получится, но письма будут прочитаны и приняты к сведению.

Об авторе

Вам что, уже и Лермонтов не угодил?! Или у вас другие любимые авторы?

из к/ф «Покровские ворота»

Автор книги, *Васильев Алексей Николаевич*, доктор физико-математических наук по специальности «Теоретическая физика» и «Теплофизика и молекулярная физика», профессор кафедры теоретической физики физического факультета Киевского национального университета имени Тараса Шевченко. Сфера научных интересов:

- компьютерное моделирование;
- синергетика;
- биофизика;
- теория жидких кристаллов;
- математическая экономика;
- математическая лингвистика.

Имеет многолетний опыт чтения лекционных курсов по программированию и математическому моделированию. Автор книг по программированию на языках C++, C#, Java и Python, книг по математическим пакетам Maple, Mathematica, Mathcad и Matlab, книг по использованию в прикладных математических вычислениях офисного приложения Microsoft Excel.

Благодарности

> Не надо громких слов. Они потрясают воздух, но не собеседника.
>
> *из к/ф «Формула любви»*

Написание книг дает возможность не только научить чему-то полезному читателей, но еще и выразить слова признательности хорошим людям за их явную и неявную поддержку.

Пользуясь случаем, хочу поблагодарить своих родителей, дочку Настю, сына Богдана и жену Илону за их терпение, понимание и любовь, которые значат для меня очень много и дают силы двигаться вперед. Спасибо моим читателям и студентам за их требовательность, готовность учиться и, конечно, за внимание к книгам. До того как попасть в руки к читателю, книга проходит длинный путь. С ней работают редакторы, переводчики, верстальщики и многие другие замечательные профессионалы своего дела. Далеко не со всеми из них я знаком лично, но всем им я искренне благодарен.

Часть I

ОСНОВЫ JAVASCRIPT

Глава 1
ЗНАКОМСТВО С JAVASCRIPT

> Так где мы можем обсудить аспекты, так сказать, нашего общего дела?
>
> *из к/ф «Клуб самоубийц, или Приключения титулованной особы»*

Итак, мы начинаем знакомство с языком JavaScript. При этом нам придется постоянно иметь дело со сценариями. В основном наша стратегия будет заключаться в том, чтобы включать блок с программным кодом, написанным на языке JavaScript, в документ с гипертекстовой разметкой. Поскольку пока что нас язык JavaScript интересует сам по себе, вне контекста веб-документа, мы будем использовать определенный очень простой шаблон документа с HTML-кодом, в который «инкапсулирована» инструкция по включению в документ сценария на языке JavaScript.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Для вставки в HTML-документ блока сценария используются дескрипторы <script> и </script>. В открывающем дескрипторе <script> описывается атрибут type со значением "text/javascript". Поэтому начинаться блок сценария будет инструкцией <script type="text/javascript">, а заканчиваться — инструкцией </script>.

НА ЗАМЕТКУ

Прежде чем перейти непосредственно к рассмотрению основ языка JavaScript, некоторое внимание придется уделить способам «инкапсуляции» сценарного кода в документ.

Сценарий должен выполнять определенные действия (иначе какой в нем смысл?). Результат этих действий должен как-то проявляться. Самый простой способ «проявления» сценария в документе — вывод информации. Информацию будем выводить в рабочую область

документа. Как иллюстрацию для начала рассмотрим очень простой пример, в котором средствами языка JavaScript в рабочем документе отображается текст.

Отображение текста в рабочем документе

Вы получите то, что желали, согласно намеченным контурам.

из к/ф «Формула любви»

В самом первом примере мы решим очень простую задачу: сценарий при выполнении отобразит в рабочем документе текст. В принципе, как решается такая задача, иллюстрировалось во *вступлении*. Здесь мы немножко расширим наше представление о возможностях сценариев в плане отображения текста в документе. Обратимся к программному коду, представленному в листинге 1.1.

Листинг 1.1. Отображение сценарием текста в документе

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 1.1</title>
</head>
<body><h3>Листинг 1.1</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript">
  document.write("Приступаем к изучению <b>JavaScript</b>")
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Речь идет об HTML-коде, содержащем, кроме прочего, сценарий. На рис. 1.1 показано окно текстового редактора с данным кодом.

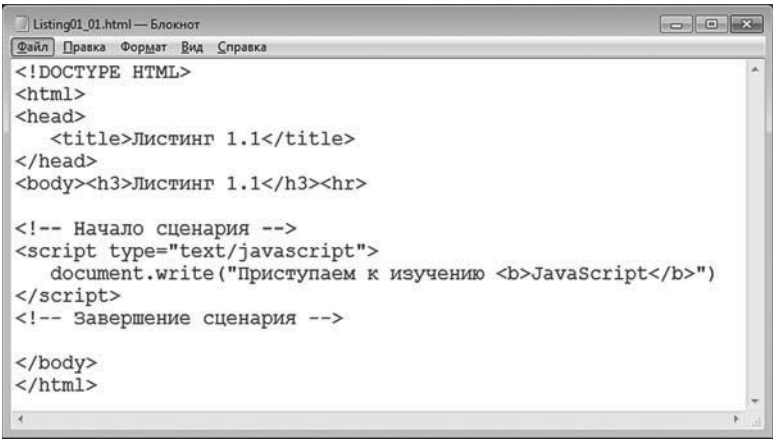

Рис. 1.1. Окно текстового редактора с кодом документа

Если этот же документ открыть в браузере, получим результат, как на рис. 1.2.

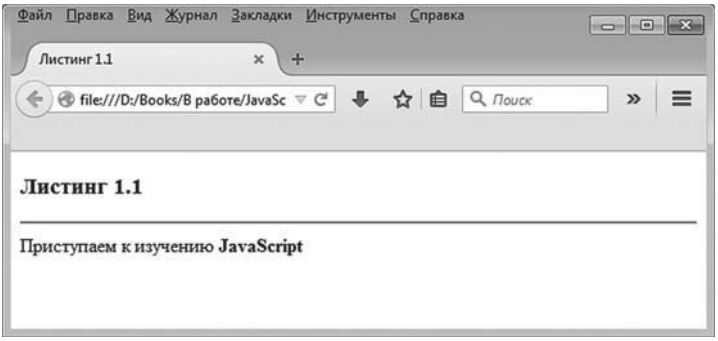

Рис. 1.2. Результат отображения веб-документа в окне браузера

Результат выполнения сценария — текст под горизонтальной линией в области документа. Все, что выше горизонтальной линии, включая саму линию, определяется HTML-кодом документа.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Код гипертекстовой разметки кратко обсуждался во *вступлении*. Здесь на всякий случай напомним назначение основных блоков HTML-кода.

Инструкция <!DOCTYPE HTML> является стандартным началом HTML-кода документа. Сам программный код заключается между дескрипто-

рами <html> и </html>. В <head>-блоке описываются основные свойства документа. В частности, в блоке между дескрипторами <title> и </title> указывается рабочее название документа (отображается в корешке вкладки окна документа в браузере). Основной код документа размещается в <body>-блоке. В данном конкретном примере этот блок содержит заголовок 3-го уровня (специально выделенный текст, отображается в рабочей области документа). Заголовок выделяется дескрипторами <h3> и </h3>. Инструкция <hr> используется для отображения горизонтальной линии в области рабочего документа. Все, что находится между инструкциями <!-- и -->, является комментарием. Сценарий, как отмечалось ранее, размещается в <script>-блоке.

Фактически код сценария состоит всего из одной команды:

```
document.write("Приступаем к изучению <b>JavaScript</b>")
```

Здесь мы имеем дело с вызовом метода write() из объекта document. Объект document — это объект рабочего документа (документ, в котором размещен код сценария). Метод write(), который вызывается из объекта документа document, отображает в данном документе текст, переданный аргументом методу. Причем отображаемый текст может содержать не только собственно текст, но и HTML-кодировку, как это имеет место в рассмотренном примере. В частности, в текстовом выражении "Приступаем к изучению JavaScript", которое передается аргументом методу write(), слово JavaScript выделено дескрипторами и . В кодировке HTML это означает выделение жирным шрифтом соответствующего текстового фрагмента при отображении его в окне браузера. Так, собственно, и происходит (см. рис. 1.2).

Таким образом, при отображении текста в рабочем документе с помощью метода write() речь идет не просто об отображении текста в окне, а об отображении HTML-кода. Такой код может содержать всевозможные HTML-дескрипторы (то есть дело не ограничивается дескрипторами и или подобными им).

НА ЗАМЕТКУ

Здесь открываются достаточно широкие возможности в плане динамического наполнения веб-документа. Другими словами, с помощью сценария можно управлять наполнением веб-документа в динамическом режиме. Технологии, подобные этой, рассматриваются в третьей части книги.

Способы реализации сценария

> Да, это от души. Замечательно. Достойно восхищения. Ложки у меня пациенты много раз глотали, не скрою. Но вот чтобы так, за обедом на десерт, и острый предмет — замечательно. За это вам наша искренняя сердечная благодарность.
>
> *из к/ф «Формула любви»*

Выше мы помещали код сценария непосредственно в HTML-код вебдокумента. Для не очень больших примеров такой подход вполне удобен. Фактически речь идет об использовании определенного шаблонного кода. Если отталкиваться от предыдущего примера, то шаблон такой, как показано ниже:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 1.1</title>
</head>
<body><h3>Листинг 1.1</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript">
...// Код сценария
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Программный код сценария размещается в том месте, где размещен комментарий // Код сценария (для удобства восприятия соответствующее место в коде выделено жирным шрифтом).

Существует и другой способ «подключения» сценария к документу. В этом случае сценарий записывается в отдельный файл, а в HTML-коде документа добавляется ссылка на этот файл. Ссылка на файл со сценарием указывается значением атрибута src в дескрипторе <script>.

Например, если файл со сценарием называется Listing01_02.js и находится в той же папке, что и файл с HTML-кодом, то HTML-код может быть таким, как показано в листинге 1.2. Жирным шрифтом выделена инструкция с указанием загружаемого в документ файла со сценарием.

Листинг 1.2. Загрузка сценария из внешнего файла

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 1.2</title>
</head>
<body><h3>Листинг 1.2</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing01_02.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Здесь, как и в предыдущем примере, мы используем дескриптор <script> для включения в веб-документ блока сценария. Вместе с тем сам <script>-блок пустой, а в открывающем дескрипторе <script> появилась инструкция src="Listing01_02.js", которой для атрибута src задается значение "Listing01_02.js" — имя файла со сценарием.

НА ЗАМЕТКУ

Если файл со сценарием находится в папке, отличной от той, где находится файл веб-документа, значением атрибута src указывается полный путь к файлу сценария.

В файл Listing01_02.js со сценарием помещаем такой код:

```
document.write("Сценарий загружается из файла")
```

На рис. 1.3 показан веб-документ с кодом из листинга 1.2, открытый в текстовом редакторе.

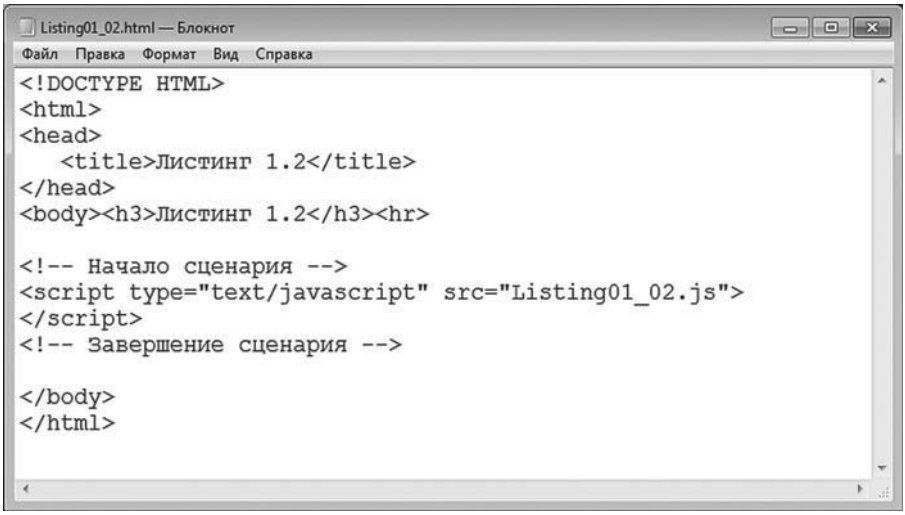

Рис. 1.3. Веб-документ открыт в текстовом редакторе

На рис. 1.4 показано окно текстового редактора с открытым в нем файлом со сценарием.

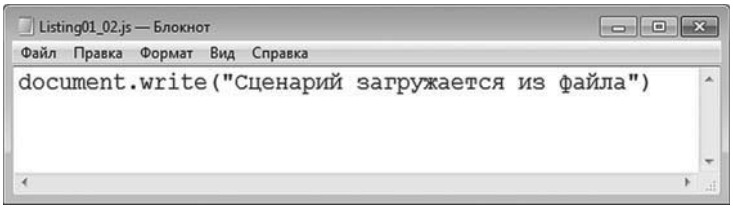

Рис. 1.4. Документ с кодом сценария открыт в текстовом редакторе

Наконец, веб-документ с кодом из листинга 1.2, который открыт в браузере, показан на рис. 1.5.

С точки зрения конечного результата загрузка сценария из внешнего файла не отличается от ситуации, когда код сценария включался непосредственно в код веб-документа.

НА ЗАМЕТКУ

Различия, конечно, есть: например, в части скорости загрузки сценария (да и надежности — файл со сценарием при неправильной ссылке на файл сценария не будет найден вовсе). Но нас такие подробности пока не интересуют.

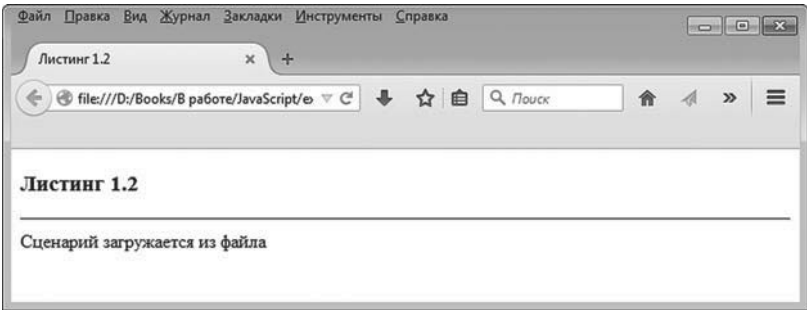

Рис. 1.5. Веб-документ открыт в браузере

На будущее, в зависимости от обстоятельств, мы будем или включать код сценария непосредственно в HTML-код документа, или записывать сценарный код в отдельный файл, а в код веб-документа добавлять инструкцию загрузки внешнего файла со сценарием.

Знакомство с переменными

> Видала я такую чепуху, по сравнению с которой эта чепуха — толковый словарь!
>
> *Л. Кэрролл «Алиса в Стране чудес»*

Вывод информации в окно рабочего документа — это хорошо. Но сценарий, состоящий из одной-единственной команды, выглядит уж слишком скромно.

Мы усложняем ситуацию и переходим на новый уровень в освоении премудростей JavaScript. Пришло время познакомиться с *переменными*.

Вообще переменная представляет собой именованную область памяти, к которой можно обращаться через имя для считывания значения и записи значения. Таким образом, у переменной есть имя (которое задается программистом). Еще у переменной есть *тип*.

НА ЗАМЕТКУ

Размер памяти, выделяемой для переменной, в принципе зависит от ее типа. Во многих языках программирования (но не в JavaScript) тип переменной указывается при ее объявлении и впоследствии не может быть изменен.

Значения, которыми оперируют в программном коде JavaScript, относятся к одному из следующих типов:

- текстовая строка;
- числовое значение;
- логическое значение (*истина* или *ложь*);
- объект;
- функция.

Объекты и функции мы пока трогать не будем. Здесь разговор будет отдельный. Пока что в зоне наших интересов текст и числа (логические значения рассмотрим при обсуждении операторов и управляющих инструкций).

ДЕТАЛИ

Если переменная принимает значение логического типа, то это означает, что она принимает одно из двух возможных значений: true (истина) или false (ложь). Обычно логические значения используются при проверке условий. Для работы с логическими значениями предназначена группа операторов, которые называются логическими.

А теперь очень важный момент: в языке JavaScript тип переменной *не фиксируется*. Буквально сказанное означает, что одна и та же переменная на разных этапах выполнения программы может принимать не просто разные значения, а значения *разных типов*. Другими словами, значением переменной сначала, например, может быть число, затем текст, затем что-то еще (в том числе объект или даже функцию — но об этом позже).

НА ЗАМЕТКУ

Для тех, кто знаком с такими языками программирования, как C++, C# или Java, означенный «демократизм» языка JavaScript в плане типизации переменных может вызвать некоторый шок. Тем не менее имеем то, что имеем.

Обычно переменные объявляются. Объявление переменной — некая декларация, цель которой в том, чтобы сообщить о намерении

использовать переменную в программном коде. Хотя такое понятие, как *тип данных,* в JavaScript существует, при объявлении переменной тип не указывается. В этом просто нет смысла, поскольку, как отмечалось выше, тип значения переменной может меняться. Более того, в JavaScript переменные можно вообще не объявлять, а сразу их использовать. Мы тем не менее будем придерживаться стиля, при котором используемые в сценарии переменные объявляются.

Сценарий с одной переменной

Как объявить переменную в сценарии? Достаточно просто. Используем ключевое слово var, после которого указываем название переменной. Если переменных несколько, их названия указываются через запятую. Как иллюстрация в листинге 1.3 приведен программный код сценария (сценарий вынесен в отдельный файл).

Листинг 1.3. Использование переменной (файл Listing01_03.js)

```
var txt
txt="Используем переменную"
document.write(txt)
```

Для тестирования кода создаем веб-документ с таким кодом:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 1.3</title>
</head>
<body><h3>Листинг 1.3</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing01_03.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

В результате выполнения сценария выводится следующее сообщение.

Результат выполнения сценария (из листинга 1.3)

```
Используем переменную
```

Сценарий содержит три команды. Командой var txt объявляется переменная с названием txt. Командой txt="Используем переменную" переменной txt присваивается текстовое значение "Используем переменную". Наконец, с помощью команды document.write(txt) значение переменной txt отображается в рабочей области документа.

ДЕТАЛИ

При желании для большей наглядности в конце команд можно ставить точку с запятой. Если каждая команда находится в новой строке, этого можно не делать. Если несколько команд находятся в одной строке, они разделяются точкой с запятой.

Текстовые литералы заключаются в двойные или одинарные кавычки. Поэтому вместо выражения txt="Используем переменную" можно было использовать команду txt='Используем переменную'.

Объявление переменной можно совмещать с присваиванием переменной значения. Например, вместо команд var txt и txt="Используем переменную" мы могли бы использовать одну команду var txt="Используем переменную".

На рис. 1.6 показан веб-документ с результатом выполнения сценария.

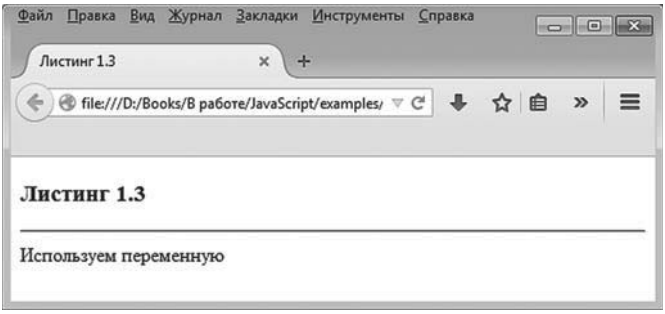

Рис. 1.6. Результат выполнения сценария, в котором использована переменная

Понятно, что переменных может быть больше, чем одна.

Сценарий с двумя переменными

Следующий пример иллюстрирует использование двух переменных. Код сценария приведен в листинге 1.4.

Листинг 1.4. Использование двух переменных (файл Listing01_04.js)

```
var txt,num
txt="Значение числа: "
num=123
document.write(txt+num)
```

Код сценария тестируем с помощью такого веб-документа:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 1.4</title>
</head>
<body><h3>Листинг 1.4</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing01_04.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Результат выполнения сценария приведен ниже.

Результат выполнения сценария (из листинга 1.4)

```
Значение числа: 123
```

В окне браузера все выглядит так, как показано на рис. 1.7. Как и в предыдущем случае, здесь все достаточно просто. Командой `var txt,num` объявляются две переменные: одна называется `txt`, другая называется `num`. Командами `txt="Значение числа: "` и `num=123` переменным присваиваются

значения (текстовое переменной txt и числовое переменной num). После присваивания переменным значения командой document.write(txt+num) в рабочей области документа отображается текст. Здесь аргументом методу write() передано выражение txt+num, которым формально вычисляется сумма текстового значения и числового значения. Подобные операции обрабатываются так: число автоматически переводится в текстовый формат, и выполняется объединение (конкатенация) текстовых строк. Скажем, если значение переменной txt равно "Значение числа: ", а значение переменной num равно 123, то результатом выражения txt+num будет текст "Значение переменной: 123".

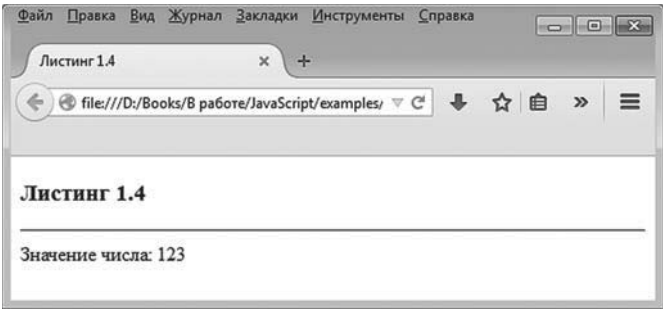

Рис. 1.7. Результат выполнения сценария с двумя переменными

Присваивание переменной значений разных типов

Практически такой же сценарий реализуем, с использованием не двух, а всего одной переменной. Сценарий приведен в листинге 1.5.

Листинг 1.5. Использование двух переменных (файл Listing01_05.js)

```
var x
x="Значение числа: "
document.write(x)
x=123
document.write(x)
```

В сценарии объявляется переменная x, которая сначала принимает текстовое значение, и это значение командой document.write(x) выводится в рабочее окно. Затем переменной x присваивается числовое значение, и снова в игру вступает команда document.write(x), благодаря которой текущее числовое значение переменной x также отображается в рабо-

чем окне (в той же строке, что и предыдущее сообщение). Чтобы проверить корректность работы данного сценария, файл со сценарием загружаем в веб-документ с помощью такого кода:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 1.5</title>
</head>
<body><h3>Листинг 1.5</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing01_05.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Непосредственно результат выполнения сценария приведен ниже.

Результат выполнения сценария (из листинга 1.5)

```
Значение числа: 123
```

Соответствующий веб-документ в окне браузера выглядит так, как показано на рис. 1.8.

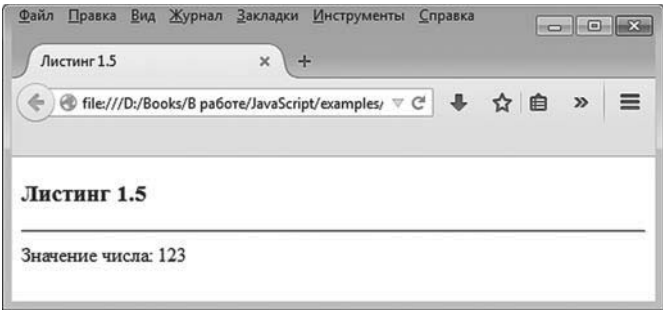

Рис. 1.8. Результат выполнения сценария с одной переменной, принимающей значения разных типов

В рассмотренном примере наиболее важный и показательный момент связан с возможностью присваивать переменной значения разных типов.

Вычисление выражений

Язык JavaScript очень гибкий в плане синтаксиса и структуры программного кода. В частности, в JavaScript есть очень полезная и эффективная функция eval().

Если аргументом функции передать текст, то результатом функции возвращается значение, которое получается при вычислении выражения, «спрятанного» в тексте. Например, выражение "3+(5*2+6)/4" является текстом. Но в этом тексте записано арифметическое выражение 3+(5*2+6)/4, которое имеет смысл: очевидно, речь идет о выражении $3+\frac{5\cdot 2+6}{4}$, которое равно 7.

Если воспользоваться командой eval("3+(5*2+6)/4"), то результат такой команды — значение выражения 3+(5*2+6)/4.

В листинге 1.6 приведен код сценария, в котором с помощью функции eval() вычисляется значение выражения, «упакованного» в текстовую строку.

Листинг 1.6. Вычисление выражения (файл Listing01_06.js)

```
var x="3 + (5*2 + 6) / 4"
document.write(x+" = ")
document.write(eval(x))
```

Здесь мы объявляем переменную x, а значением переменной присваивается текст "3 + (5*2 + 6) / 4" (для удобства восприятия между арифметическими операторами добавлены пробелы).

Командой document.write(x+" = ") данный текст (с добавленным знаком равенства) отображается в рабочем окне. А вот при выполнении команды document.write(eval(x)) отображается значение 7 (результат вычисления выражения в тексте).

Файл со сценарием загружаем в веб-документ, для чего используем следующий код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 1.6</title>
</head>
<body><h3>Листинг 1.6</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing01_06.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

В итоге результат выполнения сценария такой.

Результат выполнения сценария (из листинга 1.6)

```
3 + (5*2 + 6) / 4 = 7
```

На рис. 1.9 показан веб-документ, открытый в браузере.

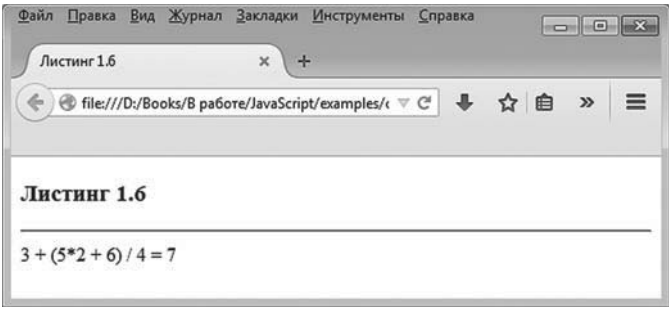

Рис. 1.9. Результат выполнения сценария с вычислением значения выражения

Понятно, что здесь проиллюстрирована довольно простая ситуация. Вместе с тем функция eval() находит самое широкое применение на практике. Особенно она полезна при реализации взаимодействия сценария с веб-документом.

Основные операторы

> Живьём брать демонов!.. Живьём брать самозванцев!
>
> *из к/ф «Иван Васильевич меняет профессию»*

Наряду с переменными в JavaScript используется достаточно много операторов. Обычно их разделяют на четыре группы:

- арифметические операторы;
- операторы сравнения;
- логические операторы;
- побитовые операторы.

Далее рассмотрим операторы каждой группы отдельно и еще кое-что.

Арифметические операторы

К *арифметическим* относятся операторы, предназначенные для выполнения арифметических действий. Арифметические операторы языка JavaScript описаны в табл. 1.1. Операнды у арифметических операторов, как правило, числовые (хотя могут быть и исключения — они обсуждаются отдельно). Все операторы, за исключением операторов *инкремента* и *декремента*, бинарные — у них по два операнда. Операторы инкремента и декремента унарные. Такие операторы используются с одним операндом.

> **НА ЗАМЕТКУ**
>
> У операторов инкремента и декремента есть постфиксная и префиксная формы. Результат вычисления выражений с такими операторами может зависеть от формы (префиксная или постфиксная) оператора.

В принципе многие из арифметических операторов достаточно точно соответствуют своим математическим аналогам, поэтому их назначение интуитивно понятно. Тем не менее имеются некоторые моменты, требующие пояснения.

Результат выражения `A%B` с оператором `%` вычисления остатка от деления рассчитывается как остаток от целочисленного деления зна-

чения операнда A на значение операнда B. Например, результатом выражения 13%5 будет 3. Объяснение такое: при делении 13 нацело на 5 получаем 2. Остаток от такого деления вычисляется как $13 - 5 \cdot 2 = 3$. Если операнды нецелые числа, принцип вычисления результата такой же. Скажем, результатом выражения 10.5%3.3 является значение 0.6, поскольку при целочисленном делении 10,5 на 3,3 получаем 3, а остаток от деления $10{,}5 - 3{,}3 \cdot 3 = 10{,}5 - 9{,}9 = 0{,}6$.

Таблица 1.1. Арифметические операторы JavaScript

Оператор	Описание
+	Оператор сложения. Результатом выражения A+B является сумма значений числовых операндов A и B
-	Оператор вычитания. Результатом выражения A-B является разность значений числовых операндов A и B
*	Оператор умножения. Результатом выражения A*B является произведение значений числовых операндов A и B
/	Оператор деления. Результатом выражения A*B является отношение значений числовых операндов A и B
%	Остаток от целочисленного деления. Результатом выражения A%B является остаток от деления нацело значения операнда A на значение операнда B. Операнды могут быть целыми или нецелыми числами
++	Оператор инкремента. В результате вычисления выражения A++ (постфиксная форма) или ++A (префиксная форма) операнд A увеличивает свое значение на 1. Таким образом, инструкция A++, равно как и инструкция ++A, эквивалентна команде A=A+1
--	Оператор декремента. При вычислении выражений --A (префиксная форма) и A-- (постфиксная форма) значение операнда A уменьшается на 1. Таким образом, каждая из инструкций --A или A-- эквивалентна команде A=A-1

НА ЗАМЕТКУ

Из-за ошибок округления при выполнении операций с числами с плавающей точкой реальный результат может несколько отличаться от «точного» значения.

Как отмечалось выше, у операторов инкремента ++ и декремента -- есть префиксная и постфиксные формы. В префиксной форме оператор указывается перед операндом (например, ++A или --A), а в постфиксной форме оператор указывается после операнда (например, A++ или A--). По отношению к значению операнда разницы в префиксной и постфиксной формах нет. Так, что в результате выполнения инс-

трукции A++, что в результате выполнения инструкции ++A значение переменной A будет увеличено на 1. Но если инструкция с оператором инкремента или декремента сама является частью более сложного выражения, то имеет значение, в какой форме (префиксной или постфиксной) использован оператор. Общее правило состоит в следующем. Если в выражении использовано подвыражение с оператором инкремента/декремента в префиксной форме, то сначала выполняется операция инкремента/декремента, и только после этого вычисляется значение выражения. Если же в выражении есть подвыражение с оператором инкремента/декремента в постфиксной форме, то сначала вычисляется значение выражения, а затем выполняется операция инкремента/декремента. Проиллюстрируем сказанное на простом примере.

Рассмотрим следующий программный код:

```
var x,y
x=10
y=x++
```

Код достаточно простой: объявляются две переменные x и y, и каждой из них присваивается значение. Вопрос такой: каковы будут значения переменных x и y после выполнения кода? Ответ состоит в том, что переменная x будет иметь значение 11, а переменная y будет иметь значение 10. Теперь постараемся разобраться, почему результат именно такой. Начнем с команды x=10, которой переменной x присваивается значение 10. Далее выполняется команда y=x++. Здесь единственная интрига связана со способом вычисления выражения. А именно, возникает вопрос, как нам следует поступить: сначала увеличить на единицу значение переменной x и уже это значение присвоить переменной y или следует сначала присвоить переменной y текущее значение 10 переменной x, а затем увеличить значение x на единицу? Поскольку оператор инкремента использован в постфиксной форме, то сначала выполняется присваивание (переменная y получает значение 10), а затем значение переменной x увеличивается на единицу (становится равным 11).

Достаточно немного изменить код, и мы получим иной результат:

```
var x,y
x=10
y=++x
```

В этом случае обе переменные x и y в конечном счете получат значение 11. Причина в том, что при выполнении команды y=++x, поскольку оператор инкремента использован в префиксной форме, сначала значение переменной x становится равным 11, а затем данное значение присваивается переменной y.

На ситуацию можно посмотреть и по-другому (и это будет даже лучше). Например, рассмотрим такой код:

```
var x,y
x=10
y=x++ + x++
```

Вопрос традиционный: какими будут значения переменных x и y? Чтобы дать ответ, будем исходить из того, что в постфиксной форме оператор инкремента увеличивает значение операнда, но возвращает старое значение. Выражение y=x++ + x++ в правой части содержит сумму двух слагаемых: x++ и x++. Значения слагаемых вычисляются справа налево. При вычислении первого слагаемого x++ значение переменной x увеличивается на 1 (значение переменной x становится равным 11), но результатом выражения x++ возвращается старое значение 10. При вычислении второго слагаемого x++ значение переменной x еще раз увеличивается на 1 (теперь значение переменной равно 12), но результатом выражения x++ возвращается старое значение 11. Таким образом, значение переменной y представляет собой сумму чисел 10 и 11, что дает 21. А значение переменной x равно 12.

Если исходить из того, что оператор инкремента в префиксной форме увеличивает на единицу значение операнда и возвращает результатом новое значение, легко проанализировать следующий программный код:

```
var x,y
x=10
y=++x + ++x
```

Теперь значение переменной x будет равно 12, а значение переменной y равняется 23. Почему? Потому что при вычислении первого слагаемого ++x получаем значение 11, переменная x также имеет значение 11. При вычислении второго слагаемого ++x получаем 12. Значение переменной x тоже равно 12. В итоге x равно 12, а значение y равно 23 (сумма 11 и 12).

НА ЗАМЕТКУ

Мы уже знаем, что операндами оператора + могут быть не только числа, но и текстовые значения. Если оба операнда — текстовые, то выполняется конкатенация строк. Если один из операндов — текст, а другой операнд — число, то выполняется автоматическое приведение числового значения к текстовому с последующей конкатенацией строк.

Помимо этого, допускается использование других арифметических операторов с текстовыми операндами, которые являются текстовым представлением числа. Подробнее данная тема обсуждается в разделе, посвященном преобразованию типов.

Операторы сравнения

Операторы *сравнения* используются для сравнения значений (обычно числовых). В табл. 1.2 представлены основные операторы сравнения, используемые в JavaScript. Все операторы бинарные (у каждого по два операнда), а результатом возвращается значение логического типа (true, если соответствующее соотношение истинно, и false в противном случае).

Если речь идет о сравнении числовых значений, то назначение каждого из операторов достаточно очевидно без дополнительных пояснений. Подчеркнем лишь принципиальные отличия между операторами == и ===, а также != и !==.

Опять же, если речь идет о сравнении числовых значений, то разницы, например, между операторами == и === нет. Скажем, результатом выражения 123==123 является значение true, и результатом выражения 123===123 также является значение true. Но вот результаты выражений 123=="123" и 123==="123" разные. Результатом выражения 123="123" является значение true. Значение выражения 123==="123" равно false. Второй случай, скорее всего, вопросов не вызовет: вполне логично, что числовое значение 123 не равно тексту "123". Почему же тогда результат выражения 123=="123" равен true? Все дело в автоматическом преобразовании типов. При вычислении выражения 123==="123" автоматическое преобразование типов не применяется, поэтому получаем вполне ожидаемый результат. А вот при вычислении выражения 123=="123" текстовое значение "123", являющееся на самом деле текстовым представлением числа 123, преобразуется в числовое значение 123. В результате получается, что оба операнда одинаковы.

Таблица 1.2. Операторы сравнения JavaScript

Оператор	Описание
==	Оператор «равно». Результатом выражения A==B является значение true, если значение операнда A *равно* значению операнда B. В противном случае значение выражения равно false. Если значения операндов A и B относятся к разным типам, автоматически предпринимается попытка приведения значений к одному типу для проведения сравнения
!=	Оператор «неравно». Результатом выражения A!=B является значение true, если значение операнда A *не равно* значению операнда B. В противном случае значение выражения равно false. Для значений операндов A и B разных типов автоматически выполняется приведение к одному типу (если возможно) для проведения сравнения
===	Оператор «строго равно». Результатом выражения A===B является значение true, если значение операнда A *равно* значению операнда B. В противном случае значение выражения равно false. В отличие от оператора == в данном случае приведение к одному типу значений операндов не выполняется
!==	Оператор «строго неравно». Результатом выражения A!==B является значение true, если значение операнда A *не равно* значению операнда B. В противном случае значение выражения равно false. В отличие от оператора != не выполняется приведение к одному типу значений операндов
>	Оператор «больше». Результатом выражения A>B является значение true, если значение операнда A *больше* значения операнда B. В противном случае значение выражения равно false
>=	Оператор «больше или равно». Результатом выражения A>=B является значение true, если значение операнда A *больше или равно* значению операнда B. В противном случае значение выражения равно false
<	Оператор «меньше». Результатом выражения A<B является значение true, если значение операнда A *меньше* значения операнда B. В противном случае значение выражения равно false
<=	Оператор «меньше или равно». Результатом выражения A<=B является значение true, если значение операнда A *меньше или равно* значению операнда B. В противном случае значение выражения равно false

Точно так же обстоят дела с операторами != и !==. В первом случае при сравнении операндов при необходимости выполняется автоматическое преобразование типов, во втором случае — нет.

Логические операторы

Логические операторы используются при работе с логическими значениями (напомним, их всего два — true и false). Логических операторов в JavaScript немного (всего три), и каждый из них соответствует определенной логической операции. В табл. 1.3 перечислены логические операторы языка JavaScript.

Таблица 1.3. Логические операторы JavaScript

Оператор	Описание
&&	Оператор для вычисления *логического и*. Оператор бинарный. Результатом выражения A&&B при обоих логических операндах A и B является значение true, если оба операнда равны true. Если хотя бы один из двух операндов равен false, результатом выражения A&&B является значение false
||	Оператор для вычисления *логического или*. Оператор бинарный. Результатом выражения A||B (при условии, что оба операнда A и B относятся к логическому типу) является значение true, если хотя бы один из операндов равен true. Если оба операнда равны false, результатом выражения A||B является значение false
!	Оператор *логического отрицания*. Унарный оператор. Если значение операнда A равно true, то результат выражения !A равен false. И наоборот, если значение операнда равно false, то результатом выражения !A является значение true

Существует формальное определение таких операций, как *логическое и*, *логическое или*, *логическое отрицание*, и некоторых других. В табл. 1.4 проиллюстрированы возможные результаты операции вида A&&B для различных значений операндов A и B.

Таблица 1.4. Таблица истинности для операции логического и

A B	true	false
true	true	false
false	false	false

Аналогичная информация, но только для операции вида A||B, приведена в табл. 1.5.

Таблица 1.5. Таблица истинности для операции логического или

A B	true	false
true	true	true
false	true	false

Есть два важных обстоятельства, на которые стоит обратить внимание. Во-первых, операторы *логического и* && и *логического или* || обрабатываются по упрощенной схеме. Так, несложно заметить, что если первый операнд A в выражении A&&B равен false, то значение всего выражения также будет false, вне зависимости от значения второго операнда B. Поэтому при вычислении выражения вида A&&B сначала вы-

числяется значение первого операнда A, и если значение равно false, то второй операнд B уже не вычисляется.

Аналогично обстоят дела с вычислением выражений вида A||B. Только теперь правило такое: если первый операнд A равен true, то значение всего выражения также равно true, вне зависимости от значения второго операнда B. В соответствии с этим происходит вычисление выражений A||B. Сначала вычисляется значение операнда A. Если значение равно true, второй операнд B не вычисляется.

> **НА ЗАМЕТКУ**
>
> На первый взгляд, механизм расчета результата выражений с операторами *логического и* и *логического или* представляется не столь уж и важным. Тем не менее его не стоит сбрасывать со счетов. Иногда случается так, что значение второго операнда не определено или определено так, что попытка его вычисления приводит к ошибке. В данном отношении механизм вычисления логических выражений по упрощенной схеме представляется относительно надежным.

Следующий важный момент связан с тем, что в принципе операндами в логических выражениях не обязательно должны быть значения логического типа. В таких случаях применяется автоматическое преобразование типов. Чтобы понять, каков результат того или иного выражения, необходимо учесть следующее.

- Если в выражении A&&B оба операнда интерпретируются как истинные, результатом возвращается значение второго операнда. Если хоть один операнд интерпретируется как ложный, результатом выражения A&&B возвращается значение первого «ложного» операнда.
- Результатом выражения A||B возвращается первый «истинный» операнд — если такой есть. Если оба операнда интерпретируются как ложные, то результатом выражения A||B возвращается значение второго операнда.

Так, числовые значения при необходимости трансформируются в логические по следующему правилу: ненулевые числа интерпретируются как логическое значение true, а нулевые числа интерпретируются как логическое значение false. Поэтому, например, значение выражения 1&&2 равно 2, поскольку оба операнда 1 и 2 ненулевые и интерпретируются как true, а результатом возвращается значение второго операнда (реальное значение, а не интерпретируемое). Поэтому

получаем результат 2. А вот результатом выражения 2&&0 является значение 0 — значение первого ложного операнда. Далее, значением выражения 0&&false является число 0, а значением выражения false&&0 является false. Объяснение простое: если хотя бы один из операндов интерпретируется как ложный, то результатом возвращается значение первого «ложного» операнда. Первым ложным операндом в выражении 0&&false является 0, а первым ложным операндом в выражении false&&0 является false.

НА ЗАМЕТКУ

Читатель может проверить свою интуицию и объяснить, почему результатом выражения 1||2 является значение 1, результатом выражения 0||2 является значение 2, результатом выражения 0||false является значение false, а результатом выражения false||0 является значение 0.

Оператор *логического отрицания* унарный. В табл. 1.6 проиллюстрирован результат его применения к значениям логических типов.

Таблица 1.6. Таблица истинности для операции логического отрицания

A	true	false
!A	false	true

Если операндом является значение типа, отличного от логического, автоматически выполняется попытка преобразования значения операнда к логическому типу. Поэтому результатом операции логического отрицания всегда является логическое значение. Например, результатом выражения !1 является значение false, а результатом выражения !0 является значение true.

НА ЗАМЕТКУ

Чтобы узнать, как некоторый «объект» A будет интерпретироваться при преобразовании к логическому типу, можно воспользоваться выражением !!A, в котором использовано двойное логическое отрицание (логическое отрицание логического отрицания).

Логические операторы обычно используются при составлении сложных условий в условных операторах и других управляющих инструкциях. Все это мы еще будем обсуждать в книге.

НА ЗАМЕТКУ

Нередко на практике возникает необходимость в выполнении такой операции, как *логическое исключающее или*. Специального оператора в языке JavaScript для данной операции нет. Однако соответствующее действие можно «сымитировать» с помощью операторов &&, || и !.

Если формально обозначить через *xor* (в JavaScript нет такого оператора!) операцию вычисления исключающего или, то результатом операции *A xor B* должно быть значение true, если значение одного из операндов равно true, а значение другого операнда равно false. Если же у обоих операндов одинаковые значения (не важно, оба операнда равны true, или оба операнда равны false), результат операции *A xor B* должен быть равен false.

Означенным выше условиям соответствует выражение !(A&&B)&&(A||B). Его значение равно true при разных значениях операндов A и B (когда один операнд равен true, а другой операнд равен false), и его значение равно false при совпадающих значениях аргументов.

Побитовые операторы

Имеется группа операций, которые выполняются с целыми числами на уровне их битового представления. Такие операции называются *побитовыми* или *поразрядными*, как и соответствующие им операторы. Для понимания принципов выполнения таких операций необходимо иметь некоторое представление о двоичной системе счисления и способах битового представления чисел в компьютере.

ДЕТАЛИ

В двоичной системе счисления числа записываются с помощью всего двух цифр — нулей и единиц. Например, двоичный код 101 соответствует в десятичной системе счисления числу 5, а код 10101 является двоичным представлением десятичного числа 21. В общем случае, если двоичный код представлен комбинацией вида $b_n b_{n-1} \ldots b_2 b_1 b_0$, в котором параметры b_k (индекс k = 0, 1, ..., n) могут принимать только одно из двух значений 0 или 1, то перевести такое число из двоичного представления в десятичное можно по формуле $\overline{b_n b_{n-1} \ldots b_2 b_1 b_0} = \sum_{k=0}^{n} 2^k \cdot b_k = 2^0 \cdot b_0 + 2^1 \cdot b_1 + \ldots + 2^n \cdot b_n$.

Сумма чисел, записанных в двоичном формате, вычисляется так же просто, как и сумма десятичных чисел. Достаточно лишь учесть

правила бинарного сложения чисел. А именно, 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1 и 1 + 1 = 10.

Есть определенные особенности в компьютерном представлении в бинарном коде отрицательных чисел. Дело в том, что с математической точки зрения отрицательное число — число со знаком «минус».

На бумаге такой «минус» очень легко написать, но как реализовать его в компьютере на уровне битового представления числа? Выход нашли в концепции использования старшего бита как знакового.

Другими словами, в компьютерном представлении старший бит в двоичном коде является идентификатором знака числа: нулевой старший бит соответствует положительным числам, а единичный старший бит соответствует отрицательным числам. Но это еще не все.

Представление отрицательных чисел бинарным кодом в компьютере выполняется иначе по сравнению с положительными числами. Чтобы понять идеологию подхода, рассмотрим небольшую иллюстрацию.

Допустим, имеется некоторое положительное число, которое обозначим как *A*. У него есть двоичный код, и мы полагаем, что он известен. Нам нужно записать двоичный код для числа *–A*. Но что такое число *–A*?

Будем исходить из того, что число *–A* такое, что, если его прибавить к числу *A*, получим в результате 0. То есть по определению *–A* + *A* = 0.

Далее рассмотрим такую операцию, как побитовая инверсия. Суть ее в том, что в двоичном коде все единицы меняются на нули, а все нули меняются на единицы.

Если через *~A* обозначить число, которое получается побитовым инвертированием числа *A*, то сумма *~A* + *A* будет числом, двоичный код которого состоит только из единиц. Объяснение простое: в двоичном коде там, где в числе *A* стоит единица, в числе *~A* будет ноль, и наоборот. Складывая соответствующие биты, в каждом разряде получим единицу.

Допустим, что для записи чисел в компьютере используется *n* битов. Это означает, что каждое целое число представлено в двоичном коде последовательностью из *n* нулей и единиц. Число *~A* + *A*, таким образом, будет состоять из *n* единиц. Теперь прибавим к числу *~A* + *A* единицу.

Формально число *~A* + *A* + 1 состояло бы из единицы в старшем разряде и еще *n* нулей во всех прочих разрядах. Но поскольку речь идет не просто о числе, а о реализации числа в компьютере, который «запоминает» только *n* битов, то старший единичный бит теряется. Поэтому в компьютере число *~A* + *A* + 1 состоит из *n* нулей, что со-

ответствует числу нуль. Таким образом, с учетом специфики реализации чисел в компьютере можем записать ~*A* + *A* + 1 = 0. С другой стороны, ~*A* + *A* = 0. Вывод простой: –*A* = ~*A* + 1. Другими словами, чтобы по коду положительного числа *A* восстановить код отрицательного числа –*A*, необходимо инвертировать код числа *A* и прибавить к нему единицу.

Например, нам нужно определить код числа –5. Для простоты полагаем, что числа записываются с помощью четырех битов. Это означает, что каждое число записывается кодом из четырех цифр. Для числа 5 двоичный код 0101. После инвертирования получаем код 1010. После прибавления единицы получаем 1011. Это и есть бинарный код числа –5.

Если необходимо выполнить обратную процедуру (по бинарному коду отрицательного числа определить это число в десятичном представлении), выполняем такие действия:

- инвертируем код отрицательного числа;
- прибавляем к полученному коду единицу;
- переводим полученный код для положительного числа в десятичное представление;
- добавляем знак «минус».

В частности, если имеем исходный код 1011, то для определения соответствующего ему десятичного отрицательного числа поступаем так:

- инвертируем код и получаем 0100;
- прибавляем единицу и получаем 0101;
- переводим в десятичный формат, что дает
  $0101 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 = 1 + 4 = 5$;
- добавляем знак «минус», в результате чего получаем число –5.

Напомним еще раз, что признаком отрицательности числа является единичный старший бит.

Формулу $\overline{b_n b_{n-1} \ldots b_2 b_1 b_0} = \sum_{k=0}^{n} 2^k \cdot b_k$ перевода двоичного кода в десятичный можно использовать, только если старший бит равен нулю (то есть $b_n = 0$). Если старший бит единичный, используется алгоритм вычисления отрицательного числа по бинарному коду.

Основные побитовые операторы языка JavaScript перечислены в табл. 1.7. За исключением оператора ~ побитовой инверсии, все операторы бинарные (соответственно, оператор побитовой инверсии унарный).

Операции вычисления *побитового и*, *побитового или* и *исключающего побитового или* выполняются по одной общей схеме. Сравниваются побитовые представления операндов.

Таблица 1.7. Побитовые операторы JavaScript

Оператор	Описание
&	Оператор *побитового и*. Результат выражения A&B представляет собой число, побитовое представление которого получается на основе побитовых представлений чисел A и B. Сравниваются биты попарно: по одному биту из A и из B. При сравнении двух битов результат равен 1, если оба бита равны 1. В противном случае (если хотя бы один из двух битов равен 0) результат сравнения равен 0
|	Оператор *побитового или*. Результат выражения A|B вычисляется на основе побитовых представлений чисел A и B. Сравниваются соответствующие биты в представлении чисел A и B. При сравнении битов результат равен 1, если хотя бы один бит равен 1. Если оба биты равны 0, результат сравнения равен 0
^	Оператор *побитового исключающего или*. Результат выражения A^B представляет собой число, побитовое представление которого получается на основе побитовых представлений чисел A и B. Сравниваются биты попарно: по одному биту из A и из B. При сравнении двух битов результат равен 1, если один бит равен 1, а другой бит равен 0. Если оба биты равны 1 или оба биты равны 0, результатом сравнения битов является число 0
~	Оператор *побитового отрицания*. Результатом выражения !A является число, получающееся заменой в двоичном представлении числа A нулей на единицы и единиц на нули (число A при этом не меняется)
<<	Оператор *побитового сдвига влево*. Результатом выражения A<<n является число, которое получается из двоичного представления числа A путем смещения (или сдвига) всего кода влево на n позиций. Старшие биты при этом теряются, а младшие заполняются нулями
>>	Оператор *побитового сдвига вправо*. Результатом выражения A>>n является число, которое получается из двоичного представления числа A путем смещения (или сдвига) всего кода вправо на n позиций. Младшие биты при этом теряются, а старшие заполняются значением знакового бита
>>>	Оператор *побитового сдвига вправо* (с замещением старших битов нулями). Оператор *побитового сдвига вправо*. Результатом выражения A>>n является число, которое получается из двоичного представления числа A путем смещения (или сдвига) всего кода вправо на n позиций. Младшие биты при этом теряются, а старшие заполняются нулями

Попарно сравнивая биты, находящиеся на одинаковых позициях в представлении операндов, получаем побитовое представление для числа-результата. Фактически разница лишь в правилах, которые используются при сравнении битов. Для операции *побитового и* набор возможных исходов сравнения проиллюстрирован в табл. 1.8.

Таблица 1.8. Таблица истинности для операции побитового и

A B	1	0
1	1	0
0	0	0

Например, вычислим значение выражения 22&27. Поскольку $22 = 16 + 4 + 2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$, то в двоичном представлении число 22 имеет вид 10110.
Аналогично имеем $27 = 16 + 8 + 2 + 1 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$, что в итоге дает код 11011. Таким образом, получаем: $\begin{array}{r} 10110 \\ \& \ 11011 \\ \hline 10010 \end{array}$.
Число с двоичным кодом 10010 соответствует значению $10010 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 16 + 2 = 18$. То есть результатом выражения 22&27 является значение 18.

ДЕТАЛИ

Вообще в JavaScript для представления чисел используется 4 байта или 32 бита. Но, поскольку здесь мы оперируем с небольшими по модулю положительными числами, старшие нулевые биты можно игнорировать — наличие этих битов на конечном результате не сказывается в силу как раз того обстоятельства, что они нулевые. Хотя технически такие биты присутствуют, и забывать об этом не стоит.

В табл. 1.9 представлены правила сравнения битов при выполнении операции *побитового или*.

Таблица 1.9. Таблица истинности для операции побитового или

A B	1	0
1	1	1
0	1	0

Если вычислить результат выражения 22|27, получим значение 31. Действительно, учитывая правила сравнения битов для операции *побитового или*, можем записать следующее: $\begin{array}{r} 10110 \\ | \ 11011 \\ \hline 11111 \end{array}$. Отсюда, учитывая, что $11111 = 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 16 + 8 + 4 + 2 + 1 = 31$, получаем результат 31. Для операции *побитового исключающего или* правила сравнения битов такие, как в табл. 1.10.

Таблица 1.10. Таблица истинности для операции побитового исключающего или

A B	1	0
1	0	1
0	1	0

Рассмотрим выражение `22^27`.

Легко получаем следующее: $\begin{array}{r} 10110 \\ \wedge\ 11011 \\ \hline 01101 \end{array}$.

Поскольку $01001 = 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 1 = 13$, то результат выражения `22^27` равен `13`.

Наконец, простой для понимания является операция *побитовой инверсии* (правила преобразования битов приведены в табл. 1.11).

Таблица 1.11. Таблица истинности для операции побитовой инверсии

`A`	`1`	`0`
`~A`	`0`	`1`

Учитывая правила записи и обработки отрицательных чисел, легко сообразить, что если в переменную `A` записано некоторое число x, то результатом выражения `~A` является значение $-(x + 1)$. Например, значение выражения `~6` равно `-7`, а результат выражения `~-5` равен `4`.

НА ЗАМЕТКУ

Стоит заметить, что операции *побитового и*, *побитового или* и *побитовой инверсии* имеют много общего соответственно с операциями *логического и*, *логического или* и *логического отрицания*.

Нелишним будет прокомментировать принципы вычисления выражений с операторами сдвига. Для простоты будем исходить из того, что числа кодируются восемью битами. Начнем с оператора `<<` сдвига влево. Результатом выражения `A<<n` является число, получающееся сдвигом влево на `n` позиций бинарного кода числа `A`. Так, результатом выражения `7<<3` является число `56`. Действительно, код числа 7 в восьмибитовом представлении равен 00000111. При сдвиге кода на 3 позиции влево получаем код 00111000. А это код числа 56, поскольку $00111000 = 2^5 + 2^4 + 2^3 = 32 + 16 + 6 = 56$. Вообще сдвиг бинарного кода на одну позицию влево эквивалентен умножению числа на 2. Поэтому сдвиг кода числа `A` на `n` позиций влево означает умножение числа `A` на `2` в степени `n`. Это замечание относится как к положительным, так и к отрицательным числам.

Если сдвиг влево означает умножение числа на 2, то сдвиг вправо на одну позицию означает целочисленное деление числа на 2. Например,

результатом выражения `56>>4` является значение 3, которое можно получить, либо поделив четыре раза число 56 на 2 (при целочисленном делении 7 на 2 получаем результат 3), либо непосредственно проанализировав операции с двоичным кодом. А именно, если сместить код 00111000 на четыре позиции вправо, получим код 00000011. Это число 3.

ДЕТАЛИ

При сдвиге вправо кода для отрицательных чисел результат может показаться неожиданным. Например, значение выражения `-56>>4` есть число `-4`. Объяснение такое, что, когда мы будем четыре раза делить число –56 на 2, после трех таких делений ожидаемо получим значение –7. При целочисленном делении числа –7 на 2 получаем результат –4, поскольку $-7 = -4 \cdot 2 + 1$. Последнее соотношение призвано подчеркнуть тот факт, что остаток от целочисленного деления на 2 должен быть числом 0 или числом 1. Можно сказать, что это следует из самого определения операции целочисленного деления. Но на самом деле ситуация такова, что целочисленное деление на 2 выполняется сдвигом вправо на одну позицию. Поясним сказанное на рассматриваемом примере.

Определим код для числа –56. Бинарный код числа 56 равен 00111000. После побитового инвертирования получаем код 11000111. Прибавляем единицу и получаем 11001000. Это и есть искомый код для числа –56. Теперь выполняем сдвиг кода на 4 позиции вправо. При этом младшие биты теряются, а старшие заполняются единицами (поскольку знаковый бит единичный). Получаем код 11111100. Выясним, что же это за число. Выполняем инверсию кода и получаем 00000011. Прибавляем единицу, получаем 00000100. Это число 4, а исходное число, соответствующее коду 11111100, равно –4.

Оператор `>>>` отличается от оператора `>>` тем, что старшие биты заполняются нулями вне зависимости от значения знакового бита.

Сокращенные формы оператора присваивания

В JavaScript существуют специальные сокращенные формы оператора присваивания. Поясним на примере. Допустим, необходимо выполнить команду вида `A = A ☺ B`, причем символом ☺ обозначен один из бинарных арифметических или побитовых операторов. Тогда соответствующую команду можно записать в более простом, или, точнее, компактном виде, а именно как `A ☺ = B`. В табл. 1.12 пе-

речислены основные варианты сокращенных форм оператора присваивания.

Таблица 1.12. Сокращенные формы оператора присваивания в JavaScript

Оператор	Пример использования	Эквивалентная команда
+=	A+=B	A=A+B
-=	A-=B	A=A-B
*=	A*=B	A=A*B
/=	A/=B	A=A/B
%=	A%=B	A=A%B
&=	A&=B	A=A&B
|=	A|=B	A=A|B
^=	A^=B	A=A^B
<<=	A<<=n	A=A<<n
>>=	A>>=n	A=A>>n
>>>=	A>>>=n	A=A>>>n

Перечисленные операторы будут время от времени использоваться в программных кодах. Наличие возможности не означает необходимости ее использовать. Поэтому читатель волен в своем выборе способа оформления программного кода и, в частности, использования или неиспользования сокращенных форм оператора присваивания.

Тернарный оператор

Обычно операторы — унарные или бинарные. У них соответственно один и два операнда. Но в языке JavaScript есть *тернарный оператор*: у этого оператора сразу три операнда. Тернарный оператор представляет собой упрощенную форму *условного оператора* (о котором речь пойдет позже). Шаблон вызова тернарного оператора такой (жирным шрифтом выделены ключевые элементы шаблона):

условие **?** значение_1 **:** значение_2

Синтаксис у команды вызова оператора достаточно простой. Первым операндом указывается условие, которое проверяется при вычислении выражения с тернарным оператором. После условия следует вопросительный знак ? (синтаксический элемент тернарного оператора) и значение (второй операнд), которое возвращается результатом

выражения, если условие истинно. Затем следует двоеточие : и еще одно значение (третий операнд тернарного оператора). Данное значение возвращается, если условие ложно.

> **НА ЗАМЕТКУ**
>
> Поскольку разделителями операндов в тернарном операторе является вопросительный знак ? и двоеточие :, то обычно тернарный оператор обозначают как ?:.

Например, рассмотрим выражение `X?A:B`. Если значение `X` равно `true` (или интерпретируется как `true`), то результатом выражения будет значение `A`. Если значение `X` равно `false` (или интерпретируется как `false`), результат выражения равен `B`.

Преобразование типов

Результат вычисления выражений не всегда очевиден. Причина во многом кроется в автоматическом преобразовании типов. Обычно такие ситуации возникают, когда в выражении операнды разных типов. Вообще возможных вариантов очень много, особенно учитывая специфику использования функций и объектов. Поскольку мы с этими синтаксическими конструкциями не знакомы, ограничимся пока что наиболее простыми ситуациями, которые могут встретиться на пути освоения премудростей языка JavaScript на начальном этапе. Некоторые алгоритмы автоматического преобразования типов ранее уже упоминались.

> **НА ЗАМЕТКУ**
>
> Наша стратегия состоит в том, чтобы решать проблемы по мере их возникновения. Вопросы, связанные с преобразованием типов, критичные для понимания материала книги, будут освещаться по мере необходимости, чтобы не перегружать читателя. Здесь мы рассмотрим лишь некоторые аспекты, связанные с преобразованием типов.

Выделим основные моменты, на которые следует обратить внимание.

- Отличные от нуля числа могут использоваться в качестве логических значений. Отличные от нуля числа интерпретируются как `true`, нуль интерпретируется как `false`.

- Если в арифметическом выражении с вычислением разности, произведения, частного (но не сложения!), побитового сдвига или другой побитовой операции имеются операнды, являющиеся текстовым представлением числа, такое текстовое представление автоматически преобразуется в число. Например, результатом выражения "300"-"200" является число 100. Или, скажем, результатом выражения "32">>"3" является число 4. Это правило, однако, не распространяется на операцию сложения. Например, результатом выражения "300"+"200" является текстовое значение "300200" (объединение текстовых строк).
- Если один из операндов в выражении вычисления суммы текстовый, а другой операнд числовой, то числовой операнд приводится к текстовому формату и выполняется конкатенация строк. Это же имеет место в том случае, если текстовый операнд является представлением числа. Выше отмечалось, что результат выражения 300+"200" равен "300200", — так же, как и результат выражения "300"+200.
- Для приведения текстового представления числа к числовому типу можно указать унарный оператор плюс перед соответствующим текстовым литералом. Например, результатом выражения +"300"+200 является число 500. Такой же результат получим при вычислении значения выражения +"300"+ +"200". Здесь использованы два оператора плюс подряд (но между ними есть пробел!). Это не оператор инкремента, а два разных оператора. Первый оператор плюс — бинарный, а второй оператор плюс — унарный (знаковый). Данная схема работает, даже если в текстовом литерале «спрятано» отрицательное число. Например, выражение +"300"+ +"-200" имеет смысл. Значение выражения есть число 100. То есть здесь унарный оператор «плюс» выполняет роль «извлекателя» числа из текста.

Могут иметь место и другие достаточно специфические ситуации, связанные с явным или неявным преобразованием типов. Все они при необходимости будут прокомментированы.

НА ЗАМЕТКУ

Вообще есть хороший рецепт, который состоит в том, чтобы избегать всяческих непонятных ситуаций. Обычно надежнее реорганизовать программный код так, чтобы результат его выполнения был предсказуем. Как говорится, самый короткий путь — тот, который знаком. Хотя и для нового нужно быть открытым. Такая вот диалектика.

Приоритет операторов

В JavaScript довольно много операторов. Рассмотренные выше операторы составляют далеко не полный список. Анализ сложных выражений, состоящих из большого количества внутренних инструкций, может представлять значительные сложности. Поэтому желательно иметь хотя бы общее представление о приоритете операторов.

ДЕТАЛИ

Приоритет — понятие относительное. Приоритет операторов влияет на прядок вычисления подвыражений в сложном выражении. Подвыражения с операторами, имеющими более высокий приоритет, вычисляются первыми. После них вычисляются подвыражения с операторами, имеющими более низкий приоритет.

Если некоторые операторы имеют одинаковый приоритет, то соответствующие подвыражения вычисляются последовательно одно за другим (обычно слева направо в порядке появления подвыражений в выражении).

В табл. 1.13 рассмотренные нами ранее операторы собраны в группы по приоритетности. Приоритет каждой группы операторов указан в виде целого числа (хотя это условный показатель). Чем меньше число, тем выше приоритет операторов — то есть в таблице операторы размещены в порядке убывания приоритета.

НА ЗАМЕТКУ

Помимо уже знакомых нам операторов, в табл. 1.13 представлены еще два оператора: унарный «плюс» и унарный «минус». Речь идет об операторах, которые определяют знак числа. Знак минус для отрицательных чисел указывается в обязательном порядке. Для положительных чисел знак плюс обычно не указывают (хотя это можно сделать).

Также стоит заметить, что у префиксной и постфиксной форм операторов инкремента и декремента разный приоритет.

Понятно, что запомнить всю эту таблицу практически нереально. Да это и не нужно. Что можно порекомендовать? Хороший подход состоит в использовании круглых скобок. Круглые скобки имеют высший приоритет и позволяют изменять порядок вычисления подвыражений в выражении. Помимо этого, выражения, содержащие круглые скобки, обычно легче анализировать. Так что круглые скобки — это не только функциональный, но еще и неплохой «декоративный» элемент.

Таблица 1.13. Приоритетность операторов в JavaScript

Приоритет (по убыванию)	Название	Оператор
1	Круглые скобки (используются для группировки выражений)	(и)
2	Постфиксная форма инкремента	++
	Постфиксная форма декремента	--
3	Логическое отрицание	!
	Побитовая инверсия	~
	Унарный «плюс»	+
	Унарный «минус»	-
	Префиксная форма инкремента	++
	Префиксная форма декремента	--
4	Умножение	*
	Деление	/
	Остаток от целочисленного деления	%
5	Сложение	+
	Вычитание	-
6	Побитовый сдвиг влево	<<
	Побитовый сдвиг вправо	>>
	Побитовый оператор сдвига вправо (с заполнением старшего бита нулями)	>>>
7	Меньше	<
	Меньше или равно	<=
	Больше	>
	Больше или равно	>=
8	Равенство	==
	Неравенство	!=
	Строгое равенство	===
	Строгое неравенство	!==
9	Побитовое «и»	&
10	Побитовое «исключающее или»	^
11	Побитовое «или»	|
12	Логическое «и»	&&
13	Логическое «или»	||
14	Тернарный оператор	?:
15	Операторы присваивания	=
		+=
		-=
		*=
		/=
		%=
		<<=
		>>=
		>>>=
		&=
		^=
		|=

Резюме

Прием окончен. Обеденный перерыв!

из к/ф «Иван Васильевич меняет профессию»

В этой главе мы узнали следующее.

- Сценарий можно непосредственно разместить в HTML-коде веб-документа, а можно сохранить в отдельном файле и затем загрузить файл со сценарием при открытии веб-документа. При сохранении сценария в отдельном файле в блоке <script> адрес файла для загрузки указывается значением атрибута src.
- Метод write() объекта document отображает текст с учетом HTML-разметки, так что в отображаемый текст можно включать HTML-дескрипторы.
- Переменные объявляются с использованием ключевого слова var. Тип переменной не указывается. В процессе выполнения программы одна и та же переменная может иметь значения разных типов.
- Основные типы данных в JavaScript: текст, числа, логические значения, объекты и функции. Текстовые литералы заключаются в одинарные или двойные кавычки. Логических значений два: true (истина) и false (ложь).
- Функция eval() позволяет вычислять выражения, «упакованные» в текстовый литерал.
- Для реализации основных действий в JavaScript имеются встроенные операторы (арифметические, логические, операторы сравнения и побитовые операторы).
- При вычислении выражений в случае необходимости имеет место автоматическое преобразование типов. Также необходимо учитывать приоритет операторов. Для изменения порядка вычисления подвыражений в выражении рекомендуется использовать круглые скобки.

Глава 2
УПРАВЛЯЮЩИЕ ИНСТРУКЦИИ

Да ты, батюшка, только скажи, а мы переймем.

из к/ф «Иван Васильевич меняет профессию»

В этой главе речь пойдет об *управляющих инструкциях* языка JavaScript. Управляющие инструкции — это условный оператор, операторы цикла, оператор выбора, ну и все, что с ними связано. Фактически далее обсуждается тот аппарат языка программирования JavaScript, который лежит в фундаменте всей концепции создания сценариев. Мы познакомимся с рабочим инструментом, без успешного владения которым невозможно стать хорошим программистом. Начнем с *условного оператора*.

Условный оператор

На улице идет дождь, а у нас идет концерт.

из к/ф «Покровские ворота»

Условный оператор позволяет выполнять разные блоки команд в зависимости от истинности или ложности некоторого условия. Нечто подобное мы наблюдали в предыдущей главе при обсуждении тернарного оператора. Но теперь все намного серьезнее.

Общий синтаксис условного оператора

В общем случае условный оператор содержит два блока кода и выражение, которое используется в качестве условия (то есть в идеале значение выражения относится к логическому типу). Первый блок выполняется в случае, если условие истинно. Если условие ложно,

выполняется второй блок команд. Это общая схема. Теперь возникает вопрос: как ее реализовать? Для условного оператора используется определенный шаблон, который представлен ниже (основные элементы шаблона выделены жирным шрифтом):

```
if(условие){
  // первый блок команд
}
else{
  // второй блок команд
}
```

Описание условного оператора начинается с ключевого слова if, после которого в круглых ((и)) скобках указывается *условие*. Далее в фигурных скобках ({ и }) размещается блок команд, выполняемых при истинности условия (то есть когда при вычислении значения выражения, указанного в круглых скобках после ключевого слова if, получаем значение true). Если условие ложно (значение выражения после ключевого слова if равно false), блок команд в фигурных скобках после ключевого слова if не выполняется, а выполняются другие команды, указанные в блоке (определяется парой фигурных скобок) после ключевого слова else.

ДЕТАЛИ

То есть выполнение условного оператора означает, что будет выполнен один и только один блок команд из двух возможных. Кроме рассматриваемой здесь общей формы условного оператора, существует еще упрощенная форма условного оператора, в которой нет else-блока. Мы рассмотрим ее позже.

Также следует отметить, что если блок команд состоит всего из одной команды, то фигурные скобки можно не использовать. Тем не менее фигурные скобки рекомендуется использовать всегда, поскольку их наличие улучшает восприятие кода и позволяет избежать случайных ошибок.

Общая схема выполнения условного оператора проиллюстрирована блок-схемой, представленной на рис. 2.1.

Поскольку лучший критерий теории — практика, имеет смысл посмотреть на условный оператор в реальном программном коде.

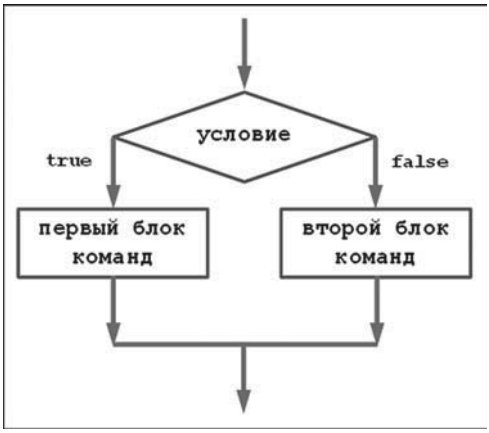

Рис. 2.1. Схема выполнения условного оператора

Пример с условным оператором

Небольшой пример, в котором используется условный оператор, приведен в листинге 2.1.

Листинг 2.1. Сценарий с условным оператором (файл Listing02_01.js)

```
var txt,name
txt="Давайте познакомимся! Как Вас зовут?"
name=prompt(txt)
if(name==""){
  document.write("Жаль, но Вы не представились!")
}
else{
  document.write("Приятно познакомиться, "+name+"!")
}
```

В первую очередь мы рассмотрим результат выполнения сценария в браузере, а уже затем проанализируем программный код сценария.

Для загрузки файла со сценарием в веб-документ используем следующий HTML-код:

```
<!DOCTYPE HTML>
<html>
```

```
<head>
  <title>Листинг 2.1</title>
</head>
<body><h3>Листинг 2.1</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing02_01.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

При открытии документа с таким кодом в окне браузера появляется диалоговое окно с полем ввода. Как все это выглядит, показано на рис. 2.2.

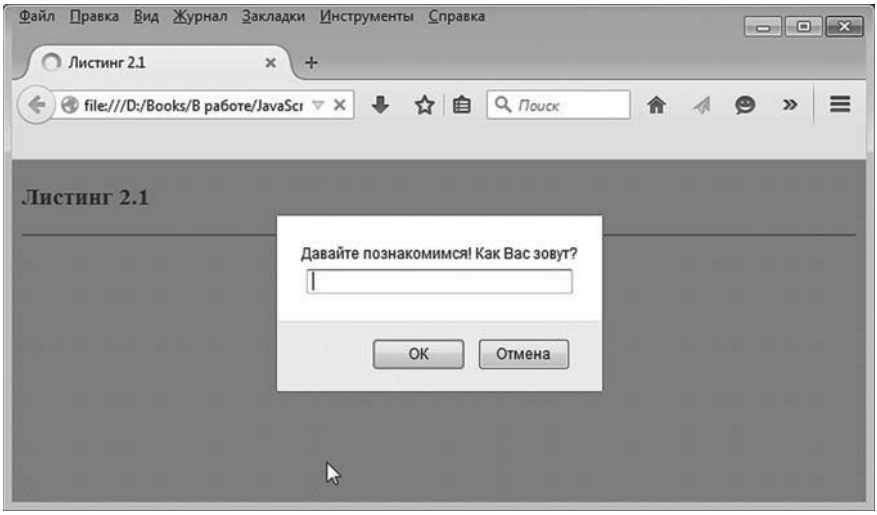

Рис. 2.2. Окно браузера при загрузке веб-документа со сценарием с условным оператором

Окно документа временно (до закрытия диалогового окна) заблокировано. У диалогового окна есть поле ввода и две кнопки (**ОК** и **Отмена**). Здесь возможны варианты. Сначала рассмотрим ситуацию, когда

все происходит стандартно: пользователь вводит в поле ввода окно и щелкает по кнопке **ОК**, как показано на рис. 2.3.

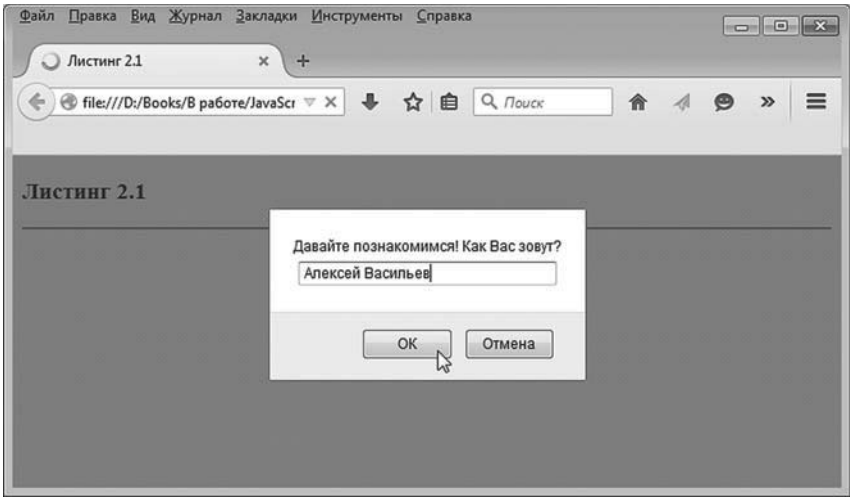

Рис. 2.3. В поле ввода вводится имя пользователя

Диалоговое окно закрывается, а в рабочем окне браузера появляется текстовое сообщение, содержащее, кроме прочего, имя, которое было введено пользователем в поле ввода. Результат описанных действий показан на рис. 2.4.

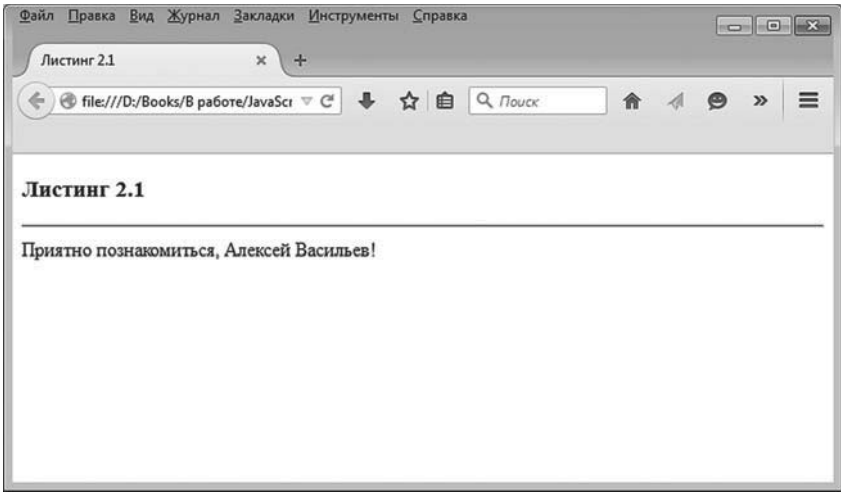

Рис. 2.4. Содержимое рабочего окна браузера после ввода пользователем имени

НА ЗАМЕТКУ

Если в браузере щелкнуть по пиктограмме перезагрузки, диалоговое окно с полем ввода появится снова.

Ситуация может развиваться в несколько ином ключе. Допустим, пользователь щелкает по кнопке **ОК** в диалоговом окне, но при этом поле ввода остается пустым (рис. 2.5).

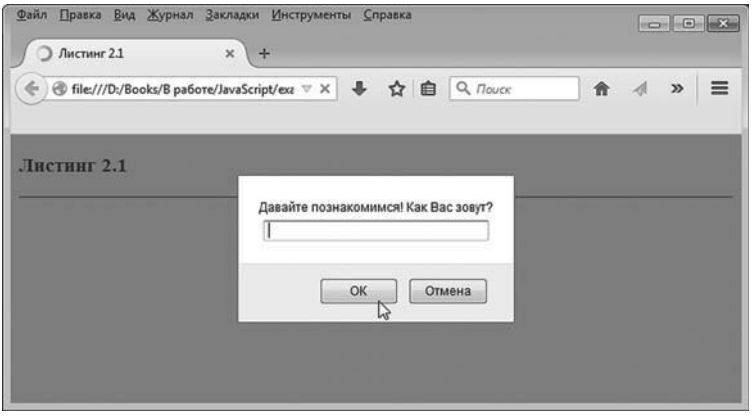

Рис. 2.5. Пользователь щелкает в диалоговом окне по кнопке OK, но при этом поле ввода не заполнено

Если так, то в рабочей области браузера появится сообщение, но уже другое (рис. 2.6).

ДЕТАЛИ

Откровенно говоря, пользователь может и не щелкать по кнопке **ОК** в диалоговом окне, а щелкнуть по кнопке **Отмена**. В этом случае метод `prompt()` возвращает пустую ссылку. В результате значением переменной `name` будет `null` — специальное значение, обозначающее пустую ссылку. При попытке отобразить значение (которое равно `null`) переменной `name` с помощью метода `write()` получим текст `"null"`.

Во избежание ненужного усложнения кода в рассматриваемом сценарии обработка ситуации, когда пользователь щелкает по кнопке **Отмена**, не предусмотрена.

Вкратце так выполняется сценарий. Теперь настал черед проанализировать его код.

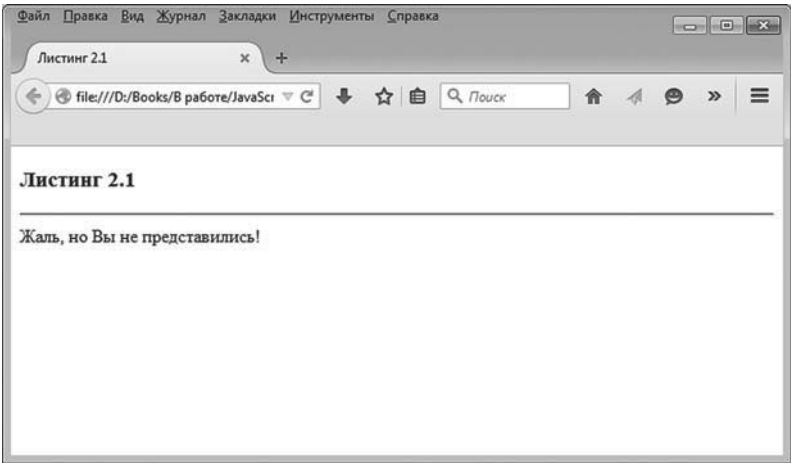

Рис. 2.6. Содержимое рабочего окна браузера после щелчка пользователем в диалоговом окне по кнопке OK при пустом поле ввода

В сценарии мы используем условный оператор и метод prompt(), с помощью которого отображается окно с полем для ввода текста (имени пользователя).

НА ЗАМЕТКУ

Хотя может показаться, что prompt() является функцией (поскольку как бы вызывается не из объекта), на самом деле — это метод. Это метод объекта окна window. Полная инструкция вызова метода выглядела бы как window.prompt(). Но поскольку объект окна window разрешается не указывать (такой вот особенный объект), то мы использовали упрощенную форму вызова метода, что создает некоторые иллюзии. Пока такой поворот дел для нас не принципиален. Нам просто необходима утилита, которая позволила бы ввести информацию в сценарий.

При вызове метода prompt() появляется, как отмечалось, окно с полем ввода. Текст, переданный аргументом методу prompt(), отображается над полем ввода. Метод возвращает результат — текст, который пользователь вводит в поле ввода в диалоговом окне (тут, правда, есть некоторые особенности, но о них немного позже).

В сценарии объявляются две переменные. Переменной txt значением присваивается текст, который предстоит отобразить в окне с полем ввода. Поэтому переменная txt передается аргументом методу prompt().

Результат вызова метода записывается в переменную name. Вся команда выглядит как name=prompt(txt).

Далее на сцену выходит условный оператор. В условном операторе проверяется условие name==””. Условие истинно, если значение переменной name равно "" (пустая текстовая строка). Если так, то командой document.write("Жаль, но Вы не представились!") в рабочей области документа отображается текст "Жаль, но Вы не представились!". В противном случае (если условие ложно), будет выполнена команда document.write("Приятно познакомиться, "+name+"!").

Текст, который отображается в документе, является объединением текстовой строки "Приятно познакомиться, ", значения переменной name и текста "!".

Упрощенная форма условного оператора

Как отмечалось ранее, у условного оператора есть упрощенная форма, которая не содержит else-ветки кода. Шаблон упрощенной формы условного оператора такой:

```
if(условие){
  // блок команд
}
```

То есть имеется if-часть оператора, где, собственно, указывается проверяемое условие, и блок команд в фигурных скобках, которые выполняются при истинном условии. Если условие ложно, команды не выполняются.

> **НА ЗАМЕТКУ**
>
> Таким образом, блок команд в условном операторе выполняется только при истинном условии. При ложном условии не происходит ничего.

На рис. 2.7 показана блок-схема, иллюстрирующая механизм выполнения условного оператора в сокращенной форме.

Наличие упрощенной формы условного оператора позволяет проявлять значительную «гибкость» при составлении программных кодов.

Посмотрим, как мог бы выглядеть сценарий, рассмотренный выше, если вместо полной формы условного оператора использовать сокращенную форму.

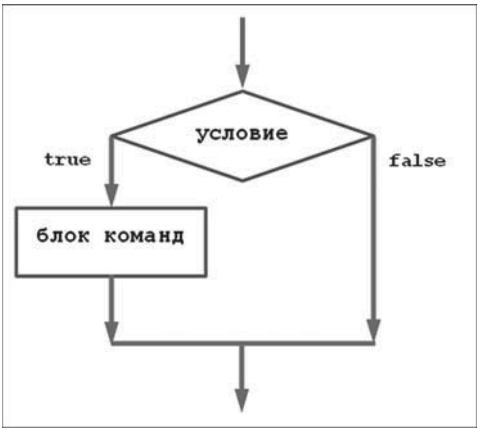

Рис. 2.7. Схема выполнения условного оператора в упрощенной форме

Пример с упрощенной формой условного оператора

Понятно, что полная и упрощенная формы условного оператора не являются эквивалентными. Вместе с тем путем реорганизации кода удается добиваться нужных результатов. Далее приводится иллюстрация к сказанному. Рассмотрим программный код сценария в листинге 2.2.

Листинг 2.2. Сценарий с условным оператором в упрощенной форме (файл Listing02_02.js)

```
var txt,name,msg
txt="Давайте познакомимся! Как Вас зовут?"
msg="Жаль, но Вы не представились!"
name=prompt(txt)
if(name!=""){
  msg="Приятно познакомиться, "+name+"!"
}
document.write(msg)
```

При создании веб-документа используем следующий HTML-код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 2.2</title>
</head>
<body><h3>Листинг 2.2</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing02_02.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Результат выполнения данного сценария точно такой же, как и в предыдущем примере (с поправкой на название листинга, которое отображается рабочем документе). Посмотрим, что изменилось в коде сценария и почему изменения в коде не сказались на внешнем результате выполнения сценария.

Формальных изменений в коде сценария не очень много. Но вся стратегия выполнения кода изменилась. Более конкретно, в сценарии, помимо переменных txt и name, появляется еще и переменная msg. Переменные txt и msg сразу получают значения. В переменную txt записывается текст, который отображается в диалоговом окне над полем ввода. В переменную msg записывается текст "Жаль, но Вы не представились!", который должен появиться в рабочем документе, если пользователь при щелчке по кнопке **ОК** оставит поле ввода пустым. Затем командой name=prompt(txt) отображается диалоговое окно и введенное пользователем значение записывается в переменную name. Следующая команда — условный оператор в упрощенной форме. Причем теперь проверяется условие name!="". Условие истинно, если значение переменной name не является пустой текстовой строкой "". Так вот если значение переменной name — не пустая строка, то командой msg="Приятно познакомиться, "+name+"!" переопределяется значение переменной msg. Если значение переменной name — пустая строка (и условие name!="" ложно), то команда в условном операторе по изменению значения переменной msg

не выполняется, и переменная остается со своим старым значением "Жаль, но Вы не представились!". Наконец, командой document.write(msg) значение переменной msg отображается в рабочем документе.

Вложенные условные операторы

Среди выполняемых в условном операторе команд могут быть другие условные операторы. В этом случае говорят о *вложенных условных операторах*. Вложенные условные операторы позволяют создавать множественные точки ветвления в программном коде. Как иллюстрацию рассмотрим небольшой пример. Соответствующий программный код представлен в листинге 2.3.

ДЕТАЛИ

Немного с опережением графика мы в сценарии создаем функцию. Вообще функциям посвящена отдельная глава. Здесь будет только маленькая зарисовка.

Вложенные условные операторы использованы в программном коде функции, которая является реализацией кусочно-гладкой зависимости $f(x)=\begin{cases}0, & x<0\\ x^2, & 0\le x<2\\ 6-x, & 2\le x<5\\ 1, & x\ge 5\end{cases}$

В сценарии описывается соответствующая функция, а затем она вызывается с разными аргументами.

Листинг 2.3. Сценарий с вложенными условными операторами (файл Listing02_03.js)

```
// Описание функции:
function f(x){
  // Если x<0
  if(x<0){
    return 0
  }
  // Если x>=0
  else{
    // Если x<2
    if(x<2){
      return x*x
```

```
    }
    // Если x>=2
    else{
      // Если x<5
      if(x<5){
        return 6-x
      }
      // Если x>=5
      else{
        return 1
      }
    }
  }
} // Окончание описания функции

document.write("<h4>Кусочно-гладкая функция</h4>")
// Добавление графика функции в документ:
document.write('<img src="function.jpg" width="500" height="300"><br>')
// Переменная для записи значения аргумента:
var z
// Значение аргумента - случайное число (от -2 до 7):
z=9*Math.random()-2
document.write("Случайный аргумент:  <b>"+z+"</b><br>")
// Вызов функции:
document.write("Значение функции:  <b>"+f(z)+"</b>")
```

В представленном сценарии достаточно много новых элементов и конструкций, так что имеет смысл сразу обсудить все основные моменты, связанные как с вложенными условными операторами, так и прочими структурными фрагментами кода.

Принципиально новым (новым в том смысле, что мы с таким еще не сталкивались) в сценарии можно полагать следующее:

- описание функции;
- вставка в веб-документ рисунка (причем с помощью кода сценария);
- генерирование случайного числа.

Самая важная часть кода — это, конечно, описание функции, в теле которой вызываются вложенные условные операторы. Что касается функциональной зависимости, которая реализуется в сценарии, то речь идет о функции от одного аргумента, которая возвращает в качестве результата числовое значение. Пикантность ситуации в том, что значения функции для разных диапазонов значений аргумента определяются разными формулами. Такая функция как бы сшита из разных «кусочков». Мы рассматриваем функцию, которая:

- при отрицательных значениях аргумента возвращает нулевое значение;
- при значениях аргумента, больших 5, возвращает единичное значение;
- если аргумент (обозначим его через x) попадает в диапазон значений от 0 до 2, результат функции вычисляется как x^2;
- если аргумент функции x находится в диапазоне значений от 2 до 5, значение функции равняется $6 - x$.

Для лучшего восприятия график функции представлен на рис. 2.8.

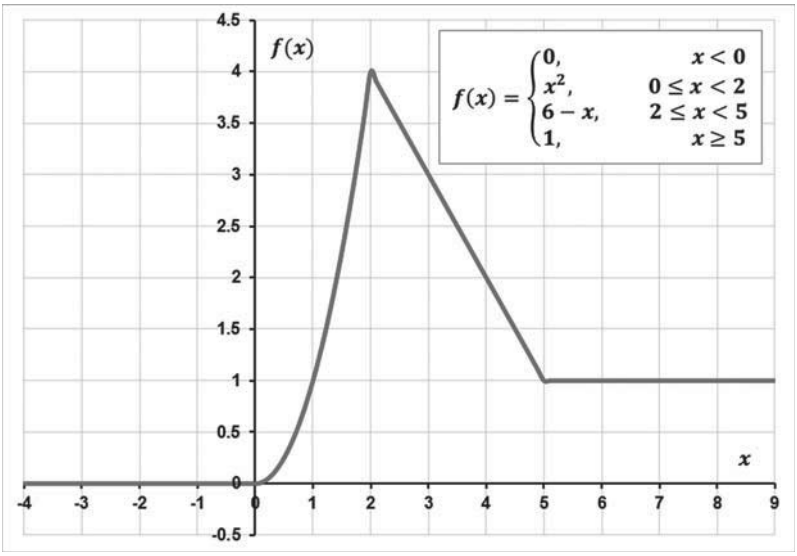

Рис. 2.8. График функции, реализованной в программном коде сценария

В сценарии описана функция f(), которая реализует описанную выше зависимость. Описание функции начинается с ключевого слова

function. Далее указывается название функции (мы незатейливо назвали функцию буквой f), а в круглых скобках — формальный аргумент, который мы обозначили как x. Тело функции (команды, которые выполняются при вызове функции) заключается в фигурные скобки.

> **НА ЗАМЕТКУ**
>
> Вообще функция — это некоторый блок кода, у которого есть имя и который по этому имени можно вызвать. При вызове функции выполняется ее программный код. У функции могут быть аргументы (или параметры), которые передаются функции при вызове и которые используются при выполнении программного кода функции. Также функция может возвращать результат. Если так, то инструкция вызова функции присваивается в качестве значения переменной, является операндом в выражении или, например, передается аргументом другой функции.
>
> Описание функции не означает выполнения ее программного кода. Программный код выполняется при вызове функции. Для вызова функции указывают ее имя и, если необходимо, аргументы, которые передаются функции (перечисляются через запятую в круглых скобках после имени функции).

В данном случае тело функции состоит из нескольких вложенных условных операторов, в которых последовательно проверяется соответствие аргумента функции некоторому критерию (попадание в определенный диапазон значений). Результат, возвращаемый функцией, определяется в соответствии с тем, в какой диапазон значений попадает аргумент функции.

Для возвращения функцией результата используется инструкция return. При выполнении команды с инструкцией return работа функции прекращается, а значение, указанное после инструкции return, возвращается в качестве результата функции. Поэтому если при проверке условия x<0 в самом первом (внешнем) условном операторе получаем значение true, выполняется команда return 0, что означает нулевое значение функции и прекращение выполнения ее кода. Если условие x<0 ложно, начинает выполняться else-блок, в котором свой условный оператор. В нем проверяется условие x<2. Данное условие проверяется уже после того, как проверка условия x<0 дала значение false. Поэтому нет необходимости проверять дополнительное условие x>=0 — оно выполняется автоматически (иначе до проверки условия x<2 дело бы не дошло).

Если условие x<2 истинно, выполняется команда return x*x, определяющая значение функции как квадрат аргумента. В противном случае выполняется else-блок данного условного оператора. В нем (сюрприз!) еще один условный оператор. При истинности условия x<5 результат возвращается инструкцией return 6-x. Если условие x<5 ложно, результат функции возвращается командой return 1. Что касается самого условия x<5, то уж если оно проверяется, то это значит, что условия x<0 и x<2 последовательно дали значение false.

Помимо описания функции, интерес представляет и прочий код сценария. Рассмотрим команды кода, размещенные в сценарии после описания функции.

Командой document.write("<h4>Кусочно-гладкая функция</h4>") отображается заголовок четвертого уровня. О том, что это именно заголовок четвертого уровня, свидетельствует пара дескрипторов <h4> и </h4>, которые включены в текстовое значение.

НА ЗАМЕТКУ

Заголовок в браузере выделяется не только жирным шрифтом, но еще и увеличенными отступами.

Следующая инструкция document.write('<img src="function.jpg" width="500" height="300">
') выполняет вставку в рабочий документ изображения из файла function.jpg (файл должен находиться в той же папке, что и веб-документ). Формально здесь речь идет о выводе в рабочей области документа дескриптора <img> вставки рисунка с соответствующими атрибутами и инструкции
 для перехода к новой строке. Поскольку выводимый текст с HTML-разметкой содержит двойные кавычки, весь текстовый литерал заключается в одинарные кавычки.

НА ЗАМЕТКУ

Напомним, что в JavaScript текстовые литералы можно заключать не только в двойные кавычки, но и в одинарные.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Для вставки изображения в веб-документ используют дескриптор <img> (закрывающего дескриптора у него нет). Дескриптор <img> со-

держит несколько атрибутов со значениями. Наиболее важный атрибут src нужен для определения файла, из которого загружается изображение. В данном случае инструкция src="function.jpg" означает, что рисунок загружается из файла function.jpg. Вообще указывается полный путь к файлу. Если указано только имя файла, то неявно подразумевается, что файл находится в том же месте, что и веб-документ.

Параметры width и height являются необязательными. Значения параметров определяют соответственно ширину и высоту области (в пикселях) для отображения рисунка (проще говоря, параметры задают ширину и высоту изображения в окне браузера). Если фактические размеры рисунка не совпадают с размерами, заданными параметрами width и height, выполняется автоматическое масштабирование изображения. Если параметры width и height не заданы, для размеров области отображения изображения используются фактические размеры рисунка.

Командой var z объявляется переменная z, в которую мы планируем записать значение аргумента для функции. Значение аргументу присваивается командой z=9*Math.random()-2. В этой команде использован вызов метода random() встроенного объекта Math. Метод random() возвращает результатом случайное действительное число в диапазоне значений от 0 до 1.

В результате переменная z принимает случайное действительное значение в диапазоне от –2 до 7.

ДЕТАЛИ

Во встроенном объекте Math реализовано много полезных математических методов. Сам объект будет обсуждаться в главах, посвященных методам объектно-ориентированного программирования. Здесь мы временно пользуемся возможностями объекта Math. В частности, нас интересует метод random(), значением которого является действительное случайное число в диапазоне от 0 до 1. Если такое число умножить на 9, получим случайное число в диапазоне возможных значений от 0 до 9. Если от полученного результата отнять 2, получим число в диапазоне возможных значений от –2 до 7.

Вообще, если через ξ обозначить случайное число, равномерно распределенное в диапазоне от 0 до 1, то случайное число $a + b \cdot x$ будет равномерно распределено в диапазоне значений от a до $a + b$. Это мы и приняли во внимание, когда использовали команду z=9*Math.random()-2 (здесь надо положить $a = -2$ и $b = 9$).

Командой document.write("Случайный аргумент: "+z+"
") в документе отображается значение аргумента, а командой document.write("Значение функции: "+f(z)+"") отображается значение функции для данного аргумента. В последнем случае аргументом методу write() передается выражение, в которое входит инструкция f(z) вызова описанной в сценарии функции с вложенными условными операторами.

НА ЗАМЕТКУ

Значение аргумента функции и значение самой функции выделяются жирным шрифтом. Стоит обратить внимание на способ включения дескрипторов и в отображаемый текст. Например, в выражении "Случайный аргумент: "+z+"
" открывающий дескриптор добавлен в конец первого текстового фрагмента, а закрывающий дескриптор включен в начало третьего фрагмента. Таким образом, при конкатенации строк значение переменной z оказывается между дескрипторами и , в силу чего отображается жирным шрифтом.

Для перехода к новой строке в конце текста добавлен дескриптор
.

Веб-документ для тестирования сценария создается с помощью следующего HTML-кода:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 2.3</title>
</head>
<body><h3>Листинг 2.3</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing02_03.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

При отображении документа в окне браузера получаем результат, как показано на рис. 2.9.

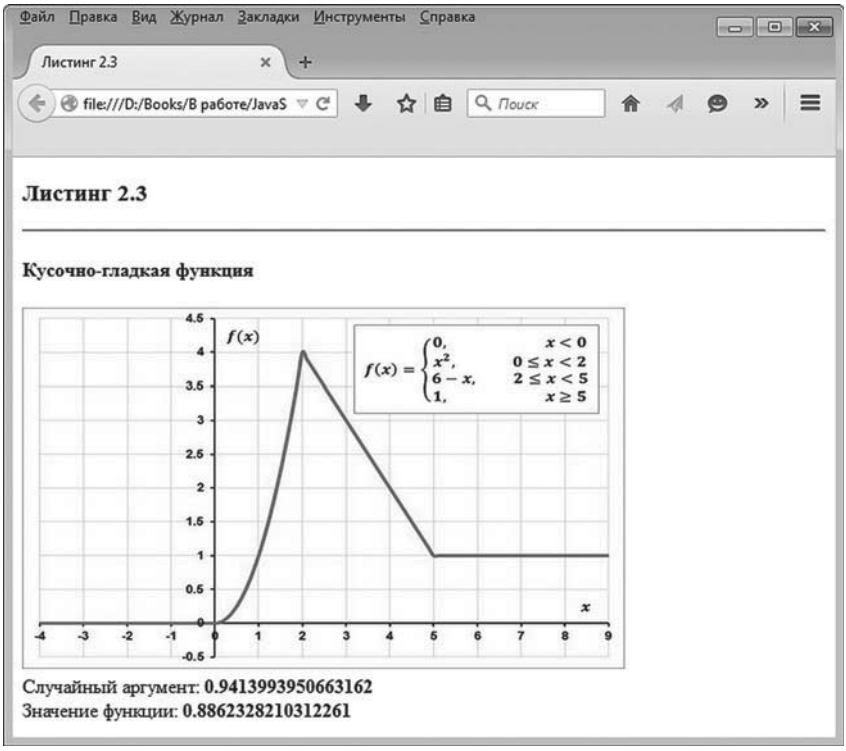

Рис. 2.9. Результат отображения документа со сценарием

Основная область документа занята изображением с графиком функции. Внизу под картинкой выводится значение аргумента (случайное число), а в следующей строке приводится значение функции в данной точке.

НА ЗАМЕТКУ

Если щелкнуть по пиктограмме перезагрузки страницы в браузере, случайным образом изменится значение аргумента, и автоматически для нового значения аргумента будет пересчитано значение функции.

Условным оператором возможности языка JavaScript не ограничиваются. В JavaScript есть несколько *операторов цикла*, которые позволяют в программном коде многократно повторять однотипные действия. Речь идет об очень мощном инструменте. Именно с ним мы и познакомимся.

Оператор цикла while

Ох, тяжко мне! Молви еще раз, ты не демон?

из к/ф «Иван Васильевич меняет профессию»

Оператор цикла while прост в использовании и интуитивно понятен. Он достаточно часто используется на практике.

Синтаксис оператора

У оператора while, пожалуй, самый минималистический синтаксис. Шаблон описания оператора такой (жирным шрифтом выделены ключевые элементы шаблона):

```
while(условие){
  // команды
}
```

После ключевого слова while в круглых скобках указывается *условие*. Оператор цикла выполняется до тех пор, пока истинно условие. Происходит все следующим образом. Сначала проверяется условие. Если условие истинно, выполняется блок команд в фигурных скобках. Затем снова проверяется условие. Если оно истинно, выполняются команды, и так далее. Выполнение оператора цикла завершается, когда при проверке условия оно оказывается ложным.

> **НА ЗАМЕТКУ**
>
> После проверки условия в операторе цикла while (при его истинности) выполняются все команды в теле оператора цикла, и лишь затем снова проверяется условие. Если блок команд состоит из одной команды, фигурные скобки можно не использовать. Тем не менее их наличие улучшает восприятие программного кода.

Принцип выполнения оператора цикла while проиллюстрирован блок-схемой, показанной на рис. 2.10.

Покажем на примере, как в программном коде используется оператор цикла while.

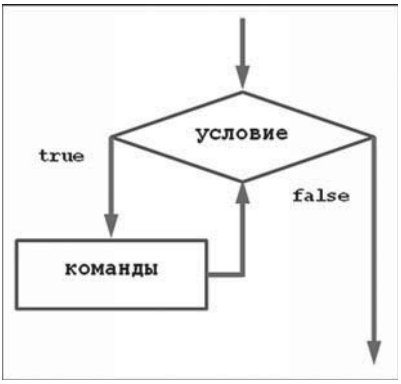

Рис. 2.10. Схема выполнения оператора цикла *while*

Пример использования оператора цикла

В представленном далее примере вычисляется сумма квадратов натуральных чисел. Основу вычислительного алгоритма составляет оператор цикла while. Обратимся к программному коду сценария, представленному в листинге 2.4.

Листинг 2.4. Вычисление суммы квадратов чисел с помощью оператора цикла while (файл Listing02_04.js)

```
// Верхняя граница суммы:
var n=100
// Индексная переменная:
var k=1
// Начальное значение суммы квадратов:
var s=0
// Текст для отображения:
var txt="1<sup>2</sup> + 2<sup>2</sup> + … + "
txt+=n+"<sup>2</sup> = "
// Вычисление суммы квадратов чисел:
while(k<=n){
  s+=k*k // Добавление нового слагаемого
  k++   // Новое значение индексной переменной
}
// Отображение результата:
document.write(txt+s)
```

В сценарии объявляется несколько переменных. Переменная n со значением 100 определяет верхнюю границу суммы — другими словами, в сценарии вычисляется сумма квадратов натуральных чисел от 1 до n включительно.

ДЕТАЛИ

Существует формула для вычисления суммы квадратов натуральных чисел. В частности, $\sum_{k=1}^{n} k^2 = 1^2 + 2^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$.

В сценарии мы эту формулу не используем, а вычисляем сумму квадратов непосредственным прибавлением слагаемых. Формула может использоваться для проверки результатов вычислений. Так, например, $1^2 + 2^2 + \ldots + 10^2 = \frac{10 \cdot 11 \cdot 21}{6} = 5 \cdot 11 \cdot 7 = 385$, а $1^2 + 2^2 + \ldots + 100^2 = \frac{100 \cdot 101 \cdot 201}{6} = 50 \cdot 101 \cdot 67 = 338350$.

Переменная k с начальным единичным значением играет роль индекса. Она используется в операторе цикла для подсчета количества слагаемых. Сумму квадратов натуральных чисел мы будем записывать в переменную s. Начальное значение этой переменной равно нулю.

НА ЗАМЕТКУ

Алгоритм вычисления суммы простой: к значению суммы (переменная s) последовательно добавляются слагаемые (квадраты натуральных чисел). Нам нужно каким-то образом подсчитывать, сколько слагаемых уже добавлено в сумму. С этой целью используется переменная k. После каждого цикла ее значение увеличивается на единицу. Такие переменные (используемые для подсчета циклов, слагаемых, индексов и прочее) мы будем называть *индексными*. Продолжаются вычисления до тех пор, пока значение переменной k не превысит значение верхней границы суммы n.

Для вывода результата нам понадобится сформировать текстовое значение. Записывается соответствующий текст в переменную txt. Ее начальное значение равно "1² + 2² + ... + ". Здесь использованы дескрипторы ^и, предназначенные для отображения верхних индексов. Нам данные дескрипторы нужны, чтобы отобразить операцию возведения натуральных чисел в квадрат (возведение в степень 2).

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Для создания верхних индексов используют дескрипторы ^и. Нижние индексы создаются с помощью дескрипторов _и. Верхний индекс отображается относительно верхней линии строки шрифтом уменьшенного размера. Нижний индекс отображается относительно нижней линии строки шрифтом уменьшенного размера. Верхние и нижние индексы обычно используются при отображении несложных математических или химических формул.

Затем к текущему значению, записанному в переменную txt, добавляется строка n+"² = ". Здесь использовано значение переменной n, которая «помнит» последнее число, входящее в сумму.

Далее начинается непосредственно вычисление суммы квадратов натуральных чисел. Для реализации задачи запускается оператор цикла. После ключевого слова while указано условие k<=n. Условие означает, что процесс выполнения циклов продолжается до тех пор, пока значение переменной k не превышает значение переменной n. В теле оператора цикла последовательно выполняются две команды. Командой s+=k*k к сумме прибавляется новое слагаемое, а командой k++ значение индексной переменной увеличивается на единицу. По завершении оператора цикла в переменную s будет записана сумма квадратов натуральных чисел от 1 до n включительно. Результат отображается командой document.write(txt+s).

Для включения сценария в веб-документ используем такой HTML-код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 2.4</title>
</head>
<body><h3>Листинг 2.4</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing02_04.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 2.11 показан результат отображения в браузере веб-документа со сценарием, в котором вычисляется сумма квадратов натуральных чисел.

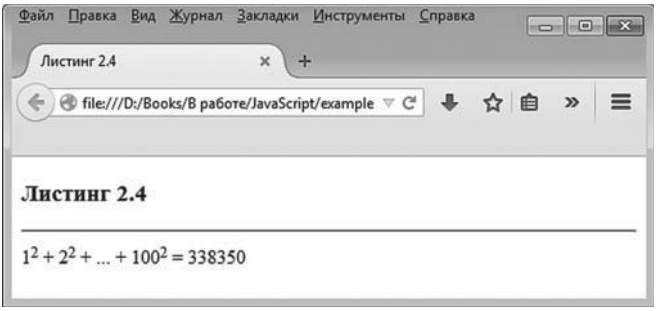

Рис. 2.11. Результат вычисления суммы квадратов натуральных чисел сценарием с оператором цикла *while*

Несложно заметить, что результат вычислен корректно (желающие могут воспользоваться для проверки готовой формулой для суммы квадратов натуральных чисел, которая приводилась ранее).

Оператор цикла do-while

> Ну, барин, ты задачи ставишь! За десять дён одному не справиться, тут помощник нужен — хомо сапиенс!
>
> *из к/ф «Формула любви»*

Очень похож на оператор цикла while другой оператор цикла, который называется do-while (во всяком случае мы его так будем называть). Настолько похож, что иногда об этих операторах говорят как о разных модификациях одного оператора.

Принципиальное отличие оператора do-while от оператора while состоит в том, что если в операторе while сначала проверяется условие, а затем выполняются команды в теле оператора, то в операторе do-while сначала выполняются команды в теле оператора и только после этого проверяется условие.

Синтаксис оператора

Описание оператора цикла do-while начинается с ключевого слова do, после которого в фигурных скобках размещаются команды, предна-

значенные для выполнения в каждом цикле. После закрывающей фигурной скобки следует ключевое слово while и в круглых скобках указывается *условие*. Ниже приведен шаблон описания оператора цикла do-while (жирным шрифтом выделены ключевые элементы шаблона):

```
do{
  // команды
}while(условие)
```

Условие проверяется после завершения выполнения команд в фигурных скобках (тело оператора цикла). Для продолжения выполнения оператора цикла необходимо, чтобы условие было истинным. Если при проверке условия оказывается, что оно ложно, выполнение оператора цикла прекращается. Принципы выполнения оператора цикла do-while иллюстрируются блок-схемой, представленной на рис. 2.12.

Рис. 2.12. Схема выполнения оператора цикла *do-while*

НА ЗАМЕТКУ

Команды в теле оператора цикла do-while выполняются по крайней мере один раз. Другими словами, если мы используем оператор цикла do-while, то те команды, которые мы помещаем в тело оператора, хотя бы один раз, но будут выполнены. Причина в том, что выполнение оператора do-while начинается с выполнения команд и только после первого цикла проверяется условие. Даже если при первой проверке условия окажется, что оно ложно, и выполнение оператора цикла прекратится, команды в теле оператора уже один раз выполнены.

Для сравнения: в операторе цикла while сначала проверяется условие. Если при первой проверке окажется, что условие ложно, команды в теле оператора цикла while выполняться не будут.

Познания относительно очередного оператора традиционно закрепляем с помощью примера.

Пример с использованием оператора цикла

Мы поступим мудро и просто — решим задачу о вычислении суммы квадратов натуральных чисел, но только теперь вместо оператора цикла while используем оператор цикла do-while. Программный код сценария приведен в листинге 2.5.

Листинг 2.5. Вычисление суммы квадратов чисел с помощью оператора цикла do-while (файл Listing02_05.js)

```
var n=100,k=1,s=0
var txt="1<sup>2</sup> + 2<sup>2</sup> + ... + "
txt+=n+"<sup>2</sup> = "
do{
  s+=k*k
  k++
}while(k<=n)
document.write(txt+s)
```

По сравнению с предыдущим примером изменения минимальные: для удобства восприятия комментарии удалены, а переменные с числовыми значениями объявляются в одном блоке. Назначение переменных такое же. Разумеется, изменился оператор цикла — теперь это do-while. Но команды в общем-то те же. И результат точно такой же. Чтобы убедиться в неизменности результата, используем следующий HTML-код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 2.5</title>
</head>
<body><h3>Листинг 2.5</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing02_05.js">
</script>
```

```
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 2.13 показано, как выглядит веб-документ со сценарием в окне браузера.

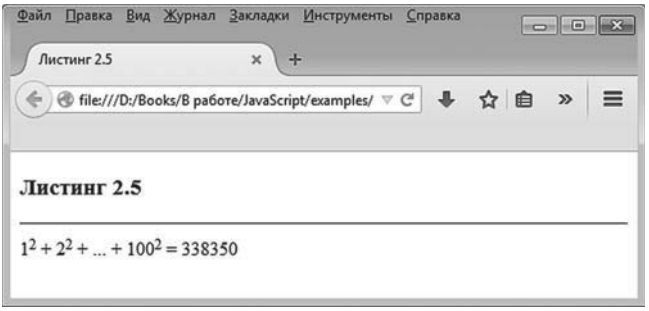

Рис. 2.13. В документе отображается результат выполнения сценария с оператором *do-while*

ДЕТАЛИ

Идея об «одинаковости» операторов цикла while и do-while — иллюзия, причем довольно опасная. Скажем, в рассмотренных выше примерах различий (с точки зрения конечного результата) в использовании операторов while и do-while не наблюдалось. Но это при значении переменной n, равном 100. Не будет различия в результатах выполнения сценариев и при единичном значении этой переменной: в обоих случаях получим сумму квадратов, равную единице (что вполне логично). Но вот если переменной n присвоить в сценарии значение 0 (или отрицательное число), результат выполнения сценария с оператором цикла while будет отличаться от результата выполнения сценария с оператором do-while. Первый из них выдаст нулевое значение для суммы квадратов, а второй — единичное. Причина в особенностях операторов.

В операторе while сначала проверяется условие k<=n, которое при значении 1 для переменной k и значении 0 для переменной n не выполняется. Как следствие, не выполняются команды в теле оператора цикла. Поэтому в документе выводится текущее значение переменной s, которое равно 0.

В операторе do-while выполняются команды s+=k*k и k++. После их выполнения (учитывая начальные значения 1 для k и 0 для s) переменная s имеет значение 1, а переменная k имеет значение 2. Только

после этого проверяется условие k<=n, которое очевидно равно false. Оператор цикла прекращает выполнение. В документе выводится значение переменной s, равное 1.

Операторы цикла while и do-while просты и удобны в использовании. Однако ими не исчерпываются механизмы реализации циклических вычислений в JavaScript.

Оператор цикла for

> Хотелось бы, так сказать, в общих чертах понять, что ему нужно.
>
> *из к/ф «Иван Васильевич меняет профессию»*

Оператор цикла for на фоне своих собратьев выглядит истинным аристократом. У него не самый тривиальный синтаксис, а о его «гибкости» можно слагать легенды. Мы постараемся уместить наиболее важную информацию об этом операторе в один раздел.

Синтаксис оператора

Шаблон описания оператора цикла for представлен ниже (жирным шрифтом выделены основные синтаксические элементы шаблона):

```
for(первый_блок;второй_блок;третий_блок){
  // команды
}
```

В общем и целом конструкция такая: ключевое слово for, после которого следуют круглые скобки с тремя блоками инструкций. Блоки разделяются между собой точкой с запятой (если в блоке несколько команд, то они разделяются запятыми). После круглых скобок указываются фигурные скобки с блоком команд (тело оператора цикла).

> **НА ЗАМЕТКУ**
>
> Если блок команд в теле оператора цикла for состоит из одной команды, фигурные скобки можно не использовать. Но «можно» не означает «нужно». Здесь как раз тот случай, когда лишними скобки не будут.

Общая принципиальная схема выполнения оператора цикла for проиллюстрирована на рис. 2.14.

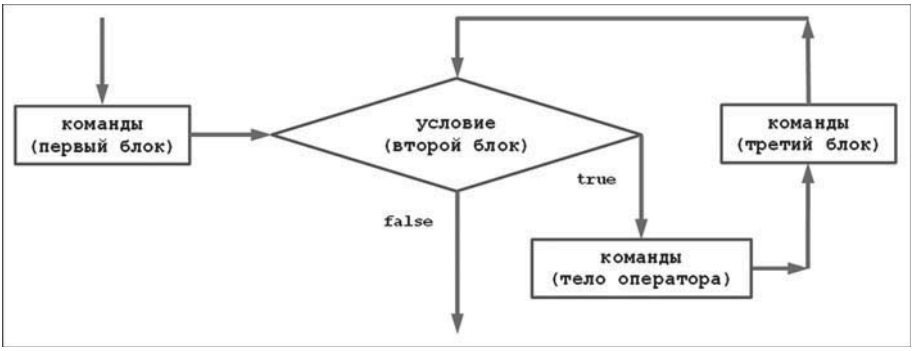

Рис. 2.14. Общая схема выполнения оператора цикла *for*

Начинается выполнение оператора цикла for с выполнения команд в первом блоке. Затем проверяется условие, указанное во втором блоке оператора цикла for. Если условие истинно, последовательно выполняются команды в теле оператора цикла (команды в фигурных скобках), а после них выполняются команды в третьем блоке оператора for.

По завершении выполнения команд третьего блока проверяется условие (второй блок). Если условие истинно, выполняются команды в фигурных скобках и в третьем блоке. Снова проверяется условие, и так далее. Выполнение оператора цикла for прекращается, когда при проверке условия оно оказывается ложным.

НА ЗАМЕТКУ

Команды в первом блоке выполняются один и только один раз в самом начале выполнения оператора цикла for.

Некоторые (или даже все) блоки могут быть пустыми. Пустой второй блок эквивалентен условию true. Формально это означает бесконечный цикл, но на практике в теле оператора цикла можно разместить инструкцию break, которая приведет к завершению выполнения оператора цикла. Причем это касается не только оператора цикла for, но и операторов while и do-while.

Для начала рассмотрим небольшой пример применения оператора цикла for на практике.

Пример использования оператора

В очередной раз реорганизуем код сценария с вычислением суммы квадратов натуральных чисел. На этот раз основной движущей силой станет оператор цикла for. Рассмотрим программный код, представленный в листинге 2.6.

Листинг 2.6. Вычисление суммы квадратов чисел с помощью оператора цикла for (файл Listing02_06.js)

```
var n=100,s=0,k
var txt="1<sup>2</sup> + 2<sup>2</sup> + … + "
txt+=n+"<sup>2</sup> = "
// Оператор цикла:
for(k=1;k<=n;k++){
  s+=k*k
}
document.write(txt+s)
```

Как и ранее, объявляем несколько переменных. Переменная txt содержит текст для отображения в документе. Переменная n содержит значение верхней границы суммы, переменная s с нулевым начальным значением предназначена для записи суммы квадратов чисел, а переменная k используется для подсчета циклов (или числа слагаемых). Однако теперь мы переменную k только объявляем, а начальное значение ей присваивается в операторе цикла for, в первом блоке (команда k=1). Эта команда выполняется в самом начале работы оператора цикла. После выполнения команды проверяется условие k<=n. При значении 100 для переменной n условие истинно. Поэтому выполняется команда s+=k*k в теле оператора цикла и после нее команда k++ в третьем блоке оператора. Далее снова проверяется условие k<=n и так далее. Процесс завершается, когда условие k<=n при проверке дает значение false. Это происходит, когда при выполнении команды k++ в третьем боке значение переменной k становится на единицу больше значения переменной n. До этого к значению переменной s последовательно были добавлены квадраты всех чисел от 1 до n включительно. Таким образом, в переменную s на момент завершения оператора цикла for записана сумма квадратов натуральных чисел от 1 до n включительно, что нам, собственно, и нужно. Результат вычислений отображается в рабочем окне командой document.write(txt+s).

Чтобы проверить корректность выполнения программного кода сценария, используем стандартный шаблон для веб-документа:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 2.6</title>
</head>
<body><h3>Листинг 2.6</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing02_06.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Результат вычислений такой же, как и в предыдущих случаях. Как подтверждение приведен рис. 2.15, на котором показан в окне браузера веб-документ с использованным сценарием из листинга 2.6.

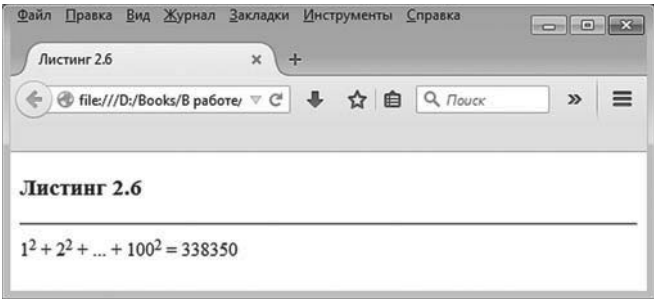

Рис. 2.15. Результат выполнения сценария с оператором цикла *for*

Важно отметить, что приведенный выше способ реализации оператора цикла for далеко не единственный. Чтобы проиллюстрировать заявленную ранее «гибкость» данного оператора цикла, рассмотрим еще несколько версий сценариев, которые можно было бы использовать вместо сценария из листинга 2.6.

НА ЗАМЕТКУ

Чтобы сохранить время, место и усилия, мы проанализируем только непосредственно программные коды со сценарием. Шаблонный HTML-код и результат тестирования сценариев (в виде изображений соответствующих веб-документов) приводить не будем. И шаблонные коды, и внешний вид соответствующих веб-документов совершенно однотипны. С точностью до несущественных деталей они совпадают с тем, что мы уже наблюдали ранее. Если читатель захочет самостоятельно протестировать рассматриваемые далее сценарии, ему достаточно будет воспользоваться приведенным выше шаблонным HTML-кодом, изменив в нем лишь название файла с загружаемым сценарием.

Итак, рассматриваемые далее сценарии предназначены для вычисления суммы квадратов натуральных чисел. Главное их отличие друг от друга и рассмотренного выше сценария состоит в способе использования оператора цикла for. Данное обстоятельство послужит нам оправданием того, что обсуждаться будет в основном оператор цикла.

Для начала рассмотрим сценарий, представленный в листинге 2.7.

Листинг 2.7. Оператор цикла for с объявлением переменных в первом блоке (файл Listing02_07.js)

```
// Переменные объявляются в первом блоке:
for(var n=100,s=0,k=1;k<=n;k++){
  s+=k*k
}
var txt="1<sup>2</sup> + 2<sup>2</sup> + ... + "
txt+=n+"<sup>2</sup> = "
document.write(txt+s)
```

Здесь, в данном сценарии, объявление переменных n, s и k, а также присваивание им значений вынесено в первый блок оператора цикла. Поскольку при определении значения переменной txt используется значение переменной n, объявление переменной txt и вычисление ее значения выполняются после окончания оператора цикла.

Следующая версия сценария, вычисляющего сумму квадратов натуральных чисел, представлена в листинге 2.8.

Листинг 2.8. Оператор цикла for с двумя пустыми блоками (файл Listing02_08.js)

```
var n=100,s=0,k=1
var txt="1<sup>2</sup> + 2<sup>2</sup> + … + "
txt+=n+"<sup>2</sup> = "
// Первый и третий блоки пустые:
for(;k<=n;){
  s+=k*k
  k++
}
document.write(txt+s)
```

В данном сценарии в операторе цикла первый и третий блоки пустые. Все переменные, в том числе и индексная переменная k, объявляются и получают начальное значение вне оператора цикла. Поэтому нет необходимости добавлять команды в первый блок. Третий блок тоже пустой. Команда k++, которая ранее присутствовала в третьем блоке, теперь вынесена в основное тело оператора цикла.

Чтобы понять, почему рассматриваемый код выполняется фактически так же, как и рассмотренные ранее сценарии, достаточно заметить, что последовательность выполнения команд и проверки условия в операторе цикла от вынесения команды из третьего блока в основное тело оператора не изменилась.

Можно поступить и иначе — вместо того чтобы выносить команду из третьего блока в основное тело оператора цикла, можем переместить команду из тела оператора в третий блок. Именно такой подход реализован в сценарии, представленном в листинге 2.9.

Листинг 2.9. Оператор цикла for с пустым телом оператора (файл Listing02_09.js)

```
var n=100,s=0,k=1
var txt="1<sup>2</sup> + 2<sup>2</sup> + … + "
txt+=n+"<sup>2</sup> = "
// Первый блок и тело оператора пустые:
for(;k<=n;s+=k*k,k++);
document.write(txt+s)
```

Здесь команда s+=k*k перекочевала из основного тела оператора в его третий блок — в нем теперь две команды, которые разделяются запятыми.

НА ЗАМЕТКУ

Как в этом, так и в предыдущем примере порядок размещения команд имеет значение.

Стоит заметить, что поскольку тело оператора цикла пустое, то там как бы нечего записывать. Поэтому после круглой скобки, завершающей третий блок, стоит точка с запятой. Если ее не поставить, то следующая команда document.write(txt+s) будет интерпретироваться как относящаяся к оператору цикла. Причина в том, что в операторе цикла for можно не использовать фигурные скобки, если тело оператора состоит из одной команды. Как следствие, если фигурных скобок нет, то первая после for-инструкции команда автоматически относится к телу оператора цикла. Вывод простой: точка с запятой — важный элемент синтаксической конструкции, и пренебрегать ею нельзя.

Достаточно экзотическая ситуация представлена в листинге 2.10. Там все три блока в операторе цикла пустые.

Листинг 2.10. Оператор цикла for с тремя пустыми блоками (файл Listing02_10.js)

```
var n=100,s=0,k=1
var txt="1<sup>2</sup> + 2<sup>2</sup> + ... + "
txt+=n+"<sup>2</sup> = "
// Все три блока пустые:
for(;;){
  s+=k*k
  k++
  if(k>n){
    break
  }
}
document.write(txt+s)
```

Надо заметить, что пустой второй блок (блок, в котором размещается условие) эквивалентен истинности условия в операторе цикла. Формально в этом случае цикл бесконечный. Поэтому в самом теле оператора цикла необходимо предусмотреть возможность завершения оператора цикла. Мы используем условный оператор, в котором проверяется условие k>n. При истинности условия выполняется инструкция break. Выполнение инструкции break завершает работу оператора цикла.

ДЕТАЛИ

Во втором блоке мы использовали условие k<=n. Это условие продолжения работы оператора цикла. В последнем примере в условном операторе проверяется условие k>n. Это условие завершения работы оператора цикла. Истинность условия k<=n означает ложность условия k>n, и наоборот.

Оператор выбора switch

— Вопрошайте.
— Готовы ли вы сказать нам всю правду?
— Ну, всю — не всю... А что вас интересует?

из к/ф «Формула любви»

Достаточно полезным является оператор выбора switch. В некотором смысле оператор switch напоминает конструкцию из вложенных условных операторов (хотя отождествлять их нельзя).

Синтаксис оператора выбора

Синтаксис оператора иллюстрирует следующий шаблонный код (жирным шрифтом выделены ключевые элементы шаблона):

```
switch(условие){
case значение_1:
  // команды 1-го блока
  break
case значение_2:
  // команды 2-го блока
  break
// прочие case-блоки
```

```
case значение_N:
  // команды N-го блока
  break
default:
  // команды блока по умолчанию
}
```

Начинается описание оператора выбора с ключевого слова switch. В круглых скобках после ключевого слова switch указывается некоторое *выражение*. Далее следуют case-блоки. Каждый такой блок начинается ключевым словом case, после которого указывается некоторое значение (контрольное значение) и двоеточие. Затем следуют команды данного case-блока. Последней командой обычно (но не всегда) является инструкция break. Последний блок помечается ключевым словом default, после которого стоит двоеточие. Это блок команд, выполняемых по умолчанию. Инструкцию break в конце default-блока не ставят (в ней просто нет смысла). Вся конструкция из case-блоков и default-блока (который, кстати, необязательный) заключается в фигурные скобки.

При выполнении оператора выбора значение данного выражения вычисляется и последовательно сравнивается со значениями, указанными в case-блоках (после ключевого слова case). Если найдено совпадение (имеется в виду совпадение значения выражения и контрольного значения в case-блоке), начинается выполнение команд данного case-блока. Если ни в одном из case-блоков контрольное значение не совпадает со значением выражения, выполняются команды в default-блоке.

НА ЗАМЕТКУ

Блок с командами, выполняемыми по умолчанию (default-блок), не является обязательным. Если такого блока в операторе выбора нет и поиск совпадения контрольного значения и значения выражения не увенчался успехом, то выполнение оператора выбора завершится.

Как отмечалось выше, обычно последней в case-блоке является инструкция break. Выполнение данной инструкции приводит к завершению работы оператора выбора. Дело в том, что если инструкцию break в конце блока не разместить, то после выполнения команд данного блока автоматически начнут выполняться команды следующего блока. Если в нем нет break-инструкции, будут выполняться команды еще

одного блока, и так далее до конца тела оператора. Если во всем этом потребности нет, используют инструкцию break.

Общая схема выполнения оператора выбора проиллюстрирована на рис. 2.16 (штрихованными линиями со стрелками показано «направление» выполнения кода при отсутствии инструкции break в case-блоке).

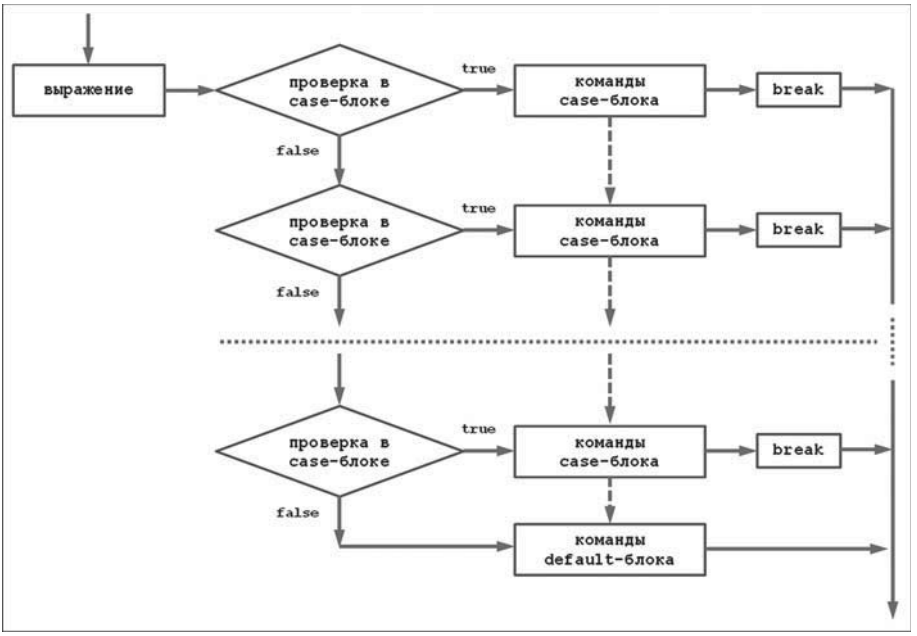

Рис. 2.16. Схема выполнения оператора выбора *switch*

Составить общее представление об операторе выбора нам поможет небольшой пример.

Примеры использования оператора выбора

Мы рассмотрим два примера, в которых используется оператор выбора switch. Оба они достаточно простые. В первом примере, программный код которого представлен в листинге 2.11, описывается функция, основу которой составляет оператор выбора switch. У функции три аргумента. Предполагается, что первые два аргумента являются числовыми, а третий — текстовый. Текстовый аргумент определяет арифметическую операцию (вычисление суммы, разности, произведения или частного), которую необходимо выполнить с первыми двумя аргументами функции. Результат отображается в документе. Для

«идентификации» типа операции по текстовому значению третьего аргумента функции и используется оператор switch. Теперь проанализируем программный код.

Листинг 2.11. Оператор выбора switch (файл Listing02_11.js)

```
// Функция с оператором выбора:
function show(x,y,op){
  // Локальная переменная:
  var msg
  // Проверяется значение аргумента:
  switch(op){
    case "сумма":       // Первый блок
      msg=x+" + "+ y+" = "+(x+y)+"<br>"
      break
    case "разность":    // Второй блок
      msg=x+" - "+ y+" = "+(x-y)+"<br>"
      break
    case "произведение": // Третий блок
      msg=x+" * "+ y+" = "+(x*y)+"<br>"
      break
    case "частное":     // Четвертый блок
      msg=x+" / "+ y+" = "+(x/y)+"<br>"
      break
    default:            // По умолчанию
      msg="<b>"+op+"</b> - неизвестная операция<br>"
  }
  // Отображение текста в документе:
  document.write(msg)
} // Окончание описания функции
// Вызов функции с разными аргументами:
show(8,4,"произведение")
show(8,4,"сумма")
show(8,4,"частное")
show(8,4,"разность")
show(8,4,"подмигивание")
```

Функция называется show(). Ее первые два аргумента обозначены как x и y. Это операнды некоторого выражения. Нужно только определить оператор. Оператор определяется текстовым значением третьего аргумента op функции show(). Другими словами, в функции по значению операнда op нужно выяснить, какую операцию следует выполнить с аргументами x и y. И здесь нам поможет оператор выбора switch.

В операторе switch проверяется значение выражения op — то есть значение третьего аргумента функции show(). В case-блоках представлены контрольные значения «сумма», «разность», «произведение» и «частное». Также есть default-блок на случай, если значение аргумента op не совпадет ни с одним из контрольных значений в case-блоках. В каждом блоке в соответствии с типом выполняемой операции присваивается значение переменной msg. После завершения выполнения оператора выбора командой document.write(msg) значение переменной msg отображается в рабочем документе.

ДЕТАЛИ

Переменная msg объявлена в теле функции. Это *локальная* переменная. Она доступна только в теле функции. За пределами программного кода функции show() об этой переменной ничего не известно.

Значение переменной msg формируется исходя из выполняемой арифметической операции. Более конкретно, в переменную записывается символьное выражение для операции и ее результат. Например, командой msg=x+" + "+ y+" = "+(x+y)+"
" в первом case-блоке в переменную msg записывается текстовая строка, которая получается объединением значения аргумента x, оператора сложения " + ", значения аргумента y, знака равенства " = ", результата вычисления суммы операндов (x+y) и инструкции перехода к новой строке "
". Скобки при вычислении суммы операндов нужны для того, чтобы выполнялось именно сложение чисел, а не конкатенация их текстовых представлений.

Конечно, описанную выше процедуру можно было реализовать более эффектно, воспользовавшись, например, функцией eval(), которая, напомним, позволяет вычислять значение выражений, «спрятанных» в текстовой строке. Но в рассматриваемом примере это не столь принципиально.

Также стоит заметить, что функция show() не возвращает результат. Она просто выполняет некоторые действия. Подробнее функции обсуждаются в следующей главе.

После того как функция show() описана в сценарии, она несколько раз вызывается с разными аргументами (фактически меняется значение

только третьего аргумента функции). Чтобы протестировать сценарий, используем веб-документ со следующим HTML-кодом:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 2.11</title>
</head>
<body><h3>Листинг 2.11</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing02_11.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 2.17 показано, как будет выглядеть данный документ в окне браузера.

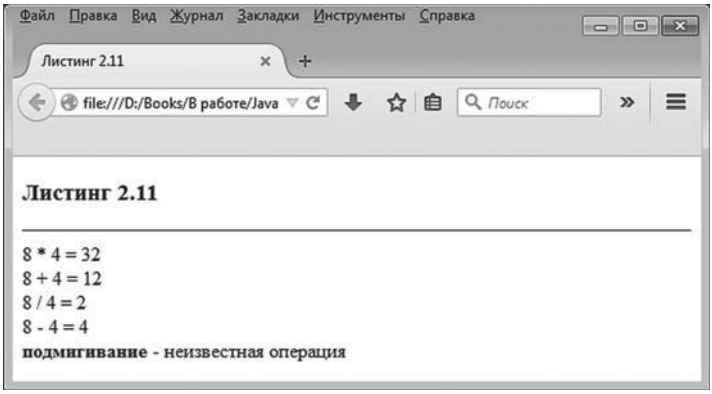

Рис. 2.17. Документ со сценарием, содержащим оператор выбора *switch*

В следующем примере также используется оператор выбора. Но на сей раз некоторые case-блоки оператора пустые. Чтобы понять причину, сначала сформулируем суть задачи, которую мы пытаемся решить.

Создается сценарий, в котором генерируется последовательность целых случайных чисел в диапазоне значений от 6 до 15 включительно. После того как очередное число сгенерировано, оно проверяется на предмет того, *простое* это число или нет, является ли оно *совершенным* и делится ли оно на пять.

ДЕТАЛИ

Простыми называются числа, которые не имеют никаких иных делителей, кроме единицы и самого себя. В диапазоне чисел от 6 до 15 простыми являются числа 7, 11 и 13.

Совершенным называется число, сумма делителей которого (не учитывая самого числа) равняется данному числу. Так, например, число 6 является совершенным, поскольку оно делится на числа 1, 2 и 3, сумма которых равняется 6. Других совершенных чисел в диапазоне значений от 6 до 15 нет (ближайшее совершенное число 28).

Наконец, в диапазоне значений от 6 до 15 на пять без остатка делятся только числа 10 и 15.

В зависимости от того, попадает ли сгенерированное число в одну из указанных категорий, в рабочем документе выводится поясняющее сообщение (с указанием числа и комментарием относительно его свойств). Сортировка чисел реализуется с помощью оператора выбора switch. Проверяемым выражением является значение случайного числа. Но пикантность ситуации в том, что для нескольких разных значений числа нужно выполнять одни и те же действия. Например, числа 7, 11 и 13 относятся к простым. Тогда соответствующие таким контрольным значениям case-блоки должны были бы содержать одинаковые команды. Вместо создания нескольких формально одинаковых (за исключением контрольного значения) case-блоков используем пустые case-блоки. Как именно реализуется данный подход, показано в сценарии в листинге 2.12.

Листинг 2.12. Оператор выбора switch с пустыми case-блоками (файл Listing02_12.js)

```
document.write("<h4>Случайные числа</h4>")
var rnd,msg
// Оператор цикла, в котором генерируются целые
// случайные числа в диапазоне от 6 до 15 включительно:
for(var k=1,n=20;k<=n;k++){
```

```
  // Генерирование случайного числа:
  rnd=6+Math.floor(10*Math.random())
  msg="<b>"+rnd+"</b> - "
  // Проверяется значение случайного числа:
  switch(rnd){
    // Совершенное число:
    case 6:
      msg+="совершенное число || "
      break
    // Простые числа:
    case 7:
    case 11:
    case 13:
      msg+="простое число || "
      break
    // Числа, которые делятся на пять:
    case 10:
    case 15:
      msg+="делится на пять || "
      break
    // Все прочие числа:
    default:
      msg+="самое обычное число || "
  }
  // Отображение информации о числе:
  document.write(msg)
}
```

Начинается сценарий командой `document.write("<h4>Случайные числа</h4>")`, которой в рабочем документе отображается заголовок четвертого уровня. В сценарии объявляются переменные `rnd` и `msg`. В переменную `rnd` будут записываться случайные числа, а в переменную `msg` записывается текст для отображения в рабочем документе.

Для генерирования случайных чисел запускается оператор цикла, в котором объявлены (в первом блоке) индексная переменная `k` с на-

чальным значением 1 и переменная n со значением 20 (количество генерируемых случайных чисел). Оператор цикла выполняется, пока истинно условие k<=n и за каждый цикл командой k++ в третьем блоке значение индексной переменной k увеличивается на единицу.

В теле оператора цикла командой rnd=6+Math.floor(10*Math.random()) генерируется и записывается в переменную rnd случайное число (диапазон значений от 6 до 15). Здесь мы использовали два метода встроенного объекта Math. Методом random() генерируется случайное действительное число в диапазоне значений от 0 до 1. Методом floor() выполняется округление действительного числа до целого (для положительных чисел все сводится к отбрасыванию дробной части числа).

ДЕТАЛИ

Метод random() возвращает случайное число, строго меньшее 1. То есть число 1 сгенерировано быть не может (но может быть сгенерировано число 0). Если результат, возвращаемый методом random(), умножить на 10 (как это сделано в команде rnd=6+Math.floor(10*Math.random())), то получим действительное число в диапазоне значений от 0 до 10 (но не 10). Если отбросить дробную часть (для чего используется метод floor()), получим целое число от 0 до 9 включительно. Если теперь прибавить 6, получим целое число от 6 до 15 (включительно).

Начальное значение переменной msg определяется командой msg=""+rnd+" - ". Это текстовое представление сгенерированного числа, выделенное жирным шрифтом, с дефисом. Далее в игру вступает оператор выбора, в котором уточняется значение переменной msg. Проверяется значение сгенерированного случайного числа rnd. В первом case-блоке число тестируется на предмет совершенства. Блок заканчивается инструкцией break, так что если число совершенное, то выполняться будут только команды данного блока.

Для проверки простых чисел используется конструкция из трех ключевых слов case с разными контрольными значениями. И только после третьей case-инструкции размещены команды для выполнения. Фактически две первые case-инструкции представляют собой пустые блоки. Но, несмотря на то что они пустые, данные блоки все равно проверяются. И если имеет место совпадение значения тестируемого выражения и контрольного значения, начинают выполняться команды вплоть до первой инструкции break. Это означает, что для всех трех

контрольных значений 7, 11 и 13 выполняется один и тот же набор команд. Аналогичная ситуация имеет место для case-блоков с контрольными значениями 10 и 15.

После того как значение переменной msg сформировано, информация о числе отображается командой document.write(msg). Напомним, что все это происходит в рамках оператора цикла — то есть описанные выше действия выполняются для каждого сгенерированного случайного числа.

Ниже представлен HTML-код, который используется для включения сценария в веб-документ:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 2.12</title>
</head>
<body><h3>Листинг 2.12</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing02_12.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 2.18 показано окно браузера с открытым в нем документом, содержащим рассмотренный выше сценарий.

Поскольку речь идет о случайных числах, то при перезагрузке страницы содержимое документа изменится.

НА ЗАМЕТКУ

При отображении текста в качестве разделителя между текстовыми блоками использовалась двойная вертикальная черта. Никакой особой функциональной нагрузки она не несет. В данном случае это просто декоративный элемент.

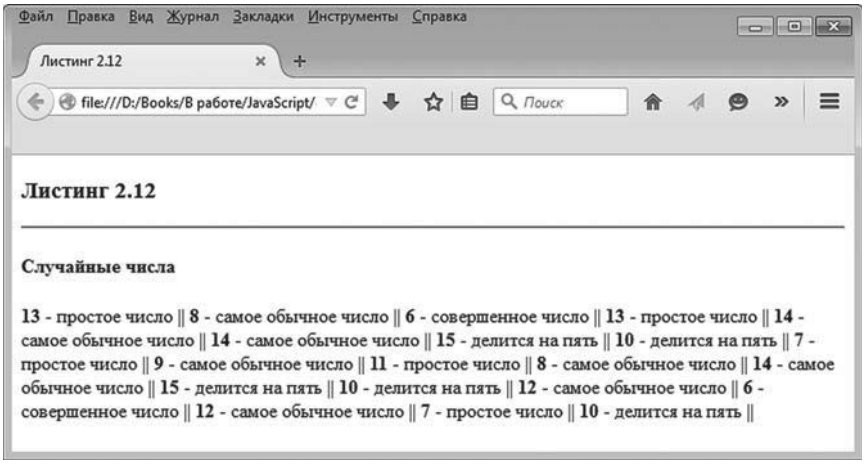

Рис. 2.18. Документ со сценарием, содержащим оператор выбора *switch* с пустыми *case*-блоками

Резюме

Искусство по-прежнему в большом долгу.

из к/ф «Покровские ворота»

Таким образом, мы познакомились с основными управляющими инструкциями языка JavaScript.

- Условный оператор if позволяет выполнять разные блоки команд в зависимости от истинности или ложности некоторого условия. Условие указывается после ключевого слова if в круглых скобках. Если условие истинно, выполняется блок команд в фигурных скобках. Если условие ложно, выполняется блок команд в фигурных скобках после ключевого слова else.
- Существует упрощенная форма условного оператора, не содержащая else-блок. В этом случае, если условие после ключевого слова if истинно, выполняется блок команд в фигурных скобках. Если условие ложно, не происходит ничего.
- Оператор цикла while позволяет многократно выполнять однотипные действия. После ключевого слова while в круглых скобках указывается условие. Оператор цикла выполняется, пока условие истинно. Команды, выполняемые в операторе цикла, заключаются в фигурные скобки. В операторе while сначала проверяется условие, а затем выполняются команды (весь блок).

- Принципиальное отличие оператора do-while от оператора while состоит в том, что в операторе do-while сначала выполняются команды, а затем проверяется условие. Команды, выполняемые в операторе цикла, заключаются в фигурные скобки и указываются после ключевого слова do. После блока команд указывается ключевое слово while и затем в круглых скобках — условие, истинность которого является необходимой для продолжения работы оператора цикла. Команды в теле оператора цикла do-while по крайней мере хотя бы раз обязательно выполняются.
- Оператор цикла for состоит из собственно ключевого слова for, круглых скобок с тремя блоками команд и тела оператора (блок команд в фигурных скобках). Блоки команд в круглых скобках после ключевого слова for разделяются точкой с запятой. Разные команды в пределах одного блока разделяются запятыми. Команды первого блока выполняются только один раз в самом начале работы оператора цикла. Затем проверяется условие во втором блоке. Если условие истинно, выполняются команды основного тела оператора цикла и команды в третьем блоке. Далее проверяется условие (второй блок). Если оно истинно, выполняются команды в фигурных скобках и в третьем блоке, опять проверяется условие и так далее. Работа оператора цикла оператора завершается, если при проверке условия оно окажется ложным.
- Некоторые блоки (или даже все) в операторе цикла for могут быть пустыми. Пустой второй блок (блок, где размещается условие) эквивалентен истинному условию.
- Оператор выбора switch используется для проверки значения некоторого выражения, которое указывается в круглых скобках после ключевого слова switch. Тело оператора заключается в фигурные скобки. Там размещаются case-блоки с контрольными значениями. Значение проверяемого выражения сравнивается с контрольными значениями в case-блоках. Если совпадение имеет место, выполняются команды, начиная с соответствующего case-блока и вплоть до окончания тела оператора выбора или до первой break-инструкции (поэтому обычно команды в case-блоке заканчиваются инструкцией break). В операторе выбора может быть default-блок, команды которого выполняются, если при проверке выражения совпадений с контрольными значениями в case-блоках не было.

Глава 3
ФУНКЦИИ

Это в твоих руках все горит, а в его руках все работает!

из к/ф «Покровские ворота»

В предыдущей главе мы уже рассматривали несколько примеров, в которых использовались *функции*. Но если там знакомство было шапочное, то здесь, в этой главе, мы подойдем к вопросу более основательно. Полностью раскрыть тему использования функций всего в одной главе достаточно сложно, однако мы твердо намерены заложить надежный фундамент на будущее.

Знакомство с функциями

Я вся такая внезапная, такая противоречивая вся...

из к/ф «Покровские ворота»

Под функцией подразумевают некоторый блок кода (иногда его называют подпрограммой), у которого есть имя и который можно вызвать через данное имя. Концепция функций очень удобна, поскольку позволяет многократно использовать один и тот же код, не переписывая его. Это экономит время и усилия, да и вероятность ошибок снижается (хотя, с другой стороны, если в функции есть ошибка, то тиражируется она также в соответствии с количеством вызовов функции).

Помимо названия, у функции могут быть аргументы (или параметры), которые ей передаются при вызове. Также функция может возвращать результат (но может и не возвращать). Если функция возвращает результат, то инструкцию вызова функции (в зависимости от типа возвращаемого результата) используют во всевозможных выражениях. В этом случае инструкция вызова функции отождествляется с ее результатом. Если функция не возвращает результат, то

вызов функции означает выполнение некоторых действий — более конкретно, тех, что описаны в функции.

Для вызова функции указывается ее имя и, если необходимо, в круглых скобках — значения аргументов. Если у функции аргументов нет, после имени функции указываются пустые круглые скобки. Теперь рассмотрим вопрос о том, как описывается функция.

Описание функции

Описание функции — это по большому счету определение того программного кода, который должен выполняться при вызове функции. Данный код должен быть «правильно оформлен».

Описываться функция может в любом месте сценария, причем совсем необязательно до того, как она вызывается. Другими словами, мы можем сначала вызвать функцию, а только затем ее описать. Ошибки не будет, поскольку при выполнении сценария, если в нем встречается инструкция вызова функции, поиск описания этой функции выполняется по всему сценарию (включая загруженные внешние файлы). Если описание функции в нем есть, то оно будет использовано. Вот если его там нет — тогда другое дело.

Итак, мы практически не ограничены в месте описания функции в сценарии. Описание функции начинается с ключевого слова function, после которого следует название функции и в круглых скобках через запятую перечисляются аргументы функции (что касается названий аргументов, то их можно выбирать по своему усмотрению). Если у функции аргументов нет, то просто ставятся пустые круглые скобки. Команды, которые выполняются при вызове функции, указываются в блоке, ограниченном фигурными скобками. Весь шаблон описания функции выглядит следующим образом (жирным шрифтом выделены ключевые элементы шаблона):

```
function название_функции(аргументы){
  // команды
}
```

Описанную в соответствии с таким шаблоном функцию в программном коде вызываем командой вида

```
название_функции(аргументы)
```

Следует учесть одну важную особенность: описание функции не означает выполнения ее программного кода. Другими словами, из того, что мы описали в сценарии функцию, не следует, что она будет вызвана. Для вызова функции необходимо использовать отдельную команду. Описание функции — всего лишь план действий на случай, если функция будет вызываться. Данное замечание касается и аргументов функции.

Когда мы описываем функцию, аргументы обозначаются лишь формально, чтобы можно было в теле функции описать те операции, которые необходимо выполнить с аргументами. А вот когда мы вызываем функцию, то аргументами ей нужно передать реальные значения, с которыми в соответствии с кодом функции будут выполняться определенные манипуляции.

Если функция возвращает значение, то в теле функции используют инструкцию return, после которой указывается возвращаемое функцией значение. Выполнение инструкции return приводит к завершению выполнения функции. Если функция не возвращает результат, то в ней можно использовать инструкцию return, после которой никакое значение не указывается. Далее рассмотрим небольшие примеры объявления и вызова функций.

Примеры объявления функций

Небольшая иллюстрация к сказанному выше приведена в листинге 3.1. Там представлен сценарий, в котором описано несколько функций. Также в сценарии есть инструкции вызова описанных функций.

Листинг 3.1. Описание функций (файл Listing03_01.js)

```
makeHeader("Знакомимся с функциями")
// Функция для генерирования случайных чисел и записи
// и в текстовую строку:
function getRandText(n){
  var txt="Случайные целые числа (от 1 до 10):<br>* "
  for(var k=1;k<=n;k++){
    txt+=myRand()+" * "
  }
```

```
  txt+="<br>"
  return txt
}
// Отображение 20 случайных чисел:
document.write(getRandText(20))
// Функция для генерирования случайных целых чисел:
function myRand(){
  return 1+Math.floor(10*Math.random())
}
// Функция для отображения заголовка:
function makeHeader(t){
  document.write("<h4>"+t+"</h4>")
}
```

В данном сценарии описано несколько функций и всего две команды (не относящиеся непосредственно к описанию функций). Самая первая команда в сценарии — makeHeader("Знакомимся с функциями"), в которой вызывается функция makeHeader() с аргументом "Знакомимся с функциями". Сама функция описана в конце сценария.

> **НА ЗАМЕТКУ**
>
> Таким образом, инструкция вызова функции в сценарии размещена до описания функции. Такая ситуация не приводит к ошибке.

Функция makeHeader() описана с одним аргументом, который формально обозначен как t. В теле функции выполняется всего одна команда document.write("<h4>"+t+"</h4>"), которой в рабочий документ выводится значение аргумента t функции, но только заключенное в дескрипторы <h4> и </h4>, благодаря чему текст, записанный в аргумент t, отображается как заголовок четвертого уровня. Это все, что делает функция. Результат функция не возвращает. Поэтому при выполнении команды makeHeader("Знакомимся с функциями") переданный аргументом текст отображается как заголовок.

В сценарии описывается еще несколько функций. Сразу после команды с вызовом функции makeHeader() представлено описание функции getRandText(). У функции один аргумент n, который, как неявно

предполагается, является целым числом. Также функция возвращает результат, который представляет собой текстовую строку, содержащую, кроме непосредственно текста, еще и несколько случайных целых чисел. Количество целых чисел определяется аргументом функции.

В теле функции getRandText() объявляется локальная переменная txt с некоторым начальным значением. Затем запускается оператор цикла, в котором индексная переменная k пробегает значения от 1 до n, и за каждый цикл к значению переменной txt «дописывается» результат вызова функции myRand() (результатом функции является целое случайное число в диапазоне значений от 1 до 10) и еще «звездочка» в качестве разделителя. После завершения оператора цикла значение переменной txt возвращается результатом функции (инструкция return txt).

НА ЗАМЕТКУ

Переменная txt, объявленная в теле функции, является локальной. Ее область доступности ограничивается телом функции. Такая же область доступности и у аргументов функции.

В теле функции getRandText() для генерирования случайных чисел вызывается функция myRand(). У функции нет аргументов, а результатом функции, как отмечалось, является целое случайное число. В описании функции всего одна инструкция return 1+Math.floor(10*Math.random()), которой определяется возвращаемое значение.

Здесь мы традиционно воспользовались методами random() и floor() встроенного объекта Math, которые соответственно генерируют случайное действительное число (диапазон значений от 0 до 1) и округляют действительное число до целого путем отбрасывания дробной части (для положительных чисел).

ДЕТАЛИ

Результат выражения Math.random() — действительное число от 0 до 1. Результат выражения 10* Math.random() — число в диапазоне значений от 0 до 10. Результат выражения Math.floor(10* Math.random()) — целое число в диапазоне значений от 0 до 9. Соответственно, результат выражения 1+Math.floor(10* Math.random()) — целое число в диапазоне значений от 1 до 10.

Текст со случайными числами отображается в рабочем документе вследствие выполнения команды document.write(getRandText(20)), которая размещена в сценарии между описаниями функций getRandText() и myRand().

НА ЗАМЕТКУ

Команды и описания функций «разбросаны» по сценарию с одной целью — продемонстрировать, что место размещения описания функций не принципиально с точки зрения корректности кода. Вместе с тем для лучшего восприятия кода рекомендуется продумывать структуру сценария, чтобы его было легче анализировать и находить места, в которые при необходимости следует внести исправления.

Для проверки корректности выполнения сценария используем веб-документ с представленным ниже HTML-кодом:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 3.1</title>
</head>
<body><h3>Листинг 3.1</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing03_01.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Соответствующий документ, открытый в окне веб-браузера, представлен на рис. 3.1.

Поскольку в данном примере мы имеем дело со случайными числами, то при перезагрузке страницы в окне браузера числа, отображаемые в документе, изменятся.

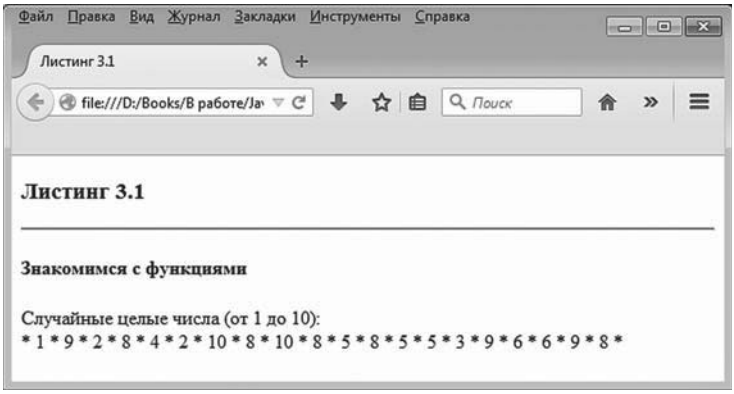

Рис. 3.1. Результат отображения в окне браузера документа со сценарием, в котором используются функции

Локальные и глобальные переменные

Он не зритель, он ваш сосед.

из к/ф «Покровские ворота»

И в этой, и в предыдущей главе мы упоминали *локальные переменные*. Контекст был такой, что локальная переменная доступна только в теле функции, а за пределами функции она недоступна. Рассмотрим данный вопрос подробнее.

Область видимости и локальные переменные в функции

Начнем с того, что определимся с *областью видимости* переменных. Область видимости (или область доступности) переменной — та часть программного кода, в которой данная переменная может быть использована. Обычно переменные классифицируют как *глобальные* или *локальные*. Все зависит от области доступности переменной. Глобальная переменная доступна «везде». Локальная переменная доступна «где-то». Что мы понимаем под «везде» и «где-то»? Ответ в общем-то зависит от специфики языка программирования. Для JavaScript «везде» — это уровень окна, в котором открыт документ, использующий сценарий. Если используется один сценарий (а мы пока что используем в веб-документах не более одного сценария), то глобальная переменная доступна везде в сценарии. Переменные, которые мы использовали ранее и которые объявлялись вне тела функции, все были глобальными.

НА ЗАМЕТКУ

Вообще глобальную переменную можно объявить в одном сценарии, а использовать в другом сценарии. Главное, чтобы сценарии загружались в одном документе.

Если мы объявляем переменную в теле функции, то переменная будет локальной. Она, как отмечалось ранее, доступна только в теле функции.

НА ЗАМЕТКУ

Локальная переменная существует, только пока выполняется код функции. По завершении выполнения функции локальные переменные удаляются из памяти.

Более того, в языке JavaScript именно функции служат «вместилищем» локальных переменных. Именно функции, их структура и иерархия определяют области доступности переменных. Правда, следует учесть, что в JavaScript имеются внутренние функции (то есть функции, описанные внутри других функций). Так что вопрос со структурой областей видимости переменных не такой очевидный, как может показаться на первый взгляд.

ДЕТАЛИ

Для читателей, знакомых с такими языками программирования, как C++, C# и Java, может стать сюрпризом то обстоятельство, что фигурные скобки { и } в JavaScript не образуют область видимости. Например, если в Java объявить переменную в блоке, выделенном фигурными скобками { и }, такая переменная доступна только в пределах данного блока. В JavaScript *такое правило не действует*. Область видимости в JavaScript определяется границами функции.

Критически важно, чтобы при объявлении переменной в теле функции использовалась инструкция var. Только в этом случае переменная, объявленная в теле функции, будет локальной. При этом если переменная объявлена в сценарии вне тела какой бы то ни было функции, то такая переменная будет глобальной.

Ситуация может сложиться так, что в сценарии объявлена переменная с точно таким же именем, что и локальная переменная в теле фун-

кции. Базовое правило состоит в том, что *локальная переменная имеет приоритет*. Ситуацию иллюстрирует сценарий, представленный в листинге 3.2.

Листинг 3.2. Совпадающие названия локальной и глобальной переменных (файл Listing03_02.js)

```
// Глобальная переменная:
var myText="Глобальная переменная"
// Отображение значения глобальной переменной:
document.write(myText+"<br>")
// Описание функции:
function show(){
  // Локальная переменная:
  var myText
  myText="Локальная переменная"
  // Отображение значения локальной переменной:
  document.write(myText+"<br>")
}
// Вызов функции:
show()
// Отображение значения глобальной переменной:
document.write(myText+"<br>")
```

При выполнении данного сценария в веб-документе отображаются такие значения.

Результат выполнения сценария (из листинга 3.2)

```
Глобальная переменная
Локальная переменная
Глобальная переменная
```

Для включения сценария в веб-документ используем следующий HTML-код:

```
<!DOCTYPE HTML>
<html>
```

```
<head>
  <title>Листинг 3.2</title>
</head>
<body><h3>Листинг 3.2</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing03_02.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 3.2 показано окно браузера, где открыт документ со сценарием, в котором есть локальная и глобальная переменные с совпадающими названиями.

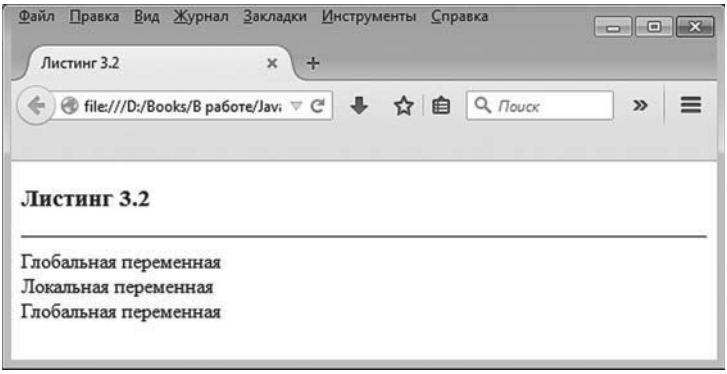

Рис. 3.2. Результат отображения документа со сценарием, в котором локальная и глобальная переменная имеют совпадающие названия

Разберем теперь программный код. Он достаточно простой. Сначала объявляется глобальная переменная myText со значением "Глобальная переменная" и значение этой переменной отображается в рабочем документе. Далее следует описание функции show(). В соответствии с описанием функции в ней объявляется локальная переменная myText, и ей присваивается значение "Локальная переменная". Значение переменной выводится в документе. Все это происходит при вызове функции командой show().

После вызова функции снова отображается значение переменной myText (на этот раз глобальной).

О чем свидетельствует результат выполнения сценария? Если мы обращаемся к переменной myText непосредственно в сценарии, то имеется в виду та переменная, что объявлена в сценарии (не в теле функции). Это глобальная переменная.

Если же обращение к переменной myText выполняется в теле функции show(), то в действие вступает локальная переменная. И это несмотря на то что у переменных одинаковые названия.

Обращение к глобальной переменной в функции

Возникает естественный вопрос: а как в теле функции обратиться к глобальной переменной? Ответ получим с помощью сценария, представленного в листинге 3.3.

Листинг 3.3. Обращение к глобальной переменной в теле функции (файл Listing03_03.js)

```
// Глобальная переменная:
var myText="Глобальная переменная"
// Отображение значения глобальной переменной:
document.write(myText+"<br>")
// Описание функции:
function show(){
  // Глобальная переменная:
  myText="Это не локальная переменная"
  // Отображение значения переменной:
  document.write(myText+"<br>")
}
// Вызов функции:
show()
// Отображение значения глобальной переменной:
document.write(myText+"<br>")
```

Результат выполнения сценария такой.

Результат выполнения сценария (из листинга 3.3)

```
Глобальная переменная
Это не локальная переменная
Это не локальная переменная
```

Чтобы увидеть результат выполнения сценария, используем веб-документ с таким кодом:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 3.3</title>
</head>
<body><h3>Листинг 3.3</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing03_03.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 3.3 представлен результат выполнения сценария.

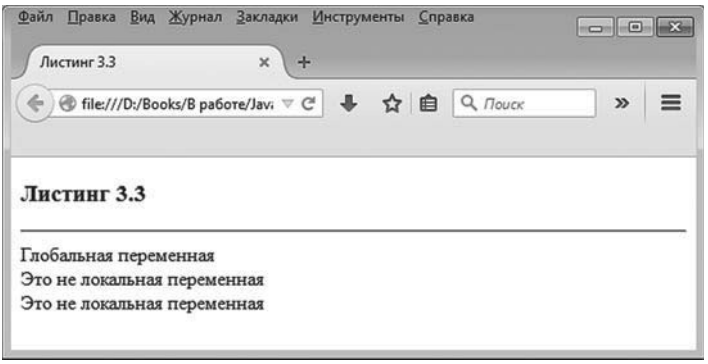

Рис. 3.3. Документ со сценарием с функцией, в которой выполняется обращение к глобальной переменной

Все, что произошло, по большому счету сводится к удалению инструкции объявления переменной myText в теле функции show(). Но последствия такого действия разительные: теперь, присваивая в теле функции значение переменной myText, мы присваиваем фактически значение той переменной, что была объявлена в сценарии. Таким образом, в теле функции show() в данном случае операции выполняются с глобальной переменной.

Создание глобальной переменной в функции

Еще более «лаконичный» случай представлен в сценарии в листинге 3.4. Там объявление глобальной переменной отсутствует вообще. Есть только команда присваивания переменной myText в теле функции show() (но переменная myText при этом в теле функции не объявляется).

Листинг 3.4. Глобальная переменная создается в теле функции (файл Listing03_04.js)

```
// Описание функции:
function show(){
  // Глобальная переменная:
  myText="Глобальная переменная"
  // Отображение значения переменной:
  document.write(myText+"<br>")
}
// Вызов функции:
show()
// Отображение значения глобальной переменной:
document.write(myText+"<br>")
```

Результат выполнения сценария представлен ниже.

Результат выполнения сценария (из листинга 3.4)

```
Глобальная переменная
Глобальная переменная
```

Несложно заметить, что, хотя глобальная переменная myText в сценарии не объявлялась, после вызова функции show() такая переменная

в сценарии есть. То есть при присваивании значения переменной myText в теле функции переменная не просто получает значение, а она создается.

> **НА ЗАМЕТКУ**
>
> Напомним, что код в теле функции выполняется только при вызове функции. Если функция не вызывается, то и ее код не выполняется.

Для включения сценария в веб-документ используем следующий HTML-код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 3.4</title>
</head>
<body><h3>Листинг 3.4</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing03_04.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Как будет выглядеть веб-документ со сценарием в окне браузера, показано на рис. 3.4.

Еще раз подчеркнем, что в рассмотренном примере глобальная переменная создается при вызове функции show(). Если функция вызывается несколько раз, то переменная создается при первом вызове функции.

> **НА ЗАМЕТКУ**
>
> Если бы мы в теле функции show() в команде присваивания значения переменной myText использовали ключевое слово var, то при вызове функции создавалась бы не глобальная, а локальная переменная.

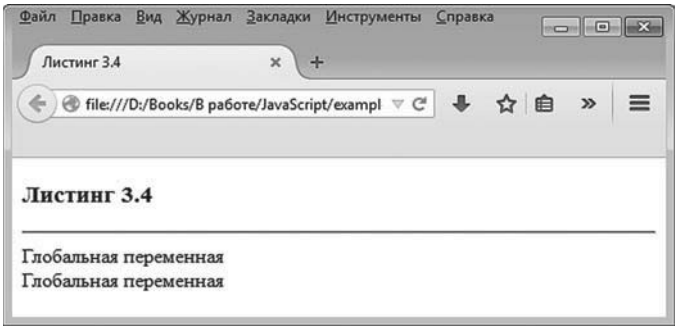

Рис. 3.4. Документ со сценарием, в котором глобальная переменная создается в теле функции

Глобальная переменная как свойство объекта окна

Зададимся вопросом: а что будет, если в сценарии объявлена глобальная переменная, а в теле функции есть локальная переменная с таким же именем и нам нужно использовать обе эти переменные в функции? Другими словами, вопрос состоит в том, как в теле функции обращаться к локальной и глобальной переменным, если у них одинаковые названия?

Для начала рассмотрим один достаточно показательный сценарий, который не решает проблему, но позволяет узнать нечто полезное о способе обработки локальных переменных в функции. Сценарий представлен в листинге 3.5.

Листинг 3.5. Попытка обращения к глобальной переменной в теле функции (файл Listing03_05.js)

```
// Глобальная переменная:
var myText="Глобальная переменная"
// Отображение значения глобальной переменной:
document.write(myText+"<br>")
// Описание функции:
function show(){
  // Присваивание значения переменной:
  myText="Переменная с намеком на глобальность"
  // Отображение значения переменной:
  document.write(myText+"<br>")
```

```
  // Объявление локальной переменной:
  var myText="Локальная переменная"
  // Отображение значения локальной переменной:
  document.write(myText+"<br>")
}
// Вызов функции:
show()
// Отображение значения глобальной переменной:
document.write(myText+"<br>")
```

Структура сценария такая.

- Создается глобальная переменная myText, и ей присваивается значение "Глобальная переменная". Значение переменной отображается в рабочем документе.
- Описывается функция show(). Сразу после описания функции размещена команда вызова функции. При вызове функции в теле функции командой myText="Переменная с намеком на глобальность" переменной myText присваивается значение, и оно отображается в документе.
- Затем выполняется команда var myText="Локальная переменная", которой объявляется локальная переменная с именем myText (таким же, как и имя глобальной переменной), переменной присваивается значение "Локальная переменная". Значение переменной отображается в рабочем документе.
- После завершения вызова функции show() в рабочий документ снова выводится значение переменной myText.

Что можно ожидать от выполнения такого кода? Вся интрига закручена вокруг команд объявления и присваивания значений переменной myText в теле функции show(). Поскольку сначала переменной myText присваивается значение "Переменная с намеком на глобальность", а затем объявляется локальная переменная с именем myText, можно было бы ожидать, что значение "Переменная с намеком на глобальность" присваивается глобальной переменной. Если так, то локальная переменная в теле функции должна бы иметь значение "Локальная переменная", а после вызова функции у глобальной переменной значение должно было бы быть "Переменная с намеком на глобальность". Но *это не так*. При выполнении функции show() значение глобальной переменной не меняется. Об этом сви-

детельствует результат выполнения сценария. Сценарием выводятся следующие сообщения.

Результат выполнения сценария (из листинга 3.5)

```
Глобальная переменная
Переменная с намеком на глобальность
Локальная переменная
Глобальная переменная
```

Проверить работу сценария можем с помощью следующего HTML-кода:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 3.5</title>
</head>
<body><h3>Листинг 3.5</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing03_05.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Результат отображения в браузере веб-документа с указанным кодом показан на рис. 3.5.

Какой из этого следует вывод? Вывод такой, что в теле функции show() командой myText="Переменная с намеком на глобальность" значение присваивается не глобальной, а локальной переменной. И это при том, что объявление локальной переменой myText выполняется после данной команды. Хотя ситуация может показаться странной, тем не менее так и есть. Чтобы понять саму возможность происходящего, следует учесть механизм создания локальных переменных при вызове функции. Важ-

ное обстоятельство связано с тем, что локальные переменные создаются в самом начале выполнения функции, причем вне зависимости от того, где именно в программном коде функции находится инструкция объявления локальной переменной. Если инструкция объявления локальной переменной содержит одновременно и присваивание значения переменной, то при загрузке кода функции под переменную выделяется место, а присваивание значения переменной выполняется в том месте кода, где размещена соответствующая команда.

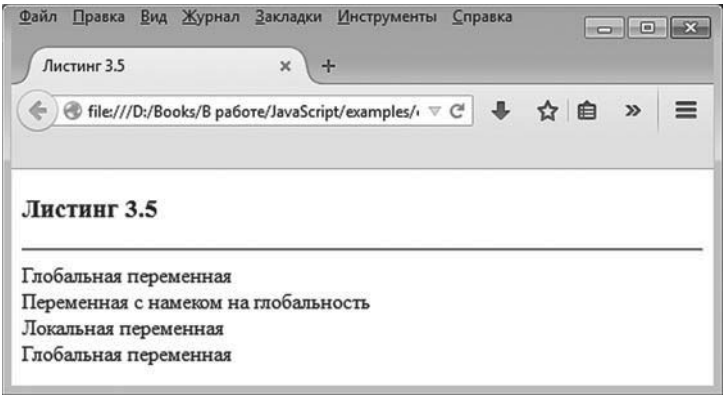

Рис. 3.5. Результат отображения документа со сценарием, в котором выполняется неудачная попытка обращения к глобальной переменной

НА ЗАМЕТКУ

Сказанное означает, что в теле функции локальной переменной можно присвоить значение и только после этого объявить переменную. Хотя такой подход и допустим, злоупотреблять им не рекомендуется.

В нашем примере, несмотря на то что в теле функции show() объявление локальной переменной myText выполняется после первого присваивания значения переменной myText, в обоих случаях речь идет об одной и той же локальной переменной.

ДЕТАЛИ

При выполнении кода функции show() на самом деле происходит примерно следующее. Создается локальная переменная myText. Переменной присваивается значение "Переменная с намеком на глобальность" и данное значение отображается в рабочем документе. Далее пе-

ременной присваивается значение "Локальная переменная" и новое значение переменной выводится в документ. При завершении выполнения кода функции переменная myText удаляется из памяти.

Мы внесли некоторую ясность в способ создания локальных переменных, но остается открытым вопрос о том, как одновременно использовать в функции локальную и глобальную переменные с одинаковыми именами. Ответ состоит в том, чтобы для глобальных переменных использовать в названии инструкцию window, которая является ссылкой на объект окна. Ссылка window указывается перед именем переменной и отделяется от него точкой. Например, если в теле функции есть локальная переменная с именем myText и нужно обратиться к глобальной с таким же именем, то просто имя myText будет означать локальную переменную, а инструкция window.myText представляет собой обращение к глобальной переменной.

ДЕТАЛИ

Инструкция window является ссылкой на объект рабочего окна документа. Поскольку это объект, у него есть *методы* и *свойства*. С методами объекта window мы уже сталкивались, когда использовали метод prompt() для отображения диалогового окна с полем ввода. Есть, например, еще метод alert(), отображающий диалоговое окно с сообщением. Метод — это та же функция, но «прикрепленная» к объекту. Свойство — это «прикрепленная» к объекту переменная. Вообще обращение к методам и свойствам объекта выполняется в «точечном» формате: указывается имя объекта и через точку имя метода или свойства. Но особенность объекта window такова, что при обращении к его свойствам и методам сам объект window можно не указывать. Этой особенностью объекта window мы и пользовались.

Глобальные переменные, которые создаются в сценарии, являются свойствами объекта window. К таким переменным можно обращаться с указанием имени объекта. Обычно в этом потребности нет. Но в теле функции с локальной переменной имя которой совпадает с именем глобальной переменной, явное указание объекта window при обращении к глобальной переменной позволяет избежать двусмысленности.

Объекты, методы и свойства обсуждаются в следующей главе.

Сценарий, в котором в теле функции используются глобальная и локальная переменные с совпадающими названиями, представлен в листинге 3.6.

Листинг 3.6. Обращения к глобальной переменной в теле функции (файл Listing03_06.js)

```
// Глобальная переменная:
var myText="Глобальная переменная"
// Отображение значения глобальной переменной:
document.write(myText+"<br>")
// Описание функции:
function show(){
  // Объявление локальной переменной:
  var myText="Локальная переменная"
  // Присваивание значения глобальной переменной:
  window.myText="Переменная с намеком на глобальность"
  // Отображение значения локальной переменной:
  document.write(myText+"<br>")
  // Отображение значения глобальной переменной:
  document.write(window.myText+"<br>")
}
// Вызов функции:
show()
// Отображение значения глобальной переменной:
document.write(myText+"<br>")
```

На рис. 3.6 представлен веб-документ, открытый в окне браузера, в котором выполняется сценарий.

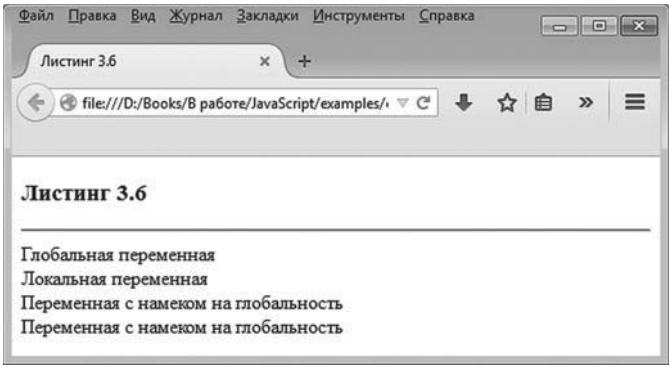

Рис. 3.6. Результат выполнения сценария с функцией, в которой используются локальная и глобальная переменная с совпадающими названиями

Для веб-документа мы использовали следующий HTML-код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 3.6</title>
</head>
<body><h3>Листинг 3.6</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing03_06.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Таким образом, при выполнении сценария в рабочий документ выводится четыре текстовых строки.

Результат выполнения сценария (из листинга 3.6)

```
Глобальная переменная
Локальная переменная
Переменная с намеком на глобальность
Переменная с намеком на глобальность
```

Первая строка представляет собой начальное значение глобальной переменной myText. После отображения значения переменной myText вызывается функция show(). В теле этой функции объявляется локальная переменная myText со значением "Локальная переменная", а глобальной переменной myText присваивается новое значение "Переменная с намеком на глобальность". Значения обеих переменных отображаются в документе (сначала значение локальной переменной myText, а затем значение глобальной переменной myText).

После завершения выполнения функции show() проверяется значение глобальной переменной myText. Как и ожидалось, оно изменилось.

ДЕТАЛИ

В теле функции show() обращение к глобальной переменной myText выполняется в виде window.myText. Вне кода функции обращаемся к глобальной переменной myText просто по имени. Необходимости указывать ссылку на объект окна в этом случае нет.

Аргументы функции

Серьезное отношение к чему бы то ни было
в этом мире является роковой ошибкой.

Л. Кэрролл «Алиса в Стране чудес»

Аргументы, указанные при описании функции, имеют область доступности локальных переменных. От локальных переменных аргументы отличаются двумя аспектами: во-первых, аргументы просто указываются (в то время как локальные переменные объявляются в теле функции с ключевым словом var), и, во-вторых, аргументам функции значения обычно не присваиваются, в то время как для использования локальной переменной ей необходимо присвоить значение.

Аргумент как локальная переменная

Как пример рассмотрим небольшой сценарий, в котором глобальная переменная и аргумент функции имеют одинаковые названия. В теле функции при обращении к аргументу возникает неоднозначность относительно того, что имеется в виду: аргумент или глобальная переменная? Ответ простой: аргумент имеет приоритет. Поэтому в теле функции обращение к аргументу выполняется так, как если бы не было глобальной переменной с таким же именем. Если нам понадобится в теле функции выполнить обращение к глобальной переменной, то перед ее именем указывается, через точку, ключевое слово window. Имеем дело с похожей ситуацией, когда совпадали названия локальной и глобальной переменных. Только теперь в роли локальной переменной выступает аргумент функции.

Небольшая иллюстрация к сказанному представлена в листинге 3.7. В сценарии описана функция с одним аргументом, и название аргумента совпадает с именем глобальной переменной.

Листинг 3.7. Название аргумента совпадает с названием глобальной переменной (файл Listing03_07.js)

```
// Глобальная переменная:
var x="Альфа"
// Вызов функции:
show("Браво")
// Описание функции с аргументом:
function show(x){
  document.write("<h4>Выполнение функции</h4>")
  // Обращение к аргументу функции:
  document.write("Аргумент: "+x+"<br>")
  // Обращение к глобальной переменной:
  document.write("Переменная: "+window.x+"<br>")
}
```

Веб-документ с результатом выполнения данного сценария представлен на рис. 3.7.

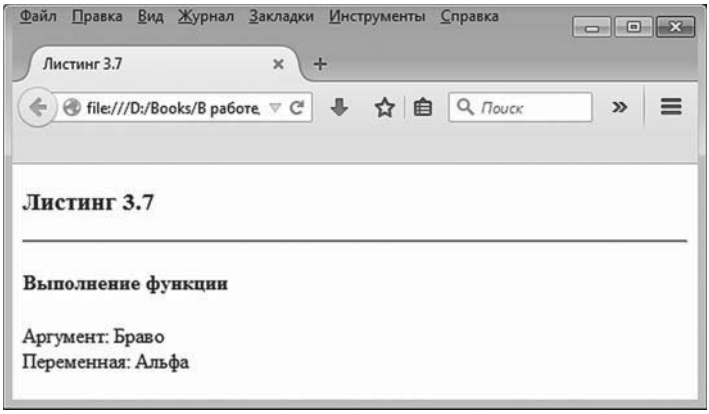

Рис. 3.7. Результат выполнения сценария с функцией, в которой название аргумента совпадает с названием глобальной переменной

Мы использовали следующий код для веб-документа:

```
<!DOCTYPE HTML>
<html>
<head>
```

```
  <title>Листинг 3.7</title>
</head>
<body><h3>Листинг 3.7</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing03_07.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

В сценарии объявлена переменная x со значением "Альфа". Такое же имя (то есть x) использовано как название аргумента функции show(). В такой ситуации инструкция x в теле функции означает обращение к аргументу функции. Если мы хотим получить доступ к глобальной переменной с таким же именем, используем полную ссылку window.x.

В теле функции show() командой document.write("<h4>Выполнение функции</h4>") отображается заголовок четвертого уровня, затем командой document.write("Аргумент: "+x+"
") отображается значение аргумента функции, и, наконец, командой document.write("Переменная: "+window.x+"
") в рабочем документе отображается значение глобальной переменной. Поэтому при вызове функции show() с аргументом "Браво" (команда show("Браво")) в рабочем документе отображается заголовок, значение аргумента и значение глобальной переменной.

НА ЗАМЕТКУ

Мы рассмотрели способ обращения к глобальной переменной в функции. С формальной точки зрения там все корректно. Однако не всегда использование глобальных переменных в функциях приветствуется. Существует концепция *функционального программирования*, в рамках которой результат функции должен определяться исключительно значениями аргументов. Использование глобальных переменных в функции в данную концепцию совершенно не вписывается. При программировании в JavaScript нет необходимости придерживаться концепции функционального программирования.

Механизм передачи аргументов функции

В JavaScript используется достаточно нетривиальный механизм передачи аргументов функции. Чтобы легче было вникнуть в суть проблемы, рассмотрим небольшой пример. Код нужного нам сценария представлен в листинге 3.8.

Листинг 3.8. Механизм передачи аргументов функции (файл Listing03_08.js)

```
// Функция с аргументом:
function f(x){
  // Значение аргумента:
  document.write("Аргумент: "+x+"<br>")
  // Изменяется значение аргумента:
  x++
  // Новое значение аргумента:
  document.write("Аргумент: "+x+"<br>")
}
// Переменная:
var num=100
// Значение переменной до вызова функции:
document.write("Переменная: "+num+"<br>")
// Вызов функции:
f(num)
// Значение переменной после вызова функции:
document.write("Переменная: "+num+"<br>")
```

В сценарии описана функция f() с одним аргументом, который, как мы предполагаем, является числом. При вызове функции отображается значение переданного ей аргумента, затем значение аргумента увеличивается на единицу, и снова отображается уже новое значение аргумента.

В сценарии объявляется переменная num со значением 100. Значение переменной отображается в рабочем документе. После этого командой f(num) вызывается функция f(), аргументом которой передана переменная num. Затем после вызова функции снова проверяется значение

переменной num. Поскольку при вызове функции, как мы помним, значение аргумента увеличивается на единицу, то логично было бы ожидать, что проверка значения переменной num после вызова функции f() даст новое (увеличенное на единицу) значение. Однако на самом деле значение переменной не меняется.

Доказательством тому служит рис. 3.8, на котором представлен веб-документ с результатом выполнения данного сценария.

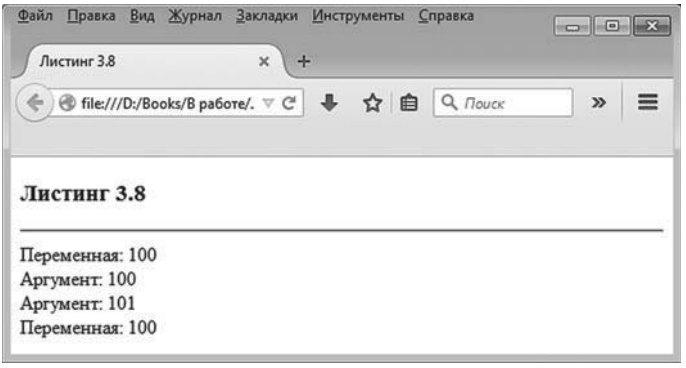

Рис. 3.8. Результат выполнения сценария с функцией, в которой выполняется попытка изменить значение аргумента

Для веб-документа нами использован следующий HTML-код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 3.8</title>
</head>
<body><h3>Листинг 3.8</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing03_08.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Результат выполнения сценария более чем странный (во всяком случае для тех, кто не сталкивался с механизмами передачи аргументов функциям и методам). Действительно, при проверке в теле функции значение аргумента увеличивается на единицу, а когда мы проверяем переменную, передававшуюся аргументом, ее значение остается неизменным.

Результат выполнения сценария (из листинга 3.8)

```
Переменная: 100
Аргумент: 100
Аргумент: 101
Переменная: 100
```

Чтобы понять причину происходящего, необходимо учесть, что при передаче аргументов функции передаются не непосредственно указанные аргументы, а их технические (создаваемые автоматически) *копии*. Такой механизм передачи аргументов называется *передачей по значению*, и именно он используется в JavaScript.

Теперь еще раз проанализируем код сценария и постараемся понять, почему происходит то, что происходит. Итак, с переменной `num` до передачи ее аргументом функции `f()` все достаточно просто: ей присваивается значение `100`, после чего данное значение отображается в рабочем документе (первая строка из четырех, выводимых при выполнении сценария). Затем вызывается функция `f()` с аргументом `num`. Но мы уже знаем, что на самом деле в функцию передается не непосредственно переменная `num`, а ее копия. У копии такое же значение, как и у исходной переменной. Поэтому, когда в функции `f()` выполняется команда для первого отображения значения переменной `num`, на самом деле отображается значение копии переменной `num`, и оно так же, как и значение оригинала, равняется `100`. Но вот при увеличении значения аргумента на самом деле увеличивается значение копии аргумента. И в документе отображается новое значение копии аргумента (которое равно `101`). А сама переменная `num` своего значения не меняет. Поэтому нет ничего удивительного, что после вызова функции `f()` проверка показывает, что значение переменной `num` равно `100` (то есть оно не изменилось).

НА ЗАМЕТКУ

Обычно можно встретить утверждение, что в JavaScript по значению передаются аргументы, относящиеся к простым типам (чис-

ла, например), а вот объекты передаются *по ссылке* (то есть непосредственно те переменные, что указаны аргументом). Данное утверждение несколько искажает реальную ситуацию. Подробнее особенности передачи аргументами объектов обсуждаются при рассмотрении методов объектно-ориентированного программирования.

Проверка типа аргумента

Тип аргументов функции (как и тип переменных при объявлении) не указывается. В некотором смысле так удобно, поскольку аргумент или переменная не «привязаны» к какому-то конкретному типу данных. Это значительно расширяет «пространство для маневра». Но, с другой стороны, часто функция создается с расчетом на то, что аргумент (или аргументы) относятся к определенному типу. Поэтому в функциях особенно важно иметь возможность проверять тип переданного функции аргумента.

Для проверки типа переменной или, более конкретно, аргумента используют оператор typeof. Операндом данного оператора указывается переменная (в нашем случае аргумент функции), тип которой следует определить. Результатом возвращается текстовое значение, определяющее тип проверяемой переменной.

НА ЗАМЕТКУ

Операнд может указываться в скобках и без скобок. Мы будем использовать скобки.
Под типом переменной в данном случае подразумевается тип значения, на которое ссылается в данный момент переменная.

Если операнд оператора typeof является числом, возвращается значение "number". Для текстовых значений возвращается строка "string".

ДЕТАЛИ

Другие возможные результаты, возвращаемые оператором typeof, такие: "boolean" (если переменная ссылается на логическое значение), "function" (если переменная является именем функции или ссылается на функцию), "object" (если переменная ссылается на объект или ее значением является пустая ссылка) и "undefined" (если значение переменной не определено).

Например, если мы хотим проверить, относится ли значение некоторой переменной x к числовому типу, используем инструкцию typeof(x)=="number". Значением данного выражения является true, если переменная x ссылается на числовое значение, и false в противоположном случае.

Небольшая иллюстрация к использованию оператора typeof для проверки типа аргумента функции представлена в листинге 3.9, в котором описана функция f() с одним аргументом (обозначенным как x). При вызове функции проверяется тип аргумента, и в зависимости от этого в рабочий документ выводится сообщение соответствующего содержания.

Листинг 3.9. Проверка типа аргумента функции (файл Listing03_09.js)

```
// Функция с аргументом:
function f(x){
  // Локальная переменная:
  var argType
  // Определение типа аргумента:
  argType=typeof(x)
  // Проверка значения локальной переменной:
  switch(argType){
    // Аргумент - число:
    case "number":
      document.write("Числовой аргумент<br>")
      break
    // Аргумент - текст:
    case "string":
      document.write("Текстовый аргумент<br>")
      break
    // Все прочие случаи:
    default:
      document.write("Не текст и не число<br>")
  }
}
// Вызов функции:
f("текст") // С текстовым аргументом
```

```
f(123)    // С целочисленным аргументом
f(10.5)   // Аргумент - действительное число
f(true)   // Логический аргумент
// Объявление переменной (без присваивания значения):
var a
f(a)      // Аргумент без значения
```

В теле функции f() объявляется локальная переменная argType, и этой переменной присваивается значение typeof(x) (команда argType=typeof(x)). Далее запускается оператор выбора switch(), в котором проверяется значение переменной argType. В первом case-блоке указано контрольное значение "number". Если значение переменной argType равно "number", выполняется команда document.write("Числовой аргумент
").

Если значение переменной argType равно "string" (контрольное значение во втором case-блоке), выполняется команда document.write("Текстовый аргумент
"). Также в операторе выбора есть default-блок, в котором команда document.write("Не текст и не число
") выполняется, если значение переменной argType отлично от "number" и "string".

Таким образом, в функции выделяются три ситуации: когда у функции числовой аргумент, когда у функции текстовый аргумент, и все прочие типы аргумента.

В сценарии есть несколько команд вызова функции с разными аргументами. При вызове функции командой f("текст") с текстовым аргументом отображается сообщение Текстовый аргумент. При вызове функции командами f(123) и f(10.5) каждый раз отображается сообщение Числовой аргумент.

В результате выполнения команды f(true) отображается сообщение Не текст и не число. Такое же сообщение появляется, если функция f() вызывается с аргументом, значение которого не определено. В частности, в сценарии командой var a объявляется переменная a (но значение ей не присваивается), после чего данная переменная (без значения) передается аргументом функции f() (команда f(a)).

> **НА ЗАМЕТКУ**
>
> Если значение переменной a не определено, то результатом выражения typeof(a) является "undefined".

Таким образом, в результате выполнения сценария появляются такие сообщения.

Результат выполнения сценария (из листинга 3.9)

```
Текстовый аргумент
Числовой аргумент
Числовой аргумент
Не текст и не число
Не текст и не число
```

Чтобы увидеть, как все это выглядит на практике, создаем веб-документ с таким HTML-кодом:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 3.9</title>
</head>
<body><h3>Листинг 3.9</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing03_09.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 3.9 показан открытый в окне браузера веб-документ, содержащий результат выполнения сценария.

Здесь мы рассмотрели очень простой пример, в котором по большому счету просто определяется тип аргумента. На практике проверка типа аргумента может выполняться для программирования реакции функции на некорректный аргумент.

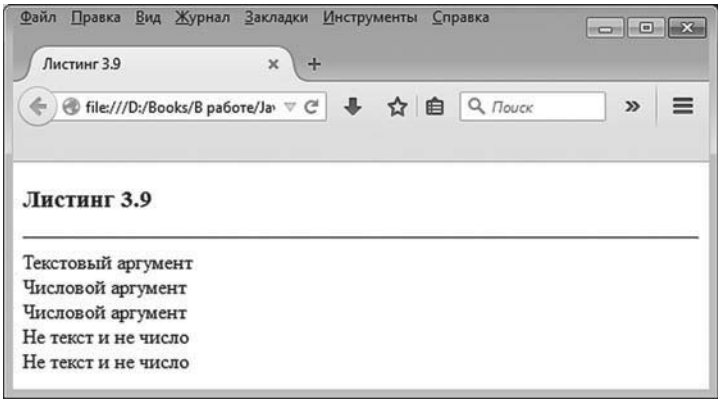

Рис. 3.9. Проверка типа аргумента функции

Количество аргументов функции

Помимо типа аргументов функции, важным фактором является количество аргументов функции. На первый взгляд ситуация с количеством аргументов достаточно банальная, поскольку аргументы явно указываются при описании функции. То есть описание функции как бы однозначно определяет, сколько аргументов должно быть у функции. Но в JavaScript есть одна важная особенность: при вызове функции несоответствие количества переданных функции аргументов количеству аргументов, указанных при описании функции, само по себе не вызывает ошибки (но ошибка может возникнуть уже в процессе выполнения кода функции). Далее мы рассмотрим пример, который иллюстрирует, как такая особенность JavaScript может быть использована на практике.

Понятно, что при вызове функции количество переданных ей аргументов может:

- совпадать с количеством аргументов, указанных при описании функции;
- превышать количество аргументов, указанных при описании функции;
- быть меньше количества аргументов, указанных при описании функции.

Мы рассмотрим последний случай. А именно попробуем создать иллюзию, что у аргументов функции есть значения по умолчанию.

НА ЗАМЕТКУ

В некоторых языках программирования (например, C++) аргументы функции могут иметь значения по умолчанию. Если у аргумента есть значение по умолчанию, то при вызове функции такой аргумент можно не указывать. Вместо отсутствующего аргумента при выполнении кода функции используется значение по умолчанию.
В JavaScript нельзя задать значение по умолчанию для аргумента. Но зато можно создать иллюзию, что такое значение у аргумента есть.

Рассмотрим сценарий, представленный в листинге 3.10.

Листинг 3.10. Эмуляция значений по умолчанию для аргументов функции (файл Listing03_10.js)

```
// Функция для возведения числа в степень:
function power(x,n){
  // Если не указан второй аргумент:
  if(typeof(n)=="undefined"){
    n=1 // Значение по умолчанию для второго аргумента
  }
  // Если не указан первый аргумент:
  if(typeof(x)=="undefined"){
    x=1 // Значение по умолчанию для первого аргумента
  }
  // Локальные переменные:
  var s=1,k
  // Вычисление степени числа:
  for(k=1;k<=n;k++){
    s*=x
  }
  // Результат функции:
  return s
}
// Вызов функции с двумя аргументами:
document.write("2<sup>3</sup> = "+power(2,3)+"<br>")
// Вызов функции с одним аргументом:
```

```
document.write("5<sup>1</sup> = "+power(5)+"<br>")
// Вызов функции без аргументов:
document.write("1<sup>1</sup> = "+power()+"<br>")
```

В сценарии описана функция power(), предназначенная для возведения числа в степень. Возводимое в степень число передается первым аргументом функции, а степень, в которую возводится число, передается вторым аргументом функции. Но функция описана так, что ей можно передавать не два, а только один аргумент или вызывать функцию и вовсе без аргументов. Если при вызове функции указан только первый аргумент, то по умолчанию значение второго аргумента устанавливается равным единице. Если при вызове функции не указаны оба операнда функции, то оба они по умолчанию полагаются равными единице. Для реализации подобного подхода в программном коде функции размещены два условных оператора. Они выполняют похожие операции. В первом условном операторе проверяется условие typeof(n)=="undefined" (через n обозначен второй аргумент функции power()). Если при вызове функции второй аргумент не указан, то его значение не определено. В таком случае результатом выражения typeof(n) является текст "undefined". Другими словами, если второй аргумент функции не передан, то условие typeof(n)=="undefined" истинно. В этом случае выполняется команда n=1, которой второму аргументу присваивается единичное значение.

ДЕТАЛИ

Ситуация на первый взгляд странная: несуществующему аргументу присваивается значение. Но здесь стоит вспомнить, что при передаче аргументов функции на самом деле передаются копии этих аргументов. Если функции второй аргумент не передан, то значение копии этого аргумента не определено. Когда в теле функции присваивается значение второму аргументу, то на самом деле оно присваивается упомянутой копии.

Аналогично, во втором условном операторе для первого аргумента x функции проверяется условие typeof(x)=="undefined". Если первый аргумент не указан при вызове функции, то условие истинно, и тогда выполняется команда x=1, которой первому аргументу присваивается единичное значение (значение по умолчанию для первого аргумента).

После выполнения условных операторов запускается оператор цикла, в котором, собственно, и вычисляется степень числа, а по завершении вычислений функцией возвращается соответствующий результат.

Сценарий содержит несколько команд, в которых функция power() вызывается с двумя аргументами (инструкция power(2,3)), с одним аргументом (инструкция power(5)) и без аргументов (инструкция power()).

НА ЗАМЕТКУ

Вызов функции power() без второго аргумента эквивалентен вызову функции со вторым единичным элементом. Вызов функции power() без аргументов эквивалентен вызову функции с двумя единичными аргументами.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Напомним, что дескрипторы ^и используются для отображения верхних индексов. Мы в данном случае используем данные дескрипторы для отображения степени числа.

Чтобы увидеть результат выполнения сценария, воспользуемся следующим веб-документом:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 3.10</title>
</head>
<body><h3>Листинг 3.10</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing03_10.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 3.10 показано, как данный документ выглядит в окне браузера.

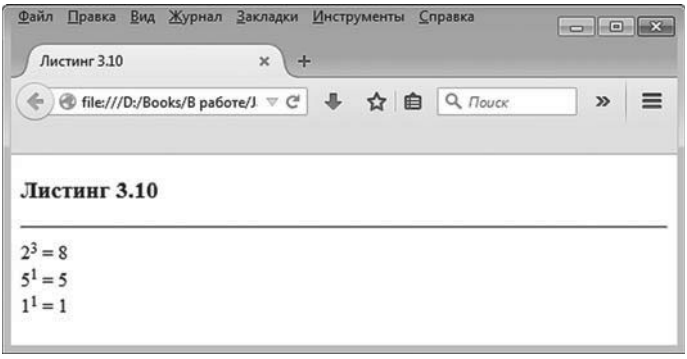

Рис. 3.10. Веб-документ с результатом выполнения сценария, в котором одна и та же функция вызывается с разным количеством аргументов

Здесь мы рассмотрели небольшую иллюстрацию относительно манипуляций с количеством аргументов функции. К данному вопросу мы еще вернемся, но немного позже.

> **НА ЗАМЕТКУ**
>
> После того как мы познакомимся с объектами и массивами, мы рассмотрим задачу о том, как создать функцию с произвольным количеством аргументов.

Передача функции аргументом

Аргументом функции может передаваться функция. Это очень мощное свойство, которое позволяет с успехом, легко и эффективно решать широкий класс задач.

С технической точки зрения передать аргументом функции другую функцию достаточно просто: соответствующий аргумент в теле функции обрабатывается как имя функции. Соответственно, если при вызове функции аргументом ей передается другая функция, то такой аргумент — это имя функции (которая передается аргументом).

Рассмотрим небольшой пример, в котором описывается функция, предназначенная для вычисления в числовом виде производной. Функция, для которой вычисляется производная, передается аргументом функции, которая вычисляет производную. Еще два аргумента: точка, в которой вычисляется производная, и приращение, на основе которого вычисляется производная.

ДЕТАЛИ

Предположим, имеется некоторая числовая математическая функция *f(x)*. Производной функцией от функции *f(x)* называется предел отношения приращения функции *f(x)* к приращению аргумента, когда последний стремится к нулю. Математически формула для производной (обозначается как *df/dx* или *f'(x)*) записывается следующим образом: $f'(x) = lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}$.

Если речь идет о вычислении производной в числовом виде, то используется приближенная формула $f'(x) \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}$.

Чем меньше приращение Δx, тем точнее вычисляемое значение для производной. В сценарии мы для вычисления производной будем использовать именно формулу $f'(x) \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}$.

С идеологической точки зрения для нас важно то, что вычисление производной представляет собой, по сути, вычисление одной функции на основе другой функции.

Далее предлагается сценарий, в котором описана функция diff(), предназначенная для вычисления производной от функции. У функции diff() три аргумента:

- первый аргумент f, как предполагается, есть имя функции, на основе которой вычисляется производная (дифференцируемая функция *f(x)* в формуле для вычисления производной);
- второй аргумент x функции diff() — это точка, в которой вычисляется производная (аргумент x в формуле для функции-производной);
- третий аргумент dx функции diff() — значение для приращения аргумента (параметр Δx в формуле для вычисления производной).

Значение функции diff() вычисляется инструкцией (f(x+dx)-f(x))/dx, которая возвращается результатом функции. Еще в сценарии описывается две вспомогательные функции. Функция G() для аргумента x возвращает значение x*x+x, а функция g() для аргумента x возвращает значение 2*x+1. Это соответственно дифференцируемая функция и функция, определяющая точное значение для производной.

ДЕТАЛИ

Если функция $G(x) = x^2 + x$, то ее производная $G'(x) = 2x + 1$. В сценарии для проверки работы функции diff() описывается функция G(), которая соответствует функции $G(x) = x^2 + x$, а функция g() соответствует производной $G'(x) = 2x + 1$.

Для функции $G(x) = x^2 + x$ производная вычисляется в числовом виде и сравнивается с точным значением, вычисленным на основе функции $G'(x) = 2x + 1$.

Проверка работы функции diff() выполняется в операторе цикла, в котором переменная z пробегает значения в определенном диапазоне и для каждого значения переменной z вычисляется производная в числовом виде, а полученное значение сравнивается с точным значением для производной в той же точке.

Теперь рассмотрим сценарий, представленный в листинге 3.11.

Листинг 3.11. Передача функции аргументом (файл Listing03_11.js)

```
// Функция для вычисления производной:
function diff(f,x,dx){
  // Результат функции:
  return (f(x+dx)-f(x))/dx
}
// Функция для передачи аргументом:
function G(x){
  // Значение функции:
  return x*x+x
}
// Точное значение производной:
function g(x){
  return 2*x+1
}
document.write("<h4>Вычисление производной</h4>")
// Вычисление производной:
for(var z=0;z<=2;z+=0.5){
  document.write(g(z)+"  vs.  "+diff(G,z,0.001)+"<br>")
}
```

Для проверки работы сценария используем такой веб-документ:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 3.11</title>
</head>
<body><h3>Листинг 3.11</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing03_11.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 3.11 показан соответствующий веб-документ, открытый в окне браузера.

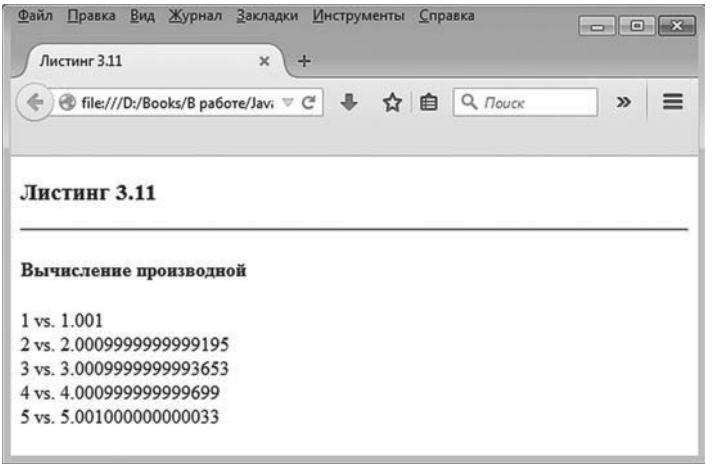

Рис. 3.11. Результат выполнения сценария с функцией, предназначенной для вычисления производной

В принципе точность вычисления производной достаточно неплохая. Но в любом случае здесь нас интересует не столько числовой резуль-

тат, сколько сам принцип организации программного кода, в котором аргументом одной функции передается другая функция.

Рекурсия

> Сердце подвластно разуму. Чувства подвластны сердцу. Разум подвластен чувствам. Круг замкнулся. С разума начали, разумом кончили.
>
> *из к/ф «Формула любви»*

В принципе мы уже имеем некоторое представление о том, как описывается функция. Здесь рассмотрим такой легендарный способ определения функции, как *рекурсия*.

Под рекурсией подразумевают ситуацию, когда в программном коде описания функции вызывается эта же функция (обычно с другим аргументом или аргументами). С помощью рекурсии создают очень эффектные, но не всегда эффективные коды.

Рассмотрим сценарий, в котором с помощью рекурсии описывается функция, предназначенная для вычисления чисел Фибоначчи.

ДЕТАЛИ

Последовательность Фибоначчи определяется следующим образом: первые два числа равны единице, а каждое следующее равняется сумме двух предыдущих. Таким образом, начальные числа в последовательности: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 и так далее.
Важно в данном случае то обстоятельство, что мы задаем не формулу для непосредственного вычисления очередного числа, а алгоритм его вычисления, причем в этом случае новое число вычисляется на основе двух предыдущих. Такого типа соотношения называются рекуррентными. Они удобны для использования рекурсии.

Рассмотрим сценарий, представленный в листинге 3.12.

Листинг 3.12. Вычисление чисел Фибоначчи с помощью рекурсии (файл Listing03_12.js)

```
// Функция для вычисления чисел Фибоначчи:
function fibs(n){
```

```
  // Первое или второе число:
  if(n==1||n==2){
    return 1
  }
  // Прочие числа:
  else{
    return fibs(n-1)+fibs(n-2)
  }
}
document.write("<h4>Числа Фибоначчи</h4>")
// Вычисление чисел Фибоначчи:
for(var k=1;k<=10;k++){
  document.write(fibs(k)+"  ")
}
```

Рекурсия используется в описании функции fibs(), предназначенной для вычисления числа Фибоначчи по его порядковому номеру в последовательности (аргумент n). В теле функции в условном операторе проверяется значение аргумента функции на предмет равенства 1 или 2. Если так, то речь идет о вычислении первого или второго числа в последовательности, которые по определению равняются единице.

Поэтому при истинности условия n==1||n==2 выполняется команда return 1. Если же условие ложно, выполняется инструкция return fibs(n-1)+fibs(n-2), которой результат функции вычисляется путем вызова этой же функции, но с измененным аргументом.

ДЕТАЛИ

Чтобы понять, как на практике работает рекурсия, рассмотрим в общих чертах процесс вычисления значения выражения fibs(5). Поскольку в этом случае аргумент функции отличен от 1 и 2, то результат функции вычисляется как fibs(4)+fibs(3). В этом выражении каждое слагаемое вычисляется отдельно. Сначала вычисляется значение выражения fibs(4). Чтобы вычислить выражение fibs(4), вычисляется значение выражения fibs(3)+fibs(2). Для вычисления выражения fibs(3) вычисляется выражение fibs(2)+fibs(1). Каждое из слагаемых в последнем выражении равно 1, поэтому удается вычислить значение fibs(3). Затем вычисляется выражение fibs(2), что позволяет вычислить вы-

ражение fibs(4) как сумму fibs(3)+fibs(2). Но это мы получаем только первое слагаемое в сумме fibs(4)+fibs(3). Остается вычислить второе слагаемое fibs(3). Поэтому снова вычисляется сумма fibs(2)+fibs(1). В итоге получаем значение для fibs(3), после чего можем получить значение суммы fibs(4)+fibs(3) — то есть значение fibs(5). Понятно, что если аргумент у функции fibs() взять побольше, то и объем вычислений вырастет существенно.

Помимо описания функции fibs(), сценарий содержит оператор цикла, в котором функция fibs() вызывается для вычисления последовательности чисел.

Ниже приведен HTML-код для веб-документа, который мы используем для тестирования сценария.

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 3.12</title>
</head>
<body><h3>Листинг 3.12</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing03_12.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Как выглядит загруженный в браузер веб-документ с данным кодом, дает представление рис. 3.12.

Следует заметить, что, хотя программный код функции для вычисления чисел Фибоначчи выглядит достаточно компактно, с точки зрения оптимальности, в силу использования рекурсии, он не идеален. Для рекурсивно определенных функций такая ситуация стандартна. Но все же рекурсию не стоит сбрасывать со счетов. Дело в том, что нередко использованию рекурсии просто нет альтернативы.

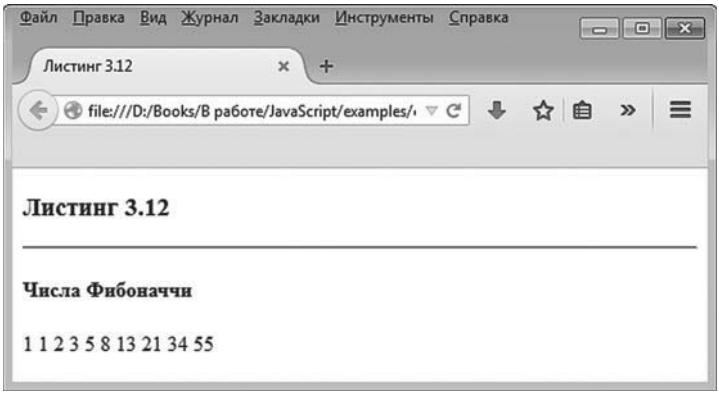

Рис. 3.12. Результат выполнения сценария, в котором с использованием рекурсии вычисляются числа Фибоначчи

Внутренние функции

> Холмс, это исключено. Сразу видно, что вы мало читаете.
>
> *из к/ф «Приключения Шерлока Холмса и доктора Ватсона»*

В JavaScript разрешается в теле функции описывать другие функции. Функции, описанные в теле функций, называются *внутренними*. Функцию, содержащую описание внутренней функции, будем называть *внешней*. Главная особенность внутренней функции связана с тем, что она имеет доступ к локальным переменным внешней функции (эта особенность внутренних функций называется *замыканием*). При этом внешняя функция не имеет доступа к локальным переменным внутренней функции. Поясним сказанное на примерах.

Для начала рассмотрим достаточно простую ситуацию. Напишем сценарий с функцией, предназначенной для вычисления натурального логарифма. В этой функции опишем внутреннюю функцию, которая вычисляет результат возведения числа в степень.

ДЕТАЛИ

Речь идет о функции *f(x)* = ln(1 + *x*) (логарифм натуральный от выражения 1 + *x*, где через *x* обозначен аргумент функции). При значениях аргумента $|x| < 1$ значение выражения ln(1 + *x*) может быть вычислено как сумма ряда $ln(1+x) \approx x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots + \frac{(-1)^{n+1} x^n}{n}$.

Чем больше значение верхней границы ряда *n*, тем точнее сумма ряда аппроксимирует значение выражения ln(1 + *x*).

Именно такой подход мы используем при описании функции mlog(), предназначенной для вычисления значения выражения ln(1 + *x*). Причем для вычисления степенных слагаемых вида x^k мы описываем внутреннюю функцию power(). При выполнении расчетов мы учли, что $ln(1+x) \approx \sum_{k=1}^{n} \frac{(-1)^{k+1} x^k}{k} = -\sum_{k=1}^{n} \frac{(-x)^k}{k}$.

Другими словами, при вычислении суммы ряда мы определяем переменную с начальным нулевым значением, а затем последовательно для разных значений индекса *k* (в диапазоне от 0 до некоторого значения *n*) от данной переменной отнимается число –*x*, возведенное в степень *k* и деленное на *k*.

Рассмотрим программный код, представленный в листинге 3.13.

Листинг 3.13. Использование внутренних функций (файл Listing03_13.js)

```
// Внешняя функция для вычисления значения логарифма:
function mlog(x){
  // Локальные переменные:
  var s=0,k
  // Количество членов ряда:
  var n=100
  // Вычисление значения ряда:
  for(k=1;k<=n;k++){
    // Вызов внутренней функции:
    s-=power(-x,k)/k
  }
  // Результат внешней функции:
  return s
  // Внутренняя функция для вычисления степени числа:
  function power(z,m){
    // Локальные переменные внутренней функции:
    var p=1,i
    // Вычисление степени числа:
    for(i=1;i<=m;i++){
```

```
      p*=z
    }
    // Результат внутренней функции:
    return p
  } // Окончание описания внутренней функции
}   // Окончание описания внешней функции
document.write("<h4>Использование внутренней функции</h4>")
// Глобальная переменная:
var x=0.5
// Вызов функции для вычисления логарифма:
document.write("ln(1+"+x+") = "+mlog(x)+"<br>")
// Проверка вычисленного значения:
document.write("Проверка:  "+Math.log(1+x)+"<br>")
```

В сценарии описана функция mlog() с одним аргументом (обозначен как x), а в этой функции описана внутренняя функция power() с двумя аргументами (обозначены как z и m). Результатом функция power() возвращает число, равное числу z в степени m.

В теле функции mlog() объявляется несколько переменных (в переменную s, имеющую начальное нулевое значение, записывается сумма ряда, индексная переменная k используется в операторе цикла, а значение переменной n определяет верхнюю границу ряда). Для вычисления ряда запускается оператор цикла, в котором за каждый цикл выполняется команда s-=power(-x,k)/k. Здесь вызывается внутренняя функция power(), которой вычисляется степень k числа -x.

ДЕТАЛИ

Поскольку $\frac{(-1)^{k+1}x^k}{k} = -\frac{(-x)^k}{k}$, то прибавление слагаемого $\frac{(-1)^{k+1}x^k}{k}$ эквивалентно вычитанию слагаемого $\frac{(-x)^k}{k}$.

Именно этим мы и воспользовались, когда вычисляли значение переменной s в теле функции mlog().

По окончании вычислений значение переменной s возвращается как результат функции mlog().

Помимо описания функции mlog(), в сценарии есть пример ее вызова. Причем для сравнения также выводится и значение того же самого логарифма, но только вычисленное с помощью специального метода log() встроенного объекта Math(). Поскольку методом log() вычисляется логарифм от аргумента метода, в то время как функцией mlog() вычисляется логарифм от выражения «аргумент плюс единица», то в сценарии на предмет совпадения сравниваются значения выражений Math.log(1+x) и mlog(x), где глобальная переменная x объявлена со значением 0.5.

Чтобы проверить функциональность сценария, используем веб-документ с таким HTML-кодом:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 3.13</title>
</head>
<body><h3>Листинг 3.13</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing03_13.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Результат выполнения сценария представлен на рис. 3.13.

Видим, что точность вычислений более чем приемлемая. Но это, так сказать, математический результат. В плане программных премудростей рассмотренный нами пример очень простой. Мы всего лишь в одной функции описали другую функцию. Хотя на самом деле внутреннюю функцию могли описать непосредственно в сценарии. Другими словами, способ использования внутренней функции в представленном выше примере самый незатейливый. На практике встречаются куда более интересные случаи (некоторые из них мы рассмотрим в книге и даже в этой главе).

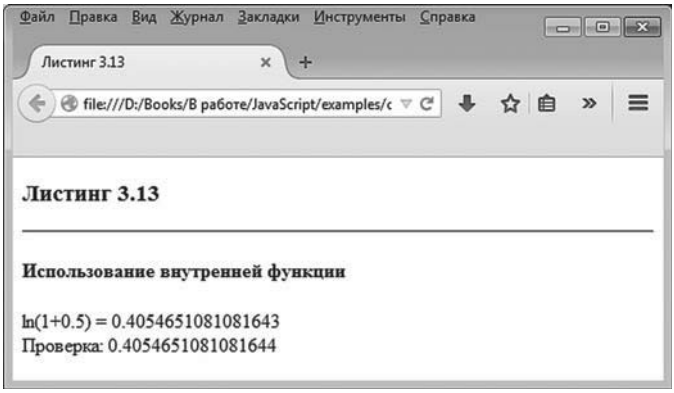

Рис. 3.13. Результат выполнения сценария с функцией, содержащей внутреннюю функцию

Чтобы показать нетривиальность внутренних функций, решим предыдущую задачу, но несколько иным способом (подразумевающим тем не менее использование внутренней функции). Рассмотрим программный код в листинге 3.14.

Листинг 3.14. Внутренняя функция и доступ к локальным переменным внешней функции (файл Listing03_14.js)

```
// Внешняя функция для вычисления значения логарифма:
function mlog(x){
  // Локальные переменные:
  var s=0,k=1,q=x
  // Количество членов ряда:
  var n=100
  // Вычисление значения ряда:
  while(k<=n){
    doNext()
  }
  // Результат внешней функции:
  return s
  // Внутренняя функция:
  function doNext(){
    s+=q/k   // Прибавляется новое слагаемое
    q*=(-1)*x // Новое значение для q
```

```
    k++     // Увеличивается индексная переменная
  } // Окончание описания внутренней функции
}   // Окончание описания внешней функции
document.write("<h4>Использование внутренней функции</h4>")
// Глобальная переменная:
var x=0.5
// Вызов функции для вычисления логарифма:
document.write("ln(1+"+x+") = "+mlog(x)+"<br>")
// Проверка вычисленного значения:
document.write("Проверка:  "+Math.log(1+x)+"<br>")
```

Существенно изменился программный код функции mlog(). Как и ранее, в теле функции объявляется несколько переменных. Переменные s, k и n нам уже знакомы (сумма ряда, индексная переменная и верхняя граница ряда соответственно). Появилась еще одна переменная q с начальным значением x (аргумент функции). Затем в теле функции вызывается оператор цикла while, в котором проверяется условие k<=n. В теле оператора цикла всего одна команда, которой вызывается (без аргументов) внутренняя функция doNext(). По завершении оператора цикла командой return s значение переменной s возвращается результатом функции. Совершенно очевидно, что весь «секрет» кода спрятан во внутренней функции doNext(). А там, в теле внутренней функции, всего три команды. Командой s+=q/k к сумме прибавляется очередное слагаемое, командой q*=(-1)*x вычисляется степенной множитель (с учетом знака), а командой k++ значение индексной переменной увеличивается на единицу. Во всех этих командах наиболее важно то, что внутренняя функция свободно обращается к локальным переменным и аргументу внешней функции.

НА ЗАМЕТКУ

Доступность во внутренней функции локальных переменных внешней функции — дорога с односторонним движением. Внешняя функция не имеет доступа к локальным переменным внутренней функции.

Таким образом, один вызов функции doNext() эквивалентен прибавлению нового слагаемого к сумме ряда. Еще каждый раз при вызове внутренней функции значение переменной k увеличивается на еди-

ницу, поэтому рано или поздно условие k<=n, проверяемое в операторе цикла while во внешней функции, станет ложным и оператор цикла завершит свое выполнение.

Чтобы проверить, к каким результатам приводит выполнение сценария, создаем веб-документ со следующим кодом:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 3.14</title>
</head>
<body><h3>Листинг 3.14</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing03_14.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 3.14 показано, как выглядит данный документ, открытый в окне браузера.

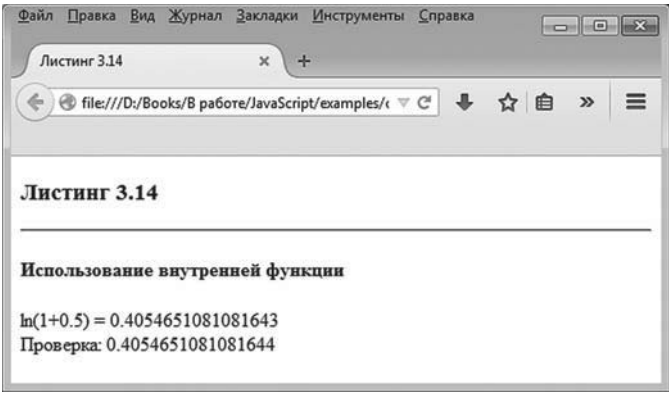

Рис. 3.14. Результат выполнения сценария с внутренней функцией, которая при вызове использует локальные переменные внешней функции

Несложно заметить, что мы получили точно такой же результат, как и в предыдущем случае.

Далее мы рассмотрим еще некоторые важные аспекты, имеющие отношение к использованию функций в JavaScript. В некоторых из этих примеров используются внутренние функции.

Присваивание функций

Ну зачем такие сложности?!

из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Функции на самом деле являются объектами. Поскольку объекты мы еще не обсуждали, то сам по себе факт «объектности» функций пока что мало о чем говорит. Обсуждать функции с «объектной» точки зрения нам только предстоит. Здесь мы лишь воспользуемся маленьким побочным эффектом, который состоит в том, что функции можно присваивать значениями переменным. Более того, имени функции также можно присвоить новое значение, причем это новое значение совсем не обязательно должно быть функцией. Чтобы не быть голословными, сразу рассмотрим пример. В листинге 3.15 представлен листинг, в котором описана функция show() с одним аргументом. Функцией в рабочем документе отображается текст, переданный ей аргументом. Также в тексте объявляется переменная f, а значением этой переменной присваивается функция show() (формально переменной f присваивается имя функции show). В результате переменная f «становится» функцией, и ее используют как функцию — вызывают с текстовым аргументом. Фактически переменная f в таком случае «дублирует» имя функции show(). Но это еще не все: идентификатору show можем присвоить значение (в данном случае текст) как самой обычной переменной. В результате show больше не будет именем функции, а станет переменной с тем значением, которое было присвоено. Все описанные действия продемонстрированы в сценарии, который представлен ниже.

Листинг 3.15. Присваивание функций (файл Listing03_15.js)

```
// Вызов функции:
show("Функция show()")
// Объявляется переменная:
var f
```

```
// Переменной присваивается функция:
f=show
f("Теперь это функция f()")
// Функции значением присваивается текст:
show="Текстовое значение"
document.write(show+"<br>")
f("Снова функция f()")
// Описание функции:
function show(msg){
  document.write(msg+"<br>")
}
```

Результат выполнения сценария следующий.

Результат выполнения сценария (из листинга 3.15)

```
Функция show()
Теперь это функция f()
Текстовое значение
Снова функция f()
```

Хотя основные действия, выполняемые в сценарии, уже описаны выше, имеет смысл заострить внимание на некоторых моментах.

Во-первых, освежим в памяти способ обработки описаний функций в сценарии. Схема такая: перед выполнением команд в сценарии просматриваются объявления функций, и в соответствии с каждым описанием создается функция. При выполнении программного кода в местах вызова функций они вызываются, а те места кода, в которых функции описаны, в некотором смысле игнорируются (просто соответствующий код уже был использован при создании функций). В данном конкретном примере не имеет совершенно никакого значения, что функция show() описана в конце сценария. В начале выполнения сценария данная функция готова к использованию.

Во-вторых, имя функции удобно интерпретировать как ссылку на функцию. То есть имя функции — нечто вроде переменной, значением которой является функция. Поэтому когда выполняется команда

f=show, то означает она следующее: переменная f теперь ссылается на то же значение, что и show, а последняя ссылается на функцию. В результате и f, и show ссылаются на одну и ту же функцию и по факту являются «синонимами». Получается, что f() и show() — одна и та же функция.

По данной логике становится понятным, что в результате выполнения команды show="Текстовое значение" идентификатор show перестает быть именем функции и становится обычной переменной с текстовым значением.

При этом f продолжает ссылаться на функцию. В итоге функция f() — это то, чем раньше была функция show().

Для проверки корректности программного кода воспользуемся веб-документом, HTML-код которого приведен ниже:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 3.15</title>
</head>
<body><h3>Листинг 3.15</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing03_15.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 3.15 показано, как выглядит веб-документ в окне браузера.

Рассмотренный механизм, позволяющий присваивать функции в качестве значений, исключительно мощный и позволяет создавать элегантные коды.

Как еще одну небольшую иллюстрацию рассмотрим пример, в котором две функции «обмениваются значениями». Программный код примера приведен в листинге 3.16.

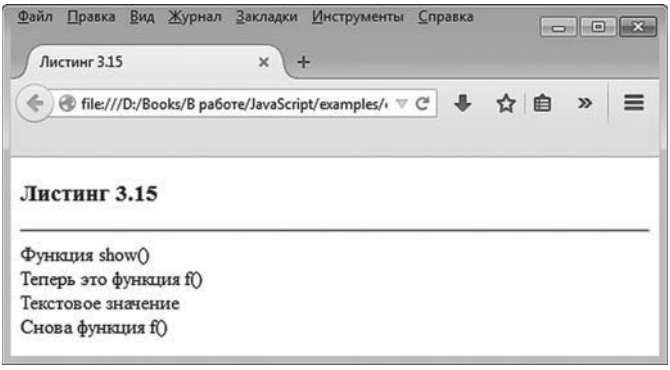

Рис. 3.15. Результат выполнения сценария, в котором функция присваивается значением переменной

Листинг 3.16. Обмен функций (файл Listing03_16.js)

```
// Первая функция:
function first(){
  document.write("Первая функция<br>")
}
// Вторая функция:
function second(){
  document.write("Вторая функция<br>")
}
// Вызов функций:
first()
second()
// Переменная:
var tmp
// Обмен значениями:
tmp=first
first=second
second=tmp
// Вызов функций:
first()
second()
```

В сценарии описываются две функции: обе они не имеют аргументов и не возвращают результат. Первая функция first() отображает в рабочем

документе сообщение Первая функция. Вторая функция second() отображает в документе сообщение Вторая функция. Сначала в сценарии вызывается каждая из функций. Затем объявляется переменная tmp, и с помощью последовательно выполняемых команд tmp=first (переменная tmp ссылается на первую функцию), first=second (теперь инструкция first является ссылкой на вторую функцию) и second=tmp (инструкция second является ссылкой на первую функцию) функции first() и second() «обмениваются значениями». В результате функция first() выводит сообщение Вторая функция, а функция second() отображает в рабочем окне сообщение Первая функция. Убеждаемся в этом, вызвав функции first() и second().

В итоге результат выполнения сценария будет таким.

Результат выполнения сценария (из листинга 3.16)

```
Первая функция
Вторая функция
Вторая функция
Первая функция
```

Проверяем работу сценария с помощью следующего веб-документа:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 3.16</title>
</head>
<body><h3>Листинг 3.16</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing03_16.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Результат выполнения сценария при загрузке веб-документа в окно браузера представлен на рис. 3.16.

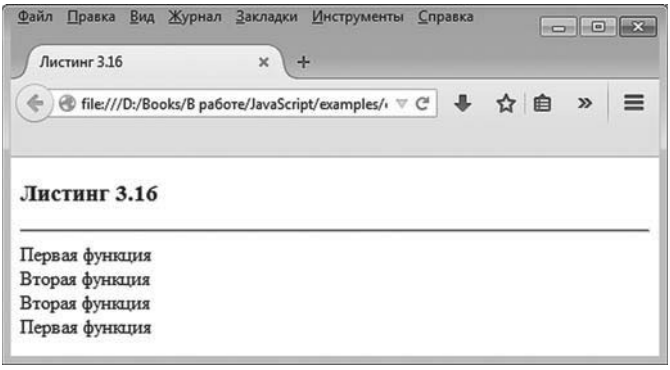

Рис. 3.16. Результат выполнения сценария, в котором функции «обмениваются значениями»

Операции присваивания, выполняемые над функциями, мы еще будем рассматривать в книге.

Анонимные функции

— Варварская игра, дикая местность — меня тянет на родину.
— Где ваша родина?
— Не знаю. Я родился на корабле, но куда он плыл и откуда — никто не помнит. А вы где родились, Жакоб?
— А я вообще еще не родился.

из к/ф «Формула любви»

Ранее каждый раз, когда мы имели дело с функциями, мы их *описывали*. Такая ситуация вполне нормальна. Но существуют и другие способы создания функций в сценарии. Один из таких способов базируется на создании *анонимной функции*. Анонимная функция — это функция, но только без имени. На первый взгляд может показаться странным и непонятным, как возможна функция без имени и зачем она нужна. Тем не менее в некоторых случаях анонимные функции весьма полезны. Поэтому имеет смысл познакомиться с тем, как такие функции создаются и что с ними разрешается делать.

Если отталкиваться от синтаксиса создания анонимной функции, то соответствующий код очень похож на код описания обычной (не анонимной) функции, только название функции не указывается. Шаб-

лон создания анонимной функции следующий (жирным шрифтом выделены ключевые элементы шаблона):

```
function(аргументы){
  // код функции
}
```

Вся приведенная выше конструкция по факту создает объект функции. Возникает естественный вопрос: что с этим объектом делать? Как минимум такой объект можно присвоить в качестве значения переменной. Также объект можно вызвать (как именно — показано немного далее).

Если объект присвоить переменной, то такая переменная становится именем функции, и с ней обращаться следует как с функцией. Приведенный ниже шаблонный код дает представление о том, как переменной присваивается в качестве значения анонимная функция:

```
var переменная=function(аргументы){
          // код функции
        }
```

Пример, в котором создается анонимная функция и присваивается значением переменной, представлен в листинге 3.17. Там же показано, как анонимная функция вызывается (без присваивания функции значением переменной).

Листинг 3.17. Обмен функций (файл Listing03_17.js)

```
// Переменной значением присваивается анонимная функция:
var f=function(msg){
      document.write(msg+"<br>")
   }
// Вызов анонимной функции чрез переменную:
f("Анонимная функция")
// Вызов анонимной функции без
// присваивания ее значением переменной:
(function(msg){
  document.write("<b>"+msg+"</b><br>")
}("Еще одна функция"))
```

Для проверки выполнения данного сценария используем веб-документ с таким HTML-кодом:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 3.17</title>
</head>
<body><h3>Листинг 3.17</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing03_17.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 3.17 показано окно браузера с загруженным в него веб-документом.

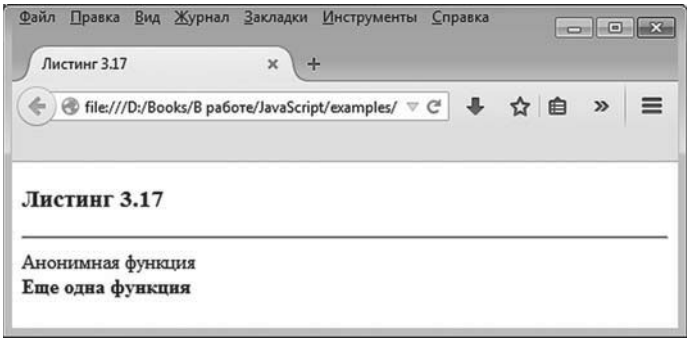

Рис. 3.17. Результат выполнения сценария с анонимными функциями

Код у сценария достаточно простой. Сначала объявляется переменная f, а значением ей присваивается анонимная функция, которая (в соответствии с кодом функции) отображает в рабочем документе значение, переданное функции аргументом. Вызываем функцию командой f("Анонимная функция"), в результате чего соответствующий текст

появляется в документе в окне браузера. Еще одна команда — пример вызова анонимной функции без предварительного присваивания этой функции значением переменной. Речь вот о чем:

```
(function(msg){
  document.write("<b>"+msg+"</b><br>")
}("Еще одна функция"))
```

Здесь имеет место описание анонимной функции, после которой в круглых скобках указан аргумент (передается при вызове функции), а вся эта конструкция заключена в круглые скобки. Анонимная функция описана так, что ею отображается значение аргумента, причем при отображении применяется жирный шрифт. Эффект от команды — как если бы мы вызвали обычную функцию с соответствующим аргументом. Недостаток данного подхода в том, что анонимную функцию указанным способом можно вызвать только один раз. Получается такая своеобразная «одноразовая» функция. Хотя иногда именно это и нужно.

ДЕТАЛИ

Чтобы понять, как все здесь устроено, удобно думать о блоке кода, которым создается анонимная функция, как об имени функции. Тогда наличие круглых скобок (с аргументом) естественным образом интерпретируется как инструкция вызова функции. Внешние круглые скобки, в которые заключается вся конструкция, нужны для того, чтобы «объединить» код создания функции и скобки с аргументом в одно целое.

На первый взгляд может показаться, что создание анонимной функции (с присваиванием ее значением переменной) не отличается от описания обычной функции. Но это не совсем так. Важное принципиальное отличие состоит в следующем. Как мы помним, при запуске сценария на выполнение предварительно создаются все описанные в нем функции, причем вне зависимости от того, в каком конкретно месте сценария функции описаны. А если функция анонимная, то создается она в том месте, где в сценарии размещена команда создания функции. Поэтому, например, в рассмотренном выше сценарии нельзя размещать команду вызова анонимной функции через переменную f до того, как функция присвоена значением переменной.

Еще один пример использования анонимных функций приведен в листинге 3.18. Там функция создается «случайным» образом.

Листинг 3.18. Создание «случайной» функции (файл Listing03_18.js)

```
// Объявляются переменные:
var k,n=6,rnd,color
// Оператор цикла:
for(k=1;k<=n;k++){
  // Генерирование случайного числа:
  rnd=Math.random()
  // Создание функции:
  if(rnd<0.2){
    color=function(){
          document.write("Красный<br>")
        }
  }
  else{
    if(rnd<0.5){
      color=function(){
            document.write("Желтый<br>")
          }
    }
    else{
      color=function(){
            document.write("Зеленый<br>")
          }
    }
  }
  // Вызов случайной функции:
  color()
}
```

Функция определяется в условном операторе. Точнее, использованы два вложенных условных оператора. В условиях проверяется

значение случайного числа rnd, равномерно распределенного в диапазоне значений от 0 до 1. Если значение числа rnd меньше 0.2 (кстати, вероятность такого события равна 0.2), то переменной color значением присваивается анонимная функция, отображающая при вызове в рабочем документе слово Красный. Если значение переменной rnd меньше 0.5 (но больше 0.2), то переменной color значением присваивается анонимная функция, отображающая в рабочем документе слово Желтый. Такое значение переменная color получает с вероятностью 0.3. Наконец, при прочих значениях переменной rnd (вероятность «прочих» значений равна 0.5) переменной color присваивается анонимная функция, отображающая при вызове в рабочем документе слово Зеленый. После присваивания значения переменной color командой color() вызывается одна из трех анонимных функций (какая именно — тайна, покрытая мраком, разгадать которую можно по результату вызова функции).

Вся эта конструкция из условных операторов «спрятана» в операторе цикла. Так что вся описанная процедура повторяется несколько раз. Результат выполнения сценария может быть таким.

Результат выполнения сценария (из листинга 3.18)

```
Зеленый
Зеленый
Красный
Красный
Желтый
Зеленый
```

Для тестирования сценария используется следующий HTML-код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 3.18</title>
</head>
<body><h3>Листинг 3.18</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing03_18.js">
```

```
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 3.18 показано, как может выглядеть веб-документ с результатом выполнения сценария (поскольку здесь имеем дело со случайными числами, то при перезагрузке страницы в окне браузера содержимое окна меняется).

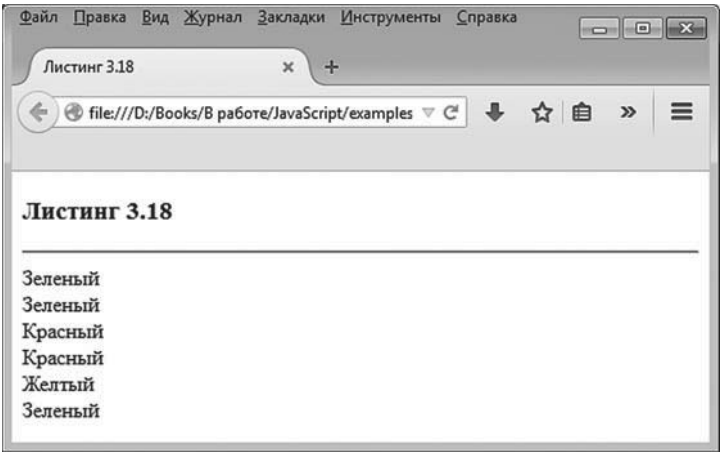

Рис. 3.18. Возможный результат выполнения сценария со случайной функцией

Понятно, что данный алгоритм, при котором в документе случайным образом отображается одно из трех слов, можно было бы реализовать намного проще. Но нас интересуют анонимные функции.

НА ЗАМЕТКУ

Использовать блоки описания обычных функций вместо команд присваивания переменной color анонимных функций — идея плохая. Если теоретически написать код с условными операторами, в которых в зависимости от некоторых условий одна и та же функция описывается разными способами, то на самом деле создана будет лишь одна такая функция — последняя описанная в сценарии. Все остальные описания игнорируются. Объяснение — в механизме создания описанных в сценарии функций. Для каждого описания со-

здается функция, и не имеет значения, в условном операторе такое описание или где-то еще. Если имеется несколько описаний одной и той же функции, в реальности будет использовано то, что обрабатывалось последним.

Функция как результат

Во всем есть своя мораль, нужно только уметь ее найти!

Л. Кэрролл «Алиса в Стране чудес»

Еще один вопрос, который рассмотрим в данной главе, — возвращение функции результатом функции. Другими словами, функция результатом может возвращать другую функцию. Откуда эта «другая» функция берется — вопрос отдельный. Для такой благородной цели можно использовать внутреннюю функцию, а результатом, например, может возвращаться анонимная функция. Именно эти два случая и рассмотрим.

В листинге 3.19 приведен пример сценария, в котором описана функция, возвращающая результатом функцию полиномиального типа (полином второго порядка). Аргументами исходной функции передаются три числовых коэффициента, определяющие полином, а результатом является функция, вычисляющая значение полинома с указанными коэффициентами в точке, определяемой аргументом функции-результата.

ДЕТАЛИ

Полиномом степени n называется зависимость вида $P(x) = a_0 + a_1x + a_2x^2 + a_nx^n$. Полином как функциональная зависимость однозначно определяется набором коэффициентов a_0, a_1, ..., a_n. Полином второй степени является зависимостью вида $P(x) = a + bx + cx^2$. В приведенном далее сценарии описывается функция. Она на основе трех аргументов (параметры a, b и c) возвращает функцию, которая для заданного аргумента x вычисляет значение $a + bx + cx^2$.

Хотя все указанное выше может показаться немного сложным, программный код на самом деле достаточно простой.

Листинг 3.19. Результатом функции возвращается анонимная функция (файл Listing03_19.js)

```
// Функция с результатом - функцией:
function makePolynom(a,b,c){
  // Результат функции:
  return function(x){
        // Результат анонимной функции:
        return a+b*x+c*x*x
      }
} // Окончание описания функции
// Переменные:
var P,Q
// Первый полином:
P=makePolynom(1,2,1)
// Второй полином:
Q=makePolynom(2,-1,1)
// Аргумент для полиномов:
var z=2
// Вычисление значений полиномов:
document.write("P("+z+") = "+P(z)+"<br>")
document.write("Q("+z+") = "+Q(z)+"<br>")
```

Функция, возвращающая результатом анонимную функцию, называется makePolynom(). У нее три аргумента — коэффициенты полинома. Результатом функции возвращается следующая конструкция:

```
function(x){
  return a+b*x+c*x*x
}
```

Здесь описана анонимная функция, которая имеет один аргумент (обозначен как x) и которая результатом возвращает значение выражения a+b*x+c*x*x. Данная анонимная функция является результатом функции makePolynom(). Вызывая функцию makePolynom() с разными аргументами, в результате получаем разные функциональные зависимости.

НА ЗАМЕТКУ

Анонимная функция на самом деле еще и внутренняя функция. При вычислении своего результата она использует аргументы внешней функции.

После описания функции makePolynom() в сценарии объявляются переменные P и Q, значения которым присваиваются, соответственно, командами P=makePolynom(1,2,1) и Q=makePolynom(2,-1,1).

ДЕТАЛИ

Функция, присвоенная значением переменной P, соответствует полиному $P(x) = 1 + 2x + x^2$. Функция, присвоенная переменной Q, соответствует полиному $Q(x) = 2 - x + x^2$. Легко проверить непосредственным вычислением, что $P(2) = 9$ и $Q(2) = 4$.

Для проверки значений полиномов используются инструкции P(z) и Q(z), где переменной z предварительно присвоено значение 2.

Ниже приведен результат выполнения сценария.

Результат выполнения сценария (из листинга 3.19)

```
P(2) = 9
Q(2) = 4
```

Для веб-документа используем следующий шаблонный код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 3.19</title>
</head>
<body><h3>Листинг 3.19</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing03_19.js">
</script>
```

```
<!-- Завершение сценария -->

</body>
</html>
```

Документ в окне браузера показан на рис. 3.19.

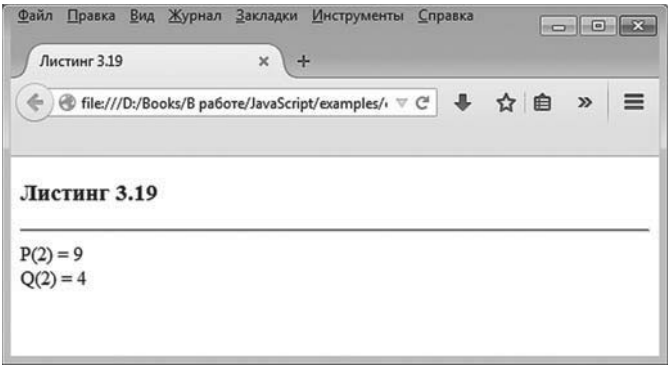

Рис. 3.19. Результат выполнения сценария с функцией, возвращающей результатом анонимную функцию

Еще один пример, который рассмотрим далее, иллюстрирует ситуацию, когда функция возвращает результатом ссылку на внутреннюю (не анонимную) функцию. А именно мы создадим функцию, которая результатом возвращает другую функцию, которая, в свою очередь, при каждом очередном вызове возвращает новое число в последовательности Фибоначчи.

НА ЗАМЕТКУ

В последовательности Фибоначчи первые два числа равны единице, а каждое следующее число равняется сумме двух предыдущих.

Обратимся к программному коду в листинге 3.20.

Листинг 3.20. Результатом функции возвращается внутренняя функция (файл Listing03_20.js)

```
// Функция результатом возвращает внутреннюю функцию:
function makeFibs(){
  // Локальные переменные внешней функции:
```

```
  var a=0,b=1
  // Результат внешней функции:
  return next
  // Внутренняя функция:
  function next(){
    // Новые значения локальных переменных
    // внешней функции:
    b=a+b
    a=b-a
    // Результат внутренней функции:
    return a
  } // Окончание описания внутренней функции
}   // Окончание описания внешней функции
// В переменную записывается результат вызова функции:
var nextFib=makeFibs()
// Генерирование чисел Фибоначчи:
for(var k=1;k<=15;k++){
  document.write(nextFib()+" | ")
}
```

В сценарии описывается внешняя функция makeFibs(), у которой нет аргументов и которая результатом возвращает функцию. Проанализируем код функции makeFibs().

В теле функции объявляются две переменные: переменная a со значением 0 и переменная b со значением 1. Командой return next результатом функции возвращается значение next, которое является именем внутренней функции. Еще имеется описание внутренней функции, и, собственно, все.

Внутренняя функция next() также не имеет аргументов, а в ее теле всего три команды: b=a+b, a=b-a и return a. Как же все это работает? Чтобы получить ответ, посмотрим, как используется функция makeFibs(). А используется она в команде nextFib=makeFibs(). При выполнении команды переменной nextFib присваивается результат вызова функции makeFibs(). Как следствие, в переменную nextFib записывается ссылка на внутреннюю функцию next() из функции makeFibs(). Но это еще не все.

При вызове функции makeFibs() создаются и получают свои начальные значения переменные a и b. Если бы функция makeFibs() не возвращала результатом внутреннюю функцию, которая использует локальные переменные a и b, то судьба последних была бы незавидной: по завершении выполнения функции makeFibs() ее локальные переменные a и b были бы удалены из памяти. Но в данном случае этого не происходит. Причина именно в том, что переменная nextFib получила ссылку на внутреннюю функцию next(), которая при выполнении обращается к переменным a и b. Поэтому данные переменные (как и внутренняя функция next()) продолжают свое существование и после завершения выполнения функции makeFibs().

ДЕТАЛИ

В несколько упрощенном виде ситуация выглядит примерно так. При вызове функции создается объект, содержащий всю информацию, касающуюся функции, в том числе и ее локальные переменные. По завершении выполнения функции объект обычно из памяти удаляется. Но если после завершения выполнения функции в программе остались активные ссылки на свойства (локальные переменные) объекта, объект остается в памяти. Отсюда и возможность использовать локальные переменные внешней функции в теле внутренней функции (да и саму внутреннюю функцию) даже после того, как внешняя функция завершила выполнение.

Теперь каждый раз при выполнении инструкции nextFib() выполняется код функции next(), которая изменяет значения переменных a и b. Например, при первом вызове функции next() переменная b получает значение 1, переменная a получает значение 1, и значение переменной a возвращается результатом. После второго вызова функции next() переменная b получает значение 2, переменная a получает значение 1, и оно возвращается результатом. При третьем вызове переменная b получает значение 3, переменная a получает значение 2 (результат функции), и так далее.

ДЕТАЛИ

Чтобы увидеть во всем этом систему, следует учесть, что сразу после первого вызова функции next() в переменные a и b записаны два последних числа в последовательности Фибоначчи и каждый новый вызов функции проводит к «смещению» на одну позицию вправо в последовательности. Так, если в какой-то момент a и b со-

держат предпоследнее и последнее число последовательности, то после выполнения команды b=a+b в переменную b записывается следующее число в последовательности. После выполнения команды a=b-a в переменную a записывается то значение, которое ранее было записано в переменную b (последнее число становится предпоследним).

В сценарии запускается оператор цикла, в котором за каждый цикл вызывается функция nextFib(). Результат выполнения сценария такой.

Результат выполнения сценария (из листинга 3.20)

```
1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233 | 377 | 610 |
```

Работу сценария проверяем с помощью веб-документа, код которого приведен ниже:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 3.20</title>
</head>
<body><h3>Листинг 3.20</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing03_20.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 3.20 показано, как выглядит данный документ в окне браузера.

Этой небольшой зарисовкой мы завершаем главу, посвященную функциям. Функции, на самом деле, обсуждать можно очень долго. Но, чтобы продолжить данную тему, разумно сначала познакомиться с объектами и принципами реализации объектно-ориентированного программирования в JavaScript.

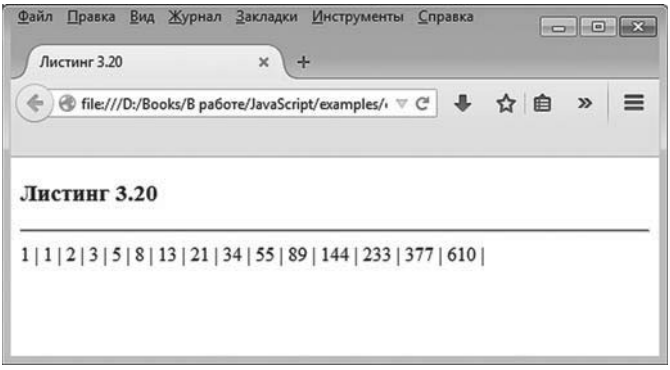

Рис. 3.20. Результат генерирования чисел Фибоначчи с помощью последовательных вызовов функции

Резюме

> Всё, что нажито непосильным трудом, всё же погибло!
>
> *из к/ф «Иван Васильевич меняет профессию»*

Подведем краткие итоги.

- Функция — блок кода, который можно вызвать по имени. У функции могут быть аргументы, и функции могут возвращать результат. В описании функции используется ключевое слово function, после которого указывается имя функции, список аргументов и тело функции с командами (заключаются в фигурные скобки). Для возвращения значения в функции используют ключевое слово return.
- Функция может быть описана в любом месте сценария. Команды использования функции могут в сценарии находиться до описания функции. При запуске сценария перед выполнением команд создаются все функции, описанные в сценарии.
- Переменные, объявленные в теле функции с ключевым словом var, являются локальными и доступны только в теле функции. Аргументы функции имеют силу локальных переменных. Если локальная и глобальная переменные имеют одинаковые названия, то в теле функции приоритет остается за локальной переменной. Для получения доступа к внешним (глобальным) переменным в таком случае используют ссылку window на объект окна.

- Аргументы функциям передаются по значению: на самом деле создаются технические копии аргументов, которые и используются при вычислениях. Аргументом функции допускается передавать, кроме прочего, функцию. При вызове функции количество аргументов может не совпадать с тем количеством аргументов, которое было указано при описании функции.
- Для проверки типов переменных и аргументов используют оператор typeof. Результатом является текстовая строка, определяющая тип значения, на которое ссылается переменная.
- При рекурсивном определении функции в описании функции содержится инструкция вызова этой же самой функции, но обычно с другим аргументом.
- Функция может содержать описание другой функции. Первая называется внешней, а вторая — внутренней. Внутренняя функция имеет доступ к аргументам внешней функции и ее локальным переменным (называется замыканием). Внешняя функция не имеет доступа к локальным переменным внутренней функции.
- Функция может быть присвоена в качестве значения переменной, а имени функции может быть присвоено значение, отличное от функции.
- Анонимная функция создается как обычная функция, но только имя для нее не указывается. Анонимная функция может быть присвоена значением переменной или вызвана (единожды) без присваивания ее значением переменной. В отличие от обычной функции, анонимная функция создается не перед началом выполнения сценария, а при выполнении соответствующей команды в сценарии.
- Результатом функции, кроме прочего, может быть функция.

Часть II
JAVASCRIPT И ООП

Глава 4
ЗНАКОМСТВО С ОБЪЕКТАМИ И ПРИНЦИПЫ ООП

> Лучший способ объяснить — это самому сделать.
>
> *Л. Кэрролл «Алиса в Стране чудес»*

Мы начинаем знакомство с принципами *объектно-ориентированного программирования* (сокращенно *ООП*). Хотя, если быть более точным, речь пойдет о том, как принципы ООП реализуются в JavaScript. Сразу следует отметить, что реализация довольно специфическая, так что будет много сюрпризов.

НА ЗАМЕТКУ

Особенно тяжело придется тем, кто знаком с такими «классическими» языками программирования, как, например, Java, C++ или C#. Чтобы осознать принципиальное отличие языка JavaScript от других языков в вопросе реализации принципов ООП, достаточно отметить, что в JavaScript есть объекты, но нет классов (в их традиционном понимании).

Прежде чем перейти к обсуждению особенностей языка JavaScript, имеет смысл кратко обсудить непосредственно парадигму ООП.

Концепция ООП

> Не надо меня щадить: пусть самая страшная — но правда.
>
> *из к/ф «Бриллиантовая рука»*

Объектно-ориентированный подход в программировании возник как ответ на проблему неуклонно возрастающих объемов программных кодов. Сразу следует отметить, что речь идет в первую очередь о чело-

веческом факторе. При увеличении размера программы становится все сложнее удержать «главную сюжетную линию». Камнем преткновения становится механизм связывания в программе данных, предназначенных для обработки, и программного кода, призванного такую обработку выполнять. Если перейти на образный язык и сравнить процесс написания программы с постройкой здания, то при постройке небоскреба желательно использовать более продвинутую технологию, чем выкладывание стенок и перегородок из кирпичей. Разумнее использовать уже готовые блоки. Причем блоки должны быть функциональными — например, с окнами, дверьми, разъемами для выключателей, с предусмотренной системой креплений и другими полезными штуками. Аналогом такого блока, из которого строится здание, в объектно-ориентированном программировании является объект. В объект «спрятаны» различные переменные (в широком смысле этого понятия) и функции, предназначенные для обработки значений этих переменных. Функции, относящиеся к объекту, обычно называют *методами* объекта, а переменные, хранящиеся в объекте, называются его *полями* или *свойствами*. Программа в таком случае представляет собой некоторый набор инструкций, которыми создаются объекты, а эти объекты взаимодействуют между собой и с «окружающим миром» через методы.

Помимо объектов, существует такое понятие как *класс*. Класс на самом деле представляет собой некий шаблон, на основе которого создается объект. На основе одного и того же класса может создаваться произвольное количество объектов. Объекты, созданные на основе одного класса, хотя и являются физически разными, имеют одинаковые наборы свойств и методов. Такой «классовый» подход используется во многих языках программирования — например, в C++, C# и Java. Но не в JavaScript. В JavaScript классов нет. Но объекты есть. Как же они создаются?

Во-первых, в JavaScript достаточно просто создать объект, что называется, «с нуля» (как это делается, мы узнаем чуть позже). Во-вторых, в JavaScript есть много встроенных объектов, которые не нужно создавать. Наконец, можно создавать новые объекты на основе существующих объектов (встроенных или созданных в программе) путем *наследования*. В этом случае создаваемый объект повторяет структуру и содержание исходного объекта. Объект, на основе которого создается другой объект, называется *родительским* и служит как бы прототипом для своего потомка (который, кстати, называется *дочерним* объектом). Такой тип наследования называется *прототипным*.

НА ЗАМЕТКУ

В таких языках программирования, как Java, C++ и C#, наследование реализовано иначе, на уровне классов. Наследуются не непосредственно объекты, а классы. А объект уже создается на основе класса-наследника.

Изложенное выше позволяет составить некоторое представление о том, как принципы ООП соотносятся с реалиями языка программирования JavaScript. Прикладное же значение имеют более прозаичные и приземленные темы: как создать объект, как его использовать, каковы его особенности. С этого и начнем наше изучение ООП.

Создание объектов

> Начинаю действовать без шума и пыли по вновь утвержденному плану.
>
> *из к/ф «Бриллиантовая рука»*

В JavaScript существует несколько способов создания объектов. Мы рассмотрим основные. Но, прежде чем приступить к разбору кодов, зададимся вопросом: что же такое объект, если посмотреть на него строго с прагматической точки зрения? В целом это набор или объединение некоторых переменных (которые, напомним, называются *свойствами*) и функций (называются *методами* объекта). Прежде чем приступить к созданию объекта, необходимо для себя решить, какие свойства и/или методы у него должны быть. Допустим, мы хотим создать объект для хранения персональных данных пользователя. Методы пока что включать в объект не будем, ограничимся лишь свойствами. Какие свойства могут быть у такого объекта? Начнем с простых характеристик: имени пользователя и его возраста. Для записи имени пользователя в объекте предусмотрим свойство `name`, а для записи возраста пользователя используем свойство `age`.

Литерал объекта

Наиболее простой способ создания объекта состоит в описании *литерала объекта*. Литерал объекта — список, заключенный в фигурные скобки. Внутри фигурных скобок через запятую указываются пары

значений: название свойства и значение свойства. Название свойства и значение разделяются двоеточиями. Если литерал объекта присвоить в качестве значения переменной, то эта переменная может быть отождествлена с объектом.

Таким образом, шаблон создания объекта путем описания его литерала имеет следующую структуру:

```
var переменная={свойство:значение,свойство:значение,...}
```

После того как объект создан, к его свойствам обращаются (для считывания значения свойства или изменения значения свойства) в «точечном» формате: указывается имя объекта и через точку название свойства. Небольшой пример, в котором создается объект obj со свойствами name и age, представлен в листинге 4.1.

Листинг 4.1. Создание объекта с помощью литерала объекта (файл Listing04_01.js)

```
// Создание объекта obj со свойствами name и age:
var obj={name:"Иван Петров",age:38}
// Отображение значений свойств объекта:
document.write("<b>Имя</b>: "+obj.name+"<br>")
document.write("<b>Возраст</b>: "+obj.age+"<br>")
// Новые значения свойств объекта:
obj.name="Петр Иванов"
obj.age++
// Отображение новых значений свойств объекта:
document.write("<b>Имя</b>: "+obj.name+"<br>")
document.write("<b>Возраст</b>: "+obj.age+"<br>")
```

Объект obj в сценарии создается командой var obj={name:"Иван Петров",age:38}, которую на самом деле можно рассматривать как объединение двух команд: команды объявления переменной obj и команды присваивания переменной obj в качестве значения литерала объекта. С объявлением переменной obj все просто и традиционно: используем ключевое слово var и указываем имя переменной. Что касается литерала объекта, то здесь тоже ничего особо сложного нет. Так, в фигурных скобках через запятую указаны две инструкции: name:"Иван Петров" и age:38. Инструкция name:"Иван Петров" означает, что у объекта (который созда-

ется с помощью литерала) имеется свойство name, а значение этого свойства равно "Иван Петров". Инструкция age:38 определяет для объекта свойство age со значением 38. Такой вот создается объект. Ссылка на этот объект записывается в переменную obj. Поэтому через переменную obj мы можем обращаться к объекту. Если нам нужно получить доступ к свойству name, используем инструкцию obj.name. Если мы хотим получить доступ к свойству age, используем инструкцию obj.age. Такие инструкции использованы в командах document.write("Имя: "+obj.name+"
") и document.write("Возраст: "+obj.age+"
"), которыми в рабочем документе отображаются значения свойств объекта.

После того как объект создан и определены его свойства и их значения, последние можно изменить. Другими словами, значения свойств объекта могут меняться в процессе выполнения программы.

НА ЗАМЕТКУ

Откровенно говоря, меняться могут не только значения свойств, но и сам набор свойств объекта: у объекта при выполнении сценария могут появляться новые свойства и исчезать старые. Но об этом позже.

Изменяются значения свойств объекта точно так же, как изменяются значения обычных переменных. Так, командой obj.name="Петр Иванов" свойству name объекта obj присваивается новое значение "Петр Иванов", а командой obj.age++ значение свойства age этого же объекта увеличивается на единицу. Новые значения свойств объекта obj отображаются в рабочем документе командами document.write("Имя: "+obj.name+"
") и document.write("Возраст: "+obj.age+"
").

Чтобы увидеть результат выполнения данного сценария, воспользуемся веб-документом с таким HTML-кодом:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 4.1</title>
</head>
<body><h3>Листинг 4.1</h3><hr>

<!-- Начало сценария -->
```

```
<script type="text/javascript" src="Listing04_01.js">
</script>
<!-- Завершение сценария -->


</body>
</html>
```

На рис. 4.1 показано, как выглядит результат выполнения сценария в рабочем окне браузера.

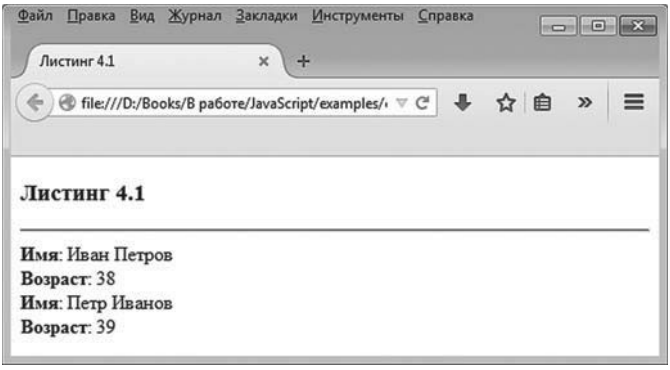

Рис. 4.1. Результат выполнения сценария, в котором создается объект путем описания литерала объекта

Что хочется сказать по поводу рассмотренного нами кода? Его имеет смысл усовершенствовать, создав специальную функцию для отображения свойств объекта. В этом случае сценарий мог бы выглядеть так, как показано в листинге 4.2.

Листинг 4.2. Использование функции для отображения свойств объекта (файл Listing04_02.js)

```
// Создание объекта obj со свойствами name и age:
var obj={name:"Иван Петров",age:38}
// Отображение значений свойств объекта:
show(obj)
// Новые значения свойств объекта:
obj.name="Петр Иванов"
obj.age++
```

```
// Отображение новых значений свойств объекта:
show(obj)
// Функция для отображения свойств объекта:
function show(a){
  document.write("<b>Имя</b>: "+a.name+"<br>")
  document.write("<b>Возраст</b>: "+a.age+"<br>")
}
```

Результат выполнения данного сценария точно такой же, как и в предыдущем случае.

> **НА ЗАМЕТКУ**
>
> Желающие могут проверить это самостоятельно, воспользовавшись шаблонным HTML-документом, в котором достаточно лишь заменить ссылку на загружаемый в документ файл со сценарием.

Вместо явного использования метода write() мы в сценарии описали функцию show(), у которой один аргумент (который обозначен как a). Предполагается, что аргументом функции передается объект, у которого есть свойства name и age. Инструкции a.name и a.age использованы в аргументе метода write(), который вызывается из объекта document для отображения в рабочем документе значений свойств объекта, переданного аргументом функции show(). В тех местах сценария, где нужно отобразить в документе значения свойств объекта obj, используется команда show(obj).

Функция — это хорошо. Но можно пойти дальше и, вместо того чтобы описывать функцию для отображения свойств объекта, добавить в объект соответствующий метод.

> **НА ЗАМЕТКУ**
>
> При вызове функции ей необходимо передать аргументом объект. Если аргумент передать некорректный, возможны проблемы. Немного иная ситуация, когда вместо функции используется метод. Методу не нужно передавать аргумент, и метод вызывается из объекта, что автоматически обеспечивает корректность доступа к свойствам объекта.

Объект с методом

Метод добавить в объект так же легко, как и обычное свойство. Аналогом названия свойства служит имя метода, а значением является анонимная функция. В листинге 4.3 приведен пример сценария, в котором процесс отображения свойств объекта в рабочем документе реализуется с помощью специального метода объекта.

Листинг 4.3. Создание объекта с методом (файл Listing04_03.js)

```
// Создание объекта obj со свойствами name и age,
// а также методом show():
var obj={name:"Иван Петров",
      age:38,
      show:function(){
        document.write("<b>Имя</b>: "+this.name+"<br>")
        document.write("<b>Возраст</b>: "+this.age+"<br>")
      }
}
// Отображение значений свойств объекта:
obj.show()
// Новые значения свойств объекта:
obj.name="Петр Иванов"
obj.age++
// Отображение новых значений свойств объекта:
obj.show()
```

Проверить работу сценария нам поможет веб-документ с таким HTML-кодом:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 4.3</title>
</head>
<body><h3>Листинг 4.3</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing04_03.js">
```

```
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Как документ выглядит в окне браузера, показано на рис. 4.2.

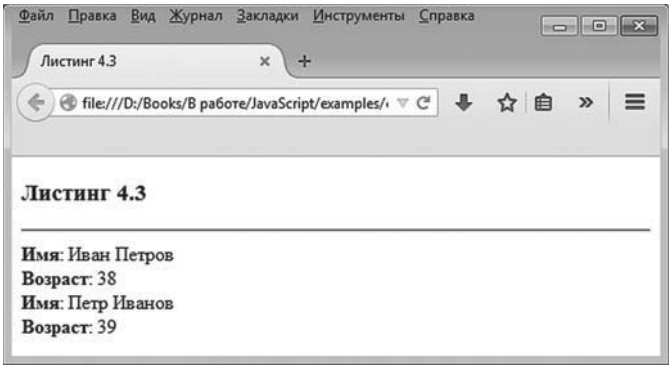

Рис. 4.2. Результат выполнения сценария, в котором создается объект со свойствами и методом

Несложно заметить, что результат выполнения сценария не изменился. Но только теперь, в отличие от предыдущих случаев, всю работу по отображению свойств объекта obj выполняет метод объекта show(). У метода show() нет аргументов, и вызывается он из объекта obj. Поэтому команда вызова метода имеет вид obj.show(). Ответим теперь на вопрос о том, как вообще у объекта obj появился метод show(). Несложно заметить, что в литерале объекта, помимо описания свойств name и age, имеется еще и описание «свойства» show, причем значением «свойства» указана следующая анонимная функция:

```
show:function(){
  document.write("<b>Имя</b>: "+this.name+"<br>")
  document.write("<b>Возраст</b>: "+this.age+"<br>")
}
```

Несложно догадаться, что именно здесь объекту добавляется метод. В некотором смысле добавление метода в объект эквивалентно добавлению свойства, значением которого является функция. Посколь-

ку функцию можно вызывать, соответствующее свойство тоже можно вызвать. В результате объект приобретает метод, а имя свойства отождествляется с именем метода.

НА ЗАМЕТКУ

Если речь идет о вызове метода, то после имени метода указываются круглые скобки (и в случае необходимости в круглых скобках передаются аргументы методу). Само по себе имя метода можно интерпретировать как свойство объекта, значением которого вызывается функция (позднее мы узнаем, что речь идет об объекте функции).

В теле анонимной функции, которая указывается значением для «свойства» show, имеются две инструкции, которые стоит прокомментировать. Речь о выражениях this.name и this.age, в которых использовано ключевое слово this.

НА ЗАМЕТКУ

Ранее в соответствующих местах кода, в том месте, где мы указали ключевое слово this, помещалось имя объекта.

Ключевое слово this является ссылкой на объект, из которого вызывается метод. В данном случае инструкция this.name означает, что выполняется обращение к свойству name того объекта, из которого вызывается метод. Аналогично, инструкция this.age представляет собой обращение к свойству age объекта, из которого вызывается метод.

ДЕТАЛИ

Поскольку мы пока что имеем дело лишь с одним объектом obj, не совсем очевидно, почему необходимо использовать в теле анонимной функции (метод объекта) ключевое слова this. Например, поскольку анонимная функция указывается значением в описании объекта obj, можно было предположить, что в описании кода функции вместо инструкций this.name и this.age можно было использовать соответственно obj.name и obj.age. Но это плохой вариант. Дело в том, что инструкция с ключевым словом this идентифицирует объект, из которого вызывается метод, в то время как инструкция с ключевым словом obj указывает на один и тот же совершенно конкретный объект. Пока речь идет об одном-единственном объекте (которым является obj), проблемы мо-

гут и не возникнуть. Но представим, что в процессе выполнения сценария имеет место присваивание объектов (например, выполняется команда newObj=obj). Если так, то метод может вызываться из объекта, отличного от obj (например, из объекта newObj). И если в теле метода использована инструкция this, то «смена» объекта автоматически учитывается при вызове метода и метод обращается к свойствам объекта вызова. Если же в теле метода использована явная ссылка на объект (в нашем случае obj), то при вызове метода из других объектов выполняется обращение к свойствам объекта obj. Это как минимум не очень удачный стиль программирования.

Присваивание объектов

В приведенных выше примерах объект создавался всего один. Далее рассмотрим небольшую иллюстрацию к тому, как объект присваивается переменной в качестве значения. Эта тривиальная, казалось бы, операция приводит к довольно неожиданным последствиям.

Итак, рассмотрим программный код, представленный в листинге 4.4.

Листинг 4.4. Присваивание объекта значением переменой (файл Listing04_04.js)

```
// Создание объекта:
var objA={name:"Иван Петров",
      age:38,
      show:function(){
        document.write("<u>Имя</u>: "+this.name+"<br>")
        document.write("<u>Возраст</u>: "+this.age+"<br>")
      }
}
// Отображение значений свойств объекта objA:
document.write("<b>Объект objA:</b><br>")
objA.show()
// Присваивание объекта значением переменной:
var objB=objA
// Отображение значений свойств объекта objB:
document.write("<b>Объект objB:</b><br>")
objB.show()
```

```
// Новые значения свойств объекта objA:
objA.name="Петр Иванов"
objA.age++
// Отображение новых значений свойств объекта objA:
document.write("<b>Объект objA:</b><br>")
objA.show()
// Отображение значений свойств объекта objB:
document.write("<b>Объект objB:</b><br>")
objB.show()
```

В сценарии создается объект objA со свойствами name и age, а также методом show().

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

В методе show() в отображаемом текстовом значении использованы дескрипторы <u> и </u>. Выделенный с помощью данных дескрипторов текст в окне браузера отображается с подчеркиванием.

С помощью команды objA.show() после создания объекта objA значения его свойств отображаются в рабочем документе. Затем объявляется переменная objB, и значением переменной присваивается объект objA (команда objB=objA). После выполнения команды присваивания командой objB.show() отображаются значения свойств объекта, на который ссылается переменная objB. Можно ожидать (и эти ожидания оправдываются), что значения свойств будут такими же, как и у объекта objA.

На следующем этапе изменяются значения свойств объекта objA (команды objA.name="Петр Иванов" и objA.age++), а после внесения изменений в значения свойств объекта objA командами objA.show() и objB.show() проверяются значения свойств объектов objA и objB. Относительно объекта objA особой интриги нет — значения свойств изменились так, как мы их изменили. Проверка это подтвердит. С объектом objB нас ожидает сюрприз: значения его полей тоже изменились, причем строго так, как изменились значения свойств объекта objA.

НА ЗАМЕТКУ

Более того, на самом деле речь идет об одном и том же объекте.

Чтобы проверить результат выполнения сценария, воспользуемся следующим веб-документом:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 4.4</title>
</head>
<body><h3>Листинг 4.4</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing04_04.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 4.3 показан результат выполнения сценария, отображаемый в веб-документе в окне браузера.

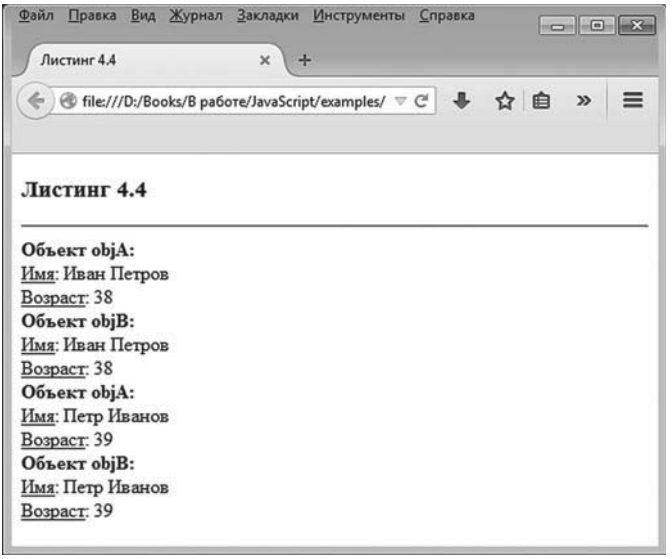

Рис. 4.3. Результат выполнения сценария, в котором переменной значением присваивается объект

Главное наблюдение, которое мы должны сделать, состоит в том, что при изменении значения свойств объекта objA синхронно изменяются значения свойств объекта objB. Объяснение очень простое, и связано оно со способом получения доступа к объектам через переменные. Дело в том, что в языке JavaScript переменная *ссылается* на объект. Ситуацию легко понять, если представить, что реальное значение переменной, которую мы отождествляем с объектом, — это не сам объект, а его адрес. Просто когда мы обращаемся через переменную к свойству объекта или методу, то соответствующая команда обрабатывается так, как если бы речь шла непосредственно об объекте. А когда одной переменной в качестве значения присваивается другая переменная (ссылающаяся на объект), то копирование значений выполняется на уровне «адресов». В результате переменная, которой присваивалось значение, получит адрес того же объекта, что и присваиваемая переменная. В результате получается, что обе переменные ссылаются на самом деле на один и тот же объект. Проще говоря, когда в сценарии выполняется команда objB=objA, то новый объект *не создается*, и в результате переменная objB будет ссылаться на тот же самый объект, на который ссылается переменная objA.

Добавление свойств и методов в объект

Свойства и методы объекта не обязательно указывать сразу при его создании. Уже после создания объекта ему могут быть добавлены (или удалены — но это мы обсудим позже) свойства и методы. Например, вполне приемлем подход, когда сначала создается пустой объект (объект, в котором не описаны свойства и методы), а уже затем у него появляются свойства и методы. Для добавления в объект свойства достаточно присвоить этому свойству значение. Сказанное буквально означает, что если мы попытаемся присвоить значение несуществующему свойству объекта, то такое свойство в объект будет добавлено (и ему будет присвоено значение в соответствии с командой присваивания). Данное замечание относится и к добавлению методов в объект. Просто при добавлении метода имя метода следует интерпретировать как свойство объекта, а значением такому свойству присваивается функция.

Приведем небольшой пример, который в общем-то дублирует рассмотренную ранее ситуацию с созданием объекта, но только теперь свойства и метод в объект добавляются не путем включения их в ли-

терал объекта, а с помощью отдельных команд уже после того, как объект создан. Рассмотрим листинг 4.5, в котором представлен код сценария.

Листинг 4.5. Создание пустого объекта с последующим добавлением свойств и метода (файл Listing04_05.js)

```
// Создание пустого объекта:
var obj={}
// Добавление свойства name:
obj.name="Иван Петров"
// Добавление свойства age:
obj.age=38
// Добавление метода show():
obj.show=function(){
  document.write("<b>Имя</b>: "+this.name+"<br>")
  document.write("<b>Возраст</b>: "+this.age+"<br>")
}
// Отображение значений свойств объекта:
obj.show()
```

Для проверки работы сценария используем такой HTML-код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 4.5</title>
</head>
<body><h3>Листинг 4.5</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing04_05.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 4.4 показано окно браузера с результатом выполнения сценария в веб-документе.

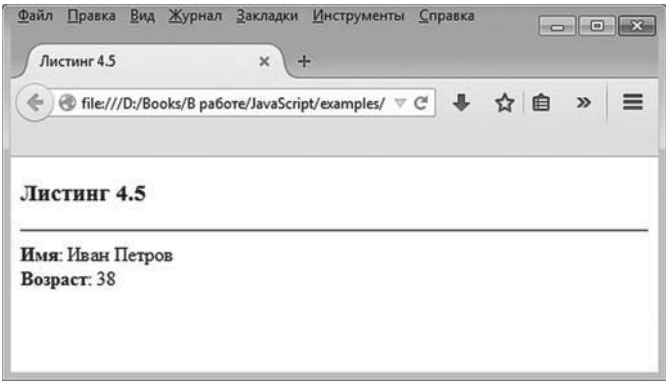

Рис. 4.4. Результат выполнения сценария, в котором создается пустой объект, а свойства и метод добавляются в объект после создания

В качестве исходного объекта совсем не обязательно должен быть именно пустой объект. Вполне подойдет любой объект с некоторыми свойствами и методами. По мере необходимости в объект можно добавить новые свойства и методы описанным выше способом.

Конструктор объектов

Само собой разумеется, что с помощью описания литералов мы можем создавать объекты практически в неограниченном количестве. Другими словами, никто не запрещает нам создавать объекты в индивидуальном порядке, описывая для каждого объекта его литерал. Но если объектов много, а набор свойств и методов у них одинаков, то процедура создания объектов будет достаточно утомительной. Поэтому при создании однотипных объектов (то есть объектов, имеющих одинаковые наборы свойств) имеет смысл использовать некое подобие конвейера. Более конкретно, существует возможность описать специальную функцию, которая позволит создавать объекты с одинаковым набором свойств. Такая функция называется *конструктором объектов*. Функция-конструктор по большому счету определяет тип объекта, набор его свойств и методов. Для создания нового объекта с помощью функции-конструктора данная функция вызывается с оператором `new`. Инструкция вызова функции (если необходимо, то с аргументами в круглых скобках) указывается после оператора `new`.

Действие оператора new состоит в том, что создается объект, а затем вызывается функция-конструктор, и в эту функцию автоматически передается ссылка this на созданный объект. Результатом такого выражения с оператором new и вызовом функции-конструктора возвращается ссылка на созданный объект. Таким образом, если мы объявляем некоторую переменную и хотим ей в качестве значения присвоить объект, созданный с помощью конструктора, то вся команда может выглядеть так:

```
var переменная=new конструктор(аргументы)
```

Роль функции-конструктора, таким образом, сводится к определению свойств и методов создаваемого объекта. Аргументами конструктору обычно передаются значения для присваивания свойствам создаваемого объекта. Ссылка на объект в теле функции-конструктора выполняется с помощью ключевого слова this.

Небольшой пример, в котором описывается функция-конструктор, а затем используется для создания объектов, представлен в листинге 4.6.

Листинг 4.6. Использование функции-конструктора для создания объектов (файл Listing04_06.js)

```
// Функция - конструктор объектов:
function Fellow(name,age){
  // Значение свойства name:
  this.name=name
  // Значение свойства age:
  this.age=age
  // Метод show():
  this.show=function(){
    document.write("<b>Имя</b>: "+this.name+"<br>")
    document.write("<b>Возраст</b>: "+this.age+"<br>")
  }
}
// Создание объектов с помощью конструктора:
var objA=new Fellow("Иван Петров",38)
var objB=new Fellow("Петр Иванов",39)
// Проверка значений полей объектов:
```

```
objA.show()
objB.show()
```

Работу сценария проверим с помощью следующего веб-документа:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 4.6</title>
</head>
<body><h3>Листинг 4.6</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing04_06.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Результат выполнения сценария представлен на рис. 4.5.

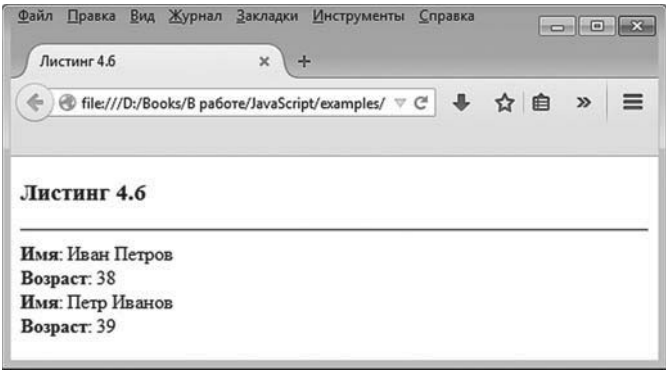

Рис. 4.5. Результат выполнения сценария, в котором объекты создаются с помощью функции-конструктора

Проанализируем программный код сценария. Функция-конструктор `Fellow()` описана с двумя аргументами `name` и `age`. Предполагается, что

это значения, которые следует присвоить одноименным свойствам создаваемого объекта. Именно так и случается при выполнении команд this.name=name и this.age=age. Здесь уместно напомнить, что при вызове функции-конструктора объект уже создан, но свойств и методов, определенных пользователем, у него нет. Ссылка на объект дается ключевым словом this. Поэтому инструкция this.name представляет собой обращение к свойству, которого у объекта нет. А мы уже знаем, что если несуществующему свойству объекта присвоить значение, то такое свойство у объекта появится. При этом просто инструкция name является аргументом функции-конструктора. В результате выполнения команды this.name=name у созданного объекта появляется свойство name, и значением свойству присваивается первый аргумент (обозначен как name) функции-конструктора. Аналогичная ситуация имеет место при выполнении команды this.age=age.

НА ЗАМЕТКУ

То, что аргументы функции-конструктора называются так же, как и свойства объекта, — совпадение. В общем случае нет необходимости, чтобы названия аргументов совпадали с названиями свойств. Но это и не запрещено. Если названия совпадают, то намного проще понять, какой аргумент для какого свойства предназначен.

Помимо двух свойств, у объекта появляется еще и метод show(). Для определения метода «свойству» show (полная инструкция выглядит как this.show) значением присваивается анонимная функция, код которой и будет выполняться при вызове метода.

НА ЗАМЕТКУ

В теле анонимной функции используется инструкция this. Там эта инструкция обозначает объект, из которого впоследствии будет вызываться метод.

После описания функции-конструктора Fellow() командами var objA=new Fellow("Иван Петров",38) и var objB=new Fellow("Петр Иванов",39) создаются два объекта objA и objB. Причем теперь это два разных объекта. Они имеют одинаковые наборы свойств и методов, но значения свойств этих объектов различны, а метод show() при вызове из разных объектов обращается именно к тому объекту, из которого вызван. Результат проверяем командами objA.show() и objB.show().

НА ЗАМЕТКУ

Пустой объект можно создать с помощью инструкции new Object(). Выполнение инструкции приводит к созданию нового пустого объекта, а результатом инструкции возвращается ссылка на созданный объект. Поэтому если подобную инструкцию присвоить в качестве значения переменной, то эта переменная будет ссылаться на объект. Методы работы с конструктором объектов Object мы рассмотрим немного позже.

Утилиты для работы с объектами

— Ахтунг! Я еще на спортивных кубках гравировал имена чемпионов.

— Гравировать имена победителей — работа, требующая самоотречения.

из к/ф «Покровские ворота»

Основные операторы мы уже обсуждали в начале книги. Но есть еще несколько важных синтаксических конструкций, которые используются в основном при работе с объектами. Их далее и рассмотрим.

Оператор with

При обращении к свойствам и методам объекта объект, как мы знаем, указывается перед именем свойства или метода. Если таких обращений (к разным свойствам и методам одного и того же объекта) много, то синтаксис команд можно несколько сократить. В этом случае используется оператор with. И хотя в современном стандарте JavaScript не рекомендуется использовать оператор with, для понимания принципов работы существующих кодов полезно знать, как он выполняется.

Синтаксис оператора следующий (жирным шрифтом выделены ключевые элементы кода):

```
with(объект){
  // команды
}
```

Конструкция достаточно простая: после ключевого слова with указывается в круглых скобках объект, а в фигурных скобках — команды.

Благодаря тому что команды находятся в with-блоке, в тех местах, где должен указываться объект, данный объект можно не указывать. Небольшой пример с использованием оператора with представлен в листинге 4.7.

Листинг 4.7. Использование оператора with (файл Listing04_07.js)

```
with(document){
  write("<h4>Знакомимся с песиком</h4>")
  var dog={name:"Рекс",breed:"Овчарка",age:3}
  with(dog){
    write("Кличка: "+name+"<br>")
    write("Порода: "+breed+"<br>")
    write("Возраст: "+age+"<br>")
  }
}
```

Весь программный код заключен в with-блок, в котором в качестве ссылки на объект указано ключевое слово document. Благодаря этому при вызове метода write() объект документа document можно не указывать. Внутри данного with-оператора, кроме команд с вызовом метода write(), имеется команда создания объекта dog с тремя свойствами, которые называются name, breed и age. Обращение к данным свойствам объекта dog в общем случае выполняется соответственно в формате dog.name, dog.breed и dog.age. Но, поскольку мы используем еще один оператор with со ссылкой на объект dog, обращение к свойствам этого объекта выполняется в упрощенной форме, без явного указания объекта dog.

Для проверки корректности работы сценария используем HTML-код, представленный ниже:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 4.7</title>
</head>
<body><h3>Листинг 4.7</h3><hr>

<!-- Начало сценария -->
```

```
<script type="text/javascript" src="Listing04_07.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Результат выполнения сценария представлен на рис. 4.6.

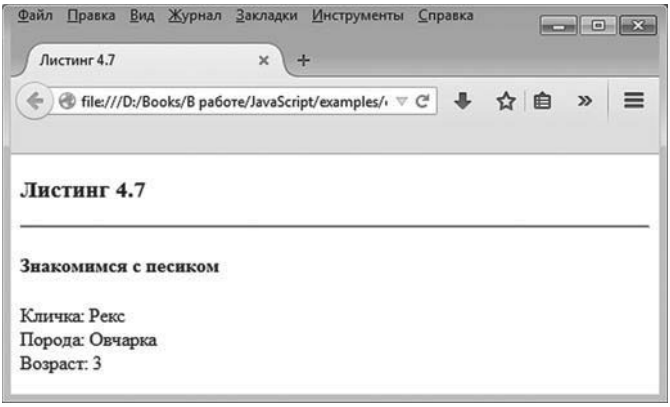

Рис. 4.6. Результат выполнения сценария с операторами *with*

НА ЗАМЕТКУ

Оператор with иногда упрощает жизнь, но во многих случаях существуют более простые способы избежать громоздких выражений. Скажем, вместо того чтобы вызывать каждый раз метод write() в формате document.write(), можно объявить некоторую переменную (например, d), присвоить ей значением ссылку на объект document (команда d=document) и затем метод write() вызывать в формате d.write().

Оператор for-in

Нередко оператор for-in позиционируют как одну из версий оператора цикла for. Даже если и так, то это «очень особая» версия. В принципе речь идет об операторе цикла, однако алгоритм его выполнения существенно отличается от традиционного оператора цикла for. Если оператор цикла for во многом универсален и имеет достаточно широкую область применения, то оператор цикла for-in, как правило,

используют для перебора по множеству свойств (и методов) объекта. Синтаксис оператора цикла for-in следующий (жирным шрифтом выделены ключевые элементы шаблона):

```
for(переменная in объект){
  // команды
}
```

После ключевого слова for в круглых скобках указывается переменная (ее можно прямо там объявить с ключевым словом var), затем ключевое слово in и, наконец, ссылка на некоторый объект. Затем в фигурных скобках размещаются команды, выполняемые в операторе цикла.

Выполняется оператор цикла for-in следующим образом: переменная, указанная в круглых скобках после ключевого слова for, последовательно принимает значения *названий* свойств (включая методы) в объекте, указанном после ключевого слова in. При каждом фиксированном значении переменной выполняются команды в теле оператора цикла (команды в фигурных скобках).

ДЕТАЛИ

Как мы узнаем далее, кроме тех методов и свойств, которые объект получает «в явном виде» при создании или в процессе выполнения, часть свойств объекты получают «неявно». Проще говоря, набор свойств и методов объектов в общем случае не ограничивается лишь теми, что описаны явно. Все свойства и методы объектов делятся на *перечисляемые* и *неперечисляемые*. Перебираются в операторе цикла for-in только перечисляемые свойства и методы. Свойства и методы, описываемые и создаваемые пользователем/программистом, по умолчанию являются перечисляемыми.

НА ЗАМЕТКУ

Важно подчеркнуть, что значением переменной, которая «перебирает» свойства объекта, в операторе цикла for-in является не значение свойства, а название свойства! Чтобы по названию свойства объекта получить значение свойства объекта, можно после имени объекта указать в квадратных скобках название свойства, то есть использовать команду вида объект[свойство].

Небольшая иллюстрация к использованию оператора цикла for-in представлена в листинге 4.8.

Листинг 4.8. Использование оператора for-in (файл Listing04_08.js)

```
document.write("<h4>Цвета</h4>")
var colors={red:"красный",yellow:"желтый",green:"зеленый"}
for(var s in colors){
  document.write(s+" - "+colors[s]+"<br>")
}
```

Результат выполнения сценария в веб-документе показан на рис. 4.7.

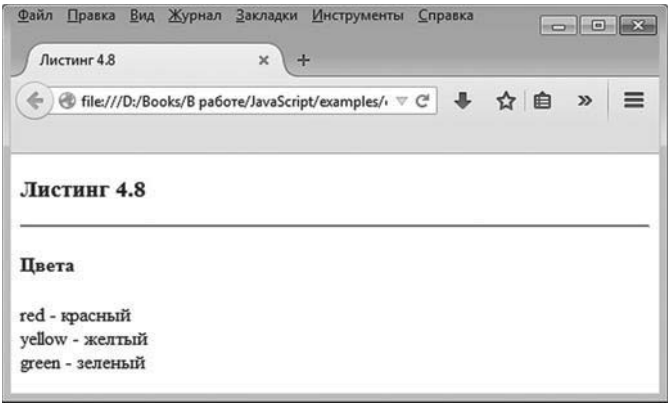

Рис. 4.7. Результат выполнения сценария с оператором цикла *for-in*

В веб-документ сценарий включается с помощью следующего HTML-кода:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 4.8</title>
</head>
<body><h3>Листинг 4.8</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing04_08.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Что касается самого сценария, то код его прост. Командой var colors={red: "красный",yellow:"желтый",green:"зеленый"} создается объект colors, у которого три поля: red со значением "красный", yellow со значением "желтый" и green со значением "зеленый". Далее запускается оператор цикла for-in. В операторе объявляется переменная s, которая должна принимать значениями названия свойств объекта colors. Таким образом, переменная s будет последовательно принимать значения "red", "yellow" и "green".

ДЕТАЛИ

Поскольку у объекта colors свойства red, yellow и green, то при переборе свойств объекта переменная в операторе цикла принимает в качестве значений названия свойств, то есть в данном случае это будут текстовые значения "red", "yellow" и "green". Однако при всем том нет гарантии, что переменная в операторе цикла будет принимать значения именно в такой последовательности, как они перечислены выше. Проще говоря, известно, какие значения переменная будет принимать, но неизвестно, в какой последовательности.

В теле оператора цикла выполняется всего одна команда document.write (s+" - "+colors[s]+"
"), которой в рабочий документ выводится название каждого свойства и, через дефис, его значение. Название свойства записано в переменную s. Значение свойства получаем с помощью инструкции colors[s].

ДЕТАЛИ

Мы знаем, что к свойству объекта обращение выполняется в «точечном» формате: после имени объекта через точку указывается название свойства. Но это не единственный способ обращения к свойству объекта. Еще можно указать название объекта, а в квадратных скобках поместить название свойства. Как мы узнаем из следующей главы, так выполняется индексирование массивов (индекс элементов указывается после имени массива в квадратных скобках). Объект аналогичен массиву, но только ассоциативному — то есть массиву, у которого индексами могут быть не только целые числа, но и значения иных типов (например, текст).

Оператор in

Часто возникает необходимость проверить объект на наличие в нем определенного свойства (или метода). В таких случаях полезен опе-

ратор in. Выражение на основе оператора in возвращает результатом значение true, если данное свойство у объекта есть, и значение false, если такого свойства у объекта нет. Формат использования оператора in следующий:

```
свойство in объект
```

Небольшой пример использования данного оператора приведен в листинге 4.9.

Листинг 4.9. Использование оператора in (файл Listing04_09.js)

```
document.write("<h4>Проверка свойств объекта</h4>")
var colors={red:"красный",yellow:"желтый",green:"зеленый"}
var a="red" in colors
var b="blue" in colors
document.write("Наличие свойства red: "+a+"<br>")
document.write("Наличие свойства blue: "+b+"<br>")
```

В сценарии создается объект colors со свойствами red, yellow и green. Затем вычисляются два выражения: в переменную a записывается результат вычисления выражения "red" in colors, которым проверяется наличие у объекта colors свойства red, а переменной b в качестве значения присваивается результат вычисления выражения "blue" in colors, которым проверяется наличие у объекта colors свойства blue. Поскольку свойство red у объекта colors есть, то результатом выражения "red" in colors является значение true. А вот свойства blue у объекта colors нет, поэтому результатом выражения "blue" in colors является значение false.

НА ЗАМЕТКУ

Обратите внимание, что значение свойства при передаче операндом в оператор in заключается в двойные кавычки — то есть на самом деле указывается текстовая строка с названием свойства. Так же поступаем, когда к свойству объекта обращаемся с использованием квадратных скобок, то есть при использовании команды вида объект[свойство].

Для проверки работы сценария воспользуемся следующим веб-документом:

```
<!DOCTYPE HTML>
<html>
```

```
<head>
  <title>Листинг 4.9</title>
</head>
<body><h3>Листинг 4.9</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing04_09.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 4.8 показано, как будет выглядеть веб-документ с результатами выполнения сценария.

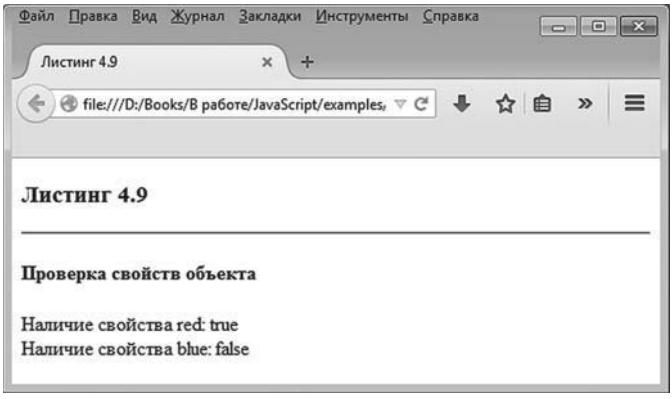

Рис. 4.8. Результат выполнения сценария, в котором проверяется наличие у объекта определенных свойств

Как видим, результат вполне ожидаемый.

НА ЗАМЕТКУ

Проверка на наличие у объекта определенного метода выполняется точно так же, с той лишь поправкой, что за метод «отвечает» свойство, название которого тождественно названию метода.

Оператор delete и удаление свойств и методов

С помощью оператора delete у объекта удаляются свойства и методы. Оператор используется в следующем формате:

delete объект.свойство

После оператора delete указывается ссылка на свойство или метод. В результате выполнения соответствующей команды свойство или метод, указанные после оператора delete, удаляются из объекта.

НА ЗАМЕТКУ

С помощью оператора delete можно удалить не только свойство или метод объекта, но и сам объект. Команда удаления объекта выглядит как delete объект. После ее выполнения объект, на который ссылается переменная, указанная после оператора delete, будет удален.

Пример, представленный в листинге 4.10, призван показать, к чему приводит применение оператора delete по отношению к свойствам и методам объекта.

Листинг 4.10. Использование оператора delete (файл Listing04_10.js)

```
document.write("<h4>Удаление свойств и методов</h4>")
// Создание объекта:
var colors={
  red:"красный",
  yellow:"желтый",
  green:"зеленый",
  show:function(){
    with(document){
      write("Свойства и методы объекта:<br>")
      for(var s in this){
        write(s+" | ")
      }
      write("<hr>")
    }
  }
```

```
} // Окончание описания объекта
// Отображение списка свойств:
colors.show()
// Удаление свойства red:
delete colors.red
// Отображение списка свойств:
colors.show()
// Удаление свойства yellow:
delete colors.yellow
// Отображение списка свойств:
colors.show()
// Удаление метода show():
delete colors.show
// Проверка наличия у объекта метода show:
var tst='"show" in colors'
document.write(tst+" -> "+eval(tst)+"<br>")
// Проверка наличия у объекта свойства green:
tst='"green" in colors'
document.write(tst+" -> "+eval(tst))
```

В сценарии создается объект colors, у которого есть свойства red, yellow и green, а также метод show(), при вызове которого отображаются названия всех свойств объекта, из которого вызывается метод, включая название самого метода show().

ДЕТАЛИ

В теле функции, указываемой значением свойства show, использован оператор with, в котором указана ссылка на объект документа document, благодаря чему при вызове метода write() объект документа, из которого вызывается метод, не указывается. Внутри with-оператора вызывается оператор цикла for-in, в котором переменная s пробегает названия свойств объекта, определяющегося ссылкой this. Данная инструкция, напомним, является стандартной ссылкой на объект, из которого вызывается метод. Командой write(s+" | "), выполняемой в каждом цикле, отображается название свойства и вертикальная черта, играющая роль декоративного разделителя.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Дескриптор <hr> используется в HTML-коде для отображения в вебдокументе горизонтальной линии.

Все это богатство добавляется в объект colors при создании объекта. После создания объекта командами delete colors.red, delete colors.yellow и delete colors.show из объекта colors удаляются соответственно свойства red, yellow и метод show(). Пока метод show() из объекта не удален, после создания объекта и после удаления очередного свойства из объекта colors вызывается метод show() для отображения списка тех свойств, что еще остались у объекта. После удаления метода show() для проверки наличия у объекта этого метода используем оператор in. Более конкретно, переменной tst значением присваивается текстовая строка '"show" in colors', представляющая собой текстовую реализацию команды проверки наличия у объекта colors свойства show. Поскольку непосредственно в текстовой строке нам необходимо использовать текстовый литерал "show", внутренний литерал берется в двойные кавычки, а внешний — в одинарные.

НА ЗАМЕТКУ

Имеет смысл напомнить, что в JavaScript литералы можно заключать, по своему усмотрению, в одинарные или двойные кавычки. Эти два способа создания литералов являются эквивалентными.

В команде document.write(tst+" -> "+eval(tst)+"
") само по себе значение переменной tst является текстом, а вот результатом инструкции eval(tst) является значение выражения, которое спрятано в переменную tst.

НА ЗАМЕТКУ

Если функции eval() передать текстовую строку, то будет выполнено выражение, «спрятанное» в текстовой строке. Результатом инструкции на основе функции eval() является результат выполненного выражения.

Для тестирования сценария используем HTML-код, который представлен ниже:

```
<!DOCTYPE HTML>
<html>
<head>
```

```
  <title>Листинг 4.10</title>
</head>
<body><h3>Листинг 4.10</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing04_10.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 4.9 показано, как выглядит окно браузера с веб-документом, в котором выполняется описанный выше сценарий.

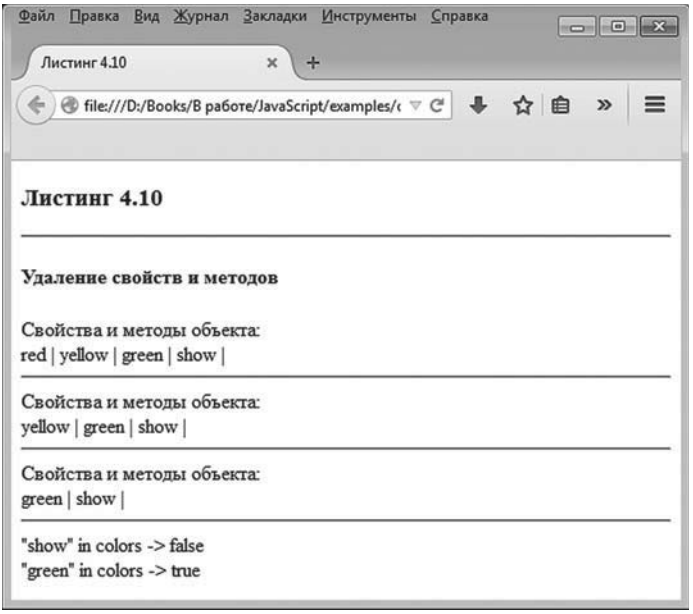

Рис. 4.9. Результат выполнения сценария, в котором у объекта удаляются свойства и методы

Понятно, что существуют и другие операторы, функции и методы, используемые при работе с объектами. С ними мы будем знакомиться по мере необходимости. А далее познакомимся с таким важным поня-

тием, как *прототип объекта*. Не будет преувеличением сказать, что данная тема является ключевой для понимания механизма реализации концепции ООП в языке JavaScript.

Прототипы

— Мастера не мудрствуют.
— А могильщики в Гамлете?
— Ремесленники!

из к/ф «Покровские ворота»

Каждому объекту в JavaScript автоматически в соответствие ставится другой объект, который называется *прототипом*. Другими словами, каждый из созданных нами ранее объектов (и те, что созданы не нами) имеет прототип, который мы будем называть прототипом объекта. Сами мы не создавали прототипы объектов и явно к ним не обращались, но они существуют. Главное назначение прототипа объекта — определить, какие у объекта есть свойства и методы.

НА ЗАМЕТКУ

На самом деле в JavaScript разделение на свойства и методы достаточно условное. Метод представляет собой свойство, значением которого является функция. Поэтому нередко, если это не приводит к недоразумениям, под свойствами мы будем подразумевать как непосредственно свойства, так и методы.

У читателя может возникнуть резонный вопрос: зачем нужен прототип, если свойства и методы объекта мы определяли сами, когда создавали этот объект? В принципе логика здесь есть. Но дело в том, что все не так просто, как кажется на первый взгляд. Если вкратце, то ситуация следующая: помимо тех свойств и методов, которые объекту добавляются, так сказать, в явном виде, непосредственно при создании, у объектов имеются еще (ну или внешне создается такая иллюзия, что они есть) некоторые свойства и методы, которые объект получает автоматически. Какие именно — зависит от прототипа объекта. Просто ранее мы об этих свойствах и методах не знали, в них не видели потребности, не обращались к ним и, соответственно, о них не упоминали. Но они есть.

НА ЗАМЕТКУ

Обычно свойства и методы объекта делятся на две категории: *собственные* и *унаследованные*. Собственные свойства и методы — те, что непосредственно добавляются в объект в сценарии при создании объекта и работе с ним. Унаследованные свойства — те, что определяются прототипом объекта. В рамках данной терминологии свойства и методы, которые мы обсуждали ранее, — собственные свойства объектов.

Таким образом, когда создается объект, то набор его свойств определяется не только непосредственно командами создания/формирования объекта, но и некоторым другим объектом, который является прототипом. Прототип можно указать явно, а можно не указывать.

В последнем случае прототип определяется автоматически в зависимости от способа создания объекта.

НА ЗАМЕТКУ

Прототип объекта — тоже объект. Следуя описанной выше логике, у него должен быть прототип. А у прототипа — свой прототип, и так далее. Получается своеобразная иерархия прототипов, в вершине которой находится единственный объект-прототип, у которого прототипа нет (об этом объекте мы поговорим несколько позже). Такая вот получается «матрешка».

Если говорить более конкретно, то имеется такой конструктор объектов, как `Object` (который мы уже упоминали ранее). Прототип верхнего уровня (объект-прототип, который находится в вершине иерархии прототипов и у которого нет своего прототипа) доступен через свойство `prototype` конструктора объектов `Object`. Более конкретно, доступ к этому «главному» прототипу можно получить с помощью инструкции `Object.prototype`.

Механизм доступа к свойствам и создание объекта на основе прототипа

Чтобы легче было понять, как объект «идеологически» связан со своим прототипом, разберемся с тем, как при обращении к свойству объекта (причем здесь имеются в виду как непосредственно свойства, так и методы) выполняется поиск этого самого свойства. Последовательность действий такая.

- Просматриваются все свойства, добавленные в объект при его создании или в процессе дальнейшего использования. Если свойство с нужным именем найдено, считывается его значение (вызывается метод). Или, проще говоря, сначала просматриваются *собственные* свойства объекта.
- Если среди собственных свойств объекта свойство с нужным именем не найдено, начинается поиск свойства с соответствующим названием в объекте-прототипе. Если там данное свойство есть, именно оно и считывается.
- Если среди свойств объекта-прототипа свойство с данным именем не найдено, начинается просмотр свойств объекта, который является прототипом для объекта-прототипа. И так далее.

Еще раз подчеркнем, что выше речь шла только о *считывании* (не о присваивании!) значения свойства (или вызове метода). *Присваивание* значения свойству выполняется несколько иначе. Допустим, выполняется команда, в которой свойству объекта присваивается значение. Если такое (собственное) свойство у объекта существует, ему присваивается новое значение. Если данного свойства у объекта нет, то оно в объект добавляется и получает значение (то, которое присваивается).

ДЕТАЛИ

При присваивании значения некоторому свойству объекта может оказаться так, что непосредственно у объекта такого свойства нет, зато такое (то есть с таким именем) свойство имеется у объекта-прототипа. Ситуация разрешается следующим образом: объекту добавляется свойство с данным именем, и свойству присваивается значение. При этом у объекта-прототипа остается свойство с таким же именем, и значение данного свойства в прототипе останется неизменным.

Фактически описанные выше схемы считывания и присваивания значений свойствам раскрывают связь между объектом и его прототипом. Но, чтобы сделать обсуждение прототипов более предметным, разумно перейти в прикладную плоскость — то есть в плоскость практического использования прототипов. Важный момент связан с тем, что при создании объекта можно в явном виде указать его прототип. Причем прототипом может выступать любой объект.

НА ЗАМЕТКУ

Когда мы ранее создавали объекты путем описания их литералов, прототипом таких объектов автоматически становился прототип Object.prototype объекта Object (то есть прототип наивысшего уровня).

Допустим, мы хотим создать некоторый объект и использовать при этом некоторый прототип. Тогда для создания такого объекта таким прототипом можно использовать метод create() конструктора объектов Object, аргументом которому передается прототип. Вся команда создания объекта может выглядеть следующим образом (жирным шрифтом выделены ключевые элементы кода):

```
var объект=Object.create(прототип)
```

Небольшой пример, иллюстрирующий особенности работы с прототипами, представлен в листинге 4.11.

Листинг 4.11. Использование прототипа (файл Listing04_11.js)

```
// Объект для использования в качестве прототипа:
var X={
  color:"красный",
  number:123,
  show:function(arg){
    document.write("<b>"+arg+"</b>: ")
    for(var s in this){
      if(s!="show"){
        document.write(s+" -> "+this[s]+" | ")
      }
    }
    document.write("<br>")
  }
} // Окончание описания объекта-прототипа
// Первый объект:
var A=Object.create(X)
// Второй объект:
var B=Object.create(X)
```

```
// Проверяем свойства объектов:
showAll()
// Присваиваем новые значения свойствам объектов:
A.color="желтый"
A.number=321
B.color="зеленый"
// Проверяем свойства объектов:
showAll()
// Добавление свойства name в объект-прототип:
X.name="прототип"
// Добавление свойства state в первый объект:
A.state=true
// Проверяем свойства объектов:
showAll()
// Удаляем свойство number объекта-прототипа:
delete X.number
// Удаляем свойство color первого объекта:
delete A.color
// Проверяем свойства объектов:
showAll()
// Функция для отображения свойств объектов:
function showAll(){
  X.show("X")
  A.show("A")
  B.show("B")
  document.write("<hr>")
}
```

Чтобы проверить, как выглядит результат выполнения сценария, воспользуемся таким веб-документом:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 4.11</title>
```

```
</head>
<body><h3>Листинг 4.11</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing04_11.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 4.10 показано, что появится в окне браузера, если загрузить документ с представленным выше сценарием.

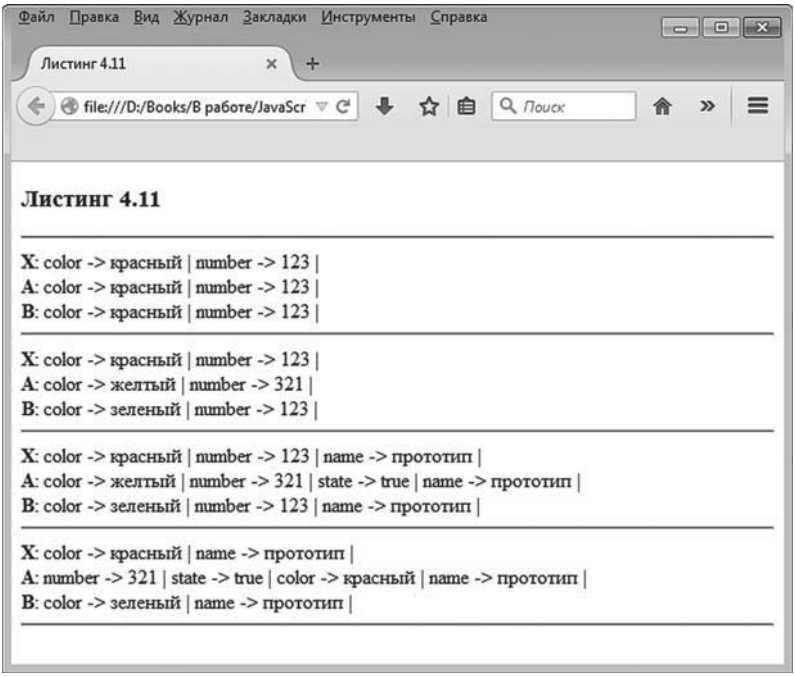

Рис. 4.10. Результат выполнения сценария, в котором объекты создаются на основе заданного прототипа

Программный код сценария достаточно большой, да и результат его выполнения не самый тривиальный. Поэтому имеет смысл проанали-

зировать код в «построчном» режиме, сопоставляя команды в сценарии с теми сообщениями, которые отображаются в рабочем документе. Для удобства ниже приведен текстовый вариант этих сообщений (вместо горизонтальных линий добавлены пустые строки).

Результат выполнения сценария (из листинга 4.11)

X: color -> красный | number -> 123 |
A: color -> красный | number -> 123 |
B: color -> красный | number -> 123 |

X: color -> красный | number -> 123 |
A: color -> желтый | number -> 321 |
B: color -> зеленый | number -> 123 |

X: color -> красный | number -> 123 | name -> прототип |
A: color -> желтый | number -> 321 | state -> true | name -> прототип |
B: color -> зеленый | number -> 123 | name -> прототип |

X: color -> красный | name -> прототип |
A: number -> 321 | state -> true | color -> красный | name -> прототип |
B: color -> зеленый | name -> прототип |

Итак, в сценарии создается объект X, который мы планируем использовать в качестве прототипа при создании двух других объектов (мы их создадим и назовем A и B). В объекте X описано свойство color со значением "красный" и свойство number со значением 123. Также в объекте описан метод show(). Этим методом при вызове отображаются названия и значения всех свойств объекта, за исключением метода show().

НА ЗАМЕТКУ

Напомним, что технически метод реализуется как свойство, значением которого является функция. При вызове метода show() отображаются значения всех свойств объекта, за исключением свойства show. То есть себя метод «игнорирует».

У метода show() один аргумент (обозначен как arg). Предполагается, что аргумент текстовый. Через этот аргумент мы планируем передавать

в метод название объекта, для которого отображается набор свойств. Делается это исключительно для большей наглядности. Командой document.write(""+arg+": ") в теле метода значение аргумента (название объекта) отображается в рабочем документе с применением жирного шрифта. Затем запускается оператор цикла for-in, в котором переменная s перебирает свойства объекта, из которого вызывается метод. Для идентификации объекта, из которого вызывается метод, используем ключевое слово this. В теле оператора цикла всего одна команда document.write(s+" -> "+this[s]+" | "), которой отображается название свойства и его значение.

ДЕТАЛИ

Командой document.write(s+" -> "+this[s]+" | ") пары свойство/значение, разделенные стрелкой, отображаются в одну строку. Разделителем между парами с названием и значением свойства служит вертикальная черта. Название свойства содержится в качестве значения в переменной s. Чтобы по названию свойства получить значение свойства, используем инструкцию this[s]. Данная инструкция означает буквально следующее: свойство с названием, записанным в переменную s, для объекта, из которого вызывается метод.

Но сама эта команда помещена внутрь условного оператора с проверяемым условием s!="show". Условие истинно, если название свойства (значение переменной s) не совпадает с названием метода show() (методу, как отмечалось, соответствует свойство show). Таким образом, команда по отображению названий и значений свойств выполняется для всех свойств, кроме метода show(), чего мы и добивались. По завершении выполнения оператора цикла командой document.write("
") в веб-документ вставляется инструкция разрыва строки.

На основе объекта X создаются объекты A и B. Для них объект X служит прототипом. Создаются объекты командами var A=Object.create(X) и var B=Object.create(X) соответственно. Сразу после создания объектов проверяются значения их свойств. Для этого вызывается функция showAll(), которая не возвращает результат и у которой нет аргументов. Сама функция описана в конце сценария, и ее код состоит из команд вызова метода show() из объектов X, A и B. В результате для всех трех объектов отображаются названия и значения их свойств.

ДЕТАЛИ

В теле функции showAll() последовательно выполняются команды X.show("X") (отображение свойств объекта-прототипа), A.show("A") (отображение свойств первого объекта) и B.show("B") (отображение свойств второго объекта), после чего командой document.write("<hr>") в рабочий документ добавляется дескриптор <hr> вставки горизонтальной линии.
Вообще функция showAll() имеет вспомогательный характер и призвана уменьшить объем программного кода. Дело в том, что по ходу сценария после внесения изменений в структуру и свойства объектов каждый раз выполняется проверка свойств всех трех объектов (X, A и B). Чтобы каждый раз не вызывать в явном виде метод show() для всех трех объектов, мы поместили команды вызова метода в отдельную функцию.

При первой проверке свойств объектов X, A и B оказывается, что у них не только одинаковый набор свойств, но и значения этих свойств совпадают (первый блок из трех строчек в сообщениях, выводимых в рабочий документ сценарием). Это не случайно. Дело в том, что оба объекта A и B созданы на основе объекта X, который стал их прототипом. Никакие дополнительные свойства (кроме тех, что описаны в объекте X) в объектах A и B не объявлялись. Поэтому когда выполняется обращение к свойству color или number объекта A или B, то на самом деле значение свойства считывается из прототипа, коим является объект X. Здесь фактически срабатывает правило, что если у объекта собственного свойства с определенным именем нет, то поиск такого свойства выполняется в объекте-прототипе.

НА ЗАМЕТКУ

Стоит заметить, что в операторе for-in перебираются не только собственные свойства объекта, но и свойства, унаследованные им из прототипа.

Далее командами A.color="желтый" и A.number=321 свойствам color и number объекта A присваиваются новые значения. Но это так кажется со стороны, что значения «новые». На самом деле у объекта появляются собственные свойства color и number, и указанные свойства получают значения соответственно "желтый" и 321. Очень важное обстоятельство: значения полей color и number объекта X в данной ситуации *совершенно не меняются*.

У объекта B меняется только значение свойства color (команда B.color="зеленый"). Свойство number объекта B мы не трогаем (оно остается, каким было). Что это означает? У объекта B появляется свое собственное свойство color, а при обращении к свойству number объекта B будет вступать в игру, как и ранее, объект-прототип X. Более конкретно, когда обращение (для считывания значения) выполняется к свойствам color и number объекта A, то, поскольку у объекта A теперь есть собственные свойства с такими названиями, они и будут задействованы. При обращении к свойствам color и number объекта B свойство color используется собственное, а значение свойства number (поскольку у объекта B нет такого собственного свойства) считывается из объекта-прототипа X. Сказанное подтверждается вызовом функции showAll(): во втором блоке из трех строк для результата сценария значения свойств объекта X остались неизменны, у объекта A изменились значения обоих свойств, а у объекта B изменилось лишь значение свойства color.

На следующем этапе в объект прототипа X добавляется свойство name, и этому свойству присваивается значение "прототип". Добавляется новое свойство в прототип просто: выполняется команда присваивания значения свойству X.name="прототип".

Помимо добавления нового свойства в прототип, объекту A добавляется свойство state со значением true (команда A.state=true). После добавления свойства name в объект-прототип X и свойства state в объект A вызываем функцию showAll() для проверки значений свойств объектов X, A и B. Результат проверки такой (третий блок из трех строк для результата сценария).

- У всех объектов появилось свойство name со значением "прототип".
- У объекта A появилось свойство state со значением true.

Все остальное без изменений. Краткий вывод следующий: добавление свойства в прототип добавляет это же свойство (с таким же значением) во все объекты, созданные на основе данного прототипа.

НА ЗАМЕТКУ

Технически свойство name появляется у объекта X. У объектов A и B такого *собственного* свойства нет. Но поскольку объект X является прототипом объектов A и B, при обращении к свойству name этих объектов на самом деле возвращается значение свойства name объекта X.

Добавление нового свойства в обычный объект (не прототип) приводит к появлению данного свойства у объекта и никак не влияет на свойства других объектов.

Наконец, командой delete X.number у объекта-прототипа удаляется свойство number. Еще командой delete A.color у объекта A удаляется свойство color. После выполнения данных команд вызывается функция showAll(). Каков будет результат? Вкратце ситуация такая (последние три строки в сообщениях, которые выводятся при выполнении сценария).

- У объектов X и B свойство number пропадает.
- У объекта A свойство number остается.
- Несмотря на удаление свойства color у объекта A, color в списке свойств объекта присутствует, но значение данного свойства изменилось — теперь значение у свойства color объекта A такое же, как и значение свойства color у объекта X.

Разберем данную ситуацию по пунктам. Итак, почему пропадает свойство number у объекта X, в принципе понятно: оно удаляется из этого объекта, и поэтому его там больше нет.

У объекта B свойство number пропадает, поскольку собственного свойства с таким названием у объекта до этого не было и объект B «получал» значение свойства number из своего прототипа — то есть из объекта X. После удаления у объекта X свойства number оно естественным образом «исчезает» из объекта B.

У объекта A было собственное свойство number. Поэтому при обращении к свойству number объекта A используется значение собственного свойства number. Свойство number объекта-прототипа X в этом процессе не участвует. Поэтому удаление свойства number у объекта X никак на собственном свойстве number объекта A не сказывается.

Со свойством color объекта A ситуация более интересная. До удаления свойства color из объекта A у него было собственное свойство с таким именем. При удалении свойства color из объекта A удаляется именно собственное свойство. Но остается свойство color у объекта-прототипа X. Поэтому, если после удаления свойства color из объекта A выполнить обращение к этому «удаленному» свойству, будет считано значение свойства color объекта-прототипа X. Что, собственно, и происходит.

НА ЗАМЕТКУ

Важное обстоятельство, на которое хочется обратить особое внимание, относится к методу show(), которым отображаются свойства объектов и который в сценарии вызывается при вызове функции showAll(). Метод show() описан в объекте X. При вызове из объекта A или B метода show() на самом деле используется свойство show из объекта-прототипа X, поскольку в объектах A и B собственного свойства show нет. Тем не менее при вызове метода show() из объекта A или B отображаются свойства объекта, из которого формально вызывается метод (соответственно объекта A или B), а не свойства объекта X.

Получение доступа к прототипу

В рассмотренном выше примере при создании объектов прототип указывался в явном виде. Если при создании объекта прототип явно не указан, он все равно существует. Естественно, возникает вопрос: как получить доступ к объекту, который является прототипом данного объекта? Ответ состоит в том, что получить ссылку на прототип объекта можно, если воспользоваться методом getPrototypeOf() объекта Object. Аргументом методу getPrototypeOf() передается объект, к прототипу которого необходимо получить доступ. Результатом возвращается ссылка на объект, являющийся прототипом объекта, переданного аргументом методу getPrototypeOf().

НА ЗАМЕТКУ

Напомним, что у конструктора объектов Object есть свойство prototype, которое позволяет получить доступ к прототипу верхнего уровня. Соответствующая инструкция имеет вид Object.prototype. У объекта Object.prototype нет прототипа. Поэтому, если попытаться получить прототип объекта Object.prototype с помощью команды Object.getPrototypeOf(Object.prototype), получим значение null. Это пустая ссылка — то есть ссылка, которая не указывает ни на какой объект.

Даже если при создании объекта мы не указывали в явном виде прототип, то к прототипу можно получить доступ и, например, воспользоваться тем приемом, который нами применялся выше: добавить в прототип свойство или метод, и это свойство (или метод) автоматически появится у всех объектов, созданных на основе данного прототипа. Как иллюстрацию к сказанному рассмотрим небольшой пример, представленный в листинге 4.12.

Листинг 4.12. Добавление свойств в прототип верхнего уровня (файл Listing04_12.js)

```
// Вспомогательная текстовая переменная:
var txt='"name" in Math'
// Проверка наличия свойства name у объекта Math:
document.write(txt+" -> "+eval(txt)+"<br>")
// Первый объект:
var A={color:"красный"}
// Второй объект:
var B=new Object()
// Добавление свойства number в объект B:
B.number=100
// Отображаются свойства объектов:
showAll()
// Добавляется свойство name в прототип Object.prototype:
Object.prototype.name="объект A"
// Проверка наличия свойства name у объекта Math:
document.write(txt+" -> "+eval(txt)+"<br>")
// Отображаются свойства объектов:
showAll()
// Добавляется свойство name в объект B:
B.name="объект B"
// Отображается значение свойства name объекта Math:
document.write("Math.name -> "+Math.name+"<br>")
// Отображаются свойства объектов:
showAll()
// Удаление свойства name у прототипа Object.prototype:
delete Object.prototype.name
// Проверка наличия свойства name у объекта Math:
document.write(txt+" -> "+eval(txt)+"<br>")
// Отображаются свойства объектов:
showAll()
// Функции для отображения свойств объектов:
function show(obj){
  for(var s in obj){
```

```
    document.write(s+" -> "+obj[s]+" | ")
  }
  document.write("<br>")
}
function showAll(){
  document.write("Объект A: ")
  show(A)
  document.write("Объект B: ")
  show(B)
  document.write("<br>")
}
```

В представленном сценарии командой var A={color:"красный"} создается объект A со свойством color, а также командой var B=new Object() создается пустой объект B (здесь мы воспользовались «услугами» конструктора объектов Object). После создания пустого объекта B командой B.number=100 в объект B добавляется свойство number.

На следующем этапе мы проверяем, какие свойства имеются у объектов A и B. Также в силу определенных причин нас интересует вопрос о наличии у встроенного объекта Math (напомним, что данный встроенный объект используется при работе с математическими функциями) свойства с названием name.

Для получения ответа на этот вопрос мы используем оператор in. Точнее, мы определяем вспомогательную переменную txt со значением '"name" in Math', которое представляет собой «упакованную» в текст команду проверки наличия у объекта Math свойства name.

> **НА ЗАМЕТКУ**
>
> Поскольку в тексте, присваиваемом значением переменной txt, использовано слово в двойных кавычках, то весь текст заключается в одинарные кавычки.

Для проверки наличия свойства name у объекта Math используется команда document.write(txt+" -> "+eval(txt)+"
"). Для вычисления выражения, содержащегося в текстовой строке, строку передаем аргументом функции eval(). Далее вызовом функции showAll() отображаются свойства (и их значения) для объектов A и B.

ДЕТАЛИ

Функция showAll() описана и используется исключительно из соображений удобства и экономии программного кода. В теле функции showAll() вызывается функция show(), которой отображаются названия и значения свойств объекта, переданного аргументом функции. Обе функции описаны в конце сценария. Подобные программные коды мы уже рассматривали, и поэтому анализировать их нет особого смысла.

Функция showAll() вызывается в сценарии в общей сложности четыре раза, поэтому результат выполнения сценария, представленный ниже, состоит из четырех блоков (каждый блок — по три строки).

Результат выполнения сценария (из листинга 4.12)

```
"name" in Math -> false
Объект A: color -> красный |
Объект B: number -> 100 |

"name" in Math -> true
Объект A: color -> красный | name -> объект A |
Объект B: number -> 100 | name -> объект A |

Math.name -> объект A
Объект A: color -> красный | name -> объект A |
Объект B: number -> 100 | name -> объект B |

"name" in Math -> false
Объект A: color -> красный |
Объект B: number -> 100 | name -> объект B |
```

В первом блоке первая строка свидетельствует о том, что свойства name у объекта Math нет, что вполне ожидаемо. Следующие две строки в первом блоке дают представление о свойствах объектов A и B соответственно (у объекта A есть свойство color, а у объекта B есть свойство number).

Далее, после вызова функции showAll(), выполняется команда Object.prototype.name="объект A". Этой командой прототипу верхнего уровня Object.prototype добавляется свойство name, и ему присваивается значение "объект A". После добавления свойства name в прототип Object.prototype

выполняются команды document.write(txt+" -> "+eval(txt)+"
") (проверка наличия свойства name у объекта Math) и showAll() (отображение названий и значений свойств объектов A и B). Результат выполнения команд (три строки во втором блоке) свидетельствует о том, что у объекта Math появилось свойство name, равно как и у объектов A и B. Значение свойства name для объектов A и B возвращается равным "объект A".

Объяснение данного факта базируется на том обстоятельстве, что при создании объекта с помощью литерала (как в случае с объектом A) или с помощью конструктора объектов Object (как в случае с объектом B) прототипом является объект Object.prototype, то есть прототип наивысшего уровня. Поэтому вполне ожидаемо, что после добавления свойства name в прототип Object.prototype у объектов A и B появляется свойство name. Здесь ситуация такая же, как и в рассмотренном ранее примере, когда прототип объекта задавался в явном виде.

Что касается объекта Math, то, хотя он и встроенный, его прототипом также является объект Object.prototype. Так что причины появления свойства name у объекта Math те же, что и для объектов A и B.

ДЕТАЛИ

Даже если бы для рассмотренных объектов прототип отличался от Object.prototype, результат был бы аналогичным (правда, при условии, что прототип объектов является частью иерархии прототипов с объектом Object.prototype в вершине иерархии). Причина кроется в механизме получения значения свойства: если у объекта нет собственного свойства с заданным именем, поиск выполняется в прототипе, затем в прототипе прототипа, и так далее — вплоть до объекта-прототипа высшего уровня, которым является Object.prototype.

Далее мы узнаем, что объект может быть создан совсем без прототипа. Добавление или удаление свойств в объект-прототип Object.prototype на объектах, созданных без прототипов, не сказывается.

После добавления в объект B собственного свойства name (команда B.name="объект B") обращение к свойству name для объектов A и Math означает обращение к свойству name объекта-прототипа Object.prototype, а вот при обращении к свойству name объекта B используется собственное свойство данного объекта. В последнем легко убедиться, если проанализировать последствия удаления свойства name из объекта-прототипа Object.prototype (команда delete Object.prototype.name). А последствия такие: у объектов A и Math свойства name больше нет, а у объекта B свойство name остается.

Для тестирования сценария используем представленный ниже веб-код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 4.12</title>
</head>
<body><h3>Листинг 4.12</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing04_12.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Результат выполнения сценария представлен на рис. 4.11.

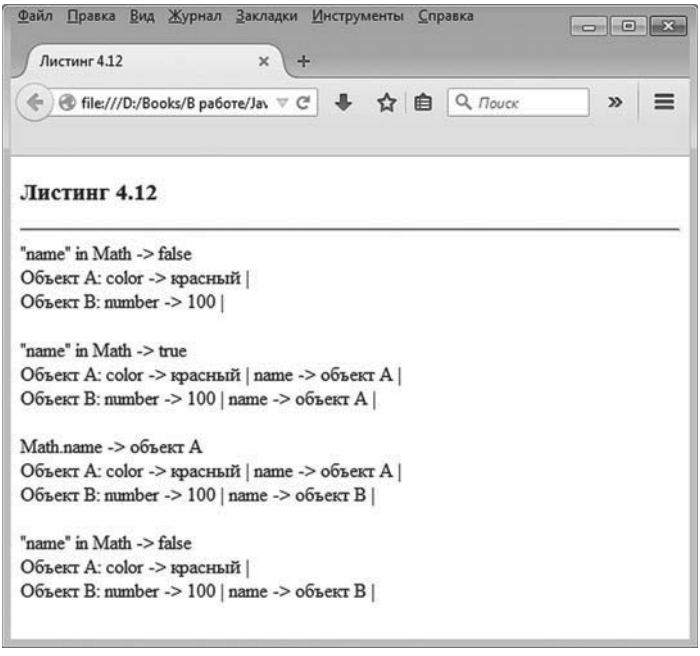

Рис. 4.11. Результат выполнения сценария, в котором в прототип высшего уровня добавляется свойство

Еще один небольшой пример касается использования метода getPrototypeOf() для определения прототипа объекта. Рассмотрим сценарий, представленный в листинге 4.13.

Листинг 4.13. Получение доступа к прототипу с помощью метода getPrototypeOf() (файл Listing04_13.js)

```
// Первый объект (создается на основе Math):
var A=Object.create(Math)
// Второй объект (создается на основе A):
var B=Object.create(A)
// Третий объект (создается на основе B):
var C=Object.create(B)
// Четвертый объект (создается на основе C):
var D=Object.create(C)
// Добавление метода f() в прототип объекта A:
Object.getPrototypeOf(A).f=function(x){
  return 2*x+1
}
// Переменная с целочисленным значением:
var t=2
// Вызов метода f() из объекта Math:
document.write("Функция f("+t+") = "+Math.f(t)+"<br>")
// Новое значение переменной t:
t=3
// Вызов метода f() из объекта D:
document.write("Функция f("+t+") = "+D.f(t)+"<br>")
// Ссылка на прототип прототипа объекта D:
var obj=Object.getPrototypeOf(Object.getPrototypeOf(D))
// Проверка равенства объектов:
document.write("obj == B -> "+(obj==B)+"<br>")
```

В сценарии с помощью метода create() создаются четыре объекта. Объект A создается на основе прототипа Math (встроенный класс). Объект A, в свою очередь, служит прототипом для объекта B. На основе объекта B создается объект C, который является прототипом для объекта D.

Командой Object.getPrototypeOf(A).f=function(x){return 2*x+1} в прототип объекта A (то есть фактически в объект Math) добавляется свойство f, значением которого является функция с одним аргументом (обозначен как x), возвращающая результатом значение 2*x+1. Проще говоря, в объект Math добавляется метод f().

НА ЗАМЕТКУ

С помощью метода f() определяется математическая функция $f(x) = 2x + 1$.

Затем мы определяем переменную t, которая нужна для передачи аргумента методу f(). Для вызова метода f() в сценарии используются инструкции Math.f(t) и D.f(t). Если с инструкцией Math.f(t) все более-менее понятно, то использование команды D.f(t) возможно благодаря тому, что объект D наследует свойства и методы объекта Math через цепочку объектов-прототипов.

Также с помощью команды var obj=Object.getPrototypeOf(Object.getPrototypeOf(D)) в переменную obj записывается ссылка на прототип, который является прототипом для объекта D (то есть ссылка на объект B).

Данное обстоятельство подтверждается при проверке значения выражения obj==B, которое равно true и означает, что ссылки в переменной obj и переменной B указывают на один и тот же объект.

Результат выполнения сценария такой.

Результат выполнения сценария (из листинга 4.13)

```
Функция f(2) = 5
Функция f(3) = 7
obj == B -> true
```

Чтобы проверить работу сценария, используем представленный ниже HTML-код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 4.13</title>
```

```
</head>
<body><h3>Листинг 4.13</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing04_13.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 4.12 показано, как в окне браузера выглядит веб-документ, содержащий данный сценарий.

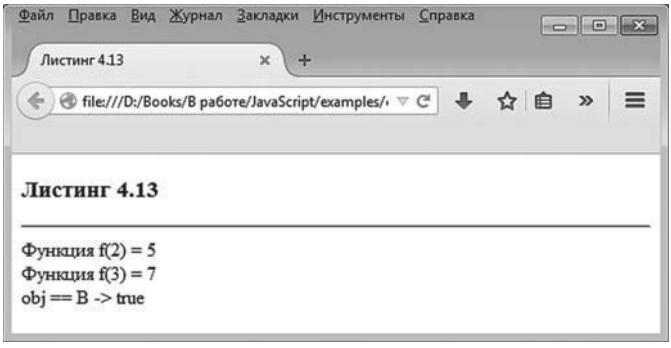

Рис. 4.12. Результат выполнения сценария, в котором использован метод *getPrototypeOf()*

Хотя наличие прототипа у объекта значительно повышает гибкость и эффективность программных кодов, иногда возникает необходимость создать объект (или объекты), не имеющий прототипа. Как уже упоминалось выше, такая возможность имеется.

ДЕТАЛИ

У объекта Object есть метод setPrototypeOf(), который позволяет задавать прототип для объекта. Метод вызывается из объекта Object, а аргументами ему передаются: ссылка на объект, для которого задается прототип, и ссылка на объект, который является прототипом.

Создание объектов без прототипа

Создать объект, у которого нет прототипа, исключительно просто. Для этого достаточно указать аргументом метода create() пустую ссылку null. Например, командой вида var obj=Object.create(null) создается объект obj, у которого нет прототипа. Некоторые особенности объектов, созданных без прототипа, иллюстрируются в примере, представленном в листинге 4.14.

Листинг 4.14. Создание объекта без прототипа (файл Listing04_14.js)

```
// Пустой объект с прототипом:
var A={}
// Пустой объект без прототипа:
var B=Object.create(null)
// Пустой объект с явно указанным прототипом:
var C=Object.create(B)
// Добавление свойства number в объект B:
B.number=100
// Добавление свойства name в прототип верхнего уровня:
Object.prototype.name="Объект A"
// Отображение свойств объектов:
show(A)
show(B)
show(C)
// Функция для отображения свойств объекта:
function show(obj){
  for(var s in obj){
    document.write(s+" -> "+obj[s])
  }
  document.write("<br>")
}
```

В сценарии создаются три пустых объекта.

- Объект A создается описанием литерала объекта (пустые фигурные скобки), поэтому по умолчанию его прототипом является прототип высшего уровня Object.prototype.

- Объект B создается с помощью инструкции Object.create(null), в которой аргументом методу create() передана пустая ссылка null. Поэтому прототипа у объекта B нет.
- Объект C создается (использована инструкция Object.create(B)) на основе прототипа, которым является объект B (у которого нет прототипа).

После создания объектов командой B.number=100 в объект B добавляется свойство number, а командой Object.prototype.name="Объект A" в прототип верхнего уровня добавляется свойство name. Затем с помощью функции show() отображаются названия и значения свойств объектов A, B и C.

ДЕТАЛИ

Функция show() описана в конце сценария. Аргументом функции передается объект. Функция не возвращает результат, а при вызове функции отображаются названия и значения свойств объекта, переданного аргументом функции. Код функции должен быть знаком и понятен читателю, поэтому нет смысла его комментировать.

Результат выполнения сценария представлен ниже.

Результат выполнения сценария (из листинга 4.14)

```
name -> Объект A
number -> 100
number -> 100
```

Поскольку собственных свойств у объекта A нет, а в прототип Object.prototype добавлено свойство name, то только это свойство отождествляется с объектом A. У объекта B нет прототипа, поэтому наличие свойства name у прототипа верхнего уровня Object.prototype на объекте B никак не сказывается. Отсюда получается, что свойства name у объекта B нет. Зато у объекта B есть собственное свойство number. Наконец, такое же свойство number (а если точнее, то это, буквально, то же самое свойство) имеется у объекта C. Хотя собственных свойств у объекта нет, но его прототипом является объект B, у которого есть данное свойство. На объекте B цепочка прототипов обрывается. Как следствие, прототип верхнего уровня Object.prototype в объекте C не наследуется, и свойства name у объекта C нет.

Представленный ниже HTML-код используем для тестирования работы сценария:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 4.14</title>
</head>
<body><h3>Листинг 4.14</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing04_14.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Как будет выглядеть окно браузера с веб-документом, в котором выполняется рассмотренный нами сценарий, показано на рис. 4.13.

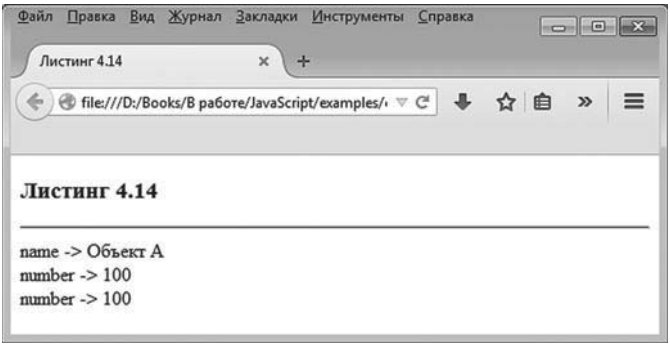

Рис. 4.13. Результат выполнения сценария, в котором создается объект без прототипа

Объекты без прототипов используются в разных ситуациях: например, для создания собственных иерархий объектов или предотвращения нежелательного автоматического наследования свойств из встроенных прототипов.

Конструкторы и прототипы

Мы уже знаем, что для создания объектов удобно описать специальную функцию, которая называется *конструктором* и которую (с оператором new) вызывают для создания объектов. Очевидное удобство конструктора связано с тем, что объекты создаются унифицированные, с одинаковым набором свойств (обычно). Но есть еще одно важное обстоятельство: все объекты, созданные с помощью некоторого конструктора, имеют один общий прототип. Данный факт означает, что уже после создания объектов мы можем добавлять и удалять свойства в эти объекты путем выполнения соответствующей операции с прототипом объектов. Доступ к прототипу объектов, которые создаются с помощью конструктора, получаем с помощью свойства prototype конструктора.

ДЕТАЛИ

Если речь идет о некотором Конструкторе, то объект создается посредством инструкции вида new Конструктор(аргументы), а для получения доступа к прототипу объектов, которые создаются с помощью Конструктора, используют инструкцию формата Конструктор.prototype.

Чтобы проиллюстрировать методы работы с прототипами объектов, создаваемых вызовом функции-конструктора, рассмотрим пример, представленный в листинге 4.15.

Листинг 4.15. Конструкторы и прототипы (файл Listing04_15.js)

```
// Конструктор объектов:
function MyObject(name,number){
  this.name=name
  this.number=number
  this.show=function(){
    for(var s in this){
      if(s!="show"){
        document.write(s+" -> "+this[s]+" | ")
      }
    }
    document.write("<br>")
  }
}
```

```
// Создание объектов на основе конструктора:
var A=new MyObject("Объект A",100)
var B=new MyObject("Объект B",200)
// Отображение свойств объектов:
document.write("<b>Свойства созданных объектов:</b><br>")
A.show()
B.show()
// Добавление свойства color в прототип объектов:
MyObject.prototype.color="прозрачный"
// Отображение свойств объектов:
document.write("<b>После добавления свойства:</b><br>")
A.show()
B.show()
// Добавление собственного свойства color в объект A:
A.color="белый"
// Удаление свойства color из прототипа объектов:
delete MyObject.prototype.color
// Отображение свойств объектов:
document.write("<b>После удаления свойства:</b><br>")
A.show()
B.show()
// Проверка прототипа объекта:
document.write("<b>Проверка прототипа объекта:</b><br>")
var txt="Object.getPrototypeOf(A)==MyObject.prototype"
document.write(txt+" -> "+eval(txt))
```

В сценарии описывается конструктор MyObject, которым создаются объекты с двумя свойствами и методом show(), предназначенным для отображения названий и значений свойств (за исключением названия и значения свойства, соответствующего методу). Конструктору передаются два аргумента, которые определяют значения свойств name и number создаваемого объекта.

НА ЗАМЕТКУ

Свойства, которые получает создаваемый с помощью конструктора объект, являются собственными свойствами данного объекта.

Объекты A и B создаются командами A=new MyObject("Объект A",100) и B=new MyObject("Объект B",200). То есть объекты создаются с помощью конструктора MyObject (с передачей аргументов). Это означает, что через конструктор MyObject можно получить доступ к прототипам объектов, созданных на его основе, что мы, собственно, и делаем. В частности, командой MyObject.prototype.color="прозрачный" в прототип объектов добавляется свойство color со значением "прозрачный". После выполнения данной операции у прототипа объектов A и B появляется свойство color, и, как следствие, оно отождествляется и с данными объектами (по схеме определения свойств объектов). После проверки свойств объектов A и B в объект A добавляется собственное свойство color со значением "белый", а из прототипа MyObject.prototype данное свойство удаляется. В результате у объекта B свойства color (унаследованного из прототипа) больше нет, а у объекта A остается собственное свойство color.

В описанной схеме ничего особенного нет. Нечто похожее мы наблюдали и ранее. Специфика ситуации связана с тем, что доступ к прототипам объектов мы получаем через их конструктор. Также следует понимать, что никто не запрещает нам получить доступ к прототипу объекта с помощью метода getPrototypeOf(). Например, командой Object.getPrototypeOf(A) возвращается ссылка на объект, являющийся прототипом объекта A. Несложно догадаться, что речь идет о том же самом объекте, что и при использовании инструкции MyObject.prototype. Поэтому в сценарии при проверке значения выражения Object.getPrototypeOf(A)==MyObject.prototype получаем значение true. В итоге результат выполнения сценария будет следующим.

Результат выполнения сценария (из листинга 4.15)

Свойства созданных объектов:

name -> Объект A | number -> 100 |

name -> Объект B | number -> 200 |

После добавления свойства:

name -> Объект A | number -> 100 | color -> прозрачный |

name -> Объект B | number -> 200 | color -> прозрачный |

После удаления свойства:

name -> Объект A | number -> 100 | color -> белый |

name -> Объект B | number -> 200 |

Проверка прототипа объекта:

Object.getPrototypeOf(A)==MyObject.prototype -> true

Для включения сценария в веб-документ используем следующий HTML-код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 4.15</title>
</head>
<body><h3>Листинг 4.15</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing04_15.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Окно браузера с веб-документом, в котором выполняется описанный выше сценарий, представлено на рис. 4.14.

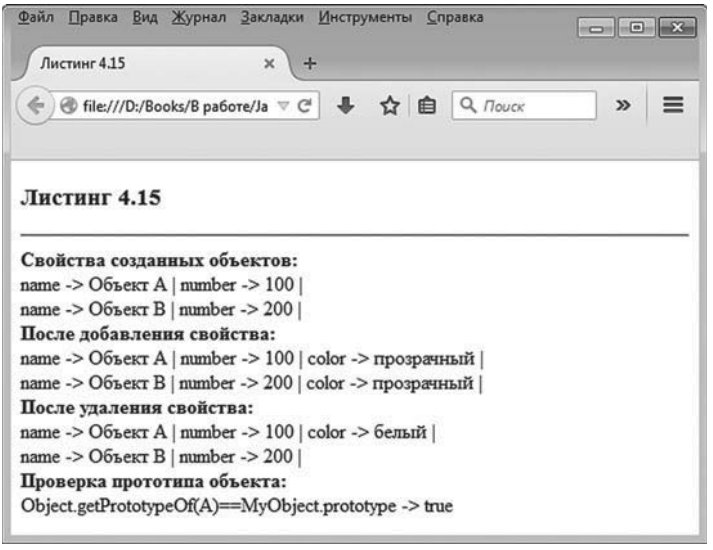

Рис. 4.14. Результат выполнения сценария, в котором используется прототип объектов, создаваемых на основе конструктора

ДЕТАЛИ

Выше речь шла о прототипе объектов, которые создаются с помощью конструктора MyObject. Но конструктор MyObject сам является объектом, и у него есть прототип. И это совсем не тот прототип, который у объектов, создаваемых с помощью конструктора. Другими словами, объекты MyObject.prototype (прототип объектов, создаваемых с помощью конструктора) и Object.getPrototypeOf(MyObject) (прототип конструктора) являются разными.

Выше мы рассматривали ситуацию, когда через конструктор получали доступ к прототипу, на основе которого создаются объекты. Ситуация может быть несколько иной, когда по объекту необходимо определить конструктор, который позволяет создавать объекты с таким же прототипом, как у данного. Чтобы через объект получить ссылку на его конструктор, используется свойство constructor объекта. В листинге 4.16 приведен программный код сценария, в котором используется означенный подход для получения доступа к конструктору объекта.

Листинг 4.16. Определение конструктора по объекту (файл Listing04_16.js)

```
// Конструктор:
function MyObj(){
  this.number=0
}
// Объект:
var A=new MyObj()
// Определение конструктора через объект:
var F=A.constructor
// Создание объекта:
var B=new F()
// Проверка значения свойства number объекта:
document.write("Свойство B.number = "+B.number+"<br>")
// Новый объект:
A={}
// Определение конструктора:
F=A.constructor
```

```
// Проверка объекта конструктора:
document.write("F==Object -> "+(F==Object)+"<br>")
// Создание объекта с помощью анонимного конструктора:
A=new function(){
  this.name="объект"
}()
// Определение конструктора:
F=A.constructor
// Создание объекта:
B=new F()
// Проверка значения свойства name объекта:
document.write("Свойство B.name = "+B.name)
```

В сценарии рассматривается три способа создания объектов: с помощью явно описанного конструктора, с помощью литерала и с помощью анонимного конструктора (это когда функция-конструктор является анонимной). В каждом из этих трех случаев с помощью свойства constructor, созданного тем или иным способом объекта, получаем доступ к конструктору и для проверки создаем с помощью «выловленного» таким способом конструктора еще один объект.

Итак, в сценарии описывается конструктор MyObj, в теле которого всего одна команда this.number=0. Таким образом, все создаваемые на основе данного конструктора объекты получают поле number с нулевым значением.

> **НА ЗАМЕТКУ**
>
> Поле number является собственным полем объекта.

Далее с помощью конструктора MyObj создается объект, для чего используется инструкция new MyObj() и ее результат записывается в переменную A. В переменную F записывается значение выражения A.constructor. Последнее представляет собой ссылку на конструктор, которым создавался объект A (то есть речь идет о конструкторе MyObj). Поэтому переменную F можно использовать так, как если бы это был конструктор (что в общем-то так и есть). Новый объект создается инструкцией new F(), а ее результат записывается в переменную B. Факти-

чески данный объект создан с помощью конструктора MyObj, поэтому у объекта B есть свойство number. Значение этого свойства проверяется в сценарии.

Новый пустой объект создается командой A={}. Доступ к конструктору объекта получаем с помощью команды F=A.constructor. При проверке условие F==Object оказывается, что оно истинно.

Истинность условия F==Object означает, что для объекта A, созданного описанием литерала, конструктором является объект Object.

Наконец, еще один объект создается с использованием анонимного конструктора. Переменной A присваивается инструкция, состоящая из ключевого слова new и вызова анонимной функции. Вся конструкция выглядит так:

```
A=new function(){
  this.name="объект"
}()
```

Непосредственно код анонимной функции следующий:

```
function(){
  this.name="объект"
}
```

Круглые скобки после описания анонимной функции в команде создания объекта означают, что она вызывается. В теле функции в силу наличия команды this.name="объект" создаваемый объект получает собственное свойство name со значением "объект". Но пикантность ситуации в том, что так как функция-конструктор анонимная, то она как бы «одноразовая»: мы ее вызвали лишь один раз, а ссылку на нее никуда не записали. Но функция все равно не «потеряется». О ней «знает» объект, созданный описанным способом. Воспользовавшись командой F=A.constructor, в переменную F записываем ссылку на функцию-конструктор, использованную при создании объекта, на который ссылается переменная A. Теперь можем с помощью этой функции создать новый объект, что и делается командой B=new F(). Непосредственной проверкой убеждаемся, что у объекта B есть свойство name со значением "объект".

Ниже представлен результат выполнения сценария.

Результат выполнения сценария (из листинга 4.16)

```
Свойство B.number = 0
F==Object -> true
Свойство B.name = объект
```

Для тестирования кода используем следующий веб-документ:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 4.16</title>
</head>
<body><h3>Листинг 4.16</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing04_16.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 4.15 показано, как выглядит веб-документ с результатом выполнения сценария.

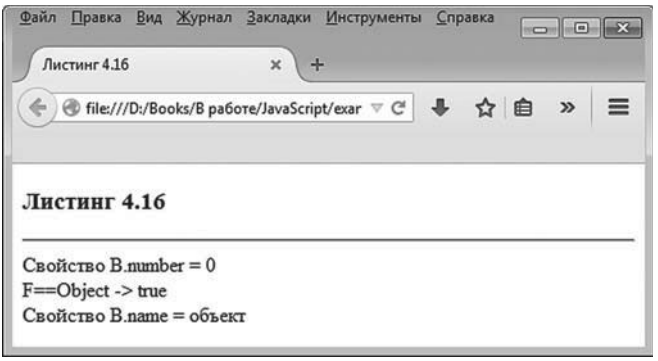

Рис. 4.15. Результат выполнения сценария, в котором по объекту определяется его конструктор

Далее мы рассмотрим некоторые аспекты, связанные с использованием свойств и методов объектов.

Свойства и методы

— Чего они от него хотят?
— Боятся за крайне хрупкую жизнь. Ступил человек на скользкий путь...
— Ну, на то и лед, чтоб скользить.

из к/ф «Покровские ворота»

Со свойствами и методами мы уже имели дело многократно, но, как будет показано далее, рассмотрели далеко не все темы, которые важны для понимания принципов их использования. Сейчас пришло время восполнить пробелы в знаниях.

Перечисляемые и неперечисляемые свойства

В рассмотренных ранее примерах мы обращались к свойству constructor объектов, которые создавали разными способами. Пикантность ситуации в том, что при создании объекта такое свойство в него не добавлялось, а если запустить оператор цикла for-in для перебора свойств объекта, то данное свойство отображено не будет. Но вот если мы проверим наличие этого свойства с помощью оператора in, то результатом будет значение true, что свидетельствует о наличии указанного свойства у объекта. Проще говоря, здесь мы сталкиваемся с ситуацией, когда свойство есть, но оно как-то «замалчивается».

Ранее мы уже отмечали, что свойства объектов делятся на две группы: *перечисляемые* и *неперечисляемые* свойства. При переборе свойств объекта с помощью оператора for-in обрабатываются только перечисляемые свойства, а неперечисляемые игнорируются. Если наличие свойства в объекте проверяется с помощью оператора in, то в расчет принимаются как перечисляемые, так и неперечисляемые свойства. По умолчанию свойства, добавляемые в объект непосредственно в сценарии, являются перечисляемыми, а многие свойства, наследуемые из стандартных прототипов, являются неперечисляемыми. Хотя положение дел может быть изменено. Как иллюстрацию к сказанному рассмотрим небольшой пример, в котором описанием литерала создается объект, а затем объект проверяется на наличие

в нем некоторых свойств. Программный код представлен в листинге 4.17.

Листинг 4.17. Перечисляемые и неперечисляемые свойства объекта (файл Listing04_17.js)

```
// Создание объекта:
var A={number:100}
// Добавление свойства в прототип:
Object.prototype.name="объект A"
for(var a in A){
  document.write(a+" | ")
}
// Проверка наличия у объекта свойств:
test('"toString" in A')
test('"valueOf" in A')
test('"constructor" in A')
// Функция для проверки наличия свойства у объекта:
function test(txt){
  document.write("<br>"+txt+" -> "+eval(txt))
}
```

В сценарии для проверки наличия у объекта определенного свойства описана и используется функция test(). Аргументом функции передается текстовое выражение, содержащее команду (предполагается, что это команда проверки наличия у объекта свойства). При вызове функции отображается команда (переданная в текстовом аргументе) и результат ее выполнения.

В сценарии объявляется переменная A, и значением ей присваивается объект со свойством number (команда var A={number:100}). Также командой Object.prototype.name="объект A" в прототип верхнего уровня добавляется свойство name. При переборе свойств объекта A с помощью оператора цикла for-in мы увидим названия свойств number (собственное свойство объекта A) и name (свойство объекта-прототипа). Но проверка объекта A на предмет наличия в нем свойств toString, valueOf и constructor показывает, что такие свойства у объекта тоже есть (собственно, первые два — это методы toString() и valueOf()).

ДЕТАЛИ

Со свойством constructor мы уже сталкивались — с его помощью получаем ссылку на объект-конструктор, которым создавался объект. Методы toString() и valueOf() мы еще не обсуждали.

Метод toString() наследуется объектами из прототипа Object.prototype и вызывается автоматически каждый раз, когда формат команды, в которой использована ссылка на объект, подразумевает использование текстового значения (например, когда объект передается аргументом методу write()).

Метод valueOf() также наследуется из прототипа Object.prototype и вызывается в тех случаях, когда объект должен преобразовываться в числовое (или иное, отличное от текстового) значение. Методы toString() и valueOf() еще будут обсуждаться.

Результат выполнения сценария представлен ниже.

Результат выполнения сценария (из листинга 4.17)

```
number | name |
"toString" in A -> true
"valueOf" in A -> true
"constructor" in A -> true
```

Для тестирования сценария полезным будет следующий HTML-код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 4.17</title>
</head>
<body><h3>Листинг 4.17</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing04_17.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Как выглядит в окне браузера документ с результатом выполнения сценария, показано на рис. 4.16.

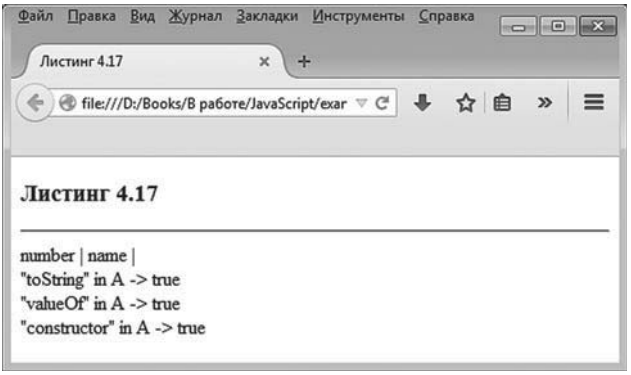

Рис. 4.16. Результат выполнения сценария, в котором проверяются перечисляемые и неперечисляемые свойства объекта

Таким образом, часть свойств отображается при переборе свойств объекта с помощью оператора for-in, а часть — нет. Критерий для дифференциации свойств — их «перечисляемойсть» или «неперечисляемость». Желательно иметь возможность как-то управлять этим процессом.

Далее мы рассмотрим подходы и методы, которые позволяют в той или иной степени влиять на характеристики свойств (такие как, например, «перечисляемость»).

В JavaScript свойства объектов имеют определенные характеристики, или *атрибуты*. В частности, для каждого свойства важны и поддерживаются на программном уровне следующие позиции.

- Собственно, значение свойства.
- Отображается ли свойство при переборе свойств объекта с помощью оператора for-in.
- Можно ли свойству присвоить значение.
- Можно ли удалить свойство.

Все перечисленные характеристики свойств допускается «регулировать» программными методами. Наиболее полное представление о характеристиках свойства объекта дает объект, который называется *дескриптором свойства*.

НА ЗАМЕТКУ

Другими словами, ситуация такая: имеется некоторое свойство с определенными атрибутами. Каковы значения этих атрибутов — определяется специальным объектом, который называется дескриптором свойства. У каждого свойства свой объект-дескриптор.

Объект-дескриптор по умолчанию имеет четыре свойства.

- Свойство value содержит значение, которое является фактическим значением свойства, описываемого объектом-дескриптором.
- Свойство enumerable определяет, относится свойство, описываемое дескриптором, к перечисляемым или к неперечисляемым. Свойство enumerable может принимать значение true (исходное свойство перечисляемое) или false (исходное свойство неперечисляемое).
- Свойство writable определяет, можно ли свойству, описываемому дескриптором, присвоить новое значение. Если значение свойства writable равно false, то исходное свойство фактически является константой — его значение изменить нельзя. Если значение свойства writable равно true, то исходному свойству можно присвоить новое значение.
- Свойство configurable может принимать значение true или false. Если значение свойства configurable равно false, то свойство, описываемое дескриптором, нельзя удалить из объекта. Чтобы исходное свойство можно было удалить из объекта, в дескрипторе свойства значение свойства configurable должно равняться true.

НА ЗАМЕТКУ

Свойство configurable объекта-дескриптора «отвечает» не только за возможность удалять описываемое дескриптором свойство, но и за возможность изменения типа свойства (его характеристик). Если свойство configurable объекта-дескриптора равняется false, то исходное свойство не только нельзя удалить, но еще и не получится изменить значения его атрибутов.

Ссылку на объект-дескриптор получают с помощью метода getOwnPropertyDescriptor() объекта Object. Аргументами методу передаются ссылки на объект и его свойство, для которого определяется дескриптор. Например, если нам необходимо получить ссылку на объект-де-

скриптор определенного свойства некоторого объекта, то соответствующая инструкция могла бы выглядеть как Object.getOwnPropertyDescriptor (объект,свойство). Результатом данного выражения является ссылка на объект-дескриптор.

НА ЗАМЕТКУ

С помощью метода getOwnPropertyDescriptor() можно получить ссылку на дескриптор только для собственного свойства объекта. Для получения дескрипторов для свойств, наследуемых объектом из прототипа, придется явно использовать объект-прототип.

Совсем небольшой пример, в котором для свойства объекта вычисляется объект-дескриптор, приведен в листинге 4.18.

Листинг 4.18. Получение доступа к дескриптору свойства (файл Listing04_18.js)

```
// Создание объекта:
var A={number:100}
// Дескриптор свойства number:
var descriptor=Object.getOwnPropertyDescriptor(A,"number")
// Отображение свойств объекта A:
show(A)
// Отображение свойств объекта descriptor:
show(descriptor)
// Функция для отображения свойств объекта:
function show(obj){
  document.write("{| ")
  for(var s in obj){
    document.write(" <b>"+s+"</b> : "+obj[s]+" |")
  }
  document.write("}<br>")
}
```

Результат выполнения сценария следующий.

Результат выполнения сценария (из листинга 4.18)

```
{| number : 100 |}
{| configurable : true | enumerable : true | value : 100 | writable : true |}
```

В сценарии создается объект A со свойством number. Значение свойства number равно 100. Для этого свойства командой Object.getOwnPropertyDescriptor (A,"number") вычисляется ссылка на объект-дескриптор. Ссылка на дескриптор записывается в переменную descriptor.

В сценарии описана функция show(), которой отображаются свойства объекта, переданного аргументом функции. Названия свойств выделяются жирным шрифтом, между названием и значением свойства размещается двоеточие. Вся конструкция заключается в фигурные кавычки, а разделителем между блоками с названием и значением свойств служит вертикальная черта.

В сценарии командами show(A) и show(descriptor) отображаются свойства для объектов A и descriptor. У объекта A всего одной свойство (имеется в виду свойство number). У объекта descriptor, являющегося дескриптором свойства number объекта A, четыре свойства.

- Свойство value объекта descriptor имеет значение 100: это фактическое значение свойства number объекта A.
- Свойство enumerable объекта descriptor имеет значение true (является перечисляемым): при переборе свойств объекта A с помощью оператора for-in свойство number попадает во множество перебираемых свойств.
- Свойство writable объекта descriptor имеет значение true: значение свойства number объекта A может быть изменено в сценарии.
- Свойство configurable объекта descriptor имеет значение true: свойство number объекта A может (при необходимости) быть удалено.

Для проверки работы сценария используем HTML-код, представленный ниже:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 4.18</title>
</head>
<body><h3>Листинг 4.18</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing04_18.js">
```

```
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 4.17 показано, как выглядит окно браузера с веб-документом, в котором выполняется описанный сценарий.

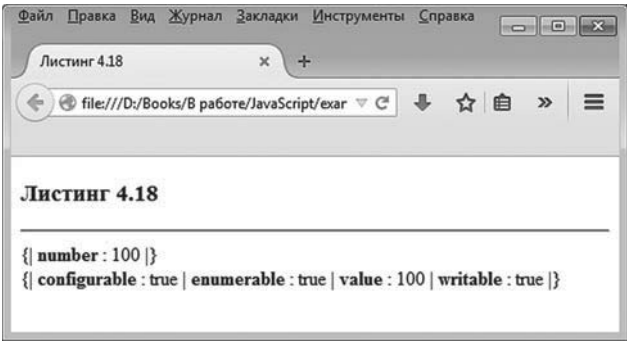

Рис. 4.17. Результат выполнения сценария, в котором для свойства вычисляется объект-дескриптор

В данном случае мы получили доступ к объекту-дескриптору, который содержит описание характеристик свойства. Возникает вопрос: а как задать (или изменить) значения атрибутов свойства? На этот случай есть метод defineProperty() объекта Object.

ДЕТАЛИ

Выше мы получали доступ к объекту-дескриптору, содержащему описание атрибутов некоторого свойства. Может возникнуть желание изменить значения атрибутов свойства непосредственно в объекте-дескрипторе. Хотя формально так можно сделать, на самом свойстве это не отразится. Причина в том, что методом getOwnPropertyDescriptor() возвращается одна из копий объекта-дескриптора. Поэтому внесение в нее изменений не сказывается на исходном свойстве.

Аргументами методу defineProperty() передается ссылка на объект, в который добавляется свойство, название добавляемого свойства, а так-

же объект-дескриптор данного свойства. Объект-дескриптор может быть передан в виде литерала объекта и может содержать не все свойства (имеются в виду свойства value, enumerable, writable и configurable объекта-дескриптора). Пример использования метода defineProperty() приведен в листинге 4.19.

Листинг 4.19. Использование метода defineProperty() (файл Listing04_19.js)

```
// Создание пустого объекта:
var A={}
// Добавление свойства number:
Object.defineProperty(A,"number",{value:100,enumerable:true,writable:true,configurable:true})
// Объект-дескриптор нового свойства:
var descriptor={value:"объект A",enumerable:true,writable:true,configurable:true}
// Добавление свойства name:
Object.defineProperty(A,"name",descriptor)
// Отображение свойств объекта A:
show(A)
// Изменение атрибутов свойства number:
Object.defineProperty(A,"number",{value:200,enumerable:false})
// Отображение свойств объекта A:
show(A)
// Отображение значения свойства number:
document.write("A.number = "+A.number)
// Функция для отображения свойств объекта:
function show(obj){
  document.write("{|")
  for(var s in obj){
    document.write(" <b>"+s+"</b> : "+obj[s]+" |")
  }
  document.write("}<br>")
}
```

В результате выполнения сценария в рабочем документе отображаются такие сообщения.

Результат выполнения сценария (из листинга 4.19)

```
{| number : 100 | name : объект A |}
{| name : объект A |}
A.number = 200
```

Сценарий начинается с команды создания пустого объекта A, после чего в объект добавляется свойство number. Свойство добавляем командой Object.defineProperty(A,"number",{value:100,enumerable:true,writable:true,configurable:true}). Здесь метод defineProperty() вызывается из объекта Object, а аргументами методу передаются:

- ссылка на объект A, в который добавляется свойство;
- название "number" добавляемого свойства;
- объект-дескриптор для добавляемого свойства. Объект задан в виде литерала {value:100,enumerable:true,writable:true,configurable:true}. В объекте-дескрипторе определяются атрибуты свойства number объекта A. В частности, свойство относится к перечисляемым (будет отображаться при переборе с помощью оператора цикла for-in), а его значение устанавливается равным 100.

Командой var descriptor={value:"объект A",enumerable:true,writable:true,configurable:true} создается объект-дескриптор, и ссылка на объект записывается в переменную descriptor.

Эта переменная используется в команде Object.defineProperty(A,"name", descriptor), которой в объект A добавляется свойство name. Затем с помощью функции show() (описание функции в конце сценария) отображаются названия и значения свойств объекта A. Поскольку в теле функции show() для перебора свойств объекта, переданного аргументом функции, используется оператор for-in, а свойства number и name создавались со значением true атрибута enumerable, то оба эти свойства отображаются.

На следующем этапе изменяются значения некоторых атрибутов свойства number. Нами использована команда Object.defineProperty(A,"number", {value:200,enumerable:false}). Формально она похожа на команду добавления свойства "number" в объект A, но здесь речь идет именно об изменении атрибутов свойства. Более того, в объекте-дескрипторе явно указаны значения только двух свойств (value и enumerable), определяющих новые значения для атрибутов свойства number объекта A. Результат проверя-

ем командой show(A). Поскольку теперь свойство number относится к неперечисляемым (так как новое значение свойства enumerable в объекте-дескрипторе равно false), то в списке свойств объекта A свойство number не отображается. Тем не менее такое свойство у объекта A есть, в чем легко убедиться с помощью команды document.write("A.number = "+A.number).

Для тестирования сценария используем следующий HTML-код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 4.19</title>
</head>
<body><h3>Листинг 4.19</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing04_19.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 4.18 показано, как выглядит веб-документ с результатом выполнения сценария.

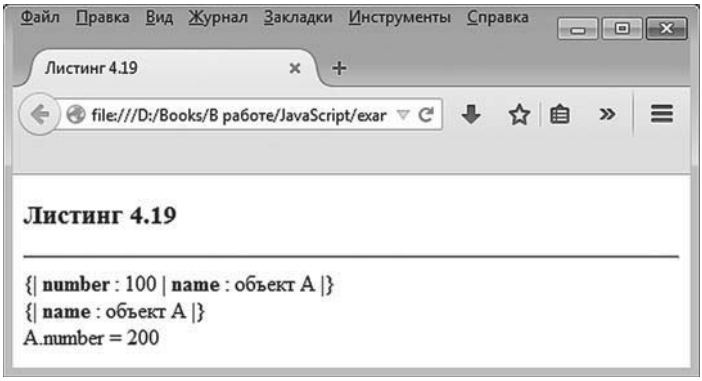

Рис. 4.18. Результат выполнения сценария, в котором используется метод *defineProperty()*

ДЕТАЛИ

Есть ряд методов объекта Object, которые бывают полезны при работе с объектами. Метод preventExtensions() применяет режим запрета для добавления свойств в объект, указанный аргументом метода. С помощью метода seal() блокируется добавление новых и удаление существующих свойств объекта, переданного аргументом методу (значение атрибута configurable для всех свойств объекта устанавливается равным false).

Методом freeze() блокируется добавление, удаление и изменение свойств объекта, переданного аргументом методу. При этом значение атрибутов configurable и writable для всех свойств объекта устанавливается равным false.

Методы isExtensible(), isSealed() и isFrozen() вызываются из объекта Object, аргументом им передается объект, а результатом является true, если для объекта ранее вызывались соответственно методы preventExtensions(), seal() и freeze() (в противном случае методами возвращается значение false).

Среди методов объекта Object имеется метод keys(). Результатом метода возвращается список перечисляемых свойств объекта. Объект передается аргументом методу. Результат возвращается в виде массива (массивы обсуждаются в следующей главе). Массив с названиями всех (не только перечисляемых) свойств объекта возвращается методом getOwnPropertyNames(). Аргументом методу передается ссылка на объект.

Полезным может быть метод hasOwnProperty(), позволяющий проверить, является ли свойство собственным свойством объекта. Метод вызывается из объекта, свойство которого проверяется. Название свойства передается аргументом методу. Результатом метода является значение true, если свойство является собственным свойством объекта, и false в противоположном случае.

Свойства с режимом доступа

Ранее мы рассматривали свойства, значения которых свободно считывались и свободно присваивались. Ключевой момент здесь связан с тем, что в принципе свойству можно присвоить какое угодно значение. Такой режим не всегда удобен. Нередко ситуация складывается так, что желательно ограничить возможности по считыванию и, самое главное, присваиванию значения свойству. В JavaScript есть механизм, который позволяет эмулировать свойства с помощью специальных методов, благодаря чему удается программными методами определить алгоритм присваивания значения свойству и считыва-

ния значения свойства. Технически все это реализуется с помощью двух методов: метода присваивания значения и метода считывания значения. Первый из означенных методов описывается с ключевым словом get (метод считывания значения), а второй — с ключевым словом set (метод для присваивания значения).

В частности, если мы хотим выполнить эмуляцию свойства, то метод для присваивания значения такому свойству выглядит так (ключевые элементы шаблона выделены жирным шрифтом):

```
get свойство(){
  // код получения доступа к значению
}
```

Метод для присваивания значения свойству описывается в соответствии со следующим шаблоном (жирным шрифтом выделены основные элементы кода):

```
set свойство(аргумент){
  // код получения доступа к значению
}
```

Важно понимать, что даже если для свойства описаны метод считывания значения (get-метод) и метод для присваивания значения (set-метод), то считывается значение свойства и присваивается значение свойству так, как если бы свойство было самым обычным: указывается имя объекта и через точку название свойства.

Небольшой пример, в котором иллюстрируются простые приемы по созданию и использованию свойств с режимом доступа, представлен в листинге 4.20.

Листинг 4.20. Свойства с режимом доступа (файл Listing04_20.js)

```
// Объект и свойство с режимом доступа:
var A={
  // Обычное свойство:
  name:"объект A",
  // Вспомогательное свойство:
  n:0,
  // Метод для считывания значения свойства number:
  get number(){
    return this.n%10
  },
```

```
  // Метод для присваивания значения свойству number:
  set number(x){
    this.n=(x%10)
  }
}
// Присваивание значения свойству с режимом доступа:
A.number=123
// Считывание значения свойства с режимом доступа:
document.write(A.name+"<br>"+A.number+"<hr>")
// Присваивание значения свойству с режимом доступа:
A.number=5
// Считывание значения свойства с режимом доступа:
document.write(A.name+"<br>"+A.number+"<hr>")
// Присваивание значения вспомогательному свойству:
A.n=12
// Считывание значения свойства с режимом доступа:
document.write(A.name+"<br>"+A.number+"<hr>")
```

Результат выполнения сценария представлен на рис. 4.19.

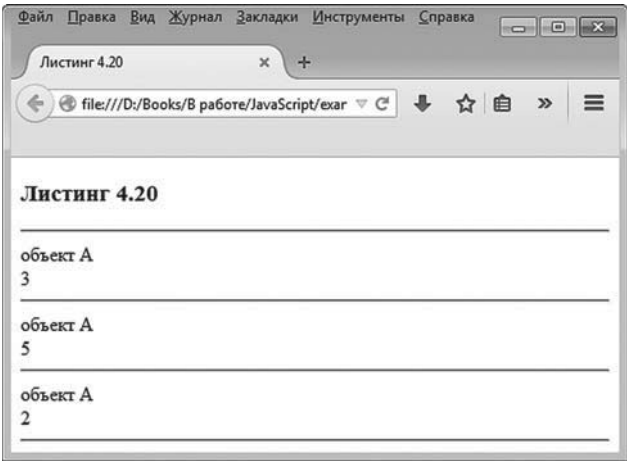

Рис. 4.19. Результат выполнения сценария, в котором у объекта создается свойство с режимом доступа

Тестируем сценарий с помощью следующего HTML-кода:

```
<!DOCTYPE HTML>
<html>
<head>
```

```
  <title>Листинг 4.20</title>
</head>
<body><h3>Листинг 4.20</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing04_20.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Что касается самого сценария, то он, как отмечалось, небольшой. Самое интересное в нем — описание объекта A. Объект создается описанием литерала объекта. У объекта имеется обычное свойство name с текстовым значением "объект A", числовое «вспомогательное» свойство n с начальным значением 0, а также свойство number с режимом доступа. Нас интересует в первую очередь свойство number. Создается оно описанием методов доступа к свойству. Метод для считывания значения свойства number описывается с get-инструкцией следующим образом:

```
get number(){
  return this.n%10
}
```

В соответствии с приведенным программным кодом результатом get-метода (оно же считываемое значение свойства number) возвращается остаток от деления значения поля n на 10 (целое число в диапазоне значений от 0 до 9).

Что происходит при присваивании значения полю number, можно понять, проанализировав программный код set-метода:

```
set number(x){
  this.n=(x%10)
}
```

У метода есть аргумент (обозначен как x) — значение, что присваивается свойству number. Но на самом деле командой this.n=(x%10) свойству n объекта присваивается остаток от деления на 10 значения, формально присваиваемого свойству number.

ДЕТАЛИ

Таким образом, у объекта A есть свойство n. При обращении к свойству number для считывания или присваивания значения на самом деле выполняется обращение к свойству n. При этом используются определенные алгоритмы считывания значения и записи значения. Так, при присваивании значения полю number в поле n записывается остаток от деления на 10 числа, которое присваивается полю number. При считывании значения поля number возвращается остаток от деления на 10 числа, записанного в поле n.

Следует также учесть, что никто не запрещает нам изменить значение свойства n, не обращаясь к полю number. Другими словами, мы можем изменить непосредственно значение поля n (командой вида A.n=значение), что повлечет за собой изменение значения поля number. Данная ситуация еще будет нами обсуждаться.

При присваивании значения полю number командой A.number=123 технически в поле n записывается значение 3 (остаток от деления числа 123 на 10). Оно же возвращается при считывании значения поля number в команде document.write(A.name+"
"+A.number+"<hr>"). Аналогичная ситуация имеет место при выполнении команд A.number=5 и document.write (A.name+"
"+A.number+"<hr>").

Важная особенность представленного кода иллюстрируется выполнением команд A.n=12 и document.write(A.name+"
"+A.number+"<hr>"). Здесь мы напрямую присваиваем значение свойству n и после этого, когда считывается значение свойства number, получаем значение 2 (остаток от деления числа 12 на 10).

НА ЗАМЕТКУ

Подобных ситуаций, когда значение свойства (в данном случае number) может изменяться неявно, лучше избегать. В таких языках программирования, как C++, C# и Java, нужного эффекта добиваются с помощью механизма ограничения доступа к «техническим» и «вспомогательным» полям. В JavaScript нет простого механизма создания закрытых полей. Тем не менее решение у проблемы есть, но его мы обсудим немного позже.

В приведенном выше примере мы описывали свойство с режимом доступа в литерале объекта. Подобное свойство также можно добавить в объект методом defineProperty(), описав атрибуты свойства с помощью объекта-дескриптора.

ДЕТАЛИ

Напомним, что в объекте-дескрипторе описываются четыре свойства: value, enumerable, writable и configurable. Но если в объекте-дескрипторе описывается свойство с режимом доступа (такой объект-дескриптор содержит get-метод и set-метод), то свойства value и writable в объекте-дескрипторе не описываются.

Альтернативный способ реализации рассмотренного выше примера (но теперь с использованием объекта-дескриптора) представлен в листинге 4.21.

Листинг 4.21. Свойства с режимом доступа и объект-дескриптор (файл Listing04_21.js)

```
// Создание объекта:
var A={name:"объект A",n:0}
// Добавление в объект свойства с режимом доступа:
Object.defineProperty(A,"number",{
  get:function(){
    return this.n%10
  },
  set:function(x){
     this.n=(x%10)
  }
})
// Присваивание значения свойству с режимом доступа:
A.number=123
// Считывание значения свойства с режимом доступа:
document.write(A.name+"<br>"+A.number+"<hr>")
// Присваивание значения свойству с режимом доступа:
A.number=5
// Считывание значения свойства с режимом доступа:
document.write(A.name+"<br>"+A.number+"<hr>")
// Присваивание значения вспомогательному свойству:
A.n=12
// Считывание значения свойства с режимом доступа:
document.write(A.name+"<br>"+A.number+"<hr>")
```

Результат выполнения данного сценария абсолютно такой же, как в предыдущем случае.

НА ЗАМЕТКУ

Для тестирования сценария можно воспользоваться стандартным HTML-шаблоном, который мы использовали ранее. Достаточно лишь изменить название загружаемого файла со сценарием.

По сравнению с предыдущим примером, здесь несколько иначе описывается объект A. В частности, сначала описанием литерала создается объект со свойствами name и n, а затем вызовом метода defineProperty() в объект добавляется свойство с режимом доступа number. Интерес представляет дескриптор свойства, переданный третьим аргументом методу defineProperty(). С формальной точки зрения в дескрипторе описываются методы get и set. То, что это методы доступа к свойству number, становится понятным, исходя из значения второго аргумента метода defineProperty().

Изменение способа создания объекта A со свойством доступа number не снимает главной проблемы: технически все операции со свойством number выполняются через свойство n, которое является открытым в том смысле, что с ним можно выполнять любые операции. Это все равно, как если бы мы строили дом на ненадежном фундаменте. Самый простой способ решить проблему состоит в том, чтобы «спрятать» свойство n объекта. Мы воспользуемся особенностью локальных переменных, которые объявляются в функции и доступны только в пределах функции. Такой «маскирующей» функцией станет конструктор объектов. Рассмотрим программный код, представленный в листинге 4.22. В данном примере для создания объекта используется функция-конструктор.

Листинг 4.22. Свойства с режимом доступа и конструктор объектов (файл Listing04_22.js)

```
// Конструктор для создания объектов:
function MyObj(name){
  // Локальная переменная:
  var n
  // Свойство name объекта:
  this.name=name
  // Добавление в объект свойства
  // number с режимом доступа:
  Object.defineProperty(this,"number",{
    // Метод для считывания значения свойства:
    get:function(){
      return n%10
```

```
    },
    // Метод для присваивания значения свойству:
    set:function(x){
      n=(x%10)
    }
  })
} // Окончание описания функции-конструктора
// Создание объекта:
var A=new MyObj("объект A")
// Присваивание значения свойству с режимом доступа:
A.number=123
// Считывание значения свойства с режимом доступа:
document.write(A.name+"<br>"+A.number+"<hr>")
// Присваивание значения свойству с режимом доступа:
A.number=5
// Считывание значения свойства с режимом доступа:
document.write(A.name+"<br>"+A.number+"<hr>")
// Присваивание значения свойству (добавление свойства):
A.n=12
// Считывание значения свойства с режимом доступа:
document.write(A.name+"<br>"+A.number+"<hr>")
```

Для тестирования сценария используем следующий шаблонный код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 4.22</title>
</head>
<body><h3>Листинг 4.22</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing04_22.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 4.20 показан результат выполнения сценария.

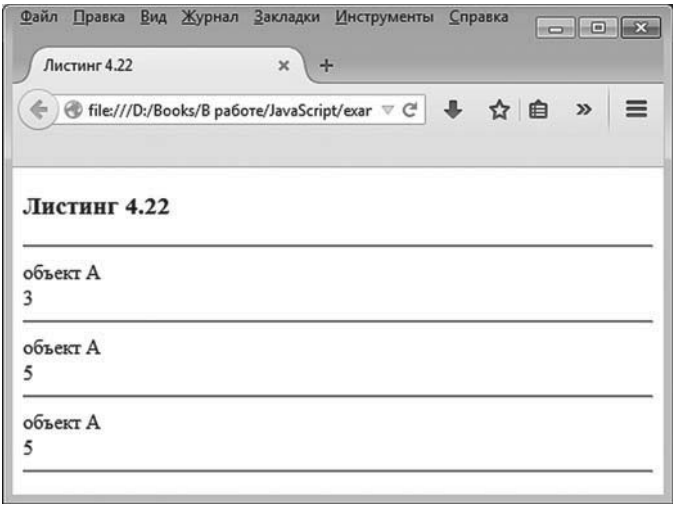

Рис. 4.20. Результат выполнения сценария, в котором свойство с режимом доступа добавляется в конструкторе объекта

Функция-конструктор, описанная в сценарии, называется MyObj(). У нее один аргумент (обозначен как name). Это значение для свойства name объекта, который создается с помощью функции-конструктора (в теле конструктора выполняется команда this.name=name).

В функции-конструкторе объявляется локальная переменная n. Ей не присваивается значение, но она «упоминается» в объекте-дескрипторе, который, в свою очередь, передается третьим аргументом методу defineProperty() (первые два — ссылка this на объект, в который добавляется свойство, и название свойства "number").

В объекте-дескрипторе описывается get-метод:

```
get:function(){
  return n%10
}
```

и set-метод

```
set:function(x){
  n=(x%10)
}
```

Здесь важно то, что операции выполняются не со свойством объекта, а с локальной переменной n (в частности, результат get-метода вычисляется на основе значения локальной переменной n функции-конструктора). Поэтому после вызова функции-конструктора при создании объекта локальная переменная n из памяти не выгружается и ее можно использовать (не напрямую, но через вызовы методов) и после завершения выполнения функции-конструктора.

НА ЗАМЕТКУ

У созданного с помощью функции-конструктора MyObj() объекта нет свойства n. Это принципиальный момент. Переменная, которая неявно используется при работе со свойством number, не является свойством объекта, и к ней в общем-то нет прямого доступа.

Объект в данном случае создается командой var A=new MyObj("объект A"). Далее следуют знакомые нам команды, которыми присваивается значение свойству number и считывается значение данного свойства. Результат выполнения таких команд уже описывался и должен быть понятен читателю. Что требует пояснения, так это результат выполнения команды A.n=12. Поскольку у объекта A свойства n нет, то выполнение данной команды приводит к добавлению такого свойства в объект. Но поскольку операции со свойством number теперь не подразумевают использование свойства n, то на значении свойства number наличие или отсутствие свойства n никак не сказывается.

Еще одно небольшое замечание касается того, что совсем необязательно описывать для свойства с режимом доступа get-метод и set-метод. Небольшой пример, в котором свойство с режимом доступа может быть только считано (для него описан только get-метод), приведен в листинге 4.23.

Листинг 4.23. Свойство с режимом доступа предназначено только для считывания значения (файл Listing04_23.js)

```
// Создание объекта:
var z={re:3,im:4}
// Добавление в объект свойства abs с режимом доступа:
Object.defineProperty(z,"abs",{
  get:function(){
    return Math.sqrt(this.re*this.re+this.im*this.im)
  }
```

```
})
// Отображение результата вычислений:
document.write("z = "+z.re+" + "+z.im+"i<br>")
document.write("|z| = "+z.abs)
```

Тестируем данный сценарий с помощью такого HTML-кода:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 4.23</title>
</head>
<body><h3>Листинг 4.23</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing04_23.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Результат выполнения сценария представлен на рис. 4.21.

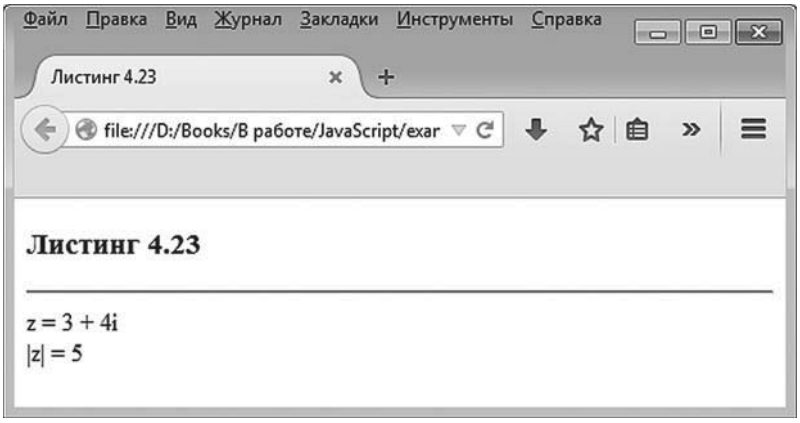

Рис. 4.21. Результат выполнения сценария, в котором свойство с режимом доступа предназначено только для считывания значения

В данном примере мы сначала создаем командой var z={re:3,im:4} объект z со свойствами re и im, а затем добавляем в объект свойство abs с режимом доступа.

Для свойства abs предусмотрен только get-метод, поэтому свойство доступно только для считывания значения.

ДЕТАЛИ

Речь идет о вычислении модуля комплексного числа. Если комплексное число $z = x + iy$, где $x = Re(z)$ и $y = Im(z)$,соответственно, действительная и мнимая части комплексного числа, а мнимая единица i такая, что $i^2 = -1$, то модулем комплексного числа называется значение $|z| = \sqrt{x^2 + y^2}$. Именно оно и вычисляется в get-методе, описанном для свойства abs объекта z.

Для вычисления квадратного корня использован метод sqrt() объекта Math. Ссылка в теле get-метода на объект, из которого метод фактически вызывается, реализована через ключевое слово this.

По большому счету в данном примере мы реализовали свойство с режимом доступа по аналогии с методом, который вызывается без аргументов и возвращает результат, вычисляемый на основе значений прочих свойств объекта.

На этом мы заканчиваем главу, но не заканчиваем обсуждение принципов ООП. Мы вернемся к обсуждению объектов и методов работы с ними, но перед этим рассмотрим очень важную тему, которая также имеет непосредственное отношение к ООП, — в следующей главе речь пойдет о *массивах*.

Резюме

> Сейчас я вам покажу, где здесь хунд бегрaбен!
>
> *из к/ф «Покровские ворота»*

Таким образом, выше мы узнали следующее.

- В JavaScript поддерживается концепция объектно-ориентированного программирования. В JavaScript используются объекты, но нет классов (в их привычном понимании).

- Существует несколько способов создания объекта: через описание литерала объекта, с помощью конструктора объектов Object, с помощью метода create() объекта Object или с помощью функции-конструктора.
- Литерал объекта описывается так: в фигурных скобках перечисляются через запятую пары из названий свойств и их значений. Название свойства и его значение разделяются двоеточиями.
- Метод представляет собой свойство, значением которого является функция. Для ссылки на объект, из которого вызывается метод, используется ключевое слово this.
- Переменные ссылаются на объект. При присваивании объектов копирования объекта не происходит, просто ссылка на объект из одной переменной копируется в другую (в результате две переменные ссылаются на один и тот же объект).
- Если несуществующему свойству объекта присваивается значение, данное свойство добавляется в объект. Для удаления свойств используют инструкцию delete.
- Функция-конструктор объектов обычно содержит команды создания свойств объектов. Если в сценарии описана функция-конструктор, то объекты создаются с помощью оператора new и вызова функции-конструктора.
- Ряд операторов упрощает обработку объектов. Среди них оператор with (позволяет не использовать явную ссылку на объект), for-in (используется для перебора свойств объекта), in (используется для проверки наличия у объекта данного свойства).
- Прототипом называется объект, свойства которого наследуются в другом объекте. При обращении к свойству объекта (для считывания значения) поиск свойства сначала выполняется непосредственно в объекте, затем в его прототипе, затем в прототипе прототипа и так далее. В соответствии с этим алгоритмом все свойства объекта делятся на собственные свойства и свойства, унаследованные из прототипов. Также свойства делятся на перечисляемые и неперечисляемые. Перечисляемые свойства отображаются в операторе for-in, а неперечисляемые — нет.
- Для создания объекта на основе прототипа можно прототип передать аргументом методу create() объекта Object (если методу create() передать аргумент null, объект создается без прототипа). Для по-

лучения доступа к прототипу используют метод getPrototypeOf(). Доступ к прототипу верхнего уровня получаем инструкцией Object.prototype. Применить прототип к созданному объекту можно с помощью метода setPrototypeOf() объекта Object.

- Объекты, созданные с помощью функции-конструктора, имеют общий прототип. Доступ к данному прототипу можно получить через свойство prototype объекта функции-конструктора. Прототип объекта, созданного с помощью функции-конструктора, отличается от прототипа самой функции-конструктора.
- Свойство constructor позволяет получить доступ к функции-конструктору, с помощью которой создавался объект.
- Для каждого свойства имеется объект, содержащий описание данного свойства. Такой объект называется дескриптором. У дескриптора имеются следующие основные свойства: value (значение исходного, описываемого дескриптором свойства), enumerable (принадлежность исходного свойства к перечисляемым), writable (возможность присваивать свойству значение) и configurable (возможность удалить свойство).
- Доступ к дескриптору собственного свойства объекта получаем с помощью метода getOwnPropertyDescriptor() объекта Object. Для создания у объекта свойства на основе явно заданного дескриптора используют метод defineProperty() объекта Object.
- Существует возможность создавать свойства со специальным режимом доступа. Для таких свойств считывание и запись значения реализуются через специальные методы: get-метод определяет процесс считывания значения свойства, а set-метод определяет процесс присваивания значения свойству. Для свойств с режимом доступа в объекте-дескрипторе свойства value и writable не определяются. Также нет необходимости описывать оба метода доступа.

Глава 5
ЗНАКОМСТВО С МАССИВАМИ

Натурально как вы играете... И царь у вас такой... типичный! На нашего Буншу похож.

из к/ф «Иван Васильевич меняет профессию»

Наконец пришло время познакомиться нам с *массивами*. Несмотря на существенную важность массивов, мы оттягивали знакомство с ними по одной простой причине: массивы в JavaScript уж очень специфичны. По большому счету, массив в JavaScript — это объект, но с определенными нестандартными свойствами. Чтобы не потерять ключевые моменты при упрощенном рассмотрении массивов, мы сначала познакомились с объектами и только теперь приступаем к рассмотрению массивов.

Создание массива

Что же вы, маэстры, молчите? Ну-ка, грянь-те нам что-нибудь.

из к/ф «Иван Васильевич меняет профессию»

Массив представляет собой упорядоченный набор элементов. Принципиально важно то, что в JavaScript элементы массива могут относиться к разным типам. Доступ к элементам массива получают по индексу или индексам. Количество индексов, необходимых для идентификации элемента массива, определяет *размерность* массива. Мы сначала рассмотрим *одномерные* массивы, в которых элемент имеет только один индекс. Количество элементов в массиве называется *размером* массива.

Явное указание элементов массива

Самый простой способ создания массива подразумевает явное указание его элементов. Элементы массива указываются в виде списка, за-

ключенного в квадратные скобки. Элементы между собой разделяются запятыми. Если всю такую конструкцию, состоящую из списка элементов, заключенных в квадратные скобки, присвоить в качестве значения переменной, то данную переменную можно отождествлять с массивом.

ДЕТАЛИ

На самом деле переменная, которой значением присваивается массив, содержит ссылку на него. Ситуация напоминает работу с объектами. Здесь нет ничего удивительного, ведь массив является объектом. Как мы увидим далее, такая особенность массивов в JavaScript имеет важные последствия.

Таким образом, команда создания массива и записи ссылки на него в некоторую переменную может выглядеть следующим образом:

```
var переменная=[значение_1,значение_2,...,значение_N]
```

Чтобы получить доступ к элементу одномерного массива, после имени массива (переменной, которая ссылается на массив) в квадратных скобках указывают индекс элемента. Важно помнить, что по умолчанию индексация элементов начинается с нуля, так что индекс первого элемента массива равен нулю, а индекс последнего элемента массива на единицу меньше размера массива (или его длины). Длина же массива определяется свойством length объекта массива: чтобы узнать длину массива, после имени переменной, ссылающейся на массив, через точку необходимо указать свойство length.

Небольшой пример с созданием массива путем явного указания его элементов представлен в листинге 5.1.

Листинг 5.1. Создание массива явным указанием элементов (файл Listing05_01.js)

```
document.write("<h4>Создание массива</h4>")
// Создание массива:
var nums=[10,true,30,"текст",75]
// Отображение содержимого массива:
document.write(nums+"<br>")
for(var k in nums){
  document.write(nums[k]+" | ")
}
```

Проверить результат выполнения сценария нам поможет веб-документ с представленным ниже кодом:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 5.1</title>
</head>
<body><h3>Листинг 5.1</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing05_01.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 5.1 показано, как будет выглядеть данный веб-документ в окне браузера.

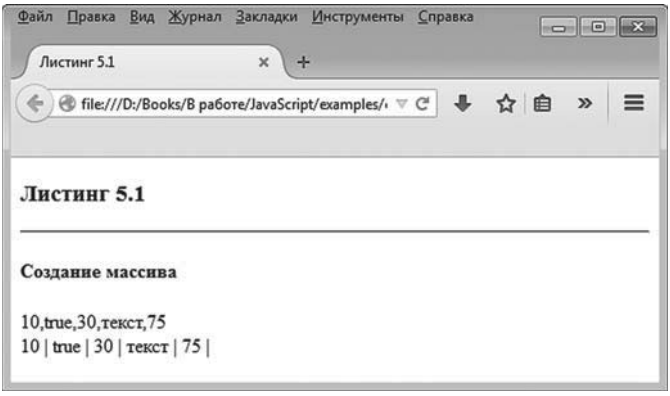

Рис. 5.1. Результат выполнения сценария, в котором путем явного указания элементов создается массив

Проанализируем основные фрагменты программного кода сценария. Он достаточно простой и, скорее всего, понятный. Создается массив командой `var nums=[10,true,30,"текст",75]`. В этой команде объявляется пере-

менная nums, а значением ей присваивается выражение [10,true,30,"текст",75]. Последнее представляет собой массив, который состоит из пяти элементов, три из которых являются целыми числами, второй (по порядку) является логическим значением, а четвертый (по порядку) элемент представляет собой текстовое значение.

Если имя массива передать аргументом методу write() объекта документа document, то в рабочем документе будет отображено содержимое массива, но не в очень эстетическом виде. Поэтому для отображения элементов массива разумнее использовать оператор цикла. В данном случае мы используем оператор for-in, позволяющий, кроме прочего, выполнять перебор по элементам массива. В круглых скобках после ключевого слова for указано выражение var k in nums, что буквально означает следующее: переменная k последовательно принимает значения индексов элементов, входящих в массив nums. Поэтому в теле оператора цикла переменную k можно интерпретировать как индекс очередного элемента массива. Отсюда инструкция nums[k] представляет собой обращение к элементу массива с соответствующим индексом.

Добавление элементов в массив

Надо сказать, что массивы в JavaScript исключительно гибкие в том смысле, что с ними можно выполнять различные интересные операции, не характерные для массивов как таковых (если сравнивать с другими языками программирования). Например, создать массив можно следующим образом: сначала создается пустой массив, а затем в этот массив последовательно добавляются элементы. Проиллюстрируем сказанное с помощью листинга 5.2, в котором представлена небольшая вариация на тему предыдущего примера.

Листинг 5.2. Создание пустого массива и добавление элементов в массив (файл Listing05_02.js)

```
document.write("<h4>Добавление элементов в массив</h4>")
// Создание пустого массива:
var nums=[]
// Добавление элементов в массив:
nums[0]=10
nums[1]=true
nums[2]=30
```

```
nums[3]="текст"
nums[4]=75
// Отображение содержимого массива:
for(var k=0;k<nums.length;k++){
  document.write(nums[k]+" | ")
}
```

В данном сценарии командой var nums=[] создается пустой массив (массив, не содержащий элементов). Далее следует серия команд, которыми элементам (несуществующим) массива присваиваются значения. Как и в случае с объектами, для которых присваивание значения несуществующему свойству приводит к добавлению такого свойства в объект, при присваивании несуществующему элементу массива значения такой элемент добавляется в массив. Так, например, командой nums[0]=10 присваивается значение первому элементу массива (напомним, что индексация элементов начинается с нуля), командой nums[1]=true присваивается значение второму элементу массива и так далее.

НА ЗАМЕТКУ

Естественным образом возникает вопрос: а что будет, если не присвоить значение элементу с каким-нибудь «внутренним» индексом? Например, если присвоить значение элементу с индексом 4, но не присвоить значение элементу с индексом 3. Ответ простой: ничего не будет. Точнее, не будет элемента, которому не присвоено значение. При этом размер массива определяется по самому большому индексу, который присутствует в массиве (длина массива на единицу больше этого индекса).

После добавления элементов в массив запускается «классический» оператор цикла, в котором индексная переменная k пробегает значения от 0 включительно и не превышает значение nums.length (длина массива, а индекс последнего элемента на единицу меньше). За каждый цикл отображается значение элемента массива с данным индексом.

Тестирование работы сценария выполняем с помощью следующего веб-документа:

```
<!DOCTYPE HTML>
<html>
<head>
```

```
  <title>Листинг 5.2</title>
</head>
<body><h3>Листинг 5.2</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing05_02.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 5.2 показано, как будет выглядеть веб-документ со сценарием, рассмотренным выше.

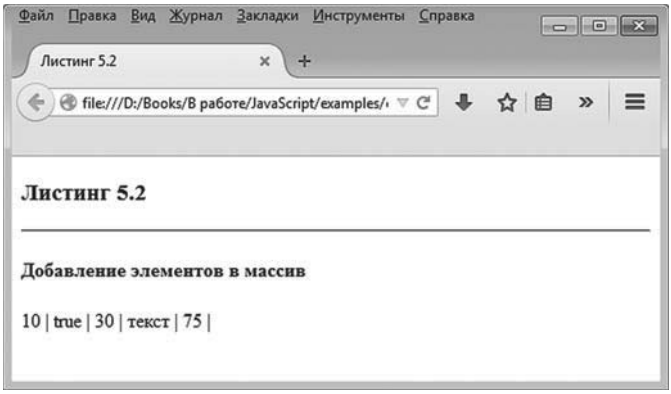

Рис. 5.2. Результат выполнения сценария, в котором создается пустой массив с последующим добавлением элементов

В обоих рассмотренных случаях элементы массива указывались в явном виде. Существует еще один способ создания массива по предопределенному набору значений. Подразумевает он использование конструктора объекта Array.

Использование объекта-конструктора Array

Массив может быть создан так, как создаются объекты в JavaScript, — с помощью конструктора. Массивы создаются вызовом функции-

конструктора Array(). Если значения элементов массива известны, то они передаются аргументами конструктору. Если аргументы не передавать, то создается пустой массив.

ДЕТАЛИ

Если конструктору Array() передается один целочисленный аргумент, то такой аргумент определяет размер массива. Например, командой new Array(5,6,7) создается массив из трех элементов со значениями 5, 6 и 7 соответственно, а вот командой new Array(5) создается массив из пяти элементов (значения элементов не определены). В то же время командой new Array("текст") создается массив из одного элемента, и значение элемента равняется "текст".

В листинге 5.3 показано, как уже знакомый нам массив мог бы быть создан с помощью конструктора Array().

Листинг 5.3. Создание массива с помощью конструктора Array() (файл Listing05_03.js)

```
document.write("<h4>Массив</h4>")
// Создание массива:
var nums=new Array(10,true,30,"текст",75)
// Отображение содержимого массива:
for(var k=0;k<nums.length;k++){
  document.write(nums[k]+" | ")
}
```

С помощью представленного ниже HTML-кода проверяем работу сценария:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 5.3</title>
</head>
<body><h3>Листинг 5.3</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing05_03.js">
</script>
```

```
<!-- Завершение сценария -->

</body>
</html>
```

Результат выполнения сценария представлен на рис. 5.3.

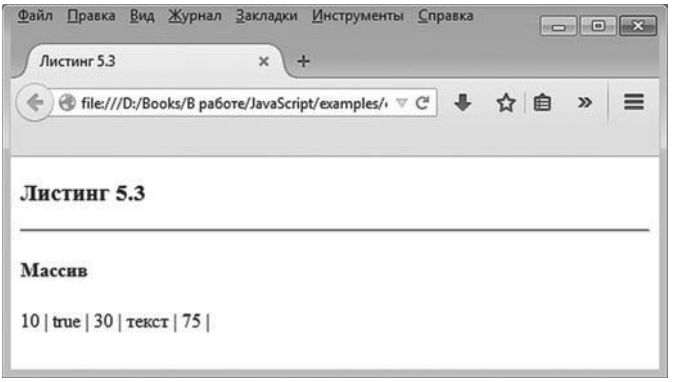

Рис. 5.3. Результат выполнения сценария, в котором массив создается с помощью конструктора *Array()*

Явно указать значения элементов массива обычно можно лишь в том случае, если массив небольшой и, соответственно, элементов в нем немного. Если элементов в массиве много, но при этом их значения подчиняются некоторой логике, разумно использовать оператор цикла. Как пример такого подхода рассмотрим сценарий, в котором создается массив, заполненный числами из последовательности Фибоначчи.

> **НА ЗАМЕТКУ**
>
> В последовательности Фибоначчи первые два числа равны единице, а каждое следующее число равняется сумме двух предыдущих.

Программный код сценария представлен в листинге 5.4.

Листинг 5.4. Заполнение массива числами Фибоначчи (файл Listing05_04.js)

```
document.write("<h4>Числа Фибоначчи</h4>")
// Размер массива и индексная переменная:
var n=15,k
```

```
// Создание массива из двух элементов:
var fibs=new Array(1,1)
// Заполнение массива числами:
for(k=2;k<n;k++){
  fibs[k]=fibs[k-1]+fibs[k-2]
}
// Отображение содержимого массива:
for(k=0;k<fibs.length;k++){
  document.write(fibs[k]+" | ")
}
```

Веб-документ, с помощью которого проверяется работа сценария, имеет следующий HTML-код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 5.4</title>
</head>
<body><h3>Листинг 5.4</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing05_04.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 5.4 показано, как выглядит результат выполнения сценария, в котором массив заполняется числами Фибоначчи.

В сценарии объявляется переменная k, которую мы будем использовать в операторе цикла, и переменная n со значением 15, которая определяет размер массива (количество чисел в последовательности, которые мы планируем вычислить). Командой var fibs=new Array(1,1) создается массив fibs из двух элементов с единичными значениями. Это

первые два числа в последовательности Фибоначчи. Затем запускается оператор цикла, в котором индексная переменная k пробегает значения от 2 (третий элемент массива) до n-1 (если в массив нужно записать n чисел, то индекс последнего элемента равен n-1). В теле оператора цикла командой fibs[k]=fibs[k-1]+fibs[k-2] значение элемента с индексом k вычисляется на основе значений двух предыдущих элементов. Причем на момент присваивания значения элемента с индексом k еще не существует, так что этому элементу не просто присваивается значение, а элемент перед этим еще и создается (добавляется в массив).

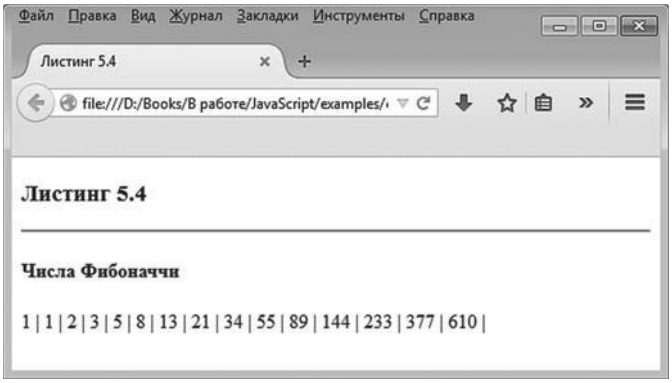

Рис. 5.4. Результат выполнения сценария, в котором массив заполняется числами Фибоначчи

По завершении оператора цикла, когда массив заполнен (а более корректно сказать — создан и заполнен), запускается еще один оператор цикла, в котором элементы массива отображаются в рабочем документе. В этом операторе индексная переменная пробегает значения от 0 (первый элемент) до fibs.length-1. Для определения размера массива fibs мы использовали свойство length, а индекс последнего элемента, как известно, на единицу меньше.

Для сравнения мы приведем еще несколько возможных способов реализации сценария, в котором создается массив и заполняется числами Фибоначчи (для сокращения объема программного кода комментарии удалены).

В листинге 5.5 приведен пример сценария с заполнением массива числами Фибоначчи, в котором при вычислении новых значений в массиве используется свойство length массива (значением свойства является количество элементов в массиве).

Листинг 5.5. Заполнение массива числами Фибоначчи (файл Listing05_05.js)

```
document.write("<h4>Числа Фибоначчи</h4>")
var n=15
var fibs=new Array(1,1)
while(fibs.length<n){
  fibs[fibs.length]=fibs[fibs.length-1]+fibs[fibs.length-2]
}
for(var k=0;k<fibs.length;k++){
  document.write(fibs[k]+" | ")
}
```

Массив создается командой var fibs=new Array(1,1). То есть на начальном этапе массив содержит два элемента. Затем запускается оператор цикла while, в котором выполняется всего одна команда fibs[fibs.length]=fibs[fibs.length-1]+fibs[fibs.length-2], которой добавляется новый элемент в конец массива. Процесс продолжается, пока истинно условие fibs.length<n (то есть пока длина массива не превышает значение переменной n). После заполнения массива значениями они отображаются в рабочем документе, для чего запускается еще один оператор цикла.

ДЕТАЛИ

В команде fibs[fibs.length]=fibs[fibs.length-1]+fibs[fibs.length-2] мы воспользовались тем обстоятельством, что размер массива (количество элементов в массиве) на единицу больше индекса последнего элемента в массиве. Размер массива возвращается свойством length. Если речь идет о массиве fibs, то его последний элемент имеет индекс fibs.length-1, а предпоследний элемент имеет индекс fibs.length-2. Соответственно, сумма последнего и предпоследнего элементов массива дается выражением fibs[fibs.length-1]+fibs[fibs.length-2]. Данное выражение присваивается значением элементу fibs[fibs.length]. Такого элемента нет, поскольку его индекс на единицу больше значения индекса последнего элемента в массиве. Но, поскольку ему присваивается значение, элемент в массив будет добавлен. Соответственно, размер массива увеличивается, и на следующей итерации свойство length будет возвращать уже новую длину массива. Поскольку в массиве должно быть n элементов, то перед последней итерацией размер массива fibs должен быть n-1. Отсюда получаем условие выполнения оператора цикла fibs.length<n (эквивалентно условию fibs.length<=(n-1)).

В листинге 5.6 представлен пример сценария, в котором для добавления элемента в конец массива использован метод push(). Метод вызывается из массива, в который добавляется элемент, а аргументом методу передается значение, которое добавляется в массив. Аргументов у метода push() может быть больше, чем один. В этом случае все они дописываются в конец массива.

Листинг 5.6. Заполнение массива числами Фибоначчи (файл Listing05_06.js)

```
document.write("<h4>Числа Фибоначчи</h4>")
var n=15,a,b
var fibs=new Array()
fibs.push(1,1)
while(fibs.length<n){
  a=fibs[fibs.length-1]
  b=fibs[fibs.length-2]
  fibs.push(a+b)
}
for(var k=0;k<fibs.length;k++){
  document.write(fibs[k]+" | ")
}
```

В представленном сценарии объявляются две дополнительные переменные a и b. Массив fibs создается пустой (инструкция new Array()), а затем командой fibs.push(1,1) в него добавляются две единицы. После этого запускается оператор цикла while, в котором командами a=fibs[fibs.length-1] и b=fibs[fibs.length-2] в переменные a и b записываются соответственно значения последнего и предпоследнего элементов массива fibs. Командой fibs.push(a+b) в массив fibs добавляется новый элемент, значение которого равняется сумме значений переменных a и b. После заполнения массива fibs числами Фибоначчи значения элементов массива отображаются в рабочем документе.

НА ЗАМЕТКУ

При выполнении всех рассмотренных сценариев получаем один и тот же результат (см. рис. 5.4). Для проверки результата достаточно воспользоваться стандартным HTML-шаблоном, заменив в нем название загружаемого файла со сценарием.

Операции с массивами

— Ну, что там происходит?
— Наши играют французскую жизнь...

из к/ф «Покровские ворота»

Некоторые несложные действия (присваивание значений элементам массива, добавление элемента в конец массива или определение длины массива с помощью свойства length) с массивами мы уже выполняли. Здесь рассмотрим более детально некоторые операции, позволяющие лучше понять особенности массивов в JavaScript.

Методы для работы с массивами

Имеется группа методов, позволяющих в удобном режиме выполнять важные операции с массивами. Некоторые методы представлены в табл. 5.1.

ДЕТАЛИ

Методы, о которых далее идет речь, наследуются в массивах из их прототипа Array.prototype. Дело в том, что, как бы ни создавался массив (с помощью конструктора Array или путем явного описания с использованием квадратных скобок), его прототипом является прототип конструктора массивов Array.prototype.

Таблица 5.1. Методы для работы с массивами

Метод	Описание
concat()	Метод для создания массива путем объединения нескольких массивов. Результат получается объединением массива, из которого вызывается метод с массивом или значениями, переданными аргументами методу. Например, если массив a=[1,2,3] и массив b=["x","y","z"], то результатом выражения a.concat(b) будет массив [1,2,3,"x","y","z"]. Массивы a и b при этом не изменяются
every()	Методом возвращается логическое значение true, если все элементы массива удовлетворяют некоторому критерию, и false — в противоположном случае. Характер критерия определяется функцией, которая передается аргументом методу. Как данный метод может быть использован на практике, будет показано позже
filter()	Результатом метода возвращается массив, который получается фильтрацией исходного массива. Алгоритм фильтрации определяется функцией, которая передается аргументом методу. Принципы использования данного метода обсуждаются далее

Метод	Описание
forEach()	Методом для каждого элемента массива (из которого вызывается метод) вызывается функция, переданная аргументом методу. Фактически в данном случае имеет место некое подобие оператора цикла, выполняемого по элементам массива. Как данный метод может быть использован на практике, будет показано позже
indexOf()	Метод предназначен для поиска элемента в массиве. Искомый элемент (значение) передается аргументом методу. Поиск выполняется в массиве, из которого вызывается метод. Результатом метод возвращает индекс первого вхождения элемента в массив. Если элемента в массиве нет, возвращается значение -1. Можно указать второй аргумент. Он определяет индекс в массиве, с которого начинается поиск элемента. Например, если массив a=[1,5,3,7,5,6], то результатом выражения a.indexOf(5) является значение 1 (первое вхождение значения 5 в массив), значением выражения a.indexOf(5,2) является 4 (поиск значения 5 начинается с позиции с индексом 2), а значение выражения a.indexOf(2) равно -1 (в массиве нет элемента со значением 2)
join()	Метод предназначен для формирования на основе значений элементов массива текстовой строки. Элементы объединяются в текстовую строку с использованием разделителя. Разделителем служит текст, переданный аргументом методу. Если у метода аргумента нет, то по умолчанию разделителем является запятая. Например, если a=[1,2,3], то результатом выражения a.join() будет текст "1,2,3". Результатом выражения a.join("*") является текст "1*2*3"
lastIndexOf()	Метод предназначен для поиска элемента в массиве, но только в данном случае (в отличие от метода indexOf()) поиск начинается с конца массива. Аргументом передается значение элемента для поиска в массиве, из которого вызывается метод. Результат метода — индекс последнего вхождения элемента в массив. Если элемента в массиве нет, возвращается значение -1. Второй необязательный аргумент метода определяет индекс в массиве, с которого начинается поиск. Например, если массив a=[1,5,3,7,5,6], то результатом выражения a.lastIndexOf(5) является значение 4 (последнее вхождение значения 5 в массив), значением выражения a.lastIndexOf(5,3) является 1 (поиск значения 5 начинается с позиции с индексом 3 с конца в начало массива), а значение выражения a.lastIndexOf(2) равно -1 (в массиве нет элемента со значением 2)
map()	В результате вызова метода создается новый массив, который получается применением функции, указанной аргументом метода, к каждому элементу исходного массива (из которого вызывается метод). Например, если массив a=[1,9,25], то результатом выражения a.map(Math.sqrt) является массив [1,3,5] (элементы массива получаются вычислением квадратного корня из элементов исходного массива). Массив a при этом не изменяется
pop()	Методом удаляется последний элемент массива. Значение удаляемого элемента массива возвращается результатом метода. Например, если массив a=[1,2,3], то результатом выражения a.pop() является 3, а после выполнения данной команды переменная a будет ссылаться на массив [1,2] (удален последний элемент массива)
push()	Метод предназначен для добавления элементов в конец массива. Метод вызывается из объекта массива, а добавляемые в массив значения передаются аргументами методу. Результатом метода возвращается новая длина массива. Например, если массив a=[1,2,3], то значение выражения a.push("x") равно 4 (количество элементов в массиве после добавления нового элемента), а после выполнения команды переменная a будет ссылаться на массив [1,2,3,"x"]

Метод	Описание
reduce()	Метод используется для сведения массива к единому значению. Алгоритм вычисления такого значения базируется на последовательном переборе элементов массива с выполнением определенных операций, которые задаются функцией, переданной аргументом методу. Элементы массива перебираются в направлении увеличения индекса. Немного позже рассматривается пример использования данного метода
reduceRight()	Метод предназначен для вычисления на основе массива некоторого значения и подразумевает последовательный перебор элементов массива. Действия, выполняемые с элементами массива, определяются функцией, переданной аргументом методу. В отличие от метода reduce(), в данном случае элементы массива перебираются в направлении уменьшения индекса. Далее в книге есть пример использования данного метода
reverse()	Метод предназначен для пересортировки элементов массива в обратном порядке. Например, если массив a=[1,2,3], то после выполнения команды a.reverse() переменная a будет ссылаться на массив [3,2,1]
shift()	Методом удаляется начальный элемент массива. Результатом метода является значение удаляемого элемента массива. Например, если массив a=[1,2,3], то результатом выражения a.shift() является значение 1 (значение первого элемента массива), а после выполнения данной команды переменная a будет ссылаться на массив [2,3] (в исходном массиве удален первый элемент)
slice()	Методом возвращается массив, который получается считыванием группы элементов исходного массива, из которого вызывается метод (возвращается *поверхностная копия* массива). Первый аргумент метода определяет индекс элемента в исходном массиве, с которого начинается считывание. Можно указать и второй аргумент, который определяет индекс первого несчитываемого элемента. Если второй аргумент не указан, считываются все элементы до конца массива. Исходный массив в любом случае не изменяется. Например, если массив a=[10,20,30,40,50], то результатом выражения a.slice(1,3) является массив [20,30] (считываются элементы с индексами от 1 до 2 включительно (числа 20 и 30 соответственно) — элемент 40 с индексом 3 не считывается), а результатом выражения a.slice(2) является массив [30,40,50] (все элементы до конца массива, начиная с элемента 30, имеющего индекс 2)
some()	Методом возвращается логическое значение true, если хотя бы один из элементов массива удовлетворяет некоторому критерию, и false в противоположном случае. Критерий проверки элементов определяется функцией, переданной аргументом методу. Принципы использования метода объясняются несколько позже при рассмотрении одного из примеров
sort()	Метод для сортировки элементов массива. Исходный массив изменяется (сортируется) и возвращается результатом метода. Важно то, что сортировка выполняется на основе текстовых представлений для элементов массива. По умолчанию критерием для сортировки является код символов в кодовой таблице, поэтому результаты сортировки могут быть на первый взгляд неожиданными. Также аргументом методу может передаваться функция, которая определяет пользовательские критерии для сортировки элементов массива. Принципы использования данного метода обсуждаются далее на отдельном примере
splice()	Метод предназначен для удаления части массива и вставки на его место нового фрагмента. Первым аргументом методу передается индекс элемента, начиная с которого выполняется удаление.

Метод	Описание
	Второй аргумент определяет количество удаляемых элементов. Если указаны другие аргументы, то они будут добавлены в массив вместо удаленного фрагмента. Результатом метода возвращается массив из удаленных элементов. Например, если массив a=[10,20,30,40,50], то результатом выражения a.splice(1,3,"x","y") является массив [20,30,40] (удаляемые элементы — всего 3 элемента, начиная с элемента с индексом 1), а переменная a после выполнения данной команды будет ссылаться на массив [10,"x","y",50] (вместо удаленных элементов вставлены элементы "x" и "y")
unshift()	Значения, переданные аргументами методу, добавляются в начало массива. Результатом метода возвращается длина массива (с учетом вставленных элементов). Например, если массив a=[1,2,3], то результатом выражения a.unshift("x","y") является число 5 (размер массива с учетом новых добавленных элементов), а переменная a будет ссылаться на массив ["x","y",1,2,3] (элементы "x" и "y" вставляются в массив в обратном порядке по сравнению с тем, как они переданы аргументами методу)

В программном коде в листинге 5.7 представлена небольшая зарисовка к использованию некоторых (наиболее простых) методов из перечисленных выше.

Листинг 5.7. Методы для работы с массивами (файл Listing05_07.js)

```
var a=[1,2,3]
document.write("Массив <code>a</code>:<br>")
show(a)
var b=["x","y","z"]
document.write("Массив <code>b</code>:<br>")
show(b)
var A=a.concat(b)
document.write("Массив <code>A=a.concat(b)</code>:<br>")
show(A)
A.reverse()
document.write("Массив <code>A</code> после выполнения команды <code>A.reverse()</
code>:<br>")
show(A)
A.shift()
document.write("Массив <code>A</code> после выполнения команды <code>A.shift()</
code>:<br>")
show(A)
A.pop()
```

```
document.write("Массив <code>A</code> после выполнения команды <code>A.pop()</
code>:<br>")
show(A)
A.unshift(10,20,30)
document.write("Массив <code>A</code> после выполнения команды <code>A.
unshift(10,20,30)</code>:<br>")
show(A)
A.push(1,0)
document.write("Массив <code>A</code> после выполнения команды <code>A.push(1,0)</
code>:<br>")
show(A)
document.write("Значение выражения <code>A.slice(2,6)</code>:<br>")
show(A.slice(2,6))
document.write("Массив <code>A</code> после выполнения команды <code>A.slice(2,6)</
code>:<br>")
show(A)
A.splice(3,2,-1,-2,-3)
document.write("Массив <code>A</code> после выполнения команды <code>A.splice(3,2,-1,-
2,-3)</code>:<br>")
show(A)
document.write("Значение выражения <code>A.map(function(x){return 2*x+1})</code>:<br>")
show(A.map(function(x){return 2*x+1}))
document.write("Массив <code>A</code> после выполнения команды <code> A.map(function(x)
{return 2*x+1})</code>:<br>")
show(A)
document.write('Значение выражения <code>eval(A.join("+"))</code>:<br>')
document.write(eval(A.join("+"))+"<br>")
document.write("Значение выражения <code>A.filter(myTest)</code>:<br>")
show(A.filter(myTest))
document.write("Массив <code>A</code> после выполнения команды <code>A.filter(myTest)</
code>:<br>")
show(A)
A.sort()
document.write("Массив <code>A</code> после выполнения команды <code>A.sort()</
code>:<br>")
show(A)
```

```
A.sort(mySort)
document.write("Массив <code>A</code> после выполнения команды <code>A.sort(mySort)</
code>:<br>")
show(A)
// Функция для отображения содержимого массива:
function show(array){
  document.write(array.join(" | ")+"<br>")
}
// Функция для использования в качестве фильтра:
function myTest(x){
  return (x>-3)&&(x<10)
}
// Функция для сравнения элементов при сортировке:
function mySort(a,b){
  if(a<b) return -1
  if(b>1) return 1
  return 0
}
```

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

В сценарии для выделения текста моноширинным шрифтом использованы дескрипторы <code> и </code>. Такой шрифт обычно используют при отображении программных кодов.

Для тестирования сценария используем HTML-код, представленный ниже:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 5.7</title>
</head>
<body><h3>Листинг 5.7</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing05_07.js">
```

```
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Результат выполнения сценария такой, как показано на рис. 5.5.

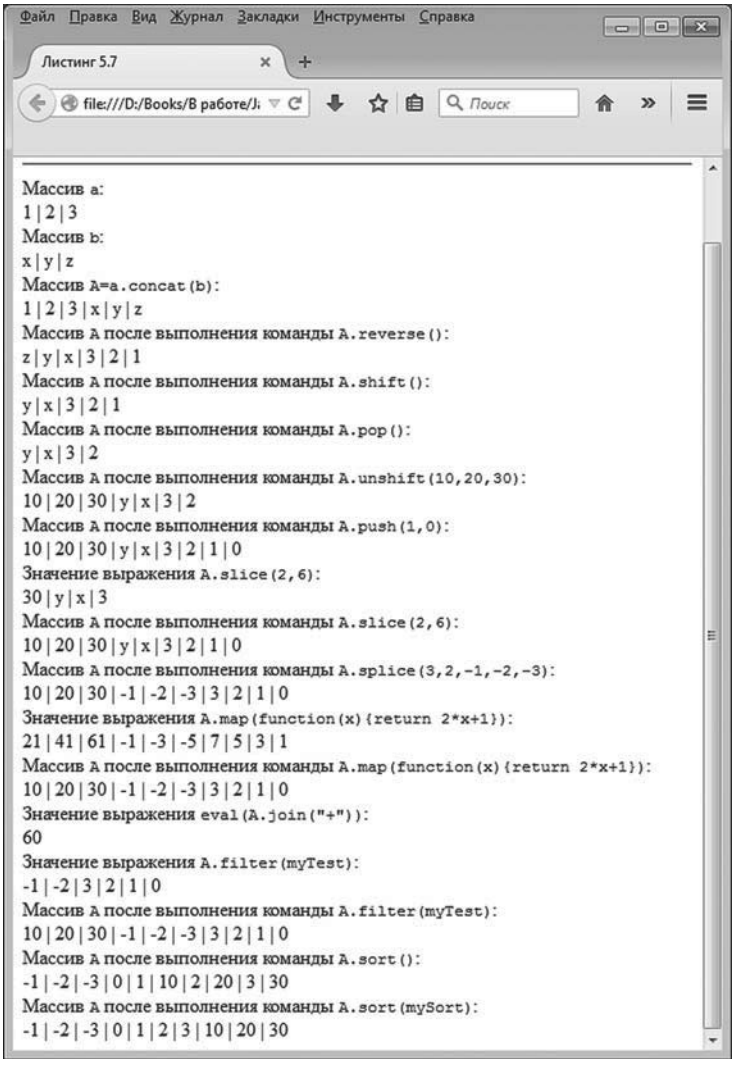

Рис. 5.5. Результат выполнения сценария, в котором используются методы для работы с массивами

Для большего удобства ниже приведена текстовая версия (с поправкой на тип использованного шрифта при выделении команд) сообщений, которые появляются в рабочем документе при выполнении сценария.

Результат выполнения сценария (из листинга 5.7)

Массив a:

1 | 2 | 3

Массив b:

x | y | z

Массив A=a.concat(b):

1 | 2 | 3 | x | y | z

Массив A после выполнения команды A.reverse():

z | y | x | 3 | 2 | 1

Массив A после выполнения команды A.shift():

y | x | 3 | 2 | 1

Массив A после выполнения команды A.pop():

y | x | 3 | 2

Массив A после выполнения команды A.unshift(10,20,30):

10 | 20 | 30 | y | x | 3 | 2

Массив A после выполнения команды A.push(1,0):

10 | 20 | 30 | y | x | 3 | 2 | 1 | 0

Значение выражения A.slice(2,6):

30 | y | x | 3

Массив A после выполнения команды A.slice(2,6):

10 | 20 | 30 | y | x | 3 | 2 | 1 | 0

Массив A после выполнения команды A.splice(3,2,-1,-2,-3):

10 | 20 | 30 | -1 | -2 | -3 | 3 | 2 | 1 | 0

Значение выражения A.map(function(x){return 2*x+1}):

21 | 41 | 61 | -1 | -3 | -5 | 7 | 5 | 3 | 1

Массив A после выполнения команды A.map(function(x){return 2*x+1}):

10 | 20 | 30 | -1 | -2 | -3 | 3 | 2 | 1 | 0

Значение выражения eval(A.join("+")):

60

Значение выражения A.filter(myTest):

-1 | -2 | 3 | 2 | 1 | 0

Массив A после выполнения команды A.filter(myTest):
10 | 20 | 30 | -1 | -2 | -3 | 3 | 2 | 1 | 0
Массив A после выполнения команды A.sort():
-1 | -2 | -3 | 0 | 1 | 10 | 2 | 20 | 3 | 30
Массив A после выполнения команды A.sort(mySort):
-1 | -2 | -3 | 0 | 1 | 2 | 3 | 10 | 20 | 30

Большинство команд просты и должны быть понятны читателю. Мы все же кратко прокомментируем их.

НА ЗАМЕТКУ

Для отображения содержимого массивов мы используем функцию show(). Аргумент функции (предполагается, что это массив) обозначен как array. А в теле функции выполняется всего одна команда document.write(array.join(" | ")+"
"), которой в рабочем документе отображается результат выражения array.join(" | "). Данное выражение, в свою очередь, представляет собой текстовую строку, которая получается объединением значений элементов массива array, а разделителем является текст " | ", переданный аргументом методу join().

Массив A создается с помощью команды A=a.concat(b) объединением массивов a и b. В результате выполнения команды A.reverse() порядок следования элементов в массиве A меняется на противоположный. Выполнение команды A.shift() приводит к удалению первого элемента массива. При выполнении команды A.pop() удаляется последний элемент массива. Командой A.unshift(10,20,30) в начало массива A добавляются элементы со значениями 10, 20 и 30. Командой A.push(1,0) в конец массива A добавляются числа 1 и 0.

Значением выражения A.slice(2,6) является массив из элементов массива A с индексами от 2 по 5 включительно. При этом массив A остается неизменным. А вот при выполнении команды A.splice(3,2,-1,-2,-3) из массива A удаляется два элемента, начиная с элемента с индексом 3, а вместо них вставляются числовые значения -1, -2 и -3.

Некоторого особого пояснения требует выражение A.map(function(x) {return 2*x+1}), в котором метод map() вызывается из массива A, а аргументом методу передается анонимная функция, описанная следующим образом:

```
function(x){
  return 2*x+1
}
```

Это числовая функция с одним аргументом. Результатом она возвращает значение, получающееся умножением значения аргумента на два с последующим прибавлением единицы. Результат выражения A.map(function(x){return 2*x+1}) формируется применением указанной функции к каждому из элементов массива A. В результате создается новый массив (массив A не изменяется), элементы которого получаются применением анонимной функции к соответствующим элементам массива A.

Значением выражения eval(A.join("+")) является сумма элементов массива A. Объяснение такое: результат выражения A.join("+") есть текстовая строка, в которой объединены значения элементов массива A, а разделителем является символ "+". Таким образом, строка содержит текстовое представление для суммы элементов массива. Строка передается аргументом функции eval(), что приводит к вычислению соответствующего выражения.

В команде A.filter(myTest) аргументом методу filter() передается имя функции myTest(). Функцией myTest() возвращается значение (x>-3)&&(x<10), которое равняется true, если аргумент функции x больше -3 и меньше 10. В противном случае функцией возвращается значение false. При вычислении результата выражения A.filter(myTest) формируется новый массив. В него включаются элементы массива A, при тестировании которых функцией myTest() (то есть когда они передаются аргументом функции) возвращается значение true. Поэтому в итоговый массив включены лишь те элементы массива A, значения которых больше -3 и меньше 10. Массив A при этом остается неизменным.

Несколько странным может показаться результат сортировки элементов массива A с помощью метода sort(), если последнему не передавать аргументы. Так, в результате выполнения команды A.sort() массив A упорядочивается довольно странным образом: число 10, например, следует перед числом 2. Здесь следует учесть, что упорядочивание выполняется в «лексикографическом» смысле: сортируются на самом деле не числовые значения, а их текстовые представления. Сравниваются коды в кодовой таблице начальных символов. Поскольку в кодовой таблице код символа "1" меньше кода символа "2", число 10

оказывается перед числом 2. Для получения более осмысленного результата при тестировании используем команду A.sort(mySort). В этой команде аргументом методу sort() передается название функции mySort(). Функция описана с двумя аргументами и возвращает значение -1, если первый аргумент меньше второго, возвращает значение 1, если первый аргумент больше второго, и возвращает значение 0 в прочих случаях (то есть когда аргументы равны). В этом случае сортировка массива базируется на сравнении числовых значений, поэтому результат такой, как надо.

НА ЗАМЕТКУ

После выполнения команды A.sort() массив A изменяется в соответствии с результатом сортировки. Поэтому при выполнении команды A.sort(mySort) на самом деле сортируется «частично отсортированный» массив.

Есть еще несколько методов, предназначенных для работы с массивами, которые представляют определенный практический интерес. Примеры их использования представлены в листинге 5.8.

Листинг 5.8. Некоторые операции с массивами (файл Listing05_08.js)

```
with(document){
  var A=[5,3,2]
  write("A = ["+A+"]<br>")
  write("Результат выполнения команды <code>A.reduce(Math.pow)</code>:<br>")
  write(A.reduce(Math.pow)+"<br>")
  write("Результат выполнения команды <code>A.reduceRight(Math.pow)</code>:<br>")
  write(A.reduceRight(Math.pow)+"<br>")
  var B=[1,2,,4,5,6,7,,9]
  write("B = ["+B+"]<br>")
  write("Результат выполнения команды <code>B.forEach(fillIt)</code>:<br>")
  B.forEach(fillIt)
  write("B = ["+B+"]<br>")
  B.splice(2,1)
  write("После выполнения команды <code>B.splice(2,1)</code>:<br>")
  write("B = ["+B+"]<br>")
  B.splice(6,1)
```

```
  write("После выполнения команды <code>B.splice(6,1)</code>:<br>")
  write("B = ["+B+"]<br>")
  write("Результат выполнения команды <code>B.some(isIt)</code>:<br>")
  write(B.some(isIt)+"<br>")
  write("Результат выполнения команды <code>B.every(isInRange)</code>:<br>")
  write(B.every(isInRange)+"<br>")
  write("Результат выполнения команды <code>A.every(isInRange)</code>:<br>")
  write(A.every(isInRange)+"<br>")
}
// Функция для передачи аргументом в метод forEach():
function fillIt(value,index,array){
  array[index]*=10
  document.write(index+": "+value+"<br>")
}
// Функция для передачи аргументом в метод some():
function isIt(value,index,array){
  return (index<=array.length/2)&&(value<30)
}
// Функция для передачи аргументом в метод every():
function isInRange(value,index,array){
  return (value>0)&&(value<50)
}
```

Для тестирования сценария можем использовать следующий HTML-код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 5.8</title>
</head>
<body><h3>Листинг 5.8</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing05_08.js">
</script>
```

```
<!-- Завершение сценария -->


</body>
</html>
```

При выполнении сценария в рабочем документе отображается последовательность таких сообщений (если не учитывать тип шрифта, используемого для отображения текста).

Результат выполнения сценария (из листинга 5.8)

```
A = [5,3,2]
Результат выполнения команды A.reduce(Math.pow):
15625
Результат выполнения команды A.reduceRight(Math.pow):
32768
B = [1,2,,4,5,6,7,,9]
Результат выполнения команды B.forEach(fillIt):
0: 1
1: 2
3: 4
4: 5
5: 6
6: 7
8: 9
B = [10,20,,40,50,60,70,,90]
После выполнения команды B.splice(2,1):
B = [10,20,40,50,60,70,,90]
После выполнения команды B.splice(6,1):
B = [10,20,40,50,60,70,90]
Результат выполнения команды B.some(isIt):
true
Результат выполнения команды B.every(isInRange):
false
Результат выполнения команды A.every(isInRange):
true
```

Как выглядит результат выполнения сценария в окне браузера, показано на рис. 5.6.

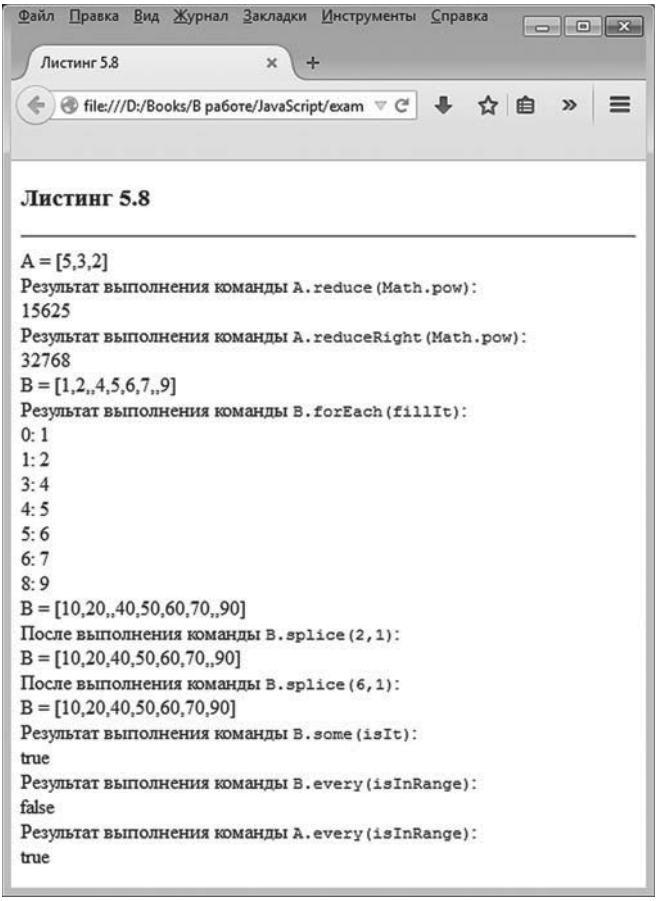

Рис. 5.6. Результат выполнения сценария, в котором использованы некоторые методы для работы с массивами

Теперь дадим пояснения к приведенному выше сценарию.

НА ЗАМЕТКУ

Основная часть кода размещена в with-блоке с указанной в круглых скобках ссылкой на объект document. Поэтому при вызове метода write() в with-блоке объект document можно не указывать. Этим мы немного экономим на объеме программного кода, хотя стоит напомнить, что современные стандарты языка JavaScript не рекомендуют использовать оператор with.

При отображении массива методом write() (то есть когда массив передается аргументом методу) по умолчанию отображаются значения элементов массива. Значения между собой разделяются запятыми.

В первую очередь остановимся на методе reduce(). Метод вызывается из массива, а аргументом ему передается функция двух переменных. Процедура выполнения метода такая: сначала переданная аргументом функция вызывается с аргументами, значения которых совпадают со значениями первых двух элементов массива. Вычисляется значение функции, которое становится первым аргументом функции. Вторым аргументом функции становится третий элемент массива. Вычисляется новое значение функции, и оно снова становится первым аргументом этой же функции. Вторым ее аргументом становится следующий (четвертый) элемент массива. И так далее, пока не будут перебраны все элементы массива. Полученное в итоге значение является результатом вызова метода reduce().

В нашем примере метод reduce() вызывается из массива A, равного [5,3,2]. Аргументом методу передана ссылка Math.pow на метод pow() объекта Math. Метод pow() предназначен для возведения числа в степень: число, переданное первым аргументом методу, возводится в степень, определяемую вторым аргументом метода. Метод reduce() вызывается из массива [5,3,2]. Это означает, что сначала вызывается метод pow() с аргументами 5 и 3. Результатом является число $5^3 = 125$. Затем метод pow() вызывается с аргументами 125 и 2, в результате получаем $125^2 = 15625$. Это и есть результат выражения A.reduce(Math.pow). Метод reduceRight() принципиально отличается от метода reduce() тем, что элементы массива перебираются не слева направо (как в методе reduce()), а справа налево. Так, при вычислении значения выражения A.reduceRight(Math.pow) сначала вызывается метод pow() с аргументами 2 и 3, в результате чего вычисляется значение $2^3 = 8$. Затем метод pow() вызывается с аргументами 8 и 5, в результате чего вычисляется значение $8^5 = 32768$.

ДЕТАЛИ

Метод reduce() достаточно полезный и позволяет быстро решать не самые тривиальные задачи. Например, чтобы вычислить сумму элементов массива, достаточно вызвать из этого массива метод reduce() с анонимной функцией function(x,y){return x+y} (функция для вычисления суммы двух слагаемых) в качестве аргумента. Например, значением выражения [1,2,3,4,5].reduce(function(x,y){return x+y}) является число 15 (сум-

ма натуральных чисел от 1 до 5 — сумма элементов массива). Так же легко вычисляется произведение элементов массива: результатом выражения [1,2,3,4,5].reduce(function(x,y){return x*y}) является число 120 (произведение натуральных чисел от 1 до 5).

Массив B имеет значение [1,2,,4,5,6,7,,9]. Важно то, что в массиве пропущены элементы с индексами 2 и 7. Такие элементы имеют значение undefined. Из данного массива вызывается метод forEach() с аргументом — названием функции fillIt(). Функция fillIt() описана с тремя аргументами value, index и array, которые соответственно обозначают значение элемента массива, индекс элемента массива и сам массив. В теле функции выполняются две команды: командой array[index]*=10 значение элемента массива array с индексом index умножается на 10, после чего командой document.write(index+": "+value+"
") в рабочем документе отображается значение индекса элемента и его значение (и вставляется дескриптор перехода к новой строке). Команда B.forEach(fillIt) выполняется следующим образом: последовательно для каждого из элементов в массиве B вызывается функция fillIt(), и аргументами ей передаются текущее значение элемента массива, индекс элемента массива и ссылка на массив. В результате выполнения команды B.forEach(fillIt) в столбик будут распечатаны старые (до умножения на 10) значения элементов массива. После выполнения команды все элементы будут умножены на 10. При переборе элементов те из них, что имеют значение undefined, игнорируются. Поэтому в столбике со значениями элементов массива отсутствуют позиции для индексов 2 и 7 (соответствующие элементы в исходном массиве пропущены).

НА ЗАМЕТКУ

В аргумент value передается значение соответствующего элемента на момент вызова функции fillIt(). Даже если в процессе вычислений значение элемента меняется, в параметре value остается старое значение элемента. Именно поэтому в приведенном примере в столбик отображаются старые значения элементов массива, а после проверки содержимого массива оказывается, что значения всех числовых элементов умножены на 10.

При выполнении команды B.splice(2,1) из массива B удаляется элемент с индексом 2, а при выполнении команды B.splice(6,1) удаляется элемент с индексом 6.

ДЕТАЛИ

Таким образом, из массива B удаляются пропущенные элементы. В исходном массиве они имеют индексы 2 и 7. Но после удаления элемента с индексом 2 прочие элементы справа сдвигаются на одну позицию влево, поэтому оставшийся пропущенный элемент в новой версии массива будет иметь индекс 6.

Также стоит обратить внимание, что методом splice() элементы удаляются из массива. Если бы мы удаляли элемент с помощью оператора delete (например, командой delete B[1]), то на самом деле элемент из массива бы не удалялся, а удалялось только значение элемента. В результате соответствующий элемент получает значение undefined.

При вычислении выражения B.some(isIt) результатом возвращается значение true, если при вызове функции isIt() для каждого из элементов массива получаем значение true. Функция isIt() описана с тремя аргументами: значение элемента value, индекс элемента index и ссылка на массив array. Функция описана так, что результатом возвращается значение (index<=array.length/2)&&(value<50). Значение данного выражения равно true, если индекс элемента меньше половины длины массива, а значение элемента меньше 30. Очевидно, что такие элементы в массиве B есть (первые два элемента массива), поэтому результатом выражения B.some(isIt) является true.

Похожим образом вычисляется значение выражения B.every(isInRange), только теперь не хотя бы один, а все элементы массива должны удовлетворять определенному условию. В частности, при вызове функции isInRange() для каждого элемента массива должно возвращаться значение true. Функцией isInRange() возвращается значение true, если значение проверяемого элемента массива попадает в диапазон от 0 до 50 (не включая границы). В массиве B есть элементы, большие 50, поэтому результатом выражения B.every(isInRange) является false. А вот результатом выражения A.every(isInRange) является значение true, поскольку все элементы массива A больше 0 и меньше 50.

НА ЗАМЕТКУ

Аргументы у функции isInRange() такие же, как и у функции isIt().

Присваивание и копирование массивов

При присваивании массивов происходит фактически то же самое, что и при присваивании объектов: ссылка с одного массива «перебрасы-

вается» на другой, и в результате две переменные указывают на один и тот же массив. В качестве небольшой иллюстрации рассмотрим пример, представленный в листинге 5.9.

Листинг 5.9. Присваивание массивов (файл Listing05_09.js)

```
var A=[1,2,3,4,5]
document.write("A = ["+A+"]<br>")
var B=["a","b","c"]
document.write("B = ["+B+"]<br>")
B=A
document.write("После присваивания <code>B = A</code>:<br>")
document.write("B = ["+B+"]<br>")
A[0]=100
document.write("После присваивания <code>A[0] = 100</code>:<br>")
document.write("B = ["+B+"]<br>")
```

Для тестирования сценария используем такой HTML-код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 5.9</title>
</head>
<body><h3>Листинг 5.9</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing05_09.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Как будет выглядеть веб-документ с результатом выполнения сценария, показано на рис. 5.7.

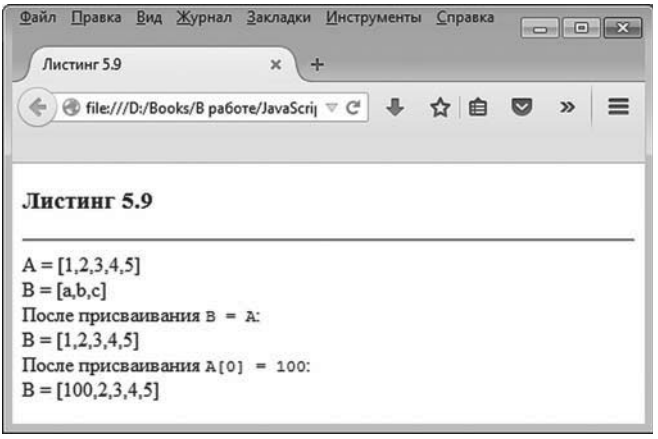

Рис. 5.7. Результат выполнения сценария, в котором имеет место присваивание массивов

Рассмотрим подробнее результат выполнения сценария.

Результат выполнения сценария (из листинга 5.9)

A = [1,2,3,4,5]

B = [a,b,c]

После присваивания B = A:

B = [1,2,3,4,5]

После присваивания A[0] = 100:

B = [100,2,3,4,5]

В сценарии создается два объекта: массив A=[1,2,3,4,5] и массив B=["a","b","c"]. Затем выполняется присваивание B=A. Непосредственная проверка показывает, что значение массива B изменилось и теперь оно такое же, как и у массива A. Но вот если выполнить команду A[0]=100, которой изменяется массив A (точнее, его начальный элемент), то аналогичное изменение произойдет и в массиве B. Причина в том, что переменная массива на самом деле содержит значением ссылку на массив. Технически об этой ссылке можно думать, как если бы это был «адрес» массива. При выполнении команды присваивания B=A значение из переменной A копируется в переменную B. Значением переменной A является «адрес» массива. В результате переменная B получает значением тот же «адрес» и, следовательно, ссылается на тот же самый массив, что и перемен-

ная A. Изменяя массив A, таким образом мы изменяем и массив B. Понятно, что оператор присваивания для копирования массивов не подходит.

Для копирования массива можно использовать метод slice() с нулевым аргументом. Например, чтобы в переменную B записать копию массива A, может быть использована команда B=A.slice(0).

НА ЗАМЕТКУ

Выражением A.slice(0) возвращается массив, получающийся считыванием элементов массива A, начиная с элемента с нулевым индексом (первый аргумент метода slice()). Поскольку второй аргумент метода slice() не указан, то элементы считываются до конца массива.

Если значения элементов массива A относятся к простым типам (данные не ссылочного типа), то такая процедура позволяет создать копию. Но это *поверхностная копия*. Чтобы понять пикантность ситуации, рассмотрим пример, представленный в листинге 5.10.

Листинг 5.10. Создание поверхностной копии массива (файл Listing05_10.js)

```
// Вспомогательная функция для отображения массива:
function show(name,array){
  document.write(name+" = "+array.join(" | ")+"<br>")
}
// Массив с числовыми элементами:
var A=[1,2,3,4,5]
show("A",A)
// Копирование массива:
var B=A.slice(0)
document.write("После присваивания <code>B = A.slice(0)</code>:<br>")
show("B",B)
// Изменение значения элемента массива:
A[0]=100
document.write("После присваивания <code>A[0] = 100</code>:<br>")
show("A",A)
show("B",B)
```

```
document.write("Массив содержит среди элементов другой массив:<br>")
// Массив с элементом - массивом:
var C=[1,[2,3],4,5]
show("C",C)
// Копирование массива:
var D=C.slice(0)
document.write("После присваивания <code>D = C.slice(0)</code>:<br>")
show("D",D)
// Изменение значений элементов массива:
C[1][0]=200
C[3]=500
document.write("После выполнения команд <code>C[1][0] = 200</code> и <code>C[3] = 500</
code>:<br>")
show("C",C)
show("D",D)
```

Сценарий в веб-документ включаем с помощью HTML-кода, приведенного ниже:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 5.10</title>
</head>
<body><h3>Листинг 5.10</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing05_10.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 5.8 показано, как выглядит в окне браузера веб-документ, содержащий данный сценарий.

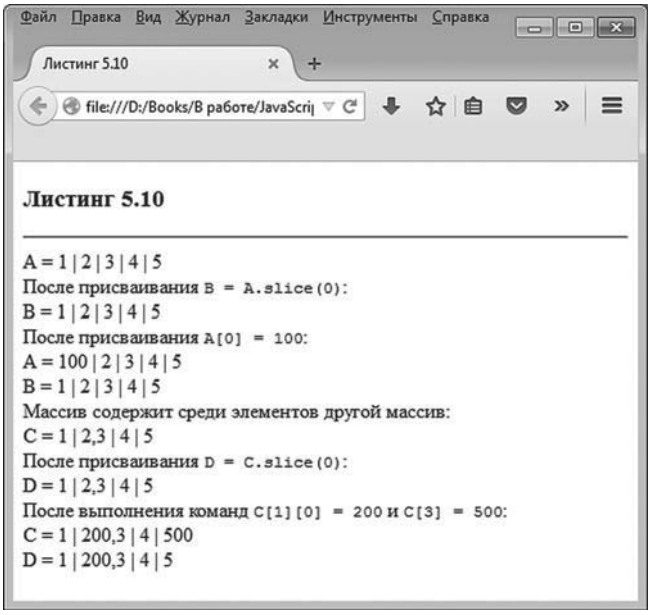

Рис. 5.8. Результат выполнения сценария, в котором создается поверхностная копия массива

Проанализируем результат выполнения сценария (он представлен ниже).

Результат выполнения сценария (из листинга 5.10)

A = 1 | 2 | 3 | 4 | 5

После присваивания B = A.slice(0):

B = 1 | 2 | 3 | 4 | 5

После присваивания A[0] = 100:

A = 100 | 2 | 3 | 4 | 5

B = 1 | 2 | 3 | 4 | 5

Массив содержит среди элементов другой массив:

C = 1 | 2,3 | 4 | 5

После присваивания D = C.slice(0):

D = 1 | 2,3 | 4 | 5

После выполнения команд C[1][0] = 200 и C[3] = 500:

C = 1 | 200,3 | 4 | 500

D = 1 | 200,3 | 4 | 5

В сценарии описывается функция show() с аргументами name и array, предназначенная для отображения содержимого массива. Первый аргумент name определяет дополнительный текст, который фактически определяет название массива. Это название отображается (вместе со знаком равенства) перед содержимым массива. Значения элементов массива, который передается в функцию через аргумент array, отображаются в одну строку с вертикальной чертой в качестве разделителя. В теле функции выполняется всего одна команда, которой в рабочем документе отображается текстовая строка, заданная выражением name+" = "+array.join(" | ")+"
".

ДЕТАЛИ

Текстовая строка получается объединением значения текстового аргумента name, знака равенства, результата инструкции array.join(" | ") и дескриптора перехода к новой строке "
". Результат выражения array.join(" | ") — последовательность значений элементов массива array, разделенных текстом " | ". Если элементы массива array являются числами, то получается текст из последовательности числовых значений, разделенных вертикальной чертой. Если среди элементов массива array есть массив из чисел, то данный массив отображается как элемент внешнего массива в виде последовательности чисел, разделенных запятыми (такой режим отображения массивов используется по умолчанию). Таким образом, если в массиве есть массив с числами, то этот «внутренний» массив отображается в виде блока из разделенных запятыми чисел.

В сценарии создается числовой массив [1,2,3,4,5], ссылка на который записывается в переменную A. Командой B=A.slice(0) создается копия массива A, и ссылка на нее записывается в переменную B.

Проверка показывает, что переменные A и B ссылаются на разные массивы. Действительно, после выполнения команды A[0]=100, которой изменяется значение начального элемента в массиве A, содержимое массива B остается неизменным. Однако ситуация кардинально меняется, если среди элементов массива есть элементы, относящиеся к данным ссылочного типа — например, массивы. В частности, в сценарии переменной C присваивается ссылка на массив [1,[2,3],4,5]. В этом массиве второй по порядку элемент (элемент C[1]) является массивом. Как и в предыдущем случае, копия массива создается с помощью метода slice() командой D=C.slice(0). Проверка содержимого массива D показывает, что у него такие же элементы, как и у массива C.

На следующем этапе командами C[1][0]=200 и C[3]=500 изменяются значения элементов массива C. Командой C[3]=500 изменяется значение числового элемента в массиве C, а командой C[1][0]=200 изменяется значение начального элемента во внутреннем массиве, который является элементом массива C.

НА ЗАМЕТКУ

Выражение C[1] является ссылкой на элемент с индексом 1 в массиве C. В массиве C данный элемент — массив [2,3]. Выражение C[1][0] является ссылкой на элемент с индексом 0 в массиве [2,3].

После проверки содержимого массива D (который, напомним, является копией массива C) оказывается, что изменение значения числового элемента в массиве C никак не сказывается на массиве D, как и должно быть. Но вот изменение значения элемента во внутреннем массиве массива C приводит к изменению соответствующего значения в массиве D. Получается, что в отношении некоторых элементов массив D является копией массива C, а часть элементов у массивов C и D общие.

Чтобы понять происходящее, следует внимательнее проанализировать процесс создания копии массива. Главный принцип, реализуемый при создании копии массива, состоит в том, что создается массив такого же размера, а затем выполняется поэлементное копирование. Если речь идет о числовых массивах, то в результате действительно создается копия массива, физически не зависящая от исходного массива. Но если среди копируемых элементов имеется массив, то его копирование происходит на уровне присваивания ссылок. Здесь удобно представить, что значением элемента-массива на самом деле является «адрес» этого массива (в нашем примере — «адрес» массива [2,3]). При копировании значений элементов копируется «адрес». Поэтому в массиве-копии соответствующий элемент будет содержать тот же «адрес», который содержится в элементе в исходном массиве. В результате и копируемый массив, и массив-копия имеют элемент, ссылающийся на один и тот же внутренний массив. Изменение элемента данного внутреннего массива приводит к тому, что меняется содержимое и массива-копии, и исходного копируемого массива.

Решить проблему можно, если при копировании элементов делать поправку на тип копируемого элемента. Как иллюстрацию рассмотрим пример, в котором создается функция, предназначенная для со-

здания копии массива и «учитывающая» возможность наличия среди элементов массива других массивов.

ДЕТАЛИ

Функция, описанная в сценарии и предназначенная для копирования массивов, называется makeCopy(). В этой функции использован встроенный метод isArray() объекта Array. У метода один аргумент. Результатом метода isArray() возвращается значение true, если аргумент является массивом (переданная аргументом переменная ссылается на массив), и false — в противоположном случае.

Рассмотрим сценарий, представленный в листинге 5.11.

Листинг 5.11. Создание функции для копирования массивов (файл Listing05_11.js)

```
// Функция для отображения содержимого массива:
function show(name,array){
  document.write(name+" = "+array.join(" | ")+"<br>")
}
// Функция для копирования массивов:
function makeCopy(array){
  // Создание массива:
  var tmp=new Array(array.length)
  // Поэлементное копирование значений:
  for(var k=0;k<tmp.length;k++){
    // Если копируемый элемент - массив:
    if(Array.isArray(array[k])){
      // Рекурсивный вызов функции:
      tmp[k]=makeCopy(array[k])
    }
    // Если копируемый элемент - не массив:
    else{
      tmp[k]=array[k]
    }
  }
```

```
  // Результат функции - массив:
  return tmp
}
// Массив с числовыми элементами:
var A=[1,2,3,4,5]
show("A",A)
// Создание копии массива:
var B=makeCopy(A)
document.write("После присваивания <code>B = makeCopy(A)</code>:<br>")
show("B",B)
// Присваивание значения элементу массива:
A[0]=100
document.write("После присваивания <code>A[0] = 100</code>:<br>")
show("A",A)
show("B",B)
document.write("Массив содержит среди элементов другой массив:<br>")
// Массив с элементом - массивом:
var C=[1,[2,3],4,5]
show("C",C)
// Создание копии массива:
var D=makeCopy(C)
document.write("После присваивания <code>D = makeCopy(C)</code>:<br>")
show("D",D)
// Присваивание значений элементам массива:
C[1][0]=200
C[3]=500
document.write("После выполнения команд <code>C[1][0] = 200</code> и <code>C[3] = 500</
code>:<br>")
show("C",C)
show("D",D)
```

Представленный выше сценарий близок к рассмотренному ранее, но, как отмечалось, он содержит пользовательскую функцию makeCopy(). Именно функция makeCopy() используется для создания копии массивов. Ее программный код представляет наибольший интерес.

У функции один аргумент array, который, как предполагается, является копируемым массивом. Результатом функция возвращает копию массива, переданного аргументом.

В теле функции объявляется локальная переменная tmp, которой в качестве значения присваивается ссылка на массив, создаваемый инструкцией new Array(array.length). Аргументом конструктору массивов Array() передается инструкция array.length (размер массива, переданного аргументом функции makeCopy()). Поэтому созданный массив имеет такой же размер, как и массив, переданный аргументом функции makeCopy(). Далее запускается оператор цикла, в котором индексная переменная k пробегает значения от 0 до tmp.length-1, перебирая тем самым все элементы в массивах array и tmp. В теле оператора цикла есть условный оператор, в котором проверяется условие Array.isArray(array[k]). Условие истинно в том случае, если копируемый на данной итерации элемент array[k] является массивом. Если так, то для копирования этого элемента вместо команды обычного присваивания вызывается функция makeCopy() с аргументом array[k]. Результат записывается в элемент tmp[k] (команда tmp[k]=makeCopy(array[k])). Фактически здесь имеет место рекурсивный вызов функции. Для копирования элементов массива вызывается функция makeCopy(), и если оказывается, что копируемый элемент является массивом, то снова вызывается функция makeCopy(), что вполне логично. Однако должен существовать «план» на случай, если копируемый элемент не является массивом. Такой «план» описан в else-блоке условного оператора. Там всего одна команда tmp[k]=array[k], которой выполняется обычное присваивание значений.

По завершении работы оператора цикла массив tmp возвращается результатом функции makeCopy().

В сценарии с использованием функции makeCopy() создается копия для числового массива и для массива, содержащего элементом другой массив. Изменяются некоторые элементы исходного, копируемого массива, и проверяется содержимое массива-копии.

Сценарий тестируем с помощью следующего HTML-кода:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 5.11</title>
</head>
```

```
<body><h3>Листинг 5.11</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing05_11.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Как выглядит результат выполнения сценария, показано на рис. 5.9.

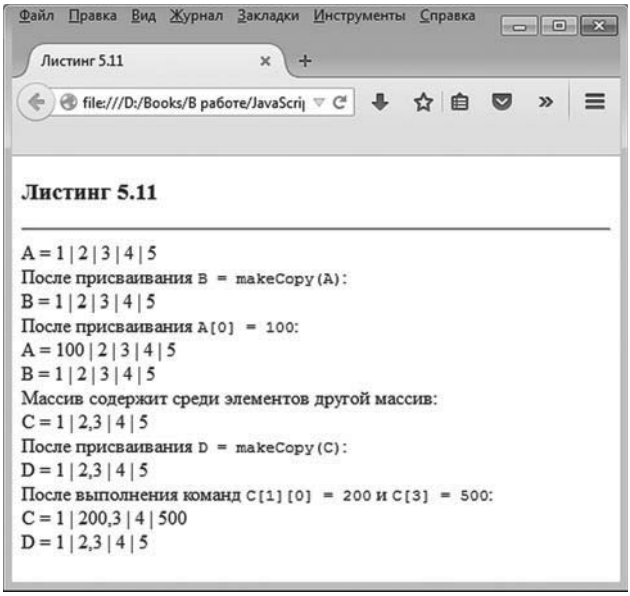

Рис. 5.9. Результат выполнения сценария, в котором описана функция для создания копии массива

Текстовая версия результата выполнения сценария приведена ниже.

Результат выполнения сценария (из листинга 5.11)

```
A = 1 | 2 | 3 | 4 | 5
После присваивания B = makeCopy(A):
B = 1 | 2 | 3 | 4 | 5
```

```
После присваивания A[0] = 100:
A = 100 | 2 | 3 | 4 | 5
B = 1 | 2 | 3 | 4 | 5
Массив содержит среди элементов другой массив:
C = 1 | 2,3 | 4 | 5
После присваивания D = makeCopy(C):
D = 1 | 2,3 | 4 | 5
После выполнения команд C[1][0] = 200 и C[3] = 500:
C = 1 | 200,3 | 4 | 500
D = 1 | 2,3 | 4 | 5
```

Как видим, в данном случае при копировании массивов, даже включая внутренние элементы-массивы, реально создается копия. Изменение значений в исходном копируемом массиве никак не сказывается на массиве-копии.

Понятно, что описанная выше функция для создания копии массива не лишена недостатков. Например, при копировании элементов-объектов столкнемся с такой же проблемой, как при копировании элементов-массивов. Но выше был продемонстрирован идеологический подход, который может быть использован при обработке более сложных ситуаций на более эвристическом уровне.

Методы toString() и valueOf()

Мы уже знаем, что добавление метода или свойства в прототип объекта приводит к тому, что такое свойство или метод появляется у всех объектов, имеющих данный прототип. Массивы в JavaScript являются объектами. Прототип массивов, которые создаются явным описанием (перечислением значений элементов массива в квадратных скобках) или с помощью конструктора массивов Array, имеют прототип Array.prototype. Поэтому если в данный прототип добавить «нечто», то это «нечто» появится во всех массивах.

> **НА ЗАМЕТКУ**
>
> Напомним, что при обращении к свойству (для считывания значения) или методу объекта, если у объекта такого собственного свойства или метода нет, поиск выполняется в прототипе объекта.

Если в прототипе объекта свойства или метода не обнаружено, поиск продолжается в прототипе прототипа и так далее.

В данном конкретном случае нас интересуют методы. Причем даже не те методы, что можно добавить в прототип Array.prototype, а те, которые там уже имеются. Более конкретно, в прототипе Array.prototype есть методы toString() и valueOf(), которые «отвечают» за автоматическое преобразование массива соответственно к текстовому формату и к примитивному типу. Методы вызываются случаях, когда по контексту команды в соответствующем выражении должно быть значение базового типа или текст. Вызываются методы неявно: другими словами, они вызываются обычно без непосредственного участия программиста (хотя методы можно вызывать и явно).

ДЕТАЛИ

Методы toString() и valueOf() наследуются из прототипа верхнего уровня Object.prototype. Поэтому данные методы есть у всех объектов, которые прямо или опосредованно наследуют прототип Object.prototype. К этим объектам относятся и массивы. Именно благодаря наличию методов toString() и valueOf() у нас была возможность передавать, например, массив аргументом методу write(). Напомним, что в таком случае по умолчанию отображаются элементы массива, разделенные запятой. Технически это происходит путем автоматического преобразования массива к текстовому формату. Если нас не устраивает результат такого преобразования, можем переопределить метод toString() и/или valueOf() для объекта или его прототипа. Переопределение методов toString() и valueOf() для обычных объектов (не массивов) обсуждается в следующей главе.

Причины и ситуации, при которых вызываются методы toString() и valueOf(), в общем и целом могут быть сформулированы следующим образом.

- Метод valueOf() вызывается в тех случаях, когда объект (массив) является частью выражения (операндом выражения) и в соответствующем месте в выражении ожидается значение примитивного (базового) типа.
- Метод toString() вызывается в тех случаях, когда объект (массив) используется в выражении в том месте, где должно быть текстовое значение.

НА ЗАМЕТКУ

Вообще алгоритм применения методов toString() и valueOf() — не самый тривиальный. Мы будем обсуждать его постепенно, по мере рассмотрения программного кода.

В листинге 5.12 приведен небольшой пример, в котором в прототипе Array.prototype переопределяются методы toString() и valueOf(), в результате чего массивы можно использовать в различных выражениях, которые сами по себе не подразумевают использование массивов.

Листинг 5.12. Создание функции для копирования массивов (файл Listing05_12.js)

```
// Переопределение в прототипе метода toString():
Array.prototype.toString=function(){
  var txt="< "+this.join(" ; ")+" >"
  return txt
}
// Переопределение в прототипе метода valueOf():
Array.prototype.valueOf=function(){
  return eval(this.join("+"))/this.length
}
// Массив:
var A=[1,[2,3],4,"текст",true]
document.write("Массив:<br>")
// Отображение массива (вызывается метод toString()):
document.write(A)
// Массив:
var B=[3,5,1,8,2]
// Отображение массива (вызывается метод toString()):
document.write(["<br>B = ",B].join(""))
// Вызывается метод valueOf():
document.write("<br>Среднее значение: "+B+"<br>")
// Явный вызов методов toString() и valueOf():
document.write("Массив "+[1,2,3,4].toString()+" - среднее "+[1,2,3,4].valueOf())
```

В сценарии для прототипа Array.prototype переопределяется метод toString(): свойству Array.prototype.toString значением присваивается функция без аргументов, которая в качестве результата возвращает текстовое значение "< "+this.join(" ; ")+" >" локальной переменной txt.

ДЕТАЛИ

Значением выражения "< "+this.join(" ; ")+" >" является текстовая строка, которая получается объединением открывающих угловых скобок "< ", строки this.join(" ; ") и закрывающих угловых скобок " >". Строка this.join(" ; ") представляет собой результат вызова из массива (ссылка this означает объект, из которого вызывается метод toString(), — в данном случае таким объектом является массив) метода join(). Получается текстовая строка, которая состоит из значений элементов массива, разделенных текстом " ; ". Данный текст передается аргументом методу join().

Аналогичным образом переопределяется метод valueOf(), но только теперь ссылке Array.prototype.valueOf присваивается функция без аргументов, возвращающая результатом числовое (как предполагается) значение eval(this.join("+"))/this.length.

ДЕТАЛИ

Для числового массива выражением eval(this.join("+"))/this.length вычисляется среднее значение (сумма элементов массива, деленная на количество элементов в массиве). Значение выражения this.join("+") является текстовой строкой, состоящей из элементов массива, объединенных знаком "+". Текстовая строка передается аргументом функции eval(), в результате чего вычисляется сумма элементов массива. Сумма делится на число this.length, равное количеству элементов в массиве.

После описания методов toString() и valueOf() в сценарии командой var A=[1,[2,3],4,"текст",true] создается массив A. Отобразить содержимое массива можем с помощью команды document.write(A). В этом случае для преобразования массива к текстовому формату автоматически вызывается метод toString().

Еще один массив создается командой var B=[3,5,1,8,2]. Один из способов отображения массива иллюстрирует команда document.write(["
B = ",B].join("")). В ней выводимая в документ текстовая строка формируется выражением ["
B = ",B].join(""). Результатом выражения является текст, который получается объединением элементов массива ["
B = ",B].

Разделителем служит пустая текстовая строка (аргумент метода join()). В массиве ["
B = ",B] два элемента: текст "
B = " и массив B. При объединении этих элементов в общую текстовую строку для преобразования массива B к текстовому формату вызывается метод toString(). А вот при выполнении команды document.write("
Среднее значение: "+B+"
") для приведения массива B к базовому формату (в данном случае числовому) вызывается метод valueOf().

ДЕТАЛИ

При вычислении выражения "
Среднее значение: "+B+"
", несмотря на то что два из трех операндов текстовые, для массива B (второй операнд) выполняется попытка приведения не непосредственно к текстовому формату, а к приемлемому базовому типу. Для выполнения такого преобразования вызывается метод valueOf().

Но если бы в сценарии отсутствовало описание метода valueOf(), то при вычислении выражения "
Среднее значение: "+B+"
" вызывался бы метод toString(). Причина в том, что если метод valueOf() явно не описан, то используется та его версия, что наследуется через прототип. Она вызывается, но если такая версия не возвращает значение простого типа (а она не возвращает), то вызывается метод toString().

Наконец, команда document.write("Массив "+[1,2,3,4].toString()+" - среднее "+[1,2,3,4].valueOf()) дает пример явного вызова методов toString() и valueOf() из массива. Последний, кстати, представлен в виде литерала [1,2,3,4].

Для тестирования сценария используем следующий HTML-код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 5.12</title>
</head>
<body><h3>Листинг 5.12</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing05_12.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 5.10 представлен результат выполнения представленного выше сценария.

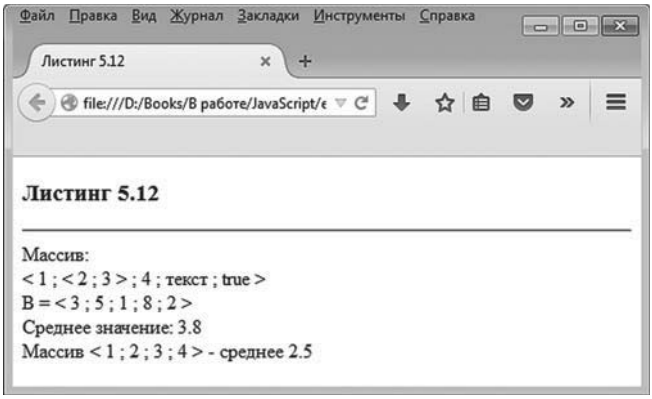

Рис. 5.10. Результат выполнения сценария, в котором вызываются (явно и неявно) методы *toString()* и *valueOf()*

Текстовая версия результата выполнения сценария представлена ниже.

Результат выполнения сценария (из листинга 5.12)

```
Массив:
< 1 ; < 2 ; 3 > ; 4 ; текст ; true >
B = < 3 ; 5 ; 1 ; 8 ; 2 >
Среднее значение: 3.8
Массив < 1 ; 2 ; 3 ; 4 > - среднее 2.5
```

К вопросу об определении методов toString() и valueOf() (в контексте работы с объектами) мы вернемся в следующей главе.

Двумерные массивы

> Нет, он сомнителен! Он сомнителен!!! Я бы ему не доверял.
>
> *из к/ф «Покровские ворота»*

Многомерный массив — это массив, в котором для идентификации элемента нужно несколько индексов (больше, чем один). На практике обычно редко встречаются массивы размерности большей, чем два. Особенности двумерных массивов рассматриваются далее.

С формальной точки зрения, *двумерный массив* — это массив, элементами которого являются одномерные массивы. Процесс создания такого массива качественно не отличается от создания одномерного массива. Проиллюстрируем это на примере. Рассмотрим программный код, представленный в листинге 5.13.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

В сценарии для отображения нижних индексов использованы дескрипторы _и.

Листинг 5.13. Создание двумерного массива (файл Listing05_13.js)

```
// Добавление метода toString() в прототип массива:
Array.prototype.toString=function(){
  return this.join(" ")+"<br>"
}
// Двумерный числовой массив:
var A=[[1,2,3],[4,5,6],[7,8,9]]
document.write("<b>Массив A</b>:<br>")
// Отображение двумерного числового массива:
document.write(A)
var i,j,m=3,n=4
// Создание одномерного массива:
var B=new Array(m)
// Заполнение массива:
for(i=0;i<B.length;i++){
  // Элемент массива - одномерный массив:
  B[i]=new Array(n)
  // Заполнение внутреннего массива:
  for(j=0;j<B[i].length;j++){
    // Текстовое значение элемента двумерного массива:
    B[i][j]="b<sub>"+(i+1)+(j+1)+"</sub>"
  }
}
document.write("<b>Массив B</b>:<br>")
// Отображение двумерного текстового массива:
document.write(B)
```

Сценарий начинается с добавления в прототип Array.prototype метода toString(). Метод описан так, что результатом возвращается текстовая строка, получающаяся объединением через пробел значений элементов массива. В конце текста добавляется дескриптор перехода к новой строке. Данный метод мы рассматриваем исключительно как вспомогательное средство, призванное облегчить процедуру вывода содержимого двумерного массива в рабочее окно.

ДЕТАЛИ

Метод toString() концептуально описан так же, как для одномерного массива. Если массив на самом деле одномерный, то результат формируется объединением значений элементов (разделитель — пробел) и добавлением в конце текста дескриптора перехода к новой строке. Если массив двумерный, то это означает, что в одномерном массиве элементы являются одномерными массивами. Принцип формирования результата для метода toString() остается тем же: элементы внешнего массива объединяются в текстовую строку, в конце которой добавляется дескриптор
. Но теперь, когда в текстовую строку объединяются элементы-массивы, для приведения каждого такого элемента-массива к текстовому формату опять вызывается метод toString(). Метод объединяет в текст элементы уже внутреннего массива. Каждая полученная последовательность заканчивается дескриптором перехода к новой строке. В результате формируется текстовая строка, являющаяся объединением тестовых строк. Разделителем между внутренними текстовыми строками является дескриптор
, причем в конце всей большой текстовой строки таких дескрипторов два (один от последней внутренней строки, и один добавляется при первом вызове метода toString()). Как следствие, при отображении двумерного массива элементы массива выводятся в документ построчно.

В сценарии командой var A=[[1,2,3],[4,5,6],[7,8,9]] создается двумерный числовой массив, который отображается командой document.write(A).

Сценарий содержит еще одну иллюстрацию к созданию двумерного массива. Для удобства в сценарии объявляются две индексные переменные i и j, а также переменная m со значением 3 и переменная n со значением 4. Они определяют соответственно количество строк и столбцов в двумерном массиве. Непосредственно создание массива начинается с выполнения команды var B=new Array(m). В данном случае создается одномерный массив из m элементов. Это «внешний» массив. Этот массив теперь следует заполнить элементами, которые сами

являются массивами. Для этого запускается оператор цикла, в котором индексная переменная i пробегает значения от 0 до B.length-1. В теле оператора цикла командой B[i]=new Array(n) создается одномерный массив из n элементов, и ссылка на него записывается в элемент B[i].

НА ЗАМЕТКУ

Таким образом, двумерный массив представляет собой одномерный массив, элементы которого являются ссылками на одномерные массивы.

Для заполнения «внутреннего» массива (массива-элемента) запускается еще один оператор цикла, в котором индексная переменная j пробегает значения от 0 до B[i].length-1. Здесь при определении верхней границы диапазона изменения индексной переменной мы воспользовались тем, что элемент B[i] является массивом и у него есть свойство length, возвращающее количество элементов в массиве (фактически количество элементов в строке с индексом i в двумерном массиве). Значение элементу присваивается командой B[i][j]="b_{"+(i+1)+(j+1)+"}".

ДЕТАЛИ

При обращении к элементу двумерного массива B указываются два индекса. Выражение B[i][j] следует понимать так: во внешнем массиве следует взять элемент с индексом i, а в этом элементе (который также является массивом) нужно взять элемент с индексом j.

Значение выражения "b_{"+(i+1)+(j+1)+"}" вычисляется так: к тексту "b_{" последовательно дописываются числовые значения (i+1) и (j+1), а затем еще и текст "}". Таким образом, числовые значения, определяющие номер строки и номер столбца в двумерном массиве, оказываются между дескрипторами _и, поэтому отображаются как нижние индексы.

Содержимое массива B проверяем с помощью команды document.write(B). Для проверки работы всего сценария используем следующий HTML-код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 5.13</title>
```

```
</head>
<body><h3>Листинг 5.13</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing05_13.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 5.11 показано, как выглядит результат выполнения описанного выше сценария, в котором создаются два двумерных массива (числовой и текстовый).

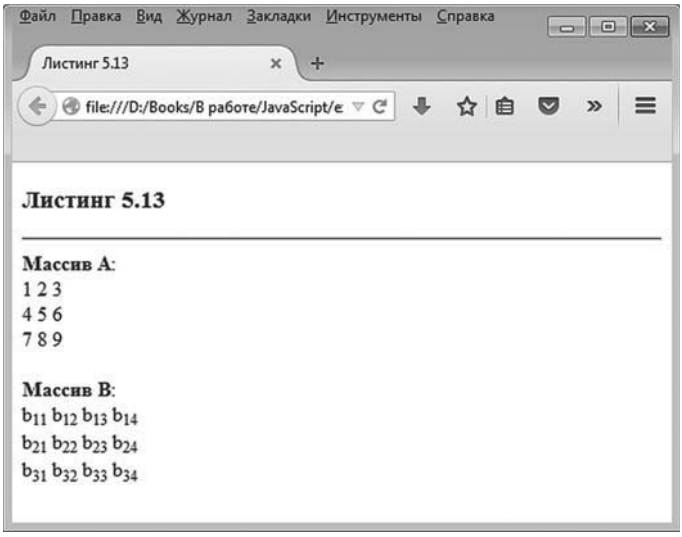

Рис. 5.13. Результат выполнения сценария, в котором создается двумерный массив

НА ЗАМЕТКУ

Анализируя приведенный пример, легко сделать вывод, что в двумерном массиве, если рассматривать его как матрицу из строк и столбцов с элементами, количество элементов в каждой строке может быть разным. Такие массивы иногда называют «рваными».

Резюме

Ну, царь, вздрогнули!

из к/ф «Иван Васильевич меняет профессию»

В этой главе состоялось краткое знакомство с массивами. Оно будет продолжено в следующих главах книги. Сейчас же стоит выделить основные моменты, которые обсуждались выше.

- Одномерный массив можно создать с помощью конструктора массивов Array или явным указанием значений элементов массива в квадратных скобках.
- Количество элементов в массиве определяют с помощью свойства length.
- Добавить элемент в массив можно путем присваивания значения этому элементу массива.
- У объекта Array есть достаточно много методов, предназначенных для работы с массивами, которые позволяют добавлять и удалять элементы, сортировать и фильтровать массив, тестировать элементы массива, выполнять преобразования и некоторые иные операции с массивами.
- Поскольку переменная массива *ссылается* на массив, то присваивание массивов приводит к тому, что две переменные указывают на один и тот же массив. Поверхностную копию массива можно создать с помощью метода slice().
- Переопределяя методы toString() и valueOf(), для массива или его прототипа задают способ автоматического преобразования массивов к текстовому формату и базовым (примитивным) типам.
- Двумерный массив представляет собой массив, элементами которого являются одномерные массивы. При обращении к элементу двумерного массива указывается два индекса — каждый в отдельных квадратных скобках.

Глава 6
ИСПОЛЬЗОВАНИЕ ОБЪЕКТОВ

> Работа секретная. Думаю, с космосом связанная. Так что читайте газеты!
>
> *из к/ф «Усатый нянь»*

В этой главе мы рассмотрим темы, которым не уделили в силу разных причин должного внимания в предыдущих главах и которые в той или иной мере имеют отношение к объектам и принципам ООП. Мы снова поговорим о функциях, массивах, обсудим вопросы, связанные с созданием объектов (в частности, проанализируем методы, предназначенные для работы с объектами), рассмотрим встроенные объекты языка JavaScript. Также далее речь пойдет о таком мощном механизме, как обработка исключительных ситуаций. Будет и повторение материала, рассмотренного ранее: на некоторые уже знакомые нам проблемы и задачи мы посмотрим под несколько иным углом зрения.

Обработка исключительных ситуаций

> Были демоны — мы этого не отрицаем. Но они самоликвидировались. Так что прошу эту глупую панику прекратить!
>
> *из к/ф «Иван Васильевич меняет профессию»*

Ранее мы скромно обходили вопрос о том, как поступать, если в программном коде возникают ошибки. Здесь имеются в виду те ошибки, которые возникают в процессе выполнения сценария. С одной стороны, ситуация вроде бы очевидная: если возникает ошибка, ее необходимо исправить. Если ошибка связана с некорректным синтаксисом кода и сценарий не выполняется, то в таком случае хотя бы понятно, что ошибка есть. Но ошибка может быть более коварной, когда формально (с точки зрения корректности синтаксиса языка JavaScript) код правильный, но при этом выполняются действия, приводящие

к ошибке. Такие ошибки отследить или предугадать достаточно сложно. Но их можно *обработать*.

Инструкция try-catch

Для обработки ошибок (или *исключительных ситуаций*, как их обычно называют) в JavaScript используют инструкцию try-catch. Синтаксис ее вызова такой (жирным шрифтом выделены ключевые элементы кода):

```
try{
  // контролируемый код
}
catch(ошибка){
  // код для обработки ошибки
}
```

После ключевого слова try в фигурных скобках указывается *контролируемый программный код*. Это программный код, при выполнении которого может возникнуть ошибка (генерируется исключительная ситуация). После try-блока указывается catch-блок. Это блок обработки исключительной ситуации. Принцип выполнения инструкции try-catch следующий. Выполняется программный код в try-блоке. Если при выполнении этого кода ошибки не возникают, то программный код в catch-блоке игнорируется. Если же при выполнении try-блока возникла ошибка, то выполнение кода в try-блоке прекращается и начинает выполняться программный код в catch-блоке. После выполнения кода catch-блока управление передается следующей команде после инструкции try-catch.

Это, так сказать, концептуальная схема выполнения инструкции try-catch. Технически все выглядит немного сложнее. Дело в том, что при возникновении ошибки автоматически создается объект, который «описывает» возникшую исключительную ситуацию (свойства объекта содержат информацию о характере возникшей ошибки). Объект ошибки передается в catch-блок. Поэтому после ключевого слова catch в круглых скобках указывается переменная, в которую записывается ссылка на объект ошибки. Данную переменную можно в случае необходимости использовать в catch-блоке для получения дополнительной информации об ошибке.

НА ЗАМЕТКУ

Переменную в catch-блоке объявлять не нужно. Внешне ситуация такая, как если бы catch-блок имел аргумент наподобие функции.

Небольшой пример использования try-catch-инструкции для обработки исключительной ситуации представлен в листинге 6.1.

Листинг 6.1. Обработка исключительной ситуации (файл Listing06_01.js)

```
// Контролируемый код:
try{
  document.write("Начало выполнения <code>try</code>-блока<br>")
  var txt
  // Отображение окна с полем ввода для считывания текста:
  txt=prompt("Введите выражение для вычисления:")
  // Попытка вычислить выражение, считанное в переменную:
  document.write("Результат вычисления выражения: "+eval(txt)+"<br>")
}
// Обработка исключительной ситуации:
catch(e){
  document.write("<b>Произошла ошибка</b><br>")
}
document.write("Выполнение сценария завершено")
```

Для проверки функциональности программного кода используем такой HTML-код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 6.1</title>
</head>
<body><h3>Листинг 6.1</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing06_01.js">
```

```
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Код сценария начинается с try-блока, в котором командой document.write("Начало выполнения <code>try</code>-блока
") отображается сообщение о начале выполнения контролируемого блока. Затем объявляется переменная txt, и командой txt=prompt("Введите выражение для вычисления:") отображается окно с текстовым полем ввода, в которое следует ввести некоторое выражение и которое записывается в переменную txt. Следующей командой document.write("Результат вычисления выражения: "+eval(txt)+"
") выполняется попытка вычислить значение введенного выражения (выражения, записанного в переменную txt) и отобразить результат в рабочем документе.

На этом этапе может возникнуть ошибка. Если так, то будет выполнена команда document.write("Произошла ошибка
") в catch-блоке. Если ошибка не возникла, то catch-блок игнорируется.

На рис. 6.1 показано окно с полем ввода, которое отображается при отображении веб-документа со сценарием.

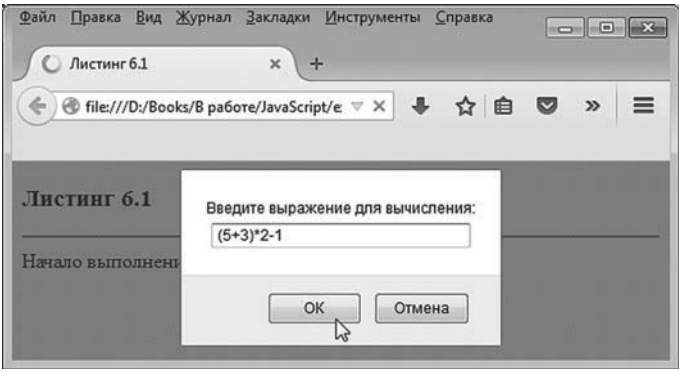

Рис. 6.1. В текстовое поле вводится выражение для вычисления в сценарии

В данном случае в текстовое поле вводится выражение (5+3)*2-1, которое и записывается в переменную txt. После щелчка по кнопке **ОК** получаем результат, как на рис. 6.2.

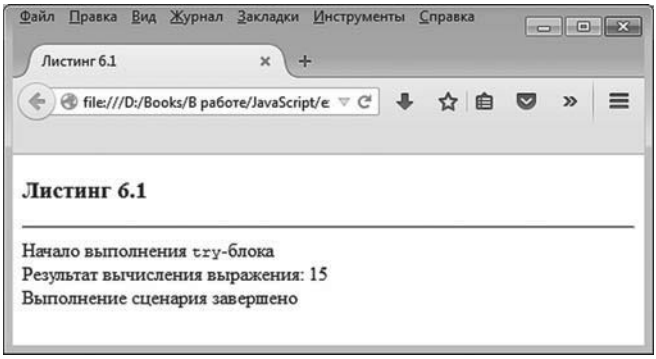

Рис. 6.2. Результат выполнения сценария, если ошибки нет

В этом случае ошибка не возникает, выражение вычисляется корректно, в рабочем документе отображается значение выражения, и catch-блок в «игру» не вступает.

На рис. 6.3 проиллюстрирована ситуация, когда в текстовое поле вводится некорректная команда A+B (переменные A и B в сценарии не объявлены).

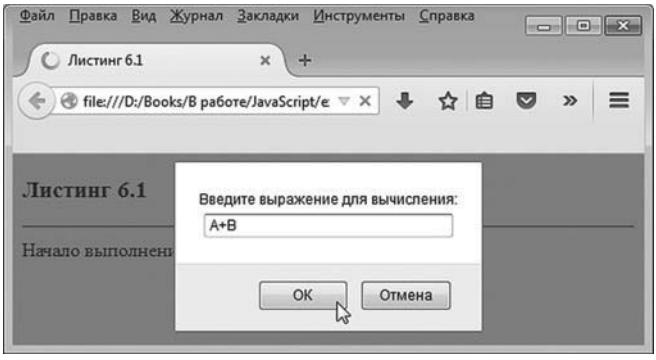

Рис. 6.3. Ввод в текстовое поле некорректного выражения для вычисления

НА ЗАМЕТКУ

Чтобы увидеть окно с полем ввода, в браузере достаточно щелкнуть по кнопке обновления (перезагрузки) веб-документа.

В таком случае попытка вычислить выражение A+B приведет к ошибке, и результат будет таким, как показано на рис. 6.4.

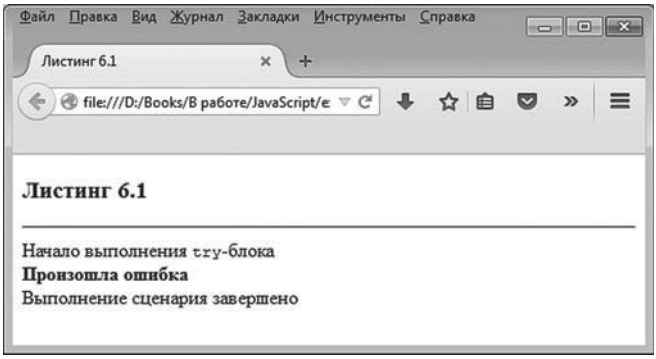

Рис. 6.4. Результат выполнения сценария, если возникла ошибка

Значение выражения уже не отображается, зато отображается сообщение, выведенное при выполнении catch-блока.

НА ЗАМЕТКУ

В сценарии catch-блок описан с переменной e, обозначающей объект исключения (объект, содержащий описание ошибки). Тем не менее данный объект в catch-блоке нами не используется.

Объект ошибки

Как отмечалось выше, через объект ошибки можно получить доступ к некоторым важным ее характеристикам, таким, скажем, как тип ошибки или краткое описание сути произошедшего «недоразумения». Все это богатство реализуется в основном через свойства объекта ошибки. Так, название (тип) ошибки определяют с помощью свойства name. Описание ошибки можно получить с помощью свойства message.

ДЕТАЛИ

У объектов ошибки также существуют браузер-специфичные свойства. Что имеется в виду? Имеется в виду то, что при работе с браузером определенного типа, помимо стандартных свойств объекта ошибки, могут быть еще и дополнительные. Как пример приведем свойство fileName, поддерживаемое браузером Mozilla Firefox и позволяющее определить, в каком файле произошла ошибка. Далее приводится пример того, как такое браузер-специфичное свойство может быть использовано на практике.

Пример использования перечисленных выше свойств объекта ошибки для классификации типа ошибки приведен в листинге 6.2. Данный пример является вариацией примера, рассмотренного нами ранее, поэтому некоторые несущественные комментарии удалены. Но идея осталась неизменной: при выполнении сценария отображается окно с полем ввода, в которое следует ввести команду для выполнения. Если при этом возникает ошибка, то она перехватывается. Только теперь информация по ошибке дается более подробно, чем это было в предыдущем примере.

Листинг 6.2. Использование объекта ошибки при обработке исключительной ситуации (файл Listing06_02.js)

```
try{
  document.write("Начало выполнения <code>try</code>-блока<br>")
  var txt=prompt("Введите выражение для вычисления:")
  document.write("Результат вычисления выражения: "+eval(txt)+"<br>")
  document.write("Выполнение <code>try</code>-блока завершено<br>")
}
catch(e){
  // Название ошибки
  var name=e.name
  // Описание ошибки:
  var message=e.message
  // Текст для отображения:
  var str
  // Проверка типа ошибки:
  switch(name){
    case "ReferenceError":
      str="Некорректная ссылка"
      break
    case "SyntaxError":
      str="Синтаксическая ошибка"
      break
    default:
      str="Ошибка "+name
  }
```

```
  // Отображение характеристик ошибки:
  document.write("<b>Внимание! Произошла ошибка!</b><br>")
  document.write("<b>Тип ошибки: </b>"+str+"<br>")
  document.write("<b>Описание ошибки: </b>"+message+"<br>")
  // Для Mozilla Firefox:
  if("fileName" in e){
    document.write("<b>Файл с ошибкой: </b>"+e.fileName+"<br>")
  }
}
document.write("Выполнение сценария завершено")
```

Ниже представлен HTML-код для тестирования сценария:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 6.2</title>
</head>
<body><h3>Листинг 6.2</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing06_02.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

В сценарии в try-блоке в переменную txt записывается текст, который пользователь вводит в окне с текстовым полем. Диалоговое окно с полем ввода на фоне рабочего документа показано на рис. 6.5.

Если пользователь вводит корректное выражение, то его значение вычисляется и отображается в рабочем документе.

На рис. 6.6 показано, как будет выглядеть рабочий документ, если в текстовое поле ввести выражение 1+2.

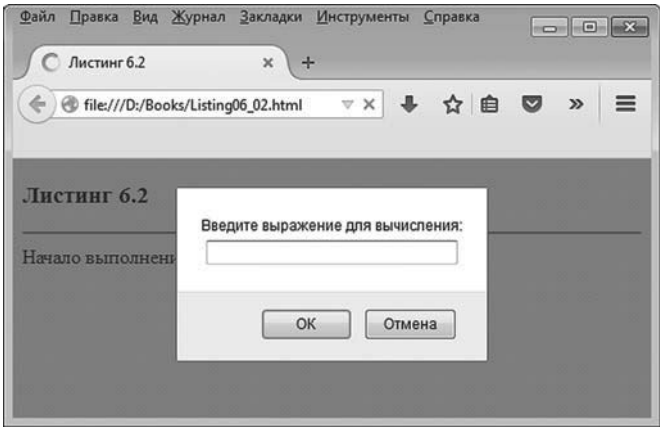

Рис. 6.5. При запуске сценария отображается окно с полем ввода

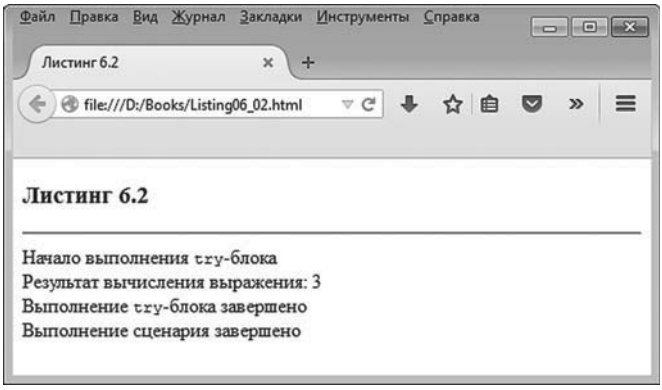

Рис. 6.6. Результат выполнения сценария в случае, когда в текстовое поле вводится выражение *1+2*

Если же пользователь вводит некорректное значение в текстовое поле, возникает ошибка, которая обрабатывается в catch-блоке. В предыдущем примере в catch-блоке была всего одна команда. Здесь, в рассматриваемом примере, команд в catch-блоке достаточно много. Объект ошибки обозначен как e. В теле catch-блока определяются такие характеристики ошибки, как ее тип (название), описание ошибки, а для браузера Mozilla Firefox еще и полный путь к файлу, в котором возникла ошибка:

- в переменную name записывается значение e.name (тип ошибки);
- в переменную message записывается значение e.message (описание ошибки).

Затем запускается оператор выбора switch, в котором тестируется значение переменной name (то есть фактически значение e.name — тип ошибки). В case-блоках проверяется два возможных значения для типа ошибки: "ReferenceError" (ошибка, связанная с неправильной ссылкой в команде) и "SyntaxError" (ошибка, связанная с некорректным синтаксисом команды). Также в операторе выбора имеется default-блок, команды которого выполняются, если при проверке в case-блоке совпадение не найдено. Что касается непосредственно назначения оператора выбора, то в нем, в зависимости от типа исключительной ситуации, формируется значение переменной str, которая затем используется при отображении основных характеристик ошибки.

ДЕТАЛИ

Данные о файле со сценарием, в котором произошла ошибка, отображаются только в том случае, если у объекта ошибки есть свойство fileName. Мы проверяем наличие у объекта ошибки e свойства fileName с помощью условного оператора, в котором условием указано выражение "fileName" in e. Если это так, то значение e.fileName отображается в рабочем документе.

На рис. 6.7 показано, как будет выглядеть рабочий документ, если пользователь в текстовое поле вводит выражение + (в таком случае возникает синтаксическая ошибка типа SyntaxError, связанная с некорректностью синтаксиса выполняемой команды).

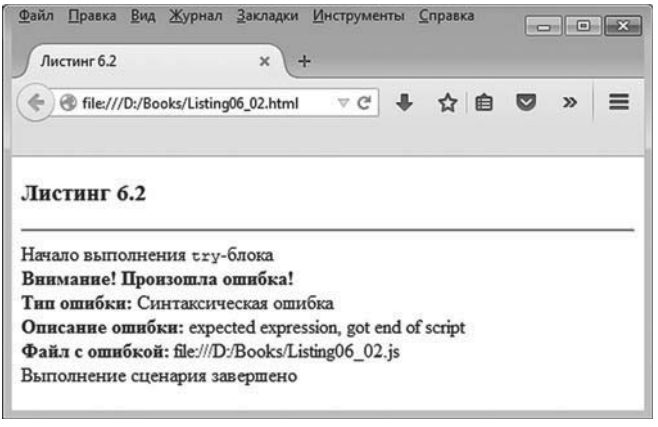

Рис. 6.7. Результат выполнения сценария в случае, когда в текстовое поле вводится выражение +, для браузера Mozilla Firefox

Если мы воспользуемся браузером, отличным от Mozilla Firefox, то результат будет несколько иным. На рис. 6.8 показано, как выглядит открытый в браузере Internet Explorer документ при возникновении синтаксической ошибки.

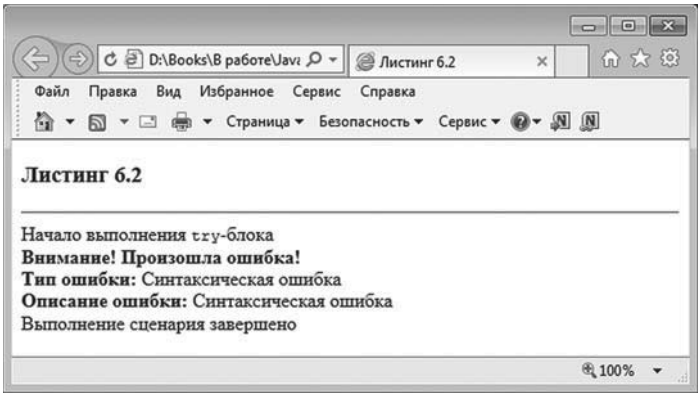

Рис. 6.8. Результат выполнения сценария в случае, когда в текстовое поле вводится выражение +, для браузера Internet Explorer

Несложно заметить, что информация о файле с ошибкой не отображается. Также несколько иначе выглядит стандартное описание ошибки.

НА ЗАМЕТКУ

То, что название ошибки совпало с описанием ошибки, — случайность.

Как выглядит рабочий документ при попытке выполнить команду A+B (возникает ошибка типа ReferenceError, связанная с некорректностью ссылок), показано на рис. 6.9.

Тот же документ, открытый при тех же «обстоятельствах», но уже с помощью браузера Internet Explorer, показан на рис. 6.10.

Наконец, если в текстовое поле диалогового окна ввести выражение new Array(-1) (команда создания массива, но количество элементов в массиве указано отрицательное), то попытка выполнить такую команду приведет к ошибке типа RangeError — ошибка при определении диапазонов. Ошибка данного типа обрабатывается в операторе выбора default-блоком. Как при такой ошибке выглядит рабочий документ, показано на рис. 6.11.

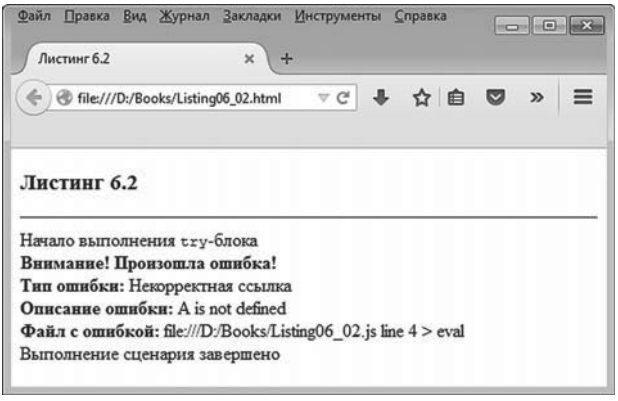

Рис. 6.9. Результат выполнения сценария в случае, когда в текстовое поле вводится выражение *A+B*, для браузера Mozilla Firefox

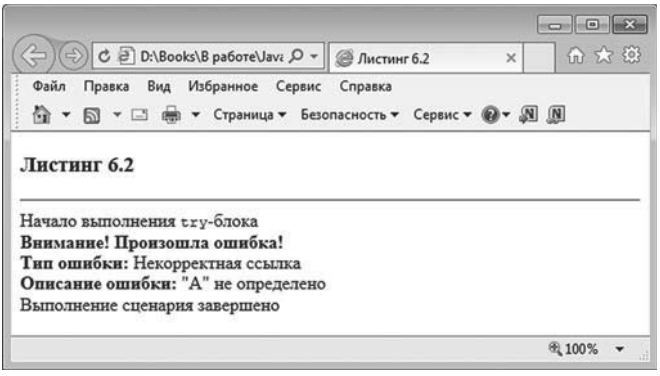

Рис. 6.10. Результат выполнения сценария в случае, когда в текстовое поле вводится выражение *A+B*, для браузера Internet Explorer

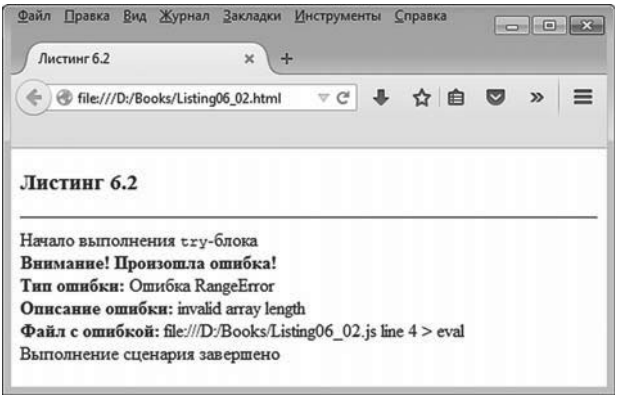

Рис. 6.11. Результат выполнения сценария в случае, когда в текстовое поле вводится выражение *new Array(-1)*, для браузера Mozilla Firefox

На рис. 6.12 для сравнения представлен аналогичный веб-документ, открытый в браузере Internet Explorer.

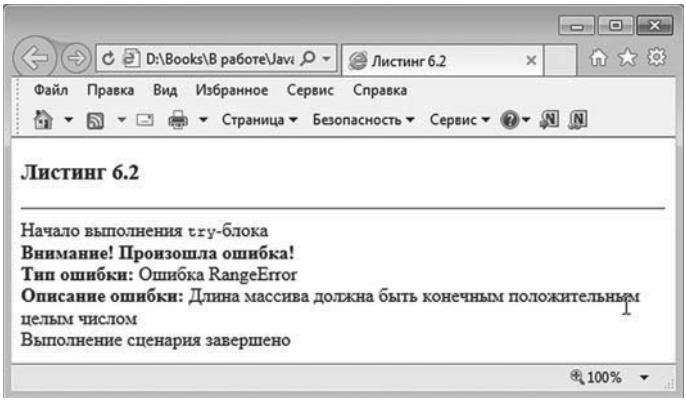

Рис. 6.12. Результат выполнения сценария в случае, когда в текстовое поле вводится выражение *new Array(-1)*, для браузера Internet Explorer

Таким образом, мы видим, что конструкция try-catch позволяет не просто обрабатывать ошибки, а еще и делать такую обработку специфичной в зависимости от типа ошибки.

НА ЗАМЕТКУ

Помимо рассмотренных нами типов ошибки RangeError, SyntaxError и ReferenceError, имеет смысл отметить ошибки типа TypeError, связанные с некорректным типом операнда в выражении, недопустимым типом переменной или аргумента.

Генерирование ошибок

В JavaScript ошибки можно не только обрабатывать, но и генерировать. На первый взгляд такая процедура (имеется в виду генерирование ошибки) может показаться странной и ненужной, но на самом деле все не так. Существует по меньшей мере две ситуации, в которых искусственное генерирование ошибки представляется эффективным способом организации программного кода. Во-первых, генерирование ошибки может использоваться для реализации точек ветвления в программе, подобно тому как это делается с помощью условных операторов. Во-вторых, генерирование ошибок может применяться при организации сложных вложенных структур из try-catch-конструкций.

Мы далее рассмотрим механизм генерирования исключений, так сказать, в «чистом виде».

НА ЗАМЕТКУ

Когда мы говорим о генерировании исключительной ситуации (ошибки), имеется в виду создание некоторой «иллюзии», что ошибка произошла. Речь не идет о преднамеренном внесении в программу ошибочного кода.

Для генерирования исключительной ситуации используют ключевое слово throw, после которого указывается объект ошибки. Команда, генерирующая ошибку, выглядит следующим образом:

throw объект

Такая команда обычно размещается в try-блоке. После ее выполнения управление передается соответствующему catch-блоку. Объект, указанный после ключевого слова throw, играет роль объекта ошибки. В принципе это может быть произвольное значение (текст, например), но нередко объект ошибки создают с помощью стандартного конструктора Error. При использовании конструктора Error создание объекта ошибки выполняется командой вида new Error(аргумент). Аргументом конструктору Error передается текстовое значение, которое затем доступно через свойство message объекта ошибки. Небольшая иллюстрация к тому, как и зачем могут генерироваться исключительные ситуации в программном коде, представлена в листинге 6.3.

Листинг 6.3. Генерирование исключительной ситуации (файл Listing06_03.js)

```
// Числовой массив:
var A=[2,5,9,1,0,3,7,8,6,4]
// Вспомогательная переменная:
var num
// Оператор цикла:
for(var k=0;k<A.length;k++){
  // Значение переменной:
  num=A[k]
  // Первый try-блок:
  try{
```

```
    if(num==0){
      // Генерируется ошибка. Объект ошибки - текст:
      throw "нулевое значение"
    }
    // Команда выполняется, если не сгенерирована ошибка:
    document.write("Отличное от нуля число: "+num+"<br>")
  }
  // Обработка ошибки (объект ошибки - текст):
  catch(e){
    document.write("Внимание: "+e+"<hr>")
    // Завершение текущего цикла и переход
    // к следующему циклу:
    continue
  }
  // Второй try-блок:
  try{
    if(num%3==0){
      // Генерируется ошибка. Объект ошибки
      // создается конструктором Error:
      throw new Error("Число делится на три")
    }
  }
  // Обработка ошибки (объект создан конструктором Error):
  catch(e){
    document.write(e.message)
  }
  // Отображение горизонтальной линии:
  document.write("<hr>")
} // Завершение оператора цикла
```

Проверить работу сценария нам поможет следующий HTML-код:

```
<!DOCTYPE HTML>
<html>
<head>
```

```
  <title>Листинг 6.3</title>
</head>
<body><h3>Листинг 6.3</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing06_03.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Результат выполнения сценария представлен на рис. 6.13.

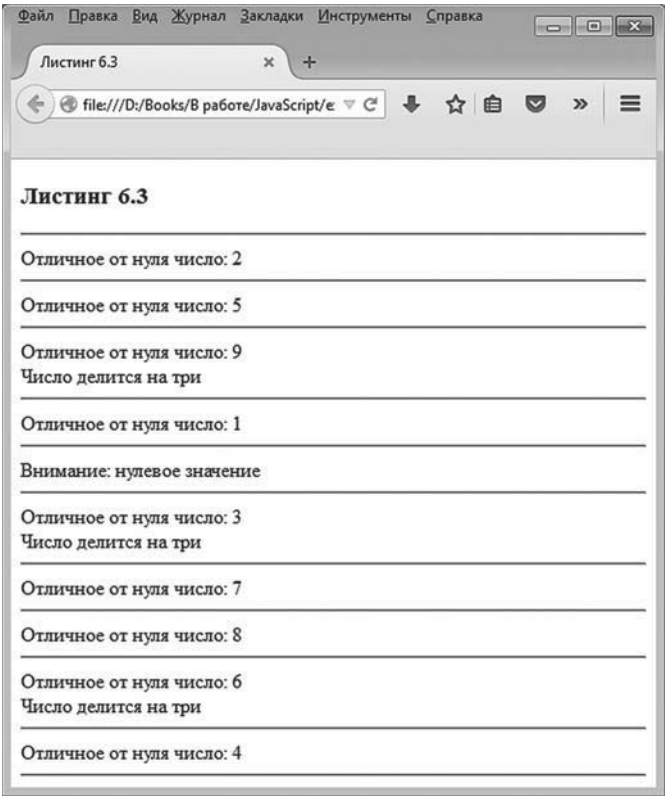

Рис. 6.13. Результат выполнения сценария, в котором искусственно генерируются ошибки

Для удобства восприятия информации ниже приведена текстовая версия результата выполнения сценария (горизонтальные линии заменены пустыми строками).

Результат выполнения сценария (из листинга 6.3)

```
Отличное от нуля число: 2

Отличное от нуля число: 5

Отличное от нуля число: 9
Число делится на три

Отличное от нуля число: 1

Внимание: нулевое значение

Отличное от нуля число: 3
Число делится на три

Отличное от нуля число: 7

Отличное от нуля число: 8

Отличное от нуля число: 6
Число делится на три

Отличное от нуля число: 4
```

В сценарии создается числовой массив A=[2,5,9,1,0,3,7,8,6,4]. Нам он понадобится для перебора числовых значений в диапазоне от 0 до 9, но в неупорядоченной последовательности (так вроде бы интересней). Также в сценарии объявляется вспомогательная переменная num. Ее мы используем в операторе цикла, когда присваиваем данной переменной значение очередного элемента из массива A. В теле оператора цикла последовательно размещены два try-catch блока. В первом try-бло-

ке в условном операторе проверяется условие num==0, и при его истинности командой throw "нулевое значение" генерируется ошибка. Объектом ошибки в данном случае выступает текстовая строка "нулевое значение". Именно она передается в catch-блок под видом переменной e. Поэтому когда в catch-блоке (который соответствует первому try-блоку) выполняется команда document.write("Внимание: "+e+"<hr>"), то переменная e обозначает именно текст "нулевое значение". После отображения текста в рабочем документе в catch-блоке выполняется команда continue. Данной командой останавливается выполнение текущего цикла и начинает выполняться новый цикл. Все это происходит, напомним, при истинном условии num==0. Если условие ложно, то ошибка не генерируется и в try-блоке выполняется команда document.write("Отличное от нуля число: "+num+"
"). Затем в игру вступает второй try-catch блок. В нем в условном операторе проверяется условие num%3==0, которое истинно, если остаток от деления значения переменной num на 3 равен 0 (другими словами, условие истинно, если число num делится без остатка на 3). В таком случае командой throw new Error("Число делится на три") генерируется новая исключительная ситуация. Объект ошибки создается конструктором Error. Текст, переданный аргументом конструктору, записывается значением свойства message объекта ошибки. Поэтому в catch-блоке при обработке объекта ошибки e значением выражения e.message является текст "Число делится на три".

Вложенные try-catch-блоки и блок finally

Блоки try-catch могут быть вложенными. В частности, try-блок (будем называть его *внешним* try-блоком) может содержать try-catch-блок (будем называть *внутренним* try-catch-блоком). Если во внутреннем try-блоке происходит ошибка, то она перехватывается внутренним catch-блоком (тем catch-блоком, который идет «в паре» с внутренним try-блоком). Если ошибка возникает во внешнем try-блоке (но не во внутреннем try-блоке), то обрабатывается она в catch-блоке, соответствующем внешнему try-блоку. Ситуация может быть более запутанной. Скажем, внутренний catch-блок генерирует ошибку (как происходит в рассматриваемом далее примере). Тогда она перехватывается внешним catch-блоком.

Помимо блоков try и catch, может использоваться блок finally. В finally-блок помещают программный код, который должен быть выполнен вне зависимости от того, возникла ли в try-блоке ошибка или нет.

Конструкция try-catch-finally выглядит следующим образом (жирным шрифтом выделены ключевые элементы кода):

```
try{
  // контролируемый код
}
catch(ошибка){
  // код обработки исключения
}
finally{
  // обязательный для выполнения код
}
```

Если в try-блоке исключение не генерируется, то после выполнения команд в этом блоке начинают выполняться команды в finally-блоке. Если в try-блоке возникла ошибка, то она обрабатывается в catch-блоке, после чего выполняется finally-блок. В данном случае польза от finally-блока не очевидна, поскольку все точно то же происходило бы, если команды из finally-блока были просто размещены после try-catch-конструкции. Ситуация несколько иная, если, например, при выполнении команд catch-блока происходит ошибка. В этом случае управление передается внешнему catch-блоку. Но перед этим будут выполнены команды в finally-блоке.

Теперь рассмотрим пример, представленный в листинге 6.4. В нем использован блок finally, а также вложенные try-catch-блоки.

Листинг 6.4. Вложенные try-catch-блоки и блок finally (файл Listing06_04.js)

```
// Оператор цикла:
for(var x=-1;x<=1;x+=2){
  // Внешний try-catch блок:
  try{
  document.write("Начало внешнего <code>try-catch</code> блока<br>")
    // Внутренний try-catch блок:
    try{
      document.write("Начало внутреннего <code>try-catch</code> блока<br>")
      // Создание массива (возможна ошибка RangeError):
      var A=new Array(x)
```

```
      // Некорректная команда:
      var B=C
    }
    // Внутренний обработчик:
    catch(e){
      // Если ошибка RangeError:
      if(e.name=="RangeError"){
        e.message="Неверный размер массива (ошибка <code>RangeError</code>)"
        // Повторное генерирование исключения:
        throw e
      }
      document.write("Некорректное присваивание<br>")
    }
    // Блок выполняется в любом случае:
    finally{
      document.write("Завершение внутреннего <code>try-catch</code> блока<br>")
    }
    document.write("Ошибки <code>RangeError</code> не было<br>")
  }
  // Внешний обработчик:
  catch(err){
    document.write(err.message+"<br>")
  }
  document.write("Завершение внешнего <code>try-catch</code> блока<hr>")
} // Завершение оператора цикла
```

Проверка работы сценария выполняется с помощью следующего HTML-кода:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 6.4</title>
</head>
<body><h3>Листинг 6.4</h3><hr>

<!-- Начало сценария -->
```

```
<script type="text/javascript" src="Listing06_04.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 6.14 показано, как будет выглядеть в окне браузера результат выполнения сценария.

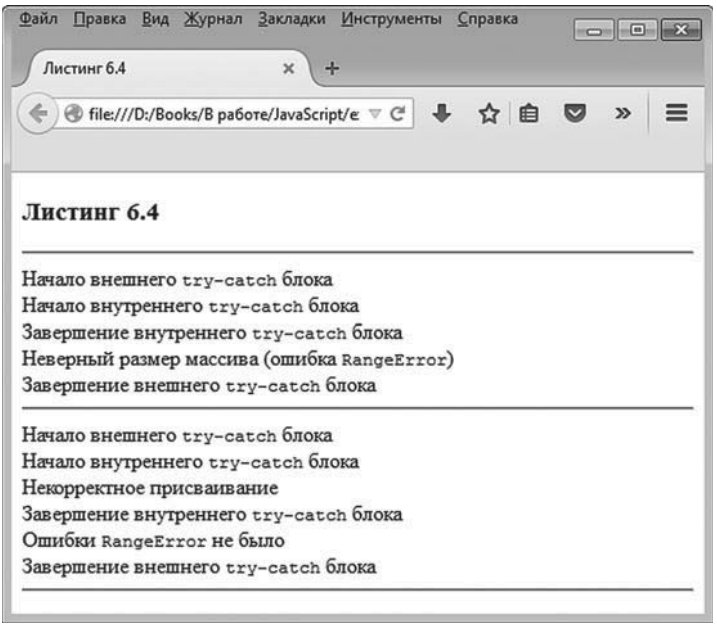

Рис. 6.14. Результат выполнения сценария, в котором есть вложенные *try-catch*-блоки и используется блок *finally*

Текстовая версия выводимых при выполнении сценария сообщений представлена ниже (горизонтальные разделительные линии заменены пустыми строками).

Результат выполнения сценария (из листинга 6.4)

```
Начало внешнего try-catch блока
Начало внутреннего try-catch блока
Завершение внутреннего try-catch блока
```

```
Неверный размер массива (ошибка RangeError)
Завершение внешнего try-catch блока

Начало внешнего try-catch блока
Начало внутреннего try-catch блока
Некорректное присваивание
Завершение внутреннего try-catch блока
Ошибки RangeError не было
Завершение внешнего try-catch блока
```

Основу сценария составляет оператор цикла, в котором переменная x принимает всего два значения: -1 и 1. При каждом из этих значений генерируется ошибка, но каждый раз разная. Для отслеживания ошибок используются вложенные try-catch-блоки. Во внутреннем try-блоке выполняется команда var A=new Array(x). Она приводит к ошибке RangeError при значении -1 для переменной x. Но даже если данная команда к ошибке не приводит (ошибки RangeError нет при значении 1 для переменной x), команда var B=C точно сгенерирует ошибку, поскольку переменная C в сценарии не определена. Итак, при выполнении внутреннего try-блока однозначно происходит ошибка. Первоначально она перехватывается во внутреннем catch-блоке. Объект ошибки в данном блоке обозначен переменной e.

В теле внутреннего catch-блока в условном операторе проверяется тип сгенерированной ошибки. Условие e.name=="RangeError" истинно, если возникла ошибка, связанная с некорректным размером для массива. В этом случае командой e.message="Неверный размер массива (ошибка <code>RangeError</code>)" изменяется значение свойства message объекта ошибки, после чего командой throw e выполняется повторное генерирование ошибки.

ДЕТАЛИ

При генерировании ошибки объектом ошибки указана переменная e — та самая переменная, что определена в catch-блоке как объект перехватываемой ошибки. Данный объект создается автоматически, когда генерируется исключение и передается во внутренний catch-блок. Там он через команду throw e «выбрасывается» во внешний catch-блок. Важно понимать, что это тот же самый объект, который был передан во внутренний catch-блок.

Команда document.write("Некорректное присваивание
") следует за условным оператором. Учитывая, что в условном операторе при истинности условия генерируется исключение, означенная команда выполняется только в том случае, если условие ложно. А это случается, если ошибка генерируется при выполнении команды var B=C, что, в свою очередь, случается при значении 1 для переменной x.

Блок finally содержит единственную команду document.write("Завершение внутреннего <code>try-catch</code> блока
"). После этого блока есть еще одна команда document.write("Ошибки <code>RangeError</code> не было
").

Теперь попробуем разобраться, что происходит при выполнении внутреннего try-блока и сопутствующего catch-блока. Возможны две ситуации.

- Ошибка типа RangeError происходит при выполнении команды var A=new Array(x).
- Команда var A=new Array(x) выполняется без ошибки, но ошибка типа ReferenceError происходит при выполнении команды var B=C. И это не ошибка типа RangeError.

Итак, если ошибка возникает при выполнении команды var A=new Array(x), то в условном операторе в catch-блоке генерируется ошибка и управление передается внешнему catch-блоку. Но перед этим будет выполнен finally-блок. А вот если ошибка возникает при выполнении команды var B=C, то во внутреннем catch-блоке выполняется команда document.write("Некорректное присваивание
"), затем выполняется блок finally и еще команда document.write("Ошибки <code>RangeError</code> не было
") после finally-блока. На этом выполнение внутренней try-catch-конструкции завершается.

Что касается внешнего catch-блока, то в нем всего одна команда document.write(err.message+"
"), которая «вступает в игру» вследствие того, что во внутреннем catch-блоке была повторно сгенерирована ошибка типа RangeError. Через err здесь обозначен объект ошибки, который передается в catch-блок.

Речь идет об объекте, который был «выброшен» из внутреннего catch-блока. Во внутреннем catch-блоке в явном виде задавалось значение свойства message для объекта ошибки. Поскольку там и здесь объект один и тот же, то нет ничего удивительного, что свойство message у объекта err точно такое же, как было присвоено объекту ошибки во внутреннем catch-блоке.

Создание ошибки пользовательского типа

Объект ошибки, как несложно понять из изложенного выше материала, или автоматически генерируется в сценарии (когда реально возникает ошибка), или создается «вручную». Если объект генерируется автоматически, то все, что нужно сделать в сценарии, — перехватить ошибку и обработать ее. При создании объекта ошибки «вручную», несмотря на наличие иных возможностей, нередко используют конструктор Error. Примеры использования данного конструктора нам уже встречались. У объектов, созданных с помощью конструктора Error, имеются свойства name и message, которые позволяют добавлять в объект ошибки некоторую полезную информацию. Также можно отметить наличие у объектов, созданных конструктором Error, метода toString(), который вызывается автоматически при приведении объекта ошибки к текстовому формату. Таким образом, некоторый набор полезных «характеристик» у конструктора Error есть, что позволяет с его помощью решать эффективно самые разные задачи. Вместе с тем иногда бывает полезно создать собственный объект, описывающий исключительную ситуацию, с генеалогией, берущий свое начало у конструктора Error. Как иллюстрацию такого подхода рассмотрим пример, представленный в листинге 6.5.

Листинг 6.5. Ошибка пользовательского типа (файл Listing06_05.js)

```
// Конструктор для объекта ошибки:
function MyError(id,message){
  // Свойство id:
  this.id=id
  // Свойство message:
  this.message=message || "Ошибка пользовательского типа"
}
// Прототип для объектов ошибки пользовательского типа:
MyError.prototype=Object.create(Error.prototype)
// Свойство name прототипа:
MyError.prototype.name="MyError"
// Метод toString() прототипа:
MyError.prototype.toString=function(){
  // Локальная переменная с текстовым значением:
```

```
  var t=this.message+": "+this.name+". "
  t+="Код ошибки: "+this.id+"."
  // Результат метода:
  return t
}
// Конструктор для объектов ошибок пользовательского типа:
MyError.prototype.constructor=MyError
// Контролируемый код (внешний блок):
try{
  // Контролируемый код (внутренний блок):
  try{
    // Генерирование исключения пользовательского типа:
    throw new MyError(200,"Рукотворная ошибка")
  }
  // Обработка исключения (внутренний блок):
  catch(e){
    document.write(e+"<br>")
    // Генерирование ошибки:
    throw new e.constructor(100)
  }
}
// Обработка исключения (внешний блок):
catch(e){
  document.write(e)
}
```

В данном случае в сценарии описывается функция-конструктор MyError(), предназначенная для создания объектов ошибок. У конструктора предполагается наличие двух аргументов: id и message соответственно. Они определяют значения одноименных свойств объекта ошибки. Предполагается, что свойство id представляет собой некоторый числовой код, а свойство message содержит описание ошибки. В теле функции-конструктора командой this.message=message || "Ошибка пользовательского типа" свойству message создаваемого объекта присваивается значение.

ДЕТАЛИ

Свойству message объекта (ссылка на созданный объект дается инструкцией this) присваивается значением выражение message || "Ошибка пользовательского типа". Формально это логическое выражение, основу которого составляет оператор *логического или* ||. Здесь мы воспользовались некоторыми особенностями вычисления результата выражения на основе оператора ||. В частности, если первый операнд (в данном случае аргумент message) имеет значение, интерпретируемое как true, то результатом всего выражения возвращается значение данного операнда. Если значением аргумента message указана текстовая строка, то такое значение интерпретируется как true, поэтому результатом выражения на основе оператора *логического или* является значение аргумента message (первый операнд). Если при создании объекта аргумент message конструктору не передается, то значение аргумента message не определено (значение undefined). Такое значение интерпретируется как false, и результатом выражения message || "Ошибка пользовательского типа" возвращается второй операнд, то есть текст "Ошибка пользовательского типа". Таким образом, получаем ситуацию, когда свойству message созданного объекта присваивается значение аргумента message, если аргумент передан конструктору при вызове, и текст "Ошибка пользовательского типа", если аргумент message конструктору не передавался. Фактически текстовое значение "Ошибка пользовательского типа" является значением по умолчанию для аргумента message конструктора.

Кроме описания функции-конструктора, сценарий содержит еще несколько важных команд. Так, командой MyError.prototype=Object.create(Error.prototype) создается прототип для объектов ошибок пользовательского типа. Объект-прототип создается командой Object.create(Error.prototype). Аргумент Error.prototype метода create() означает, что объект-прототип сам создается на основе прототипа объекта Error. Ссылка на созданный объект-прототип записывается в свойство prototype конструктора MyError. Командой MyError.prototype.name="MyError" для объекта-прототипа задается свойство name со значением "MyError". Соответственно, у всех объектов, созданных с помощью функции-конструктора MyError(), будет свойство name по умолчанию со значением "MyError". Также в объекте-прототипе конструктора MyError определяется код для метода toString(). Свойству MyError.prototype.toString значением присваивается функция, которая результатом возвращает текстовую строку, содержащую, кроме прочего, значения свойств message, name и id объекта (имеется в виду тот объект ошибки, который приводится к текстовому формату). Еще од-

ной командой MyError.prototype.constructor=MyError задается значение свойства constructor для объекта-прототипа конструктора MyError. Благодаря этому свойству на основе объекта ошибки пользовательского типа можно получить доступ к конструктору объекта ошибки. Понятно, что таким конструктором является MyError.

Проверка возможностей функции-конструктора MyError() выполняется с помощью вложенных блоков try-catch. Во внутреннем try-блоке командой throw new MyError(200,"Рукотворная ошибка") генерируется ошибка пользовательского типа. Объект ошибки создается инструкцией new MyError(200,"Рукотворная ошибка"). Ссылка на соответствующий объект ни в какую переменную не записывается (получается, таким образом, анонимный объект), а сразу передается в throw-выражение.

Ошибка перехватывается и обрабатывается в catch-блоке. Там командой document.write(e+"
") с неявным вызовом метода toString() отображается информация об объекте ошибки e, а затем командой throw new e.constructor(100) генерируется новая ошибка. Она перехватывается во внешнем catch-блоке, в котором сведения о новом объекте ошибки отображаются в рабочем документе с помощью команды document.write(e) (как и в предыдущем случае, здесь e обозначает объект ошибки).

ДЕТАЛИ

Свойство constructor объекта ошибки e возвращает ссылку на конструктор, которым создавался объект ошибки. В данном примере речь идет о конструкторе MyError. Поэтому выражение new e.constructor(100) представляет собой аналог выражения new MyError(100), которым, собственно, и создается объект ошибки. Что касается свойства constructor, то объекты ошибки получают его через свой прототип: мы предусмотрительно задали значение свойства constructor для прототипа объектов ошибок, создаваемых с помощью функции-конструктора MyError().

Для проверки работы сценария используем приведенный ниже HTML-код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 6.5</title>
</head>
```

```
<body><h3>Листинг 6.5</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing06_05.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Как выглядит результат выполнения сценария, в котором создается объект ошибки пользовательского типа, показано на рис. 6.15.

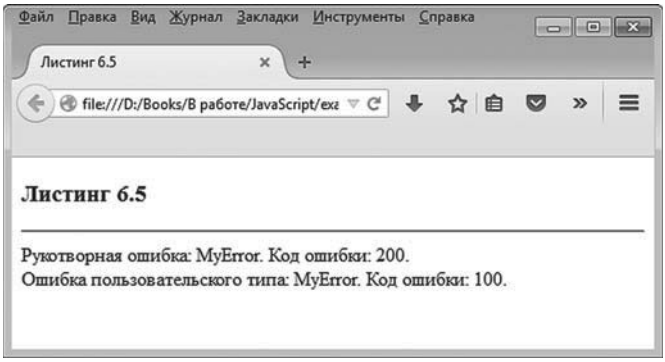

Рис. 6.15. Результат выполнения сценария, в котором генерируется ошибка пользовательского типа

Текстовая версия результата выполнения сценария представлена ниже.

Результат выполнения сценария (из листинга 6.5)

```
Рукотворная ошибка: MyError. Код ошибки: 200.
Ошибка пользовательского типа: MyError. Код ошибки: 100.
```

Понятно, что рассмотренный выше пример простой и иллюстративный. Вместе с тем он демонстрирует основной идеологический подход, применяемый при использовании ошибок пользовательского типа.

Объекты и массивы

> Не знаю, Лёва, как это с самоотречением, а тонкости эта работа не требует. Она выполняется шпитцштихелем!
>
> *из к/ф «Покровские ворота»*

Далее мы остановимся на некоторых особенностях объектов, которые при умелом использовании позволяют значительно повысить эффективность программных кодов. Также нам понадобятся некоторые сведения из области массивов.

Объект как ассоциативный массив

Мы помним, что классический (одномерный) массив представляет собой упорядоченный набор элементов, для обращения к которым используется название массива и индекс элемента (индексация начинается с нуля). Но есть такое понятие, как *ассоциативный массив* (иногда еще называют *словарем*). Принципиальное отличие ассоциативного массива от обычного массива состоит в том, что в ассоциативном массиве «индексами» являются не числа, а произвольные значения (значения нечисловых типов). В этом смысле обычный массив является частным случаем ассоциативного массива. В ассоциативном массиве вместо понятия *индекс* используется понятие *ключ*. Таким образом, в ассоциативном массиве «спрятан» набор элементов (теперь это неупорядоченный набор), у каждого из которых есть *ключ* и *значение*. Если указать имя массива и ключ элемента в массиве, можно узнать значение соответствующего элемента массива.

> **НА ЗАМЕТКУ**
>
> Ассоциативный массив — это концепция, общий подход, идеология. То, что называется массивом в JavaScript, реализуется так, как описывалось ранее, — там ничего не изменилось. Другими словами, в JavaScript есть обычные массивы, которые мы рассматривали в одной из предыдущих глав.

В языке JavaScript объекты реализованы по принципу ассоциативных массивов. Роль ключей играют названия свойств объекта (свойства здесь подразумеваются в широком смысле — то есть с учетом мето-

дов, интерпретируемых как свойства, значениями которых являются функции), а значения свойств отождествляются со значениями элементов ассоциативного массива. Это как бы общая аналогия. Но есть и практическая сторона вопроса. Состоит она в том, что при обращении к свойству объекта вместо «точечного» синтаксиса (свойство указывается через точку после названия объекта) разрешается использовать «индексный» синтаксис: название свойства указывается после имени объекта в квадратных скобках, аналогично индексу для элемента массива.

НА ЗАМЕТКУ

На самом деле с таким способом обращения к свойствам объектов мы уже встречались, когда использовали оператор цикла in-for для перебора свойств объекта.

Небольшая иллюстрация к сказанному представлена в листинге 6.6.

Листинг 6.6. Объект как ассоциативный массив

```
// Создание объекта:
var obj={name:"объект",code:123}
// Добавление свойства в объект:
obj["text"]="текст"
// Формирование названий новых свойств
// и добавление их в объект:
for(var k=1;k<=5;k++){
  obj["symbol_"+k]=String.fromCharCode("A".charCodeAt(0)+k-1)
}
// Отображение названий и значений свойств объекта:
for(var s in obj){
  document.write("Свойство <b>"+s+"</b>: "+obj[s]+"<br>")
}
// Массив с названиями свойств объекта:
var list=Object.keys(obj)
// Отображение названий свойств объекта:
document.write("["+list.join(" | ")+"]")
```

Проверку работы сценария реализуем с помощью следующего HTML-кода:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 6.6</title>
</head>
<body><h3>Листинг 6.6</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing06_06.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Текстовая версия результата выполнения сценария представлена ниже.

Результат выполнения сценария (из листинга 6.6)

```
Свойство name: объект
Свойство code: 123
Свойство text: текст
Свойство symbol_1: A
Свойство symbol_2: B
Свойство symbol_3: C
Свойство symbol_4: D
Свойство symbol_5: E
[name | code | text | symbol_1 | symbol_2 | symbol_3 | symbol_4 | symbol_5]
```

На рис. 6.16 показано, как будет выглядеть веб-документ со сценарием, открытый в окне браузера.

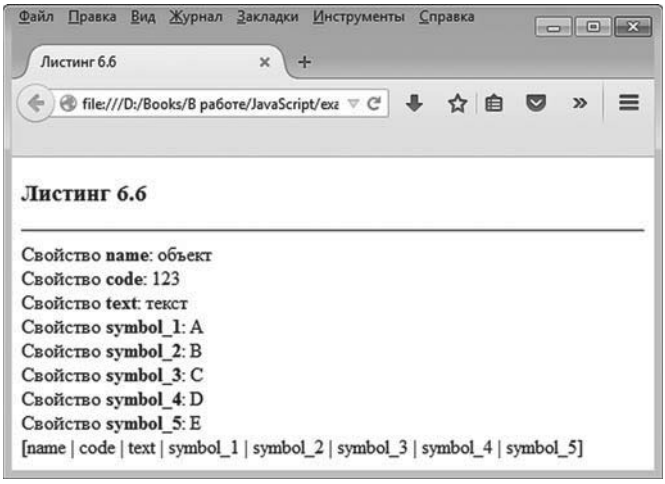

Рис. 6.16. Результат выполнения сценария, в котором обращение к свойствам объекта выполняется в «индексной» нотации

Программный код сценария достаточно простой. Главная его особенность состоит в том, что обращение к свойствам объекта выполняется в «индексном» формате, когда название свойства указывается в кавычках в квадратных скобках после имени объекта.

В сценарии командой `var obj={name:"объект",code:123}` создается объект `obj` с двумя свойствами `name` и `code`. Еще одно свойство `text` добавляется в объект при выполнении команды `obj["text"]="текст"`. В данном случае вместо инструкции `obj.text` мы использовали альтернативное выражение `obj["text"]`. Затем запускается оператор цикла, в котором индексная переменная `k` пробегает значения от `1` до `5` и за каждый цикл выполняется команда `obj["symbol_"+k]=String.fromCharCode("A".charCodeAt(0)+k-1)`. Данная команда заслуживает некоторых пояснений. Причем как правая, так и левая ее части.

В левой части выражения находится инструкция `obj["symbol_"+k]`, которая представляет собой обращение к свойству объекта `obj`. Название свойства определяется выражением `"symbol_"+k`. Это текст, представляющий собой объединение строки `"symbol_"` и текущего значения индексной переменной `k`. Здесь мы сталкиваемся с очень важной особенностью «индексной» нотации при обращении к свойствам объекта. Дело в том, что если обращение к свойству объекта выполняется по обобщенному индексу, то такой индекс может быть *вычисляемым*. Другими словами, значением индекса-свойства может быть не только константное тек-

стовое значение с названием свойства, но и выражение, которое при вычислении дает текстовую строку, содержащую название свойства. Такой механизм очень гибкий и весьма эффективный.

В правой части выражения расположена достаточно громоздкая конструкция String.fromCharCode("A".charCodeAt(0)+k-1), которая на самом деле простая. Значением выражения "A".charCodeAt(0) является код (в кодовой таблице) символа с нулевым индексом в текстовой строке "A". Текстовая строка "A" состоит из одного символа A, который, естественно, имеет нулевой индекс (индексация символов в текстовой строке начинается с нуля, поэтому первый по порядку символ имеет нулевой индекс). Но результат выражения — не сам символ, а его код в кодовой таблице (данный код равен 65, но в данном случае это не важно). К коду прибавляется текущее значение индексной переменной k и вычитается 1. Полученное число передается аргументом методу fromCharCode() встроенного объекта String(). Методом fromCharCode() по коду символа в кодовой таблице возвращается собственно символ.

НА ЗАМЕТКУ

Таким образом, для определения кода символа используем метод charCodeAt(), который вызывается из объекта текстовой строки, а аргументом методу передается индекс символа в этой текстовой строке. Выполнить обратную процедуру (то есть по коду символа определить сам символ) позволяет метод fromCharCode(), который вызывается из встроенного объекта String, а аргументом методу передается код символа.

Для отображения названий свойств объекта использован оператор цикла in-for, в котором переменная цикла s последовательно принимает значениями названия свойств объекта obj. В таком случае s отождествляется со свойством (его названием), а обращение к этому свойству объекта obj выполняется инструкцией obj[s].

Также в сценарии есть пример использования метода keys() встроенного объекта Object. Методу аргументом передается объект, а результатом возвращается массив с названиями собственных свойств объекта. Поэтому после выполнения команд var list=Object.keys(obj) переменная list содержит ссылку на массив с названиями свойств объекта obj. После выполнения команды document.write("["+list.join(" | ")+"]") в рабочем документе отображается список с названиями свойств объекта obj.

НА ЗАМЕТКУ

Напомним, что методом join() возвращается текстовая строка, получающаяся объединением значений элементов массива, из которого вызывается метод. Разделителем при объединении значений элементов служит текст, переданный аргументом методу.

Методы toString() и valueOf() для объектов

Мы встречались с методами toString() и valueOf() при обсуждении массивов. Вообще же данные методы имеют отношение не только к массивам, но и ко всем объектам. Метод toString() отвечает за преобразование объекта к текстовому формату, а метод valueOf() позволяет преобразовывать объекты к значениям простых (базовых) типов.

ДЕТАЛИ

Методы toString() и valueOf() определены в объекте-прототипе Object.prototype, который является прототипом верхнего уровня. Поэтому обычно объект наследует данные методы через цепочку наследования прототипов (исключение составляют те объекты, которые создаются без прототипа или на основе объектов, которые создавались без прототипа). Другое дело, что стандартное определение этих методов делает их малопригодными для использования с пользовательскими объектами. Поэтому обычно данные методы в объекте переопределяются (то есть описываются заново в соответствии с потребностями программы и решаемой задачи).

В листинге 6.7 представлен небольшой сценарий, в котором используется переопределение методов toString() и valueOf() для пользовательского объекта.

Листинг 6.7. Переопределение методов toString() и valueOf() для пользовательского объекта (файл Listing06_07.js)

```
// Метод toString() определяется
// в прототипе верхнего уровня:
Object.prototype.toString=function(){
  var t="<u>Собственные свойства объекта:</u><br>"
  for(var s in this){
    t+="<b>"+s+"</b>: "+this[s]+"</br>"
```

```
  }
  return t
}
// Метод valueOf() определяется
// в прототипе верхнего уровня:
Object.prototype.valueOf=function(){
  return Object.keys(this).length
}
// Первый объект:
var A={one:100,two:200,three:300,four:400,five:500}
// Неявный вызов метода toString() при отображении
// свойств объекта:
document.write(A)
// Неявный вызов метода valueOf() для определения
// количества собственных свойств у объекта:
document.write("Количество свойств в объекте - "+A+"<br>")
// Второй объект:
var B={first:"первый",second:"второй",third:"третий"}
// Неявный вызов метода toString() при отображении
// свойств объекта:
document.write(B)
// Неявный вызов метода valueOf() для определения
// количества собственных свойств у объекта:
document.write("Количество свойств в объекте - "+B+"<br>")
// Функция-конструктор:
function Person(name,age,gender){
  // Локальная переменная:
  var g
  // Определение значения локальной переменной:
  if(gender){
    g="муж."
  }
  else{
    g="жен."
```

```
  }
  // Значение свойства name:
  this.name=name
  // Значение свойства age:
  this.age=age
  // Свойство gender доступно только для считывания:
  Object.defineProperty(this,"gender",{
    get:function(){
      return g
    }
  })
}
// Определение метода toString() для
// прототипа конструктора Person:
Person.prototype.toString=function(){
  var t="<u>Персональные данные:</u><br>"
  t+="<b>Имя:</b> "+this.name+"<br>"
  // Неявный вызов метода valueOf():
  t+="<b>Возраст:</b> "+this+" лет<br>"
  t+="<b>Пол:</b> "+this.gender+"<br>"
  return t
}
// Определение метода valueOf() для
// прототипа конструктора Person:
Person.prototype.valueOf=function(){
  return this.age
}
// Объекты создаются с помощью конструктора Person:
var X=new Person("Ирина",20,false)
var Y=new Person("Андрей",25,true)
// Неявный вызов метода toString() для объектов, созданных
// с помощью конструктора Person:
document.write(X)
document.write(Y)
```

```
// Неявный вызов метода valueOf() для объектов, созданных
// с помощью конструктора Person:
document.write("Возраст Ирины - "+X+" лет<br>")
document.write("Возраст Андрея - "+Y+" лет")
```

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

В сценарии использованы дескрипторы <u> и </u> для отображения текста с подчеркиванием, а также и для применения жирного шрифта.

Для проверки работы сценария используем следующий HTML-код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 6.7</title>
</head>
<body><h3>Листинг 6.7</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing06_07.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Далее представлена текстовая версия для результата выполнения сценария.

Результат выполнения сценария (из листинга 6.7)

Собственные свойства объекта:

one: 100

two: 200

three: 300

four: 400

five: 500

Количество свойств в объекте - 5

Собственные свойства объекта:

first: первый

second: второй

third: третий

Количество свойств в объекте - 3

Персональные данные:

Имя: Ирина

Возраст: 20 лет

Пол: жен.

Персональные данные:

Имя: Андрей

Возраст: 25 лет

Пол: муж.

Возраст Ирины - 20 лет

Возраст Андрея - 25 лет

На рис. 6.17 показано рабочее окно веб-документа со сценарием, открытое в окне браузера.

Что касается непосредственно сценария, то в нем метод toString() добавляется в прототип Object.prototype (прототип высшего уровня). В соответствии с этим определением метода преобразование объектов к текстовому формату выполняется так: отображаются названия свойств объекта и их значения. Поскольку такой метод toString() задан для прототипа Object.prototype, то фактически все объекты, которые создаются стандартным образом (например, описанием литерала объекта), будут преобразовываться к текстовому формату так, как описано выше.

НА ЗАМЕТКУ

Метод toString() нами определен так, что результирующая строка содержит дескрипторы HTML-разметки. Это подход не очень хороший, поскольку «привязывает» результат метода к среде выполнения сценария. На практике подобных ситуаций следует избегать.

Но поскольку пример иллюстративный и нас интересует в первую очередь наглядность результата, то думается, что большой беды в том нет.

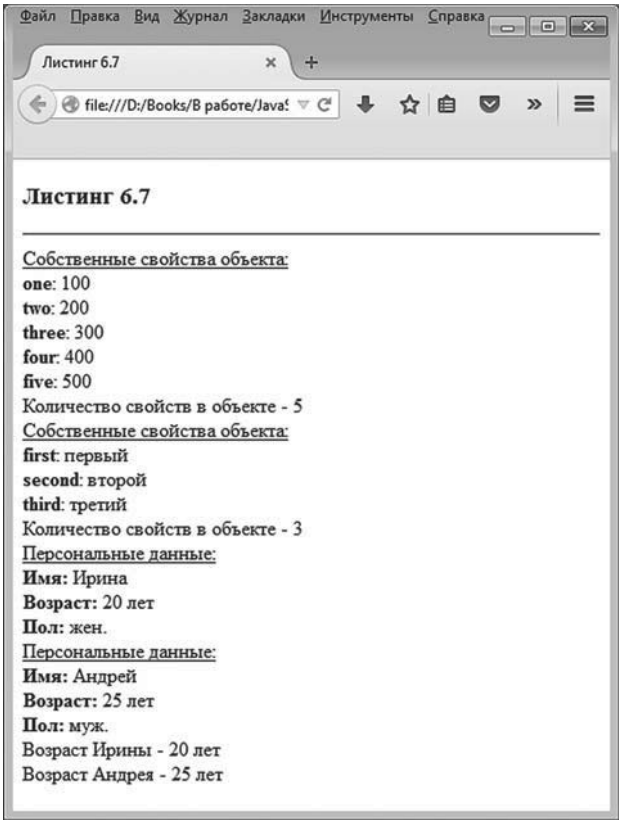

Рис. 6.17. Результат выполнения сценария, в котором определяются методы *toString()* и *valueOf()* для пользовательского объекта

Метод valueOf() также определяется в прототипе верхнего уровня Object.prototype. В соответствии с кодом метода результатом он возвращает значение выражения Object.keys(this).length. Это количество собственных свойств объекта (под свойствами здесь подразумеваются и собственные методы объекта).

ДЕТАЛИ

Метод keys() встроенного объекта Object возвращает результатом массив с названиями свойств объекта. Ссылка на объект (в данном случае ключевое слово this) передается аргументом методу. Свой-

ство length массива позволяет определить количество элементов в массиве.

В сценарии создаются два объекта: A и B. Это разные объекты, но поскольку методы toString() и valueOf() определены для прототипа высшего уровня, то оба объекта «подпадают» под юрисдикцию данных методов. При выполнении команд document.write(A) и document.write(B) вызывается метод toString(), а при выполнении команд document.write("Количество свойств в объекте — "+A+"
") и document.write("Количество свойств в объекте — "+B+"
") вызывается метод valueOf().

НА ЗАМЕТКУ

Метод valueOf() вызывается в тех случаях, когда объект, исходя из контекста команды, должен преобразовываться в значение простого типа. Метод toString() вызывается в случаях, когда объект должен преобразоваться к текстовому формату. Также напомним, что методы valueOf() и toString() можно вызывать в явном виде (как обычные методы объекта).

Еще в сценарии описана функция-конструктор Person(). Предполагается, что у функции-конструктора три аргумента. Объект, который создается конструктором, имеет три свойства: name (имя), age (возраст) и gender (пол). Причем последнее является свойством с режимом доступа и может быть только считано.

ДЕТАЛИ

В теле функции-конструктора объявляется локальная переменная g. Значение этой переменной присваивается в условном операторе на основе значения третьего аргумента gender функции-конструктора. Если третий аргумент конструктора интерпретируется как логическое значение true, то переменная g получает значение "муж." (мужской пол). Если третий аргумент конструктора интерпретируется как логическое значение false, то переменная g получает значение "жен." (женский пол).

Свойство gender добавляется в объект командой Object.defineProperty(this,"gender",{get:function(){return g}}). В команде мы используем метод defineProperty() объекта Object. Методом в объект, указанный первым аргументом (ссылка this) добавляется свойство, указанное вторым аргументом (имя "gender") с характеристиками, определяемыми третьим аргументом. В данном случае третьим аргументом пере-

дается описание get-метода для считывания значения свойства. Несложно заметить, что результатом возвращается значение переменной g. Хотя данная переменная локальная, но поскольку она определяет результат метода, вызываемого после завершения работы конструктора, то данная переменная будет сохраняться в памяти, даже когда конструктор завершит свое выполнение. Причем для каждого нового вызова конструктора создается и «запоминается» новая локальная переменная g. В этом смысле переменная g играет роль закрытого поля объекта, по аналогии с языками C++, C# и Java.

Для прототипа конструктора Person методы toString() и valueOf() переопределяются. Поэтому при выполнении команд document.write(X) и document.write(Y) вызывается версия метода toString(), определенная для объекта-прототипа Person.prototype, а при выполнении команд document.write("Возраст Ирины - "+X+" лет
") и document.write("Возраст Андрея - "+Y+" лет") вызывается версия метода valueOf(), определенная для этого же прототипа. Объекты X и Y предварительно создаются с помощью конструктора Person (команды var X=new Person("Ирина",20,false) и var Y=new Person("Андрей",25,true)).

ДЕТАЛИ

Для объекта-прототипа Person.prototype метод valueOf() определен так, что результатом возвращается свойство age объекта. Данная особенность определения метода valueOf() использована в описании метода toString(). Речь идет об инструкции t+="Возраст: "+this+" лет
" в теле метода toString(). В данной инструкции к текстовому значению прибавляется ссылка this на объект, из которого вызывается метод. Это как раз та ситуация, когда в дело вступает метод valueOf(), преобразующий объект в числовое значение (под преобразованием имеется в виду вычисление на основе параметров объекта некоторого числового значения).

Массивы и объекты как свойства объекта

Массив или объект может быть свойством объекта. Особенного в том ничего нет, хотя на некоторые моменты внимание обратить стоит. Главный из них состоит в том, что доступ к массивам и объектам выполняется через ссылки, поэтому некоторые операции, наподобие копирования объектов и массивов, требуют особого

внимания. Хотя, конечно, многое зависит от конкретики решаемой задачи.

Далее рассматривается небольшой пример, в котором создается объект, у которого среди свойств есть и массив объектов. Если более конкретно, то в рассматриваемом далее сценарии решается задача о создании так называемого дерева: есть объект, который содержит ссылки на несколько объектов, каждый из которых содержит ссылки на несколько объектов, и так далее (понятно, что где-то этот процесс заканчивается). Такую конструкцию будем называть *деревом объектов*. На рис. 6.18 схематически представлено дерево объектов, в котором каждый объект ссылается на два других объекта (бинарное дерево) и содержит четыре уровня в иерархической структуре.

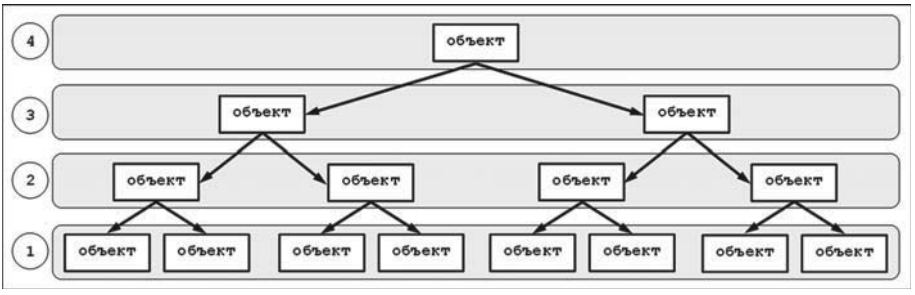

Рис. 6.18. Схематическое представление для бинарного дерева объектов. Цифрами обозначен ранг уровня в иерархии дерева. Стрелки обозначают ссылки на объекты

Каждый уровень в иерархии дерева будем обозначать *рангом*: самый высокий ранг имеет объект в вершине иерархии, объекты, на которые ссылается данный объект, имеют ранг на единицу меньше, и так далее.

ДЕТАЛИ

В общем случае, если в дереве m уровней иерархии и каждый объект ссылается на n других объектов, то объект в вершине иерархии имеет ранг m, на следующем уровне с рангом $m - 1$ находится n объектов, на следующем уровне (ранг уровня $m - 2$) всего имеется n^2 объектов, и так далее. На уровне с рангом k суммарно имеется n^{m-k} объектов. На последнем уровне с рангом 1 имеется n^{m-1} объектов. Если речь идет о бинарном дереве (значение $n = 2$) из четырех уровней (значение $m = 4$), то на последнем уровне будет $2^3 = 8$ объектов.

При реализации задачи по созданию дерева объектов мы будем исходить из следующего. Чтобы организовать в объекте ссылки на объекты более низкого ранга (объекты следующего уровня), в свойства объекта добавим массив, элементы которого собственно и будут объектами (точнее, «адресами» объектов), на которые ссылается данный объект.

ДЕТАЛИ

Здесь уместно вспомнить, что доступ к объектам осуществляется через ссылку. Поэтому если некоторой переменной значением присваивается объект, то на самом деле данная переменная содержит ссылку на объект. Массив объектов создается с учетом данного обстоятельства. Элементы такого массива являются ссылками на объекты. Сначала создается массив, а затем элементам данного массива в качестве значений присваиваются ссылки на объекты. Сами объекты предварительно должны быть созданы.

Помимо массива со ссылками на объекты (такой массив есть у каждого объекта, за исключением объектов нижнего уровня с единичным рангом), каждый объект имеет:

- свойство `rank`, определяющее ранг уровня, к которому принадлежит объект;
- свойство `number`, определяющее номер объекта;
- а также свойство `code`, определяющее некоторый код объекта, позволяющий однозначно идентифицировать объект в иерархии дерева.

НА ЗАМЕТКУ

Свойство `number`, как отмечено выше, определяет номер объекта. Это на самом деле порядковый номер элемента (на единицу больше индекса) в свойстве-массиве объекта, который содержит ссылку на данный объект. Таким образом, если каждый объект содержит ссылку на *n* других объектов, то номера объектов находятся в диапазоне от 1 до *n* включительно. В этом смысле номер объекта не является его однозначным идентификатором. Для идентификации используется текстовый код объекта (реализуется через свойство `code`). Код позволяет проследить иерархию ссылок на данный объект, начиная с объекта на верхнем уровне. Код состоит из отдельных блоков. Например, код `|4->1|` означает объект с номером `1` на наивысшем уровне ранга `4`. Код `|4->1|3->1|` означает объект, который находится на уров-

не ранга 3, а ссылка на него записана первым элементом в массиве со ссылками в объекте, находящемся на уровне ранга 4. Далее, код | 4 -> 1 | 3 -> 1 | 2 -> 2 | соответствует объекту, который находится на уровне с рангом 2. Ссылка на этот объект является вторым (по порядку) элементом в массиве, который «спрятан» в объекте на уровне ранга 3. На данный объект, в свою очередь, ссылка содержится в первом (по порядку) элементе в объекте на уровне ранга 4. Наконец, код | 4 -> 1 | 3 -> 1 | 2 -> 2 | 1 -> 2 | будет у объекта, который находится на уровне ранга 1. Ссылка на объект записана во второй элемент в массиве объекта, находящегося на уровне ранга 2. На него, в свою очередь, ссылается второй элемент в массиве объекта, находящегося на уровне ранга 3. А на этот объект ссылка содержится в первом элементе в массиве объекта, находящегося на уровне 4.

Рассмотрим и проанализируем программный код, представленный в листинге 6.8.

Листинг 6.8. Дерево объектов (файл Listing06_08.js)

```
// Конструктор для создания дерева объектов:
function Tree(rank,refs,number,code){
  // Ранг уровня, на котором находится объект:
  this.rank= rank
  // Номер объекта:
  this.number=number || 1
  // Формальный код объекта:
  this.code=code || "| "
  this.code+=this.rank+" -> "+this.number+" | "
  // Создание массива объектов:
  if(this.rank>1){
    // Массив:
    this.next=new Array(refs)
    for(var k=0;k<this.next.length;k++){
      // Создание объекта - элемента массива. Имеет
      // место рекурсивный вызов функции-конструктора:
      this.next[k]=new Tree(this.rank-1,refs,k+1,this.code)
    }
  }
}
```

```
// Переопределение метода toString() для прототипа
// конструктора дерева объектов:
Tree.prototype.toString=function(){
  // Локальная текстовая переменная:
  var t="Объект: ранг - "+this.rank+", номер - "+this.number
  t+=", код - "+this.code+"<br>"
  // Если объект не на последнем уровне:
  if(this.rank>1){
    for(var k=0;k<this.next.length;k++){
      // Рекурсивный вызов метода toString():
      t+=this.next[k].toString()
    }
  }
  // Результат метода:
  return t
}
// Создание дерева объектов:
var myTree=new Tree(4,2)
// Отображение содержимого дерева объектов:
document.write(myTree)
```

В сценарии описывается конструктор Tree, предназначенный для создания дерева объектов. У конструктора четыре аргумента: rank (ранг уровня, на котором находится объект), refs (количество объектов, на которые ссылается данный объект), number (номер объекта), code (код объекта). Командой this.rank=rank определяется значение для свойства rank объекта. Свойство number объекта определяется командой this.number=number || 1. Результат выражения number || 1 равен 1, если аргумент number не определен, и равен значению number, если при вызове конструктора аргумент number задан.

Значение свойства code определяется в несколько этапов. Сначала командой this.code=code || "| " задается начальное значение для свойства code. Если при вызове конструктора Tree() аргумент code задан, то именно он определяет начальное значение одноименного свойства объекта. Если аргумент code не задан, то начальным значением свойства code будет текст "| ". Далее к текущему значению свойства code командой

this.code+=this.rank+" -> "+this.number+" | " добавляется информация о значении ранга и номера объекта.

> **НА ЗАМЕТКУ**
>
> Стратегический план состоит в том, что при создании дерева объектов с помощью конструктора Tree ему будут передаваться только два первых аргумента (rank и refs). Два других аргумента (number и code) не передаются. Поэтому вариант вызова конструктора без двух аргументов соответствует созданию объекта верхнего уровня (объекта, который находится в вершине иерархии объектов). В теле конструктора вызывается этот же конструктор, но уже со всеми четырьмя аргументами. Поэтому вариант вызова конструктора со всеми аргументами соответствует созданию объекта на одном из внутренних уровней.

Далее в условном операторе проверяется условие this.rank>1 (ранг объекта больше единицы, а значит, объект не находится на самом нижнем уровне). Если условие истинно, то командой this.next=new Array(refs) создается массив next, являющийся одновременно свойством объекта. С помощью оператора цикла массив заполняется: каждый его элемент this.next[k] получает значением ссылку на объект. Объект создается командой new Tree(this.rank-1,refs,k+1,this.code). Речь идет о тех объектах, на которые ссылается текущий объект. Здесь мы в теле конструктора Tree вызываем этот же конструктор (то есть имеет место рекурсия). Аргументы при вызове конструктору передаются такие:

- ранг this.rank-1 на единицу меньше ранга текущего объекта;
- количество ссылок refs в новом объекте такое же, как и в исходном объекте;
- номер объекта k+1 на единицу больше индекса элемента в массиве, который ссылается на создаваемый объект;
- начальное this.code значение для кода создаваемого объекта такое же, как код текущего объекта.

Собственно, на этом описание конструктора Tree заканчивается.

> **НА ЗАМЕТКУ**
>
> Свойство-массив next есть у каждого объекта в дереве, за исключением объектов, находящихся на нижнем уровне с рангом 1.

Для прототипа конструктора Tree.prototype определяется метод toString(). В теле метода командой var t="Объект: ранг - "+this.rank+", номер - "+this.number определяется локальная текстовая переменная, начальное значение которой содержит информацию о ранге и номере объекта, для которого (явно или неявно) вызывается метод. Далее командой t+=", код - "+this.code+"
" в переменную добавляется информация о коде текущего объекта. После этого в условном операторе при истинности условия this.rank>1 запускается оператор цикла, в котором индексная переменная k пробегает значения индексов массива next текущего объекта. На каждом цикле выполняется команда t+=this.next[k].toString(). Командой в переменную t дописывается результат сведения к текстовому формату объектов из массива next. Здесь мы рекурсивно вызываем метод toString() для объектов из массива next. Конечное значение переменной t возвращается результатом метода toString().

ДЕТАЛИ

Чтобы понять принцип работы конструктора Tree или метода toString(), имеет смысл представить, что соответственно создается объект верхнего уровня и метод toString() вызывается из этого объекта. Начнем с конструктора.

В конструкторе сначала заполняются свойства создаваемого объекта, а затем создается массив. При заполнении массива объектами вызывается конструктор, который создает каждый из объектов второго уровня и, в свою очередь, вызывает конструктор для создания объектов третьего уровня. Все это продолжается до тех пор, пока не будут созданы объекты с единичным рангом.

Примерно по той же схеме действует метод toString(). Сначала формируется строка на основе свойств объекта верхнего уровня, а затем для каждого из объектов, на которые ссылается данный объект, вызывается метод toString(). При вызове метода снова происходит вызов toString(), и так вплоть до объектов единичного ранга. Получается своеобразная «цепная реакция». Каждый вызов метода добавляет в текстовую строку, возвращаемую результатом, информацию об очередном объекте.

Дерево объектов создается в сценарии командой var myTree=new Tree(4,2). Речь, очевидно, идет о бинарном дереве из четырех уровней. Увидеть содержимое созданной структуры можно с помощью команды document.write(myTree). Проверка сценария выполняется с помощью такого HTML-кода:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 6.8</title>
</head>
<body><h3>Листинг 6.8</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing06_08.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Текстовая версия для результата выполнения сценария представлена ниже.

Результат выполнения сценария (из листинга 6.8)

```
Объект: ранг - 4, номер - 1, код - | 4 -> 1 |
Объект: ранг - 3, номер - 1, код - | 4 -> 1 | 3 -> 1 |
Объект: ранг - 2, номер - 1, код - | 4 -> 1 | 3 -> 1 | 2 -> 1 |
Объект: ранг - 1, номер - 1, код - | 4 -> 1 | 3 -> 1 | 2 -> 1 | 1 -> 1 |
Объект: ранг - 1, номер - 2, код - | 4 -> 1 | 3 -> 1 | 2 -> 1 | 1 -> 2 |
Объект: ранг - 2, номер - 2, код - | 4 -> 1 | 3 -> 1 | 2 -> 2 |
Объект: ранг - 1, номер - 1, код - | 4 -> 1 | 3 -> 1 | 2 -> 2 | 1 -> 1 |
Объект: ранг - 1, номер - 2, код - | 4 -> 1 | 3 -> 1 | 2 -> 2 | 1 -> 2 |
Объект: ранг - 3, номер - 2, код - | 4 -> 1 | 3 -> 2 |
Объект: ранг - 2, номер - 1, код - | 4 -> 1 | 3 -> 2 | 2 -> 1 |
Объект: ранг - 1, номер - 1, код - | 4 -> 1 | 3 -> 2 | 2 -> 1 | 1 -> 1 |
Объект: ранг - 1, номер - 2, код - | 4 -> 1 | 3 -> 2 | 2 -> 1 | 1 -> 2 |
Объект: ранг - 2, номер - 2, код - | 4 -> 1 | 3 -> 2 | 2 -> 2 |
Объект: ранг - 1, номер - 1, код - | 4 -> 1 | 3 -> 2 | 2 -> 2 | 1 -> 1 |
Объект: ранг - 1, номер - 2, код - | 4 -> 1 | 3 -> 2 | 2 -> 2 | 1 -> 2 |
```

На рис. 6.19 представлен веб-документ со сценарием, открытый в окне браузера. Желающие могут проанализировать структуру кодов объектов, которые отображаются в рабочем документе, или проверить результат выполнения сценария для деревьев иной структуры (в команде var myTree=new Tree(4,2) достаточно изменить числовые значения, которые передаются в конструктор).

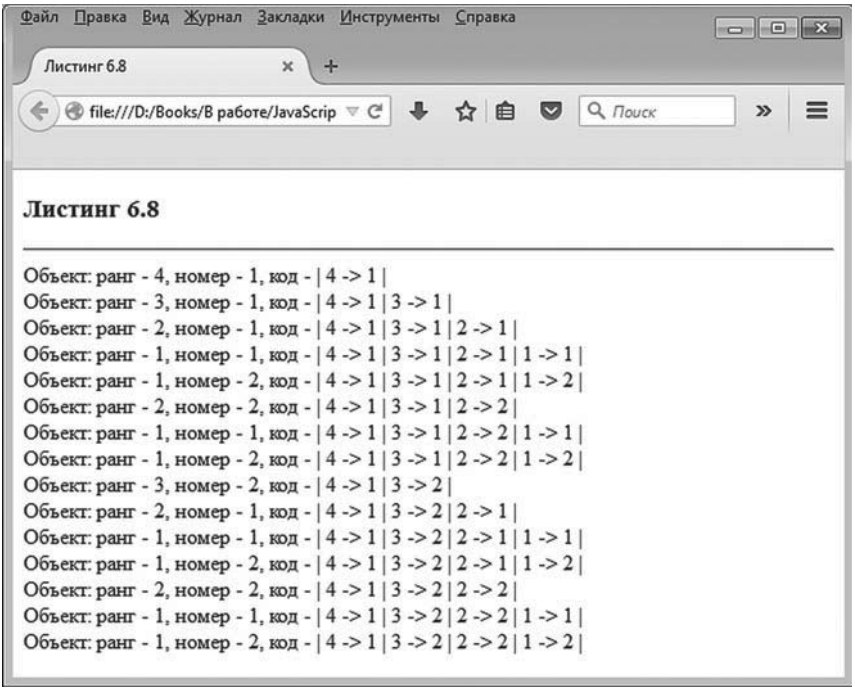

Рис. 6.19. Результат выполнения сценария, в котором создается дерево объектов

НА ЗАМЕТКУ

Дерево объектов, состоящее из *m* уровней при условии, что каждый объект ссылается на *n* объектов, содержит в общей сложности объекты в количестве $n^0 + n^1 + n^2 + \ldots n^{m-1} = \frac{n^m - 1}{n - 1}$.

Это число может быть очень большим. А для каждого объекта в дереве выводится строчка с информацией. Так что при экспериментах с деревьями гигантоманией увлекаться не стоит.

Можно проверить уровень своего понимания рассмотренного выше примера, заменить в сценарии команду document.write(myTree) на document.write(myTree.next[1]) или document.write(myTree.next[0].next[0]) и попытаться объяснить результат.

Функция как объект

> Этот дуэт двух наших форвардов сегодня ослепителен. От него можно ждать всяких неожиданностей.
>
> *из к/ф «Покровские ворота»*

Функции мы уже рассматривали в одной из глав. Здесь мы затронем некоторые вопросы, связанные в основном с тем, что в JavaScript функции являются объектами. Также мы рассмотрим ряд тем, которые в силу разных причин не были подняты ранее.

Итак, функция является объектом. Как у любого объекта, у функции есть свойства. Причем свойства очень полезные.

Для начала отметим один немаловажный момент: помимо тех способов создания функций, которые мы рассматривали (описание функции с ключевым словом function и присваивание переменной значением анонимной функции), существует еще один способ, подразумевающий использование конструктора Function. Данный метод создания функции далеко не самый популярный, однако вполне приемлемый. Формат создания функции с помощью конструктора Function выглядит так (жирным шрифтом выделена основная часть конструкции):

```
var имя_функции=new Function(аргументы, код функции)
```

Начальные аргументы конструктора Function — текстовые названия аргументов функции, а последний аргумент — текстовая строка с программным кодом, выполняемым при вызове функции. Скажем, если мы хотим объявить функцию, которая вычисляет сумму двух своих аргументов, то соответствующий фрагмент кода мог бы выглядеть следующим образом:

```
var f=new Function("x","y","return x+y")
```

После этого переменную f можем вызывать как функцию: например, значение выражения f(3,4) равно 7 (сумма аргументов).

Нас конструктор Function интересует не как средство создания функций, а по несколько иной причине. Причина эта связана с тем, что любая функция, каким бы способом (из трех означенных) она ни создавалась, имеет один и тот же прототип, и данный прототип — прототип Function.prototype конструктора Function. У прототипа Function.prototype име-

ется ряд интересных свойств и методов. А поскольку Function.prototype является прототипом и для пользовательских функций, фактически любая функция имеет доступ к означенным свойствам и методам. Их и обсудим далее.

Количество аргументов функции

Начнем со свойства length, которое хотя и не критично, но может быть достаточно полезным на практике. Свойство length значением возвращает количество аргументов, указанных при объявлении функции. Если после названия функции через точку указать свойство length, то результатом будет количество аргументов, указанных *при описании* этой функции.

> **НА ЗАМЕТКУ**
>
> Стоит напомнить, что количество аргументов, передаваемых функции при вызове, может не совпадать с количеством аргументов, указанных при описании функции.

Небольшой пример использования свойства length для функций представлен в листинге 6.9.

Листинг 6.9. Свойство length для функций (файл Listing06_09.js)

```
// Функция для проверки количества аргументов,
// указанных при описании функции:
function testArgs(func){
  // Количество аргументов функции:
  var n=func.length
  var words
  // Определение значения текстовой переменной:
  switch(n){
    case 0:
      words="нет аргументов."
      break
    case 1:
      words="один аргумент."
```

```
      break
    case 2:
    case 3:
    case 4:
      words=n+" аргумента."
      break
    default:
      words=n+" аргументов."
  }
  document.write("Функция "+func+": "+words+"<br>")
}
// Вспомогательная функция без аргументов:
function show(){
  document.write("Всем привет!")
}
// Вспомогательная функция с шестью аргументами:
function F(a,b,c,d,e,f){
  return a*b*c*d*e*f
}
// Проверка количества аргументов функции:
testArgs(eval)
testArgs(Math.pow)
testArgs(show)
testArgs(F)
// Функция возвращает результатом ту из функций,
// у которой при описании указано меньше аргументов:
function getFunc(f1,f2){
  if(f1.length<=f2.length){
    return f1
  }
  else{
    return f2
  }
}
```

```
// Примеры вызова функции getFunc():
var x=getFunc(Math.pow,F)(2,3)
document.write(x+"<br>")
getFunc(show,eval)()
```

Представленный ниже HTML-код поможет проверить работу сценария:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 6.9</title>
</head>
<body><h3>Листинг 6.9</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing06_09.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Далее показано, как выглядит результат выполнения сценария.

Результат выполнения сценария (из листинга 6.9)

```
Функция function eval() { [native code] }: один аргумент.
Функция function pow() { [native code] }: 2 аргумента.
Функция function show(){ document.write("Всем привет!") }: нет аргументов.
Функция function F(a,b,c,d,e,f){ return a*b*c*d*e*f }: 6 аргументов.
8
Всем привет!
```

Веб-документ со сценарием, открытый в окне браузера, показан на рис. 6.20.

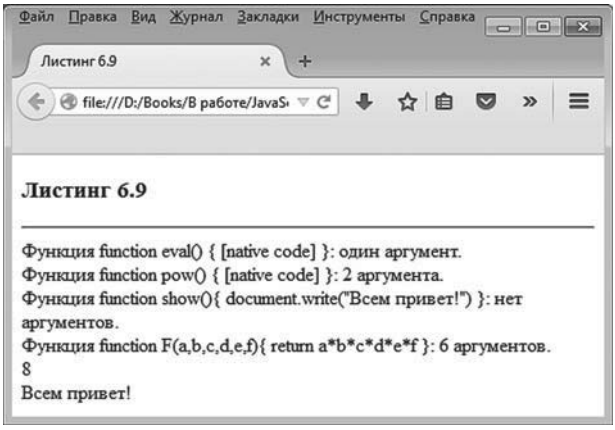

Рис. 6.20. Результат выполнения сценария, в котором используются свойство *length* для функций

В сценарии описывается функция testArgs(). При ее вызове отображается информация о количестве аргументов в описании другой функции, переданной аргументом функции testArgs() (этот аргумент обозначен как func). В теле функции вычисляется значение func.length, и его результат записывается в переменную n. Далее с помощью оператора выбора switch в зависимости от фактического значения переменной n формируется текстовая строка, которая и отображается в рабочем окне.

Для проверки функция testArgs() вызывается с передачей ей аргументом названия встроенной функции eval(), метода Math.pow() и описанных в сценарии вспомогательных функций show() и F(). Как несложно заметить, количество аргументов (указанных при описании функций) определяется корректно.

НА ЗАМЕТКУ

Стоит обратить внимание, как методом write() отображается название функции: фактически отображается не только название функции, но и другие ее вспомогательные атрибуты. За эту «работу» отвечает метод toString() который вызывается автоматически для преобразования «названия функции» (по факту объекта функции) к текстовому формату.

Также в сценарии описывается функция getFunc() с двумя аргументами. Предполагается, что эти аргументы являются функциями (названия функций, а на самом деле — объекты функций). Результатом возвра-

щается та из функций, у которой меньше аргументов (при равенстве аргументов возвращается первая из функций-аргументов). Поэтому, например, результатом выражения getFunc(Math.pow,F) возвращается метод Math.pow, и команда getFunc(Math.pow,F)(2,3) эквивалентна (в данном конкретном случае) выражению Math.pow(2,3).

> **НА ЗАМЕТКУ**
>
> Передача методов аргументами функциям имеет некоторые особенности, связанные с передачей контекста — ссылки на объект, из которого вызывается метод. Данный вопрос обсудим немного позже.

Аналогично, инструкция getFunc(show,eval)() эквивалента вызову show().

Функция с произвольным количеством аргументов

Как отмечалось выше, количество аргументов, которые фактически передаются функции при вызове, может не совпадать с количеством аргументов, указанных при ее описании. Другими словами, свойство length не позволяет определить, сколько аргументов передано функции при вызове. Зато в теле функции доступен объект arguments, который сопоставим с локальной переменной в теле функции и представляет собой аналог массива со значениями аргументов, переданных функции при вызове.

> **НА ЗАМЕТКУ**
>
> Объект arguments на самом деле не является массивом. Вместе с тем его можно индексировать как массив, и у него есть свойство length, определяющее количество элементов в объекте arguments.
> Объект arguments не нужно объявлять в теле функции. Он доступен автоматически.

С помощью объекта arguments можно тестировать фактически переданные аргументы функции. Одно из возможных направлений использования данного объекта — создание функций с произвольным количеством аргументов. Небольшой пример, в котором описывается функция, предназначенная для вычисления суммы аргументов, переданных функции, приведен в листинге 6.10.

Листинг 6.10. Функция с произвольным количеством аргументов (файл Listing06_10.js)

```
// Функция для вычисления суммы аргументов:
function sum(){
  // Локальная переменная для вычисления суммы:
  var s
  // Начальное значение локальной переменной:
  if((arguments.length>0)&&(typeof(arguments[0])=="string")){
    s=""
  }
  else{
    s=0
  }
  // Вычисление суммы:
  for(var k=0;k<arguments.length;k++){
    s+=arguments[k]
  }
  // Результат:
  return s
}
// Примеры вызова функции:
document.write("Сумма чисел: "+sum(1,4,2,7,3)+"<br>")
document.write("Сумма чисел: "+sum(1,2,3)+"<br>")
document.write("Сумма чисел: "+sum()+"<br>")
document.write("Сумма слов: "+sum("один"," четыре"," два"," семь"," три"))
```

Для проверки сценария используем такой HTML-код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 6.10</title>
</head>
<body><h3>Листинг 6.10</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing06_010.js">
```

```
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Текстовая версия результата выполнения сценария представлена ниже.

Результат выполнения сценария (из листинга 6.10)

```
Сумма чисел: 17
Сумма чисел: 6
Сумма чисел: 0
Сумма слов: один четыре два семь три
```

На рис. 6.21 показано, как выглядит веб-документ со сценарием, когда документ открывается в окне браузера.

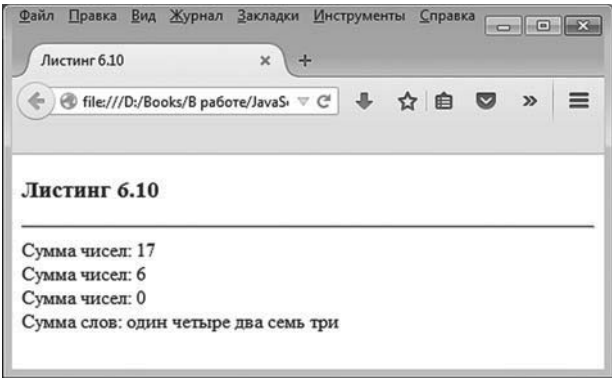

Рис. 6.10. Результат выполнения сценария, в котором описана функция, допускающая передачу произвольного количества аргументов

В сценарии описывается функция sum(), у которой формально нет аргументов. Однако мы предполагаем, что при вызове функции аргументы могут быть и результатом функции должна возвращаться их сумма. Причем мы ставим перед собой задачу написать такую функцию, чтобы она могла не только вычислять сумму числовых аргументом, но еще и объединять (формально суммировать) текстовые аргументы. Поэтому в теле функции объявляется локальная переменная s,

начальное значение которой определятся в зависимости от типа первого переданного функции аргумента. Для этого используется условный оператор, в котором проверяется условие (arguments.length>0)&&(typeof (arguments[0])=="string"). Условие истинно, если функции передан хотя бы один аргумент (условие arguments.length>0) и одновременно первый из переданных аргументов текстовый (условие typeof(arguments [0])=="string"). Если условие в условном операторе истинно, переменная s получает в качестве начального значения пустую текстовую строку. В противном случае начальное значение переменной нулевое.

ДЕТАЛИ

При проверке условия (arguments.length>0)&&(typeof(arguments[0])=="string") сначала вычисляется значение выражения arguments.length>0. Если данное выражение равно true, то как минимум один аргумент функции передан и имеет смысл обращение arguments[0] к первому аргументу функции. Если значение выражения arguments.length>0 равно false, то аргументы функции не передавались. В таком случае обращение arguments[0] некорректно. Вместе с тем, поскольку оператор *логического и* && вычисляется по сокращенной схеме, то при значении false первого операнда второй не вычисляется. Поэтому при значении false выражения arguments.length>0 до вычисления значения выражения typeof(arguments[0])=="string" дело не доходит.

Далее в теле функции запускается оператор цикла, в котором в переменную s последовательно добавляются значения элементов (при индексе k обращение к аргументу функции выполняется в формате arguments[k]). Помимо описания функции sum(), сценарий содержит примеры вызова функции. Как несложно заметить, при разном количестве аргументов результатом функции возвращается их сумма, в чем и состояла наша цель.

Передача контекста функции

Чтобы понять суть проблемы, которая обсуждается далее, сразу рассмотрим сценарий, представленный в листинге 6.11.

Листинг 6.1. Использование ключевого слова this в описании функции

```
// Функция с указателем this описана вне объекта:
function f(text,number){
```

```
  this.text=text
  this.number=number
}
// Объекты:
var A={}
var B={show:function(){
  for(var s in this){
    document.write(s+": "+this[s]+"<br>")
  }
}}
var C={}
C.method=f
// Вызов функции:
f("функция",100)
// Вызов из объекта функции метода call():
f.call(A,"объект A",200)
// Вызов из объекта функции метода apply():
f.apply(B,["объект B",300])
// Вызов метода объекта:
C.method("объект C",400)
// Проверка значений глобальных
// переменных и свойств объектов:
document.write(text+" | "+number+"<br>")
document.write(A.text+" | "+A.number+"<br>")
document.write(B.text+" | "+B.number+"<br>")
document.write(C.text+" | "+C.number+"<br>")
// Проверка наличия у объектов метода f():
document.write(("f" in A)+"<br>")
document.write(("f" in B)+"<br>")
document.write(("f" in C)+"<br>")
// Проверка свойств и методов объектов:
document.write("<b>Объект B:</b><br>")
B.show()
document.write("<b>Объект A:</b><br>")
```

```
B.show.call(A)
document.write("<b>Объект C:</b><br>")
B["show"].call(C)
```

Работу сценария проверим с помощью такого HTML-кода:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 6.11</title>
</head>
<body><h3>Листинг 6.11</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing06_11.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

На рис. 6.22 показано, как в окне браузера выглядит веб-документ с представленным выше сценарием.

Проанализируем код сценария и поясним результаты его выполнения. При этом полезной будет текстовая версия результата выполнения сценария, представленная ниже.

Результат выполнения сценария (из листинга 6.11)

```
функция | 100
объект A | 200
объект B | 300
объект C | 400
false
false
false
Объект B:
```

```
show: function (){ for(var s in this){ document.write(s+": "+this[s]+"
") } }
text: объект B
number: 300
Объект A:
text: объект A
number: 200
Объект C:
method: function f(text,number){ this.text=text this.number=number }
text: объект C
number: 400
```

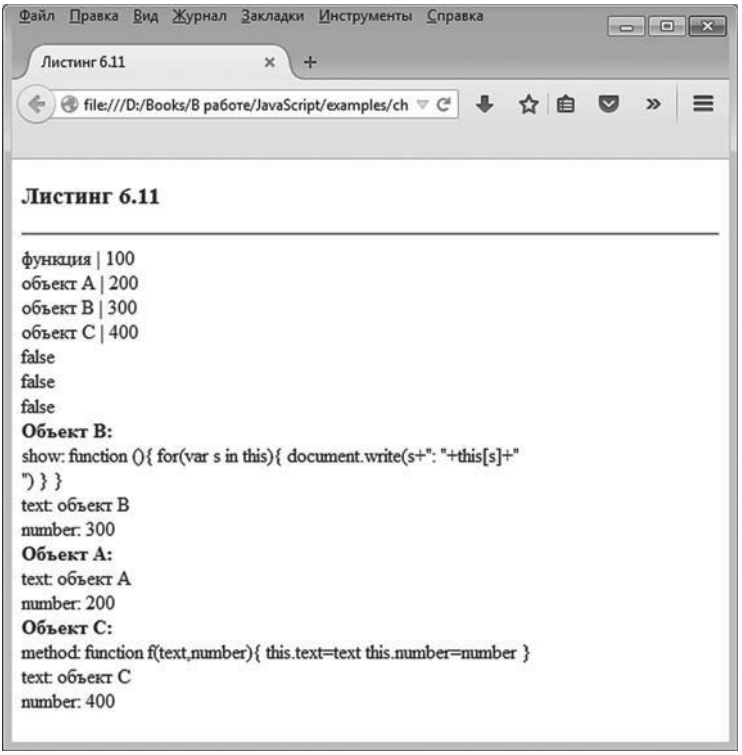

Рис. 6.22. Результат выполнения сценария, в котором используется функция, описанная с ключевым словом *this*, а также методы *call()* и *apply()*

Вся интрига в сценарии закручена вокруг функции f(), которая описана с двумя аргументами (обозначены как text и number), а в теле данной

функции использовано ключевое слово this. В частности, командами this.text=text и this.number=number выполняется присваивание значений свойствам text и number, только непонятно какого объекта, поскольку функция описана вне какого бы то ни было из них.

Другими словами, поскольку функция f() определена сама по себе, то возникает вопрос: что подразумевать под this? Ответ на вопрос зависит от способа вызова функции. Если функция вызывается как обычная функция (не метод), то под this подразумевается объект рабочего окна window. Поэтому, например, при выполнении команды f("функция",100) будут созданы две глобальные переменные text и number, и им будут присвоены соответствующие значения. В последнем легко удостовериться с помощью команды document.write(text+" | "+number+"
").

НА ЗАМЕТКУ

Напомним, что глобальная переменная представляет собой свойство объекта рабочего окна window.

В сценарии создается три объекта: A, B и C. Объект A пустой, объект B содержит метод show(), которым отображаются названия и значения всех собственных свойств и методов объекта, а объекту C командой C.method=f добавляется свойство method, значением которого является ссылка на функцию f().

Далее в сценарии используются встроенные методы call() и apply(), которые вызываются из объекта функции или метода. Вообще методы call() и apply() позволяют вызывать функцию или метод в контексте некоторого объекта. Принципиальное различие между методами call() и apply() состоит в способе передачи аргументов: методу apply() они передаются в виде массива (за исключением первой ссылки на объект), а методу call() аргументы передаются списком. Более конкретно формат вызова метода call() таков:

```
функция.call(объект,аргументы)
```

Результат команды такой, как если бы функция (из объекта которой вызывается метод call()) вызывалась из объекта (первый аргумент метода call()) с аргументами (все прочие, кроме первого, аргументы метода call()).

Аналогично используется метод apply():

```
функция.apply(объект,массив)
```

Результатом такого выражения является результат вызова функции (из объекта которой вызывается метод apply()) из объекта (первый аргумент метода apply()) с аргументами, переданными в виде массива (второй аргумент метода apply()).

Поэтому результат выполнения команды f.call(A,"объект A",200) такой, как если бы у объекта A был метод f() и он вызывался с аргументами "объект A" и 200. Аналогично, результат выполнения команды f.apply(B,["объект B",300]) эквивалентен вызову из объекта B метода, аналогичного f(), с аргументами "объект B" и 300. Но здесь важно понимать, что на самом деле метода f() ни у объекта A, ни у объекта B нет, хотя при выполнении соответствующих команд под ключевым словом this имеется в виду ссылка на объект, в контексте которого вызывается функция (объект A для первой команды и объект B для второй команды). А вот у объекта C есть метод method(), который можно вызвать (команда C.method("объект C",400)), причем в данном случае реально выполняется код функции f(), а под ключевым словом this подразумевается объект C.

С помощью команд document.write(A.text+" | "+A.number+"
"), document.write(B.text+" | "+B.number+"
") и document.write(C.text+" | "+C.number+"
") легко убедиться, что у объектов A, B и C появляются свойства text и number. А вот проверка на предмет наличия у указанных объектов метода f() (команды document.write(("f" in A)+"
"), document.write(("f" in B)+"
") и document.write(("f" in C)+"
") показывает, что такого метода у данных объектов (в том числе и у объекта C!) нет.

НА ЗАМЕТКУ

У объекта C метода f() нет, зато есть метод method(), при вызове которого на самом деле выполняется код функции f().

Для проверки свойств и методов объекта B вызываем из объекта метод show().

НА ЗАМЕТКУ

При отображении «значения» метода в рабочий документ выводится программный код метода. Код метода show() содержит в текстовых литералах дескрипторы перехода к новой строке
. Поэтому в соответствующих местах текста кода в окне браузера выполняется переход к новой строке.

Метод show() есть у объекта B, но его нет у объектов A и C. Тем не менее метод show() объекта B можно вызвать в контексте объекта A. Другими словами, несмотря на то что show() является методом объекта B, его можно вызвать так, как если бы он был методом объекта A. Нужный эффект достигается, например, командой B.show.call(A). В данном случае из объекта метода B.show вызывается метод call(), которому передается аргументом объект A, в контексте которого следует вызывать метод. Если бы у метода show() были аргументы, они бы перечислялись после ссылки на объект A в списке аргументом метода call().

Команда B["show"].call(C) по своему назначению аналогична предыдущей, с той лишь разницей, что при обращении к методу show() объекта B мы используем не точечный синтаксис, а индексную нотацию, указав название метода в квадратных скобках после имени объекта.

Ситуация с передачей контекста при вызове функций и методов может быть достаточно нетривиальной. В качестве иллюстрации рассмотрим еще один пример. Он представлен в листинге 6.12.

Листинг 6.12. Передача контекста с помощью метода bind() (файл Listing06_12.js)

```
// Функция результатом возвращает функцию,
// переданную аргументом:
function caller(func){
  return func
}
// Функция отображает значение переданного ей аргумента:
function show(txt){
  document.write(txt+"<br>")
}
// Объект со свойством text, методом hi()
// и переопределенным методом toString():
var obj={text:"объект obj",hi:function(){
    document.write(this.text+"<br>")
  },
  toString:function(){
    return this.text
  }
```

```
}
// Вызов функций show() через функцию caller():
caller(show)("функция show()")
obj.hi() // Вызов метода объекта
// Неудачная попытка вызвать
// метод hi() через функцию caller():
caller(obj.hi)()
// Вызов метода hi() через функцию caller():
caller(obj.hi.bind(obj))()
// Определение новой функции на основе метода:
var powerOfTwo=Math.pow.bind(Math,2)
var n=5
document.write("2<sup>"+n+"</sup> = "+powerOfTwo(n)+"<br>")
// Вспомогательная функция с ключевым словом this в теле:
function f(x){
  return this+x
}
// Определение новых функций на основе
// вспомогательной функции:
var one=f.bind(10)
var two=f.bind(obj)
// Вызов функций:
document.write(one(5)+"<br>")
document.write(two(" - это он")+"<br>")
// Новое значение свойства text объекта obj:
obj.text="тот же объект"
// Вызов функции:
document.write(two(" - новое значение")+"<br>")
```

Тестирование сценария выполняется посредством следующего HTML-кода:

```
<!DOCTYPE HTML>
<html>
<head>
```

```
  <title>Листинг 6.12</title>
</head>
<body><h3>Листинг 6.12</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing06_12.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Результат выполнения сценария представлен на рис. 6.23.

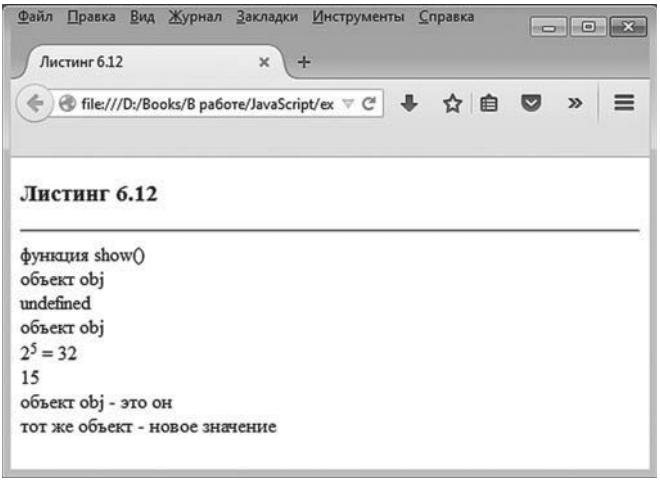

Рис. 6.23. Результат выполнения сценария, в котором передача контекста выполняется с помощью метода *bind()*

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Для отображения в рабочем документе верхнего индекса (операция возведения в степень) использованы дескрипторы ^и.

В сценарии описывается функция caller() с одним аргументом, который она и возвращает результатом функции. Мы предполагаем, что аргументом функции caller() будет передаваться другая функция. Та-

ким образом, вызов функции caller() означает вызов той функции, имя которой передано аргументом (во всяком случае, в таком режиме мы планируем использовать функцию caller()).

Функция show() вспомогательная. У нее один аргумент, значение которого отображается при вызове функции. Также в сценарии создается объект obj со свойством text, методом hi() (методом отображается значение свойства text), и еще для объекта явно описан метод toString() (описан так, что результатом приведения объекта к текстовому формату является значение свойства text).

При вызове функций show() через функцию caller() с помощью команды caller(show)("функция show()") получаем ожидаемый результат — в рабочем документе отображается текст "функция show()". Здесь все понятно: при выполнении команды caller(show)("функция show()") на самом деле вызывается функция show() с аргументом "функция show()". Вполне ожидаемый результат и при выполнении команды obj.hi() — методом hi() при вызове из объекта obj отображается текст "объект A".

Неожиданностью, скорее всего, будет результат выполнения команды caller(obj.hi)(). Можно было бы ожидать, что в данном случае вызывается метод hi() объекта obj, но в рабочем документе появляется текст "undefined". Причина в том, что передача функции caller() аргументом объекта obj.hi не означает передачу контекста вызова метода hi(). Проще говоря, когда мы передаем аргументом конструкцию obj.hi, то из формата данной инструкции понятно, о каком методе идет речь, но при вызове метода параметр this не определен. Исправить ситуацию позволяет метод bind(), который вызывается из объекта функции или метода. Первым аргументом методу bind передается ссылка, которая определяет параметр this в теле метода. Если имеются и другие аргументы, они также указываются аргументом метода bind(). Например, инструкция obj.hi.bind(obj) означает ссылку на метод hi() объекта, который при вызове получает в качестве this ссылку на объект obj. Поэтому команда caller(obj.hi.bind(obj))() означает не что иное, как вызов из объекта obj метода hi() через функцию caller(). Результатом является текст "объект obj" в рабочем документе.

Еще один пример использования метода bind() реализуется командой var powerOfTwo=Math.pow.bind(Math,2). Здесь переменной powerOfTwo значением присваивается результат выражения Math.pow.bind(Math,2). Сразу отметим, что это функция и поэтому переменную powerOfTwo следует интерпретировать как имя функции. Вопрос только в том, какой

функции? Метод bind() вызывается из объекта Math.pow. Метод pow() встроенного объекта Math вызывается с двумя аргументами и результатом возвращает значение первого аргумента в степени, определяемой вторым аргументом. Аргументы Math и 2 метода bind() означают, что в метод pow() следует в качестве ссылки this передать объект Math, а первым аргументом нужно передать значение 2. Недостающий второй аргумент для метода pow() передается при вызове функции, записанной в переменную powerOfTwo. Другими словами, выражением Math.pow.bind(Math,2) определяется функция от одного аргумента. Вызов функции с определенным аргументом означает вызов метода pow() из объекта Math с первым аргументом, равным 2, и вторым аргументом — собственно аргументом функции. Более конкретно, выражение powerOfTwo(n) означает вычисление значения 2 в степени n (для справки заметим, что $2^5 = 32$).

НА ЗАМЕТКУ

В одном из предыдущих примеров мы передавали аргументом функции ссылку на метод pow() встроенного объекта Math. Собственно, там проблемы с контекстом вызова метода не наблюдалось. Дело в том, что, хотя формально метод pow() вызывается из объекта Math, на самом деле при вычислении результата ссылка на сам объект не используется. Поэтому результат метода не зависит от того, «наполнена ли реальным смыслом» ссылка this.

Наконец, в сценарии определяется функция f() с аргументом x, которая результатом возвращает выражение this+x. Мы используем данную функцию для определения (с привлечением метода bind()) двух новых функций. Функции определяются командами var one=f.bind(10) и var two=f.bind(obj). Выражение f.bind(10) определяет такую функцию: при выполнении кода функции f() в качестве ссылки this передается значение 10. Таким образом, функцией one() для некоторого аргумента x возвращается значение 10+x. В этом смысле значение выражения one(5) равно 15.

Выражение f.bind(obj) определяет такую функцию: выполняется код функции f(), в котором ссылка this означает объект obj. Для аргумента x функцией two() возвращается результат obj+x. Поскольку для объекта obj определен метод toString(), которым при приведении объекта к текстовому формату возвращается значение свойства text объекта, то фактически вычисляется значение text+x. Если аргумент x текстовый — то выполняется конкатенация текстовых значений.

НА ЗАМЕТКУ

Операция типа прибавления к объекту значения вообще обрабатывается методом valueOf(). При этом данный метод должен возвращать значение простого (базового) типа. Если не так, то автоматически вызывается метод toString(). Здесь именно такой случай.

При вызове функции two() она «считывает» значение свойства text объекта obj. Легко заметить, что после изменения значения свойства text объекта obj изменяется и результат вызова функции two().

Еще один небольшой пример, который рассмотрим далее, иллюстрирует некоторые особенности работы с массивами, элементы которых являются функциями или методами. Рассмотрим программный код в листинге 6.13.

Листинг 6.13. Функции и методы как элементы массива (файл Listing06_13.js)

```
// Функция с одним аргументом:
function zero(txt){
  document.write("<b>"+txt+"</b><hr>")
}
// Метод toString() для объекта функции zero():
zero.toString=function(){
  var t="Название функции - zero<br>"
  t+="Количество аргументов - "+this.length+"<br>"
  return t
}
// Метод с одним аргументом:
function one(txt){
  document.write("<b>"+txt+"</b><br>")
  for(var s in this){
    document.write(this[s]+"<br>")
  }
  document.write("<hr>")
}
// Метод toString() для объекта метода one():
```

```
one.toString=function(){
  var t="Метод one()<br>"
  t+="Количество аргументов - "+this.length+"<br>"
  return t
}
// Пустой массив:
var A=[]
// Первый (начальный) элемент массива:
A[0]=zero
// Второй (с индексом один) элемент массива:
A[1]=one
// Вызов функции (первый элемент массива):
A[0]("Начальный элемент массива")
// Вызов метода (второй элемент массива):
A[1]("Элемент с единичным индексом")
```

Работа программного кода проверяется с помощью такого HTML-кода:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 6.13</title>
</head>
<body><h3>Листинг 6.13</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing06_13.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Как выглядит окно браузера с открытым в нем веб-документом, содержащим данный сценарий, показано на рис. 6.24.

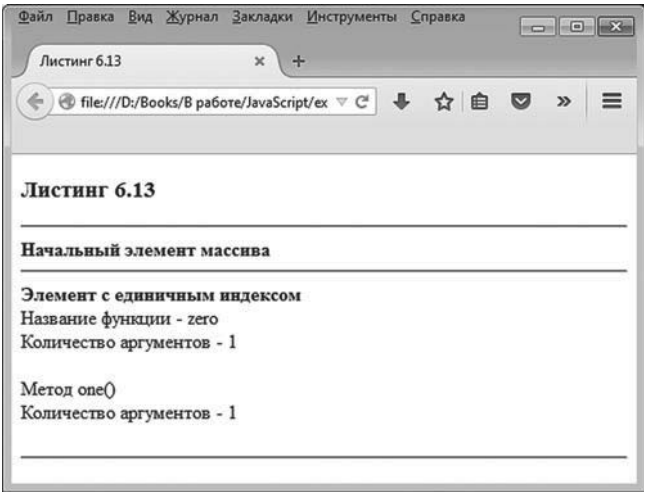

Рис. 6.24. Результат выполнения сценария, создающего массив, элементы которого являются функциями

В сценарии описывается функция zero() с одним аргументом и метод one() тоже с одним аргументом. И для функции, и для метода определяется метод toString(). Благодаря этому попытка «отобразить» название метода в рабочем документе имеет более приемлемые последствия: вместо отображения кода функции/метода выводится информация о названии функции/метода и количестве аргументов (то количество аргументов, что указано при описании функции/метода). Метод one() принципиально отличается от функции zero() тем, что содержит в своем коде ключевое слово this.

ДЕТАЛИ

Описание метода toString() как для функции zero(), так и для метода one() содержит ключевое слово this. Но в теле метода toString() данная ссылка означает объект, для которого вызывается функция или метод, — то есть объект функции и объект метода соответственно. Ключевое слово this в теле метода toString(), определяемого для объекта метода one(), — далеко не то же самое, что ключевое слово this в теле метода one(). Это разные ссылки, хотя и называются одинаково.

В сценарии создается массив A (сначала пустой), и затем первому его элементу значением присваивается функция zero() (команда A[0]=zero), а второму элементу массива значением присваивается метод one() (команда A[1]=one). После этого команда A[0]("Начальный элемент массива") оз-

начает вызов функции zero() с соответствующим аргументом, а команда A[1]("Элемент с единичным индексом") означает вызов метода one(). Причем несложно заметить, что объектом, из которого вызывается метод, является весь массив. Проще говоря, в метод one() при таком вызове в качестве this передается ссылка на объект массива A.

Встроенные объекты

— Ты такую машину сделал?

из к/ф «Иван Васильевич меняет профессию»

Мы уже знаем, что в языке JavaScript имеются встроенные объекты. С некоторыми из них мы даже работали. Далее приводится очень краткий обзор основных встроенных объектов, которые наиболее интересны с точки зрения их практического использования.

Объект Math

Со встроенным объектом Math мы сталкивались, когда вызывали методы этого объекта при решении небольших математических задач. Здесь мы расширим наши познания в плане математических возможностей языка JavaScript, реализуемых через объект Math. Объект Math имеет ряд полезных свойств, значения которых соответствуют наиболее часто используемым математическим константам и выражениям. Собственные свойства объекта Math представлены в табл. 6.1.

Таблица 6.1. Свойства объекта Math

Свойство	Описание
E	Значение константы Эйлера $e \approx 2{,}718281828459045$
LN10	Значение натурального логарифма от 10: $ln(10) \approx 2{,}302585092994046$
LN2	Значение натурального логарифма от 2: $ln(2) \approx 0{,}6931471805599453$
LOG10E	Значение десятичного логарифма (то есть логарифма по основанию 10) от константы Эйлера e: $lg(e) \approx 0{,}4342944819032518$
LOG2E	Значение двоичного логарифма (то есть логарифма по основанию 2) от константы Эйлера e: $log_2(e) \approx 1{,}4426950408889634$
PI	Значение иррационального числа $\pi \approx 3{,}141592653589793$
SQRT1_2	Значение выражения, равного единице, деленной на корень квадратный из 2: $\frac{1}{\sqrt{2}} \approx 1{,}4142135623730951$
SQRT2	Значение корня квадратного из 2: $\sqrt{2} \approx 1{,}4142135623730951$

Также у объекта Math имеется группа методов, каждый из которых определяет некоторую математическую функцию. Методы объекта Math для вычисления значения основных математических функций перечислены в табл. 6.2.

Таблица 6.2. Методы объекта Math

Метод	Описание
abs()	Метод для вычисления модуля числа
acos()	Метод для вычисления арккосинуса числа
asin()	Метод для вычисления арксинуса числа
atan()	Метод для вычисления арктангенса
atan2()	Методом возвращается угол (в радианах) на точку на плоскости, координаты которой определяются аргументами метода. Так, если метод вызывается в формате Math.atan2(y,x), то результатом возвращается угол на точку с координатами x и y
ceil()	Метод для округления действительного значения до целого числа. Результатом возвращается наименьшее возможное целое число, которое больше или равно значению аргумента метода (округление вверх)
cos()	Метод для вычисления косинуса
exp()	Метод для вычисления экспоненты
floor()	Метод предназначен для округления действительного значения до целого числа. Результатом возвращается набольшее возможное целое число, которое меньше или равно значению аргумента метода (округление вниз)
log()	Метод для вычисления натурального логарифма
max()	Методом возвращается значение наибольшего из чисел, переданных аргументами методу
min()	Результатом метода возвращается наименьшее из чисел, переданных аргументами методу
pow()	Метод для вычисления степени числа. Результатом возвращается значение, равное значению первого аргумента, возведенного в степень, определяемую вторым аргументом
random()	Метод для генерирования случайного числа в диапазоне от 0 до 1
round()	Метод предназначен для округления действительных чисел до целочисленных значений. Округление выполняется до ближайшего целого числа
sin()	Метод для вычисления синуса
sqrt()	Метод для вычисления квадратного корня
tan()	Метод для вычисления тангенса

ДЕТАЛИ

В зависимости от типа используемого браузера в объекте Math могут поддерживаться и некоторые другие методы. Так, браузером Mozilla Firefox поддерживаются методы sinh() (синус гиперболический), cosh() (косинус гиперболический), tanh() (тангенс гиперболичес-

кий) и ряд других. Однако следует помнить, что есть браузеры, которые данные методы не поддерживают.

Математические методы объекта Math «перекрывают» основные математические операции. Но список, разумеется, не является исчерпывающим. В случае необходимости обычно не составляет труда описать собственную пользовательскую математическую функцию.

Объект Number

Встроенный объект Number предназначен для работы с числовыми значениями и позволяет выполнять некоторые полезные операции, связанные с числовыми данными. Объект используется не часто, но эффективно.

Объект Number является конструктором — с помощью инструкции вида new Number(*аргумент*) можно создать числовой объект (хотя, как отмечалось выше, потребность в этом возникает не часто). Аргументом конструктору обычно передается число или текстовое представление числа (текстовая строка, содержащая число). Если аргументом конструктору передается некорректное значение (значение, которое не может быть преобразовано в число), то результатом возвращается значение NaN (сокращение от *Not A Number* — не число).

ДЕТАЛИ

Числа можно указывать в экспоненциальной нотации. В таком случае число представляется в виде *мантиссы* (обычно число большее или равное 1 и меньшее 10), умноженной на 10 в некоторой целочисленной степени. Например, число 123,5 в экспоненциальной нотации представляется как $1{,}235 \times 10^{2}$, а число 0,0123 записывается в виде $1{,}23 \times 10^{-2}$. В программном коде используется представление чисел с символом E (или e): после мантиссы указывается буква E (или e) и затем показатель степени (положительное значение для показателя степени указывается со знаком «плюс» или без знака). Например, число $1{,}235 \times 10^{2}$ записывается в виде 1.235E2, а число $1{,}23 \times 10^{-2}$ записывается как 1.23E-2.

Среди свойств объекта Number можно выделить MAX_VALUE, определяющее максимальное доступное числовое значение (значение

1.7976931348623157e+308), а также MIN_VALUE, определяющее минимально возможное (по модулю), но отличное от нуля число (значение 5e-324).

Методы, которые упоминаются далее, наследуются из прототипа Number.prototype. Наибольший интерес представляют методы toExponential(), toFixed() и toPrecision(). Далее приводится краткое описание данных методов, а затем — небольшой пример их использования.

- Метод toExponential()предназначен для преобразования числа к формату в экспоненциальной нотации. Аргументом методу передается число, определяющее количество цифр после десятичной точки в мантиссе числа.
- Метод toFixed() предназначен для приведения числа к формату в обычном (не экспоненциальном) представлении. Аргументом методу передается число, определяющее количество цифр после десятичной точки в представлении числа.
- Методом toPrecision() возвращается текстовая строка, содержащая представление числа с указанной точностью. Количество цифр в представлении числа (всех, а не только после десятичной точки) указывается аргументом метода.

Небольшой пример с использованием упомянутых выше методов приведен в листинге 6.14.

Листинг 6.14. Использование методов объекта Number (файл Listing06_14.js)

```
var x=new Number(12.3478)
document.write(x.toExponential(10)+"<br>")
document.write(x.toExponential(2)+"<br>")
document.write(x.toFixed(10)+"<br>")
document.write(x.toFixed(2)+"<br>")
document.write(x.toPrecision(5)+"<br>")
document.write(x.toPrecision(2)+"<br>")
```

Для тестирования кода используем следующий HTML-код:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 6.14</title>
```

```
</head>
<body><h3>Листинг 6.14</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing06_14.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Текстовая версия результата выполнения сценария представлена ниже.

Результат выполнения сценария (из листинга 6.14)

```
1.2347800000e+1
1.23e+1
12.3478000000
12.35
12.348
12
```

Как выглядит окно браузера с открытым в нем веб-документом со сценарием, показано на рис. 6.25.

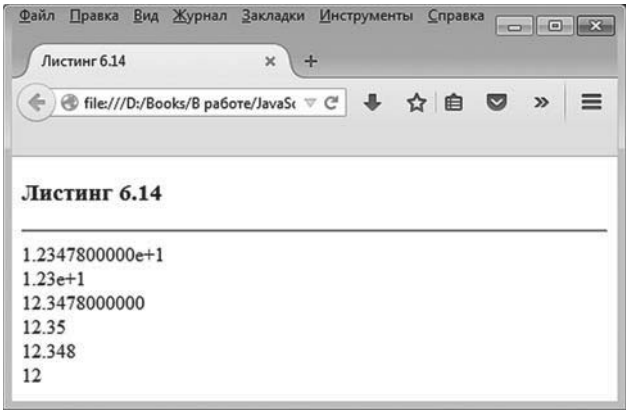

Рис. 6.25. Результат выполнения сценария, в котором используются методы объекта *Number*

Желающие могут самостоятельно поэкспериментировать с методами объекта Number. Мы же на данном объекте больше останавливаться не будем.

Объект Boolean

Встроенный объект Boolean используется для представления логических значений. Аргументом конструктору Boolean может передаваться логическое значение (true или false), равно как и значение другого типа. Если аргумент конструктора не относится к логическому типу, то правило вычисления логического объекта на основе нелогического аргумента сводится к следующему:

- при нулевом значении аргумента, пустой текстовой строке "", значении null, false, NaN и undefined (а также при отсутствии аргумента) создается объект, соответствующий логическому значению false;
- во всех прочих случаях создается объект, соответствующий логическому значению true.

Важно понимать разницу между объектом, созданным с помощью конструктора Boolean, и базовыми логическими значениями true и false. Объект — это объект. Если объект указывается в месте, где должно быть логическое значение, то выполняется автоматическое приведение данного объекта к логическому значению. Можно было бы ожидать, что объекты, созданные с помощью конструктора Boolean, приводятся в соответствии с их «внутренним» значением. Но это не так. Действует общее правило преобразования объектов к логическому типу: если объект отличен от undefined и null, он интерпретируется как значение true. В этом смысле показательным является программный код в листинге 6.15.

Листинг 6.15. Использование объектов, созданных конструктором Boolean (файл Listing06_15.js)

```
var myTrue=new Boolean(true)
var myFalse=new Boolean(false)
if(myTrue){
  document.write("Объект myTrue<br>")
}
if(myTrue==true){
```

```
  document.write("Снова объект myTrue<br>")
}
if(myFalse){
  document.write("Объект myFalse<br>")
}
if(myFalse==false){
  document.write("Снова объект myFalse")
}
```

Ниже представлен HTML-код для тестирования сценария:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 6.15</title>
</head>
<body><h3>Листинг 6.15</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing06_15.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Результат выполнения сценария такой.

Результат выполнения сценария (из листинга 6.15)

```
Объект myTrue
Снова объект myTrue
Объект myFalse
Снова объект myFalse
```

На рис. 6.26 показано, как выглядит веб-документ со сценарием, если его открыть в браузере.

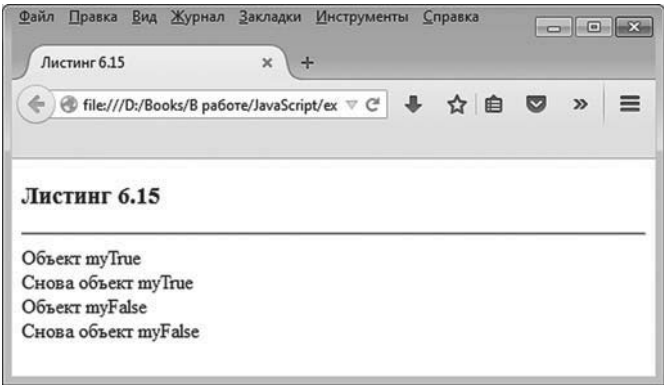

Рис. 6.26. Результат выполнения сценария, в котором используется конструктор *Boolean*

Несложно заметить, что объект myFalse, созданный на основе значения false, в условном операторе интерпретируется как значение true. Для «выявления сущности» данного объекта используем выражение myFalse==false, значение которого равняется true.

Объект String

Встроенный объект String предназначен для работы с текстовыми значениями. С помощью конструктора String создается объект, в который «упакована» текстовая строка. Создать текстовый объект String-типа можно с помощью команды следующего вида (жирным шрифтом выделены ключевые элементы кода):

```
var переменная=new String(аргумент)
```

Аргументом конструктору String обычно передается текстовое значение (базового типа).

ДЕТАЛИ

Объект, созданный с помощью конструктора String, отличается от базового текстового значения. Другими словами, текстовый литерал в двойных (или одинарных) кавычках — не одно и то же, что объект, созданный посредством конструктора String (даже если в объект «спрятана» точно такая же строка, что и текстовый литерал).

Вместе с тем со значениями базового текстового типа можно использовать методы объекта String. Причина в том, что в JavaScript действует автоматическое приведение базовых текстовых значе-

ний к объектам String-типа. Если из базового текста вызывается метод, то соответствующий текст базового типа приводится к объекту String-типа, из которого и вызывается метод.

Среди свойств объекта String достойно упоминания свойство length. Оно возвращает количество символов в тексте для объекта, у которого запрашивается свойство. Например, значением выражения "abcdefg". length является число 7, поскольку в текстовой строке "abcdefg" ровно 7 символов.

Некоторые наиболее интересные методы объекта String представлены в табл. 6.3 (за исключением метода fromCharCode(), который является собственным методом объекта String, все прочие методы наследуются из прототипа String.prototype). Многие методы напоминают методы для работы с массивами, и это в принципе не случайно: текстовая строка представляет собой упорядоченный набор символов, что можно интерпретировать как частный случай массива.

Таблица 6.3. Методы объекта String

Метод	Описание
anchor()	Метод предназначен для программного размещения якоря в веб-странице. Метод вызывается из текста гиперссылки, а аргумент метода служит значением атрибута name в дескрипторе <a>
charAt()	Метод для определения символа в текстовой строке по его индексу. Индекс символа в строке передается аргументом методу. Индексация символов в строке начинается с нуля
charCodeAt()	Методом возвращается код в кодовой таблице для символа с определенным индексом в текстовой строке. Индекс передается аргументом методу
concat()	Методом возвращается текстовая строка, которая получается объединением текстовой строки, из которой вызывается метод, а также строк, переданных аргументами методу
indexOf()	Методом возвращается индекс первого вхождения в текстовую строку фрагмента, указанного аргументом метода. Если в строке данного фрагмента нет, возвращается значение -1. Вторым аргументом методу можно передать индекс, начиная с которого выполняется поиск
fromCharCode()	Метод вызывается из объекта String, и результатом возвращается строка из символов, коды которых передаются аргументами методу
lastIndexOf()	Методом возвращается индекс последнего вхождения в текстовую строку фрагмента, указанного аргументом метода. Если в строке данного фрагмента нет, возвращается значение -1. Второй необязательный аргумент метода определяет индекс, начиная с которого выполняется поиск

Метод	Описание
link()	Метод для создания гиперссылки. Объект, из которого вызывается метод, становится текстом гиперссылки, а адрес перехода по гиперссылке определяется аргументом метода
localeCompare()	Результатом метода возвращается число, служащее индикатором того, как будут размещаться сравниваемые строки (строка, из которой вызывается метод, и строка, переданная аргументом методу) при сортировке. Отрицательное число означает, что строка вызова при сортировке размещается перед строкой-аргументом, ноль означает одинаковый приоритет строк, а положительное число означает, что строка вызова при сортировке будет находиться после строки-аргумента
match()	Метод используется для сопоставления строки, из которой он вызывается, и выражения, переданного аргументом методу. Методом возвращается объект Array-типа, содержащий информацию о результатах сопоставления
replace()	Результатом возвращается текстовая строка, которая получается заменой в исходной строке (из которой вызывается метод) одних фрагментов на другие. Заменяемый фрагмент указывается первым аргументом метода, а замещающий фрагмент — вторым аргументом
search()	Методом выполняется поиск выражения, переданного аргументом методу, в строке, из которой вызывается метод. Результатом возвращается индекс, определяющий позицию первого вхождения искомого фрагмента в строку. Если строка фрагмент не содержит, то результатом возвращается -1
slice()	Методом возвращается подстрока той строки, из которой вызывается метод (сама строка не изменяется). Первый аргумент метода определяет начальную позицию (индекс символа в исходной строке) для считывания символов. Второй необязательный аргумент — индекс первого не входящего в подстроку символа. Если второй аргумент не указан, символы считываются до конца строки. Если второй аргумент отрицательный, то его модуль определяет позицию символа, начиная с конца строки
split()	Результатом метода возвращается массив. Элементами массива являются подстроки, на которые разбивается строка, из которой вызывается метод (сама строка при этом не изменяется). Разделитель (например, символ или последовательность символов) для разбивки строки на подстроки указывается аргументом метода. Если разделитель не указан, то массив-результат состоит из одного элемента, содержащего исходную строку. Если аргументом указана пустая текстовая строка, то результатом является массив из символов, формирующих исходную строку
substr()	Метод возвращает подстроку символов из строки вызова метода. Аргументами передается начальный индекс символа и количество считываемых символов
substring()	Методом возвращается подстрока той строки, из которой вызывается метод. Аргументами методу передается индекс элемента, начиная с которого считывается подстрока. Второй необязательный аргумент определяет индекс первого не включенного в подстроку символа. Если второй аргумент не указан, то считывание выполняется до конца строки

Метод	Описание
toLocaleLowerCase()	Метод для преобразования символов текстовой строки, из которой вызван метод, в нижний регистр (строчные символы) с учетом региональных настроек. Исходная строка не изменяется, результатом возвращается новая строка
toLocaleUpperCase()	Метод для преобразования символов текстовой строки, из которой вызван метод, в верхний регистр (прописные символы) с учетом региональных настроек. Исходная строка не изменяется, результатом возвращается новая строка
toLowerCase()	Методом возвращается текстовая строка, которая получается переводом всех символов в нижний регистр в строке, из которой вызывается метод (в строке-результате все буквы строчные)
toUpperCase()	Методом возвращается текстовая строка, которая получается из исходной строки (той, из которой вызывается метод) переводом всех символов в верхний регистр (в строке-результате все буквы прописные)
trim()	Методом возвращается строка, которая получается из исходной строки (той, из которой вызывается метод) удалением пробелов в начале и в конце строки

НА ЗАМЕТКУ

Есть еще, например, методы toString() и valueOf(), которые для текстового объекта возвращают «упакованную» в объект текстовую строку.

Как иллюстрация в листинге 6.16 представлены примеры использования некоторых из перечисленных выше методов.

Листинг 6.16. Методы для работы с текстовыми значениями (файл Listing06_16.js)

```
// Исходная текстовая строка:
var text="Мы изучаем JavaScript"
document.write("<b>"+text+"</b><br>")
// Обращение к символам строки через индекс:
for(var k=11;k<text.length;k++){
  document.write(text[k])
}
// Использование метода slice():
document.write("<br>"+text.slice(0,10))
document.write("<br>"+text.slice(11))
// Использование метода substring():
document.write("<br>"+text.substring(11))
// Использование метода substr():
document.write("<br>"+text.substr(11))
```

```
// Использование метода toUpperCase():
document.write("<br>"+text.toUpperCase())
// Использование метода toLowerCase():
document.write("<br>"+text.toLowerCase())
// Использование метода search():
document.write("<br>"+text.search("Java"))
// Использование метода replace():
document.write("<br>"+text.replace("изучаем","любим"))
// Исходная строка не изменилась:
document.write("<br><b>"+text+"</b>")
```

Для тестирования сценария используем HTML-код, приведенный ниже:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 6.16</title>
</head>
<body><h3>Листинг 6.16</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing06_16.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Текстовая версия для результата выполнения сценария такая.

Результат выполнения сценария (из листинга 6.16)

```
Мы изучаем JavaScript
JavaScript
Мы изучаем
JavaScript
JavaScript
JavaScript
```

МЫ ИЗУЧАЕМ JAVASCRIPT
мы изучаем javascript
11
Мы любим JavaScript
Мы изучаем JavaScript

На рис. 6.27 показано, как выглядит окно браузера с открытым в нем веб-документом с описанным выше сценарием.

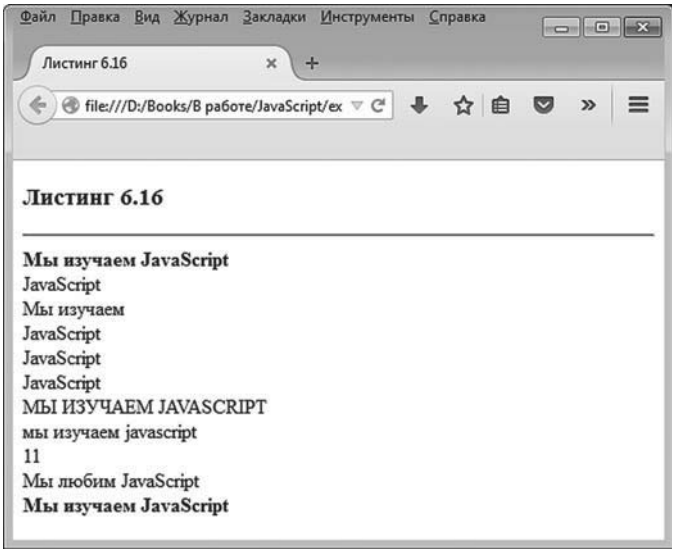

Рис. 6.27. Результат выполнения сценария, в котором использованы методы для работы с текстовыми значениями

Помимо прочего хочется отметить, что к символам в текстовой строке можно обращаться через индексы (индекс указывается в квадратных скобках после собственно текстовой строки). В таком случае следует помнить, что индексация символов начинается с нуля. Пример выполнения подобной операции есть в представленном выше сценарии.

Объект Date

На основе конструктора Date создаются объекты, предназначенные для работы с датой и временем. Объект, созданный в соответствии

с инструкцией new Date(), содержит информацию о текущей дате и времени (на момент выполнения команды). Дата и время, как несложно догадаться, определяются по системным настройкам. Конструктор Date также можно вызвать с аргументом или аргументами. В таком случае аргументы конструктора идентифицируют дату и/или время для создаваемого объекта.

ДЕТАЛИ

Аргументами конструктору Date могут передаваться целочисленные значения (все или только часть из перечисленных далее), определяющие: год, месяц (индексация месяцев начинается с нуля), часы, минуты и секунды. Можно указать один целочисленный аргумент, который в таком случае определяет время, которое прошло с 1 января 1970 года, в миллисекундах (миллисекунда — одна тысячная секунды). Кроме перечисленных способов, аргументом конструктору может передаваться текстовая строка формата "число месяц год часы:минуты:секунды", как, например, строка "22 Oct 1974 06:20:15", что соответствует 22 октября 1974 года, 6 часам 20 минутам и 15 секундам.

Для понимания принципов оперирования с объектами следует принять во внимание, что концептуально дата определяется как время (в миллисекундах), прошедшее с 1 января 1970 года (начиная с момента времени 00:00:00).

Другими словами, дата — это на самом деле число, которое равно интервалу времени в миллисекундах между данной датой и 1 января 1970 года. Это число при необходимости преобразуется в дату календаря. Но базовые операции с датами обычно выполняются на уровне представления даты в миллисекундах.

Ситуация усложняется тем, что разные люди (а значит, и разные пользователи) проживают в разных часовых поясах. Есть такое понятие, как *всемирное координированное время* (сокращенно обозначается как *UTC* в соответствии с английским названием *Coordinated Universal Time*) — международный стандарт времени, который с некоторой натяжкой можно рассматривать как аналог *среднего времени по Гринвичу* (сокращенно *GMT* от *Greenwich Mean Time*).

Местное время отличается от всемирного координированного времени. Разница между местным и всемирным координированным временем определяется *временным сдвигом*.

НА ЗАМЕТКУ

На самом деле не все так просто, но нас в данном случае вопросы стандартов времени интересуют не очень сильно. Важно то, что есть некое «универсальное» время и есть местное время.
Также стоит отметить, что для всемирного координированного времени перевод на зимнее и летнее время не выполняется — изменяется местное время, и в силу этого изменяется и сдвиг часовых поясов.

Поскольку дата фактически определяется через относительный параметр, а на практике интерес представляет календарная дата, то принципиальное значение имеет вопрос о том, в каком часовом поясе находится конечный пользователь (точнее, какой часовой пояс задан в операционной системе, под управлением которой запускается браузер). Отсюда при операциях с датами необходимо учитывать, о каком времени идет речь: местном или всемирном координированном. При этом концептуальное положение состоит в том, что дата — это число (количество миллисекунд, отсчитанное от 1 января 1970 года по всемирному координированному времени). А вот преобразование «числовой» даты в календарную дату происходит с учетом местного часового пояса.

Следует понимать, что хотя объект Date-типа и содержит информацию о дате и времени, для использования данной информации ее необходимо оттуда «извлечь». Для решения такой и ряда смежных задач используются методы объекта Date, большинство которых наследуется из прототипа Date.prototype (исключение составляют методы UTC(), now() и parse()).

ДЕТАЛИ

С какими датами оперирует тот или иной метод, в принципе, можно определить исходя из наличия или отсутствия ключевой фразы *UTC* в названии метода. Наличие фразы *UTC* в названии метода свидетельствует о том, что дата обрабатывается по стандарту всемирного координированного времени.
Еще один совет касается следующего: если название метода начинается с ключевого слова *get*, то метод предназначен для считывания некоторого значения. Если название метода начинается с ключевого слова *set*, то метод предназначен для присваивания значения некоторому свойству или определения некоторой характеристики.

Основные методы объекта Date представлены в табл. 6.4.

Таблица 6.4. Методы объекта Date

Метод	Описание
getDate()	Методом возвращается день месяца в дате (указанной по местному времени), содержащейся в объекте, из которого вызывается метод
getDay()	Для даты (по местному времени), записанной в объекте, из которого вызывается метод, возвращается день недели (при этом 0 соответствует *воскресенью*)
getFullYear()	Методом для даты (указанной по местному времени), записанной в объекте, из которого вызывается метод, возвращается год
getHours()	Методом для даты (указанной по местному времени), записанной в объекте, из которого вызывается метод, возвращаются часы
getMilliseconds()	Методом для даты (указанной по местному времени) возвращаются миллисекунды
getMinutes()	Методом для даты (указанной по местному времени), записанной в объекте, из которого вызывается метод, возвращаются минуты
getMonth()	Методом для даты (указанной по местному времени), записанной в объекте, из которого вызывается метод, возвращается месяц. Значение 0 соответствует *январю*, 1 соответствует *февралю*, и так далее, до значения 11, которое соответствует *декабрю*
getSeconds()	Методом для даты (указанной по местному времени), записанной в объекте, из которого вызывается метод, возвращаются секунды
getTime()	Методом возвращается количеством миллисекунд от даты 1 января 1970 года по всемирному координированному времени до даты (по местному времени), записанной в объект, из которого вызывается метод
getTimezoneOffset()	Методом возвращается значение (в минутах) смещения местного часового пояса
getUTCDate()	Методом возвращается день месяца в дате (по всемирному координированному времени), содержащейся в объекте, из которого вызывается метод
getUTCDay()	Для даты (по всемирному координированному времени), записанной в объекте, из которого вызывается метод, возвращается день недели (при этом 0 соответствует *воскресенью*)
getUTCFullYear()	Методом для даты (по всемирному координированному времени), записанной в объекте, из которого вызывается метод, возвращается год
getUTCHours()	Методом для даты (по всемирному координированному времени), записанной в объекте, из которого вызывается метод, возвращаются часы
getUTCMilliseconds()	Методом для даты (по всемирному координированному времени) возвращаются миллисекунды

Метод	Описание
getUTCMinutes()	Методом для даты (по всемирному координированному времени), записанной в объекте, из которого вызывается метод, возвращаются минуты
getUTCMonth()	Методом для даты (по всемирному координированному времени), записанной в объекте, из которого вызывается метод, возвращается месяц. Значение 0 соответствует *январю*, 1 соответствует *февралю*, и так далее, до значения 11, которое соответствует *декабрю*
getUTCSeconds()	Методом для даты (по всемирному координированному времени), записанной в объекте, из которого вызывается метод, возвращаются секунды
now()	Результатом метода является количество миллисекунд, прошедших с 1 января 1970 года по всемирному координированному времени до настоящего момента. Метод вызывается из объекта Date
parse()	Метод вызывается из объекта Date, а аргументом ему передается текстовое значение для даты (по местному времени). Результатом метод возвращает количество миллисекунд, прошедших с 1 января 1970 года по всемирному координированному времени до даты, переданной аргументом методу
setDate()	Методом задается день месяца (аргумент метода — число от 1 до 31) для данной даты (определяется объектом, из которого вызывается метод) по местному времени
setFullYear()	Методом задается год (аргумент метода) для данной даты (определяется объектом, из которого вызывается метод) по местному времени
setHours()	Методом задаются часы (аргумент метода — от 0 до 23) для данной даты (определяется объектом, из которого вызывается метод) по местному времени
setMilliseconds()	Методом задаются миллисекунды (аргумент метода — значение от 0 до 999) для данной даты (определяется объектом, из которого вызывается метод) по местному времени
setMinutes()	Методом задаются минуты (аргумент метода — значение от 0 до 59) для данной даты (определяется объектом, из которого вызывается метод) по местному времени
setMonth()	Методом задается месяц (аргумент метода — значение от 0 до 11) для данной даты (определяется объектом, из которого вызывается метод) по местному времени
setSeconds()	Методом задаются секунды (аргумент метода — значение от 0 до 59) для данной даты (определяется объектом, из которого вызывается метод) по местному времени
setTime()	Методом задается дата в миллисекундах (аргумент метода) по отношению к 1 января 1970 года по всемирному координированному времени
setUTCDate()	Методом задается день месяца (аргумент метода — число от 1 до 31) для данной даты (определяется объектом, из которого вызывается метод) по всемирному координированному времени

Метод	Описание
setUTCFullYear()	Методом задается год (аргумент метода) для данной даты (определяется объектом, из которого вызывается метод) по всемирному координированному времени
setUTCHours()	Методом задаются часы (аргумент метода — от 0 до 23) для данной даты (определяется объектом, из которого вызывается метод) по всемирному координированному времени
setUTCMilliseconds()	Методом задаются миллисекунды (аргумент метода — значение от 0 до 999) для данной даты (определяется объектом, из которого вызывается метод) по всемирному координированному времени
setUTCMinutes()	Методом задаются минуты (аргумент метода — значение от 0 до 59) для данной даты (определяется объектом, из которого вызывается метод) по всемирному координированному времени
setUTCMonth()	Методом задается месяц (аргумент метода — значение от 0 до 11) для данной даты (определяется объектом, из которого вызывается метод) по всемирному координированному времени
setUTCSeconds()	Методом задаются секунды (аргумент метода — значение от 0 до 59) для данной даты (определяется объектом, из которого вызывается метод) по всемирному координированному времени
toDateString()	Методом возвращается текстовое представление для даты (и только для даты)
toISOString()	Методом возвращается текстовая строка в формате ISO с информацией о дате и времени
toJSON()	Методом возвращается представление объекта Data-типа в стандарте JSON
toLocaleDateString()	Методом возвращается текстовое представление (с учетом региональных настроек) для даты (и только для даты)
toLocaleString()	Текстовое представление для даты с учетом региональных настроек
toLocaleTimeString()	Методом возвращается текстовое представление (с учетом региональных настроек) для времени (и только времени) в дате
toString()	Методом возвращается текстовое представление для объекта даты по местному времени
toTimeString()	Методом возвращается текстовое представление для времени (и только времени) в дате
toUTCString()	Методом возвращается текстовое представление для объекта даты по всемирному координированному времени
UTC()	Методу аргументами передаются числовые значения, определяющие год, месяц, день, часы, минуты и секунды для даты по всемирному координированному времени. Результатом возвращается количество миллисекунд, прошедших с 1 января 1970 года (по всемирному координированному времени) до даты, определяемой аргументом метода
valueOf()	Метод для приведения объекта Date-типа к значению базового (примитивного) типа. Более конкретно методом возвращается числовое значение, равное количеству миллисекунд до данной даты, прошедших с 1 января 1970 года по всемирному координированному времени

Некоторые примеры выполнения операций с датами приведены в листинге 6.17.

Листинг 6.17. Операции с датой и временем (файл Listing06_17.js)

```
// Текущая дата и время:
var today=new Date()
// Следующий день:
var tomorrow=new Date(today)
tomorrow.setDate(tomorrow.getDate()+1)
// Предыдущий день:
var yesterday=new Date(today)
yesterday.setDate(yesterday.getDate()-1)
// Через месяц:
var monthAfter=new Date(Date.now())
monthAfter.setMonth(monthAfter.getMonth()+1)
// Через год:
var yearAfter=new Date(Date.now())
yearAfter.setFullYear(yearAfter.getFullYear()+1)
// Отображение дат:
document.write("<b>Сегодня:</b> "+today+"<br>")
document.write("<u>Год:</u> "+today.getFullYear()+"<br>")
document.write("<u>Дата:</u> "+today.toLocaleDateString()+"<br>")
document.write("<u>Время:</u> "+today.toLocaleTimeString()+"<br>")
document.write("<b>Завтра:</b> "+tomorrow+"<br>")
document.write("<b>Вчера:</b> "+yesterday+"<br>")
document.write("<b>Через месяц:</b> "+monthAfter+"<br>")
document.write("<b>Через год:</b> "+yearAfter+"<br>")
// Время встречи (явно задаем дату и время):
var meeting=new Date(2015,8,30,8,0,0)
// Отображение дат:
document.write("<b>Сегодня:</b> "+today.toLocaleString()+"<br>")
document.write("<b>Время и дата встречи:</b> "+meeting.toLocaleString()+"<br>")
// Изменение даты:
meeting.setDate(meeting.getDate()+1)
```

```
// Отображение новой даты:
document.write("<b>Новое время и дата встречи:</b> "+meeting.toLocaleString()+"<br>")
// Время до встречи (в миллисекундах):
document.write("<u>До встречи:</u> "+(meeting-today)+" миллисекунд<br>")
// Время до встречи (в днях):
document.write("<u>До встречи:</u> "+Math.round((meeting-today)/1000/60/60/24)+" дней<br>")
```

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Мы использовали в сценарии дескрипторы <u> и </u> для отображения текста с подчеркиванием, а также и для применения жирного шрифта.

Сценарий тестируем с помощью следующего HTML-кода:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 6.17</title>
</head>
<body><h3>Листинг 6.17</h3><hr>

<!-- Начало сценария -->
<script type="text/javascript" src="Listing06_17.js">
</script>
<!-- Завершение сценария -->

</body>
</html>
```

Текстовая версия результата выполнения сценария представлена ниже.

Результат выполнения сценария (из листинга 6.17)

Сегодня: Sun Jun 28 2015 19:29:59 GMT+0300

Год: 2015

Дата: 28.06.2015

Время: 19:29:59

Завтра: Mon Jun 29 2015 19:29:59 GMT+0300

Вчера: Sat Jun 27 2015 19:29:59 GMT+0300

Через месяц: Tue Jul 28 2015 19:29:59 GMT+0300

Через год: Tue Jun 28 2016 19:29:59 GMT+0300

Сегодня: 28.06.2015, 19:29:59

Время и дата встречи: 30.09.2015, 8:00:00

Новое время и дата встречи: 01.10.2015, 8:00:00

До встречи: 8166600948 миллисекунд

До встречи: 95 дней

На рис. 6.28 показано, как выглядит веб-документ со сценарием, открытый в окне браузера.

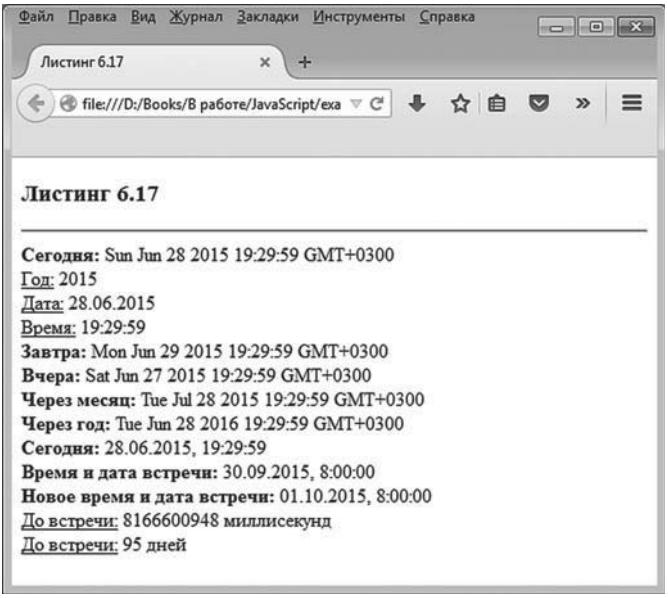

Рис. 6.28. Результат выполнения сценария, в котором задействованы операции с датой и временем

Сценарий достаточно простой, поэтому прокомментируем лишь некоторые команды. Так, инструкцией var today=new Date() создается объект today, содержащий текущую дату и время. Объекты tomorrow и yesterday со-

здаются как копии объекта today — этот объект передается аргументом конструктору Date. Затем для каждого из объектов tomorrow и yesterday изменяется день в дате: для объекта tomorrow он увеличивается на единицу, а для объекта yesterday уменьшается на единицу. Выполняются данные операции просто: методом getDate() считывается текущее значение, а методом setDate() присваивается новое. Похожим образом обрабатываются и объекты monthAfter и yearAfter. Однако при их создании конструктору передается выражение Date.now(), которое определяет текущую дату в миллисекундах. При изменении значения для месяца используются методы setMonth() (присваивание нового значения для месяца в дате) и getMonth() (считывание текущего значения для месяца в дате), а при изменении значения года используем методы setFullYear() (присваивание значения для года в дате) и getFullYear() (считывание значения для года в дате).

ДЕТАЛИ

Вместо выражения new Date(Date.now()) на практике разумнее использовать выражение new Date(). В обоих случаях создается объект, соответствующий текущему моменту времени, просто при выполнении инструкции new Date(Date.now()) для текущего момента вычисляется интервал в миллисекундах, а затем данное значение передается конструктору. Мы использовали команды вида new Date(Date.now()) с целью проиллюстрировать возможность передачи конструктору Date единственного целочисленного аргумента.

Также следует отметить, что теоретически, создавая объекты инструкциями вида new Date() или вызывая метод now(), каждый раз получаем новые результаты, поскольку речь идет соответственно о создании объекта для текущего момента времени или вычислении интервала в миллисекундах для текущего момента времени. А «текущий момент» постоянно меняется. Но если сценарий небольшой и выполняется быстро, то все происходит в доли секунды, так что разница обычно малозаметна.

Объект meeting создается с явной передачей аргументов конструктору Date (год 2015, месяц 8 (сентябрь), число 30, время — 8 часов, 0 минут и 0 секунд). Затем число (день месяца) увеличивается на единицу. Формально мы должны были бы получить дату *31 сентября*, которой в природе не существует. Однако ситуация обрабатывается корректно, и в итоге получаем *1 октября*.

Чтобы узнать интервал времени от текущего момента до даты, записанной в meeting, используем выражение meeting-today. Благодаря авто-

матическому вызову метода valueOf() в результате получаем разницу между датами в миллисекундах. Чтобы получить то же значение, но в днях, делим интервал в миллисекундах на 1000 (получаем значение в секундах), на 60 (получаем значение в минутах), на 60 (получаем значение в часах) и на 24 (получаем значение в днях). Полученное число будет нецелым, поэтому округляем его с помощью метода round() стандартного встроенного объекта Math.

Резюме

> Соев, голубчик, я уважаю вашу супругу, я глубоко ценю ее вкус, но прошу вас, посмотрите финал!
>
> *из к/ф «Покровские ворота»*

Подытоживая, имеет смысл отметить следующее.

- В JavaScript существует система перехвата и обработки исключительных ситуаций. Основу системы обработки исключений составляет блок try-catch-finally.
- Контролируемый код помещается в блок try. Если при выполнении данного кода возникает ошибка, то она перехватывается в блоке catch: выполнение try-блока прекращается, и начинается выполнение catch-блока. Код, помещенный в блок finally, выполняется независимо от того, возникала в try-блоке ошибка или нет.
- В catch-блок передается объект ошибки, который имеет ряд свойств (таких, как name и message), позволяющих получить информацию об ошибке.
- Исключение может генерироваться искусственно. Для этого используется ключевое слово throw, после которого указывается объект ошибки. Обычно объект ошибки создается с помощью стандартного встроенного конструктора Error.
- К свойству или методу объекта можно обращаться не только с использованием точечного синтаксиса (когда свойство или метод указывается через точку после имени объекта), но и с помощью индексной нотации: имя (в кавычках) свойства или метода указывается в квадратных скобках после имени объекта.
- Для объектов целесообразно определять методы toString() и valueOf(), автоматически вызываемые, соответственно, при преобразовании

объекта к текстовому формату и приведении объекта к значению простого (базового) типа.

- Функция является объектом. Для определения количества аргументов, указанных при описании функции, используют свойство length. Информацию о фактически переданных функции аргументах получают с помощью объекта arguments, доступного в теле функции. Объект arguments аналогичен массиву: он допускает индексацию и имеет свойство length. Для объекта функции бывает целесообразно переопределить метод toString().
- Имеется группа встроенных объектов, существенно расширяющих функциональные возможности программных кодов. Помимо рассмотренных ранее встроенных объектов, можно выделить следующие: Math (математические утилиты), Number (средства для работы с числовыми значениями), Boolean (средства для работы с логическими значениями), String (утилиты для работы с текстовыми значениями) и Date (средства для работы с датой и временем).

Часть III

ИСПОЛЬЗОВАНИЕ JAVASCRIPT

Глава 7
ВЕБ-ДОКУМЕНТЫ И СЦЕНАРИИ

> Если ты еще раз вмешаешься в опыты академика и встанешь на пути технического прогресса, я тебя...
>
> *из к/ф «Иван Васильевич меняет профессию»*

В этой главе мы начинаем знакомство с методами использования в веб-документах сценариев, написанных на JavaScript. Речь будет идти о том, как сценарии могут «инкапсулироваться» в веб-документ, а также о том, как организовать взаимодействие сценария с браузером и открытым в нем документом.

Место и роль сценария в веб-документе

> Каюсь, что не по своей воле, а по принуждению князя Милославского временно исполнял обязанности царя.
>
> *из к/ф «Иван Васильевич меняет профессию»*

В принципе у нас есть некоторый опыт размещения сценариев в веб-документах. Но, откровенно говоря, он не очень большой (хотя сценариев уже рассмотрено немало). Практически каждый раз мы добавляли сценарий в веб-документ через ссылку на файл со сценарием в блоке, выделенном дескрипторами <script> и </script>. Но там речь шла о тестировании работы сценариев. Теперь же мы переходим к изучению вопроса об их эффективном применении на практике, поэтому важно иметь представление обо всем спектре возможностей.

Размещение сценария в документе

Обычно выделяют четыре способа инкапсуляции сценария в HTML-код (два из них нам знакомы). Итак, добавить сценарий в веб-документ можно следующим образом.

- Разместив код сценария между дескрипторами <script> и </script>.
- Разместить в блоке, выделенном дескрипторами <script> и </script>, ссылку на файл со сценарием. Ссылка на файл указывается значением атрибута src.
- Использовать код JavaScript в веб-документе в качестве URL-адреса. Соответствующая ссылка должна содержать идентификатор javascript, служащий признаком наличия в ссылке кода, который следует выполнять под управлением интерпретатора JavaScript.
- Использовать JavaScript-код для создания *обработчиков событий* всевозможных элементов в веб-документе.

Две последние позиции требуют, пожалуй, пояснений. Начнем с добавления программного кода в инструкцию с URL-адресом. Принцип достаточно простой: в том месте, где должен быть указан URL-адрес, указывается инструкция, содержащая ключевое слово javascript, и после двоеточия набор команд на языке JavaScript. Между собой команды разделяются точкой с запятой. Основные места, где может быть URL-адрес (и, соответственно, команды JavaScript), — это срока ввода адреса в браузере и значение атрибута href в гиперссылке. В последнем случае вся эта конструкция берется в двойные или одинарные кавычки.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Если в веб-документе необходимо создать гиперссылку, то соответствующий блок текста (который должен стать гиперссылкой) выделяется дескрипторами <a> и . Место (URL-адрес), куда выполняется переход при щелчке по гиперссылке, задается в качестве значения атрибута href в открывающем дескрипторе <a>.

Например, нам необходимо выполнить следующие команды (вычисление суммы натуральных чисел от 1 до 10 и отображение результата в специальном диалоговом окне):

```
var s=0
for(var k=1;k<=10;k++){
  s+=k
}
alert("1+2+...+10="+s)
```

ДЕТАЛИ

В данном случае мы используем функцию alert() (если точнее, то это метод объекта окна window, просто при его вызове разрешается не указывать ссылку на объект window). При вызове функции alert() в окне браузера отображается модальное диалоговое окно (*модальное* означает, что, пока окно не закрыто, область веб-документа заблокирована). В области диалогового окна отображается текст, переданный аргументом функции alert().

В таком случае мы вводим в адресную строку браузера следующую инструкцию:

```
javascript:var s=0;for(var k=1;k<=10;k++){s+=k};alert("1+2+...+10="+s)
```

Как все это выглядит в реальности, показано на рис. 7.1.

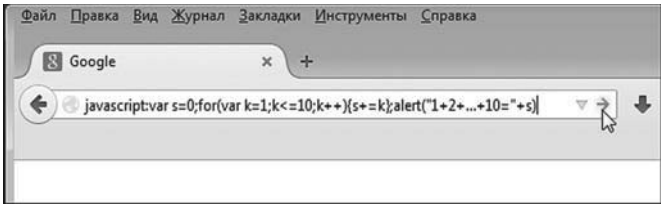

Рис. 7.1. Ввод в адресной строке браузера команды для выполнения интерпретатором JavaScript

Результат (после нажатия клавиши **Enter**) будет таким, как показано на рис. 7.2.

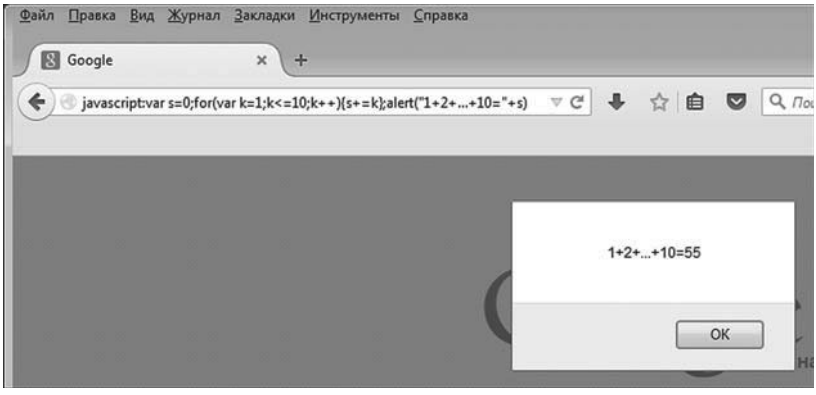

Рис. 7.2. Результат выполнения кода JavaScript, введенного в адресную строку браузера

Хотя для разных браузеров диалоговое окно с результатом вычислений выглядит по-своему, принцип остается неизменным: есть окно, и в этом окне есть сообщение с результатом вычислений.

Если мы хотим реализовать те же вычисления, но так, чтобы начинались они при щелчке по гиперссылке, то вполне приемлемым станет HTML-код, содержащий команды на JavaScript, представленный в листинге 7.1.

Листинг 7.1. Щелчок по гиперссылке приводит к выполнению программного кода (файл Listing07_01.html)

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 7.1</title>
</head>
<body><h3>Листинг 7.1</h3><hr>
<!-- Гиперссылка с командой JavaScript -->
<a href="javascript: var s=0; for(var k=1; k<=10;k++){s+=k}; alert('1+2+...+10='+s)">
  Вычислить сумму от 1 до 10
</a>
<!-- Завершение гиперссылки -->

</body>
</html>
```

Как будет выглядеть соответствующий документ, открытый в окне браузера, показано на рис. 7.3.

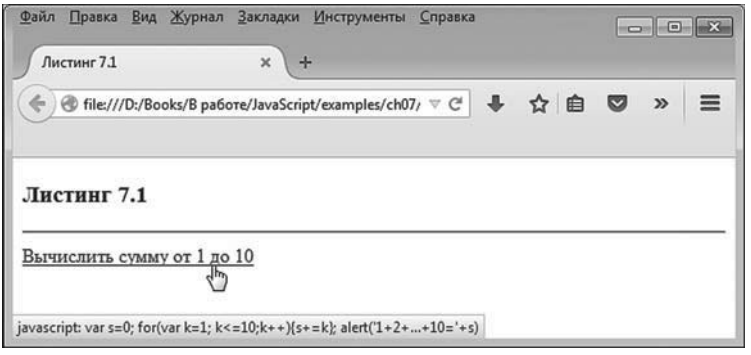

Рис. 7.3. Документ с гиперссылкой, щелчок по которой приводит к выполнению программного кода JavaScript

НА ЗАМЕТКУ

Обычно при наведении курсора мыши на гиперссылку по умолчанию отображается адрес, по которому выполняется переход при щелчке по гиперссылке. В данном случае роль такого «адреса перехода» играет программный код, выполняемый при щелчке по гиперссылке.

Результат, который получаем после щелчка по гиперссылке, проиллюстрирован на рис. 7.4.

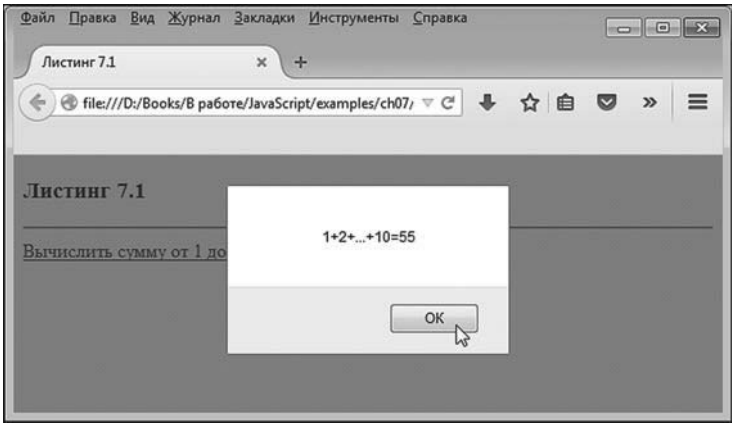

Рис. 7.4. В результате щелчка по гиперссылке появляется диалоговое окно с информацией о значении суммы натуральных чисел

При щелчке по гиперссылке появляется диалоговое модальное окно с информацией о результате вычисления суммы натуральных чисел.

Выше мы весь выполняемый при щелчке по гиперссылке код указывали непосредственно в гиперссылке (значением атрибута href). Очевидно, что это не всегда удобно. Если программный код слишком громоздкий, его разбивают на несколько блоков. Пример такого подхода проиллюстрирован в листинге 7.2. Там решается похожая задача (по сравнению с предыдущим примером), однако программный код реализован несколько иначе.

Листинг 7.2. Гиперссылка и программный код (файл Listing07_02.html)

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 7.2</title>
```

```
  <!-- Начало блока с кодом JavaScript -->
  <script type="text/javascript">
    // Функция для вычисления суммы натуральных чисел:
    function findSum(n){
      var s=0
      for(var k=1;k<=n;k++){
        s+=k
      }
      return s
    }
    // Функция для считывания верхней границы суммы:
    function get(){
      var n
      n=parseInt(prompt("Укажите число:"))
      return n
    }
  </script>
    <!-- Завершение блока с кодом JavaScript -->
</head>
<body><h3>Листинг 7.2</h3><hr>
<!-- Гиперссылка с командой JavaScript -->
<a href="javascript: var n=get(); alert('1+2+...+'+n+'='+findSum(n))">
  Вычислить сумму натуральных чисел
</a>
<!-- Завершение гиперссылки -->
</body>
</html>
```

В представленном веб-документе основная часть программного кода вынесена в <script>-блок. Более конкретно, между дескрипторами <script> и </script> размещено описание функции findSum(), предназначенной для вычисления суммы натуральных чисел (верхняя граница суммы передается аргументом функции), а также описание функции get(), предназначенной для считывания числового значения с помощью диалогового окна с полем ввода.

ДЕТАЛИ

У функции get() нет аргументов. В теле функции объявляется переменная n, которой присваивается значение parseInt(prompt("Укажите число:")) и которая возвращается результатом функции. Инструкция prompt("Укажите число:") означает отображение диалогового окна с полем ввода. Над полем ввода отображается текст, переданный аргументом методу prompt() (как и в случае с методом alert(), метод prompt() может вызываться без указания ссылки на объект окна window). Методом prompt() возвращается результат — текстовое значение, которое пользователь указал в поле ввода. С помощью функции parseInt() текст преобразуется в целочисленный формат (мы предполагаем, что пользователь вводит в поле ввода целое число, и поэтому текст-результат является текстовым представлением целого числа). Это и есть результат функции get(). Функция findSum() описана с одним аргументом (обозначен как n). В теле функции объявляется несколько локальных переменных, запускается оператор цикла, в котором вычисляется сумма натуральных чисел от 1 до n, и полученное значение возвращается как результат функции.

Значением атрибута href гиперссылки указано значение "javascript: var n=get(); alert('1+2+...+'+n+'='+findSum(n))". Здесь «зашифрована» последовательность команд, которыми сначала считывается числовое значение, указанное пользователем в поле ввода, а затем данное значение используется (как верхняя граница суммы) при вычислении результата и отображении его в новом диалоговом окне. Все это, напомним, происходит при щелчке по гиперссылке. На рис. 7.5 показано, как выглядит веб-документ с гиперссылкой перед щелчком по ней. На рис. 7.6 показано (на фоне окна с веб-документом) диалоговое окно с полем ввода, в котором пользователь должен указать верхнюю границу для вычисления суммы. После щелчка по кнопке **ОК** открывается новое диалоговое окно, в котором отображается результат вычисления суммы натуральных чисел. Это окно показано на рис. 7.7.

НА ЗАМЕТКУ

При использовании разных браузеров диалоговые окна будут иметь разный вид. Но принцип выполнения кода всегда один и тот же: сначала появляется окно с полем ввода, а затем отображается окно с информацией о результатах вычисления суммы.

Также стоит заметить, что в рассмотренном примере мы не предусмотрели обработку ошибок, которые могут возникнуть при неправильно введенном значении для верхней границы суммы. Желающие могут попробовать решить такую задачу самостоятельно.

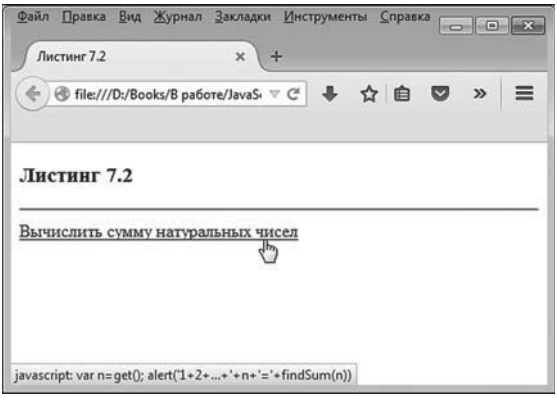

Рис. 7.5. Веб-документ с гиперссылкой, щелчок по которой приводит к выполнению программного кода

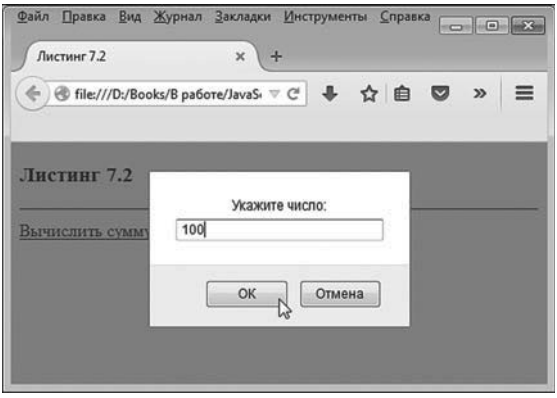

Рис. 7.6. В поле ввода указывается число, определяющее верхнюю границу при вычислении суммы натуральных чисел

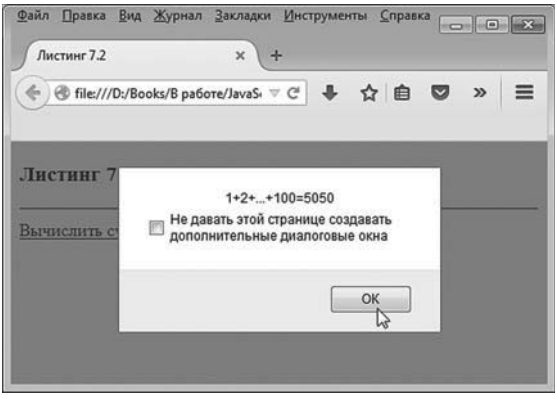

Рис. 7.7. Диалоговое окно с результатом вычисления суммы натуральных чисел

Отметим два важных обстоятельства, касающихся размещения <script>-блоков в сценариях.

- Блоков со сценарием в веб-документе может быть несколько.
- Размещать блок со сценарием в веб-документе можно практически где угодно. Однако следует помнить, что веб-документ загружается в память последовательно, в соответствии со своим кодом. Так что чем ближе блок со сценарием к началу документа, тем быстрее код загрузится. Нередко важный программный код включается в <head>-раздел.

Обработка событий

Очень удобно использовать сценарии или блоки кода для обработки *событий*. Дело в том, что элементы в веб-документе умеют реагировать на определенные действия, которые с ними выполняются. Набор таких действий предопределен и в известном смысле зависит от элемента. Скажем, есть событие, которое состоит в том, что завершена *загрузка* документа в окно браузера. Есть событие, состоящее в *щелчке* по кнопке. А есть событие, состоящее в *передаче фокуса* полю ввода. Мы со временем рассмотрим вопрос с событиями и их обработкой подробнее. Здесь в качестве иллюстрации проанализируем пример, в котором сценарий запускается на выполнение по завершении загрузки документа.

Существует несколько способов определять реакцию элементов на события. Это можно делать программными методами непосредственно в сценарии. Но самый простой путь — задать для элемента атрибут, который, собственно, и определяет реакцию элемента на определенное событие.

Как отмечалось выше, нас интересует событие, состоящее в загрузке документа. За данное событие отвечает атрибут onload. Он указывается в дескрипторе <body> веб-документа. Значением атрибуту onload присваивается текстовая строка с выполняемым кодом. Удобно код, предназначенный для выполнения, предварительно организовать в виде функции в сценарии, а затем указать команду вызова функции в качестве значения атрибута onload. В результате данная функция вызывается после загрузки веб-документа. Сам код с функцией может быть описан, например, в <head>-блоке. Пример такой ситуации представлен в листинге 7.3.

Листинг 7.3. Запуск сценария при загрузке документа (файл Listing07_03.html)

```
<!DOCTYPE HTML>
<html>
<head>
<title>Листинг 7.3</title>
<!-- Блок с кодом функции -->
<script type="text/javascript">
  // Функция отображает окно с сообщением:
  function show(){
    alert("Веб-документ загружен!")
  }
</script>
<!-- Завершение блока с кодом функции -->
</head>
<body onload="show()"><h3>Листинг 7.3</h3><hr>
  При загрузке документа отображается диалоговое окно.
</body>
</html>
```

Здесь мы имеем дело с документом, в котором после загрузки браузером начинает выполняться определенный программный код — точнее, как только документ загружен, выполняется функция show(). Функция описана в <script>-блоке, который, в свою очередь, находится в <head>-блоке документа. Функция описана так:

```
function show(){
  alert("Веб-документ загружен!")
}
```

При вызове функции show() отображается диалоговое окно с сообщением "Веб-документ загружен!". Эта функция указана значением атрибута onload в дескрипторе <body>. Поэтому, как только документ загружен в браузер, в окне браузера отображается диалоговое окно, как показано на рис. 7.8.

Стоит заметить, что поскольку окно отображается после загрузки документа, то на момент отображения диалогового окна содержимое веб-документа уже отображается в окне браузера.

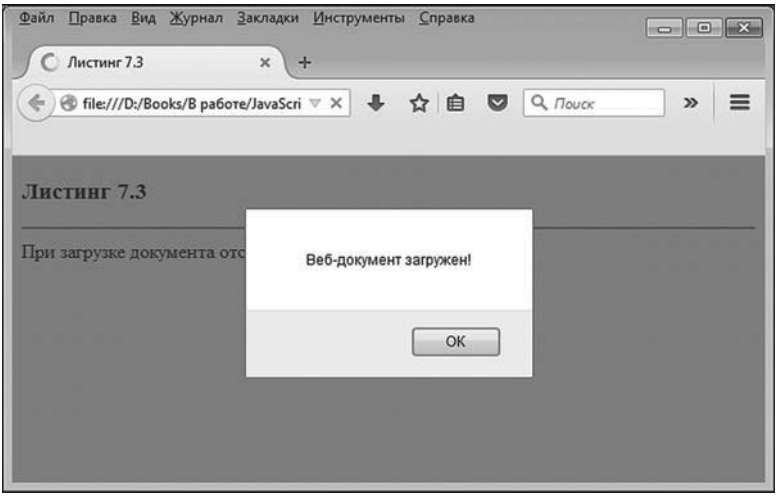

Рис. 7.8. При загрузке веб-документа в окно браузера выполняется код, отображающий диалоговое окно с сообщением

Более детально методы обработки событий рассматриваются немного позже.

Объект окна window

Отдайте его моим бармалейчикам.

из к/ф «Усатый нянь»

Мы знаем, что язык JavaScript поддерживает концепцию объектно-ориентированного программирования (ООП). Мы также уже имели дело со встроенными объектами, равно как и создавали собственные объекты. Когда речь заходит о работе с веб-документом, то на сцену выходит новый класс объектов — это объекты, связанные с окном браузера и рабочим документом. Фактически для успешной работы с веб-документом необходимо знать, какие с ним связаны объекты и что с данными объектами можно делать.

Объектные модели

Для понимания принципов реализации JavaScript-сценариев при работе с веб-документами необходимо обозначить некоторые концептуальные моменты. Дело в том, что основы JavaScript мы уже разобрали. Теперь нам нужно понять, как собственно сценарий на языке

JavaScript «привязывается» к конкретному веб-документу. И здесь важно хотя бы в общих чертах понимать, что происходит, когда веб-документ открывается в браузере, и как выполнение сценария влияет на процесс отображения документа.

Итак, при загрузке веб-документа в окно браузера на основе HTML-кода документа создается определенная структура. Элементами структуры являются объекты, соответствующие элементам в документе. А вся конструкция обычно упоминается как *объектная модель документа* (сокращенно DOM от *Document Object Model*). С помощью сценариев можно влиять на объектную модель документа как в плане изменения характеристик отдельных объектов, так и в плане изменения всей структуры.

НА ЗАМЕТКУ

Если сценарий вносит изменения в объектную модель документа, то эти изменения затрагивают модель, которая хранится в памяти. Исходный HTML-код документа при этом не меняется. Поэтому при перезагрузке документа все возвращается к исходному состоянию, которое определяется HTML-кодом документа.

В большинстве современных браузеров принципы реализации объектной модели достаточно универсальны (хотя, конечно, у разных браузеров есть свои особенности). Это важный момент при написании сценариев, ведь при создании кода важно знать, что он корректно выполняется в разных браузерах.

НА ЗАМЕТКУ

Стандарт для объектной модели, реализуемой браузерами, разработан консорциумом W3C (полное название — *World Wide Web Consortium*). Данная организация занимается разработкой и внедрением интернет-стандартов. Такие стандарты являются ориентиром для разработчиков программного обеспечения. В частности, это касается и разработки браузеров. В большинстве наиболее популярных и востребованных браузеров реализуется (в основном) стандарт W3C DOM для объектной модели документа. Хотя это и не снимает проблему универсальности кода: очень часто при написании сценариев приходится предусматривать проверку на предмет того, поддерживаются ли браузером те или иные свойства и методы. Что касается самого стандарта W3C DOM, то он определяет объекты, входящие в модель, а также их свойства и методы.

Кроме объектной модели документа еще явно выделяют *объектную модель браузера* (сокращенно BOM от *Browser Object Model*). Фактически объектная модель браузера — это все, что не входит в объектную модель документа.

Другими словами, объектная модель браузера представляет собой структуру из объектов, через которую реализуется доступ собственно к свойствам браузера. В принципе у каждого браузера объектная модель существенно своя. Тем не менее имеются некоторые общие «точки соприкосновения».

Далее рассмотрим особенности объекта window. Это объект рабочего окна, и мы его уже упоминали (и использовали — явно или неявно) в предыдущих главах книги. Здесь данный объект окажется предметом более пристального анализа.

Диалоговые окна

У объекта window есть методы (мы их, собственно, уже использовали), позволяющие отображать в рабочем окне диалоговые окна.

- Методом alert() отображается диалоговое окно с сообщением, которое передается аргументом методу. Окно содержит кнопку **ОК**, по которой следует щелкнуть, чтобы окно закрылось.
- Методом confirm() отображается диалоговое окно с сообщением (текст сообщения передается аргументом методу), а само окно содержит кнопки **ОК** и **Cancel (Отмена)**. Методом возвращается результат: если пользователь щелкает по кнопке **ОК**, то результатом метода возвращается значение true. Если пользователь щелкает по кнопке **Cancel (Отмена)**, методом возвращается значение false.
- Методом prompt() отображается диалоговое окно с полем ввода. Текст, отображаемый над полем ввода, передается аргументом методу. У диалогового окна две кнопки: при щелчке по кнопке **ОК** возвращается содержимое текстового поля, а при щелчке по кнопке **Cancel (Отмена)** возвращается значение null.

> **НА ЗАМЕТКУ**
>
> При вызове методов alert(), confirm() и prompt() объект window можно не указывать. Мы так поступали ранее. Далее объект window будет указываться в явном виде.

Небольшой пример с использованием перечисленных выше методов приведен в листинге 7.4.

Листинг 7.4. Диалоговые окна (файл Listing07_04.html)

```
<!DOCTYPE HTML>
<html>
<head>
<title>Листинг 7.4</title>
</head>
<body><h3>Листинг 7.4</h3><hr>
  <b>Выполнение команд.</b><br>
  <!-- Начало сценария -->
  <script type="text/javascript">
    // Оператор цикла:
    while(window.confirm("Готовы ввести команду?")){
      var res
      // Отображение окна с полем ввода:
      res=window.prompt("Введите команду")
      if(res!=null){
        // Отображение результата выполнения команды:
        document.write(res+" = "+eval(res)+"<br>")
      }
      else{
        // Завершение оператора цикла:
        break
      }
    }
    // Отображение окна с сообщением:
    window.alert("Выполнение сценария завершено")
  </script>
  <!-- Завершение сценария -->
</body>
</html>
```

Сценарий размещен в <script>-блоке. Основу сценария составляет оператор цикла while, в котором проверяемым условием указано выражение

window.confirm("Готовы ввести команду?"). Здесь уместно напомнить, что результатом метода возвращается значение true, если пользователь щелкает по кнопке **ОК**, и false, если пользователь щелкает по кнопке **Cancel** (**Отмена**). Поэтому каждый раз при проверке условия отображается диалоговое окно подтверждения, и для продолжения выполнения оператора цикла необходимо каждый раз щелкать в окне по кнопке **ОК**. Далее в теле оператора цикла командой res=window.prompt("Введите команду") отображается окно с полем ввода. Результат (текст в поле ввода или значение null) записывается в переменную res. Значение переменной проверяется в условном операторе, и, если оно отлично от значения null, выполняется команда document.write(res+" = "+eval(res)+"
"), которой отображается введенная пользователем команда и ее результат (значение соответствующего выражения). В противном случае командой break завершается выполнение оператора цикла.

По завершении выполнения сценария командой window.alert("Выполнение сценария завершено") отображается диалоговое окно с сообщением о завершении выполнения сценария.

На рис. 7.9 показано, как выглядит окно браузера при загрузке в него данного веб-документа.

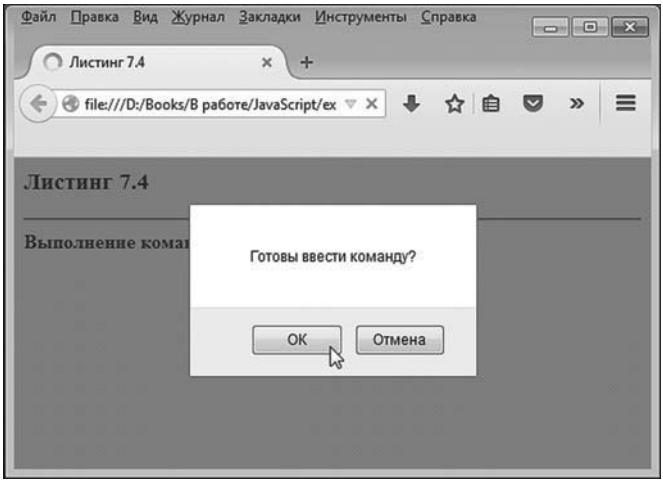

Рис. 7.9. Веб-документ в самом начале выполнения сценария

После щелчка по кнопке **ОК** появляется диалоговое окно с полем ввода, в котором указывается некоторое выражение (например, 2+3), как показано на рис. 7.10.

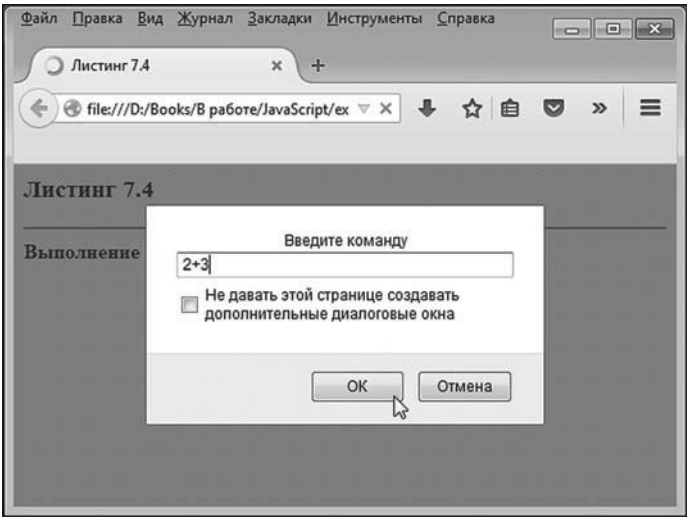

Рис. 7.10. Окно с полем для ввода команды

После ввода выражения снова появляется диалоговое окно с кнопками **ОК** и **Cancel** (**Отмена**).

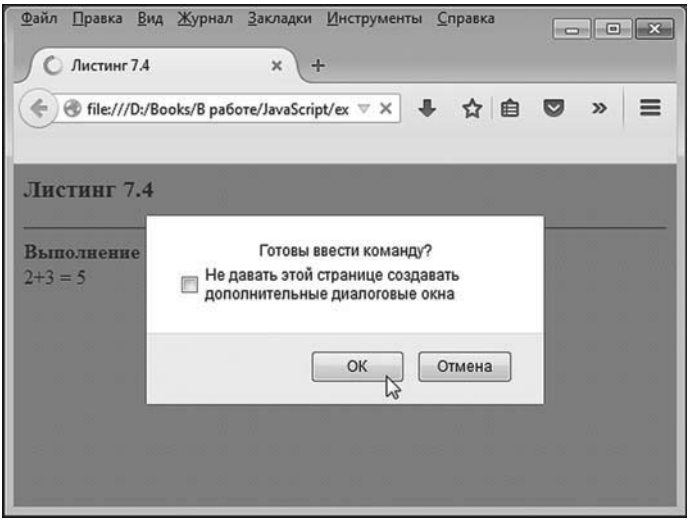

Рис. 7.11. Окно для подтверждения ввода новой команды

Если пользователь щелкает по кнопкеё **ОК**, появляется окно с полем ввода, в которое вводится команда (скажем, `5*3/2`). Ситуацию иллюстрирует рис. 7.12.

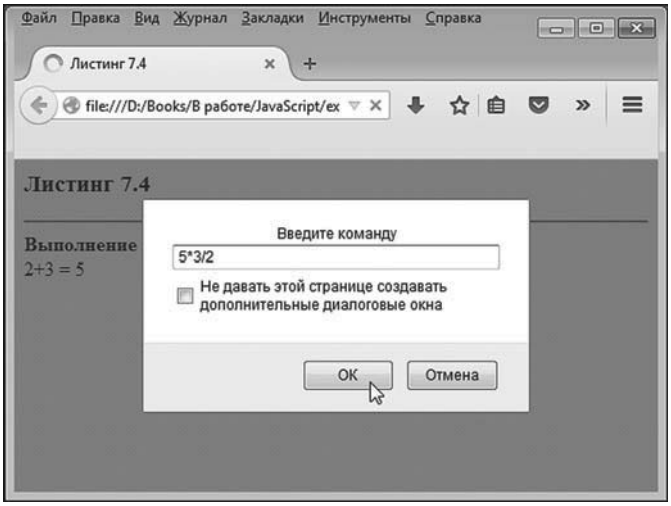

Рис. 7.12. Ввод очередной команды в поле в диалоговом окне

Опять появляется окно подтверждения. На рис. 7.13 представлена ситуация, когда в окне подтверждения пользователь щелкает по кнопке **Отмена**.

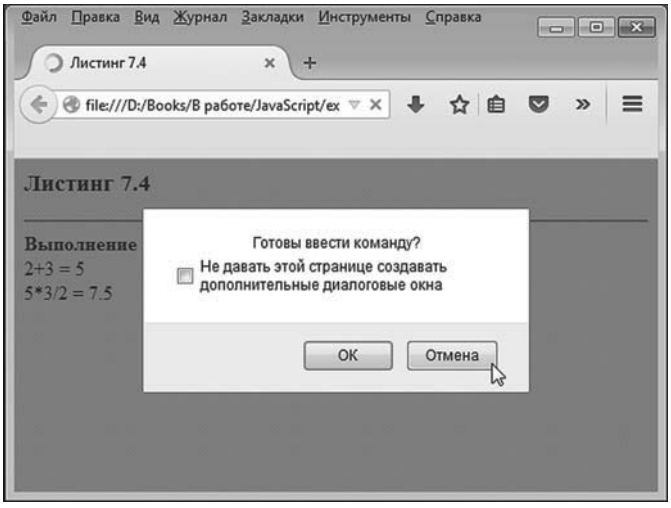

Рис. 7.13. Завершение ввода команд

На этом выполнение оператора цикла завершается и появляется диалоговое окно с сообщением о завершении выполнения сценария, как проиллюстрировано на рис. 7.14.

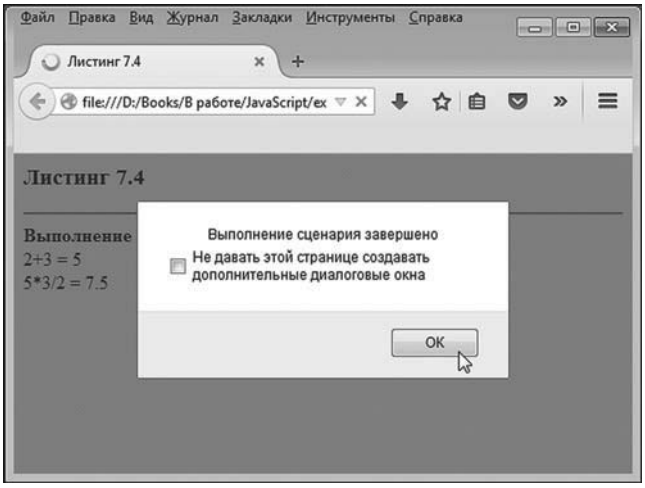

Рис. 7.14. Диалоговое окно появляется перед завершением работы сценария

НА ЗАМЕТКУ

Если в окне с полем ввода щелкнуть по кнопке **Отмена**, получим такой же результат — работа оператора цикла завершается, после чего появляется диалоговое окно с сообщением о завершении работы сценария.

На рис. 7.15 показано, как может выглядеть рабочий документ после выполнения сценария.

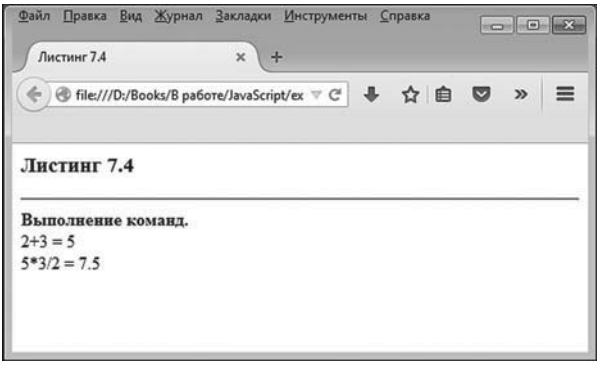

Рис. 7.15. Вид документа после завершения работы сценария

Легкость создания диалоговых окон многих завораживает. Но все же «общение» с пользователем через диалоговые окна выглядит как

минимум, не очень эстетично поэтому данный способ программирования — не самый лучший.

Открытие и закрытие окна

Программными методами можно открывать и закрывать окна. Для открытия окна используют метод open() объекта window, а закрыть окно можно с помощью метода close(), который вызывается из объекта закрываемого окна.

> **НА ЗАМЕТКУ**
>
> В принципе имеет значение не только наше желание открыть или закрыть окно, но и обстоятельства, при которых мы пытаемся выполнить соответствующую операцию. Многое зависит от типа браузера и его настроек. Обычно закрыть можно только то окно, которое было открыто сценарием.

Аргументом методу open() передается текстовая строка с URL-адресом документа, который следует открыть (или полным путем к файлу на компьютере).

Далее мы рассмотрим небольшой пример, в котором программными методами реализуется операция по открытию и закрытию окна. Более конкретно, мы создадим документ с двумя гиперссылками. Щелчок по одной из гиперссылок приводит к открытию окна с определенным веб-документом (который создан заранее).

Данный веб-документ будем называть *вспомогательным*. Щелчок по другой гиперссылке приводит к закрытию окна с документом. Но это еще не все. Если вспомогательный документ уже открыт (щелчком по гиперссылке), то щелчок по гиперссылке для открытия документа приведет к появлению диалогового окна с сообщением о том, что документ уже открыт.

Если вспомогательный документ закрыт (щелчком по гиперссылке или по системной пиктограмме), то щелчок по гиперссылке для закрытия окна приведет к появлению диалогового окна с сообщением о том, что документ уже закрыт. В общих чертах такова функциональность разрабатываемого веб-документа. А теперь рассмотрим программный код, представленный в листинге 7.5.

Листинг 7.5. Открытие и закрытие окна (файл Listing07_05.html)

```
<!DOCTYPE HTML>
<html><head><title>Листинг 7.5</title>
<!-- Начало сценария -->
<script type="text/javascript">
  // Функция для открытия документа
  // в новом окне (вкладке):
  function openWindow(){
    // Проверяем, открыто ли окно с документом:
    if(window.mywindow&&!mywindow.closed){
      // Отображается диалоговое окно с сообщением:
      window.alert("Документ уже открыт")
    }
    else{
      // Открывается новое окно с документом:
      window.mywindow=window.open("myhello.html")
    }
  }
  // Функция для закрытия окна с документом:
  function closeWindow(){
    // Проверяем, открыто ли окно с документом:
    if(window.mywindow&&!mywindow.closed){
      // Закрывается окно с документом:
      mywindow.close()
      // "Обнуляется" ссылка на окно:
      window.mywindow=null
    }
    else{
      // Отображается диалоговое окно с сообщением:
      window.alert("Окно уже закрыто")
    }
  }
</script>
```

```
<!-- Завершение сценария -->
</head>
<body><h3>Листинг 7.5</h3><hr>
  <b>Открытие и закрытие окна</b><br>
  <a href="javascript:openWindow()">Открыть файл myhello.html</a><br>
  <a href="javascript:closeWindow()">Закрыть окно с файлом myhello.html</a>
</body>
</html>
```

Сценарий содержится непосредственно в веб-документе в <script>-блоке, который, в свою очередь, находится в <head>-блоке документа.

Сценарий состоит из описания двух функций, поэтому при его загрузке в браузер никакие «активные» действия не выполняются — функции загружаются в память, и их можно использовать (но команды на использование нет). Вызываются функции при щелчке по гиперссылкам, которые размещены с помощью стандартного HTML-кода в основной части документа.

Функция openWindow() предназначена для открытия документа (файл с названием myhello.html) в новом рабочем окне (на самом деле это будет новая вкладка в окне браузера).

Чтобы понять логику выполнения кода функции, разумно проанализировать общий подход. А общий подход состоит в том, что если открывается новое окно, то ссылка на объект этого окна записывается в свойство mywindow объекта window того окна, в котором вызывается код.

> **НА ЗАМЕТКУ**
>
> Свойства mywindow у объекта window нет, поэтому его необходимо создать. Свойство создается присваиванием ему значения, но нужно иметь в виду, что при проверке значения свойства mywindow его может просто не существовать.

Перед тем как открыть окно, проверяется, во-первых, наличие свойства mywindow у объекта окна, а во-вторых, что окно не было закрыто. Последнее обстоятельство проверяется с помощью свойства closed объекта mywindow.

НА ЗАМЕТКУ

Свойство mywindow объекта window является ссылкой на объект окна, поэтому на практике с mywindow можно обращаться как с объектом — в частности, у данного объекта есть свойства, среди которых свойство closed, равное true, если соответствующее окно закрыто, и false — в противном случае.

Итак, в теле функции в условном операторе проверяется условие window.mywindow&&!mywindow.closed, которое истинно при одновременном выполнении двух условий:

- свойство window.mywindow существует;
- окно, на которое ссылается свойство mywindow, не закрыто.

ДЕТАЛИ

Существование свойства window.mywindow на самом деле сводится к тому, что оно отлично от значений null (пустая ссылка — свойство существует, но не ссылается на объект) и undefined (значение свойства не определено — свойство фактически не существует).

Вторая часть условия в выражении window.mywindow&&!mywindow.closed вычисляется только в том случае, если операнд window.mywindow истинен (а это означает, что свойство window.mywindow как минимум существует и имеет значение, отличное от null). Значение выражения !mywindow.closed благодаря использованному в нем оператору логического отрицания ! равно true, если окно, на которое ссылается свойство mywindow, открыто, и false — в противном случае.

При истинном условии в условном операторе командой window.alert("Документ уже открыт") отображается диалоговое окно с сообщением о том, что рабочее окно с документом уже открыто. Если условие ложно, командой window.mywindow=window.open("myhello.html") открывается новое рабочее окно (вкладка) с документом myhello.html. Ссылка на объект нового окна записывается в свойство mywindow объекта window.

НА ЗАМЕТКУ

Таким образом, window — это объект текущего окна, в котором загружен HTML-код со сценарием, а mywindow — объект окна, в котором открыт файл myhello.html.

Также стоит заметить, что файл myhello.html должен быть в том же каталоге, что и файл со сценарием. В противном случае аргументом

метода open() указывается полный путь к файлу, который следует открыть в новом окне.

Функция closeWindow() для закрытия окна с документом состоит из условного оператора, в котором проверяется уже знакомое нам условие window.mywindow&&!mywindow.closed. При истинности условия (новое окно открыто) выполняется команда mywindow.close(), которой окно закрывается, после чего командой window.mywindow=null «обнуляется» ссылка на объект окна. Если окно закрыто или еще не открывалось (условие window.mywindow&&!mywindow.closed ложно), выполняется команда window.alert("Окно уже закрыто"), которой отображается диалоговое окно с сообщение о том, что окно закрыто. В теле веб-документа, кроме прочего, размещены две гиперссылки. Значением атрибута href первой гиперссылки указан текст "javascript:openWindow()", а значением атрибута href указан текст "javascript:closeWindow()". Поэтому при щелчке по первой гиперссылке вызывается функция openWindow(), а при щелчке по второй гиперссылке вызывается функция closeWindow().

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Гиперссылка создается с помощью дескрипторов <a> и . Текст, размещенный между данными дескрипторами, является текстом гиперссылки.

НА ЗАМЕТКУ

Использование в гиперссылках конструкций вида href="javascript:openWindow()" и href="javascript:closeWindow()" является идеей не самой блестящей. Причин к тому несколько. Например, такая ссылка становится нефункциональной, если в браузере пользователя отключена поддержка JavaScript. На практике следует избегать ситуаций, когда атрибут href в гиперссылке вместо реального адреса содержит нечто иное.

Код вспомогательного документа myhello.html, который открывается и закрывается в процессе работы с основным документом, представлен ниже:

```
<!DOCTYPE HTML>
<html>
<head>
<title>Файл myhello.html</title>
```

```
</head>
<body><h1>Файл <code>myhello.html</code></h1><hr>
<b>Этот файл открыт с помощью сценария...</b>
</body>
</html>
```

Как выглядит веб-документ со сценарием в самом начале, показано на рис. 7.16.

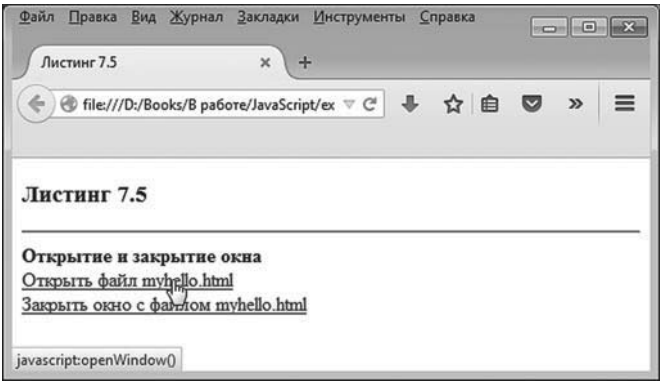

Рис. 7.16. Веб-документ с двумя гиперссылками для открытия и закрытия окна со вспомогательным документом

При щелчке по первой гиперссылке открывается новая вкладка с документом, реализованным через файл myhello.html. Результат выполнения такой операции представлен на рис. 7.17.

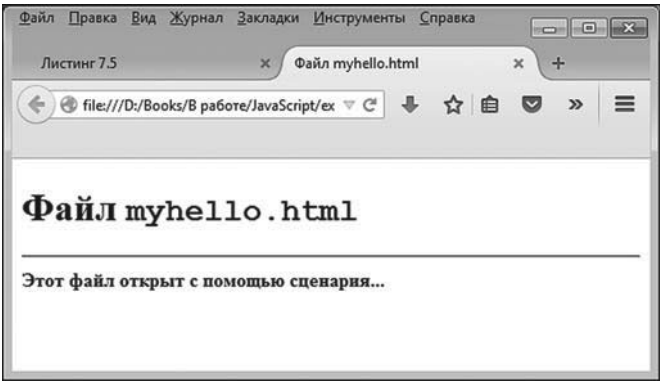

Рис. 7.17. В новом окне открыт документ *myhello.html*

Если еще раз попытаться щелкнуть по первой гиперссылке, получим результат, как на рис. 7.18.

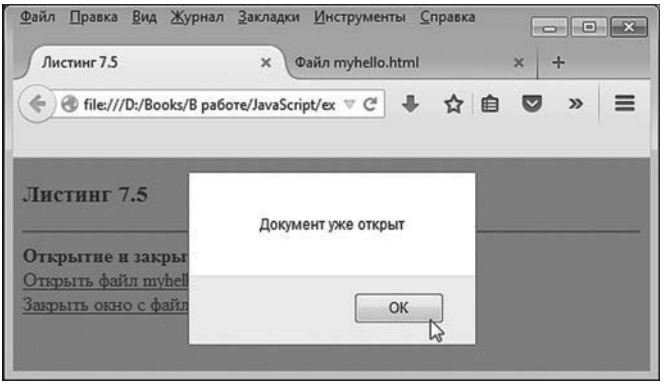

Рис. 7.18. При попытке открыть уже открытый документ появляется диалоговое окно

В данном случае еще одно окно не открывается, а вместо этого появляется диалоговое окно с сообщением, что соответствующий документ уже открыт. Если щелкнуть по второй гиперссылке, то окно документом из файла myhello.html будет закрыто. Ситуация проиллюстрирована на рис. 7.19 (вспомогательное окно закрыто, а курсор мыши наведен на вторую гиперссылку).

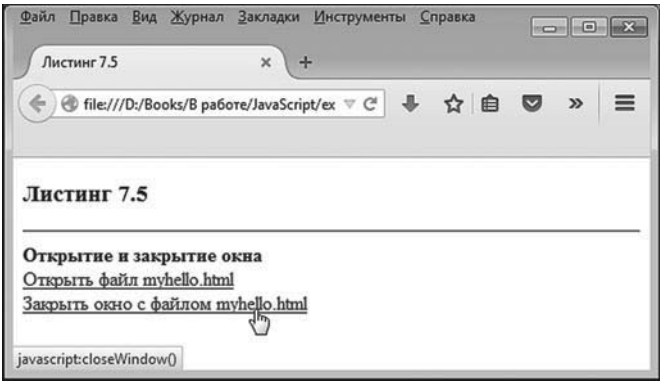

Рис. 7.19. Рабочий документ после закрытия вспомогательного окна с документом

Повторная попытка закрыть вспомогательное окно приводит к тому, что появляется окно с сообщением, как показано на рис. 7.20.

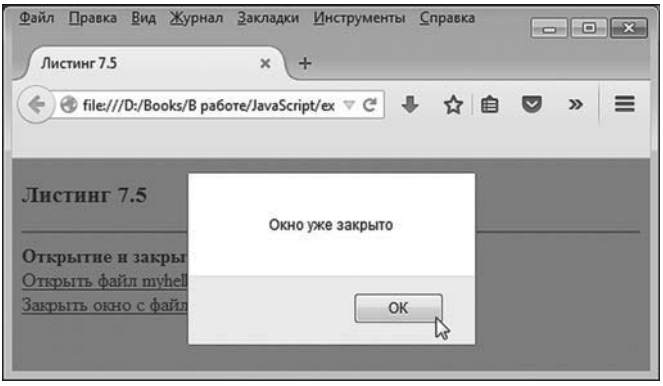

Рис. 7.20. При попытке закрыть окно, которое уже закрыто, появляется диалоговое окно

НА ЗАМЕТКУ

Следует отметить, что если мы открываем вспомогательное окно через гиперссылку, а затем закрываем его (не через гиперссылку), то попытка закрыть окно через гиперссылку приведет к появлению сообщения о том, что окно уже закрыто. Такое же диалоговое окно появится, если мы попытаемся закрыть (через гиперссылку) еще не открытое окно. С другой стороны, если открыть файл myhello.html «вручную» (не через гиперссылку), то при щелчке по гиперссылке открытия окна будет открыто еще одно окно. Причина в том, что в сценарии операции со вспомогательным окном реализуются через свойство mywindow, которое создается в сценарии, и его свойства в основном регулируются через сценарий. А сценарий «вступает в игру» при щелчке по гиперссылкам.

Загрузка документа

У объекта окна есть свойство location, которое является ссылкой на объект Location, содержащий информацию об URL-адресе документа, загруженного в рабочем окне. Более конкретно, за адрес документа отвечает свойство href объекта Location. Вместе с тем автоматическое приведение объекта Location к текстовому формату (через метод toString()) реализовано так, что значением возвращается адрес, записанный в свойство href. Поэтому (и учитывая приведенные далее обстоятельства) напрямую свойство href используется не очень часто.

Какой именно документ загружен в рабочем окне браузера, определяется значением свойства window.location.href. Изменение данного свойства

приводит к смене документа, загруженного в рабочем окне. Проще говоря, чтобы программными методами загрузить в рабочем окне (текущем, не новом!) новый документ, достаточно изменить значение свойства window.location.href.

Для изменения свойства window.location.href из объекта window.location вызывается метод assign(). Аргументом методу передается адрес того документа, который должен быть загружен в текущем рабочем окне. Пример выполнения программной загрузки в текущем рабочем окне нового документа приведен в документе из листинга 7.6.

Листинг 7.6. Загрузка документа (файл Listing07_06.html)

```
<!DOCTYPE HTML>
<html><head><title>Листинг 7.6</title>
<!-- Начало сценария -->
<script type="text/javascript">
  // Функция для загрузки документа
  // (адрес аргумента - аргумент функции):
  function doIt(addr){
    window.location.assign(addr)
  }
</script>
<!-- Завершение сценария -->
</head>
<body><h3>Листинг 7.6</h3><hr>
  <b>Загрузка документа</b><br><br>
  <!-- Поле ввода -->
  <input type="text" id="myinput" size="40"><br><br>
  <!-- Кнопка -->
  <button onclick="doIt(myinput.value)">Загрузить</button>
</body>
</html>
```

Здесь представлен HTML-код со сценарием. В сценарии описана функция, назначение которой обсуждается далее. Сам документ содержит поле ввода и кнопку. В поле ввода указывается адрес документа,

который должен быть загружен. Щелчок по кнопке приводит к тому, что документ по соответствующему адресу загружается в окно.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Элементы для ввода значений (и, в частности, поле ввода) создаются с помощью дескриптора <input> (закрывающего дескриптора нет). Атрибут type определяет тип элемента: полю ввода соответствует значение "text". Атрибут size определяет размер (ширину) поля ввода. Атрибут id является идентификатором, позволяющим однозначно выделить поле среди прочих объектов в документе. Значение данного атрибута является произвольным (определяется пользователем).
Кнопка создается с помощью дескрипторов <button> и </button>. Текст, указанный между данными дескрипторами, является названием, отображаемым на кнопке. Атрибут onclick, указанный в дескрипторе <button>, определяет обработчик события, которое связано со щелчком по кнопке. Значение атрибута onclick — текст с программным кодом, который выполняется, когда пользователь щелкает по кнопке в рабочем окне.
Также следует отметить, что обычно такие элементы управления, как поля ввода и кнопки, размещаются внутри *формы*. Форма выделяется дескрипторами <form> и </form>. Главное преимущество использования формы — возможность переслать настройки элементов управления формы на сервер. Однако в рассматриваемом примере необходимости в таком действии нет, поэтому нет и необходимости использовать форму.

НА ЗАМЕТКУ

Для удобства восприятия в документ между элементами (поле ввода и кнопка) добавлены пустые строки (в HTML-коде использованы двойные инструкции перехода к новой строке).

В сценарии описана функция doIt(). Предполагается, что у функции один аргумент (обозначен как addr), который представляет собой адрес документа, предназначенного для загрузки в текущем рабочем окне. Данная задача решается с помощью команды window.location.assign(addr) в теле функции. Таким образом, для загрузки нового документа в рабочее окно достаточно вызвать функцию doIt() и передать ей аргументом адрес документа.

Функция doIt() использована нами в инструкции onclick="doIt(myinput.value)", которая, в свою очередь, размещена в дескрипторе <button>, за-

действованном при создании кнопки. Атрибутом onclick определяется обработчик события щелчка по кнопке. Таким образом, при щелчке по кнопке будет выполняться команда doIt(myinput.value). Аргументом функции doIt() передана ссылка myinput.value на значение в текстовом поле. Почему так? Все очень просто. При описании поля значением атрибута id указан текст "myinput". Данное обстоятельство означает, что для получения доступа к объекту поля можно использовать идентификатор myinput.

НА ЗАМЕТКУ

Способ «непосредственного» использования значения атрибута id для получения доступа к элементу не является рекомендуемым, хотя и поддерживается основными браузерами. Имеется в виду ситуация, когда значение атрибута id служит ссылкой на объект соответствующего элемента. Для получения доступа к элементу по его атрибуту id рекомендуется использовать метод getElementById() объекта документа document. Аргументом методу передается текстовое значение атрибута id элемента. Результатом метод возвращает ссылку на объект элемента. Для атрибута onclick кнопки в рассматриваемом примере вместо значения "doIt(myinput.value)" разумнее было бы использовать значение "doIt(document.getElementById('myinput').value)". Далее в книге мы будем в основном вызывать метод getElementById() для получения доступа к объектам элементов. Вместе с тем большое количество созданных ранее сценариев используют напрямую значение атрибута id. Хороший программист, разумеется, должен уметь «читать» и такие коды. Поэтому в рассматриваемом и одном из следующих примеров как иллюстрацию мы задействовали означенный «нерекомендуемый» стиль.

Понятно, что атрибут id можно использовать для идентификации объектов не только для полей ввода, но и для других элементов документа.

У объекта поля имеется свойство myinput.value, значением которого является текст, содержащийся в поле ввода (на момент запроса значения свойства value). Поэтому выражение myinput.value — это на самом деле текст в поле ввода. Следовательно, при щелчке по кнопке в окно будет загружаться документ, адрес которого содержится в поле ввода. На рис. 7.21 показано, как выглядит веб-документ со сценарием в окне браузера.

В данном случае в поле ввода введено название myhello.html файла с документом, который следует загрузить в окно браузера (речь идет

о том же файле myhello.html, который рассматривался в предыдущем примере). Что, собственно, и происходит при щелчке пользователя по кнопке **Загрузить**.

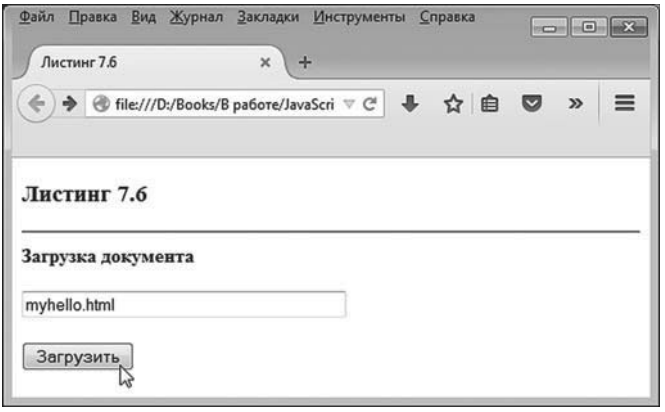

Рис. 7.21. Документ перед загрузкой нового документа: в поле ввода указано название загружаемого файла

НА ЗАМЕТКУ

Поскольку в данном случае указано только имя файла myhello.html (без полного пути к нему), то данный файл должен находиться в той же папке, что и файл исходного веб-документа со сценарием.

На рис. 7.22 показано рабочее окно браузера после загрузки в него файла myhello.html.

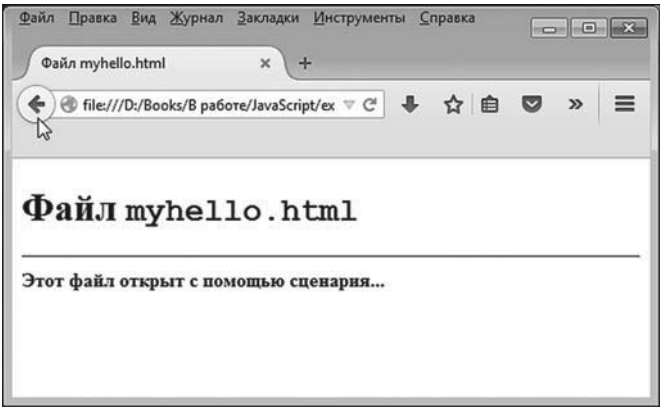

Рис. 7.22. Результат загрузки файла с локального диска

Если щелкнуть по кнопке обратного перехода (перехода к предыдущему документу), вернемся к документу со сценарием. На рис. 7.6 показано соответствующее окно, но на этот раз в поле ввода введен адрес http://www.vasilev.kiev.ua веб-страницы в сети, которую следует открыть в окне браузера. Результат загрузки веб-документа показан на рис. 7.24.

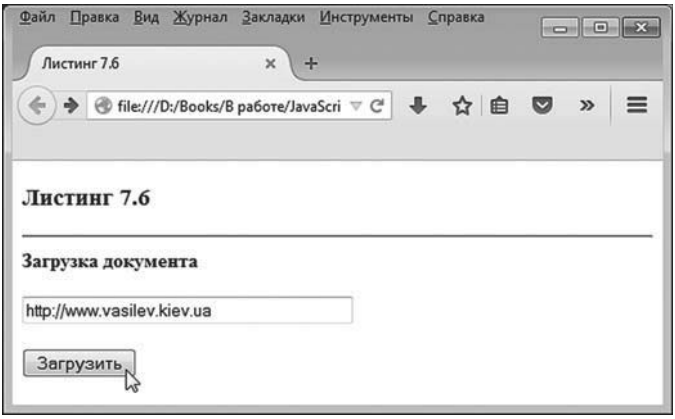

Рис. 7.23. Документ перед загрузкой нового документа: в поле ввода указан адрес веб-страницы в сети

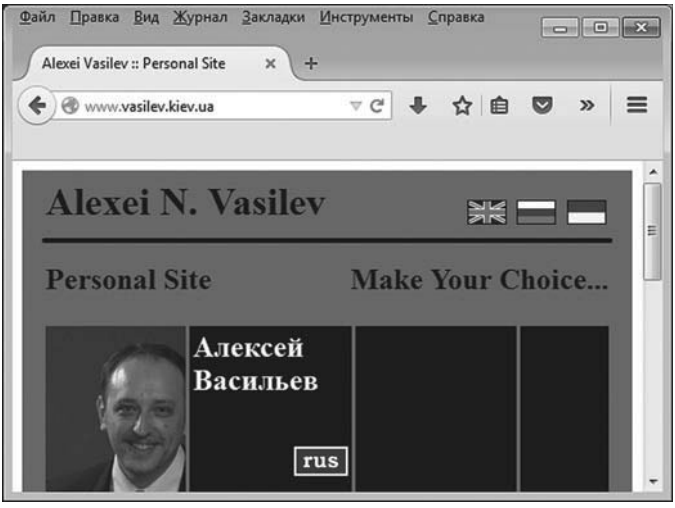

Рис. 7.24. Результат загрузки веб-страницы из сети

Как видим, загружаются как документы с компьютера клиента, так и страницы из сети.

ДЕТАЛИ

Стоит заметить, что вместо вызова метода assign() мы могли воспользоваться простым присваиванием значения свойству location или location.href. Другими словами, вместо команды window.location.assign(addr) в теле функции doIt() могла бы быть команда window.location=addr или window.location.href=addr.

Также альтернативой к методу assign() является метод replace(). Соответственно, вместо команды window.location.assign(addr) можно было использовать команду window.location.replace(addr). Однако здесь уже имеются некоторые различия. В частности, загрузка документа с помощью метода replace() не только, собственно, загружает документ в окно браузера, но еще и удаляет исходный документ (документ со сценарием) из истории загрузки веб-документов. Как следствие, мы не сможем с помощью кнопки возврата перейти от вновь загруженного документа к исходному документу.

Свойства и методы объекта window

Кроме нескольких рассмотренных выше, у объекта window много и других свойств и методов, позволяющих получить доступ к основным параметрам браузера или выполнять некоторые важные операции. Кое-что в данном случае зависит от используемого браузера, поскольку ряд свойств и методов объекта window являются браузер-зависимыми (поддерживаются одними браузерами и не поддерживаются другими). Тем не менее базовые свойства и методы для наиболее популярных браузеров универсальны.

В принципе, чтобы узнать, какие свойства и методы поддерживаются тем или иным браузером, можно обратиться к справке для этого браузера. Но если имеется возможность воспользоваться браузером, так сказать, непосредственно, то имеет смысл задействовать несложный код, выполнение которого приводит к отображению свойств и методов объекта window. Простая версия такого кода представлена в листинге 7.7.

Листинг 7.7. Свойства и методы объекта window (файл Listing07_07.html)

```
<!DOCTYPE HTML>
<html><head><title>Листинг 7.7</title>
</head>
<body><h3>Листинг 7.7</h3><hr>
  <b>Свойства и методы объекта <code>window</code></b><br>
```

```
  <!-- Начало сценария -->
  <script type="text/javascript">
    // Индексная переменная:
    var k=1
    // Отображение свойств и методов:
    for(var prop in window){
      // Отображение свойства или метода:
      document.write("<b>"+k+"</b>: "+prop+"<br>")
      // Новое значение индексной переменной:
      k++
    }
  </script>
  <!-- Завершение сценария -->
</body>
</html>
```

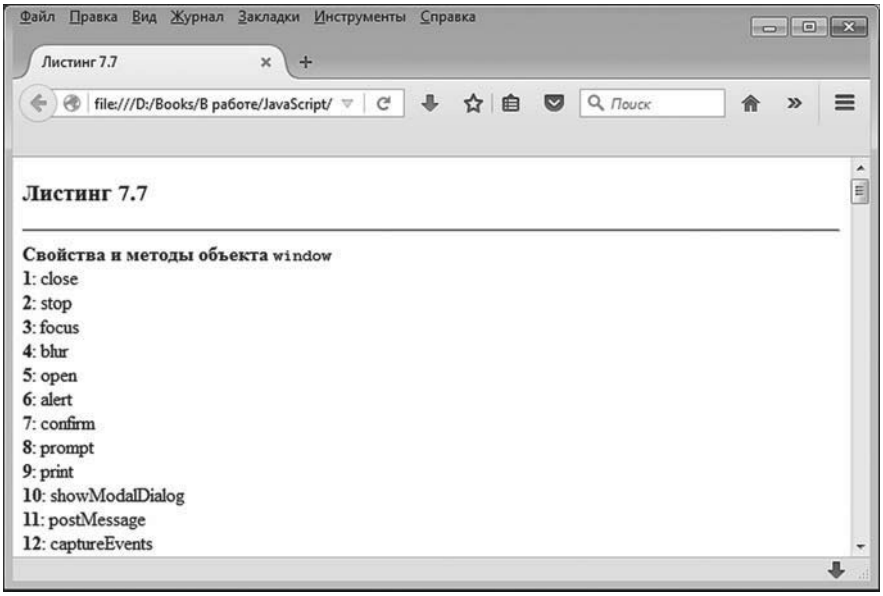

Рис. 7.25. Документ открыт в браузере Mozilla Firefox

В сценарии, содержащемся в HTML-документе, выполняется оператор цикла, в котором переменная prop пробегает значения, являющиеся названиями свойств и методов объекта window.

Каждое название отображается в отдельной строке, а для удобства выполняется нумерация строк (с помощью индексной переменной k). Результат выполнения сценария зависит от того, в каком браузере открывается документ.

На рис. 7.25 показано окно браузера Mozilla Firefox с открытым в нем документом.

Тот же документ, но открытый в браузере Internet Explorer, показан на рис. 7.26.

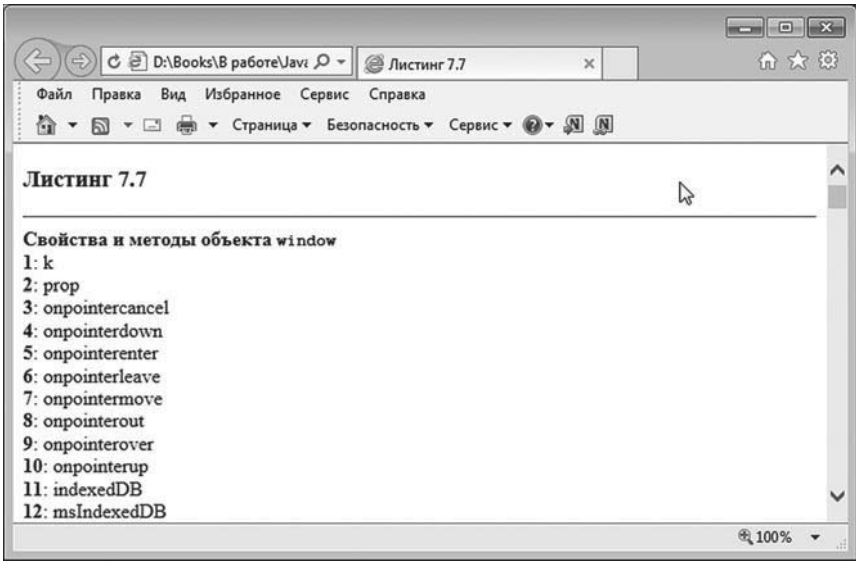

Рис. 7.26. Документ открыт в браузере Internet Explorer

В принципе, кроме разного порядка перечисления свойств и методов, отличия имеются и в самих списках. Как отмечалось выше, некоторые свойства и методы являются уникальными для того или иного браузера, или наоборот — не поддерживаются в некоторых браузерах. Здесь и далее, если не будет особой необходимости, мы будем использовать и обсуждать лишь те свойства и методы, которые поддерживаются в наиболее популярных на момент написания книги браузерах (в частности, это браузеры Mozilla Firefox, Internet Explorer, Opera и Google Chrome).

В табл. 7.1 перечислены свойства объекта window, которые поддерживаются (в той или иной степени) перечисленными выше браузерами.

Таблица 7.1. Свойства объекта window

Свойство	Свойство	Свойство
applicationCache	length	performance
closed	localStorage	screen
console	location	screenX
devicePixelRatio	name	screenY
document	navigator	self
frameElement	opener	sessionStorage
frames	outerHeight	status
history	outerWidth	top
indexedDB	pageXOffset	window
innerHeight	pageYOffset	
innerWidth	parent	

Методы объекта window перечислены в табл. 7.2.

Таблица 7.2. Методы объекта window

Метод	Метод
addEventListener()	moveBy()
alert()	moveTo()
atob()	open()
blur()	postMessage()
btoa()	print()
cancelAnimationFrame()	prompt()
captureEvents()	releaseEvents()
clearInterval()	removeEventListener()
clearTimeout()	requestAnimationFrame()
close()	resizeBy()
confirm()	resizeTo()
dispatchEvent()	scroll()
focus()	scrollBy()
getComputedStyle()	scrollTo()
getSelection()	setInterval()
matchMedia()	setTimeout()

Также следует учесть, что для объекта окна window могут обрабатываться события. Обработчики событий объекта window (свойства, значениями которым присваиваются функции-обработчики) представлены в табл. 7.3.

Таблица 7.3. Обработчики событий для объекта window

Обработчик	Обработчик	Обработчик
onabort	onhashchange	onpause
onbeforeunload	oninput	onplay
onblur	onkeydown	onplaying
oncanplay	onkeypress	onpopstate
oncanplaythrough	onkeyup	onprogress
onchange	onload	onratechange
onclick	onloadeddata	onreset
oncontextmenu	onloadedmetadata	onresize
ondblclick	onloadstart	onscroll
ondrag	onmessage	onseeked
ondragend	onmousedown	onseeking
ondragenter	onmouseenter	onselect
ondragleave	onmouseleave	onstalled
ondragover	onmousemove	onsubmit
ondragstart	onmouseout	onsuspend
ondrop	onmouseover	ontimeupdate
ondurationchange	onmouseup	onunload
onemptied	onoffline	onvolumechange
onended	ononline	onwaiting
onerror	onpagehide	
onfocus	onpageshow	

Небольшой пример, в котором используются свойства объекта window, представлен ниже. В частности, в примере использованы:

- свойство innerWidth — определяет ширину (в пикселях) внутренней рабочей области окна);
- свойство innerHeight — определяет высоту (в пикселях) внутренней рабочей области окна);
- свойство outerWidth — определяет ширину (в пикселях) окна браузера;
- свойство outerHeight — определяет высоту (в пикселях) окна браузера;
- свойство screenX — определяет горизонтальную координату окна браузера;
- свойство screenY — определяет вертикальную координату окна браузера.

НА ЗАМЕТКУ

Координаты элемента (или, как в данном случае, окна браузера) определяются по отношению к левому верхнему углу контейнера, в котором содержится элемент. Для окна браузера контейнером выступает экран, поэтому координаты окна определяются по отношению к левому верхнему углу экрана. При этом задаются координаты для левого верхнего угла элемента. В данном конкретном случае, когда речь идет об определении координат окна браузера, то такими координатами является расстояние (в пикселях) вдоль горизонтали от левого верхнего угла экрана до левого верхнего угла окна и расстояние вдоль вертикали от левого верхнего угла экрана до левого верхнего угла окна.

Документ, который мы используем, имеет код, представленный в листинге 7.8.

Листинг 7.8. Использование некоторых свойств объекта window (файл Listing07_08.html)

```
<!DOCTYPE HTML>
<html><head><title>Листинг 7.8</title>
<!-- Начало сценария -->
<script type="text/javascript">
  // Функция для вычисления параметров окна:
  function showParameters(){
    // Текст для отображения в окне:
    var txt="Внутренние ширина и высота: "+window.innerWidth
    txt+=" и "+window.innerHeight+"<br>"
    txt+="Внешние ширина и высота: "+window.outerWidth
    txt+=" и "+window.outerHeight+"<br>"
    // Отображение текста в окне:
    mytext.innerHTML=txt
  }
</script>
<!-- Завершение сценария -->
</head>
<body onload="showParameters()" onresize="showParameters()">
```

```
  <h3>Листинг 7.8</h3><hr>
  <b>Размеры окна</b><br>
  <!-- Текст -->
  <p id="mytext">
  </p>
  <!-- Кнопка -->
  <button onclick="alert('Координаты: '+window.screenX+' и '+window.screenY)">
    Показать координаты окна
  </button>
</body>
</html>
```

Как выглядит документ, будучи открытым в окне браузера, показано на рис. 7.27.

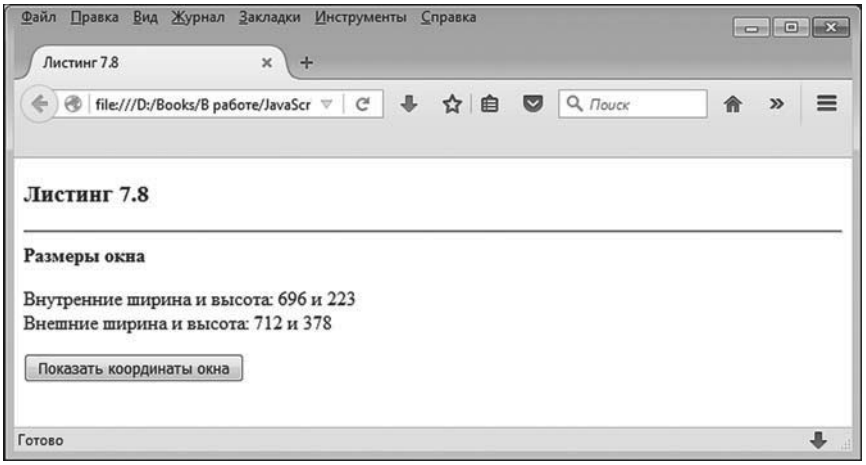

Рис. 7.27. Документ с кнопкой содержит информацию о размерах окна браузера

Документ содержит текст со значениями для ширины и высоты (внутренней и внешней) окна браузера, а также кнопку с названием **Показать координаты окна**.

При изменении размеров окна отображаемые в рабочей области окна значения его размеров (ширина и высота) изменяются автоматически. На рис. 7.28 показано данное окно, но с измененными размерами.

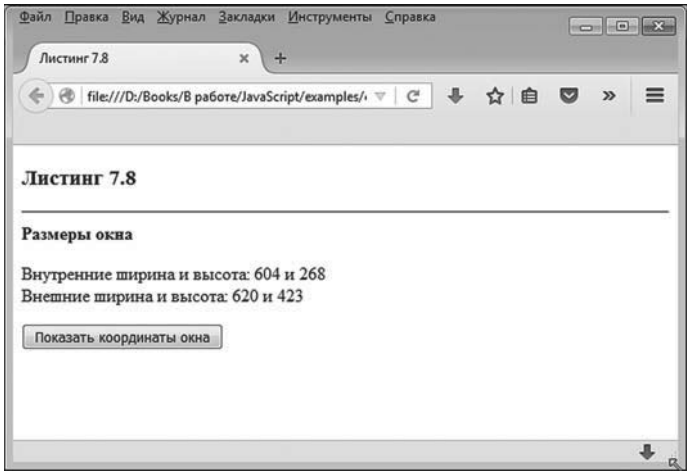

Рис. 7.28. При изменении размеров окна в области окна автоматически изменяются отображаемые числовые значения для ширины и высоты окна

НА ЗАМЕТКУ

Внутренние ширина и высота — это размеры рабочей области. Внешние ширина и высота — это размеры (внешние) окна браузера. Данные параметры связаны между собой и отличаются на величину, определяемую размерами внутренних элементов окна (например, полей и панелей между внешней стороной окна и собственно рабочей областью). Если при изменении размеров окна структура и расположение внутренних элементов не меняются, то внутренние ширина и высота отличаются от внешней ширины и соответственно высоты на постоянные значения.

При щелчке по кнопке **Показать координаты окна** появляется диалоговое окно с координатами окна браузера. Такая ситуация представлена на рис. 2.29.

Теперь проанализируем программный код (см. листинг 7.8), с помощью которого реализуется документ с описанными выше функциональными возможностями. Что касается общей структуры кода, то имеет смысл отметить два момента.

Во-первых, в <head>-блоке документа содержится сценарий с описанием функции showParameters().

Во-вторых, в документе в <body>-разделе используются обработчики событий onload (загрузка документа) и onresize (изменение размеров

окна), и при обработке соответствующих событий вызывается функция showParameters().

В-третьих, в документе имеется кнопка, для которой использован обработчик события onclick (щелчок по кнопке). Все это представляет интерес для нашего анализа.

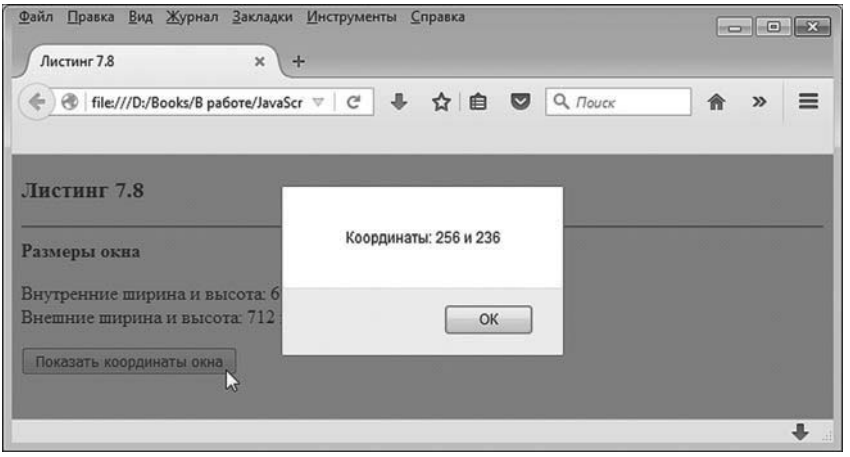

Рис. 7.29. При щелчке по кнопке открывается диалоговое окно с координатами окна браузера

Мы начнем с программного кода функции showParameters(), но прежде отметим важное обстоятельство: в <body>-блоке документа есть внутренний блок, выделенный дескрипторами и (тестовый абзац или блок). Концептуально важно то, что в дескрипторе для этого блока содержится инструкция id="mytext". Этой инструкцией для атрибута id элемента задается значение mytext. Поэтому через идентификатор mytext можно обращаться к данному -блоку, что, собственно, и происходит в сценарии. В частности, в теле функции showParameters() сначала формируется текстовое значение для переменной txt, а затем командой mytext.innerHTML=txt значение переменной txt «вставляется» в -блок с идентификатором mytext.

Формально в команде mytext.innerHTML=txt свойству innerHTML элемента с идентификатором mytext присваивается значением текст из переменной txt. Свойство innerHTML для -элемента отвечает за внутреннее содержимое этого элемента: через свойство innerHTML виртуально можно изменить текст (HTML-код), расположенный между дескрипторами и .

НА ЗАМЕТКУ

В документе -блок со значением mytext атрибута id описан с пустым «телом»: в документе между дескрипторами и ничего нет. То есть формально этот блок пустой. Если бы не было сценария, то в соответствующем блоке ничего бы не отображалось. Что же меняет наличие сценария? При загрузке документа в памяти создается объектная модель для документа. Эта модель создается в соответствии с кодом документа. Выполнение сценария приводит к тому, что в эту объектную модель вносятся изменения. В частности, при вызове функции showParameters() при выполнении команды mytext.innerHTML=txt в объектной модели документа происходят изменения: в -блок со значением mytext атрибута id добавляется «внутреннее содержимое». Но это «содержимое» добавляется на уровне объектной модели в памяти, а код документа остается неизменным.

Что касается механизма доступа к -блоку по значению атрибута id, то хочется напомнить, что в таких случаях рекомендуется использовать метод getElementById(). Например, в рассматриваемом сценарии в описании функции showParameters() можно было бы объявить переменную mytext и присвоить ей значение document.getElementById("mytext"). После этого переменная mytext будет ссылаться на объект -блока.

Как альтернатива, вместо команды mytext.innerHTML=txt в теле функции showParameters() можно использовать команду document.getElementById("mytext").innerHTML=txt и при этом не объявлять локальную переменную mytext.

Что касается собственно содержимого -блока, то значение переменной txt последовательно формируется несколькими командами, в которых, кроме прочего, используются инструкции window.innerWidth, window.innerHeight, window.outerWidth и window.outerHeight. В этих инструкциях выполняется обращение к свойствам объекта окна window.

Таким образом, при вызове функции showParameters() в рабочем документе отображается текст со значениями (текущими на момент вызова функции) внутренних и внешних размеров окна. Функция вызывается при загрузке документа и при изменении размеров окна. Для этого в дескрипторе <body> описаны атрибуты onload и onresize, через которые определяются обработчики соответственно для события, связанного с загрузкой документа, и события, связанного с изменением размеров окна с документом. Для каждого из атрибутов значением указано выражение "showParameters()", означающее вызов функции showParameters(). Кнопка в документе создается с помощью дескрипторов <button> и </button>. Текст (имеется в виду Показать координаты окна), размещенный между этими дескрипторами, отображается в области кнопки. В дескрипторе <button> описан

атрибут onclick, определяющий обработку события, связанного со щелчком по кнопке. Значением для атрибута onclick указана текстовая строка "alert('Координаты: '+window.screenX+' и '+window.screenY)". Текстовая строка содержит команду вызова метода alert(), которым открывается диалоговое окно с сообщением. Текст сообщения передается аргументом методу alert(). При формировании аргумента метода использованы инструкции window.screenX и window.screenY, с помощью которых определяются координаты окна.

ДЕТАЛИ

Кроме упомянутых выше свойств объекта window, существуют и другие достаточно часто используемые свойства. С некоторыми мы уже встречались, а с другими — нет. Имеет смысл напомнить или сообщить, что:

- значение свойства closed (доступно только для чтения) равно true, если окно закрыто, и false — в противном случае;
- значением свойства document является ссылка на документ, который содержится в окне;
- значением свойства frames является ссылка на коллекцию фреймов, используемых в окне;
- доступное только для чтения свойство history возвращает ссылку на объект History, который, в свою очередь, содержит информацию о веб-документах, загружавшихся ранее в окно браузера;
- свойство length позволяет определить количество фреймов в окне;
- с помощью свойства location получают доступ к объекту Location, содержащему сведения о размещении загруженного в окно документа;
- свойство name позволяет присвоить имя окну или прочитать имя окна (не следует путать с названием документа);
- значением свойства navigator является ссылка на объект Navigator, содержащий информацию о браузере;
- значением свойства opener является ссылка на объект окна, из которого открывалось текущее окно;
- свойство parent позволяет получить доступ к объекту родительского окна;
- свойство screen возвращает ссылку на объект экрана;
- ссылка на объект окна может быть получена через свойство window или self;
- значением свойства status является текст, отображаемый в строке состояния окна браузера (предварительно следует убедиться, что настройки браузера позволяют использовать строку состояния);
- значением свойства top является ссылка на объект родительского окна самого верхнего уровня.

Некоторые из перечисленных свойств мы еще будем использовать впоследствии.

Что касается методов объекта window, то некоторые из них (такие, скажем, как методы для отображения диалоговых окон) мы уже рассматривали. Описание (краткое) наиболее актуальных методов (включая и уже знакомые нам) представлено в табл. 7.4.

Таблица 7.4. Краткое описание некоторых методов объекта window

Метод	Описание
alert()	Методом отображается окно с сообщением (передается аргументом методу) и кнопкой подтверждения
blur()	Методом снимается фокус с окна
clearInterval()	Методом выполняется остановка процесса по выполнению периодических действий, который был запущен с помощью метода setInterval()
clearTimeout()	Метод отменяет команду выполнения действия с отсрочкой, которая реализуется через вызов метода setTimeout()
close()	Вызов метода приводит к закрытию окна, из объекта которого вызывался метод
confirm()	При вызове метода отображается диалоговое окно подтверждения с сообщением и двумя кнопками (подтверждения и отмены)
focus()	При вызове метода окну, из объекта которого вызывается метод, передается фокус, в результате чего окно выводится на передний план
moveBy()	Метод позволяет переместить окно на указанное расстояние (в пикселях) по горизонтали (первый аргумент) и вертикали (второй аргумент)
moveTo()	Метод позволяет переместить окно в указанное место. Новая позиция окна определяется аргументами метода (расстояние по горизонтали и вертикали от левого верхнего угла экрана до левого верхнего угла окна)
open()	Метод позволяет открыть в окне новый документ
prompt()	Методом отображается диалоговое окно с полем ввода
resizeBy()	Метод позволяет изменить размеры окна на указанные аргументами метода значения
resizeTo()	Метод предназначен для изменения размеров окна: новые размеры окна (в пикселях) передаются аргументами методу
scroll()	Метод позволяет выполнить прокрутку содержимого окна до указанной позиции
scrollBy()	Метод позволяет выполнить прокрутку содержимого окна на количество позиций (в пикселях), указанных аргументом метода
scrollTo()	Метод позволяет выполнить прокрутку содержимого окна до указанной позиции
setInterval()	Метод используется для запуска процесса по периодическому (с определенным интервалом) выполнению некоторых действий
setTimeout()	Метод позволяет выполнить команду с некоторой временной отсрочкой

Выше перечислены относительно универсальные методы, которые имеются у объекта window большинства наиболее популярных браузеров. Вообще же следует иметь в виду, что у каждого браузера есть свои уникальные свойства (и методы) объекта window, а в новые версии браузеров вносятся изменения, которые могут касаться и специфики реализации объекта window.

НА ЗАМЕТКУ

Даже если тот или иной метод или свойство поддерживается браузером, следует учитывать, что реализация такой «поддержки» в каждом браузере потенциально может иметь некоторые особенности. Например, в браузере Mozilla Firefox методы, связанные с изменением размеров и положения окна (такие, например, как resizeBy(), resizeTo(), moveBy() или moveTo()), могут применяться только к тем окнам, которые созданы с помощью метода open(). Также соответствующие операции не могут применяться к окнам, которые содержат несколько вкладок.

Мы не планируем использовать в большом количестве методы объекта window — за некоторыми исключениями, которые обсудим несколько позже. Небольшой комментарий посвятим обработке событий при работе с объектом окна. Специфика обработки событий такова, что все они обрабатываются в общем потоке и одно и то же событие может обрабатываться разными элементами. Обычно обработка событий выполняется на уровне элементов документа. Вместе с тем в некоторых случаях ее выполняют на уровне объекта окна. Ранее в табл. 7.3 приводился список обработчиков, которые в принципе могут использоваться при работе с объектом окна. Пример использования обработчиков для объекта window приведен в листинге 7.9.

Листинг 7.9. Обработчики событий для объекта window (файл Listing07_09.html)

```
<!DOCTYPE HTML>
<html><head><title>Листинг 7.9</title></head>
<body>
  <h3>Листинг 7.9</h3><hr>
  <!-- Текстовый блок с явным указанием стиля -->
  <p id="mydata" style="font-family:arial;font-size:25pt;color:black;">
    <b>Клавиша не нажата: черный текст на белом фоне</b>
```

```
  </p>
  <!-- Завершение текстового блока -->
  <hr>
  <b>Обратите внимание на цвет фона и текст, представленный выше.</b>
  <!-- Начало сценария -->
  <script type="text/javascript">
    // Обработчик для события, состоящего в нажатии
    // клавиши при активном окне:
    window.onkeydown=function(){
      // Установка желтого цвета для фона:
      document.body.style.backgroundColor="yellow"
      // Ссылка на объект текстового блока:
      var mydata=document.getElementById("mydata")
      // Установка синего цвета для текста:
      mydata.style.color="blue"
      // Установка текста для отображения в документе:
      mydata.innerHTML="<b>Нажата клавиша: синий текст на желтом фоне</b>"
    }
    // Обработчик для события, состоящего в отпускании
    // нажатой клавиши при активном окне:
    window.onkeyup=function(){
      // Установка белого цвета для фона:
      document.body.style.backgroundColor="white"
      // Ссылка на объект текстового блока:
      var mydata=document.getElementById("mydata")
      // Установка черного цвета для текста:
      mydata.style.color="black"
      // Установка текста для отображения в документе:
      mydata.innerHTML="<b>Клавиша не нажата: черный текст на белом фоне</b>"
    }
  </script>
  <!-- Завершение сценария -->
</body>
</html>
```

Представленным кодом реализуется документ, в котором для объекта window использована обработка событий, связанных с нажатием клавиши и отпусканием клавиши. За данные события «отвечают» обработчики onkeydown и onkeyup соответственно. Как выглядит документ в окне браузера, показано на рис. 7.30.

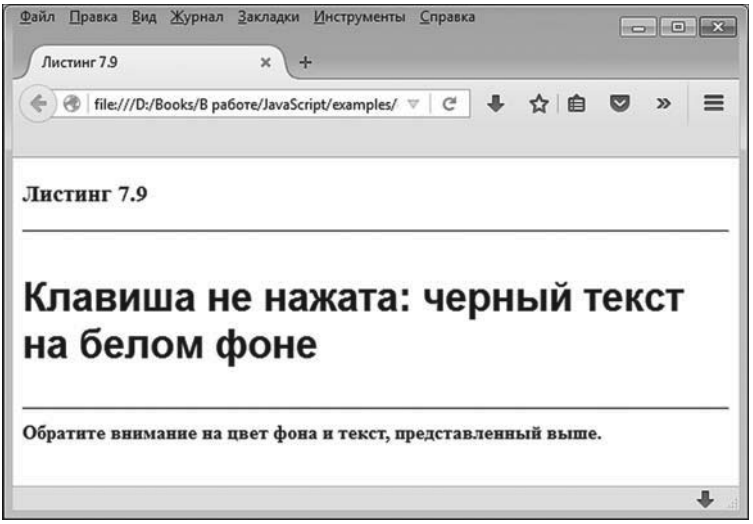

Рис. 7.30. Вид, который имеет окно, если на клавиатуре ни одна (любая) клавиша не нажата

Центральную часть документа составляет текстовая область (выделена двумя горизонтальными линиями) с текстом Клавиша не нажата: черный текст на белом фоне. Если при активном окне (то есть когда окну передан фокус) нажать на клавиатуре клавишу (любую) и удерживать ее, то фон документа станет желтым, цвет текста станет синим, а сам текст изменится на Нажата клавиша: синий текст на желтом фоне, как показано на рис. 7.31.

Как только клавишу отпускаем, документ возвращается в исходное (см. рис. 7.30) состояние (черный текст на белом фоне, с соответствующим изменением текста в документе). Такой нехитрый пример. Теперь проанализируем, за счет чего описанный эффект достигается. Для этого обратимся к коду документа (см. листинг 7.9).

В <body>-разделе документа описан текстовый <p>-блок, в котором явно задан способ форматирования текста в блоке — определен шрифт (тип, размер и цвет).

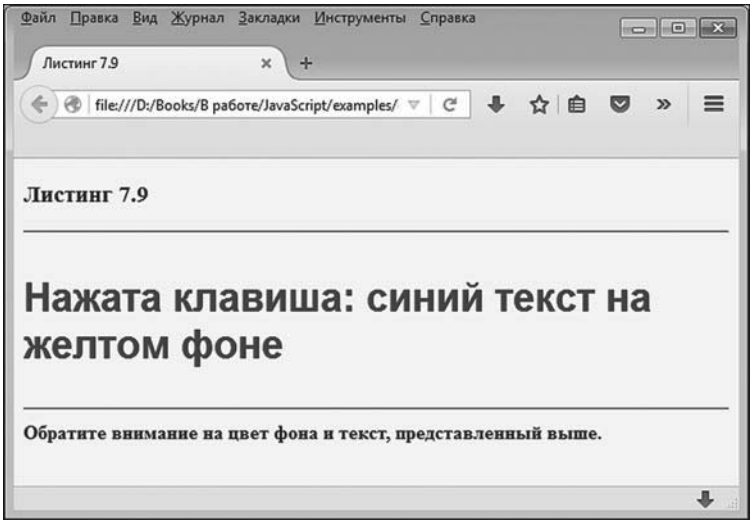

Рис. 7.31. Вид, который имеет активное окно (когда окну передан фокус) при условии, что на клавиатуре удерживается нажатой клавиша (любая)

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

В <p>-блоке в соответствующем дескрипторе использован атрибут style. Значение атрибута определяет стиль, с применением которого отображается текст в данном текстовом блоке. Значением атрибута style является текстовая строка, собственно и задающая стиль. В данном конкретном случае значением атрибута style указана текстовая строка "font-family:arial;font-size:25pt;color:black;". Строка состоит из трех внутренних блоков, заканчивающихся точкой с запятой. Каждый блок, в свою очередь, представляет собой пару из названия свойства стиля и значения этого свойства. Название свойства и его значение разделяются двоеточием. Например, пара font-family:arial означает, что текст отображается с использованием шрифта Arial. Пара font-size:25pt означает, что размер шрифта устанавливается равным 25, а пара color:black означает, что используется черный (значение black) цвет для шрифта.

Также в <p>-блоке использован идентификатор id со значением "mydata". Через значение данного идентификатора мы получим доступ к текстовому блоку в сценарии. Собственно текстовый блок содержит код Клавиша не нажата: черный текст на белом фоне, что означает отображение на начальном этапе (до выполнения кода сценария) в текстовом блоке соответствующего текста жирным стилем. Как отмечалось выше, текстовый блок визуально выделен горизонтальными линиями, для

чего использована инструкция <hr> перед <p>-блоком и после <p>-блока. Также после текстового блока размещен статический (неизменяемый при выполнении сценария) текст (имеется в виду код Обратите внимание на цвет фона и текст, представленный выше.).

Таким образом, концептуально документ сформирован из текстовой «шапки», текстового блока и текста под текстовым блоком. При выполнении кода сценария может меняться фон документа (который по умолчанию белый), а также текст в текстовом блоке и цвет этого текста.

Далее обсудим собственно программный код сценария. Фактически в <script>-блоке описано два обработчика onkeydown и onkeyup, которые реализованы в виде методов объекта window. Эти методы вызываются автоматически при наступлении события, связанного с нажатием клавиши, и события, связанного с отпусканием клавиши. Здесь мы явно принимаем в расчет, что метод можно интерпретировать как свойство, значением которого является ссылка на функцию. Поэтому обработчики определяются так: значениями свойствам onkeydown и onkeyup объекта window (инструкции window.onkeydown и window.onkeyup) присваиваются анонимные функции, которые описаны без аргументов, а код функции определяет действия, выполняемые при обработке события.

НА ЗАМЕТКУ

Напомним, что создание анонимной функции подразумевает использование ключевого слова function, после которого в круглых скобках перечисляются (через запятую) аргументы функции, а в фигурных скобках размещаются команды, выполняемые при вызове функции.

Но и здесь не все так просто. Дело в том, что при обработке события нередко приходится обращаться к *объекту события* — например, чтобы получить информацию об элементе, вызвавшем событие. В большинстве браузеров (в том числе и в Mozilla Firefox) объект события, который создается автоматически при возникновении события, передается аргументом обработчику события. Правда, в старых версиях браузера Internet Explorer доступ к объекту события осуществляется через свойство event объекта window. Но в данном случае это не важно. В рассматриваемом примере мы в явном виде объект события не используем, а функции, которые в качестве значений присваиваются обработчикам событий, описываются без аргументов. Здесь не возникает проблемы, поскольку в JavaScript при вызове функции или метода не выполняется проверка на предмет

формального соответствия количества фактически переданных аргументов и количества указанных при описании функции или метода аргументов.

В частности, при нажатии клавиши (любой) выполняются следующие команды (команды в теле функции, присваиваемой значением инструкции window.onkeydown).

- Командой document.body.style.backgroundColor="yellow" устанавливается желтый цвет для фона документа. Инструкция в левой части команды присваивания довольно длинная, но на самом деле простая. В объекте документа document мы обращаемся к свойству body. Это свойство позволяет получить доступ к объекту <body>-блока в документе. В этом объекте нас интересует свойство style, являющееся ссылкой на объект, соответствующий стилю, применяемому при отображении содержимого <body>-блока документа. В объекте стиля мы обращаемся к свойству backgroundColor, текстовое значение которого определяет цвет, применяемый для фона при отображении документа. Текстовое значение "yellow" означает желтый цвет.
- Командой var mydata=document.getElementById("mydata") объявляется локальная переменная mydata, и значением ей присваивается ссылка на объект текстового блока со значением "mydata" атрибута id. Ссылку по значению атрибута id получаем с помощью метода getElementById(). Метод вызывается из объекта документа document, а аргументом методу передается значение атрибута id.
- Командой mydata.style.color="blue" устанавливается синий цвет для текста в текстовом блоке (определяемом переменной mydata). Здесь с помощью переменной mydata обращаемся к объекту текстового блока. Свойство style данного объекта является ссылкой на объект, задающий стиль для отображения текстового блока. В свою очередь, у объекта стиля есть свойство color, определяющее цвет текста для отображения в блоке. Присвоив свойству color значение "blue", устанавливаем синий цвет для текста в блоке.
- Командой mydata.innerHTML="Нажата клавиша: синий текст на желтом фоне" задается текст, отображаемый в текстовом блоке. В данном случае мы используем свойство innerHTML текстового блока, доступ к которому получаем через переменную mydata. Напомним, что значением свойства innerHTML является HTML-код, содержащийся в соответствующем блоке.

Аналогичным образом определяется обработчик для события, состоящего в отпускании ранее нажатой клавиши (обработчик onkeyup). Принципиальная разница лишь в тех значениях, которые присваиваются свойствам при выполнении кода функции, присвоенной значением выражению window.onkeyup.

НА ЗАМЕТКУ

Обработка событий — процесс кропотливый и нередко чреват сюрпризами. Например, с рассмотренным выше примером можно проделать такой эксперимент. Открываем документ и при активном окне нажимаем и удерживаем какую-нибудь клавишу. Фон документа становится желтым, а текст в текстовом блоке — синим. Далее, продолжая удерживать нажатой клавишу, сворачиваем окно (щелчком кнопкой мыши по соответствующей пиктограмме) и сразу отпускаем клавишу. То есть клавиша отпускается сразу после того, как свернуто окно. Если теперь развернуть окно, то фон документа будет желтым, а текст в текстовом блоке — синим. Проще говоря, документ выглядит так, как если бы была нажата и удерживалась клавиша, хотя клавишу мы давно отпустили. Почему так происходит? Объяснение в общих чертах следующее.

При выполнении кода сценария (в том числе кода, связанного с обработкой событий) в общем случае вносятся изменения в объектную модель документа, которая хранится в памяти. Когда мы нажимаем и удерживаем нажатой клавишу, то объектная модель документа изменяется таким образом, что текст в текстовом блоке отображается синим цветом (ну и сам текст имеет соответствующее значение), а фон документа желтый. При сворачивании окна (и удерживаемой в нажатом состоянии клавише) модель документа в памяти не изменяется. Она не изменяется и после того, как пользователь отпускает клавишу при свернутом окне. Когда мы разворачиваем окно на экране, используется объектная модель документа, которая была при сворачивании документа. Причина «неизменности» объектной модели в данном случае связана с тем, что при отпускании клавиши окно было свернуто, и поэтому сценарий обработку соответствующего события не выполнил.

Если после этого снова нажать и отпустить клавишу (или перезагрузить страницу) — все вернется на круги своя. Тем не менее, чтобы устранить означенную «особенность» документа, можно добавить в сценарий, например, еще обработчик для события, состоящего в передаче фокуса окну. За данное событие «отвечает» обработчик onfocus объекта window. Скажем, желающие могут последней строкой кода в сценарии из документа добавить инструкцию window.onfocus=window.onkeyup. Данной инструкцией свойству onfocus объекта window

присваивается значение свойства onkeyup того же объекта. Это ссылка на функцию, которая вызывается при отпускании клавиши. В результате каждый раз, когда окну передается фокус (когда окно становится активным — в том числе и при разворачивании свернутого окна), будет выполняться тот же код, что и при отпускании нажатой клавиши. Хотя, конечно, это не единственный способ решения проблемы.

Есть и другие полезные события, которые можно использовать при работе с объектом окна window. Некоторые из наиболее актуальных (в том числе и некоторые из рассмотренных выше) обработчиков с кратким описанием приведены в табл. 7.5.

Таблица 7.5. Краткое описание некоторых обработчиков для объекта window

Обработчик	Описание
onbeforeunload	Обработчик вызывается непосредственно перед началом выгрузки ресурсов окна
onblur	Обработчик вызывается при отмене фокуса для окна
onclick	Обработка событий, связанных со щелчком кнопкой мыши в области окна
onfocus	Обработка события, связанного с передачей фокуса окну
onkeydown	Обработка события, связанного с нажатием клавиши на клавиатуре
onkeypress	Обработка события, связанного со щелчком клавиши на клавиатуре
onkeyup	Обработка события, связанного с отпусканием нажатой клавиши на клавиатуре
onload	Обработка события, связанного с завершением загрузки ресурсов в окно
onmousedown	Обработка события, связанного с нажатием кнопки мыши
onmousemove	Обработка события, связанного с перемещением курсора мыши
onmouseout	Обработка события, связанного с перемещением курсора мыши за пределы области окна
onmouseover	Обработка события, связанного с перемещением курсора мыши в область окна
onmouseup	Обработка события, связанного с отпусканием нажатой кнопки мыши
onresize	Обработка событий, связанных с изменением размеров окна
onsubmit	Обработка события, связанного с отправкой содержимого формы в окне
onunload	Обработчик вызывается сразу после выгрузки ресурсов из окна

Более подробно вопрос обработки событий мы еще будем обсуждать, но сделаем это немного позже.

Таймеры

Выше мы уже кратко рассматривали методы объекта window. Здесь мы в некотором смысле вернемся к данному вопросу, поскольку есть несколько методов, которые хочется выделить особо. Их нередко называют *таймерами*, так как методы позволяют выполнять команды с временной задержкой или повторять некоторую команду многократно с определенным временным интервалом. Решаются подобные задачи с помощью методов setInterval() и setTimeout().

> **НА ЗАМЕТКУ**
>
> Хотя здесь речь идет о методах объекта window, мы в силу традиции при вызове методов в явном виде данный объект указывать не будем. Также уместно напомнить, что функции, используемые в сценарии, на самом деле являются методами объекта window. В этом смысле не будет большой ошибкой интерпретировать обсуждаемые методы как функции.

Метод setTimeout() позволяет вызвать функцию, но не сразу, а через определенный промежуток времени. Имя вызываемой функции передается первым аргументом методу setTimeout(). Вторым аргументом методу передается числовое значение, определяющее интервал времени (в миллисекундах), по истечении которого должна быть вызвана функция, переданная первым аргументом методу. Если при вызове функции ей нужно передать аргументы, то они передаются в метод setTimeout(), начиная с третьего аргумента. Допустим, мы вызываем метод setTimeout() в следующем формате:

```
setTimeout(функция,время,аргументы)
```

В таком случае через указанное время будет вызвана функция с указанными аргументами.

Важная особенность метода setTimeout() связана с тем, что вызов метода не останавливает процесс выполнения других команд. Вызов метода setTimeout() запускает «счетчик», и, не дожидаясь окончания «отсчета», сразу выполняется следующая команда.

Метод setTimeout() возвращает значение, которое является «идентификатором» процесса. Этот идентификатор можно передать аргументом методу clearTimeout(), в результате чего «счетчик» будет остановлен.

Фактически с помощью метода clearTimeout() можно отменить отложенную инструкцию, определенную с помощью метода setTimeout().

Метод setInterval() позволяет периодически выполнять определенные действия. Первым аргументом методу передается имя функции, которую следует вызывать. Второй аргумент метода определяет временной интервал (в миллисекундах) между вызовом функции (переданной первым аргументом метода). Если функции при вызове передаются аргументы, то они указываются аргументами метода setInterval(), начиная с третьего. Другими словами, вызов метода setInterval() выполняется в таком формате:

```
setInterval(функция,время,аргументы)
```

Как следствие, будет периодически вызываться функция с указанными аргументами, а интервал между вызовами определяется временем, указанным вторым аргументом метода setInterval(). После вызова метода setInterval() сразу выполняется следующая команда. При этом продолжается периодический вызов функции, указанной первым аргументом метода setInterval(). Для прекращения процесса периодического вызова функции, запущенного методом setInterval(), можно вызвать метод clearInterval(), передав ему аргументом переменную, в которую предварительно записан результат вызова метода setInterval().

Далее мы рассмотрим пример, в котором используется метод setInterval(). Прежде чем рассмотреть программный код соответствующего документа, кратко опишем его внешние, так сказать, особенности. На рис. 7.32 показано, как документ выглядит при открытии в окне браузера.

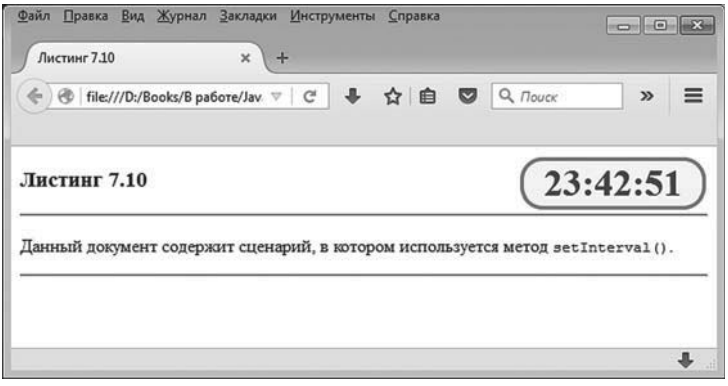

Рис. 7.32. Документ с отображаемыми в правом верхнем углу импровизированными часами

Особенность документа в том, что в его правом верхнем углу есть импровизированные часы, показывающие текущее время. Время отображается в стандартном формате с указанием часов, минут и секунд с двоеточиями между ними. Блок, в котором выводится текущее время, имеет закругленные края, рамку красного цвета, желтый фон, а значение момента времени внутри блока выполняется синим цветом. Еще одна особенность документа в том, что при изменении размеров окна часы все равно остаются в правом верхнем углу, как, например, показано на рис. 7.33.

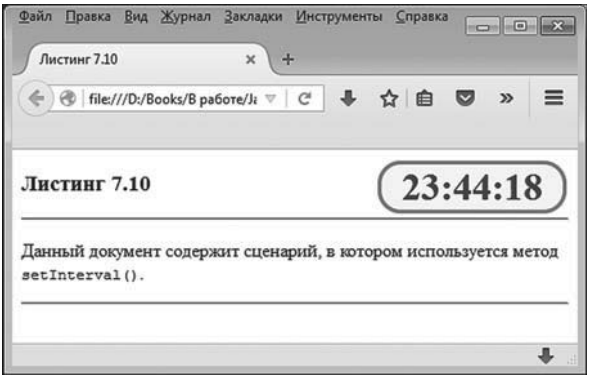

Рис. 7.33. При изменении размеров окна часы остаются в правом верхнем углу рабочей области окна

В документе процесс отображения текущего времени в блочном элементе реализуется с помощью сценария. Все остальное выполняется стандартными средствами HTML (с использованием каскадных таблиц стилей). Код рассматриваемого документа представлен в листинге 7.10.

Листинг 7.10. Использование метода setInterval() (файл Listing07_10.html)

```
<!DOCTYPE HTML>
<html><head><title>Листинг 7.10</title>
<!-- Стиль для блочного элемента -->
<style type="text/css">
  #myblock{
    font: bold 25pt sans;
    color: blue;
    width: 160px;
```

```
    height: 40px;
    background: yellow;
    border: solid 3px red;
    border-radius: 20px;
    position: absolute;
    top: 10px;
    right: 10px;
    text-align: center;
  }
</style>
<!-- Завершение описания стиля -->
<!-- Начало сценария -->
<script type="text/javascript">
  // Функция для определения
  // содержимого блочного элемента:
  function setMyTime(){
    // Объект даты:
    var time=new Date()
    // Содержимое блочного элемента - текущее время:
    document.getElementById("myblock").innerHTML=time.toLocaleTimeString()
  }
</script>
<!-- Завершение сценария -->
</head>
<body onload="setInterval(setMyTime,1000)">
  <h3>Листинг 7.10</h3>
  <hr>
  <p>Данный документ содержит сценарий, в котором используется метод <code>setInterval()</
code>.</p>
  <!-- Блочный элемент -->
  <div id="myblock"></div>
  <hr>
</body>
</html>
```

В документе в <head>-блоке описан блок <style>, определяющий стиль для блочного элемента (обсуждается далее во врезке). В этом же <head>-блоке есть сценарий, в котором описана функция setMyTime(). В функции выполняется простой код. Сначала командой var time=new Date() объявляется локальная переменная time, и значением ей присваивается объект даты, созданный на основе объекта-конструктора Date. Созданный таким образом объект, на который ссылается переменная time, содержит значение текущего времени. С помощью метода toLocaleTimeString(), который вызывается из объекта, на который ссылается переменная time, получаем текстовое представление для текущего значения времени. Данное текстовое значение реализуется с учетом региональных настроек системы. Результат выражения time.toLocaleTimeString() присваивается значением свойству innerHTML блочного элемента, доступ к которому получаем с помощью инструкции document.getElementById("myblock") (команда document.getElementById("myblock").innerHTML=time.toLocaleTimeString()). Доступ к объекту -блока получаем с помощью метода getElementById(), передав ему аргументом значение "myblock" атрибута id, с которым описан блок. Таким образом, свойство innerHTML определяет «внутреннее содержимое» для блочного элемента, описанного с атрибутом id, имеющим значение "myblock".

НА ЗАМЕТКУ

Блочный элемент описан в <body>-блоке документа инструкцией . В общем случае между дескрипторами и указывается HTML-код или обычный текст, который формирует содержимое блочного элемента (отображается внутри элемента). В данном случае между дескрипторами и ничего не указано. Поэтому в самый начальный момент соответствующий блок пустой. Однако он заполняется при выполнении сценария.

При вызове функции setMyTime() в блочном элементе отображается текущее время. С другой стороны, в <body>-блоке в дескрипторе <body> описан атрибут onload со значением "setInterval(setMyTime,1000)". В данном случае мы используем обработку события, состоящего в завершении загрузки документа. Инструкция onload="setInterval(setMyTime,1000)" в дескрипторе <body> означает, что при завершении загрузки документа с интервалом в одну секунду (интервал в 1000 миллисекунд) вызывается функция setMyTime(). Как следствие, содержимое блочного элемента будет обновляться с интервалом в секунду, причем каждый раз отображается текущее время.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Стиль для -блока описывается в блоке <style>. Атрибут type в дескрипторе <style> определяет синтаксис, который используется при описании стиля. В данном случае <style>-блок состоит всего из одного «стиля» (хотя в принципе их могло бы быть больше). Описание стиля начинается с инструкции #myblock, а собственно значения для параметров стиля описываются в фигурных скобках: указывается название свойства и через двоеточие — значение этого свойства. Каждая такая инструкция заканчивается точкой с запятой.

Ключевое слово myblock после инструкции # означает, что стиль задается для элемента, значение атрибута id которого равно "myblock". В нашем случае речь идет о -блоке, в котором отображается время.

Далее перечислим свойства, которые задаются с помощью <style>-блока. Значением свойства font (задает шрифт) указано выражение bold 25pt sans, означающее, что используется жирный (значение bold) шрифт Sans размера 25. Свойство color определяет цвет, которым отображается содержимое блока, — указанное значение blue означает синий цвет. Свойство width определяет ширину блока в пикселях (в данном случае 160), а свойство height задает высоту блока в пикселях (в данном случае 40). Свойство background определяет цвет фона блока (значение yellow означает желтый цвет). Тип, толщина (в пикселях) и цвет границы задаются с помощью свойства border. Значением свойства указано выражение solid 3px red, означающее, что линия для отображения границы сплошная (значение solid), толщина линии составляет 3 пикселя, а цвет линии красный (значение red). Свойство border-radius задает радиус закругления углов для рамки блока. Мы используем радиус закругления в 20 пикселей. Еще три свойства имеют отношение к позиционированию блочного элемента в окне документа. Значение absolute для свойства position означает, что положение для элемента задается в пикселях относительно рабочей области окна (то есть абсолютное положение, не привязанное к положению других элементов). Мы задаем значения 10px для свойств top и right, и это означает, что блочный элемент позиционируется в окне отступом от верхней границы на 10 пикселей и отступом от правой границы на 10 пикселей. Наконец, с помощью свойства text-align задается способ выравнивания (вдоль горизонтали) содержимого в блочном элементе. Значение center означает, что содержимое блока выравнивается по центру.

Далее рассматривается еще один пример использования функций setInterval() и clearInterval(). Как и в предыдущем случае, мы сначала опишем функциональные возможности соответствующего документа. На рис. 7.34 показан документ, открытый в окне браузера.

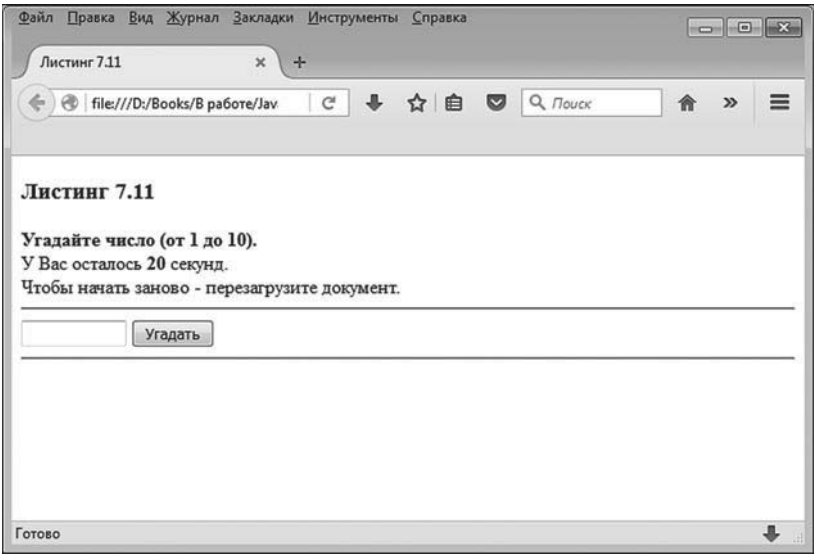

Рис. 7.34. Окно с полем ввода для угадывания числа, кнопкой для подтверждения введенного числового значения и с обратным отсчетом времени (в секундах)

Окно с документом содержит текст с предложением угадать число (значение в диапазоне от 1 до 10 включительно). Сразу под этим текстом содержится сообщение с указанием оставшегося для угадывания времени (в секундах). Пользователю дается 20 секунд для угадывания числа. Ниже размещается поле ввода и кнопка с названием **Угадать**.

При загрузке документа в окно браузера сценарием генерируется случайное целое число (диапазон возможных значений от 1 до 10). Пользователю нужно его угадать. При этом постоянно идет обратный отсчет времени. Угадываемое число вводится в поле ввода, после чего пользователь щелкает по кнопке **Угадать**. Такая ситуация проиллюстрирована на рис. 7.35.

Далее возможны две ситуации (если, конечно, не учитывать случай, когда пользователь вводит некорректное значение в поле ввода): пользователь угадал число или пользователь не угадал число. Допустим, пользователь число, не угадал. В этом случае внизу под полем ввода отображается сообщение с информацией о введенном числе и указано, что это число неправильное (то есть не то, которое «загадано» генератором случайных чисел). Такая ситуация проиллюстрирована на рис. 7.36.

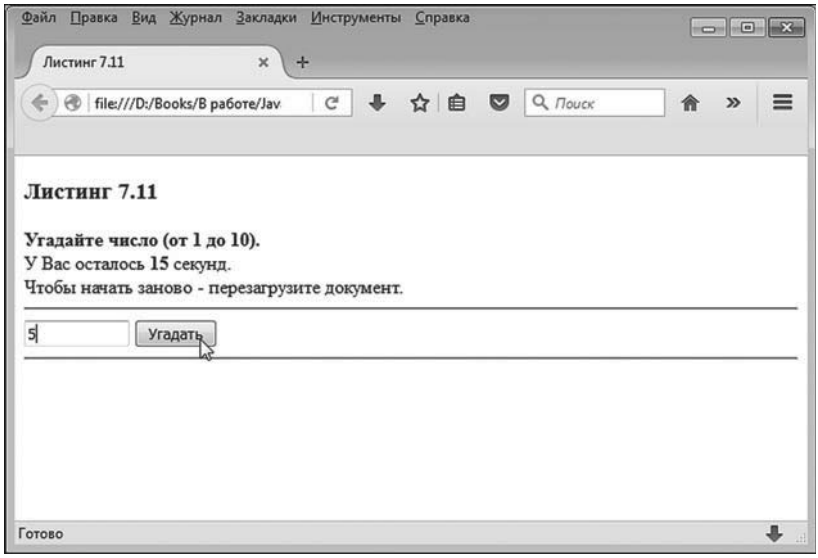

Рис. 7.35. В процессе угадывания числа необходимо ввести число в поле ввода и щелкнуть по кнопке **Угадать**

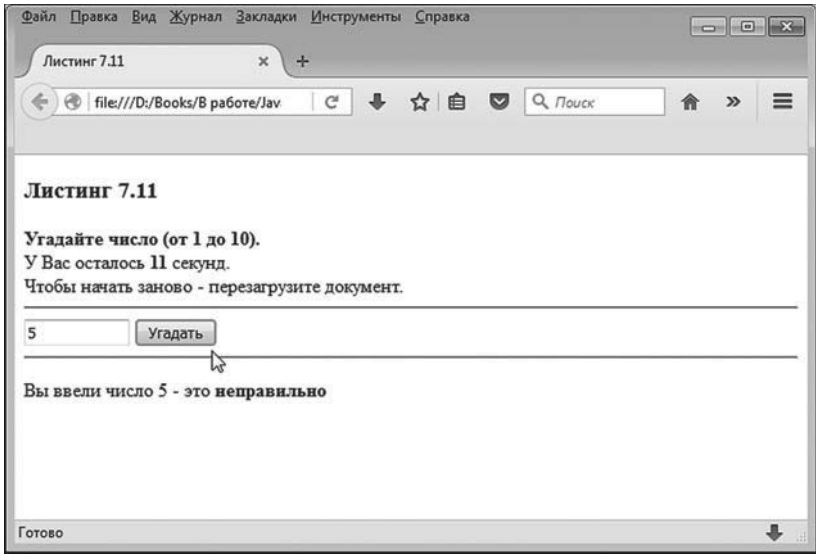

Рис. 7.36. Если пользователь не угадал число, то под полем ввода отображается соответствующее сообщение

При этом время не сбрасывается (то есть обратный отсчет времени продолжается), но у пользователя остается возможность и дальше угадывать число (но только пока не закончилось отведенное время).

Если до конца отведенного времени пользователь все же угадывает число, то в нижней части окна появляется соответствующее сообщение, обратный отсчет времени останавливается, а кнопка **Угадать** блокируется (становится неактивной).

Например, на рис. 7.37 показано, как может выглядеть документ, в котором пользователь дважды угадал число только с третьего раза.

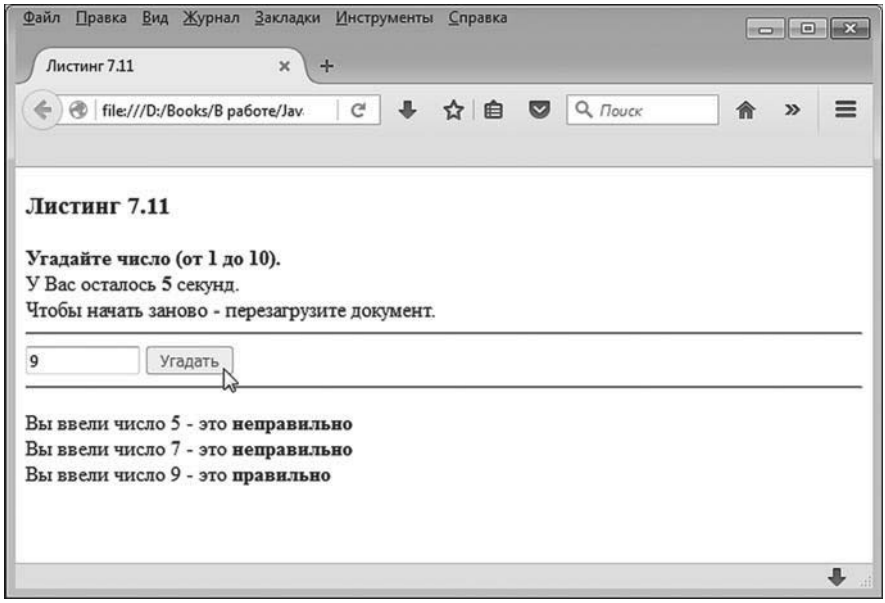

Рис. 7.37. Если пользователь угадал число, то в документе появляется соответствующее сообщение, обратный отсчет времени останавливается, а кнопка **Угадать** становится неактивной

В данном случае пользователю нужно угадать число за 5 секунд до окончания времени, отведенного для угадывания числа.

Возможен и другой вариант, когда пользователь за отведенное время не успевает угадать число.

В таком случае кнопка **Угадать** блокируется, а обратный отсчет останавливается на нулевом значении, как показано на рис. 7.38.

В данном конкретном случае пользователь трижды пытался угадывать число, но так и не угадал. Также следует заметить, что перезагрузка документа в окне браузера приводит к тому, что процесс «угадывания» начинается заново.

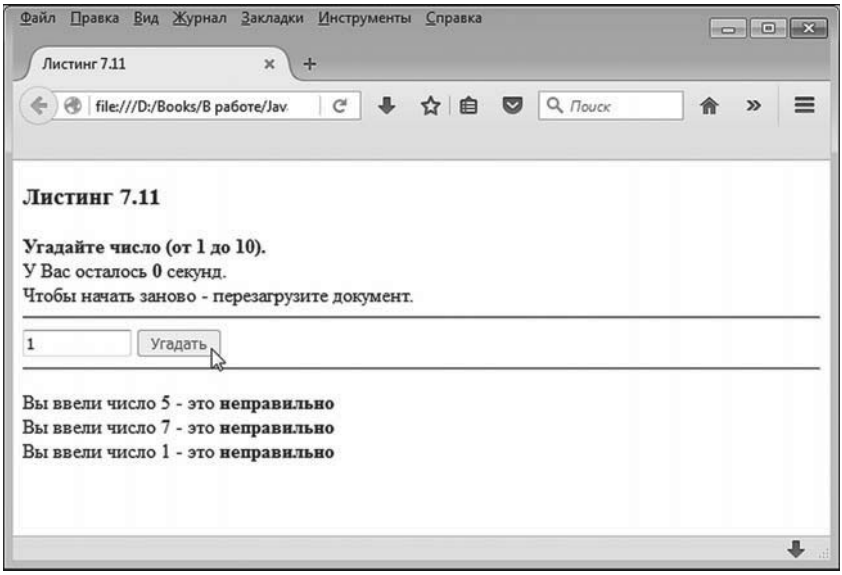

Рис. 7.38. Если пользователь не успел угадать число за отведенное время, то обратный отсчет останавливается на нулевом значении, а кнопка **Угадать** становится неактивной

Далее проанализируем код документа, представленный в листинге 7.11.

Листинг 7.11. Использование методов setInterval() и clearInterval() (файл Listing07_11.html)

```
<!DOCTYPE HTML>
<html><head><title>Листинг 7.11</title>
<!-- Начало сценария -->
<script type="text/javascript">
  // Случайное целое число от 1 до 10:
  var myrand=Math.floor(10*Math.random())+1
  // Переменная для записи в нее результата вызова
  // метода setInterval():
  var si
  // Переменные для записи ссылок на текстовый фрагмент,
  // поле ввода, кнопку и текстовый блок:
  var mytime,myfield,mybtn,output
  // Функция для отображения оставшегося времени:
```

```
function setNewTime(){
  // Новое значение для оставшегося времени:
  var num=parseInt(mytime.innerHTML)-1
  // Отображение нового значения в документе:
  mytime.innerHTML=num
  // Если значение нулевое:
  if(num==0){
    // Кнопка становится неактивной:
    mybtn.disabled=true
    // Процесс отсчета времени прекращается:
    clearInterval(si)
  }
}
// Функция для обработки щелчка на кнопке:
function pressMe(){
  // Числовое значение в поле ввода:
  var num=parseInt(myfield.value)
  // Отображение текста в документе:
  output.innerHTML+="Вы ввели число "+num
  // Если пользователь не угадал число:
  if(num!=myrand){
    // Сообщение о том, что число не угадано:
    output.innerHTML+=" - это <b>неправильно</b><br>"
  }
  // Если пользователь угадал число:
  else{
    // Сообщение о том, что число угадано:
    output.innerHTML+=" - это <b>правильно</b><br>"
    // Кнопка становится неактивной:
    mybtn.disabled=true
    // Прекращение процесса отсчета времени:
    clearInterval(si)
  }
}
```

```
  // Обработчик для события загрузки документа:
  window.onload=function(){
    // Получение ссылки на текстовый фрагмент:
    mytime=document.getElementById("mytime")
    // Получение ссылки на поле ввода:
    myfield=document.getElementById("myfield")
    // Получение ссылки на кнопку:
    mybtn=document.getElementById("mybtn")
    // Получение ссылки на текстовый блок:
    output=document.getElementById("output")
    // Кнопка активна:
    mybtn.disabled=false
    // Запуск процесса отсчета времени:
    si=setInterval(setNewTime,1000)
  }
</script>
<!-- Завершение сценария -->
</head>
<body>
  <h3>Листинг 7.11</h3>
  <b>Угадайте число (от 1 до 10).</b><br>
  <!-- Отображение оставшегося времени -->
  У Вас осталось <b id="mytime">20</b> секунд.<br>
  Чтобы начать заново - перезагрузите документ.
  <hr>
  <!-- Текстовое поле (для ввода числа) -->
  <input id="myfield" type="text" size="10px">
  <!-- Кнопка -->
  <input id="mybtn" type="button" value="Угадать" onclick="pressMe()">
  <hr>
  <!-- Текстовый блок для отображения введенных чисел -->
  <p id="output"></p>
  </body>
</html>
```

Сценарий содержится в <head>-блоке документа, но мы начнем анализ с <body>-блока, в котором есть несколько элементов, используемых в сценарии.

Прежде всего стоит заметить, что в текстовой фразе об оставшемся для угадывания числа времени содержится фрагмент, выделяемый дескрипторами и (жирный текст). В нем использован атрибут id со значением "mytime". В этом месте сценарием будет вписываться числовое значение, определяющее оставшееся время (в секундах). Собственно, в коде документа в данном фрагменте указано значение 20. Это число отображается сразу при загрузке документа и фактически определяет интервал времени, который отводится на угадывание числа.

Текстовое поле (поле ввода) и кнопка создаются с помощью <input>-блоков. Разница только в значении атрибута type: для поля значение атрибута равно "text", а для кнопки значение атрибута равно "button". Название, отображаемое на кнопке, задается с помощью атрибута value (значением атрибута указан текст "Угадать"). Также для обоих элементов (поле ввода и кнопка) использован атрибут id. Для поля значение атрибута равно "myfield", а для атрибута кнопки указано значение "mybtn". Из поля мы (с помощью сценария) планируем считывать значение, а происходить это будет при щелчке по кнопке. Поэтому понятно, что как-то к этим элементам в сценарии необходимо обращаться. Мы это сделаем с помощью атрибута id. Помимо перечисленных, в документе еще имеется текстовый блок, выделенный дескрипторами и . Значение атрибута id для данного блока равно "output". Блок мы используем при отображении сообщений о том, угадано или не угадано число.

Теперь проанализируем код сценария. Самой первой командой var myrand=Math.floor(10*Math.random())+1 в сценарии объявляется переменная myrand, а значением этой переменной присваивается выражение Math.floor(10*Math.random())+1. Чтобы понять, каков результат данного выражения, следует учесть, что методом random() объекта Math возвращается случайное действительное число. Диапазон его возможных значений лежит от 0 (включительно) до 1 (строго меньше). Таким образом, значением выражения 10*Math.random() является случайное число в диапазоне от 0 (включительно) до 10 (строго меньше). Вся эта конструкция передается аргументом методу floor() объекта Math, в результате чего получаем целое число, которое вычисляется отбрасыванием дробной

части в выражении, переданном аргументом методу floor(). Несложно сообразить, что это целое число в диапазоне значений от 0 (включительно) до 9 (включительно). Если к этому выражению прибавить единицу, получаем случайное число в диапазоне возможных значений от 1 до 10 включительно.

Также в сценарии объявляется глобальная переменная si, значение которой присваивается несколько позже. Кроме этой глобальной переменной, сценарий содержит объявление глобальных переменных mytime, myfield, mybtn и output. Переменные нужны для записи в них ссылок на объекты текстового фрагмента, поля ввода, кнопки и текстового блока. Для удобства названия переменных совпадают со значениями атрибута id для каждого из перечисленных элементов (но вообще это не обязательно). Присваивание значений переменным выполняется при обработке события, связанного с загрузкой документа.

В сценарии описано несколько функций. Мы начнем с последней (в том порядке, как функции описаны в сценарии). Точнее, речь идет о методе onload() объекта окна window. Причем это не просто метод, а метод, который вызывается для обработки события, состоящего в загрузке документа в окно. Поэтому код метода onload() автоматически выполняется по завершении загрузки документа.

Свойству onload объекта window значением присваивается анонимная функция, в теле которой командами mytime=document.getElementById("mytime"), myfield=document.getElementById("myfield"), mybtn=document.getElementById("mybtn") и output=document.getElementById("output") вычисляются ссылки на текстовый фрагмент, поле ввода, кнопку и текстовый блок. Ссылки записываются в глобальные переменные. После этого переменные могут использоваться для получения доступа к указанным элементам документа.

НА ЗАМЕТКУ

Поскольку обработчик загрузки документа выполняется сразу после того, как документ загружен, а ссылки на элементы документа записываются в глобальные переменные, то на момент вызова прочих функций (обработчиков) данные переменные уже будут иметь значения.

Затем выполняются команды mybtn.disabled=false и si=setInterval(setNewTime, 1000). Командой mybtn.disabled=false свойству disabled для объекта кнопки присваивается значение false. В результате кнопка станет активной.

НА ЗАМЕТКУ

По умолчанию кнопка и так активна. Но если перезагружать документ, в котором кнопка переведена в неактивное состояние, то некоторые браузеры могут не обновить состояние кнопки в объектной модели, хранящейся в памяти, и кнопка останется неактивной после перезагрузки. Чтобы этого избежать, на всякий случай в сценарий добавляем команду mybtn.disabled=false, которой кнопка переводится в активное состояние. Свойство disabled отвечает за «неактивность», поэтому если значение свойства равно false, то кнопка активна, а если значение свойства равно true, то кнопка неактивна.

Командой si=setInterval(setNewTime,1000) запускается процесс обратного отсчета времени. Формально это выглядит как периодический вызов функции setNewTime(), а интервал между вызовами функции составляет 1000 миллисекунд (или 1 секунду). Результат вызова метода setInterval() записывается в глобальную переменную si, что дает возможность впоследствии остановить процесс обратного отсчета времени.

В теле функции setNewTime() выполняется несколько команд. Так, командой var num=parseInt(mytime.innerHTML)-1 объявляется локальная переменная num, значением которой присваивается число, на единицу меньшее числа, отображаемого в блоке с информацией об оставшемся времени. Процедура вычислений здесь следующая. Значением выражения mytime.innerHTML является HTML-код, содержащийся внутри фрагмента, описанного со значением "mytime" для атрибута id (это фрагмент текста, в котором отображается время). Хотя там записано число, но «считывается» оно как текст. Поэтому его необходимо преобразовать в числовой формат. Для этого используем стандартную функцию parseInt(). Аргументом ей передается выражение mytime.innerHTML. От полученного числового значения отнимается единица. Таким образом, значение переменной num будет на единицу меньше, чем текущее значение для оставшегося времени.

Далее командой mytime.innerHTML=num новое значение записывается в блок, в котором отображается оставшееся время для отгадывания числа. Таким образом, при вызове функции setNewTime() на единицу уменьшается значение для оставшегося времени. Но это еще не все. После того как новое значение записано в соответствующий блок, в теле функции запускается условный оператор. Это оператор в упрощенной форме — в нем нет else-блока. В операторе проверяется условие num==0, которое состоит в том, что в документе отображается нулевое значение. Если так, то командой mybtn.disabled=true кнопка переводится в неактивное со-

стояние, а командой clearInterval(si) завершается процесс обратного отсчета времени. Здесь для остановки процесса периодического вызова функции (речь о функции setNewTime()) мы вызвали метод clearInterval(), а аргументом ему передали переменную si, чтобы идентифицировать процесс, который следует остановить.

Код функции pressMe() выполняется при щелчке по кнопке. Происходит это благодаря инструкции onclick="pressMe()" в дескрипторе <input>, с помощью которого описывается кнопка. Другими словами, анализируемый далее код выполняется, когда пользователь щелкает по кнопке в документе.

В теле функции командой var num=parseInt(myfield.value) из поля ввода считывается значение, введенное пользователем. В этой команде выражение myfield.value возвращает текстовое значение из поля ввода, которое преобразуется в целочисленное значение с помощью функции parseInt(), а общий результат записывается в локальную переменную num (переменная доступна только в теле функции и не имеет никакого отношения к локальной переменной с таким же именем из функции setNewTime()).

> **НА ЗАМЕТКУ**
>
> Чтобы сократить объем программного кода и не отвлекаться на несущественные в данном случае детали, мы не выполняем обработку исключительных ситуаций, которые могут возникнуть, например, если пользователь вводит некорректное значение в поле ввода.

Далее в текстовый блок, определяемый переменной output, выводится текст (в том числе со значением переменной num). Для проверки того, угадал пользователь число или нет, используется условный оператор. В нем проверяется условие num!=myrand (значение переменной num отлично от сгенерированного случайного числа, записанного в глобальную переменную myrand). В этой части код должен быть понятен читателю. Также помимо отображения текста, если число угадано (проверяемое условие ложно), командой mybtn.disabled=true кнопка переводится в неактивное состояние, а командой clearInterval(si) останавливается процесс обратного отсчета времени.

Еще один пример иллюстрирует использование методов setTimeout() и clearTimeout(). Идея, реализованная в документе, очень простая: через определенное время после открытия документа отображается диалоговое окно с сообщением. При этом в документе имеется специаль-

ная кнопка, щелкнув по которой, можно предотвратить отображение диалогового окна. На рис. 7.39 показано, как выглядит окно браузера с загруженным в нем документом.

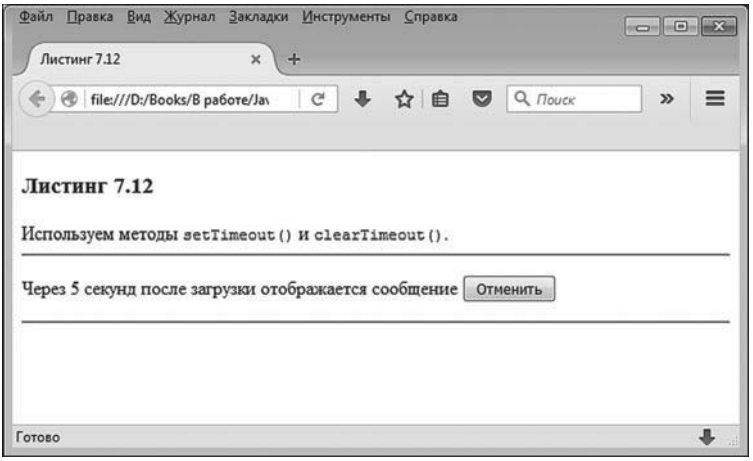

Рис. 7.39. Вид документа при загрузке в окно браузера

Окно содержит текст, информирующий, что через определенный промежуток времени (в данном случае 5 секунд) будет отображено диалоговое окно. Также справа от данного текста имеется кнопка с названием **Отменить**. Если ничего не делать, то через 5 секунд после загрузки документа действительно появится диалоговое окно, как это показано на рис. 7.40.

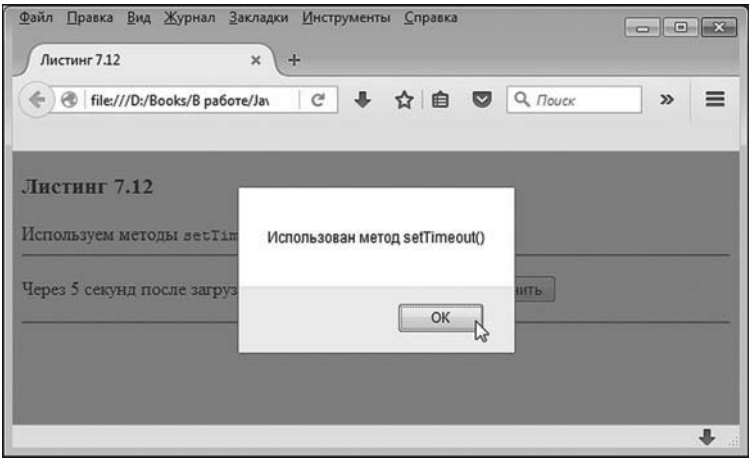

Рис. 7.40. Через определенное время в области документа отображается диалоговое окно с сообщением

После закрытия диалогового окна вид документа станет иным: кнопка **Отменить** пропадает, а также изменяется текст в документе. Как выглядит окно с документом после того, как было закрыто диалоговое окно с сообщением, показано на рис. 7.41.

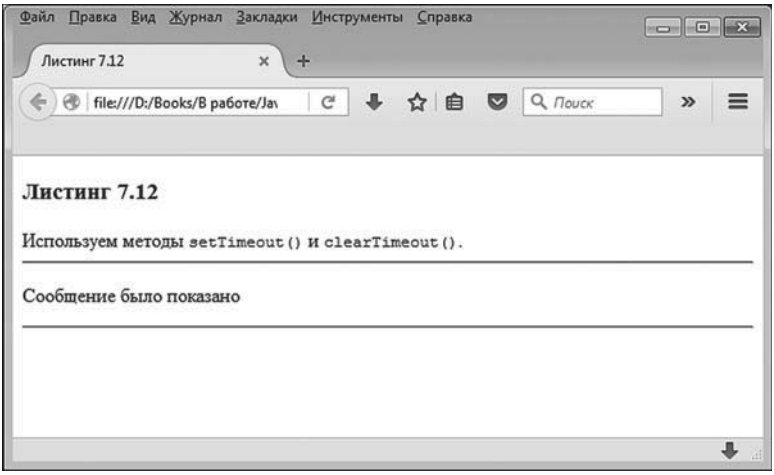

Рис. 7.41. Вид документа после закрытия диалогового окна с сообщением

Наконец, если в самом начале (см. рис. 7.39), еще до отображения диалогового окна, щелкнуть по кнопке **Отменить**, то данная кнопка также пропадет, и изменится текст в области документа. Как все это выглядит, показано на рис. 7.42.

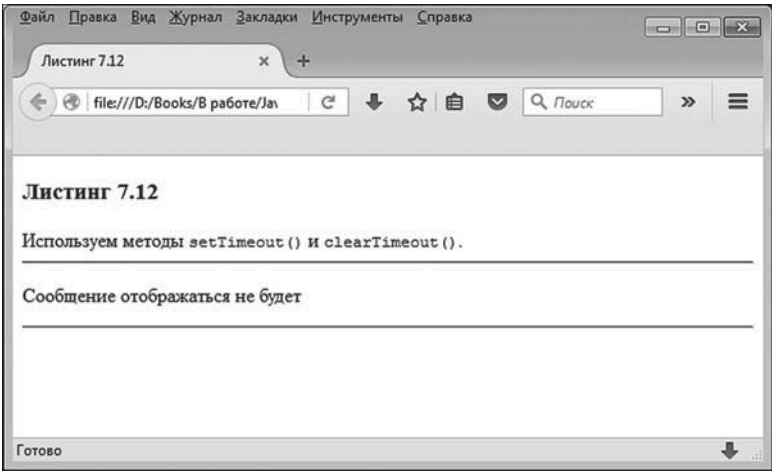

Рис. 7.42. Вид документа после щелчка по кнопке **Отменить**

Понятно, что в данном случае диалоговое окно отображаться не будет. Это вкратце те функциональные возможности документа, которые для нас представляют первоочередной интерес. Теперь посмотрим, как они реализуются с помощью программного кода. Код документа представлен в листинге 7.12.

Листинг 7.12. Использование методов setTimeout() и clearTimeout() (файл Listing07_12.html)

```
<!DOCTYPE HTML>
<html><head><title>Листинг 7.12</title>
<!-- Начало сценария -->
<script type="text/javascript">
  // Переменная для записи результата
  // вызова метода setTimeout():
  var st
  // Переменная для записи ссылки на текстовый блок:
  var myblock
  // Обработчик для события загрузки документа:
  window.onload=function(){
    // Ссылка на текстовый блок:
    myblock=document.getElementById("myblock")
    // Переменная определяет время (в секундах),
    // через которое отображается диалоговое окно:
    var num=5
    // Текстовое значение для отображения
    // в текстовом блоке:
    var txt="Через "+num+" секунд после загрузки отображается сообщение "
    txt+="<button onclick='makeclick()'>Отменить</button>"
    // Вызов функции show() с временной задержкой:
    st=setTimeout(show,num*1000,"Использован метод setTimeout()")
    // Определение содержимого текстового блока:
    myblock.innerHTML=txt
  }
  // Функция для отображения сообщения:
  function show(str){
```

```
    // Отображение сообщения:
    alert(str)
    // Определение содержимого текстового блока:
    myblock.innerHTML="Сообщение было показано"
  }
  // Функция для обработки щелчка на кнопке:
  function makeclick(){
    // Отмена отложенного вызова функции show():
    clearTimeout(st)
    // Определение содержимого текстового блока:
    myblock.innerHTML="Сообщение отображаться не будет"
  }
</script>
<!-- Завершение сценария -->
</head>
<body>
  <h3>Листинг 7.12</h3>
  Используем методы <code>setTimeout()</code> и <code>clearTimeout()</code>.<br>
  <hr>
  <!-- Текстовый блок -->
  <p id="myblock"></p>
  <hr>
  </body>
</html>
```

Помимо сценария, в коде документа интерес представляет разве что текстовый <p>-блок в <body>-блоке. Данный текстовый блок описан со значением "myblock" для атрибута id, и вначале он пустой, а при выполнении сценария в блоке отображается текст (и не только).

Теперь собственно о сценарии. Там объявляются глобальные переменные st и myblock, а также описывается несколько функций. В частности, в сценарии описывается метод onload() для объекта окна window (обработчик события, связанного с загрузкой документа в окно). В теле метода описаны следующие команды. Сначала командой myblock=document.getElementById("myblock") в глобальную переменную myblock записывается

ссылка на текстовый блок. Затем создается локальная переменная num со значением 5 (время в секундах, через которое отображается диалоговое окно). После этого объявляется еще одна локальная переменная txt, значением которой присваивается текст. Текст нетривиальный, поэтому для большей элегантности процесс формирования значения переменной txt разбит на несколько команд. Здесь есть два момента, на которые хочется обратить внимание. Во-первых, в тексте использовано значение переменной num. Во-вторых, значение переменной txt, кроме собственно текста, содержит еще и HTML-код, определяющий кнопку. Речь идет о текстовом фрагменте "<button onclick='makeclick()'>Отменить </button>", который «дописывается» к текущему значению переменной txt. Вся эта конструкция, если не считать внешних двойных кавычек, соответствует блоку, определяемому дескрипторами <button> и </button>, то есть блоку, задающему кнопку. Название кнопки указывается между дескрипторами. Также в дескрипторе <button> описан атрибут onclick со значением 'makeclick()'. При определении значения атрибута мы используем одинарные кавычки, поскольку двойными кавычками выделена вся конструкция. Если бы мы так описали кнопку, то при щелчке по ней вызывалась бы функция makeclick() (функция описывается далее). Фактически так и происходит, когда командой myblock.innerHTML=txt значение переменной txt применяется в качестве внутреннего содержимого текстового блока. Данное содержимое интерпретируется и обрабатывается именно как HTML-код, поэтому наличие инструкции, описывающей кнопку, приводит к тому, что кнопка реально появляется в документе и для нее валидны все описанные атрибуты. В частности, при щелчке по кнопке вызывается функция makeclick().

Однако перед выполнением команды myblock.innerHTML=txt командой st=set Timeout(show,num*1000,"Использован метод setTimeout()") выполняется отложенный вызов функции show(). Аргументом функции передается текст "Использован метод setTimeout()", а вызывается функция через промежуток времени (в миллисекундах), определяемый значением выражения num*1000 (то есть получается интервал, в секундах равный num). Результат вызова функции setTimeout() записывается в переменную st. Мы воспользуемся этой переменной, когда нужно будет отменить отложенный вызов функции show().

НА ЗАМЕТКУ

Важно подчеркнуть, что выполнение команды st=setTimeout(show,num*1000," Использован метод setTimeout()") не приостанавливает выполнение следую-

щей команды. Другими словами, выполнение указанной выше команды приводит к вызову (одному!) функции show() через заданный интервал времени. Но, не дожидаясь этого вызова, выполняется следующая команда myblock.innerHTML=txt.

У функции show() есть аргумент, который обозначен как str. Данный аргумент используется в команде alert(str), которой отображается диалоговое окно с сообщением. Текст сообщения определяется аргументом str. Затем командой myblock.innerHTML="Сообщение было показано" изменяется содержимое текстового блока. Это именно тот блок, в котором отображалась кнопка, и теперь кнопки в блоке нет.

Функция makeclick() вызывается щелчком по кнопке. В теле функции выполняется команда clearTimeout(st), благодаря чему отменяется отложенный вызов функции show(). Также командой myblock.innerHTML="Сообщение отображаться не будет" меняется содержимое текстового блока. В данном случае также исчезает кнопка. То есть получается, что при щелчке по кнопке выполняется код, который «удаляет» кнопку.

Объект документа document

— Он своеобразен, не правда ли?
— Да, в нем есть нечто подлинное.

из к/ф «Покровские ворота»

Теперь мы обсудим объект документа document. Через объект document реализуется доступ к свойствам и содержимому документа, открытого в окне браузера. Полная ссылка на объект документа выглядит как window.document, но, поскольку ссылку на объект окна window указывать необязательно, обычно просто используют идентификатор document.

Свойства и методы объекта document

Далее мы рассмотрим основные свойства и методы объекта document. Их достаточно много, поэтому мы сначала сделаем общий обзор, а затем на некоторых свойствах и методах остановимся подробнее. Также следует учесть, что некоторые свойства и методы являются зависимыми от браузера. В любом случае, как неоднократно отмечалось, при реализации прикладного программного кода не следует забы-

вать, что для каждого браузера в принципе реализуется своя объектная модель документа.

Чтобы получить список свойств и методов, поддерживаемых для объекта document браузером, можем воспользоваться кодом, аналогичным тому, что применялся при идентификации свойств и методов объекта window. Интересующий нас код представлен в листинге 7.13. От кода из листинга 7.7 его принципиально отличает лишь то, что оператор цикла в сценарии выполняется по коллекции свойств и методов объекта document, а не объекта window, как это было ранее.

Листинг 7.13. Свойства и методы объекта document (файл Listing07_13.html)

```
<!DOCTYPE HTML>
<html><head><title>Листинг 7.13</title>
</head>
<body><h3>Листинг 7.13</h3><hr>
  <b>Свойства и методы объекта <code>document</code></b><br>
  <!-- Начало сценария -->
  <script type="text/javascript">
    // Индексная переменная:
    var k=1
    // Отображение свойств и методов:
    for(var prop in document){
      // Отображение свойства или метода:
      document.write("<b>"+k+"</b>: "+prop+"<br>")
      // Новое значение индексной переменной:
      k++
    }
  </script>
  <!-- Завершение сценария -->
</body>
</html>
```

Как будет выглядеть документ с соответствующим кодом, открытый в браузере Mozilla Firefox, показано на рис. 7.43.

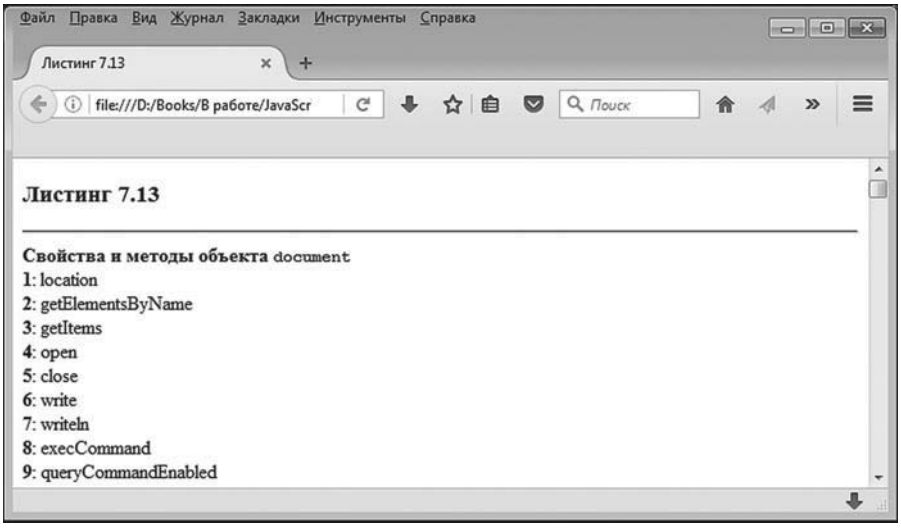

Рис. 7.43. Документ со списком свойств и методов объекта *document* открыт в браузере Mozilla Firefox

При открытии документа с помощью других браузеров получаем нечто похожее (правда, список свойств и методов будет несколько иным).

Свойства, связанные с объектом документа document, обычно используются для выполнения настроек или получения сведений общего характера, имеющих отношение ко всему документу. Общее представление о характере таких свойств дает табл. 7.6, в которой представлено краткое описание некоторых наиболее часто используемых свойств объекта document, являющихся общими для браузеров (то есть поддерживающихся) Mozilla Firefox, Internet Explorer, Google Chrome и Opera.

Таблица 7.6. Описание некоторых свойств объекта document

Свойство	Описание
activeElement	Значением свойства является ссылка на элемент документа, активный (которому передан фокус) на данный момент
alinkColor	Свойство задает цвет активной ссылки в документе. Ссылка является активной в период времени между нажатием и отпусканием кнопки мыши (при щелчке по гиперссылке). Значением свойства является текстовая строка с названием цвета или шестнадцатеричным кодом цвета. Хотя свойство поддерживается браузерами, имеет место тенденция к отказу от его использования
all	Через данное свойство реализуется доступ к элементам документа. Хотя свойство и поддерживается браузерами, оно не входит в современный стандарт веб-программирования

Свойство	Описание
anchors	Значением свойства является ссылка на список из элементов-анкеров в документе
applets	Значением свойства является ссылка на список апплетов (функциональные элементы, реализованные на языке программирования Java), содержащихся в документе
bgColor	Свойство определяет фоновый цвет для области документа. Значением свойства является текстовая строка с названием цвета или шестнадцатеричным кодом цвета. Хотя свойство поддерживается браузерами, оно не входит в современный стандарт веб-программирования
body	Значением свойства является ссылка на объект для тела документа (элемент, выделенный в HTML-коде дескрипторами <body> и </body>)
characterSet	Значением свойства является текстовая строка с названием используемой в документе кодировки символов. Свойство доступно только для чтения
charset	Предназначенное только для чтения свойство позволяет узнать кодировку документа. Не рекомендуется к использованию (вместо данного свойства рекомендуется применять свойство characterSet)
childNodes	Значением свойства является объект с коллекцией дочерних узлов (находящихся непосредственно внутри исходного узла) для данного документа (речь идет об элементах, содержащихся внутри документа)
compatMode	Свойство определяет режим обработки документа (в частности, нужно ли использовать режим обратной совместимости со значением "BackCompat" для свойства — применяется при работе с документами устаревших стандартов)
cookie	Свойство определяет *cookies*-значение, ассоциированное с данным документом. Значение свойства реализуется в виде текстовой строки формата *ключ=значение*
defaultView	Значением свойства является ссылка на объект окна window, в котором открыт документ. Свойство доступно только для чтения
designMode	Свойство определяет доступность документа для редактирования. Возможные значения свойства "on" и "off"
dir	Свойство определяет направление текста. Значение свойства реализуется строкой "ltr" (от *left to right* — текст отображается слева направо, — значение по умолчанию) или "rtl" (от *right to left* — текст отображается справа налево)
doctype	Значением свойства является ссылка на объект, содержащий информацию о типе документа
documentElement	Значением свойства является ссылка на объект для корневого элемента документа (в случае HTML-документа это элемент, выделенный дескрипторами <html> и </html>)
domain	Свойство определяет доменное имя сервера для данного документа
embeds	Значением свойства является список ссылок на объекты, вставленные в документ

Свойство	Описание
fgColor	Свойство определяет цвет текста для отображения в документе. Значением свойства является текстовая строка с названием цвета или шестнадцатеричным кодом цвета. Хотя свойство поддерживается браузерами, оно не является частью современного стандарта веб-программирования
firstChild	Свойством возвращается ссылка на первый дочерний элемент (узел) документа
forms	Значением свойства является список форм (элемент, выделенный дескрипторами <form> и </form>) в документе
head	Значением свойства является ссылка на объект для элемента заголовка документа (элемент, выделенный дескрипторами <head> и </head>)
hidden	Свойство определяет видимость документа. Если документ отображается на экране, то значение свойства равно false. Если окно с документом находится в свернутом состоянии, то значение свойства равно true
images	Значением свойства является список рисунков, содержащихся в документе
implementation	Свойством возвращается ссылка на объект реализации документа
inputEncoding	Аналог свойства characterSet, которое рекомендуется использовать
lastChild	Свойством возвращается ссылка на последний дочерний элемент (узел) документа
lastModified	Значением свойства является текстовая строка с датой последнего изменения документа
linkColor	Свойство задает цвет, которым отображаются гиперссылки в документе. Значением свойства является текстовая строка с названием цвета или шестнадцатеричным кодом цвета. Хотя свойство поддерживается браузерами, оно не включено в современный стандарт веб-программирования
links	Значением свойства является список гиперссылок в документе (под «учет» попадают элементы, для которых задан атрибут href)
location	Значением свойства является ссылка на объект, содержащий информацию о размещении документа
nextSibling	Свойством возвращается ссылка на элемент, следующий сразу после данного элемента в списке дочерних узлов для родительского элемента. Родительским является элемент, в котором содержится данный элемент
nodeName	Свойством возвращается текстовая строка с названием данного узла (в данном случае речь идет об объекте документа, и значение свойства равно "#document")
nodeType	Свойство позволяет определить тип узла. В случае объекта документа возвращается целочисленная константа DOCUMENT_NODE со значением 9
nodeValue	Свойством возвращается (или задается) значение для узла. Для объекта документа значение по умолчанию равно null
ownerDocument	Свойством возвращается ссылка на объект документа, содержащего данный узел. Если данный узел сам является документом, то значение свойства равно null

Свойство	Описание
parentNode	Свойством возвращается родительский элемент для данного элемента — то есть элемент, в котором содержится данный элемент
plugins	Значением является список встроенных объектов
previousSibling	Свойством возвращается ссылка на элемент перед данным элементом в списке дочерних узлов для родительского элемента. Родительским является элемент, в котором содержится данный элемент
readyState	Значение свойства позволяет узнать состояние загрузки документа (возможные значения "loading", "interactive" и "complete")
referrer	Значением свойства является текст с адресом документа, через гиперссылку в котором был открыт текущий документ
scripts	Значением свойства является список со ссылками на объекты элементов сценариев (элементы, которые в документе выделены дескрипторами <script> и </script>)
styleSheets	Значением свойства является список явно подключенных или описанных в документе стилей
textContent	Значением свойства является текстовое представление для узла. Если речь идет об объекте документа, то значением свойства является null. Для получения содержимого, записанного в документ, используют свойство textContent объекта, возвращаемого свойством documentElement объекта документа
title	Значением свойства является название документа (элемент, выделенный дескрипторами <title> и </title>)
URL	Значением свойства является текст с адресом документа
visibilityState	Значение свойства определяет состояние документа в плане его отображения на экране. Возможные значения свойства: hidden (документ не отображается), prerender (документ загружен и не отображается), visible (документ отображается) и unloaded (документ выгружается)
vlinkColor	Свойство определяет цвет для отображения гиперссылок, по которым уже выполнялись переходы. Значением свойства является текстовая строка с названием цвета или шестнадцатеричным кодом цвета. Хотя свойство поддерживается браузерами, оно не является частью современного стандарта веб-программирования

Многие из перечисленных выше свойств есть не только у объекта документа document, но также и у внутренних элементов, которые могут входить в документ. Их мы будем обсуждать по мере знакомства с элементами документа.

Существует много методов, которые могут использоваться при работе с объектом document. В табл. 7.7 перечислены и кратко описаны основные, общие для браузеров Mozilla Firefox, Internet Explorer, Opera и Google Chrome методы объекта document.

Таблица 7.7. Некоторые методы объекта document

Метод	Описание
addEventListener()	Метод используется при регистрации обработчиков событий
adoptNode()	Методом выполняется «заимствование» узла (узел с его поддеревом удаляется из исходного документа и связывается с текущим документом)
appendChild()	Метод используется для добавления дочернего узла в узел, из которого вызывается метод. В данном случае метод позволяет добавить узел в документ
captureEvents()	Используется для перехвата событий определенного типа. Метод не является частью современного стандарта, хотя и поддерживается браузерами
clear()	Исходное назначение метода состояло в очистке содержимого документа. В современных браузерах данный метод, хотя формально и поддерживается, но не выполняет никаких действий
cloneNode()	Метод используется для клонирования (создания копии) узлов
close()	Методом завершается процесс записи в документ, который был открыт с помощью метода open() объекта документа document
compareDocumentPosition()	Метод используется для сравнения положений узлов
createAttribute()	Методом создается и возвращается в качестве результата новый узел атрибута
createAttributeNS()	Методом создается и возвращается новый узел атрибута в указанном пространстве имен
createCDATASection()	Метод для создания узла CDATA-раздела
createComment()	Методом создается и возвращается узел комментария
createDocumentFragment()	Методом создается пустой фрагмент документа
createElement()	Метод позволяет создать новый элемент. Тип (в виде текстовой строки) создаваемого элемента передается аргументом методу
createElementNS()	Метод для создания узла для нового элемента в указанном пространстве имен
createEvent()	Метод для создания объекта события
createNodeIterator()	Метод для создания объекта-итератора для итерирования (перебора) узлов
createProcessingInstruction()	Метод используется для создания узла XML-директивы
createRange()	Методом создается и возвращается Range-объект, через который реализуется фрагмент документа
createTextNode()	Методом создается текстовый узел
createTreeWalker()	Методом создается и возвращается TreeWalker-объект, через который реализуются узлы дерева документа

Метод	Описание
dispatchEvent()	Метод используется при диспетчеризации (искусственном генерировании) событий
elementFromPoint()	По заданным координатам возвращается ссылка на элемент, находящийся в соответствующем месте документа
execCommand()	Метод для выполнения команд форматирования в режиме редактирования документа
getElementById()	Метод позволяет получить доступ к элементу по его атрибуту id (значение атрибута передается аргументом методу)
getElementsByClassName()	Методом возвращается список всех элементов указанного класса, которые имеются в документе
getElementsByName()	Методом возвращается список со ссылками на элементы с определенным (передается аргументом методу) значением атрибута name
getElementsByTagName()	Методом возвращается список элементов определенного типа. Тип элементов в данном случае определяется дескриптором, с которым описан элемент. Название дескриптора передается аргументом методу
getElementsByTagNameNS()	Методом возвращается список элементов определенного типа, определяемого дескриптором, для указанного пространства имен
getSelection()	Методом возвращается ссылка на объект, содержащий текст, который на данный момент выделен в документе
hasChildNodes()	Метод возвращает истинное значение, если элемент имеет дочерние узлы
hasFocus()	Метод используется для проверки того, передан ли документу фокус
importNode()	Методом возвращается клон (копия) узла из внешнего документа
insertBefore()	Метод используется для вставки одного узла перед другим узлом
isDefaultNamespace()	Метод позволяет определить, является ли указанное пространство имен пространством имен по умолчанию для данного узла
isEqualNode()	Метод используется для сравнения узлов на предмет их совпадения (одинаковый тип и содержимое)
lookupNamespaceURI()	Метод по заданному префиксу для данного узла возвращает URI для пространства имен
lookupPrefix()	Методом возвращается префикс для указанного пространства имен
normalize()	С помощью метода узел переводится в «нормальное» состояние, когда он не содержит пустых или смежных текстовых элементов

Метод	Описание
open()	Метод позволяет открыть документ для записи
queryCommandEnabled()	Методом возвращается истинное значение, если для указанного диапазона может быть выполнена команда форматирования
queryCommandIndeterm()	Метод возвращает истинное значение, если команда форматирования для текущего диапазона находится в неопределенном состоянии
queryCommandState()	Метод возвращает истинное значение, если команда форматирования была выполнена для текущего диапазона
queryCommandSupported()	Метод возвращает истинное значение, если команда форматирования поддерживается для текущего диапазона
queryCommandValue()	Метод возвращает текущее значение для текущего диапазона для команды форматирования
querySelector()	Метод возвращает ссылку на узел первого элемента в документе в соответствии с указанным критерием
querySelectorAll()	Метод возвращает список со ссылками на узлы элементов в документе на основе заданных критериев
releaseEvents()	Метод используется для перехода в режим игнорирования событий определенного типа. Метод не является частью современного стандарта веб-программирования
removeChild()	Метод удаляет узел из объектной модели и возвращает удаленный узел в качестве результата
removeEventListener()	Метод используется для удаления обработчика событий
replaceChild()	Метод используется для замены одного узла на другой
write()	Метод предназначен для записи в документ
writeln()	Метод предназначен для записи в документ с переходом к новой строке

Методов достаточно много, и не все они используются одинаково часто.

Среди наиболее актуальных можно выделить группу методов, предназначенных для получения доступа к элементам документа, а также методы, предназначенные для изменения структуры и содержания документа.

Некоторые из методов обсуждаются далее в этой главе и следующих главах книги.

При работе с объектом документа document нередко используется обработка событий. Обработчики событий для объекта document перечислены в табл. 7.8.

Таблица 7.8. Обработчики событий для объекта document

Обработчик	Обработчик	Обработчик
onabort	onended	onplay
onblur	onerror	onplaying
oncanplay	onfocus	onprogress
oncanplaythrough	oninput	onratechange
onchange	onkeydown	onreadystatechange
onclick	onkeypress	onreset
oncontextmenu	onkeyup	onresize
ondblclick	onload	onseeked
ondrag	onloadeddata	onseeking
ondragend	onloadedmetadata	onselect
ondragenter	onloadstart	onstalled
ondragleave	onmousedown	onsubmit
ondragover	onmousemove	onsuspend
ondragstart	onmouseout	ontimeupdate
ondrop	onmouseover	onvolumechange
ondurationchange	onmouseup	onwaiting
onemptied	onpause	

Обработчики отдельно описывать не будем, поскольку они достаточно универсальны, и мы планируем знакомиться с ними по мере рассмотрения примеров. Отметим лишь, что документ, кроме прочего, может «реагировать» на события, связанные с процессом загрузки и выгрузки документа, манипуляциями мышью в области документа, а также на некоторые другие события.

Далее мы кратко рассмотрим примеры использования свойств и методов объекта document, а также коснемся обработки некоторых событий.

НА ЗАМЕТКУ

Многие свойства, методы и обработчики, хотя и поддерживаются для объекта документа document, на самом деле имеют ограниченную область применимости. Опять же операции с объектом документа обычно подразумевают манипуляции на уровне всего документа и его структуры. Это специфическая задача, и ее логичнее рассматривать в контексте операций с содержимым документа. Поэтому предлагаемые далее примеры носят скорее ознакомительный характер. Тем более что некоторые примеры использования объекта document и его свойств и методов мы уже рассматривали ранее.

Настройки цвета

Группа свойств объекта документа document позволяет выполнить цветовые настройки, применяемые для всего документа. Небольшой пример использования таких свойств представлен кодом в листинге 7.14.

> **НА ЗАМЕТКУ**
>
> Рассматриваемые далее свойства (а именно речь идет о свойствах bgColor, fgColor, alinkColor, linkColor и vlinkColor) не являются частью современного стандарта, хотя эти свойства все еще поддерживаются основными браузерами. Проще говоря, их не рекомендуется использовать при разработке новых проектов. Тем не менее многие ранее созданные документы и сценарии их содержат. Поэтому мы также уделим им немного внимания.

Листинг 7.14. Цветовые настройки документа (файл Listing07_14.html)

```
<!DOCTYPE HTML>
<html><head><title>Листинг 7.14</title>
  <!-- Начало сценария -->
  <script type="text/javascript">
    // Функция задает цвет фона и текста, а также
    // цвет гиперссылок:
    function setColor(){
      // Желтый фон:
      document.bgColor="yellow"
      // Синий текст:
      document.fgColor="#0000ff"
      // Красные гиперссылки:
      document.linkColor="#ff0000"
      // Зеленые ссылки, по которым выполнялся переход:
      document.vlinkColor="green"
      // Белые активные ссылки:
      document.alinkColor="white"
    }
  </script>
```

```
  <!-- Завершение сценария -->
</head>
<body onload="setColor()"><h3>Листинг 7.14</h3><hr>
  <b style="font-size:200%;">Синий текст на желтом фоне</b><br>
  <a href="http://www.vasilev.kiev.ua">Первая гиперссылка</a><br>
  <a href="http://www.google.com">Вторая гиперссылка</a><br>
</body>
</html>
```

В данном случае мы описываем в <head>-разделе документа сценарий с функцией setColor(), при вызове которой задается цвет фона документа, цвет текста, а также цвет, которым отображаются гиперссылки. В частности, командой document.bgColor="yellow" задается желтый фон для документа, а командой document.fgColor="#0000ff" мы задаем синий цвет для текста. В первой команде значением свойству bgColor присваивается текстовая строка "yellow" с названием цвета, а во второй команде цвет определяется текстовой строкой "#0000ff", содержащей целочисленное шестнадцатеричное значение 0000ff, определяющее цвет (в данном случае синий) в формате RGB.

НА ЗАМЕТКУ

Формат RGB (сокращение от *Red-Green-Blue*, то есть *красный-зеленый-синий*) определяет цвет как «смесь» красного, зеленого и синего цветов. Доля каждого цвета определяется числом в диапазоне от 0 до 255. В шестнадцатеричном формате числа от 0 до 255 кодируются значениями от 00 до ff. Код цвета состоит из шести позиций, по две позиции на каждый цвет. Первые две позиции определяют долю красного цвета, следующие две позиции определяют долю зеленого цвета, и две последние позиции определяют долю синего цвета. Например, красному цвету соответствует код ff0000, зеленому цвету соответствует код 00ff00, а синему цвету соответствует код 0000ff.

Командой document.linkColor="#ff0000" задается красный цвет для отображения гиперссылок в документе. Командой document.vlinkColor="green" задается зеленый цвет для отображения гиперссылок, по которым уже выполнялся переход. Наконец, командой document.alinkColor="white" задается режим отображения активных ссылок белым цветом. Как выглядит документ, открытый в окне браузера, показано на рис. 7.44.

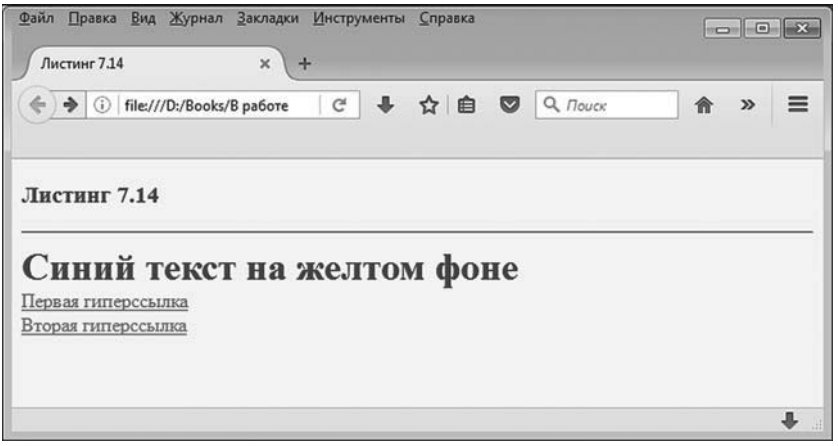

Рис. 7.44. Документ с желтым фоном и синим текстом

Собственно, документ содержит заголовок, текст и две гиперссылки. Заголовок и текст отображаются синим цветом. При описании текста в <b>-дескрипторе (блок жирного шрифта) использована инструкция style="font-size:200%;", которой для данного блока устанавливается размер шрифта в 200% от обычного (используемого по умолчанию) размера.

Гиперссылки вначале отображаются в документе красным цветом. При щелчке по любой из них данная гиперссылка в момент щелчка становится белой, после чего выполняется переход по ссылке. Если еще раз открыть данный документ, то соответствующая гиперссылка будет отображаться зеленым цветом.

НА ЗАМЕТКУ

Еще раз подчеркнем, что рекомендуемым является способ определения параметров документа (и его элементов) через свойства стиля. Доступ к объекту стиля обычно можно получить с помощью свойства style. Например, для определения желтого цвета фона документа могла бы быть использована команда document.body.style.backgroundColor="yellow". Аналогично, задать синий цвет для текста в документе можно с помощью инструкции document.body.style.color="#0000ff".

Код функции setColor() вызывается после выполнения загрузки документа. Реализуется это благодаря инструкции onload="setColor()", которая размещена в <body>-дескрипторе.

Методы write() и writeln()

Ранее в книге мы многократно использовали метод write() для отображения в документе результатов и сопутствующей информации, связанных с выполнением сценариев. Напомним, что метод write() выводит в документ значение (или значения), переданное методу аргументом. У этого метода имеются некоторые особенности, которые мы и собираемся обсудить далее. Кроме того, есть еще и метод writeln(), позволяющий выводить в документ значения с последующим переходом к новой строке. Метод writeln() также будет предметом нашего рассмотрения.

Специфика метода write() состоит в том, что он выводит значение аргумента (или аргументов, если их несколько) непосредственно в HTML-документ в процессе загрузки последнего. Общая схема выглядит следующим образом. При загрузке документа для него формируется объектная модель. Модель формируется по мере загрузки блоков, определяющих элементы документа. Если при этом загружается сценарий с выполняемым кодом, содержащим инструкцию с методом write(), то начинается дописывание в документ значения (или значений), переданного методу. Все эти значения автоматически преобразуются к текстовому формату. Текстовые строки записываются без какой бы то ни было проверки их структуры, то есть так, как есть. Поэтому в принципе с помощью метода write() в документ можно заносить произвольные фрагменты HTML-кода. После завершения загрузки документа поток записи закрывается. Если теперь попытаться выполнить метод write(), то все содержимое документа будет автоматически удалено и последующая запись выполняется в пустой документ. Поэтому метод write() имеет достаточно ограниченную область применимости.

Принципиальное отличие метода writeln() от метода write() состоит в том, что методом writeln() автоматически добавляется инструкция перехода к новой строке. Но проблема в том, что при отображении HTML-документа такая инструкция игнорируется (так же, как, например, несколько пробелов в отображаемом тексте автоматически сокращаются до одного). Выход из ситуации может состоять в том, чтобы использовать <pre>-блоки, в которых текст отображается с учетом всех особенностей его формата без автоматических упрощений и преобразований. Небольшой пример использования методов write() и writeln() приведен в листинге 7.15.

Листинг 7.15. Методы write() и writeln() (файл Listing07_15.html)

```
<!DOCTYPE HTML>
<html><head><title>Листинг 7.15</title>
</head>
<body><h3>Листинг 7.15</h3><hr>
  <h4>Использование метода <code>write()</code></h4>
  <script type="text/javascript">
    // Переменная для оператора цикла:
    var k
    // Массив с текстовыми значениями:
    var mynums=["один","два","три","четыре","пять"]
    // Вызов метода write() с несколькими аргументами:
    document.write("Аргументы метода: ",123," и <b style='font-size:120%;'>",321,"</b><br>")
    // Вызов метода write() в операторе цикла:
    for(k=0;k<mynums.length;k++){
      document.write(mynums[k]," ")
    }
  </script>
  <h4>Использование метода <code>writeln()</code></h4>
<!-- Блок предварительно отформатированного текста -->
<pre>
<script>
  // Вызов метода writeln() с несколькими аргументами:
  document.writeln("Аргументы метода: ",123," и <b style='font-size:120%;'>",321,"</b>")
  // Вызов метода writeln() в операторе цикла:
  for(k=0;k<mynums.length;k++){
    document.writeln(mynums[k])
  }
</script>
</pre>
</body>
</html>
```

Как выглядит данный документ, будучи открытым в окне браузера, показано на рис. 7.45.

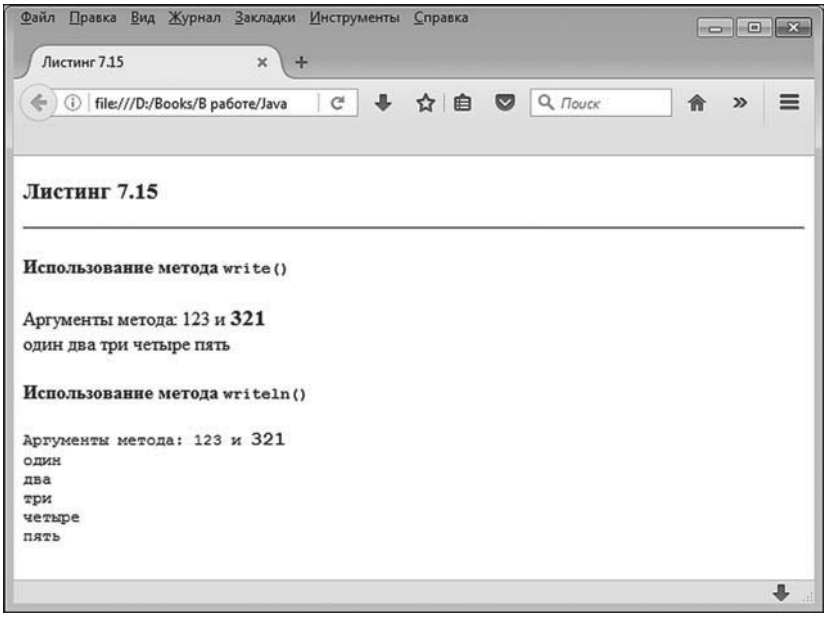

Рис. 7.45. Документ, в котором используются методы *write()* и *writeln()*

Документ очень простой. Он содержит два сценария непосредственно в <body>-блоке документа. В первом сценарии используется метод write(), а во втором сценарии используется метод writeln(). Команда document.write("Аргументы метода: ",123," и ",321,"
") дает пример использования метода write() с несколькими аргументами. Все аргументы отображаются последовательно один за другим. Причем некоторые текстовые аргументы содержат инструкции HTML-разметки. Например, в текстовой строке " и " внутри -дескриптора содержится инструкция применения стиля для данного блока (блок жирного текста). Стиль предполагает, что размер шрифта в блоке составляет 120% от размера стандартного используемого в документе шрифта. Разные аргументы «компонуются» при отображении вместе, и то, что получилось, интерпретируется как HTML-код.

Еще один пример вызова метода write() представлен в теле оператора цикла командой document.write(mynums[k]," "). В данном случае отображаются текстовые значения, предварительно оформленные в виде массива mynums. Второй аргумент метода write() является текстовой строкой из одного символа пробела. Эта строка добавляется после отображения каждого элемента массива mynums, так что текстовые значения разделены пробелом.

НА ЗАМЕТКУ

Как уже отмечалось, в документе используется два сценария. Массив mynums объявляется в первом сценарии, а используется и в первом, и во втором сценарии. Так можно делать, поскольку переменные (и массивы в том числе), если только они не объявлены в теле функции, доступны в любом сценарии документа.

Второй сценарий в документе иллюстрирует использование метода writeln(). Причем собственно сценарий находится внутри <pre>-блока предварительно отформатированного текста. В частности, в сценарии командой document.writeln("Аргументы метода: ",123," и ",321,"") метод writeln() вызывается с несколькими аргументами — аналогично тому, как это делалось для метода write(). Принципиальное отличие от метода write() состоит в том, что после отображения значений аргументов выполняется переход к новой строке. Переход к новой строке происходит и при выполнении команды document.writeln(mynums[k]) в операторе цикла.

Стоит заметить, что если не использовать <pre>-блок, то использование метода writeln() не будет приводить к переходу к новой строке. Более того, важно понимать, что в <pre>-блоке обычный текст отображается в том виде, в котором он непосредственно указан в документе. Также имеет значение, где именно размещен блок сценария в <pre>-блоке. Например, если для открывающего <script>-дескриптора сделать отступ, то с отступом будет отображаться и текст, выводимый первой командой с методом writeln(). Поэтому обычно вместо метода writeln() проще использовать метод write(), добавив в конец отображаемого методом текста инструкцию
 перехода к новой строке. Также стоит напомнить, что методами write() и writeln() запись выполняется при загрузке документа. Для изменения содержимого уже загруженного документа обычно используют иные средства — в частности, свойство innerHTML для элементов, содержимое которых следует изменить.

Программное создание документа

Еще один небольшой пример, который мы рассмотрим далее, дает некоторое представление о создании документа программными методами. Правда, нас будет интересовать не столько сам созданный документ, как способы использования свойств и методов объекта документа document. Код исходного документа представлен в листинге 7.16.

Листинг 7.16. Программное создание документа (файл Listing07_16.html)

```
<!DOCTYPE HTML>
<html><head><title>Листинг 7.16</title>
  <!-- Сценарий с функцией -->
  <script>
    // Функция вызывается для создания нового документа:
    function create(){
      // Переменная для записи ссылки на новое окно:
      var wnd
      // Открывается новое окно:
      wnd=window.open()
      // В новом окне создается пустой документ:
      wnd.document.open()
      // Запись в документ инструкции, определяющей
      // тип созданного документа:
      wnd.document.writeln("<!DOCTYPE HTML>")
      // Запись начальных дескрипторов:
      wnd.document.writeln("<html>")
      wnd.document.writeln("<head><title></title></head>")
      // Вписываем дескриптор тела документа:
      wnd.document.writeln("<body>")
      // Добавляем блок заголовка:
      wnd.document.writeln("<h3>Новый документ</h3>")
      // Дописываем текстовый блок:
      wnd.document.writeln("<p>Этот документ создан с помощью сценария.</p>")
      // Дописываем текстовый блок в созданный документ:
      wnd.document.writeln("<p>Этот блок записан с помощью метода <code>writeln()</code>.</p>")
      // Создание нового элемента для текстового блока:
      var element=wnd.document.createElement("p")
      // Содержимое блока:
      element.innerHTML="Этот блок создан с помощью метода <code>createElement()</code>."
      // Коллекция элементов для текстовых блоков
      // созданного документа:
      var plist=wnd.document.getElementsByTagName("p")
```

```
      // Вставка созданного блока перед
      // последним текстовым блоком:
      wnd.document.body.insertBefore(element,plist[plist.length-1])
      // Коллекция элементов для блоков сценария
      // исходного документа:
      var slist=document.getElementsByTagName("script")
      // Дописываем текстовый блок в созданный документ:
      wnd.document.writeln("<p>В исходном документе последний сценарий содержит команду:</p>")
      // Код последнего сценария из исходного документа:
      wnd.document.writeln("<code>"+slist[slist.length-1].innerHTML+"</code>")
      // Закрывающие дескрипторы:
      wnd.document.writeln("</body>")
      wnd.document.write("</html>")
      // Созданный документ закрывается для записи:
      wnd.document.close()
      // Считывание содержимого в текстовом поле
      // исходного документа:
      var txt=document.getElementById("myfield").value
      // Если поле пустое:
      if(txt==""){
        txt="Документ без названия"
      }
      // Определяем название созданного документа:
      wnd.document.title=txt
    }
  </script>
</head>
<body><h3>Листинг 7.16</h3><hr>
  <h3>Программное создание документа</h3>
  <p>Для создания нового документа следует щелкнуть кнопку. Название документа вводится
  в текстовое поле. По умолчанию (если поле пустое) используется название <b>Документ
  без имени</b></p>
  <!-- Текстовое поле -->
  <input id="myfield" type="text" size="50">
  <!-- Кнопка -->
```

```
  <input type="button" value="Создать новый документ" onclick="create()">
  <!-- Линия -->
  <hr>
  <!-- Свойства документа -->
  <h3>Некоторые свойства документа</h3>
  <!-- Кодировка -->
  <p>В документе использована кодировка <b>
    <script>
      // Получение данных о кодировке документа:
      document.write(document.characterSet)
    </script>
  </b></p>
  <!-- Размещение документа -->
  <p>Размещение документа <b>
    <script>
      // Получение данных о размещении документа:
      document.write(document.URL)
    </script>
  </b></p>
  <!-- Состояние загрузки документа -->
  <p>Состояние документа <b>
    <script>
      // Определение статуса загрузки документа:
      document.write(document.readyState)
    </script>
  </b></p>
  <!-- Линия -->
  <hr>
  <!-- Последний сценарий в документе -->
  <script>document.write("Выполнен последний сценарий.")</script>
  <br>
  <!-- Текст -->
  Документ загружен. Это последняя строка в документе.
</body>
</html>
```

Документ, несмотря на свой объем, простой. В частности, в его <head>-блоке описана функция create(), которая вызывается для создания нового документа. В <body>-блоке документа описано несколько текстовых <p>-блоков, содержащих сценарии. Также документ содержит описание кнопки и текстового поля для ввода названия создаваемого документа. Текстовое поле создается в виде <input>-блока. В данном блоке использован атрибут id со значением "myfield", который используется в функции create() при считывании содержимого поля. Атрибут type со значением "text" определяет тип данного элемента (то есть речь идет о текстовом поле ввода), а атрибут size со значением "50" задает ширину поля ввода.

Кнопка также описывается с помощью <input>-блока. Но теперь у атрибута type значение "button", поскольку речь идет о кнопке. Значение "Создать новый документ" атрибута value определяет текст, отображаемый на кнопке. Также для кнопки задано значение "create()" для атрибута onclick. Здесь мы определяем обработчик события, связанного со щелчком по кнопке. А именно при щелчке по кнопке будет вызываться функция create(), описанная в сценарии в <head>-блоке.

Несколько сценариев использовано внутри небольших <p>-блоков в документе для получения свойств документа. Например, с помощью инструкции document.characterSet мы определяем используемую в документе кодировку, инструкция document.URL позволяет получить адрес размещения документа, а с помощью инструкции document.readyState получаем статус готовности документа. Последним сценарием в документе выполняется команда document.write("Выполнен последний сценарий."), которой просто выводится текст, а завершает тело документа текст, не выделенный в отдельный <p>-блок. Как выглядит документ, будучи открытым в браузере, показано на рис. 7.46.

НА ЗАМЕТКУ

Следует отметить, что сценарии в <body>-блоке документа выполняются по мере загрузки документа. В частности, инструкция document.readyState выполняется в процессе загрузки документа, поэтому значением свойства является loading.

Если ничего не вводить в текстовое поле и щелкнуть по кнопке **Создать новый документ**, то будет создан документ с названием **Документ без названия**, которое отображается на вкладке документа, как это показано на рис. 7.47.

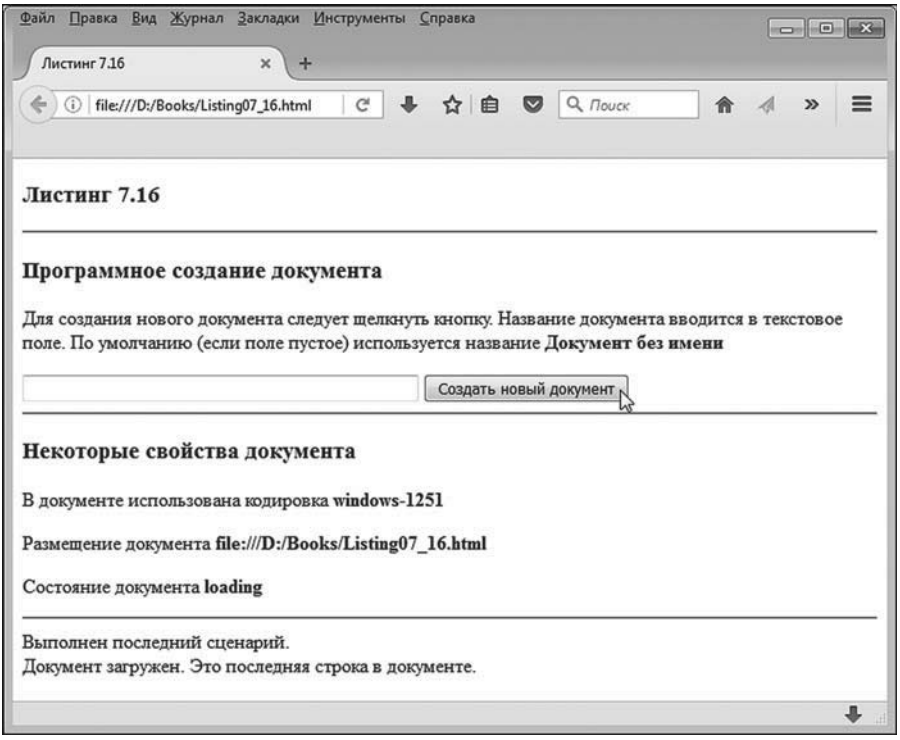

Рис. 7.46. Исходный документ открыт в окне браузера. Перед щелчком по кнопке **Создать новый документ** поле с названием нового документа пустое

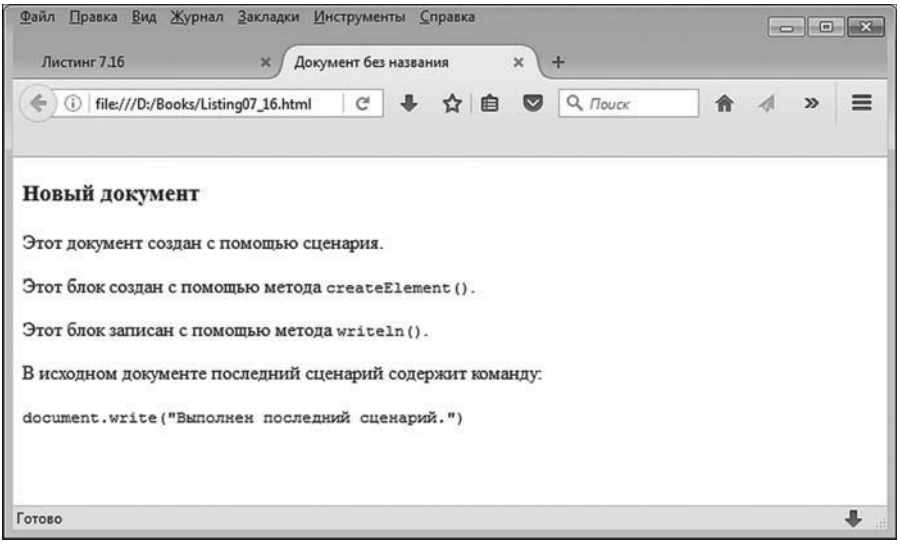

Рис. 7.47. Создан новый документ с названием по умолчанию

Мы можем указать название документа в поле ввода перед созданием, как показано на рис. 7.48.

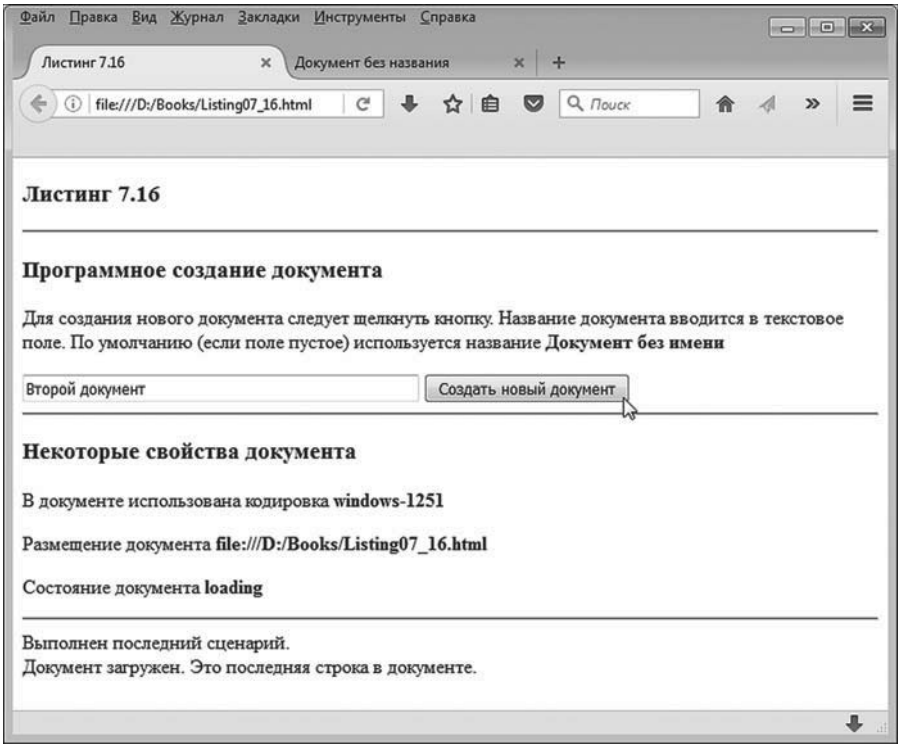

Рис. 7.48. Перед созданием нового документа в текстовом поле указано название для создаваемого документа

В данном случае перед щелчком по кнопке **Создать новый документ** мы ввели в текстовое поле название **Второй документ** для нового документа. В результате создается практически такой же документ, как и в предыдущем случае, но только с другим названием. Ситуация проиллюстрирована на рис. 7.49.

В любом случае при щелчке по кнопке создается новый документ, содержимое которого определяется кодом функции create(), вызываемой для создания документа. Проанализируем код этой функции.

В программном коде функции командой wnd=window.open() открывается новое окно, ссылка на которое записывается в переменную wnd. Далее командой wnd.document.open() в созданном окне открывается новый документ. После этого начинается формирование содержимого докумен-

та. Например, командой wnd.document.writeln("<!DOCTYPE HTML>") в документ записывается начальная инструкция, определяющая тип документа. Аналогичным образом в документ вносятся прочие открывающие и закрывающие дескрипторы.

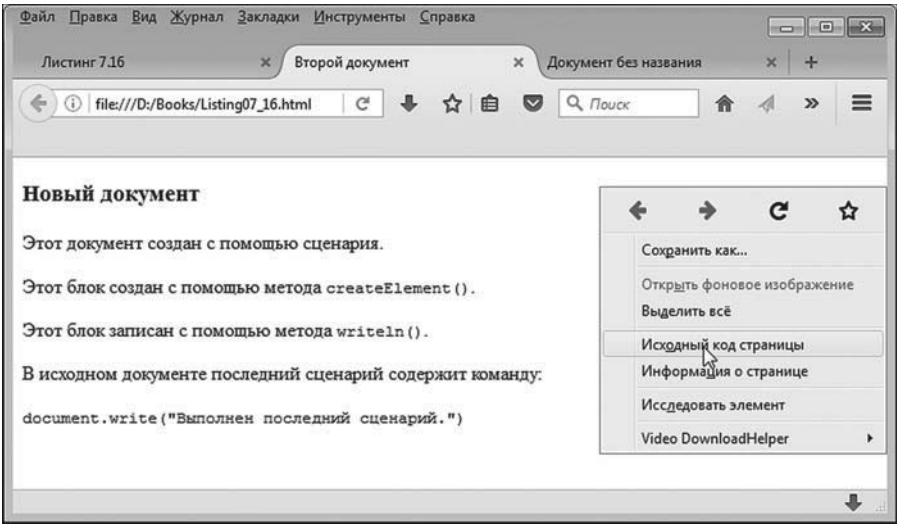

Рис. 7.49. Создан документ с указанным названием

НА ЗАМЕТКУ

Мы намеренно оставляем <title>-блок названия документа пустым.

Таким же способом мы добавляем в документ некоторые текстовые блоки. Однако один блок создается особым способом. Сначала командой var element=wnd.document.createElement("p") создается новый элемент для текстового блока. Ссылка на созданный элемент записывается в переменную element. Тип создаваемого элемента определяется текстовым значением, переданным аргументом методу createElement(). Данное текстовое значение — это название дескриптора, соответствующего создаваемому блоку. Поскольку мы создаем элемент для <p>-блока, то аргументом методу createElement() передается текст "p". Созданный таким образом элемент пустой. Туда нужно добавить содержание. Делаем это присваиванием значения свойству innerHTML объекта element. Но создание элемента не означает, что он появляется в документе. Элемент в документ следует добавить. Для добавления элемента в документ мы используем метод insertBefore(). В таком случае нам необходимо указать не только добавляемый элемент, но еще и элемент, перед

которым выполняется вставка. Мы планируем вставить элемент перед последним (на данный момент) <p>-блоком. Для определения последнего текстового блока мы сначала с помощью команды var plist=wnd.document.getElementsByTagName("p") в переменную plist записываем ссылку на коллекцию всех элементов для <p>-блоков. Коллекция возвращается методом getElementsByTagName(). Ссылка на последний элемент в этой коллекции выполняется инструкцией plist[plist.length-1]. Здесь мы учли, что размер коллекции определяется свойством length, а индекс последнего элемента в коллекции на единицу меньше размера коллекции (поскольку индексация элементов начинается с нуля).

НА ЗАМЕТКУ

Методом getElementsByTagName() возвращается коллекция элементов для блоков определенного типа. Тип элементов определяется названием соответствующего дескриптора, которое в виде текста передается аргументом методу. Значение "p" означает, что коллекция формируется из элементов для <p>-блоков.

Вставка элемента выполняется командой wnd.document.body.insertBefore(element,plist[plist.length-1]). Стоит заметить, что элемент добавляется в <body>-блок документа, поэтому метод insertBefore() вызывается не из объекта document, а из объекта body, являющегося свойством объекта document.

НА ЗАМЕТКУ

То есть в данном случае мы используем метод insertBefore() не объекта document, а объекта body.

Также мы хотим получить доступ к коду последнего сценария в исходном документе. Для этого командой var slist=document.getElementsByTagName("script") в переменную slist записывается ссылка на коллекцию из элементов для сценариев, которые с позиции гипертекстовой разметки являются <script>-блоками. Поэтому аргументом методу getElementsByTagName() передается текст "script". Ссылка на последний элемент в данной коллекции выглядит как slist[slist.length-1]. Содержимое элемента сценария получаем с помощью свойства innerHTML.

После выполненных манипуляций командой wnd.document.close() созданный документ закрывается для записи. Но сценарий продолжает выполняться, и командой var txt=document.getElementById("myfield").value из текс-

тового поля исходного документа считывается значение, введенное в поле. Полученное значение записывается в переменную txt. Для получения доступа к полю мы вызвали метод getElementById() из объекта document. Аргументом методу передается значение атрибута id элемента, доступ к которому следует получить. В рассматриваемом примере текстовое поле описано со значением "myfield" для атрибута id, поэтому в качестве результата метод возвращает ссылку на элемент для поля ввода. Свойство value возвращает значение элемента — в данном случае это текст, введенный в поле. После этого в условном операторе проверяется условие txt=="", истинное, если поле пустое (и в переменную txt, как следствие была записана пустая текстовая строка). Если так, то командой txt="Документ без названия" переменной txt присваивается новое значение (название по умолчанию для созданного документа). Командой wnd.document.title=txt текстовое значение из переменной txt присваивается свойству title объекта document из окна wnd. Данное свойство определяет название документа.

НА ЗАМЕТКУ

Если мы хотим получить ссылку на объект исходного документа, то используем инструкцию document (которая является сокращенной формой инструкции window.document). Если нам нужна ссылка на объект созданного документа, мы используем инструкцию wnd.document.

Для созданного документа можно просмотреть HTML-код. Окно с HTML-кодом созданного документа представлено на рис. 7.50.

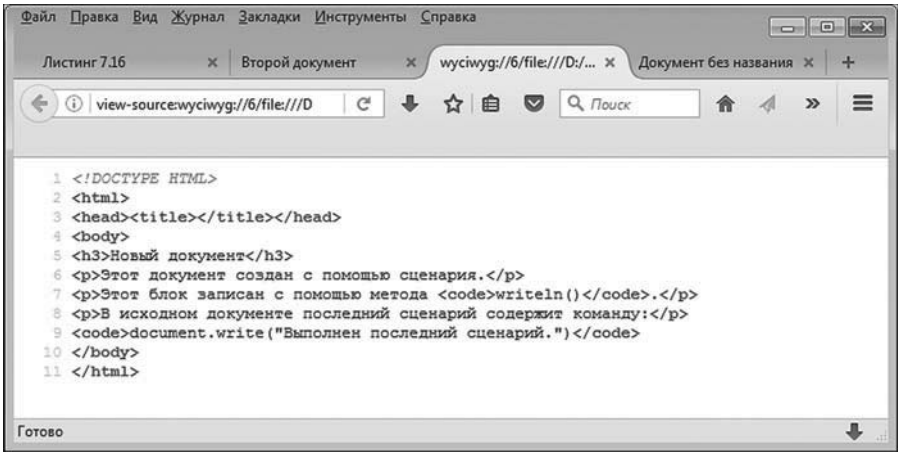

Рис. 7.50. Окно с HTML-кодом созданного документа

В частности, HTML-код, отображаемый для созданного нами документа, выглядит так, как показано ниже:

```
<!DOCTYPE HTML>
<html>
<head><title></title></head>
<body>
<h3>Новый документ</h3>
<p>Этот документ создан с помощью сценария.</p>
<p>Этот блок записан с помощью метода <code>writeln()</code>.</p>
<p>В исходном документе последний сценарий содержит команду:</p>
<code>document.write("Выполнен последний сценарий.")</code>
</body>
</html>
```

Мы видим, что один из текстовых блоков, отображаемых в окне браузера, в коде не представлен. Речь идет именно о том блоке, который был создан с помощью метода createElement(). Более того, если для вновь созданного документа выполнить перезагрузку в окне браузера, то получим результат, как на рис. 7.51.

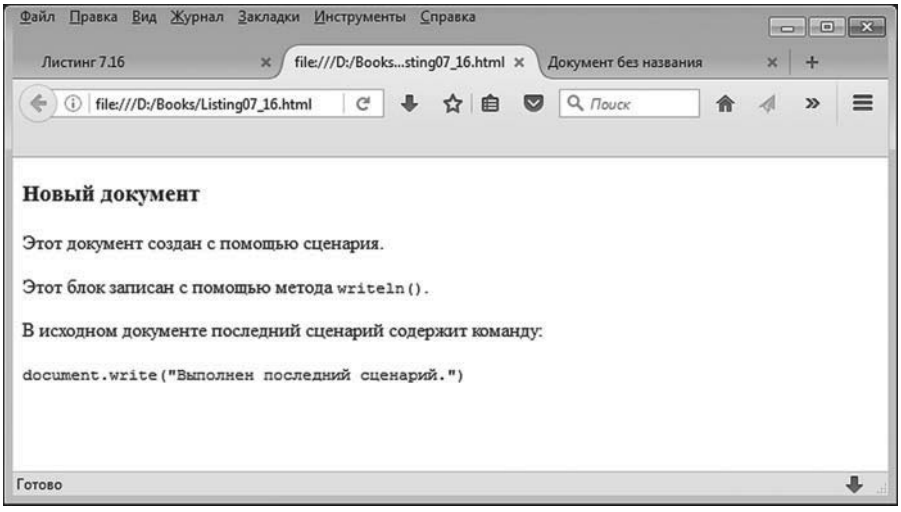

Рис. 7.51. Созданный документ после перезагрузки «теряет» один из блоков и название

Здесь важны два момента: документ «теряет» название (теперь на корешке вкладки выводится путь к исходному документу), а также

пропадает текстовый блок, созданный в сценарии с помощью метода createElement().

> **НА ЗАМЕТКУ**
>
> Для записи HTML-кода в созданный документ программными методами мы используем в основном метод writeln(). В принципе можно было использовать и метод write(), но тогда при просмотре HTML-кода созданного документа мы бы увидели весь код, отображенный в одной строке. Использование метода writeln() позволяет выполнять переход к новой строке на уровне представления HTML-кода созданного документа.

Резюме

Аллес гемахт, Маргарита Павловна!

из к/ф «Покровские ворота»

Таким образом, в этой главе мы узнали следующее.

- Сценарий в документе может быть описан непосредственно в <script>-блоке. Сценарий может быть описан в отдельном файле и включен в сценарий с помощью атрибута src в <script>-блоке. Сценарий может быть использован в гиперссылке или в описании кода для обработчиков событий.
- Для отображения диалоговых окон могут использоваться методы объекта окна window: метод alert() отображает окно с сообщением, метод prompt() отображает диалоговое окно с полем ввода, метод confirm() отображает диалоговое окно с сообщением и кнопками **OK** и **Cancel** (**Отмена**).
- Открыть и закрыть окно можно соответственно с помощью методов open() и close() объекта окна window. Аналогичные (с такими же названиями) методы используются для открытия и закрытия документа. Сам документ реализуется через объект document.
- Метод setTimeout() объекта окна window позволяет выполнить команду с задержкой во времени, метод setInterval() используется для периодического выполнения команд, а методы clearTimeout() и clearInterval() позволяют прекращать соответствующие процессы.
- Объекты window и document обладают широким набором свойств, методов и обработчиков, позволяющих выполнять с этими объектами всевозможные манипуляции и настройки.

Глава 8
ЭЛЕМЕНТЫ УПРАВЛЕНИЯ И ОБРАБОТКА СОБЫТИЙ

Вы кандидат? Вы давно уже мастер! Ох вы и мастер!

из к/ф «Покровские ворота»

В этой главе мы «погрузимся» вглубь документа, рассмотрим некоторые аспекты, связанные с применением сценариев для управления содержимым веб-документа на уровне его структуры, а также научимся манипулировать с элементами в документе. Некоторые подобные примеры мы уже рассматривали ранее. Здесь проблема будет исследована более основательно.

Элементы управления в веб-документе

Я не имею права подписывать такие исторические документы!

из к/ф «Иван Васильевич меняет профессию»

Веб-документ может содержать различные элементы управления — такие как кнопки, поля ввода, раскрывающиеся списки, опции, переключатели и некоторые другие. Перечисленные элементы добавляются в документ средствами HTML. Сценарии помогают «оживить» их — проще говоря, придать элементам нужную функциональность.

НА ЗАМЕТКУ

Элементы управления вроде полей, опций, переключателей и кнопок нередко используются не сами по себе, а в составе такого элемента, как форма. Форма представляет собой контейнер с элементами управления и предназначена для передачи на сервер настроек, выполненных пользователем в форме.

Документ может содержать не только элементы управления, но и встроенные объекты (например, рисунки), которые с помощью сценариев легко сделать активными узлами документа. Обо всем этом далее и пойдет речь. Для удобства восприятия мы будем рассматривать не каждый элемент в отдельности, а сразу по несколько элементов.

Кнопки и поля ввода

И с кнопками, и с полями ввода мы уже неоднократно имели дело. Очевидным является и назначение этих элементов: в поле ввода вводится некоторый текст, а по кнопке можно щелкнуть (запустив тем самым определенный процесс).

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Поле ввода создается с помощью <input>-блока (закрывающего дескриптора у блока нет). Атрибуту type присваивается значение "text". Ширина поля (в символах) определяется значением атрибута size. Содержимое текстового поля задают через атрибут value.

Кнопку также можно создать с помощью <input>-блока, но только в этом случае значением атрибута type должно быть "button". Название кнопки определяется атрибутом value, а геометрические размеры (ширина и/или высота) задаются через атрибут стиля style. Например, инструкция style="width:200px;height:30px;" задает для кнопки ширину в 200 пикселей и высоту в 30 пикселей.

Еще для создания кнопки может использоваться блок <button>. Название кнопки размещают между открывающим <button> и закрывающим </button> дескрипторами. Размеры кнопки можно определить с помощью атрибута style, как это описано выше. Желательно также указать значение "button" для атрибута type. Кроме значения "button", атрибут type может принимать значения "reset" и "submit", которые используются при создании кнопок очистки формы и отправки данных формы.

С позиции написания сценария для элементов правления, в том числе полей ввода и кнопок, важно знать, на какие события элементы могут реагировать.

НА ЗАМЕТКУ

В принципе, если мы имеем дело с некоторым элементом управления, то задача может заключаться в изменении вида и способа

отображения этого элемента в документе или/и его функциональных возможностях. Настройку внешнего вида элемента обычно выполняют средствами HTML. Поэтому если мы не собираемся управлять способом отображения элемента в динамическом режиме, то остается применять сценарии для обработки событий, связанных с элементом.

Список событий, доступных для обработки, достаточно большой. Но набор наиболее часто используемых событий не такой уж и значительный. В отношении кнопки обычно обрабатывают события, связанные со щелчком по кнопке, наведением на область кнопки курсора мыши или перемещением курсора мыши за область кнопки.

При работе с полями используют обработку событий, связанных с вводом символов в поле, наведением курсора на область поля и перемещением курсора мыши за область поля. Поле часто используется и как статический элемент по своему прямому назначению — для записи и считывания текста.

Мы рассмотрим пример, в котором присутствуют кнопки и поле ввода. В примере используется обработка событий. Но, прежде чем приступить к анализу программного кода, мы рассмотрим функциональные возможности соответствующего документа. Как выглядит документ, открытый в окне браузера, показано на рис. 8.1.

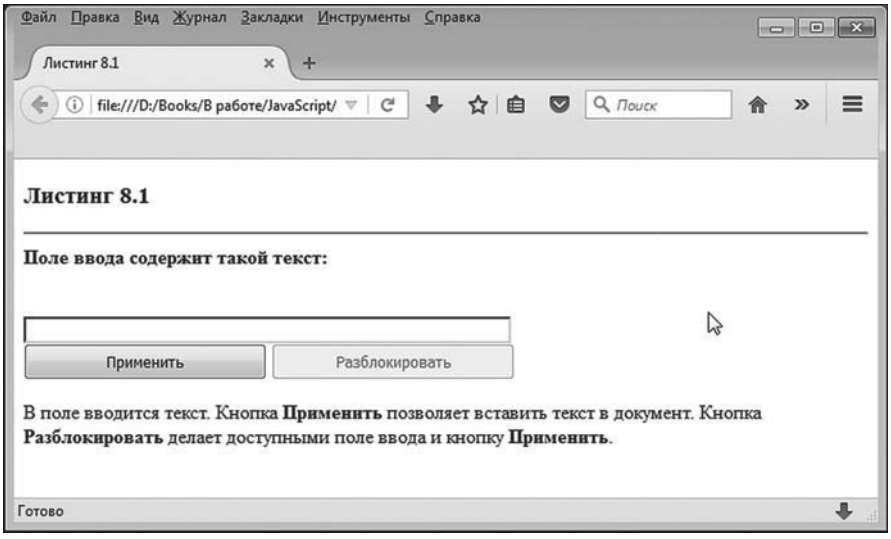

Рис. 8.1. Документ с кнопками и полем ввода открыт в окне браузера

Документ содержит поле ввода и две кнопки: одна называется **Применить**, другая называется **Разблокировать**. В начальный момент кнопка **Разблокировать** неактивна (заблокирована). Над полем ввода имеется область, предназначенная для отображения текста (при загрузке документа эта область пустая). Под кнопками отображается текст. Если навести курсор мыши на кнопку **Применить**, то этот текст изменится, как показано на рис. 8.2.

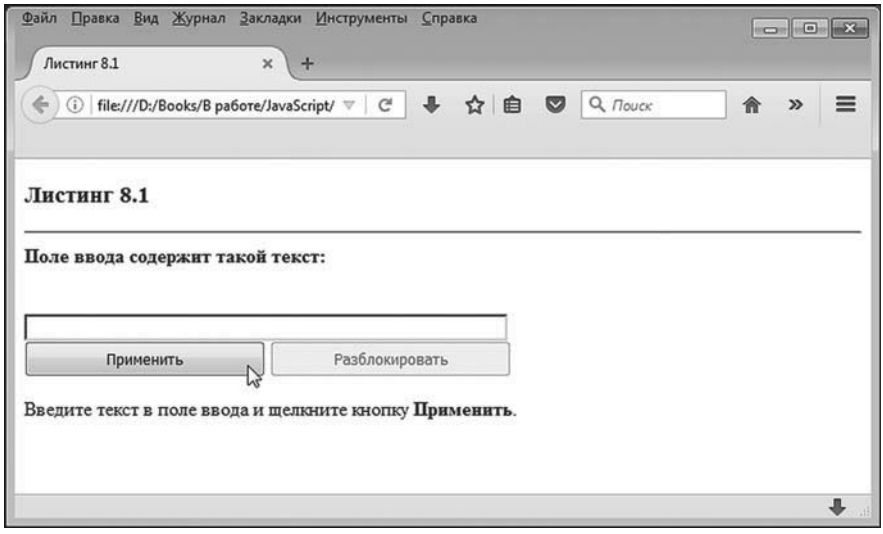

Рис. 8.2. При наведении курсора мыши на незаблокированную кнопку **Применить** в нижней части документа меняется текст

При перемещении курсора мыши за область кнопки **Применить** снизу отображается исходный (как при загрузке документа) текст. Если навести курсор мыши на область поля ввода, то фон поля ввода станет желтым, а рамка вокруг поля ввода отображается красным цветом. Такая ситуация проиллюстрирована на рис. 8.3.

Если переместить курсор мыши за пределы поля ввода, фон поля станет белым, а рамка — серой (такой вид поле имеет с самого начала при загрузке документа). На рис. 8.4 показана ситуация, когда в поле ввода введен текст, а курсор мыши наведен на кнопку **Применить** (но щелчок еще не выполнен).

После щелчка по кнопке **Применить** текст из поля ввода отображается над этим полем курсивом увеличенного размера красного цвета. Поле становится неактивным (в него нельзя ввести текст), а при на-

ведении на поле курсора мыши цвет поля не меняется. Также становится недоступной кнопка **Применить**, зато отменяется блокировка кнопки **Разблокировать**. Как в такой ситуации выглядит документ, показано на рис. 8.5.

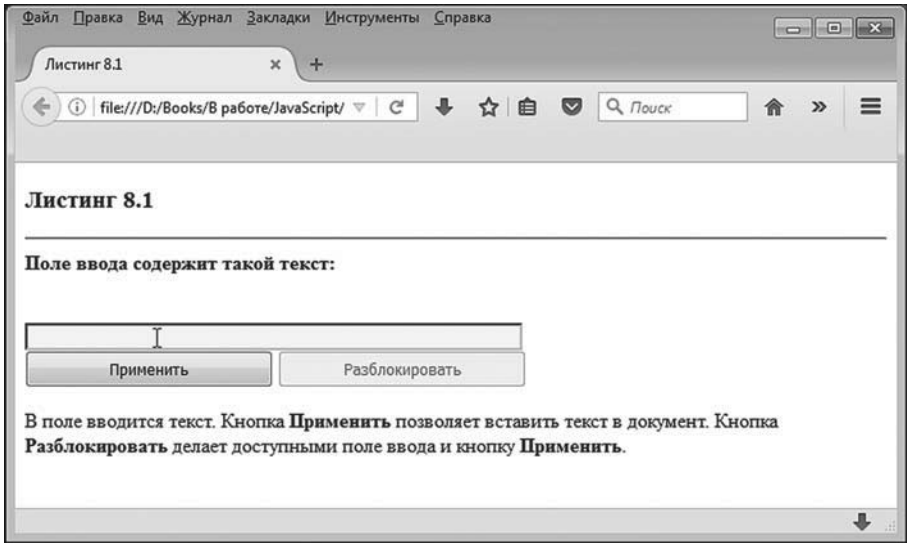

Рис. 8.3. При наведении курсора мыши на поле ввода фон поля становится желтым, а рамка вокруг поля — красной

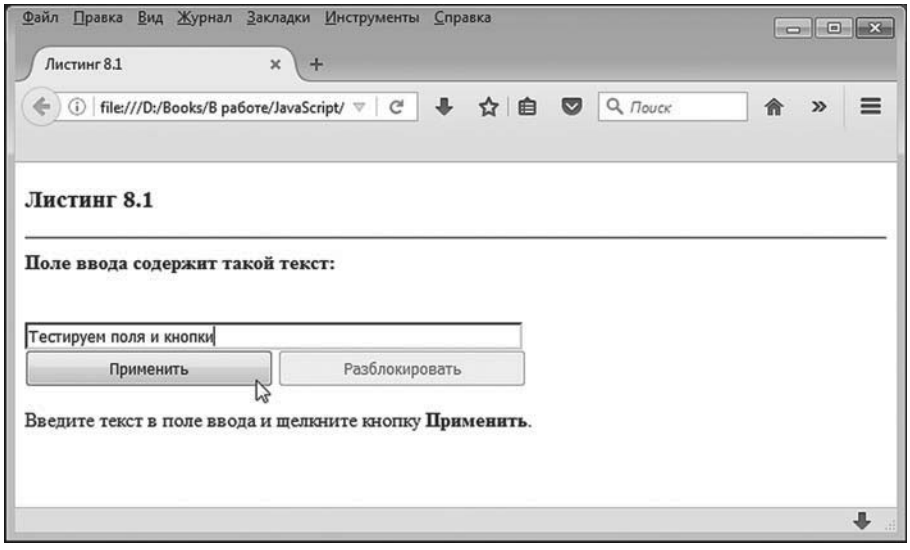

Рис. 8.4. Документ с введенным в поле текстом перед щелчком по кнопке **Применить**

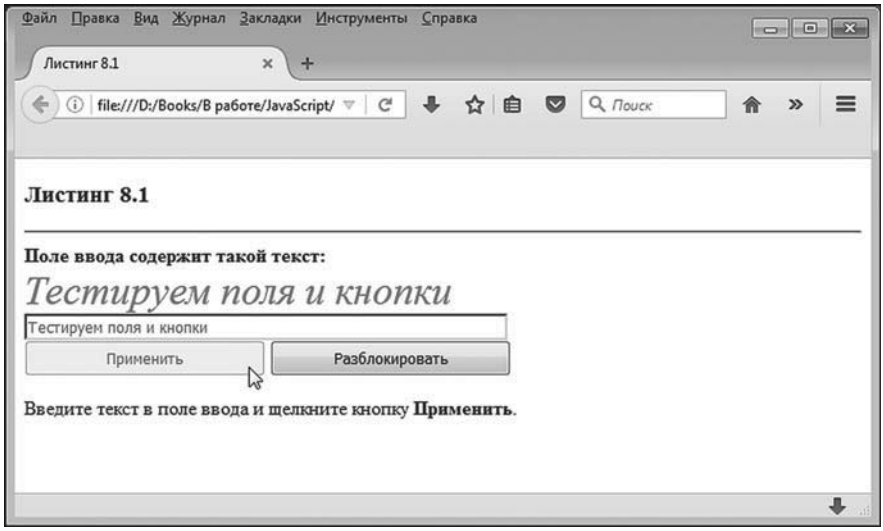

Рис. 8.5. Документ после щелчка по кнопке **Применить**: поле и кнопка **Применить** становятся неактивными, активной становится кнопка **Разблокировать**, а текст из поля ввода отображается в документе (курсивом красного цвета)

При наведении курсора мыши на незаблокированную кнопку **Разблокировать** в нижней части документа меняется текст подсказки, как это показано на рис. 8.6.

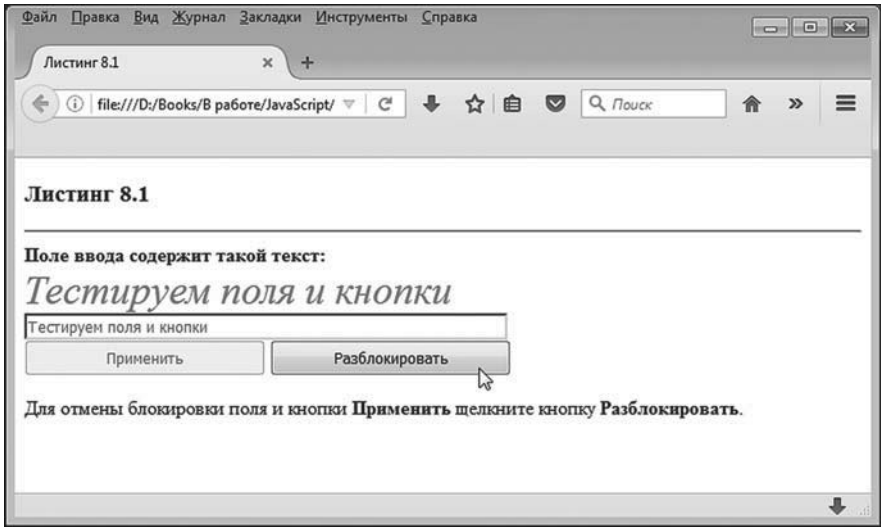

Рис. 8.6. При наведении курсора мыши на кнопку **Разблокировать** в нижней части документа изменяется текст подсказки

Щелчок по кнопке **Разблокировать** приводит к разблокированию кнопки **Применить**, очистке и разблокированию поля ввода. Кнопка **Разблокировать** становится неактивной. Как выглядит документ после щелчка по кнопке **Разблокировать,** показано на рис. 8.7.

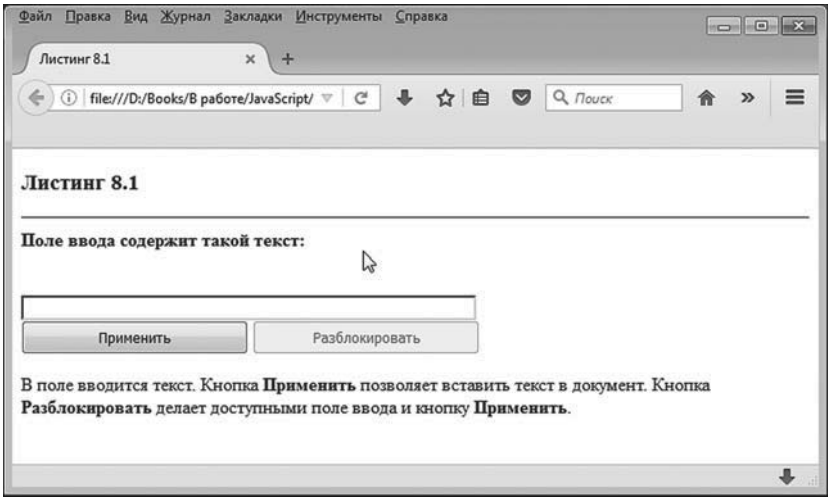

Рис. 8.7. Вид документа после щелчка по кнопке **Разблокировать**: кнопка **Разблокировать** блокируется, отменяется блокирование кнопки **Применить** и поля ввода, выполняется очистка поля

Фактически документ возвращается к состоянию первой загрузки.

Вкратце мы описали функциональные возможности документа. Теперь проанализируем код, которым эти возможности реализуются. Код документа представлен в листинге 8.1.

Листинг 8.1. Кнопки и поле ввода

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 8.1</title>
<!-- Начало сценария -->
<script type="text/javascript">
  // Переменные с текстом подсказки:
  var msgA="В поле вводится текст. Кнопка <b>Применить</b> позволяет вставить текст
  в документ. Кнопка <b>Разблокировать</b> делает доступными поле ввода и кнопку
  <b>Применить</b>."
```

```
  var msgB="Введите текст в поле ввода и щелкните кнопку <b>Применить</b>."
  var msgC="Для отмены блокировки поля и кнопки <b>Применить</b> щелкните кнопку
<b>Разблокировать</b>."
  // Переменные для записи ссылки на поле ввода,
  // кнопки и блоки с текстом:
  var tf,ab,ub,mt,mp
  // Обработка события, связанного с загрузкой документа:
  window.onload=function(){
    // Ссылка на текстовое поле:
    tf=document.getElementById("textField")
    // Ссылка на кнопку "Применить":
    ab=document.getElementById("applyButton")
    // Ссылка на кнопку "Разблокировать":
    ub=document.getElementById("unlockButton")
    // Ссылка на блок с текстом:
    mt=document.getElementById("mytext")
    // Ссылка на блок с подсказкой:
    mp=document.getElementById("myprompt")
    // Вызов функции для определения вида поля ввода:
    setColor(false)
    // Пустая строка в поле ввода:
    tf.value=""
    // Поле активно (не заблокировано):
    tf.disabled=false
    // Кнопка "Применить" активна (не заблокирована):
    ab.disabled=false
    // Кнопка "Разблокировать" неактивна:
    ub.disabled=true
    // Текст подсказки:
    mp.innerHTML=msgA
    // Обработчик события, связанного с наведением
    // курсора мыши на поле ввода:
    tf.onmouseover=function(){
      // Если поле не заблокировано:
      if(!this.disabled){
```

```
      setColor(true)
    }
  }
  // Обработчик события, связанного с перемещением
  // курсора мыши за пределы поля ввода:
  tf.onmouseout=function(){
    // Если поле не заблокировано:
    if(!this.disabled){
      setColor(false)
    }
  }
  // Обработка щелчка на кнопке "Применить":
  ab.onclick=function(){
    // Текст из поля копируется в документ:
    mt.innerHTML=tf.value
    // Блокируется поле ввода:
    tf.disabled=true
    // Отменяется блокировка кнопки "Разблокировать":
    ub.disabled=false
    // Блокирование кнопки "Применить":
    ab.disabled=true
  }
  // Обработка события, связанного с наведением курсора
  // мыши на кнопку "Применить":
  ab.onmouseover=function(){
    // Если кнопка не заблокирована:
    if(!this.disabled){
      applyPrompt(msgB)
    }
  }
  // Обработка события, связанного с перемещением
  // курсора мыши за пределы кнопки "Применить":
  ab.onmouseout=function(){
    // Если кнопка не заблокирована:
```

```
    if(!this.disabled){
      applyPrompt(msgA)
    }
  }
  // Обработка щелчка на кнопке "Разблокировать":
  ub.onclick=function(){
    // Очистка поля ввода:
    tf.value=""
    // Очистка текста из блока документа:
    mt.innerHTML=""
    // Отмена блокировки поля ввода:
    tf.disabled=false
    // Отмена блокировки кнопки "Применить":
    ab.disabled=false
    // Блокирование кнопки "Разблокировать":
    ub.disabled=true
  }
  // Обработка события, связанного с наведением курсора
  // мыши на кнопку "Разблокировать":
  ub.onmouseover=function(){
    // Если кнопка не заблокирована:
    if(!this.disabled){
      applyPrompt(msgC)
    }
  }
  // Обработка события, связанного с перемещением
  // курсора мыши за пределы кнопки "Разблокировать":
  ub.onmouseout=function(){
    // Если кнопка не заблокирована:
    if(!this.disabled){
      applyPrompt(msgA)
    }
  }
}
```

```
  // Функция для изменения цвета фона и рамки поля:
  function setColor(arg){
    if(arg){
      // Желтое поле и красная рамка:
      tf.style.backgroundColor="yellow"
      tf.style.borderColor="red"
    }
    else{
      // Белое поле и серая рамка:
      tf.style.backgroundColor="white"
      tf.style.borderColor="gray"
    }
  }
  // Функция для применения текста подсказки:
  function applyPrompt(msg){
    mp.innerHTML=msg
  }
</script>
<!-- Завершение сценария -->
</head>
<body>
  <h3>Листинг 8.1</h3><hr>
  <b>Поле ввода содержит такой текст:</b><br>
  <!-- Блок с курсивом для отображения содержимого поля ввода. Вначале блок пустой
  (как и поле ввода) -->
  <i id="mytext" style="font-size:25pt;color:red;"></i><br>
  <!-- Поле ввода -->
  <input type="text" id="textField" size="62"><br>
  <!-- Кнопка "Применить" -->
  <input type="button" id="applyButton" value="Применить" style="width:200px;height:30px;">
  <!-- Кнопка "Разблокировать" -->
  <button id="unlockButton" style="width:200px;height:30px;">Разблокировать</button>
  <!-- Текст с подсказкой -->
  <p id="myprompt"></p>
</body>
</html>
```

Анализ кода документа начнем с описания <body>-блока, в котором содержатся HTML-инструкции, приводящие к созданию поля ввода и кнопок. Кроме заголовка (создается <h3>-дескриптором), в документе есть блок текста, выделяемого жирным шрифтом (использован <b>-блок). Это статический текст, который постоянно отображается в документе. После него следует пустой (без текста) <i>-блок курсивного текста. В блоке указан атрибут id со значением "mytext". Через этот атрибут мы будем в сценарии получать доступ к блоку для добавления в него текста.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Атрибут style со значением "font-size:25pt;color:red;" определяет стиль отображения курсивного текста, в соответствии с которым используется шрифт размера 25 красного цвета.

Поле ввода создается с помощью <input>-блока. Для получения доступа к полю ввода из сценария мы задаем атрибут id для данного элемента. Значение атрибута равно "textField".

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Значение атрибута type равно "text", и именно поэтому «на выходе» получаем поле ввода. Атрибут size со значением "62" определяет ширину поля ввода (в символах).

Кнопка с названием **Применить** также создается на основе <input>-блока. Но только на этот раз значением атрибута type указан текст "button", поэтому создается кнопка (напомним, что для поля ввода значение атрибута type было "text"). Атрибут id получает значение "applyButton". Название кнопки определяется значением "Применить" для атрибута value. Еще одна кнопка создается с помощью <button>-блока. Название кнопки размещается между открывающим и закрывающим дескриптором. Для кнопки **Разблокировать** атрибут id имеет значение "unlockButton".

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

В описании блоков для обеих кнопок использована инструкция style="width:200px;height:30px;" означающая, что соответствующая кнопка имеет ширину в 200 пикселей и высоту в 30 пикселей.

Наконец, под кнопками расположен текстовый <p>-блок, который вначале не содержит текста. Доступ к этому блоку из сценария осуществляется по значению "myprompt" атрибута id.

> **НА ЗАМЕТКУ**
>
> Таким образом, у нас имеется две кнопки и два блока для вставки текста. Доступ к каждому из них реализуется на основе значения атрибута id. Для кнопок значение атрибута id равно "applyButton" и "unlockButton", а для блоков с текстом значения атрибута id равны "mytext" и "myprompt".

Все прочее происходит и реализуется в блоке сценария (блок <script>), который размещен в заголовочной части документа (блок <head>).

В первую очередь стоит заметить, что в сценарии объявляются три глобальные переменные msgA, msgB и msgC, которые сразу получают текстовые значения. Это три варианта текста, который может отображаться в нижней части документа (текст подсказки). Переменные tf, ab, ub, mt и mp используются для записи в них ссылок на объекты поля ввода, кнопки **Применить**, кнопки **Разблокировать**, блока для отображения текста из поля ввода и текстового блока с подсказкой соответственно.

В сценарии описывается несколько функций. Так, функция setColor() используется для определения цвета фона и рамки для поля ввода. Аргумент функции имеет название arg. Предполагается, что это логическое значение. В теле функции в условном операторе проверяется значение аргумента, и если оно истинно, то выполняются команды tf.style.backgroundColor="yellow" и tf.style.borderColor="red".

> **НА ЗАМЕТКУ**
>
> Напомним, что переменная tf является ссылкой на объект поля ввода. Значение (в виде ссылки на объект поля ввода) переменная tf получает сразу после загрузки документа, когда выполняется обработка соответствующего события. Так что на момент вызова функции setColor() переменной tf уже присвоено «правильное» значение.

Если значение аргумента функции ложно, то выполняются команды tf.style.backgroundColor="white" и tf.style.borderColor="gray". В обоих случаях мы за-

даем значения свойствам backgroundColor и borderColor объекта style (объект стиля), который является свойством объекта tf (объект поля ввода). Свойство backgroundColor «отвечает» за цвет фона поля ввода, а свойство borderColor определяет цвет рамки вокруг поля. Следовательно, при истинном аргументе функции setColor() для поля ввода устанавливается желтый цвет фона и красный цвет рамки, а при ложном значении аргумента фон поля ввода устанавливается белым, а рамка — серой.

У функции applyPrompt() код очень простой: в теле функции выполняется всего одна команда mp.innerHTML=msg, которой аргумент функции msg присваивается значением свойству innerHTML объекта mp. Последний представляет собой ссылку на блок с текстом подсказки, которая отображается в нижней части документа (значение переменная mp получает при обработке события, связанного с загрузкой документа). Таким образом, вызов функции applyPrompt() позволяет задать новое содержимое для подсказки в конце документа.

Помимо перечисленных функций, в сценарии представлен обработчик события, связанного с загрузкой документа, который, в свою очередь, содержит описание нескольких обработчиков событий для кнопок и поля. Событие, связанное с загрузкой документа, мы описываем на уровне свойства onload объекта окна window. Значением свойству window.onload присваивается анонимная функция, в теле которой выполняется ряд команд. Начальная группа команд предназначена для получения ссылок на объекты элементов управления в документе и записи этих ссылок в глобальные переменные. Все команды однотипные: из объекта документа document вызывается метод getElementById(), а аргументом методу передается текстовое значение атрибута id того элемента, на который необходимо получить ссылку. Например, командой tf=document.getElementById("textField") в переменную tf записывается ссылка на объект элемента, атрибут id которого имеет значение "textField". Понятно, что речь идет о поле ввода. Аналогично поступаем и с прочими элементами управления.

После того как получены ссылки на интересующие нас элементы управления, командой setColor(false) задается белый цвет фона и серый цвет рамки для поля ввода. Командой tf.value="" выполняется очистка поля ввода: формально значением свойству value объекта tf присваивается пустая текстовая строка. Для объектов tf (поле ввода), ab (кнопка **Применить**) и ub (кнопка **Разблокировать**) задается значение свойства disabled, отвечающее за режим доступности элемента. Значение true

означает, что соответствующий элемент заблокирован (недоступен). Значение false означает, что элемент не заблокирован (доступен). Вначале поле ввода и кнопка **Применить** не заблокированы, а кнопка **Разблокировать** заблокирована.

Командой mp.innerHTML=msgA задается текст подсказки в нижней части документа (здесь мы также могли использовать функцию applyPrompt() с аргументом msgA). Все обозначенные команды, напомним, выполняются при загрузке документа.

Обработчик события, связанного с наведением курсора мыши на поле ввода, реализуется через анонимную функцию, которая присваивается значением свойству onmouseover объекта поля tf. В теле функции вызывается условный оператор, в котором проверяется условие !this.disabled. Условие истинно, если значение свойства disabled объекта, из которого вызывается функция (а точнее, метод), равно false. Ссылка на объект, из которого вызывается метод, возвращается через ключевое слово this — то есть речь идет об объекте поля ввода tf. Свойство disabled объекта поля ввода равно false, если поле ввода не заблокировано. Таким образом, условие !this.disabled истинно, если поле ввода не заблокировано. В этом случае командой setColor(true) для поля ввода применяется желтый цвет фона и красный цвет рамки. В противном случае (если поле заблокировано) не происходит ничего.

> **НА ЗАМЕТКУ**
>
> Цвет фона поля ввода меняется при наведении курсора мыши, но только если поле не заблокировано.

Аналогичным образом организована обработка события, связанного с перемещением курсора мыши за пределы поля ввода. Она реализована через свойство onmouseout объекта tf. Если поле не заблокировано, то при перемещении курсора за пределы области поля выполняется команда setColor(false), в результате чего цвет фона поля ввода будет белым, а рамка вокруг поля — серой.

События, связанные с наведением курсора мыши на элемент и перемещением курсора мыши за пределы элемента, обрабатываются и для кнопок. Речь идет о свойствах onmouseover и onmouseout объектов ab и ub. Обработка состоит в том, что при активном элементе (кнопке) вызывается функция applyPrompt(), аргументом которой передается

переменная с текстовым значением, определяющим текст подсказки в нижней части документа.

Событие, связанное со щелчком по кнопке **Применить**, обрабатывается так (код функции, присвоенной свойству onclick объекта ab).

- Командой mt.innerHTML=tf.value текст из поля ввода (значение свойства value объекта tf) копируется в документ (вставляется в блок mt как внутренний HTML-код — реализуется через свойство innerHTML).
- Командой tf.disabled=true блокируется поле ввода.
- Командой ub.disabled=false отменяется блокировка кнопки **Разблокировать**.
- Командой ab.disabled=true блокируется кнопка **Применить**.

Обработка щелчка по кнопке **Разблокировать** предполагает выполнение таких действий (код функции, присвоенной значением свойству onclick объекта ub):

- Командой tf.value="" выполняется очистка поля ввода.
- С помощью команды mt.innerHTML="" выполняется очистка текста в том блоке документа, который предназначен для отображения содержимого поля ввода.
- Командой tf.disabled=false отменяется блокирование поля ввода.
- Для отмены блокировки кнопки **Применить** использована команда ab.disabled=false.
- Блокирование кнопки **Разблокировать** выполняется с помощью команды ub.disabled=true.

Еще один небольшой пример, рассматриваемый далее, иллюстрирует способы выполнения настроек (с помощью сценария) для текстового поля. На рис. 8.8 показан документ, который мы будем анализировать далее. В документе отображается два поля ввода, причем текст вводить можно только в нижнее поле (верхнее поле для ввода текста недоступно). Фокус передан нижнему полю, поэтому там мигает курсор. Рамка нижнего поля пунктирная, каждая ее сторона разного цвета: левая красная, верхняя зеленая, правая синяя и нижняя черная. У верхнего поля правый нижний и верхний левый углы закруглены, а рамка пурпурного цвета. Рамки у нижнего и верхнего полей утолщенные (5 пикселей).

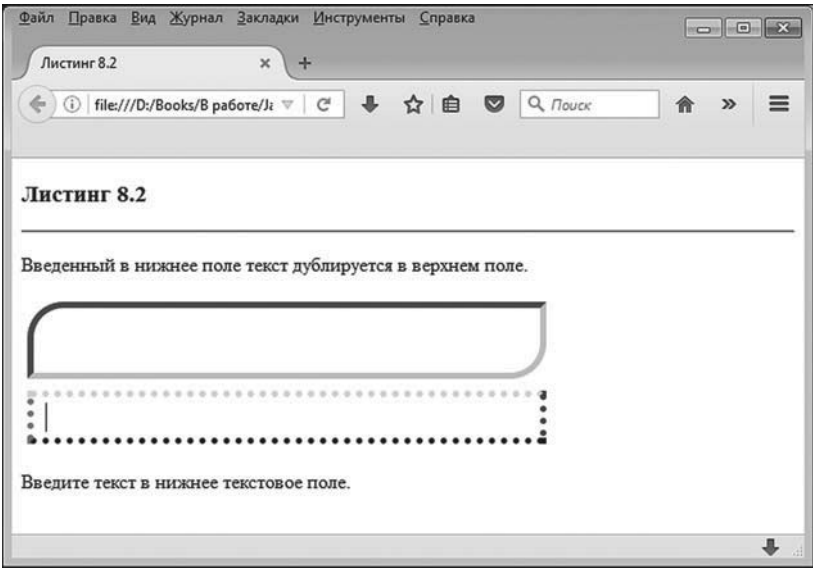

Рис. 8.8. Документ с двумя текстовыми полями

Если вводить текст в нижнее поле, то он автоматически, по мере ввода, дублируется и в верхнем поле. Ситуация проиллюстрирована на рис. 8.9.

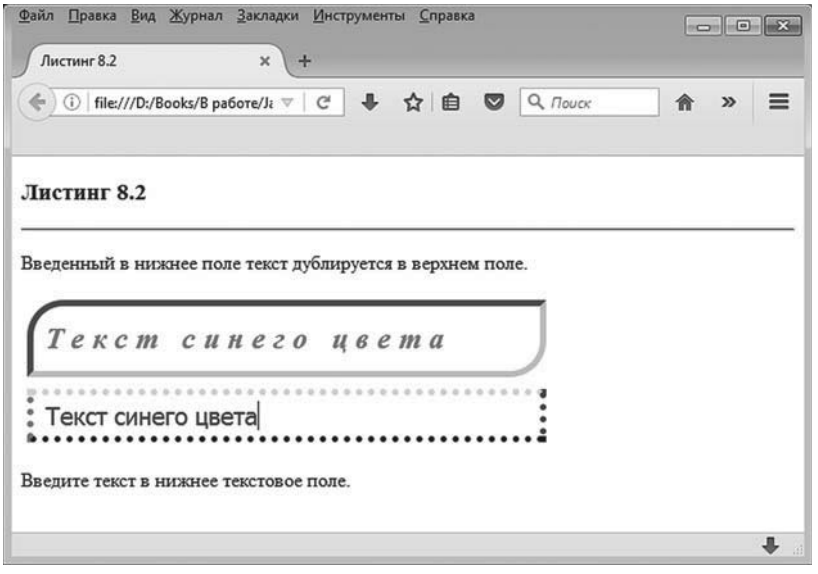

Рис. 8.9. При вводе текста в нижнее поле он автоматически дублируется в верхнем поле

В нижнем поле текст отображается синим цветом. В верхнем поле шрифт красного цвета, жирный курсив. Кроме этого, между символами в верхнем поле выдерживается определенный интервал (расстояние между соседними символами составляет 7 пикселей).

Далее мы проанализируем программный код документа, который представлен в листинге 8.2.

Листинг 8.2. Использование полей ввода

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 8.2</title>
<!-- Начало сценария -->
<script type="text/javascript">
  // Обработчик события, связанного с загрузкой документа:
  window.onload=function(){
    // Переменные для записи ссылок на поля и объекты
    // стилей полей:
    var inFld,outFld,inStl,outStl
    // Верхнее поле. Определение основных параметров.
    // Ссылка на объект верхнего поля:
    outFld=document.getElementById("outField")
    // Ссылка на объект стиля верхнего поля:
    outStl=outFld.style
    // Высота верхнего поля:
    outStl.height="50px"
    // Ширина верхнего поля:
    outStl.width="400px"
    // Белый фон для верхнего поля:
    outStl.backgroundColor="white"
    // Цвет рамки для верхнего поля:
    outStl.borderColor="magenta"
    // Радиус закругления для нижнего правого угла
    // верхнего поля:
```

```
outStl.borderBottomRightRadius="30px"
// Радиус закругления для верхнего левого угла
// верхнего поля:
outStl.borderTopLeftRadius="30px"
// Толщина рамки для верхнего поля:
outStl.borderWidth="5px"
// Внешние отступы для верхнего поля:
outStl.margin="5px"
// Внутренние отступы для верхнего поля:
outStl.padding="2px 10px 2px 10px"
// Расстояние между символами для верхнего поля:
outStl.letterSpacing="7px"
// Тип шрифта для верхнего поля:
outStl.fontFamily="Times New Roman"
// Стиль шрифта для верхнего поля:
outStl.fontStyle="italic"
// Толщина шрифта для верхнего поля:
outStl.fontWeight="900"
// Шрифт красного цвета для верхнего поля:
outStl.color="red"
// Размер шрифта для верхнего поля:
outStl.fontSize="18pt"
// Режим "только чтение" для верхнего поля:
outFld.readOnly=true
// Содержимое верхнего поля:
outFld.value=""
// Нижнее поле. Определение основных параметров.
// Ссылка на объект нижнего поля:
inFld=document.getElementById("inField")
// Ссылка на объект стиля нижнего поля:
inStl=inFld.style
// Высота нижнего поля:
inStl.height="30px"
// Ширина нижнего поля:
```

```
    inStl.width="400px"
    // Толщина рамки для нижнего поля:
    inStl.borderWidth="5px"
    // Стиль рамки (пунктирная) для нижнего поля:
    inStl.borderStyle="dotted"
    // Цвет рамки слева для нижнего поля:
    inStl.borderLeftColor="#ff0000"
    // Цвет рамки сверху для нижнего поля:
    inStl.borderTopColor="#00ff00"
    // Цвет рамки справа для нижнего поля:
    inStl.borderRightColor="#0000ff"
    // Цвет рамки снизу для нижнего поля:
    inStl.borderBottomColor="#000000"
    // Внутренние отступы для нижнего поля:
    inStl.padding="2px 10px 2px 10px"
    // Внешние отступы для нижнего поля:
    inStl.margin="5px"
    // Размер шрифта для нижнего поля:
    inStl.fontSize="15pt"
    // Синий цвет шрифта для нижнего поля:
    inStl.color="blue"
    // Содержимое нижнего поля:
    inFld.value=""
    // Передача фокуса нижнему полю:
    inFld.focus()
    // Обработчик события, связанного с отпусканием
    // клавиши, для нижнего поля:
    inFld.onkeyup=function(){
      outFld.value=this.value
    }
  }
</script>
<!-- Завершение сценария -->
</head>
```

```
<body>
  <h3>Листинг 8.2</h3><hr>
  <p>Введенный в нижнее поле текст дублируется в верхнем поле.</p>
  <!-- Верхнее поле -->
  <input type="text" id="outField"><br>
  <!-- Нижнее поле -->
  <input type="text" id="inField">
  <p>Введите текст в нижнее текстовое поле.</p>
</body>
</html>
```

В <body>-блоке документа, помимо двух вспомогательных текстовых блоков, не используемых в сценарии, есть два <input>-блока, через которые реализуются поля ввода. Верхнее поле описано со значением "outField" атрибута id, а нижнее поле описано со значением "inField" атрибута id. Сценарий, используемый в документе, состоит из описания обработчика события, связанного с загрузкой документа (свойство onload объекта окна window).

В теле соответствующей функции используются четыре переменные inFld, outFld, inStl и outStl, в которые записываются ссылки на объекты полей ввода и объекты стилей для полей ввода. В частности, ссылки на объекты полей определяются командами outFld=document.getElementById("outField") (объект верхнего поля) и inFld=document.getElementById("inField") (объект нижнего поля), а ссылки на объекты стилей определяем командами outStl=outFld.style (объект стиля для верхнего поля) и inStl=inFld.style (объект стиля нижнего поля).

НА ЗАМЕТКУ

Ссылки на объекты стилей записываются в отдельные переменные исключительно ради удобства. А поскольку ссылки на объекты полей и объекты стилей полей используются только в обработчике события загрузки документа, то соответствующие переменные объявляются в теле обработчика.

С помощью свойств height и width объекта стиля для каждого из полей задается соответственно высота и ширина поля в пикселях. Высота верхнего поля составляет 50 пикселей (команда outStl.height="50px"), вы-

сота нижнего поля составляет 30 пикселей (команда inStl.height="30px"), а ширина каждого из полей установлена равной 400 пикселям (команды outStl.width="400px" и inStl.width="400px").

Командой outStl.backgroundColor="white" для верхнего поля задается белый цвет фона. По умолчанию фон и так белый. Но для данного конкретного поля командой outFld.readOnly=true устанавливается режим «только для чтения». В такое поле нельзя ввести текст с клавиатуры. Разные браузеры в таком режиме могут отображать поля по-разному. Для большей универсальности мы задаем цвет фона в явном виде.

НА ЗАМЕТКУ

Режим «только для чтения» немного напоминает режим неактивного (заблокированного) поля. Различие между режимами в том, что заблокированный элемент игнорируется при отправке данных формы, а в режиме «только для чтения» просто нельзя ввести текст в поле. При этом обычно в таком поле можно разместить каретку ввода и выделить уже имеющийся там текст (хотя ввод с клавиатуры невозможен).

Цвет рамки вокруг верхнего поля задается командой outStl.borderColor="magenta". Когда мы задаем цвет рамки для нижнего поля, то для каждой стороны рамки используется разный цвет. Так, командой inStl.borderLeftColor="#ff0000" задается красный цвет для левой стороны рамки поля. Цвет определяем инструкцией "#ff0000" в формате RGB (код содержания красного, зеленого и синего цветов). Цвет рамки сверху определяется инструкцией inStl.borderTopColor="#00ff00" (зеленый), цвет рамки справа определяем командой inStl.borderRightColor="#0000ff" (синий), а рамка снизу отображается черным цветом (команда inStl.borderBottomColor="#000000").

Командами outStl.borderBottomRightRadius="30px" и outStl.borderTopLeftRadius="30px" задается радиус закругления в 30 пикселей для правого нижнего и левого верхнего угла рамки верхнего поля. Как следствие, эти углы рамки отображаются закругленными.

НА ЗАМЕТКУ

Радиус закругления для левого нижнего и правого верхнего углов можно задать с помощью свойств borderBottomLeftRadius и borderTopRightRadius объекта стиля style, являющегося свойством объекта поля ввода.

Толщина рамки (в пикселях) задается с помощью свойства borderWidth объекта стиля. Для обоих полей толщина рамки составляет 5 пикселей (команды outStl.borderWidth="5px" и inStl.borderWidth="5px").

Свойство margin используется для определения внешних отступов (расстояний между внешними границами) при позиционировании элементов в документе. Команды outStl.margin="5px" и inStl.margin="5px" означают, что поля позиционируются в документе с отступом между границами в 5 пикселей (по всем направлениям). Внутренние отступы (расстояние между текстом и внутренними границами полей) определяются с помощью свойства padding. Команды outStl.padding="2px 10px 2px 10px" и inStl.padding="2px 10px 2px 10px" означают наличие «зазора» в 2 (или более) пикселя между текстом и верхней границей, между текстом и правой границей оставляется 10 пикселей, между текстом и нижней границей — не менее 2 пикселей, и расстояние между текстом и левой границей равно 10 пикселям.

Расстояние между символами для верхнего поля определяется командой outStl.letterSpacing="7px", означающей, что между соседними буквами расстояние составляет 7 пикселей.

Группа команд определяет свойства шрифта, применяемого для текстовых полей. Например, командой outStl.fontFamily="Times New Roman" для верхнего поля устанавливается тип шрифта. Курсивный стиль шрифта для верхнего окна применяется с помощью команды outStl.fontStyle="italic". Чтобы сделать шрифт не только курсивом, но и жирным, используем команду outStl.fontWeight="900", определяющую толщину символов шрифта.

> **НА ЗАМЕТКУ**
>
> Значением свойства fontWeight может быть текстовое представление для положительного целого числа от 100 до 900 (с шагом дискретности 100). Число определяет толщину символов. Значение 900 соответствует жирному шрифту. Нормальному шрифту соответствует значение 400. В качестве значений свойства также могут указываться некоторые ключевые слова: например, значение bold соответствует жирному шрифту, а значение normal соответствует нормальному шрифту.

Командой outStl.color="red" для верхнего поля устанавливаем красный цвет для шрифта, а командой inStl.color="blue" для шрифта в нижнем поле ввода применяется синий цвет.

Размер шрифта определяется свойством fontSize стилевого объекта. Для верхнего поля размер шрифта равен 18, а для нижнего поля устанавливается размер шрифта, равный 15 (команды outStl.fontSize="18pt" и соответственно inStl.fontSize="15pt").

Чтобы установить режим отображения рамки нижнего поля пунктирной линией, используем команду inStl.borderStyle="dotted".

> **НА ЗАМЕТКУ**
>
> Среди прочих возможных значений свойства borderStyle есть такие: "none" (отсутствие границы), "solid" (сплошная линия), "dashed" (штрихованная линия), "double" (двойная линия), "groove" (рифленая граница), "ridge" (трехмерная ребристая рамка), "inset" (трехмерная вдавленная граница), "outset" (трехмерная выступающая граница) и некоторые другие.

Командами outFld.value="" и inFld.value="" выполняется очистка полей ввода. Вызвав метод focus() из объекта нижнего поля (команда inFld.focus()) добиваемся передачи фокуса нижнему полю. Наконец, для определения обработчика события, состоящего в отпускании клавиши на клавиатуре, свойству onkeyup объекта нижнего поля присваивается функция (метод), в теле которой выполняется единственная команда outFld.value=this.value. В соответствии с данной командой значение свойства value объекта, из которого вызывается метод (на котором произошло событие — это объект нижнего поля), присваивается полю value объекта верхнего поля. Поэтому каждый раз, когда пользователь при вводе текста в нижнее поле отпускает клавишу, считывается значение в нижнем поле и записывается в верхнее поле.

> **НА ЗАМЕТКУ**
>
> Свойство onkeyup используется для обработки события, связанного с отпусканием клавиши. Свойство onkeydown используется для обработки события, связанного с нажатием клавиши. Свойство onkeypress используют для обработки события, связанного с вводом символа.

Опции, переключатели и списки

В следующем примере мы рассмотрим способы использования таких элементов управления, как опции, переключатели и списки.

Опция представляет собой элемент управления, который может находиться в двух состояниях. Визуально состояние опции обычно определяется наличием или отсутствием флажка. В разных операционных системах и разных браузерах опция может отображаться по-разному, но обычно это белый квадратик с галочкой (опция установлена) или без галочки (опция не установлена). Рядом с таким квадратиком, как правило, размещается короткий поясняющий текст.

Переключатель концептуально очень близок к опции. Это элемент, который также может находиться в одном из двух состояний. Важное отличие от опции состоит в том, что переключатели объединяются в группы и в пределах группы может быть установлен один и только один переключатель. Визуально переключатель отображается в виде кружка, внутри которого может быть жирная точка (если переключатель установлен).

И опции, и переключатели в веб-документе создаются с помощью <input>-блока. Для опции значением атрибута type указывается "checkbox", а для переключателя значение атрибута type равно "radio". Чтобы было понятно, как переключатели распределяются по группам, у всех переключателей группы задают одинаковое значение для атрибута name.

Раскрывающийся список внешне напоминает поле. Но значение не вводится, как в поле, а выбирается из списка, который раскрывается при щелчке по пиктограмме в правой части раскрывающегося списка. Кроме раскрывающихся списков используют и такие, в которых содержимое для выбора отображается сразу в развернутом виде. При этом, если содержимое списка не помещается в развернутом виде в области отображения списка, используется полоса прокрутки. Такие списки будем называть списками выбора, чтобы отличать их от раскрывающихся списков. Списки обоих типов создаются с помощью блока <select>. Между открывающим дескриптором <select> и закрывающим дескриптором </select> размещаются <option>-блоки, формирующие содержимое списка. Если в дескрипторе <select> атрибут size указан со значением "1", то список будет раскрывающимся. Если значением атрибута указать текстовое представление для числа больше единицы, получим список выбора, в котором отображается сразу несколько пунктов. Количество одновременно отображаемых пунктов определяется значением атрибута size. Если в списке больше

элементов, чем отображается, то автоматически добавляется полоса прокрутки.

Далее мы исследуем функциональные возможности документа, содержащего опции, переключатели и списки. На рис. 8.10 показано, как документ выглядит при загрузке его в окно браузера.

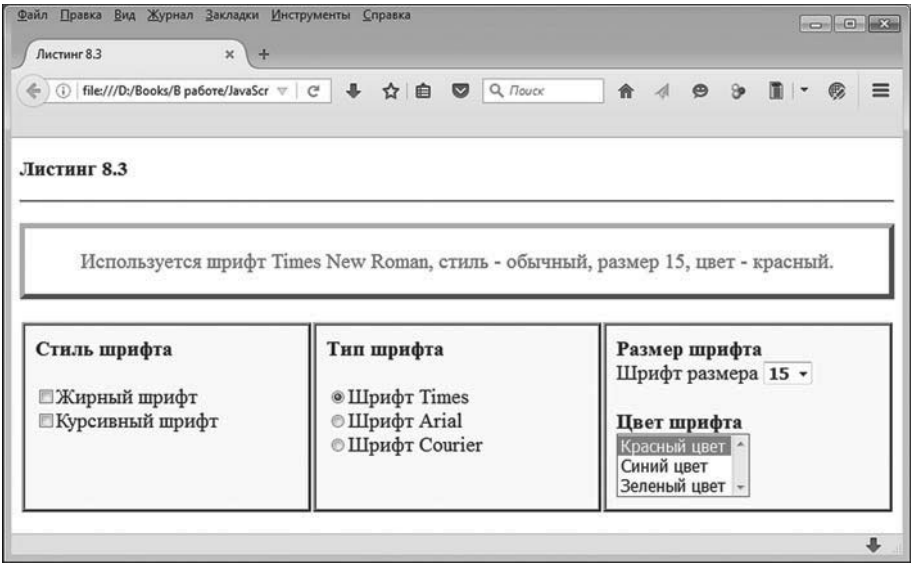

Рис. 8.10. Документ с опциями, переключателями и списками

Идея, реализованная в документе, исключительно проста. Во-первых, в документе имеется область, в которой отображается некоторый шаблонный текст. Под областью с текстом имеется три блока с элементами управления (блок с опциями, блок с переключателями и блок со списками), настройки которых позволяют в интерактивном режиме изменять основные характеристики текста (такие, как стиль, тип шрифта, его размер и цвет). Более того, сам шаблонный текст содержит информацию о том, каковы параметры используемого шрифта.

При загрузке документа флажки у опций применения курсивного стиля и жирного стиля не установлены. Соответственно, эти стили не применяются (и в таком случае используется нормальный, или обычный, стиль). В группе переключателей установлен первый переключатель, что соответствует использованию шрифта Times New Roman.

В раскрывающемся списке выбрано значение 15 для размера шрифта, а в списке выбора выделен первый пункт, соответствующий красному цвету. Поэтому шаблонный текст отображается красным цветом, шрифтом Times New Roman размера 15, обычным стилем (не жирный и не курсив). И информация обо всех перечисленных параметрах содержится непосредственно в шаблонном тексте (см. рис. 8.10).

Теперь допустим, что мы меняем настройки. Например, устанавливаем флажок опции **Жирный шрифт**, как показано на рис. 8.11.

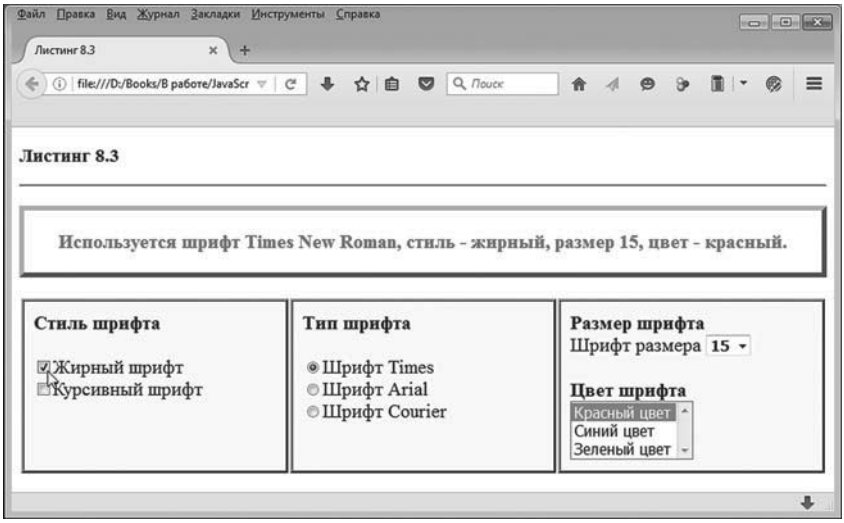

Рис. 8.11. В документе установлена опция применения жирного стиля

Результат будет в том, что, во-первых, к шаблонному тексту применится жирный стиль и, во-вторых, в самом тексте теперь указано, что использован жирный стиль. Аналогично все происходит и при изменении прочих настроек. На рис. 8.12 показан документ, в котором установлен второй переключатель в группе переключателей, что соответствует шрифту Arial.

В результате шаблонный текст отображается именно таким шрифтом, а название шрифта указано в шаблонном тексте. На рис. 8.13 в списке выделен пункт, соответствующий синему цвету, а в раскрывающемся списке производится выбор для размера шрифта.

После применения нового размера документ будет выглядеть так, как показано на рис. 8.14.

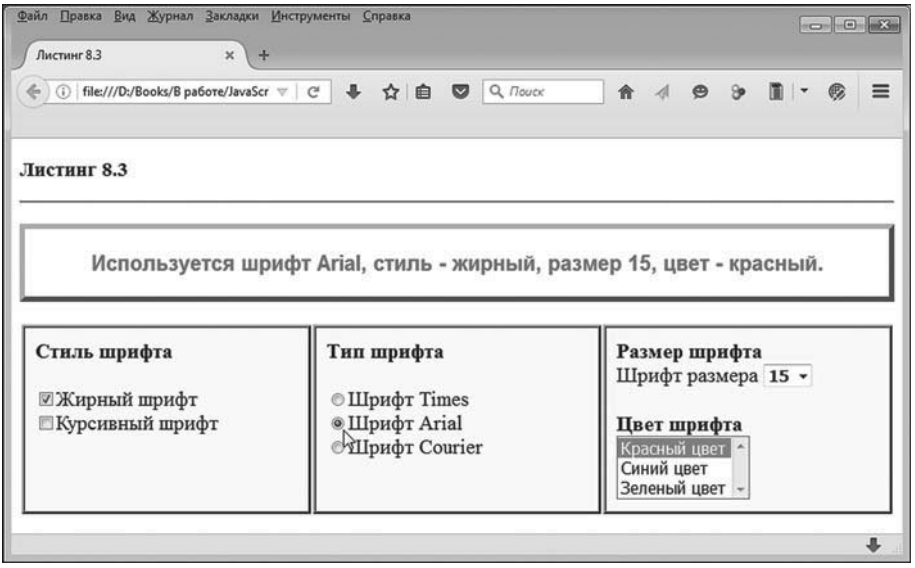

Рис. 8.12. В документе с помощью переключателя установлен тип шрифта Arial

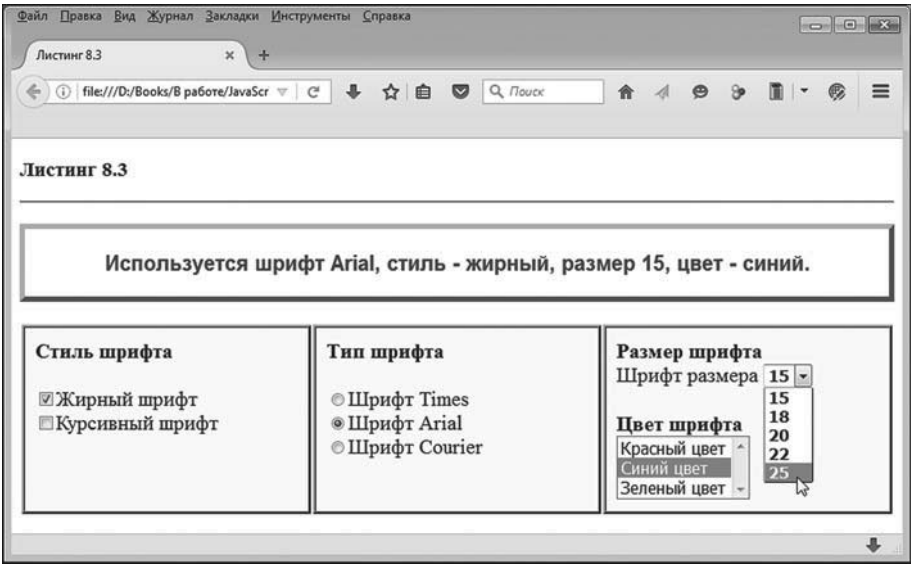

Рис. 8.13. В документе применен синий цвет для шрифта, а в раскрывающемся списке выполняется выбор размера для шрифта

На рис. 8.15 показано, как выглядит документ, если шаблонный текст отображается шрифтом Courier New зеленого цвета, с размером 22 и использованием жирного курсивного стиля.

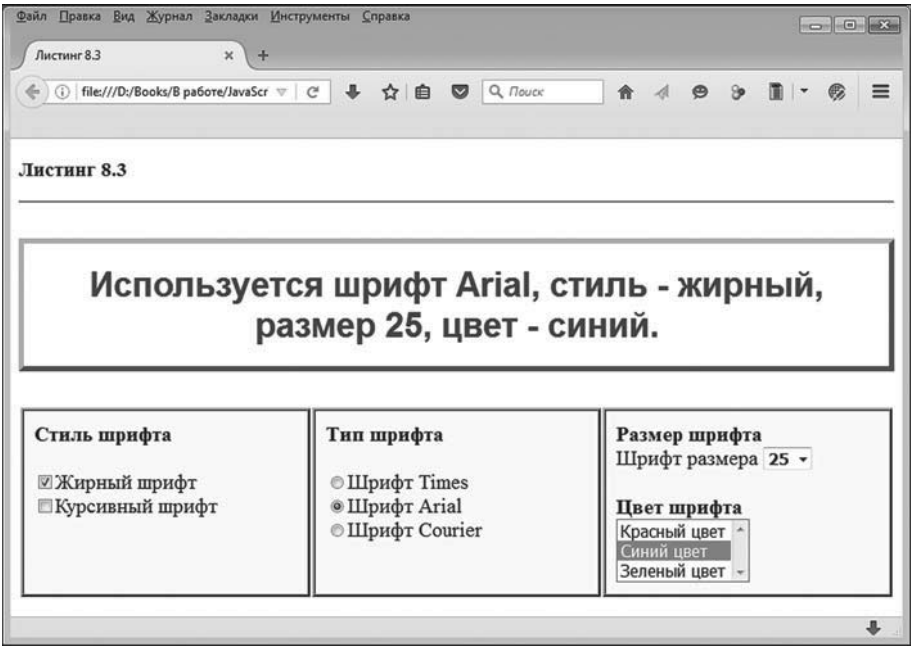

Рис. 8.14. Результат изменения размера шрифта в документе

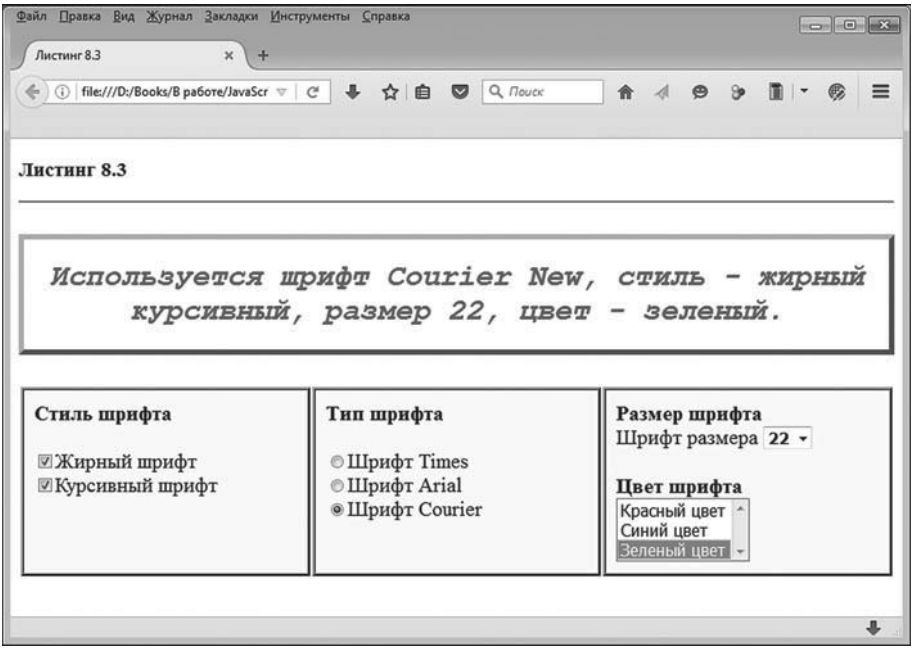

Рис. 8.15. В документе применен жирный курсивный стиль, использован шрифт Courier размера 22, зеленого цвета

На рис. 8.16 показано, как выглядит документ при использовании курсивного стиля и шрифта Times New Roman синего цвета размера 20.

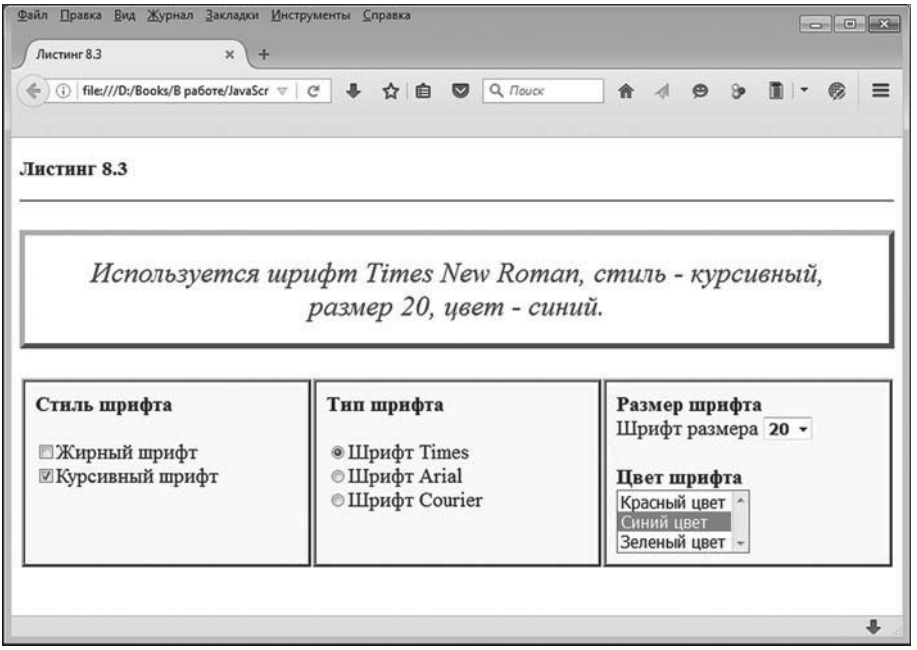

Рис. 8.16. В документе использован курсивный стиль при шрифте Times New Roman размера 20, синего цвета

Вообще разных комбинаций в плане настроек параметров шрифта существует достаточно много (если точнее, то всего 80). Поэтому нет смысла перебирать все возможные варианты. Вместо этого лучше проанализируем программный код документа. Обратимся к листингу 8.3.

Листинг 8.3. Опции, переключатели и списки выбора

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 8.3</title>
<!-- Начало сценария -->
<script type="text/javascript">
  // Переменные для записи ссылок на объекты элементов:
```

```
var fitalic,fbold,family,fsize,fcolor,stext,tfamily,tstyle,tsize,tcolor
// Функция для определения ссылок на объекты:
function setRefs(){
  // Ссылка на текстовый блок с шаблонным текстом:
  stext=document.getElementById("sampleText")
  // Ссылка на опцию применения курсивного стиля:
  fitalic=document.getElementById("italicFont")
  // Ссылка на опцию применения жирного стиля:
  fbold=document.getElementById("boldFont")
  // Ссылка на группу переключателей, используемую
  // для определения типа шрифта:
  family=document.getElementsByName("fontFamily")
  // Ссылка на раскрывающийся список, предназначенный
  // для выбора размера шрифта:
  fsize=document.getElementById("fontSize")
  // Ссылка на список выбора, используемый
  // для определения цвета шрифта:
  fcolor=document.getElementById("fontColor")
  // Ссылка на блок для вставки названия шрифта:
  tfamily=document.getElementById("A")
  // Ссылка на блок для вставки стиля шрифта:
  tstyle=document.getElementById("B")
  // Ссылка на блок для вставки размера шрифта:
  tsize=document.getElementById("C")
  // Ссылка на блок для вставки цвета шрифта:
  tcolor=document.getElementById("D")
}
// Функция для определения обработчиков событий
// для элементов управления в документе:
function setHandlers(){
  // Обработчик события, связанного с изменением
  // состояния опции применения курсивного стиля:
  fitalic.addEventListener("change",getStyle)
  // Обработчик события, связанного с изменением
```

```
  // состояния опции применения жирного стиля:
  fbold.onchange=getStyle
  // Обработчики для переключателей:
  for(var k=0;k<family.length;k++){
    family[k].addEventListener("change",getFamily)
  }
  // Обработчик для раскрывающегося списка:
  fsize.addEventListener("change",getSize)
  // Обработчик для списка выбора:
  fcolor.onchange=function(){
    getColor()
  }
}
// Функция для определения стиля шрифта:
function getStyle(){
  // Текст с описанием стиля шрифта:
  var txt=""
  // Если установлен флажок опции применения
  // жирного стиля:
  if(fbold.checked){
    // Применение жирного стиля:
    stext.style.fontWeight="bold"
    // Уточнение текста с описанием стиля:
    txt+=" жирный"
  }
  // Если флажок опции применения жирного стиля
  // не установлен:
  else{
    // Применение обычного стиля:
    stext.style.fontWeight="normal"
  }
  // Если установлен флажок опции применения
  // курсивного стиля:
  if(fitalic.checked){
```

```
      // Применение курсивного стиля:
      stext.style.fontStyle="italic"
      // Уточнение текста с описанием стиля:
      txt+=" курсивный"
    }
    // Если флажок опции применения курсивного стиля
    // не установлен:
    else{
      // Применение обычного стиля:
      stext.style.fontStyle="normal"
    }
    // Проверка текстового значения:
    if(txt==""){
      txt=" обычный"
    }
    // Добавление текста в текстовый блок:
    tstyle.innerHTML=txt
  }
  // Функция для определения типа шрифта:
  function getFamily(){
    // Поиск выбранного пункта списка:
    for(var k=0;k<family.length;k++){
      // Если элемент выбран:
      if(family[k].checked){
        // Применение стиля:
        stext.style.fontFamily=family[k].value
        // Добавление текста в текстовый блок:
        tfamily.innerHTML=family[k].value
        // Завершение выполнения функции:
        return
      }
    }
  }
  // Функция для определения размера шрифта:
```

```
function getSize(){
  // Применение стиля:
  stext.style.fontSize=fsize.value+"pt"
  // Добавление текста в текстовый блок:
  tsize.innerHTML=fsize.value
}
// Функция для определения цвета шрифта:
function getColor(){
  // Применение стиля:
  stext.style.color=fcolor.value
  // Добавление текста в текстовый блок:
  tcolor.innerHTML=convertColor(fcolor.value)
}
// Функция для преобразования английского названия
// цвета в русское название:
function convertColor(clr){
  if(clr=="red") return "красный"
  if(clr=="blue") return "синий"
  if(clr=="green") return "зеленый"
}
// Функция для выполнения при загрузке документа:
function set(){
  // Определение ссылок на объекты элементов:
  setRefs()
  // Флажок опции применения жирного стиля
  // не установлен:
  fitalic.checked=false
  // Флажок опции применения курсивного стиля
  // не установлен:
  fbold.checked=false
  // Начальный выбор в группе переключателей
  // для определения типа шрифта:
  family[0].checked=true
  // Начальный выбор в списке определения размера:
```

```
    fsize[0].selected=true
    // Начальный выбор в списке определения цвета:
    fcolor[0].selected=true
    // Определение обработчиков для элементов:
    setHandlers()
    // Вызов функций для формирования шаблонного текста
    // и применения стилевых параметров в соответствии
    // с начальными настройками:
    getStyle()  // Стиль шрифта
    getFamily() // Тип шрифта
    getSize()   // Размер шрифта
    getColor()  // Цвет шрифта
  }
</script>
<!-- Завершение сценария -->
<!-- Определение стилей -->
<style type="text/css">
  /* Стиль для столбика таблицы */
  td{
    width: 30%;
    background-color: #fcfcfc;
    vertical-align: top;
    font-size: 15pt;
    border-style: ridge;
    padding: 10px;
  }
  /* Стиль для блока с текстом */
  #sampleText{
    border: outset;
    border-width: 5px;
    border-color: gray;
    text-align: center;
    padding: 20px;
  }
```

```
</style>
<!-- Завершение описания стилей -->
</head>
<body onload="set()">
  <h3>Листинг 8.3</h3><hr>
  <!-- Блок с шаблонным текстом -->
  <p id="sampleText">
  Используется шрифт <span id="A"></span>, стиль -<span id="B"></span>, размер <span
  id="C"></span>, цвет - <span id="D"></span>.
  </p>
  <!-- Таблица с элементами управления -->
  <table style="width:100%;">
    <!-- Первая и единственная строка таблицы -->
    <tr>
      <!-- Первый столбец -->
      <td>
        <b>Стиль шрифта</b><br><br>
        <!-- Опция применения жирного стиля -->
        <input type="checkbox" id="boldFont">Жирный шрифт<br>
        <!-- Опция применения курсивного стиля -->
        <input type="checkbox" id="italicFont">Курсивный шрифт
      </td>
      <!-- Второй столбец -->
      <td>
        <b>Тип шрифта</b><br><br>
        <!-- Первый переключатель -->
        <input type="radio" name="fontFamily" value="Times New Roman">Шрифт Times<br>
        <!-- Второй переключатель -->
        <input type="radio" name="fontFamily" value="Arial">Шрифт Arial<br>
        <!-- Третий переключатель -->
        <input type="radio" name="fontFamily" value="Courier New">Шрифт Courier
      </td>
      <!-- Третий столбец -->
      <td>
        <b>Размер шрифта</b><br>
```

```
        Шрифт размера
        <!-- Раскрывающийся список -->
        <select size="1" id="fontSize" style="font-size:11pt;font-weight:900;">
          <option value="15">15</option>
          <option value="18">18</option>
          <option value="20">20</option>
          <option value="22">22</option>
          <option value="25">25</option>
        </select><br><br>
        <b>Цвет шрифта</b><br>
        <!-- Список выбора -->
        <select size="3" id="fontColor" style="font-size:12pt;">
          <option value="red">Красный цвет</option>
          <option value="blue">Синий цвет</option>
          <option value="green">Зеленый цвет</option>
        </select>
      </td>
    </tr>
  </table>
</body>
</html>
```

Код содержит несколько функций в разделе сценария, но мы начнем анализ с той части кода, в которой, собственно, создаются основные элементы документа. По большому счету <body>-блок состоит из двух конструкций: это <p>-блок с шаблонным текстом и таблица, состоящая из одной строки и трех столбцов. Каждый столбец содержит определенный набор элементов управления.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Таблица создается с помощью парного дескриптора <table>. Строки таблицы в <table>-блоке выделяются парами дескрипторов <tr> и </tr>. Между этими дескрипторами размещаются парные <td>-блоки, определяющие столбцы таблицы (в данной строке).

Текстовый <p>-блок описан со значением "sampleText" для атрибута id. По значению этого атрибута мы получим доступ к элементу из сце-

нария. В сценарии в отношении данного элемента в зависимости от настроек элементов управления будут применяться те или иные стилевые настройки. Но в данном случае можно вести речь о настройках блока как такового. Проблема же в том, что при изменении настроек элементов управления должен меняться не только способ отображения шаблонного текста, но и сам текст. Мы используем подход, при котором шаблонный текст, помимо собственно текста (которые не меняется), содержит специальные -блоки, предназначенные для вставки в них текста при выполнении сценария. Всего используется четыре таких блока в тексте. В первый блок вставляется название для стиля, во второй блок вставляется название шрифта, в третий блок вставляется размер шрифта, а в четвертый блок вставляется название цвета. Каждый -блок содержит атрибут id. Значения атрибута для блоков — это символы от "A" до "D". Доступ к бокам из сценария реализуется на основе значения атрибута id соответствующего блока.

Как отмечалось, кроме блока с шаблонным текстом документ содержит еще таблицу. Таблица состоит из одной строки, в которой три столбца (инструкция style="width:100%;" означает применение стиля, при котором таблица по ширине автоматически масштабируется до размеров рабочей области окна). В первом столбце размещен текст и два <input>-блока. В обоих блоках значение атрибута type равно "checkbox", поэтому в результате получаем два опционных элемента. Текст, отображаемый возле опции, указан непосредственно после <input>-блоков. У опции, «отвечающей» за применение жирного стиля, атрибуту id задано значение "boldFont". Атрибут id опции, используемой для применения курсивного стиля, имеет значение "italicFont".

Во втором столбце размещена группа из трех переключателей. Переключатели реализуются в виде <input>-блоков со значением "radio" для атрибута type. Чтобы относились они к одной группе, каждый блок содержит атрибут name со значением "fontFamily". Доступ к группе переключателей в сценарии осуществляется на основании значения данного атрибута. Кроме этого, для каждого <input>-блока указано оригинальное значение для атрибута value, причем значение атрибута не просто оригинальное, а совпадает с названиями шрифтов, которые планируется использовать в документе. В частности, мы используем значения "Times New Roman", "Arial" и "Courier New". Как и в случае с опциями, текст, отображаемый возле переключателей, указывается непосредственно после <input>-блоков.

В третьем столбце размещается текст и два списка. Список для выбора размера шрифта создается с помощью <select>-блока, в котором атрибут size имеет значение "1". Это раскрывающийся список. Атрибут id этого элемента имеет значение "fontSize". Данный атрибут используется в сценарии для получения доступа к элементу. Инструкция style="font-size:11pt;font-weight:900;" в описании дескриптора <select> определяет стиль для раскрывающегося списка, при котором пункты раскрывающегося списка отображаются жирным шрифтом размера 11. Содержимое раскрывающегося списка определяется <option>-блоками. Содержимое (названия пунктов), которое будет отображаться в раскрывающемся списке, размещается между открывающим дескриптором <option> и закрывающим дескриптором </option>. Но кроме этого каждый <option>-блок содержит атрибут value. Текстовое значение атрибута совпадает (дублирует) отображаемое в списке значение. В сценарии мы будем оперировать именно со значениями атрибута value пунктов списка, а не с теми значениями, что отображаются в списке (хотя формально в данном конкретном случае они и совпадают).

Еще один список используется для определения цвета шрифта. Список также реализуется в виде <select>-блока. Инструкция size="3" означает, что в списке одновременно отображается 3 позиции. Поскольку весь список состоит из трех позиций, то список фактически отображается целиком. Атрибут id имеет значение "fontColor", а инструкция style="font-size:12pt;" определяет стиль для отображения пунктов списка с помощью шрифта размера 12. Внутренние <option>-блоки содержат атрибут value, значения которого "red", "blue" и "green" для разных пунктов представляют собой «легитимные» текстовые названия цветов. Эти значения используются в сценарии.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Способ отображения столбцов таблицы и некоторые характеристики стиля для текстового блока задаются в <style>-блоке. Фрагмент кода, выделенный в блок с заглавной инструкцией td, определяет стиль для элементов, определяемых на основе <td>-блоков, — то есть это стиль, применяемый при отображении столбцов таблиц. В соответствии с использованными инструкциями ширина столбца составляет 30% от ширины таблицы (хотя в данном случае наличие такой настройки не принципиально), вдоль вертикали выравнивание выполняется по верхнему краю (инструкция vertical-align:top), используется шрифт размера 15 (инструкция font-size:15pt), тип рамки вокруг ячейки — трехмерная ребристая (инструкция border-style:ridge), внут-

ренние отступы от границ ячейки составляют не менее 10 пикселей (инструкция padding:10px), а цвет фона определяется инструкцией background-color:#fcfcfc. В данном случае мы учли, что белому цвету соответствует код #ffffff, а код «посерее» можно сформировать, синхронно уменьшив три шестнадцатеричных числа, из которых состоит код цвета. Проще говоря, вместо трех чисел ff берем три числа fc. Получившийся код #fcfcfc соответствует едва заметному серому цвету.

Стиль для блока с текстом озаглавлен инструкцией #sampleText. Она означает, что соответствующий стиль применяется к элементу со значением "sampleText" атрибута id (в нашем случае это блок с шаблонным текстом). В соответствии со стилем текстовый блок отображается с трехмерной выступающей рамкой (инструкция border:outset) толщиной в 5 пикселей (инструкция border-width:5px), рамка серого цвета (инструкция border-color:gray), текст в блоке вдоль горизонтали выравнивается по центру (инструкция text-align:center), а внутренние отступы между текстом и рамкой составляют не менее 20 пикселей (инструкция padding:20px).

Также стоит отметить, что комментарии внутри блока описания стиля в соответствии со стандартами кодовых таблиц стилей размещаются между символами /* и */.

Теперь настал черед сценария. В первую очередь в сценарии объявляется несколько глобальных переменных, предназначенных для записи в них ссылок на объекты элементов документа. Вычисление и запоминание ссылок выполняются в функции setRefs(). Во всех случаях, за исключением переменной family, для получения ссылки используется метод getElementById(), аргументом которому передается значение атрибута id соответствующего элемента. Что касается переменной family, то ее значение вычисляется с помощью команды family=document.getElementsByName("fontFamily"). Здесь методу getElementsByName() аргументом передается значение "fontFamily" атрибута name переключателей. Результатом метод возвращает коллекцию элементов, у которых атрибут name равен "fontFamily" — то есть в переменную family будет записана ссылка на коллекцию переключателей.

Функция setHandlers() предназначена для определения обработчиков событий элементов управления, использованных в документе. Тело функции состоит из команд, которыми для основных функциональных элементов определяются обработчики. Мы (исключительно ради разнообразия) используем разные способы определения обработчиков. Например, командой fitalic.addEventListener("change",getStyle) для объекта опции, связанной с применением курсивного стиля, регис-

трируется обработчик события, связанного с изменением состояния опции. В представленной команде из объекта fitalic (ссылка на опцию применения курсивного стиля), для которого определяется обработчик, вызывается метод addEventListener(). Первым аргументом методу передается название события, для которого регистрируется обработчик. Событие "change" связано с изменением состояния элемента. Второй аргумент метода addEventListener() является названием функции, вызываемой при обработке события. Инструкция fitalic.addEventListener ("change",getStyle), таким образом, означает, что при изменении состояния опции вызывается функция getStyle() (описывается далее).

> **НА ЗАМЕТКУ**
>
> Событие change соответствует свойству onchange объекта, для которого регистрируется событие. Очевидная корреляция между названиями событий и названиями свойств, предназначенных для регистрации обработчика события, прослеживается и для прочих событий.

Эта же функция getStyle() регистрируется для обработки такого же события change для опции fbold, связанной с применением жирного стиля. Но реализуется это несколько иначе, посредством инструкции fbold. onchange=getStyle: мы просто присвоили свойству onchange имя функции, которую следует вызывать при обработке события change.

При регистрации обработчиков события change для переключателей также используется метод addEventListener(). Но специфика ситуации в том, что переключатели «спрятаны» в коллекции family, которую мы обрабатываем как массив с помощью оператора цикла. Количество элементов в коллекции получаем с помощью инструкции family.length. При заданном значении индекса k ссылка на отдельный переключатель вычисляется инструкцией family[k]. Для каждого элемента для обработки события change регистрируется функция getFamily() (описывается далее).

Командой fsize.addEventListener("change",getSize) функция getSize() (описывается далее) регистрируется для обработки события change, генерируемого на раскрывающемся списке выбора размера шрифта. Наконец, при регистрации обработчика события change для списка выбора цвета значением свойству onchange объекта fcolor присваивается анонимная функция, в теле которой вызывается функция getColor() (также будет анализироваться далее).

Для понимания принципов функционирования программного кода необходимо проанализировать функции, используемые при обработке событий, связанных с изменением состояния управляющих элементов в документе. Этим и займемся.

Функция getStyle() вызывается при изменении состояния любой из опций. В теле функции объявляется текстовая переменная txt с начальным значением в виде пустой строки. Далее в действие вступают условные операторы. В первом условном операторе проверяется условие fbold.checked. Свойство checked объекта fbold имеет значение true, если у опции применения жирного стиля установлен флажок. В данном случае командой stext.style.fontWeight="bold" мы применяем жирный стиль для шрифта, которым отображается шаблонный текст в текстовом блоке. Также командой txt+=" жирный" к текущему значению текстовой переменной txt добавляется текст " жирный". Если флажок опции не установлен (и проверяемое условие ложно), то командой stext.style.fontWeight="normal" устанавливается нормальный (не жирный) стиль для шрифта в текстовом блоке. Значение текстовой переменной txt при этом не меняется.

В следующем условном операторе проверяемым условием указано выражение fitalic.checked. Условие истинно, если установлен флажок опции применения курсивного цвета. В таком случае командой stext.style.fontStyle="italic" для шрифта в текстовом блоке применяется курсивный стиль, а командой txt+=" курсивный" дописывается текст " курсивный" к текущему значению переменной txt. Если флажок опции применения курсивного стиля не установлен, выполняется всего одна команда stext.style.fontStyle="normal", которой к шрифту применяется обычный (не курсивный) стиль.

> **НА ЗАМЕТКУ**
>
> За жирный и курсивный стили отвечают разные свойства объекта стиля. Для определения курсивного стиля используется свойство fontStyle, а толщина шрифта (фактически определяющая жирный и не жирный шрифт) задается через свойство fontWeight. Обычный стиль шрифта (не жирный и не курсивный) подразумевает, что значения обеих указанных свойств равны "normal".

После выполнения двух операторов цикла возможны четыре значения переменной txt: " жирный", " курсивный", " жирный курсивный" и "". Последний

вариант реализуется, если должен применяться обычный шрифт. Поэтому с помощью условного оператора, в котором проверяется условие txt=="", мы командой txt=" обычный" производим изменение значения переменной txt. Таким образом, текст с описанием стиля шрифта сформирован, и командой tstyle.innerHTML=txt выполняется вставка соответствующего текстового значения в первый -блок текстового блока.

Для определения типа (названия) шрифта используется функция getFamily(). В теле этой функции с помощью оператора цикла перебираются элементы, формирующие коллекцию family (коллекция переключателей). В теле оператора цикла размещен условный оператор с проверяемым условием family[k].checked. Условие истинно, если элемент family[k] (переключатель) установлен. В таком случае командой stext.style.fontFamily=family[k].value выполняется применение выбранного шрифта к текстовому блоку. Здесь уместно напомнить, что значения атрибута value для переключателей указаны так, что они совпадают с названиями шрифтов. Значением свойства value объекта family[k] является значение одноименного атрибута.

После применения выбранного шрифта его название добавляется во второй -блок командой tfamily.innerHTML=family[k].value. А поскольку дальнейший перебор переключателей теряет смысл (может быть установлен только один переключатель), то с помощью инструкции return завершается выполнение функции.

НА ЗАМЕТКУ

Общая схема такая: в операторе цикла последовательно перебираются переключатели. Как только найден установленный переключатель, считывается значение его атрибута value (название шрифта), и это значение присваивается свойству fontFamily объекта стиля для текстового блока. Затем это же значение записывается во второй -блок шаблонного текста. На этом выполнение функции завершается.

Функция getSize() нужна для определения размера шрифта. В теле функции имеется всего две команды. Сначала командой stext.style.fontSize=fsize.value+"pt" к текстовому блоку применяется соответствующий стиль, а затем размер шрифта вносится в третий -блок командой tsize.innerHTML=fsize.value. В данном случае мы учли, что значе-

нием свойства value объекта fsize (раскрывающийся список) является значение атрибута value того <option>-блока, который выбран в списке на данный момент. Напомним, что это текстовое представление числа, определяющего размер шрифта. Поскольку значением свойству fontSize стилевого объекта присваивается текст, состоящий из размера шрифта и суффикса "pt", то нужная комбинация получается вычислением выражения fsize.value+"pt", представляющим собой объединение двух текстовых строк.

Функция для определения цвета шрифта называется getColor() и также содержит всего две команды. Сначала командой stext.style.color=fcolor.value к шрифту в текстовом блоке применяется выбранный цвет. Как и в случае с раскрывающимся списком, значением свойства value списка fcolor является значение атрибута value, выбранного в списке <option>-блока. Данные <option>-блоки были описаны так, что их атрибуты value содержали корректные названия для цветов, которые можно присвоить, например, свойству color объекта стиля. Что мы, собственно, и сделали. Но при вставке текста в четвертый -блок нам понадобится русское название шрифта. Для этой цели мы используем специально описанную функцию convertColor(), которой передается текст с английским названием шрифта, а функция возвращает русское название шрифта. В итоге команда вставки названия шрифта в шаблонный текст выглядит как tcolor.innerHTML=convertColor(fcolor.value).

НА ЗАМЕТКУ

Наша функция convertColor() «знает» всего три цвета: "red" (красный), "blue" (синий) и "green" (зеленый).

Все описанные утилиты собраны в функции set(), которая выполняется при загрузке документа благодаря инструкции onload="set()" в описании дескриптора <body>. При вызове функции set() сначала вызывается функция setRefs(). Благодаря этому получают значения переменные, объявленные в сценарии для выполнения ссылок на элементы управления. Затем командами fitalic.checked=false и fbold.checked=false устанавливается режим отсутствия флажков у опций применения курсивного и жирного стилей. Командой family[0].checked=true устанавливается первый переключатель из группы переключателей. Начальный выбор в списке определения размера и списке определения цвета выполняется командами fsize[0].selected=true и fcolor[0].selected=true со-

ответственно. Вызвав функцию setHandlers(), выполняем регистрацию обработчиков для элементов управления. На данном этапе настройки всех управляющих элементов выполнены, а обработчики зарегистрированы. Далее последовательно вызываются такие функции: getStyle() (определяется стиль шрифта), getFamily() (определяется тип шрифта), getSize() (определяется размер шрифта) и getColor() (определяется цвет шрифта).

Работа с изображениями

— Создайте какое-нибудь полотно!
— Зачем ему полотно, что вы его сбиваете!

из к/ф «Покровские ворота»

Хороший веб-документ обычно содержит изображения. Следовательно, при написании сценария для работы с таким документом, скорее всего, придется иметь дело с изображениями. Далее мы рассмотрим некоторые наиболее характерные ситуации.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Вставка в документ изображения выполняется с помощью одинарного дескриптора <img>. Чтобы определить файл, из которого загружается изображение в документ, свойству атрибута src дескриптора значением присваивается текстовая строка, определяющая путь к графическому файлу и его название. Адрес может быть абсолютным или относительным. Абсолютный адрес начинается с адреса сервера и инструкции http://. Относительный адрес может быть достаточно замысловатый. Если просто указать имя графического файла, то поиск файла будет выполняться в той же папке, в которой находится файл с документом. Косая черта в описании пути разделяет вложенные папки. Начальная косая черта означает путь, начинающийся с корня сайта (но это работает только на сервере). Две точки и косая черта ../ в пути означают переход на один уровень вверх.

Атрибуты width и height определяют, соответственно, ширину и высоту рисунка. Атрибут alt позволяет задать текст, который отображается в области рисунка, если загрузка рисунка оказалась невозможной. Также стоит заметить, что для изображения можно описать атрибут title. Текстовое значение атрибута определяет всплывающую подсказку, появляющуюся при наведении курсора на изображение.

Просмотр изображений

Наиболее простой случай обработки изображений с помощью сценария связан с необходимостью изменять изображение в документе при возникновении некоторых событий или выполнении определенных действий.

Механизм реализации данной процедуры базируется на том, чтобы с помощью сценария изменить значение атрибута src в соответствующем <img>-блоке. Именно такой подход используется в рассматриваемом далее примере. Мы создадим документ с раскрывающимся списком. Пункты списка — названия животных. При выборе того или иного животного из раскрывающегося списка рядом со списком отображается картинка с изображением животного, а внизу под списком — краткое описание.

На рис. 8.17 показано, как выглядит загруженный в окно браузера документ.

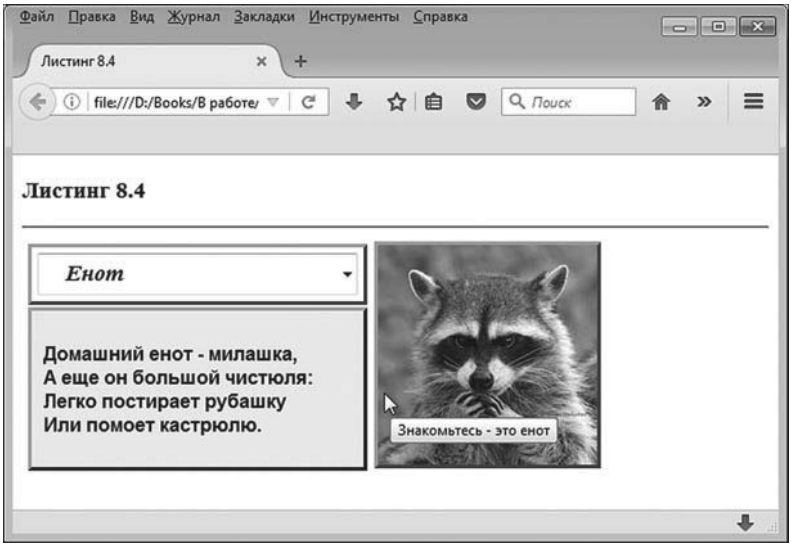

Рис. 8.17. Документ с раскрывающимся списком и изменяемым изображением

По умолчанию в раскрывающемся списке выбран пункт **Енот**. Справа отображается соответствующая картинка. При наведении на нее курсора мыши всплывает подсказка. Внизу под списком приводится небольшой текстовый фрагмент, имеющий отдаленное смысловое отношение к еноту.

Если щелкнуть по раскрывающемуся списку, увидим, какие в принципе есть возможности для выбора. Ситуация проиллюстрирована на рис. 8.18.

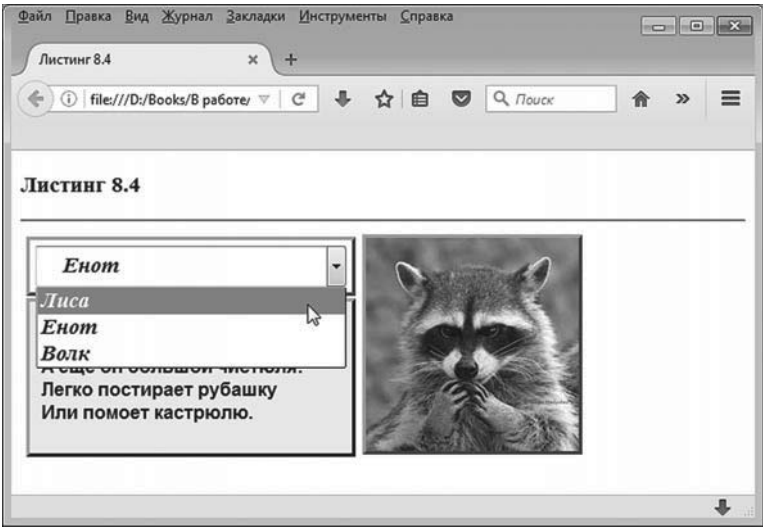

Рис. 8.18. Выбор нового пункта в раскрывающемся списке

При выборе в раскрывающемся списке пункта **Лиса** получаем результат, как на рис. 8.19.

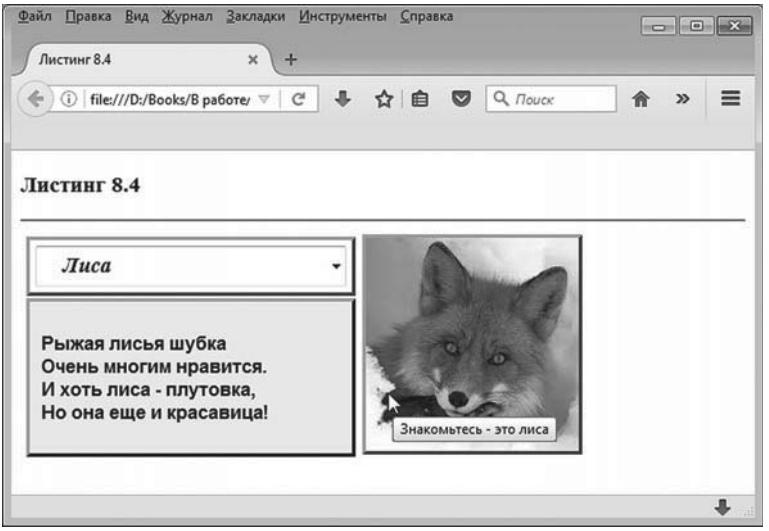

Рис. 8.19. Результат выбора нового пункта в раскрывающемся списке

Видим, что поменялось изображение и текст под раскрывающимся списком. Картинка и текст снова поменяются, если в раскрывающемся списке выбрать пункт **Волк**. Как будет выглядеть документ в этом случае, показано на рис. 8.20.

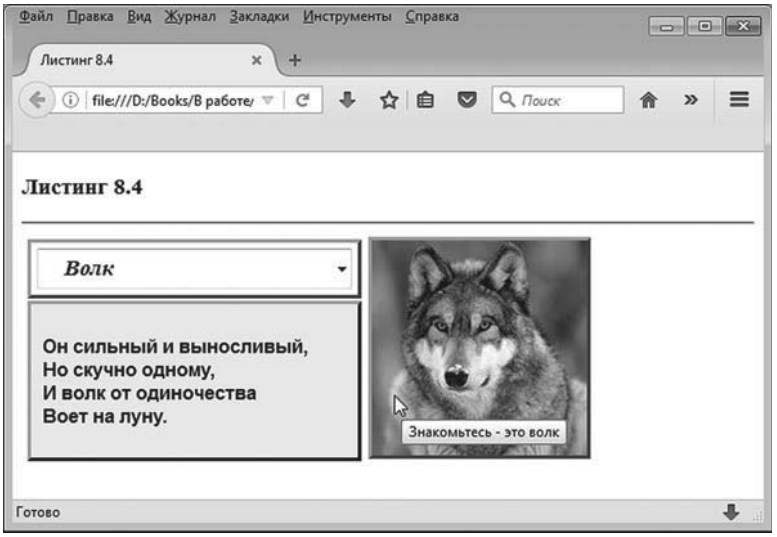

Рис. 8.20. В раскрывающемся списке выбран очередной пункт

Далее мы проанализируем программный код документа со сценарием, обрабатывающим изменение пунктов в раскрывающемся списке. Еще одна особенность документа связана с тем, что сам раскрывающийся список формируется с помощью сценария, а не описывается статически в теле документа, как это было в предыдущем примере. Итак, обратимся к коду в листинге 8.4.

Листинг 8.4. Просмотр изображений

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 8.4</title>
<!-- Начало сценария -->
<script type="text/javascript">
  // Названия файлов с изображениями:
  var fileNames=["fox.jpg","raccoon.jpg","wolf.jpg"]
```

```
// Названия зверей для раскрывающегося списка:
var animals=["Лиса","Енот","Волк"]
// Описание для лисы:
var fox="Рыжая лисья шубка<br>Очень многим нравится.<br>И хоть лиса - плутовка,<br>Но она еще и красавица!"
// Описание для енота:
var raccoon="Домашний енот - милашка,<br>А еще он большой чистюля:<br>Легко постирает рубашку<br>Или помоет кастрюлю."
// Описание для волка:
var wolf="Он сильный и выносливый,<br>Но скучно одному,<br>И волк от одиночества<br>Воет на луну."
// Массив с текстовыми описаниями зверей:
var text=[fox,raccoon,wolf]
// Путь к файлам изображений:
var path="../images/"
// Переменные для записи ссылок на объекты элементов:
var lst,mytext,pict
// Обработка события, связанного с загрузкой документа:
window.onload=function(){
  // Ссылка на объект раскрывающегося списка:
  lst=document.getElementById("mylist")
  // Ссылка на объект изображения:
  pict=document.getElementById("mypict")
  // Ссылка на объект текстового блока:
  mytext=document.getElementsByTagName("p")[0]
  // Формирование раскрывающегося списка:
  for(var k=0;k<animals.length;k++){
    // Добавление нового пункта в список:
    lst.options[k]=new Option(animals[k],"Знакомьтесь - это "+animals[k].toLowerCase())
  }
  // Обработчик для события, связанного с выбором
  // пункта в раскрывающемся списке:
  lst.onchange=function(){
    // Ссылка на файл с изображением:
```

```
      pict.src=path+fileNames[lst.selectedIndex]
      // Всплывающая подсказка для изображения:
      pict.title=lst.value
      // Текст в текстовом блоке:
      mytext.innerHTML=text[lst.selectedIndex]
    }
    // Индекс выбранного пункта в списке:
    var i=1
    // Выбирается пункт в списке:
    lst.options[i].selected=true
    // Задается изображение, соответствующее выбранному
    // пункту в списке:
    pict.src=path+fileNames[i]
    // Всплывающая подсказка для изображения:
    pict.title=lst.value
    // Текст в текстовом блоке:
    mytext.innerHTML=text[i]
  }
</script>
<!-- Описание стиля для списка -->
<style type="text/css">
  select{
    font-family: Times New Roman;
    font-size: 14pt;
    font-style: italic;
    font-weight: bold;
    height: 100%;
    width: 100%;
    padding:1px 20px;
  }
</style>
<!-- Завершение описания стиля -->
</head>
<body>
```

```
<h3>Листинг 8.4</h3><hr>
<!-- Внешняя таблица -->
<table>
  <!-- Первая и единственная строка таблицы -->
  <tr>
    <!-- Первый столбец внешней таблицы -->
    <td style="width:280px;vertical-align:top;border-style:none;">
      <!-- Внутренняя таблица -->
      <table style="width:100%;">
        <!-- Первая строка внутренней таблицы -->
        <tr>
          <!-- Первый и единственный столбец -->
          <td style="width:100%;border-style:outset;height:35px;padding:5px;">
            <!-- Раскрывающийся список -->
            <select size="1" id="mylist"></select>
          </td>
        </tr>
        <!-- Вторая строка внутренней таблицы -->
        <tr>
        <!-- Первый и единственный столбец -->
          <td style="border-style:outset;height:110px;background-color:#f0f0f0;padding:10px;">
            <!-- Текстовый блок -->
            <p style="font-size:12pt;font-family:Arial;font-weight:bold;"></p>
          </td>
        </tr>
      </table>
    </td>
    <!-- Второй столбец внешней таблицы -->
    <td style="width:200px;border-style:none;">
      <img id="mypict" style="width:185px;height:185px;border-style:outset;">
    </td>
  </tr>
</table>
</body>
</html>
```

Тело документа состоит из таблицы (будем называть ее внешней). У таблицы одна строка и два столбца. Во втором столбце размещается изображение. В первом столбце содержится еще одна таблица (будем называть ее внутренней). У внутренней таблицы две строки и один столбец. В первой строке внутренней таблицы размещается раскрывающийся список, а во втором столбце внутренней таблицы находится текстовый блок, в который выводится текст с пояснениями к изображению.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

В описании первого столбца внешней таблицы инструкцией style="width:280px;vertical-align:top;border-style:none;" устанавливается ширина столбца в 280 пикселей, вдоль вертикали выравнивание выполняется по верхнему краю, и рамка вокруг столбца не отображается.

Данный столбец содержит таблицу (внутренняя таблица). Инструкция style="width:100%;" устанавливает ширину внутренней таблицы по ширине столбца.

В свою очередь первый и единственный столбец внутренней таблицы содержит инструкцию style="width:100%;border-style:outset;height:35px;padding:5px;", которой ширина столбца задается по ширине таблицы, применяется трехмерная выступающая рамка, высота столбца (в первой строке) равна 35 пикселей, а внутренний отступ между границами ячейки и содержимым равен 5 пикселям. Стиль столбца во второй строке внутренней таблицы определяется инструкцией style="border-style:outset;height:110px;background-color:#f0f0f0;padding:10px;". Здесь мы используем трехмерную выступающую рамку, устанавливаем высоту столбца в 110 пикселей, для фона ячейки задается светло-серый цвет с кодом #f0f0f0, а внутренний отступ между границами ячейки и содержимым равен 10 пикселям.

Второй столбец внешней таблицы описан инструкцией style="width:200px;border-style:none;", определяющей стиль отображения столбца. Здесь задано всего два параметра — ширина столбца в 200 пикселей, и рамка вокруг ячейки не отображается.

В ячейку в первой строке внутренней таблицы помещается <select>-блок. Этот блок пустой, он не содержит <option>-блоков. Как отмечалось ранее, пункты в раскрывающийся список добавляются при выполнении сценария.

Данный <select>-блок описан с атрибутом id, значение которого равно "mylist". Также <select>-блок содержит директиву size="1" (количество отображаемых пунктов в списке).

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Стиль элемента <select> заданы в <style>-блоке описания стилей. Он содержит всего один блок с ключевым словом select. Поэтому соответствующий стиль применяется для <select>-блоков в документе (такой блок, напомним, всего один). В соответствии с описанным стилем внутри списка данные отображаются шрифтом Times New Roman размера 14, курсивного жирного стиля. Инструкция padding: 1px 20px означает, что сверху и снизу используется внутренний отступ в 1 пиксель, а слева и справа используется внутренний отступ в 20 пикселей.
Ширина и высота элемента определяются инструкциями width: 100% и height: 100%, благодаря чему раскрывающийся список масштабируется по ширине и высоте соответствующей ячейки таблицы (с учетом наличия внутренних отступов).

В ячейку во второй строке внутренней таблицы добавлен <p>-блок. Инструкция style="font-size:12pt;font-family:Arial;font-weight:bold;" задает шрифт для текстового блока: размер шрифта 12, тип шрифта Arial, жирный стиль. При этом текстовый блок текста не содержит, и атрибут id для данного блока не описан.

Второй столбец внешней таблицы содержит <img>-блок, предназначенный для вставки изображения. В дескрипторе указано значение "mypict" для атрибута id, а также есть инструкция style="width:185px;height:185px;border-style:outset;", определяющая следующие параметры для отображения рисунка: высота и ширина изображения составляют 185 пикселей, а кроме этого, используется трехмерная выступающая рамка.

НА ЗАМЕТКУ

Изображения, предназначенные для отображения в документе, имеют размеры в 185 пикселей по ширине и высоте.

Теперь кратко опишем основные моменты, связанные с использованным в документе сценарием.

В сценарии задействовано несколько массивов. В первую очередь это массив с названиями файлов с изображениями, которые планируется отображать в документе. Массив создается командой fileNames=["fox.jpg","raccoon.jpg","wolf.jpg"]. Соответствующие файлы размещаются в папке images, которая находится на том же уровне по отношению к папке, в которой находится файл с документом.

НА ЗАМЕТКУ

Допустим, файл с документом находится в папке examples. Тогда папка images с файлами изображений должна находиться в той же папке, в которой находится папка examples.

Поэтому при указании пути к файлу с изображением необходимо перед именем файла дописать текст "../images/" (инструкция "../" означает переход на один уровень вверх). Мы объявляем переменную path со значением "../images/". В дальнейшем формирование пути к файлу с изображением будет выполняться объединением значения переменной path и элемента из массива fileNames.

Названия животных, которые будут отображаться в виде пунктов раскрывающегося списка, собраны в массиве, который создается командой animals=["Лиса","Енот","Волк"]. Три переменные fox, raccoon и wolf в качестве значений содержат ссылки на текст, описывающий животных. Данный текст предназначен для отображения в текстовом блоке под раскрывающимся списком. Для удобства все три переменные объединяются в массив. Массив создаем командой text=[fox,raccoon,wolf].

Переменные lst, mytext и pict используются для записи ссылок соответственно на объект списка, объект текстового блока и на объект изображения. Присваивание значений переменным выполняется при обработке события, связанного с загрузкой документа. В теле функции, которая присваивается значением свойству onload объекта окна window, ссылка на объект раскрывающегося списка выполняется с помощью команды lst=document.getElementById("mylist"). Аналогично с помощью метода getElementById() получаем ссылку на объект изображения (имеется в виду команда pict=document.getElementById("mypict")). Несколько иначе получаем ссылку на объект текстового блока. Напомним, что текстовый блок описан без атрибута id. Поэтому ссылку на блок вычисляем командой mytext=document.getElementsByTagName("p")[0]. Здесь мы воспользовались методом getElementsByTagName("p"), который возвращает результатом коллекцию ссылок на все элементы, соответствующие определенному дескриптору. Дескриптор, определяющий тип элементов, на которые возвращаются ссылки, передается в виде текстовой строки аргументом методу. Текстовый аргумент "p" означает, что возвращается коллекция ссылок на объекты всех <p>-блоков, которые есть в документе. Поскольку мы точно знаем, что такой блок в документе всего один,

то коллекция состоит всего из одной ссылки, и, следовательно, это элемент с нулевым индексом.

Формирование содержимого раскрывающегося списка выполняется с помощью оператора цикла. Количество пунктов в раскрывающемся списке определяется по количеству элементов в массиве animals с названиями животных (выражение animals.length). Добавление новых пунктов в раскрывающийся список выполняется так. Сначала с помощью оператора new и функции-конструктора Option() создается объект пункта меню. Аргументом функции конструктору передаются название пункта и значение свойства value соответствующей опции.

Ссылка на созданный объект записывается в качестве значения очередному элементу коллекции options, являющейся свойством объекта раскрывающегося списка. Например, инструкцией new Option(animals[k],"Знакомьтесь - это "+animals[k].toLowerCase()) создается объект для пункта меню с отображаемым названием animals[k], а значение свойства value данного пункта определяется выражением "Знакомьтесь - это "+animals[k].toLowerCase(). Последнее представляет собой объединение двух текстовых строк. Причем вторая строка вычисляется выражением animals[k].toLowerCase(), в котором из текстового значения animals[k] вызывается метод toLowerCase(). Результатом является текстовая строка, которая получается из исходной (имеется в виду значение выражения animals[k]) приведением всех символов к нижнему регистру.

Ссылка на созданный объект присваивается выражению lst.options[k]. Поскольку индексная переменная последовательно пробегает значения от 0 до animals.length-1 включительно, то каждый раз в результате такого присваивания добавляется новый элемент в коллекцию.

В теле обработчика для события, связанного с загрузкой документа, определяется обработчик события, связанного с изменением состояния раскрывающегося списка. Для этого свойству onchange объекта списка lst присваивается функция, код которой выполняется при выборе нового пункта в раскрывающемся списке. В теле функции командой pict.src=path+fileNames[lst.selectedIndex] для объекта изображения задается ссылка на файл с изображением. Технически это выглядит так, что свойству src (которое отвечает за связь с файлом изображения) объекта изображения pict присваивается результат объединения строк path и fileNames[lst.selectedIndex]. В выражении fileNames[lst.selectedIndex] использовано выражение lst.selectedIndex, значением которого является индекс выбранного на данный момент пункта меню.

НА ЗАМЕТКУ

Проще говоря, для того чтобы узнать индекс выбранного в раскрывающемся списке пункта, следует воспользоваться свойством selectedIndex. Чтобы узнать значение атрибута value выбранного пункта в раскрывающемся списке, следует обратиться к свойству value списка.

Командой pict.title=lst.value задается всплывающая подсказка для выбранного изображения. Подсказка, как несложно догадаться, совпадает со значением свойства value выбранного в раскрывающемся списке пункта.

Текст в текстовом блоке задается командой mytext.innerHTML=text[lst.selectedIndex]. Здесь мы также воспользовались выражением lst.selectedIndex для определения индекса выбранного в раскрывающемся списке пункта. Все эти команды выполняются при выборе нового пункта в списке.

В обработчике события, связанного с загрузкой документа, есть еще группа команд, назначение которых в том, чтобы задать начальное состояние документа. Так, объявляется переменная i со значением 1. Данная переменная определяет индекс выбранного в списке пункта.

НА ЗАМЕТКУ

Единичное значение индекса для выбранного пункта меню означает, что при загрузке документа будет выбран второй по порядку пункт в раскрывающемся списке (соответствует пункту **Енот**).

Командой lst.options[i].selected=true задается выбранный пункт в раскрывающемся списке. Доступ к нужному пункту получаем через свойство-коллекцию options и индекс пункта (индекс равен значению переменной i). Свойству selected этого пункта присваивается значение true, что соответствует выделению (программными методами) пункта в списке. Далее в соответствии с выбранным пунктом определяется изображение (команда pict.src=path+fileNames[i]), всплывающая подсказка (команда pict.title=lst.value) и текст в текстовом блоке (команда mytext.innerHTML=text[i]).

Рисование изображения и текста

Далее мы рассмотрим пример, формально похожий на предыдущий. Но механизмы, задействованные при решении, совершенно иные. Традиционно начнем с рассмотрения возможностей документа. На

рис. 8.21 показано, как выглядит окно браузера с загруженным в него документом.

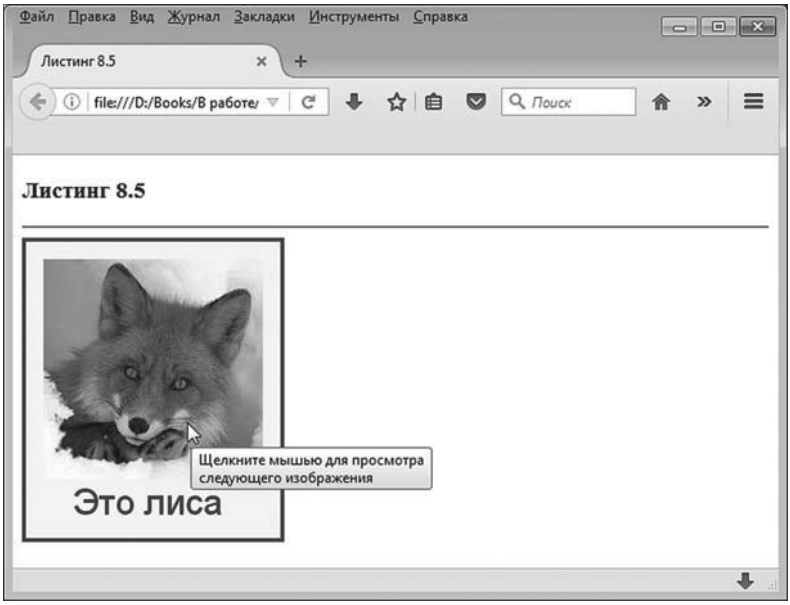

Рис. 8.21. Документ с графической областью, в которой изображена лиса

Документ содержит прямоугольную графическую область, которая обведена синей рамкой, с фоном желтого цвета. Бóльшая часть графической области занята изображением. При загрузке документа выводится изображение лисы (такое же, как в предыдущем примере). Внизу под изображением синим цветом выведена надпись "Это лиса". При наведении курсора мыши на графическую область появляется подсказка с информацией о том, что для просмотра следующего изображения необходимо щелкнуть кнопкой мыши в графической области. Текст подсказки отображается в две строки (см. рис. 8.21).

Если щелкнуть левой кнопкой мыши в графической области, то появится новое изображение и новая подпись. В частности, после изображения лисы появится изображение енота, как показано на рис. 8.22.

Если щелкнуть по изображению с енотом, появится изображение волка, как показано на рис. 8.23.

Щелчок по изображению волка приводит к тому, что снова появляется изображение лисы (см. рис. 8.21), и так далее.

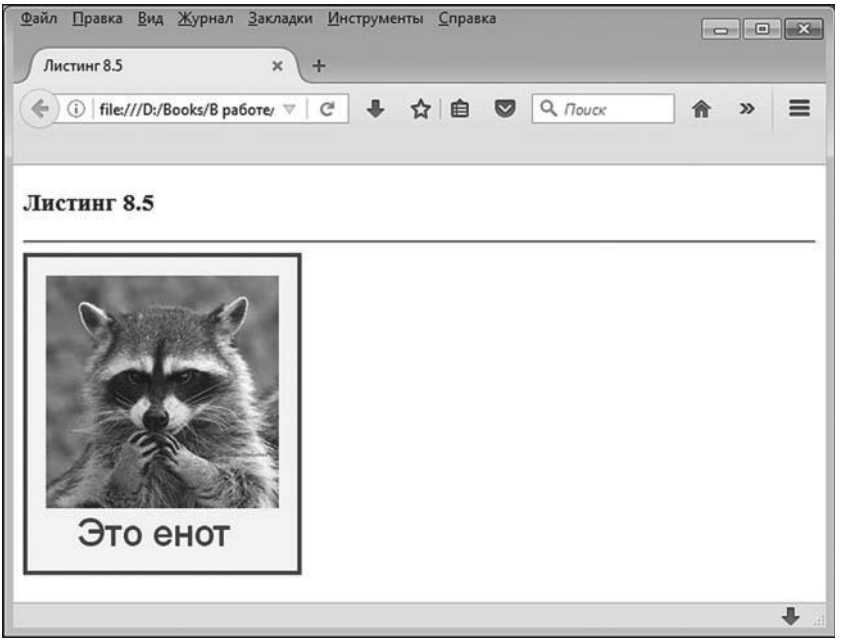

Рис. 8.22. После щелчка мышью по графической области появляется изображение енота

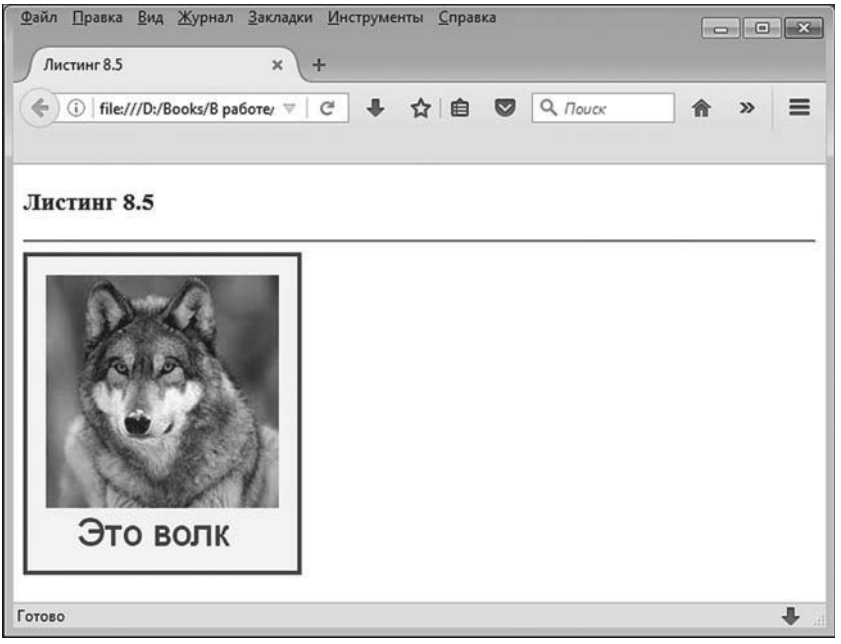

Рис. 8.23. Следующим после изображения енота появляется изображение волка

Теперь рассмотрим программный код документа, представленный в листинге 8.5.

Листинг 8.5. Рисование изображений и текста

```
<!DOCTYPE HTML>
<html>
<head>
<title>Листинг 8.5</title>
<!-- Начало сценария -->
<script type="text/javascript">
  // Переменные для записи ссылок на объекты:
  var pict,cnv,ctx
  // Массив с названиями файлов:
  var files=["fox.jpg","raccoon.jpg","wolf.jpg"]
  // Массив с текстовыми подписями:
  var text=["Это лиса","Это енот","Это волк"]
  // Переменная с указанием пути к графическим файлам:
  var path="../images/"
  // Индекс выбранного изображения:
  var sld=0
  // Обработчик события, связанного с загрузкой документа:
  window.onload=function(){
    // Создание объекта изображения:
    pict=new Image()
    // Получение ссылки на объект графической области:
    cnv=document.getElementById("mycanvas")
    // Всплывающая подсказка для графической области:
    cnv.title="Щелкните мышью для просмотра\пследующего изображения"
    // Ссылка на объект графического контекста:
    ctx=cnv.getContext("2d")
    // Размер и тип шрифта:
    ctx.font="30px Arial"
    // Вызов функции, выполняющей заливку графической
    // области и рисование текста:
    set()
    // Обработчик события, связанного с загрузкой
    // изображения из файла:
    pict.onload=function(){
```

```
      // Рисование изображения:
      ctx.drawImage(pict,15,15)
    }
    // Обработка события, связанного со щелчком мышью
    // в графической области:
    cnv.onclick=function(){
      // Новое значение для индекса
      // выбранного изображения:
      sld=(sld+1)%(files.length)
      // Вызов функции, выполняющей заливку графической
      // области и рисование текста:
      set()
    }
  }
  // Функция для заливки графической области и рисования
  // текста:
  function set(){
    // Желтый цвет заливки:
    ctx.fillStyle="yellow"
    // Заливка графической области желтым цветом:
    ctx.fillRect(0,0,cnv.width,cnv.height)
    // Синий цвет заливки:
    ctx.fillStyle="blue"
    // Ссылка на файл с изображением:
    pict.src=path+files[sld]
    // Рисование текста в графической области:
    ctx.fillText(text[sld],40,230)
  }
</script>
<!-- Завершение сценария -->
</head>
<body>
  <h3>Листинг 8.5</h3><hr>
  <!-- Графическая область -->
  <canvas height="250" width="215" id="mycanvas" style="border:3px solid #0000ff;"></canvas>
</body>
</html>
```

В основном теле документа содержится всего один блок, который определяет графическую область для отображения изображения и рисования текста. Графическая область реализуется в виде <canvas>-блока.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Блок, описанный с помощью дескрипторов <canvas> (открывающий) и </canvas> (закрывающий), задает прямоугольную область в документе, внутри которой может осуществляться отображение графики. Размеры этой графической области задают с помощью атрибутов width и height.

В описании дескриптора <canvas> указан атрибут id со значением "mycanvas", инструкцией style="border:3px solid #0000ff;" определяется стиль, в соответствии с которым вокруг графической области отображается сплошная рамка синего цвета толщиной в 3 пикселя. Ширина графической области составляет 215 пикселей, а высота графической области равна 250 пикселям.

НА ЗАМЕТКУ

Используемые в данном примере изображения имеют в ширину и высоту по 185 пикселей.

Все остальные особенности документа связаны с программным кодом сценария. В сценарии объявляются глобальные переменные pict, cnv и ctx, предназначенные для записи в них ссылок на объекты (какие именно — обсуждается далее). Массив files содержит названия файлов с изображениями, которые отображаются в документе (файлы те же и в том же месте, что и в предыдущем примере). В переменную path записан путь к папке, в которой размещены графические файлы (папка images, расположенная на том же уровне, что и папка с файлом документа). Текстовый массив text содержит подписи, которые отображаются под изображениями в графической области.

Мы используем еще одну переменную, которая называется sld. В эту переменную в процессе выполнения сценария записывается значение индекса изображения, которое в данный момент отображается в документе. Данная переменная получает нулевое начальное значение, означающее, что сначала отображается изображение, соответст-

вующее элементу с нулевым индексом в массиве files, — то есть при загрузке документа появляется изображение лисы.

Сценарий содержит описание обработчика события, связанного с загрузкой документа. В теле соответствующей функции командой pict=new Image() создается объект изображения, а ссылка на него записывается в переменную pict. Но на этом этапе объект изображения не связан ни с одним из графических файлов. Такая связь будет установлена позже.

С помощью команды cnv=document.getElementById("mycanvas") в переменную cnv записывается ссылка на объект графической области. Получив ссылку на этот объект, мы с помощью инструкции cnv.title="Щелкните мышью для просмотра\nследующего изображения" задаем текст для всплывающей подсказки, которая будет отображаться при наведении курсора мыши на графическую область. Особенность данной команды в том, что текст, который присваивается значением свойству title объекта cnv, содержит инструкцию \n разрыва строки. В результате в том месте, где размещена данная инструкция, во всплывающей подсказке будет выполняться переход к новой строке (поэтому всплывающая подсказка отображается в две строки — см. рис. 8.21).

Напомним, что графическая область нам необходима для осуществления графического вывода. Но самого доступа к графической области для рисования в ней изображений мало. Необходимо еще иметь специальный объект, который в некотором смысле является посредником и с помощью которого, собственно, и выполняется рисование в графической области. Данный объект называется *объектом графического контекста*. Получить доступ к объекту графического контекста можно с помощью метода getContext(). Метод вызывается из объекта графической области. Результатом метод возвращает ссылку на объект графического контекста. Аргументом методу передается текстовое значение, определяющее тип возвращаемого объекта графического контекста. Обычно используют стандартный объект, и в таком случае методу getContext() передается аргумент "2d".

Таким образом, в результате выполнения команды ctx=cnv.getContext("2d") в переменную ctx записывается ссылка на объект графического контекста, связанного с графической областью, ссылка на объект которой записана в переменную cnv. Переменную ctx мы используем в команде ctx.font="30px Arial" при определении типа и размера шрифта. Здесь имеется в виду шрифт, применяемый при рисовании текста в графической

области. Затем вызывается функция set(). Мы сразу проанализируем код этой функции — так будет легче понять, что происходит. Итак, выполнение функции set() означает выполнение пяти команд. В первую очередь командой ctx.fillStyle="yellow" задается желтый цвет заливки. Этот цвет используется при заливке графической области. Заливку графической области выполняем командой ctx.fillRect(0,0,cnv.width,cnv.height), в которой из объекта графического контекста вызывается метод fillRect(). Методу передается четыре аргумента. Первые два аргумента определяют координаты левого верхнего угла прямоугольной области, для которой выполняется заливка. Нулевые аргументы означают, что левый верхний угол области заливки совпадает с левым верхним углом графической области. Третий и четвертый аргументы метода fillRect() определяют соответственно ширину и высоту (в пикселях) заливаемой области. Выражения cnv.width и cnv.height — это ширина и высота графической области. Как результат, желтым цветом заливается вся графическая область.

После заливки графической области желтым цветом мы командой ctx.fillStyle="blue" устанавливаем синий цвет заливки. Этот цвет используется, когда командой ctx.fillText(text[sld],40,230) в графической области рисуется текст. Рисуемое текстовое значение передается первым аргументом методу fillText(). В данном случае это выражение text[sld], являющееся элементом массива text с индексом sld. Второй и третий аргументы метода fillText() определяют координаты точки в графической области, начиная с которой выполняется рисование текста. Сам метод fillText() вызывается из объекта графического контекста.

Наконец, командой pict.src=path+files[sld] для объекта изображения устанавливается связь с файлом изображения. Технически задача решается присваиванием свойству src объекта pict текстового значения, определяющего полный путь к файлу с изображением.

НА ЗАМЕТКУ

Графические координаты определяются в пикселях по отношению к левому верхнему углу графической области. Точка в левом верхнем углу графической области имеет нулевые координаты. Горизонтальная координата отсчитывается вправо, а вертикальная координата отсчитывается вниз.

Многие методы вызываются из объекта графического контекста. Следует понимать, что этот объект существует не сам по себе,

а в привязке к определенной графической области. Поэтому, используя определенный объект графического контекста, мы автоматически оперируем с определенной графической областью.
Подход, использованный в рассматриваемом сценарии в части рисования, сводится к следующим действиям. Сначала область заливается желтым цветом. Затем цвет заливки меняется на синий, и этим цветом рисуется текст. После этого рисуется изображение (пояснения идут далее).

Важно следующее: определение значения свойства src для объекта pict не означает, что изображение появится в графической области. Это не так. Выполнение команды pict.src=path+files[sld] приводит к началу загрузки изображения в документ. Чтобы отобразить изображение, из объекта графического контекста ctx необходимо вызвать метод drawImage(). Первым аргументом методу передается ссылка на объект изображения, которое следует нарисовать в графической области. Второй и третий аргументы метода определяют координаты точки в графической области, начиная с которой рисуется изображение (в этой точке будет размещен левый верхний угол изображения). Мы в данном случае используем команду ctx.drawImage(pict,15,15). Но есть одна проблема. Состоит она в том, что если на момент выполнения данной команды изображение не успеет загрузиться, то оно не будет отображено. Чтобы избежать неприятностей, мы размещаем команду ctx.drawImage(pict,15,15) в теле обработчика события, связанного с загрузкой изображения. Обработчик описывается так: свойству onload объекта изображения pict присваивается анонимная функция, в теле которой содержится означенная выше команда. В результате происходит следующее: при вызове функции set() выполняется команда, приводящая к загрузке изображения. Когда загрузка изображения завершается, в игру автоматически вступает обработчик события, связанного с загрузкой изображения. Выполнение обработчика приводит к рисованию изображения в графической области.

НА ЗАМЕТКУ

Аргументы методу drawImage() могут передаваться по-разному. В частности, можно задать не только точку в графической области, начиная с которой отображается рисунок, но еще и задать точку в области рисунка, начиная с которой рисуется изображение. Также задают ширину и высоту отображаемого фрагмента, равно как ширину и высоту области, в которую выводится фрагмент изображения.

Для отображения (рисования) текста, кроме метода fillText() нередко используют метод strokeText(). Методом strokeText() отображаются полые (не закрашенные внутри) буквы. Цвет рамки для символов определяется с помощью свойства strokeStyle.

В обработчике события, связанного с загрузкой документа, кроме описанных выше команд еще описывается обработчик события, связанного со щелчком в графической области. Свойству onclick объекта графической области cnv присваивается функция, в теле которой выполняется две команды.

Сначала командой sld=(sld+1)%(files.length) вычисляется новое значение переменной sld. Хотя команда немного замысловатая, на самом деле все просто: значение переменной sld циклически увеличивается на единицу. Проще говоря, если текущее значение переменной sld меньше индекса последнего элемента в массиве files, то оно увеличивается на единицу. Если текущее значение переменной sld равно значению индекса последнего элемента массива files, то переменная получает нулевое значение.

НА ЗАМЕТКУ

Значение выражения (sld+1)%(files.length) — это остаток от деления выражения sld+1 на число files.length, равное, очевидно, количеству элементов в массиве files. Результат лежит в диапазоне от 0 до files.length-1 включительно.

После определения нового значения переменной sld вызывается функция set(). Вызов функции приводит к заливке желтым цветом графической области, отображению текста и инициирует загрузку изображения. По завершении загрузки изображения вызывается обработчик, и изображение рисуется в графической области.

НА ЗАМЕТКУ

Заливка графической области (в данном случае желтым цветом) играет важную роль. Если ее не выполнять, то каждый новый текст будет рисоваться поверх уже существующего текста. Поэтому заливка области на самом деле позволяет очистить область от уже содержащихся там графических построений.

Для очистки графической области может использоваться метод clearRect().

Создание изображений в сценарии

Изображения в документе можно создавать программными методами, с помощью сценария. Проще говоря, сценарий может содержать команды, которыми в документе отображаются такие графические примитивы, как линии, прямоугольники или, например, дуги. Для построения этих примитивов имеются специальные методы, которые вызываются из объекта графического контекста (здесь речь идет о стандартном объекте графического контекста, который возвращается методом getContext() при передаче ему аргумента "2d").

Например, для создания линий используется метод lineTo(). Аргументами методу передаются координаты конечной точки. Начальной точкой для создания линии является текущая точка. В начальный момент определить точку, с которой все начинается, можно посредством метода moveTo(). Аргументом методу передаются координаты точки, с которой начинается рисование.

Для создания полого (не закрашенного) прямоугольника используют функцию strokeRect(). Заполненный прямоугольник создается методом fillRect(). Аргументами методам передаются координаты левого верхнего угла прямоугольника и его размеры (ширина и высота).

Дугу можно создать с помощью метода arc(). Аргументами методу передаются координаты точки центра окружности, радиус окружности, углы, определяющие положение граничных точек дуги, а также логический параметр, определяющий направление рисования дуги (по часовой стрелке или против часовой стрелки).

Толщина линий, которыми рисуется контур фигуры, определяется свойством lineWidth объекта графического контекста. Цвет линий задается через свойство strokeStyle, а цвет заливки определяется свойством fillStyle. Причем нужно иметь в виду, что настройки цвета линий применяются для всего контура. В начале рисования контура вызывается метод beginPath(). Создание контура не означает его автоматического отображения в графической области. Для отображения контура после его создания следует вызвать метод stroke(). Как все это работает на практике, иллюстрирует документ, представленный в листинге 8.6.

Листинг 8.6. Создание изображений в сценарии

```
<!DOCTYPE HTML>
<html>
<head>
```

```
<title>Листинг 8.6</title>
<!-- Начало сценария -->
<script type="text/javascript">
  window.onload=function(){
    // Ссылка на объект графического контекста:
    var ctx=document.getElementsByTagName("canvas")[0].getContext("2d")
    // Ширина графической области:
    ctx.canvas.width=580
    // Высота графической области:
    ctx.canvas.height=330
    // Толщина, тип и цвет рамки вокруг
    // графической области:
    ctx.canvas.style.border="1px solid #009900"
    // Рисование наклонных линий разного цвета
    // и разной толщины.
    // Координаты начальной точки для рисования и
    // характерный размер для рисования линий:
    var x=10,y=10,L=200
    // Рисование наклонных линий:
    for(var k=0;k<=7;k++){
      // Начало нового контура:
      ctx.beginPath()
      // Толщина линии:
      ctx.lineWidth=1+k
      // Цвет линии:
      ctx.strokeStyle="rgb("+(255-k*20)+","+(20*k)+",255)"
      // Новое значение для горизонтальной координаты:
      x+=10
      // Начальная точка:
      ctx.moveTo(x,y)
      // Создание линии:
      ctx.lineTo(x+L/5,y+L)
      // Отображение линии:
      ctx.stroke()
```

```
}
// Рисование спирали.
// Горизонтальная координата для начальной точки:
x+=L/5+20
// Технические переменные, определяющие
// знак приращения для каждой из координат:
var sx=1,sy=-1
// Создание нового контура:
ctx.beginPath()
// Толщина линии:
ctx.lineWidth=3
// Цвет линии:
ctx.strokeStyle="blue"
// Начальная точка:
ctx.moveTo(x,y)
// Создание спирали:
while(L>0){
  // Новое значение для горизонтальной координаты:
  x+=sx*L
  // Горизонтальная линия:
  ctx.lineTo(x,y)
  // Изменение знака для приращения
  // по вертикальной координате:
  sy*=-1
  // Новое значение для вертикальной координаты:
  y+=sy*L
  // Вертикальная линия:
  ctx.lineTo(x,y)
  // Изменение длины отрезка спирали:
  L-=10;
  // Изменение знака для приращения
  // по горизонтальной координате:
  sx*=-1
}
```

```
// Отображение контура (спирали):
ctx.stroke()
// Рисование многоугольника.
// Массив с координатами точек:
var points=[[400,50],[450,20],[550,60],[510,170],[420,120]]
// Создание нового контура:
ctx.beginPath()
// Толщина линии:
ctx.lineWidth=5
// Цвет линии:
ctx.strokeStyle="rgb(255,0,0)"
// Цвет заливки:
ctx.fillStyle="yellow"
// Начальная точка:
ctx.moveTo(points[0][0],points[0][1])
// Соединение точек линиями:
for(var i=1;i<points.length;i++){
  // Создание линии:
  ctx.lineTo(points[i][0],points[i][1])
}
// Замыкание контура (соединение первой
// и последней точек):
ctx.closePath()
// Заливка области контура:
ctx.fill()
// Отображение контура (линий):
ctx.stroke()
// Рисование прямоугольника.
// Новое значение для горизонтальной координаты:
x=20
// Новое значение для вертикальной координаты:
y=230
// Высота прямоугольника:
L=80
```

```
// Создание нового контура:
ctx.beginPath()
// Толщина линии:
ctx.lineWidth=15
// Цвет линии:
ctx.strokeStyle="rgb(0,100,0)"
// Цвет заливки:
ctx.fillStyle="rgb(0,255,0)"
// Создание прямоугольника:
ctx.strokeRect(x,y,2*L,L)
// Заливка области прямоугольника:
ctx.fillRect(x,y,2*L,L)
// Отображение контура (рамки прямоугольника):
ctx.stroke()
// Рисование первой дуги.
// Радиус дуги:
var R=80
// Начало нового контура:
ctx.beginPath()
// Толщина линии:
ctx.lineWidth=5
// Цвет линии:
ctx.strokeStyle="rgb(100,100,100)"
// Цвет заливки:
ctx.fillStyle="rgb(255,100,100)"
// Создание дуги:
ctx.arc(x+2*L+20+R,y,R,0,Math.PI,false)
// Замыкание контура:
ctx.closePath()
// Заливка контура:
ctx.fill()
// Отображение контура:
ctx.stroke()
// Рисование второй дуги.
```

```
    // Начало нового контура:
    ctx.beginPath()
    // Создание дуги:
    ctx.arc(x+2*L+40+3*R,y+R,R,0,Math.PI,true)
    // Замыкание контура:
    ctx.closePath()
    // Заливка контура:
    ctx.fill()
    // Отображение контура:
    ctx.stroke()
  }
</script>
<!-- Завершение сценария -->
</head>
<body>
  <h3>Листинг 8.6</h3><hr>
  <!-- Графическая область -->
  <canvas></canvas>
</body>
</html>
```

Тело документа содержит описание пустого <canvas>-блока. Код сценария размещен в обработчике события, связанного с загрузкой документа. Как выглядит загруженный в окно браузера документ, показано на рис. 8.24.

Документ содержит обведенную тонкой рамкой графическую область, внутри которой отображаются:

- группа косых линий переменной толщины, цвет которых изменяется от сиреневого до светло-фиолетового;
- прямоугольная спираль синего цвета;
- неправильный пятиугольник с рамкой красного цвета, закрашенный желтым цветом;
- прямоугольник с рамкой темно-зеленого цвета, закрашенный зеленым цветом;

- выгнутая вниз полуокружность с рамкой серого цвета, закрашенная бледно-красным цветом;
- выгнутая вверх полуокружность с серой рамкой и бледно-красным фоном.

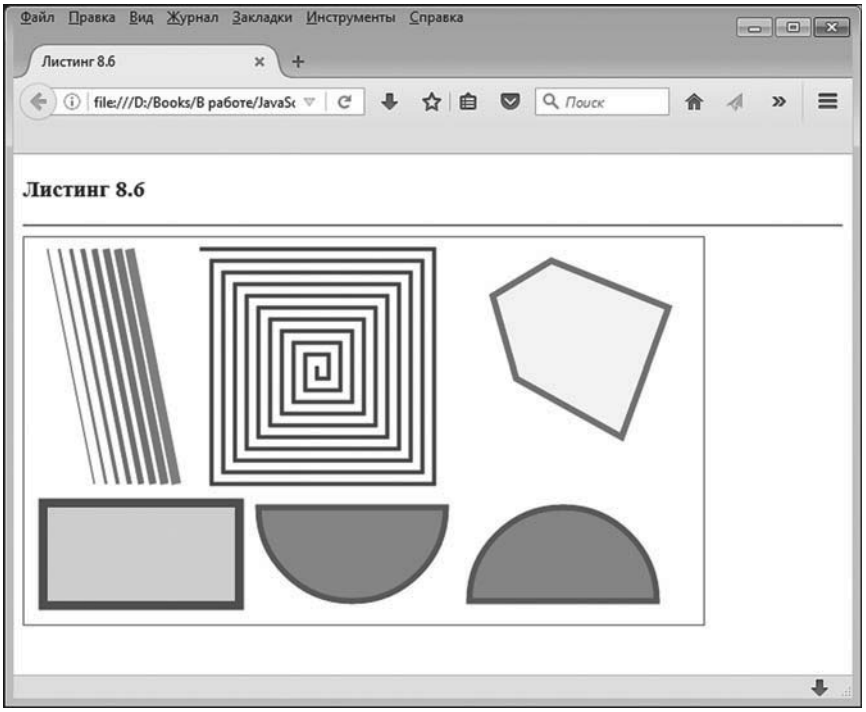

Рис. 8.24. Документ с изображениями, созданными программными методами

Проанализируем программный код, с помощью которого создается все означенное великолепие.

Начальные команды в коде функции, используемой для обработки события, связанного с загрузкой документа, имеют отношение к настройке параметров графической области. Так, командой `ctx=document.getElementsByTagName("canvas")[0].getContext("2d")` мы получаем ссылку на объект графического контекста, связанного с графической областью в документе. Мы использовали громоздкую, но показательную команду. Значением выражения `document.getElementsByTagName("canvas")` является ссылка на коллекцию, содержащую ссылки на графические области документа. Мы знаем, что такая область одна. Доступ к ней (ссылку на эту область) получаем, добавив после указанного выражения ну-

левой индекс (в квадратных скобках). Уже через эту ссылку вызывается метод getContext(), который и возвращает ссылку на объект графического контекста.

Высоту и ширину графической области мы задаем в сценарии. Но преднамеренно созданная проблема состоит в том, что мы получали ссылку на объект графического контекста, не запоминая ссылку на объект графической области. Ранее уже отмечалось, что объект графического контекста и объект графической области связаны между собой. Мы до этого получали доступ к объекту графического контекста через объект графической области (вызвав метод getContaxt()). Чтобы выполнить обратную процедуру и получить ссылку на объект графической области, связанной с объектом графического контекста, используют свойство canvas. Так, ширина графической области определяется командой ctx.canvas.width=580, а для определения высоты графической области используем команду ctx.canvas.height=330. Толщина, тип и цвет рамки вокруг графической области определяются командой ctx.canvas.style.border="1px solid #009900".

После того как параметры графической области заданы, начинается создание изображений.

Для рисования наклонных линий (разного цвета и разной толщины) объявляются вспомогательные переменные x и y со значением 10 каждая, а также переменная L со значением 200. Переменные x и y используются при определении начальной точки, с которой рисуются линии, а переменная L используется при определении длины таких линий.

Создание наклонных линий реализуется через оператор цикла, в котором индексная переменная k пробегает значения от 0 до 7 включительно. Каждый цикл начинается командой ctx.beginPath(), означающей начало создания нового контура. Затем командой ctx.lineWidth=1+k определяется толщина линии для данного контура. Таким образом, толщина первой линии равна 1 пикселю, а толщина каждой следующей линии увеличивается на 1 пиксель по отношению к толщине предыдущей линии. Цвет линии задается выражением ctx.strokeStyle="rgb("+(255-k*20)+","+(20*k)+",255)". Правая часть данного выражения представляет собой текстовую строку, которая вычисляется следующим образом: это объединение строки "rgb(", значения выражения (255-k*20), текстовой строки ",", числового значения (20*k) и еще одной строки ",255)". То, что получается в результате, представляет собой текстовую строку, в ко-

торой записано выражение вида rgb(число,число,число). Данное выражение является кодом цвета в формате RGB.

> **НА ЗАМЕТКУ**
>
> Кроме уже известных нам способов определения цвета (текст с названием цвета и текст с шестнадцатеричным кодом цвета в формате RGB), существует возможность задавать цвет в виде текстовой строки, которая содержит ключевое слово rgb и три числа в круглых скобках (числа разделяются запятыми). Каждое число лежит в диапазоне значений от 0 до 255 и определяет содержание соответственно красного, зеленого и синего цвета в данном цвете. Например, код "rgb(255,0,0)" означает красный цвет, код "rgb(0,255,0)" соответствует зеленому цвету, а код "rgb(0,0,255)" означает синий цвет.

Таким образом, для каждого нового цикла в коде цвета содержание красного цвета уменьшается, содержание зеленого цвета увеличивается, а содержание синего цвета остается неизменным. Соответственно, цвет линий меняется.

Командой x+=10 на 10 увеличивается значение переменной x, после чего командой ctx.moveTo(x,y) задается начальная точка для создания линии. Линия создается командой ctx.lineTo(x+L/5,y+L), в соответствии с которой конечная точка отстоит от начальной точки по горизонтали на расстояние L/5, а по вертикали на расстояние L. Но создание линии не означает, что она будет отображена. Для отображения построенного контура (состоящего в данном случае из одной линии) используем команду ctx.stroke().

Следующий этап — рисование спирали. Для начала командой x+=L/5+20 мы задаем горизонтальную координату начальной точки, с которой будет рисоваться спираль. Сама спираль рисуется так: сначала рисуется горизонтальная линия, затем вертикальная, затем снова горизонтальная и так далее. При рисовании горизонтальных линий горизонтальная координата конечной точки сначала увеличивается, затем уменьшается, затем снова увеличивается и уменьшается и так далее. Аналогично с вертикальными линиями: сначала вертикальная координата конечной точки увеличивается, затем уменьшается, снова увеличивается, и так по очереди. Еще один момент связан с тем, что величина, на которую изменяются горизонтальная и вертикальная координаты за каждую итерацию, уменьшается (за одну итерацию рисуются горизонтальная и вертикальная линии). Если бы величина

приращения не изменялась, то уже через две итерации мы получили бы квадрат.

Для решения стоящей перед нами задачи мы объявляем переменные sx и sy, которые будут принимать значения 1 и -1, определяя тем самым знак приращения соответственно горизонтальной и вертикальной координаты конечной точки при создании очередной линии, входящей в спираль. Командой ctx.beginPath() начинаем формирование нового контура (спирали), команда ctx.lineWidth=3 определяет толщину линий в 3 пикселя, а команда ctx.strokeStyle="blue" задает синий цвет для линии спирали. Начальную точку для рисования спирали задаем командой ctx.moveTo(x,y). Сама спираль создается с помощью оператора цикла while. Оператор выполняется, пока истинно условие L>0. В теле оператора цикла командой x+=sx*L получает приращение горизонтальная координата для конечной точки горизонтальной прямой. Прямая создается командой ctx.lineTo(x,y). Далее командой sy*=-1 меняем знак приращения для вертикальной координаты, после чего благодаря команде y+=sy*L вертикальная координата получает новое значение. Оно используется в команде ctx.lineTo(x,y) при создании вертикальной линии. Затем командой L-=10 уменьшается значение для длины линии, а командой sx*=-1 изменяется знак для приращения по горизонтальной координате. По завершении оператора цикла командой ctx.stroke() спираль отображается в графической области.

Для создания многоугольника объявляется массив points, содержащий значения координат точек, образующих вершины многоугольника (элементами массива points являются массивы из двух элементов каждый). Создание нового контура начинается командой ctx.beginPath(). Толщина линии задается командой ctx.lineWidth=5. Цвет линии определяем командой ctx.strokeStyle="rgb(255,0,0)" (красный цвет). Также командой ctx.fillStyle="yellow" мы устанавливаем желтый цвет заливки. Начальная точка определяется командой ctx.moveTo(points[0][0],points[0][1]). Очевидно, что это первая точка в массиве points. Далее запускается оператор цикла, в котором перебираются все точки, кроме самой первой. Линии создаются командами ctx.lineTo(points[i][0],points[i][1]). Здесь (и выше) мы учли, что элемент массива points — это массив из двух элементов, первый из которых определяет горизонтальную координату точки, а второй — вертикальную. По завершении оператора цикла все точки, описанные в массиве points, последовательно соединены линиями. Осталось соединить последнюю и первую точки. Для этого используем команду ctx.closePath(), которая в общем случае соединяет линией

последнюю и первую точки контура. Для заливки полученной фигуры используем команду ctx.fill(). Отображение линий выполняется командой ctx.stroke().

На следующем этапе рисуется закрашенный прямоугольник. Точнее, это два прямоугольника, которые накладываются друг на друга. Перед началом построений переменные x, y и L получают новые значения. Толщина линии устанавливается равной 15 пикселям. Цвет линии определяется инструкцией ctx.strokeStyle="rgb(0,100,0)" (темно-зеленый), а цвет заливки определяем командой ctx.fillStyle="rgb(0,255,0)" (зеленый). Закрашенный прямоугольник создаем командами ctx.strokeRect(x,y,2*L,L) и ctx.fillRect(x,y,2*L,L). Но на самом деле ситуация такая. Сначала первой командой создается не закрашенный прямоугольник (то есть прямоугольная рамка), а затем в то же место, поверх первого, помещается закрашенный прямоугольник. Рамка в первом прямоугольнике достаточно толстая. При наложении второго прямоугольника она частично перекрывается. Внешне эффект такой, как если бы у нас был закрашенный прямоугольник с рамкой.

НА ЗАМЕТКУ

Желающие могут поэкспериментировать: поменять порядок выполнения команд ctx.strokeRect(x,y,2*L,L) и ctx.fillRect(x,y,2*L,L) и посмотреть, каков будет визуальный эффект (не закрашенный прямоугольник в таком случае накладывается на закрашенный).

Аргументы у методов strokeRect() и fillRect(): координаты левого верхнего угла прямоугольника, его ширина и высота.

Также в документе рисуются две дуги (полуокружности). Их радиус определяется переменной R. При рисовании первой полуокружности устанавливаем толщину линии равной 5 пикселям, цвет линии определяем командой ctx.strokeStyle="rgb(100,100,100)" (один из оттенков серого цвета), а цвет заливки задаем командой ctx.fillStyle="rgb(255,100,100)" (бледно-красный). Дуга создается командой ctx.arc(x+2*L+20+R,y,R,0,Math.PI,false). Первый и второй аргументы метода arc() определяют координаты центра окружности, на основе которой строится дуга. Третий аргумент — радиус окружности. Четвертый и пятый аргументы — это угол на начальную и конечную точку дуги. В данном случае начальный угол нулевой, а конечный равен π. Значение числа π мы получаем с помощью выражения Math.PI. Но эти точки (начальную и конечную) можно соединять дугой двумя способами: по ходу часовой стрелки

и против хода часовой стрелки. Если соединять точки следует по ходу часовой стрелки (от начальной точки к конечной), то шестой аргумент указывается равным false. Если соединять точки следует против хода часовой стрелки, то шестой аргумент метода arc() равен true. Используя это отличие, получаем две полуокружности: одна выгнута вниз, другая выгнута вверх. При построении второй полуокружности параметры линии и цвет заливки не меняются, поэтому они такие же, как и при создании первой дуги. Для замыкания контура в обоих случаях вызывается метод closePath(). Заливку области контура выполняем с помощью метода fill(). При создании второй дуги (команда ctx.arc(x+2*L+40+3*R,y+R,R,0,Math.PI,true)) координата центра окружности смещается не только вправо, но еще и вниз.

События

> Знаешь, одна из самых серьезных потерь в битве — это потеря головы.
>
> *Л. Кэрролл «Алиса в Стране чудес»*

До этого мы много раз использовали обработку событий. Вместе с тем есть некоторые вопросы, касающиеся обработки событий, которые желательно рассмотреть более детально. Именно этим и займемся далее.

Объект события

Когда происходит то или иное событие, для него автоматически создается объект. Этот объект содержит полезную информацию, которую нередко необходимо использовать в сценарии. Примером такой информации могут быть сведения о положении курсора мыши в момент, когда произошло событие. Но в любом случае до того, как получить из объекта события нужную информацию, предварительно необходимо получить доступ к самому объекту события. Как это сделать? Ответ простой. Состоит он в том, что при обработке события объект события передается аргументом функции-обработчику.

> **НА ЗАМЕТКУ**
>
> Ранее при обработке различных событий мы присваивали обработчикам в качестве значений анонимные функции. Эти функции не

имели аргументов. На самом деле в функцию-обработчик аргументом передается объект события. Противоречия здесь нет, поскольку в рассматриваемых ранее примерах объект события напрямую не использовался. Тот факт, что он все равно передается в обработчик, мало что меняет. Наличие у функции «лишних» аргументов обычно не приводит к проблемам.

Если функцию-обработчик описать с аргументом, то этот аргумент будет отождествляться с объектом события. Соответственно, в программном коде функции с таким аргументом можно и должно обращаться как с объектом события. Как иллюстрацию к использованию объекта события мы рассмотрим пример. Код соответствующего документа представлен в листинге 8.7.

Листинг 8.7. Координаты курсора мыши

```
<!DOCTYPE HTML>
<html>
<head>
<title>Листинг 8.7</title>
<!-- Начало сценария -->
<script type="text/javascript">
  // Обработка события, связанного с движением
  // курсора мыши:
  document.onmousemove=function(evt){
    // Ссылка на объект блока вывода:
    var d=document.getElementsByTagName("div")[0]
    // Переменная для формирования
    // содержимого блока вывода:
    var txt="<b>Координаты курсора мыши</b><br>"
    // Горизонтальная координата курсора мыши:
    txt+="Горизонтальная координата: <b>"+evt.clientX+"</b><br>"
    // Вертикальная координата курсора мыши:
    txt+="Вертикальная координата:    <b>"+evt.clientY+"</b>"
    // Содержимое блока вывода:
    d.innerHTML=txt
  }
```

```
</script>
<!-- Завершение сценария -->
</head>
<body>
  <h3>Листинг 8.7</h3><hr>
  <!-- Блок вывода -->
  <div style="width:280px;height:80px;border-style:outset;font-size:15pt;padding:10px;backgroun
d-color:#f5f5f5;"><b>Координаты курсора мыши</b></div>
</body>
</html>
```

Тело документа содержит -блок, в котором жирным стилем отображается текст Координаты курсора мыши. Стиль блока определяется инструкцией style="width:280px;height:80px;border-style:outset;font-size:15pt;padding:10px;background-color:#f5f5f5;", которой задаются ширина и высота блока (280 и 80 пикселей соответственно), трехмерная выступающая рамка, шрифт размера 15, внутренние отступы в 10 пикселей и определяется светло-серый цвет фона.

Еще одно место в документе, представляющее интерес, — сценарий, состоящий из описания обработчика для события, связанного с движением мышью. Более конкретно, свойству onmousemove объекта документа document присваивается анонимная функция. Специфика ситуации в том, что у функции имеется аргумент, обозначенный как evt. Этот аргумент отождествляется с объектом события, которое обрабатывается.

В теле функции командой d=document.getElementsByTagName("div")[0] мы получаем ссылку на -блок (будем называть его блоком вывода) и записываем ее в переменную d. Затем в несколько этапов формируется значение текстовой переменной txt, и командой d.innerHTML=txt это значение применяется к блоку вывода как внутреннее содержимое. Интерес для нас в данном случае представляют команды evt.clientX и evt.clientY, используемые при формировании значения переменной txt. Свойства clientX и clientY объекта события возвращают координаты (в рабочей области документа) курсора мыши на момент, когда произошло событие. Таким образом, при перемещении курсора мыши в области документа (а именно это событие обрабатывается) блок вывода будет содержать координаты курсора мыши. Причем эти значения будут

изменяться синхронно с изменением положения курсора мыши в области документа.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Как известно, в HTML кратные пробелы игнорируются. Поэтому добиться некоторого подобия форматирования за счет добавления в текст лишних пробелов просто так не получится. Для «насильственного» добавления пробела в веб-документ в HTML-коде используют инструкцию ` `. Соответственно, три такие инструкции означают вставку трех пробелов. Данный прием мы использовали при формировании текстового значения переменной `txt` для того, чтобы числовые значения для горизонтальной и вертикальной координат курсора мыши отображались в блоке вывода примерно на одном уровне.

На рис. 8.25 показано, как будет выглядеть документ при загрузке.

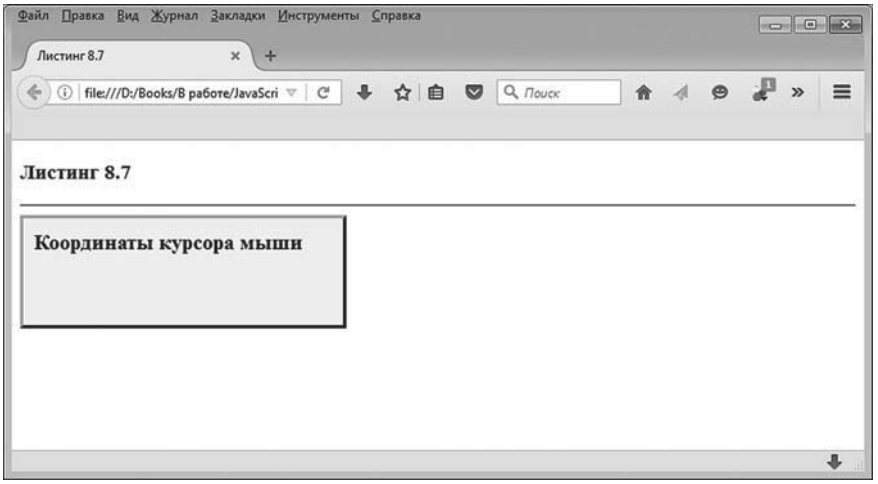

Рис. 8.25. Вид документа при загрузке

Блок вывода содержит только лишь надпись **Координаты курсора мыши**. Это тот текст, который указан непосредственно в `<div>`-блоке документа. Но как только курсор мыши оказывается в рабочей области документа, в блоке вывода появляется информация о координатах курсора мыши, как это показано на рис. 8.26.

Как отмечалось выше, при перемещении курсора мыши в области документа автоматически меняются значения для координат курсора

мыши, отображаемые в блоке вывода. Ситуация проиллюстрирована на рис. 8.27: изменение положения курсора привело к изменению отображаемых значений.

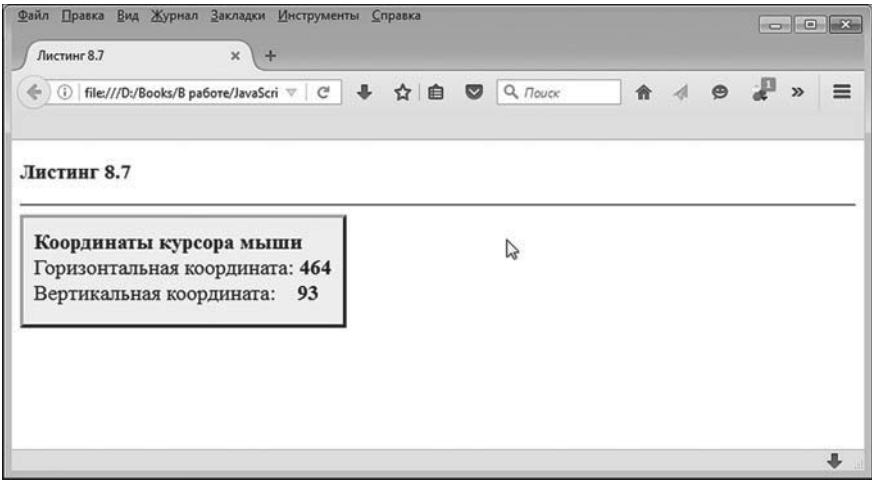

Рис. 8.26. При движении курсора мыши в области документа отображаются координаты курсора мыши

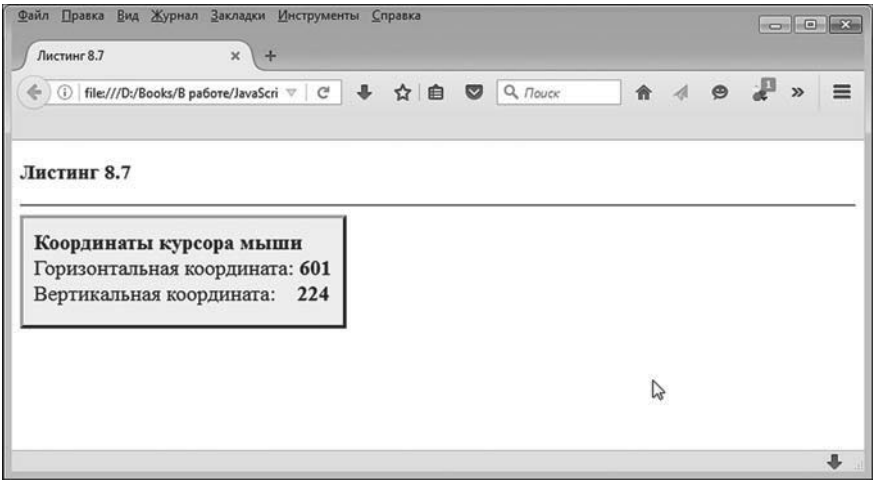

Рис. 8.27. Изменение положения курсора приводит к автоматическому изменению отображаемых значений для координат курсора мыши

Понятно, что мы рассмотрели простой пример. Вместе с тем он дает некоторое представление о принципах использования объекта события.

ДЕТАЛИ

В старых версиях браузера Internet Explorer объект события не передавался аргументом обработчику, а реализовывался в виде свойства event объекта окна window. И хотя в последних версиях данного браузера ситуация исправлена, все же мы кратко остановимся на общих подходах, используемых при ее решении, поскольку соответствующие приемы могут стать полезными в иных подобных ситуациях.

Итак, если нет уверенности в том, что браузер поддерживает механизм передачи объекта события аргументом обработчика, имеет смысл разместить небольшой дополнительный блок кода в теле обработчика. В частности, код функции-обработчика может быть таким:

```
function(evt){
  evt=evt||window.event
  // Код обработки события
}
```

Выражение evt||window.event возвращает значением ссылку на объект evt, если эта ссылка не пустая, а в противном случае результатом будет ссылка на window.event. То есть в данном случае схема такая: проверяется наличие аргумента у функции-обработчика. Если он передан, то объект используется. Если аргумент не передан, то используется свойство event объекта window.

Тот же подход может быть реализован несколько иным способом:

```
function(evt){
  if(!evt){
    evt=window.event
  }
  // Код обработки события
}
```

В данной ситуации проверяется «ложность» аргумента evt. Поскольку речь идет об объекте, то наличие непустой ссылки соответствует истинному значению. Соответственно, при пустой ссылке (аргумент не передан) получаем ложное значение, и тогда значение выражения !evt будет истинным. В таком случае командой evt=window.event в переменную evt записывается ссылка на объект события event, являющийся свойством объекта window.

Для иллюстрации работы с объектами событий мы рассмотрим еще один пример. В данном случае мы по объекту события будем определять тип события и источник события — компонент, на котором произошло событие. Рассмотрим код документа, представленный в листинге 8.8.

Листинг 8.8. Тип и источник события

```
<!DOCTYPE HTML>
<html>
<head>
<title>Листинг 8.8</title>
<!-- Начало сценария -->
<script type="text/javascript">
  // Массив с кодами цвета для применения
  // к фону блока вывода
  var colors=new Array(5)
  // Переменная для запоминания индекса выбранного цвета:
  var selected
  // Формирование массива с кодами цвета для применения
  // в качестве фона к блоку вывода:
  for(var k=0;k<colors.length;k++){
    // Значение элемента массива:
    colors[k]="rgb(0,"+(215+k*10)+",0)"
  }
  // Функция для применения цвета фона в соответствии со
  // значением индекса выбранного цвета:
  function setColor(){
    // Ссылка на блок вывода:
    var ref=document.getElementById("mydiv")
    // Применение фонового цвета к блоку:
    ref.style.background=colors[selected]
    // Отображение кода цвета в блоке вывода:
    ref.innerHTML=colors[selected]
  }
  // Функция для применения следующего цвета из массива:
  function next(){
    // Циклическое увеличение значения индекса:
    selected=(selected+1)%colors.length
    // Применение цвета:
    setColor()
  }
```

```
// Функция для применения предыдущего цвета из массива:
function prev(){
  // Если текущее значение индекса положительно:
  if(selected>0){
    // Значение индекса уменьшается на единицу:
    selected--
  }
  // Если текущее значение индекса нулевое:
  else{
    // Значение индекса последнего элемента в массиве:
    selected=colors.length-1
  }
  // Применение цвета:
  setColor()
}
// Функция для использования в качестве
// обработчика событий:
function handler(evt){
  // Переменные для записи ссылок на объекты кнопок:
  var nxt,prv
  // Ссылка на объект кнопки "Следующий":
  nxt=document.getElementById("next")
  // Ссылка на объект кнопки "Предыдущий":
  prv=document.getElementById("prev")
  // Если событие связано с завершением загрузки:
  if(evt.type=="load"){
    // Нулевое значение индекса для выбранного цвета:
    selected=0
    // Применение цвета:
    setColor()
    // Обработчик для кнопки "Следующий":
    nxt.onclick=handler
    // Обработчик для кнопки "Предыдущий":
    prv.onclick=handler
```

```
      // Завершение обработки:
      return
    }
    // Если событие связано с щелчком мышью:
    if(evt.type=="click"){
      // Если событие произошло на кнопке "Следующий":
      if(evt.target==nxt){
        // Применение следующего цвета:
        next()
        // Завершение обработки:
        return
      }
      // Если событие произошло на кнопке "Предыдущий":
      if(evt.target==prv){
        // Применение предыдущего цвета:
        prev()
      }
    }
  }
  // Обработчик для события, связанного
  // с загрузкой документа:
  window.onload=handler
</script>
<!-- Завершение сценария -->
<!-- Описание стилей -->
<style type="text/css">
  /* Стиль кнопок */
  button{
    width: 180px;
    height: 30px;
    font-size: 13pt;
    font-family: Courier New;
    font-weight: bold;
  }
```

```
  /* Стиль блока вывода */
  div{
    width: 360px;
    height: 100px;
    border-style: ridge;
    font: Courier New;
    font-size: 20pt;
    font-weight: bold;
    text-align: center;
    line-height: 100px;
  }
</style>
<!-- Завершение описания стилей -->
</head>
<body>
  <h3>Листинг 8.8</h3><hr>
  <!-- Блок вывода -->
  <div id="mydiv"></div><br>
  <!-- Кнопки -->
  <button id="prev"><< Предыдущий</button>
  <button id="next">Следующий >></button>
</body>
</html>
```

Документ содержит -блок вывода и две кнопки. Кнопки создаются с помощью <button>-блоков. Стиль кнопок и блока вывода задается в <style>-блоке.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

В описании стиля для кнопок (элементов типа button) установлена ширина элемента в 180 пикселей, высота равна 30 пикселям, определен жирный шрифт Courier New размера 13.

Для блока вывода (элемент типа div) ширина равна 360 пикселей, высота составляет 100 пикселей, используется трехмерная ребристая рамка, жирный шрифт Courier New размера 20. Текст вдоль горизонтали выравнивается по центру (инструкция text-align:center),

а инструкцией line-height:100px высота строки установлена равной высоте блока, так что вдоль вертикали текст находится примерно посередине.

Каждый из трех блоков (блок вывода и две кнопки) описан с атрибутом id. Из сценария доступ к элементам осуществляется на основе значения данного атрибута. Сам сценарий начинается с объявления массива colors, состоящего из пяти элементов. Массив предназначен для хранения кодов цвета, который применяется в качестве фонового для блока вывода. Формирование массива происходит при выполнении оператора цикла. При заданном значении индексной переменной k элемент массива colors[k] получает значением "rgb(0,"+(215+k*10)+",0)". Это текстовая строка, определяющая код цвета. Несложно заметить, что мы используем оттенки зеленого цвета.

В сценарии объявлена переменная selected, в которую предполагается записывать индекс элемента из массива colors, который соответствует выбранному цвету. Переменная используется в функции setColor(), вызываемой для применения цвета для фона блока вывода в соответствии с текущим значением переменной selected. В теле функции командой ref=document.getElementById("mydiv") получаем ссылку на объект блока вывода, после чего командами ref.style.background=colors[selected] и ref.innerHTML=colors[selected] соответственно задается цвет фона, и код использованного цвета отображается в блоке вывода.

Кроме функции setColor(), в сценарии используется еще несколько вспомогательных функций. Так, функция next() позволяет выбрать следующий цвет в массиве и применить его для фона блока вывода. Выбор цвета выполняется циклически: если на данный момент выбран цвет, соответствующий последнему элементу в массиве colors, то следующим будет выбран цвет, соответствующий начальному элементу массива. В теле функции командой selected=(selected+1)%colors.length вычисляется новое значение для индекса, после чего вызовом функции setColor() применяется выбранный цвет (и отображается его код).

ДЕТАЛИ

Значение выражения (selected+1)%colors.length — это остаток от деления выражения selected+1 на colors.length. Если текущее значение переменной selected меньше индекса colors.length-1 последнего элемента в массиве, то результат выражения (selected+1)%colors.length совпадает со значением selected+1. Если текущее значение переменой selected совпадает

с индексом colors.length-1 последнего элемента в массиве, то новое значение переменной selected будет равно 0.

Функция для применения предыдущего цвета из массива называется prev(). В ее теле выполняется условный оператор. Если текущее значение переменной selected больше 0, то командой selected-- значение переменной уменьшается на единицу. Если текущее значение переменной selected нулевое, то новое значение переменной равно colors.length-1 (индекс последнего элемента в массиве). После вычисления нового значения переменной selected, благодаря команде setColor(), изменения, связанные с цветом фона, вступают в силу.

Но центральное место в сценарии занимает функция handler(), которая используется в качестве обработчика события, связанного с загрузкой документа, а также событий, связанных со щелчком по любой из кнопок, расположенных в документе. Другими словами, согласно нашему стратегическому плану, одна и та же функция должна вызываться при возникновении разных событий, происходящих на разных элементах. Очевидно, что в таком случае должна существовать возможность уже в процессе выполнения функции каким-то образом классифицировать события по типу и месту. Сделать это можно на основе объекта события, который передается аргументом функции. В данном конкретном случае функция handler() описана с аргументом, который называется evt. Поскольку функция будет использоваться как обработчик событий, то мы можем отождествлять ее аргумент evt с объектом события.

В теле функции командами nxt=document.getElementById("next") и prv=document.getElementById("prev") вычисляются ссылки на объекты кнопок. Далее в игру вступает условный оператор, в котором проверяется условие evt.type=="load". В этом выражении мы используем свойство type объекта события. Значением свойства является текст, определяющий тип события. Для события, связанного с загрузкой, тип события определяется как "load". Поэтому выражение evt.type=="load" истинно, если событие связно с загрузкой ресурса (в данном случае документа).

НА ЗАМЕТКУ

Название для типа события обычно можно угадать по названию соответствующего свойства, через которое реализуется обработчик — отличие в начальной приставке on. Так, для события, связанного со щелчком мышью, событие имеет тип "click", в то время как для обработки данного события предназначено свойство onclick.

Если так, и речь идет об обработке события, связанного с загрузкой (документа), то командой selected=0 присваивается начальное значение переменной selected, вызовом функции setColor() к блоку вывода применяется цвет фона (и выводится код цвета), а также командами nxt.onclick=handler и prv.onclick=handler для объектов кнопок задается обработчик события, связанного со щелчком по кнопке, и этот обработчик — функция handler(). Проще говоря, при вызове функции handler() для обработки события, связанного с загрузкой документа, она сама себя определяет в качестве обработчика для кнопок.

Инструкцию return мы используем для того, чтобы завершить выполнение функции без выполнения прочих команд в ее теле. Поэтому второй условный оператор в теле функции handler() выполняется только в том случае, если условие в первом условном операторе оказалось ложным.

Во втором условном операторе проверяется условие evt.type=="click", истинное в случае, если обрабатывается событие, связанное со щелчком мышью (по кнопке). Но этой информации мало. Необходимо еще знать, на какой именно кнопке произошло событие. Поэтому используются вложенные условные операторы, проверяющие объект (кнопку), на котором произошло событие. Для получения ссылки на объект, на котором произошло событие, используем свойство target объекта события. Так, условие evt.target==nxt истинно в случае, если событие произошло на объекте nxt. Если так, то выполняется команда next(), и процесс завершается инструкцией return. Если приведенное выше условие ложно, то проверяется условие evt.target==prv в следующем условном операторе, и при его истинности вызывается функция prev().

НА ЗАМЕТКУ

Стоит отметить, что в старых версиях браузера Internet Explorer ссылку на объект, на котором произошло событие, можно получить через свойство srcElement элемента события. Последние версии этого браузера поддерживают свойство target.

После описания функции handler() размещена команда window.onload=handler, которой функция регистрируется как обработчик события, связанного с загрузкой документа.

Как выглядит загруженный в браузер документ, показано на рис. 8.28.

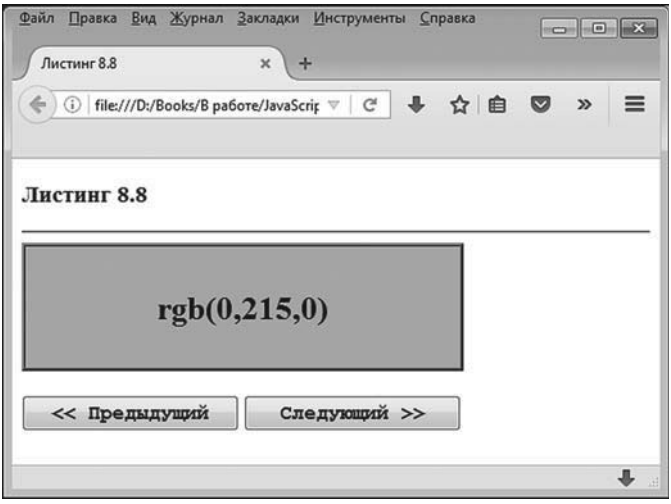

Рис. 8.28. Документ с блоком вывода и двумя кнопками

Последовательные щелчки по кнопке **Следующий** приводят к «просветлению» зеленого фона блока вывода. На рис. 8.29 показана ситуация, когда по кнопке **Следующий** выполнено последовательно три щелчка.

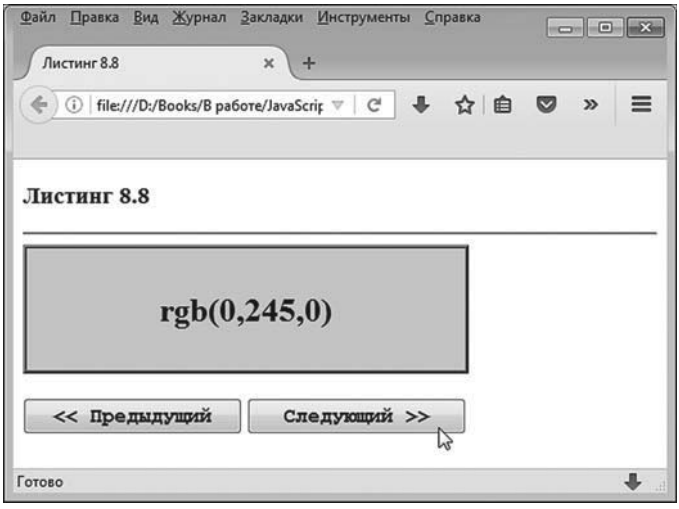

Рис. 8.29. Документ после трех щелчков по кнопке **Следующий**

На рис. 8.30 показано, как выглядит документ после двух щелчков по кнопке **Предыдущий**.

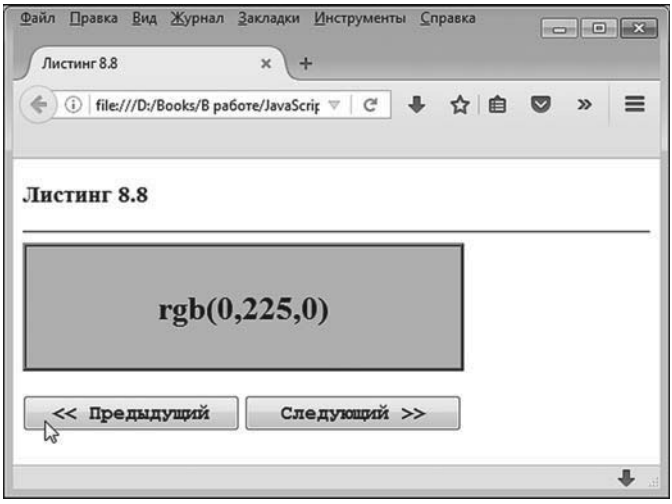

Рис. 8.30. Документ после двух щелчков по кнопке **Предыдущий**

Еще раз подчеркнем, что в данном примере мы использовали одну функцию handler() для обработки события, связанного с загрузкой документа, и событиями, связанными со щелчком по кнопкам в документе.

Непосредственно в сценарии функция регистрируется как обработчик для события, связанного с загрузкой документа. Когда функция вызывается первый раз для обработки данного события, в процессе выполнения кода функции происходит ее регистрация в качестве обработчика для событий, связанных со щелчком по кнопкам. При первом и последующих вызовах функции на основе объекта события производится идентификация типа произошедшего события и, если необходимо, — элемента, на котором оно произошло.

Диспетчеризация событий

Далее мы рассмотрим механизмы перехвата событий в документе. Чтобы легче было понять суть проблемы, мы рассмотрим небольшой пример. Итак, имеется документ, в котором есть три вложенных -блока (имеется в виду, что один -блок содержится в другом и эта конструкция помещена в третий -блок). Для каждого из трех блоков предусмотрен обработчик события, связанного со щелчком кнопкой мыши в области блока. Код документа представлен в листинге 8.9.

Листинг 8.9. Вложенные блоки и обработка событий

```
<!DOCTYPE HTML>
<html>
<head>
<title>Листинг 8.9</title>
<!-- Начало сценария -->
<script type="text/javascript">
  // Обработчик события, связанного с загрузкой документа:
  window.onload=function(){
    // Обработчик для третьего блока:
    document.getElementById("divRed").onclick=function(){
      showColor("Красный")
    }
    // Обработчик для второго блока:
    document.getElementById("divYellow").onclick=function(){
      showColor("Желтый")
    }
    // Обработчик для первого бока:
    document.getElementById("divGreen").onclick=function(){
      showColor("Зеленый")
    }
  }
  // Функция для отображения в текстовом
  // блоке названия цвета:
  function showColor(clr){
    // Текст для добавления в текстовый блок:
    var txt="<b>"+clr+"</b><br>"
    // К текущему содержимому текстового блока
    // дописывается новый текст:
    document.getElementById("result").innerHTML+=txt
  }
</script>
<!-- Завершение сценария -->
</head>
<body>
```

```
  <h3>Листинг 8.9</h3><hr>
  <!-- Первый блок -->
  <div id="divGreen" style="width:200px;height:200px;background-color:green;">
    <!-- Второй блок -->
    <div id="divYellow" style="width:140px;height:140px;background-color:yellow;">
      <!-- Третий блок -->
      <div id="divRed" style="width:80px;height:80px;background-color:red;">
      </div>
    </div>
  </div>
  <!-- Текстовый блок -->
  <p id="result" style="width:170px;height:70px;border-style:ridge;padding:10px;"></p>
</body>
</html>
```

У внешнего -блока (будем называть его первым) зеленый фон. В этом блоке размещен еще один -блок желтого цвета (будем называть этот блок вторым). Второй -блок содержит еще один -блок (назовем его третьим) с фоном красного цвета. Блоки разного размера, поэтому перекрываются только частично. Внизу под первым (самым большим) блоком расположен текстовый блок, выделенный рамкой. На рис. 8.31 показано, как выглядит документ при загрузке.

В сценарии содержится описание функции showColor(). При вызове функции ей передается текстовый аргумент. В соответствии с кодом функции значение аргумента отображается в текстовом блоке. Обработчик события, связанного с загрузкой документа, содержит команды, которыми для каждого из -блоков задаются обработчики события, связанного со щелчком по блоку. Все обработчики однотипные: вызывается функция showColor(), но для каждого блока ей передается свой аргумент. Для первого зеленого блока функции передается текст "Зеленый", для второго желтого блока функции передается тест "Желтый", для третьего красного блока функции передается текст "Красный". Таким образом, можно было бы ожидать, что щелчок мышью в области внутреннего красного блока приведет к отображению в текстовой области значения **Красный**. Реальный результат может быть несколько неожиданным. На рис. 8.32 показано, как будет выглядеть документ, если навести курсор на внутренний красный блок и щелкнуть по нему левой кнопкой мыши.

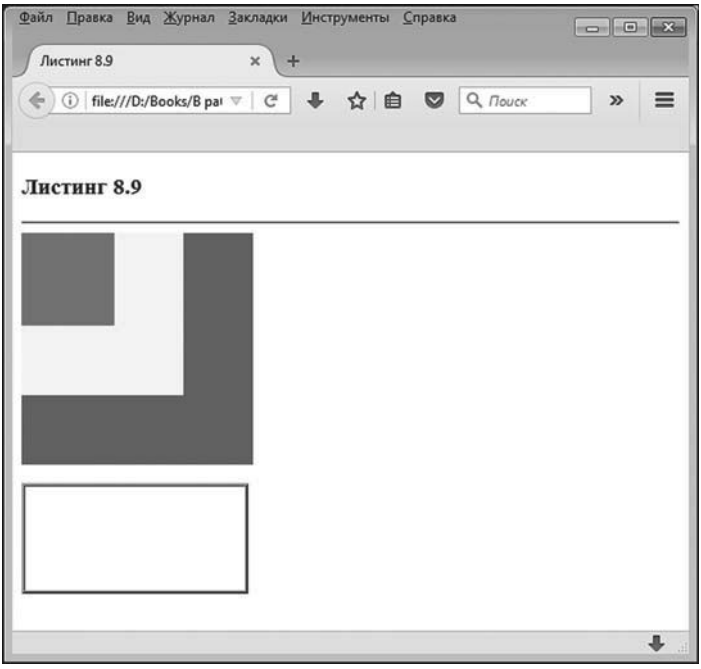

Рис. 8.31. Документ с тремя вложенными блоками

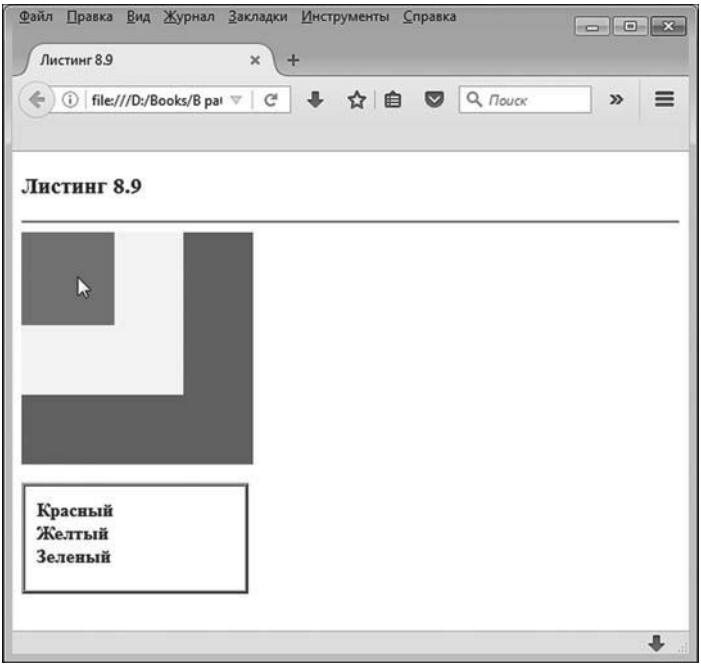

Рис. 8.32. Результат щелчка мышью по внутреннему красному блоку

Видим, что последовательно появились три сообщения (**Красный**, **Желтый** и **Зеленый**). Нечто подобное происходит, если продолжить наши эксперименты. Так, перезагрузим документ и щелкнем по среднему желтому блоку. Результат показан на рис. 8.32.

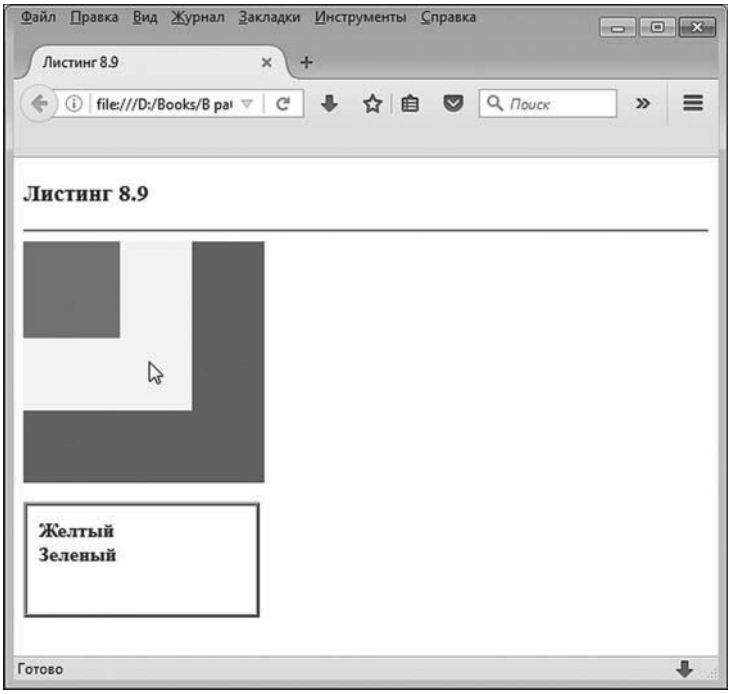

Рис. 8.33. Результат щелчка мышью по среднему желтому блоку

В данном случае появляется два сообщения: **Желтый** и **Зеленый**. Наконец, что будет, если после перезагрузки документа щелкнуть по внешнему зеленому блоку, показано на рис. 8.34.

В этом случае появляется только одно сообщение: **Зеленый**.

НА ЗАМЕТКУ

Если перед щелчком мышью по блоку не произвести перезагрузку документа, то новые сообщения в текстовом блоке будут добавляться к уже существующим.

Объяснение происходящего кроется в способе, которым обрабатываются события. Если некоторое событие происходит на вложенном

компоненте (таком, например, как третий красный блок), то после его обработки обработчиком этого компонента событие, как правило, передается дальше, контейнеру. Если у контейнера имеется обработчик данного типа событий, событие обрабатывается контейнером и снова передается к контейнерному элементу. В нашем случае при щелчке по красному блоку событие обрабатывается обработчиком красного блока и передается для обработки в желтый блок. При обработке события в желтом блоке оно передается дальше в зеленый блок. Поэтому при щелчке по красному блоку появляется три сообщения, а не одно. При щелчке по желтому блоку событие обрабатывается обработчиком желтого блока и передается для обработки в зеленый блок.

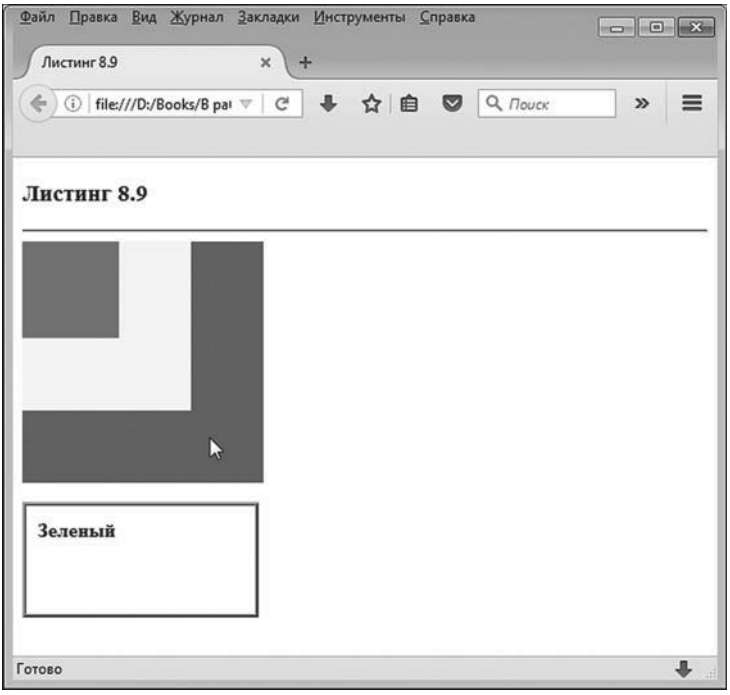

Рис. 8.34. Результат щелчка мышью по внешнему зеленому блоку

По такой схеме по умолчанию обрабатываются многие (но не все) события. Обычно этот процесс называют *всплытием события* — он напоминает то, как в воде всплывают воздушные пузыри.

Казалось бы, это довольно запутанная схема, но на самом деле все происходит еще сложнее. Дело в том, что при обработке событий, когда у нас есть вложенные компоненты, объект события начинает свое

«путешествие» от самого внешнего компонента к вложенному компоненту. Просто на этом «прямом пути» события по умолчанию не обрабатываются. Первая обработка начинается, когда объект события «достигает дна». На нашем примере это выглядит примерно так:

- объект события передается от зеленого блока желтому;
- от желтого блока объект события передается красному;
- в красном блоке событие обрабатывается и передается в желтый блок;
- в желтом блоке событие обрабатывается, и объект события передается в зеленый блок;
- событие обрабатывается в зеленом блоке.

Для нас во всей этой схеме важны два момента. Во-первых, процесс триумфальной передачи объекта события для обработки от компонента к компоненту можно остановить. Для этого из объекта события следует вызвать метод stopPropagation(). Во-вторых, существует возможность перехватывать события на «прямом пути», а не на «обратном», как происходит по умолчанию. Для этого обработчик события следует добавлять с помощью метода addEventListener(), причем последним аргументом метода должно быть указано значение true (первый аргумент — тип события, второй аргумент — ссылка на функцию-обработчик).

Небольшая модификация предыдущего документа представлена в листинге 8.10 (для удобства основные комментарии удалены, а новые важные места кода выделены жирным шрифтом).

Листинг 8.10. Перехват событий

```
<!DOCTYPE HTML>
<html>
<head>
<title>Листинг 8.9</title>
<script type="text/javascript">
  window.onload=function(){
    // В обработчике используется объект события:
    document.getElementById("divRed").onclick=function(evt){
      showColor("Красный")
```

```
      // Блокировка передачи объекта события:
      evt.stopPropagation()
    }
    document.getElementById("divYellow").onclick=function(){
      showColor("Желтый")
    }
    // Перехват события:
    document.getElementById("divGreen").addEventListener("click",function()
    {showColor("Зеленый")},true)
  }
  function showColor(clr){
    var txt="<b>"+clr+"</b><br>"
    document.getElementById("result").innerHTML+=txt
  }
</script>
</head>
<body>
  <h3>Листинг 8.10</h3><hr>
  <div id="divGreen" style="width:200px;height:200px;background-color:green;">
    <div id="divYellow" style="width:140px;height:140px;background-color:yellow;">
      <div id="divRed" style="width:80px;height:80px;background-color:red;">
      </div>
    </div>
  </div>
  <p id="result" style="width:170px;height:70px;border-style:ridge;padding:10px;"></p>
</body>
</html>
```

По сравнению с предыдущим документом здесь есть два принципиальных изменения. Во-первых, обработчик для третьего (красного) блока описан с аргументом (объект события evt), и в тело обработчика добавлена инструкция evt.stopPropagation(). Поэтому после обработки события дальше оно передаваться не будет.

Во-вторых, добавление обработчика для первого (зеленого) блока выполняется командой document.getElementById("divGreen").addEventListener("click",func

tion(){showColor("Зеленый")},true). Важно здесь то, что обработчик добавляется с помощью метода addEventListener(). Первым аргументом методу передается тип события "click". Второй аргумент — анонимная функция, описанная как function(){showColor("Зеленый")}. Она будет вызываться при обработке события. Третий аргумент true означает, что событие следует перехватывать при «прямой передаче». Последствия от внесенных изменений такие. Если в документе щелкнуть по красному блоку, появятся сообщения **Зеленый** и **Красный**, как показано на рис. 8.35.

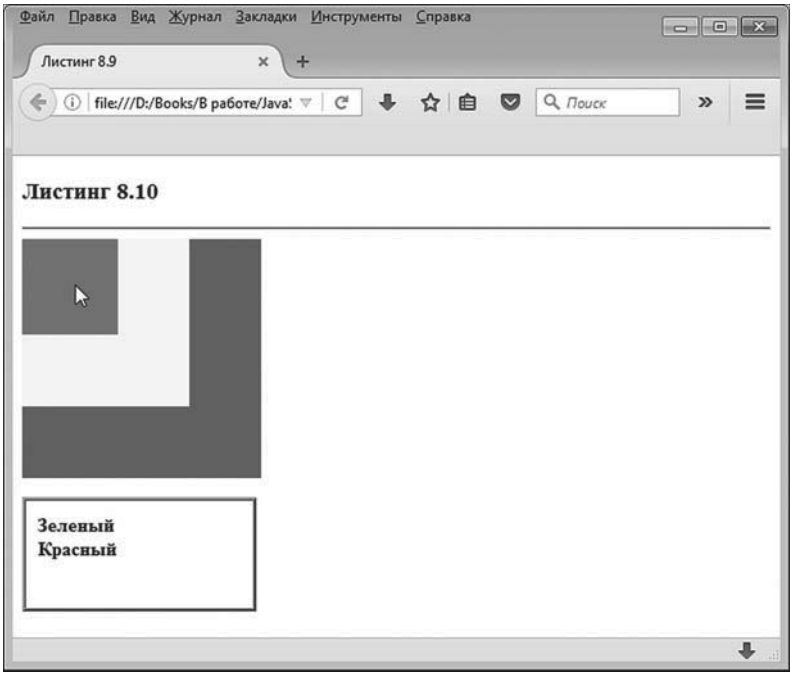

Рис. 8.35. При щелчке по красному блоку появляются сообщения **Зеленый и Красный**

При щелчке (после перезагрузки документа) по желтому блоку появятся сообщения **Зеленый** и **Желтый**, как показано на рис. 8.36.

При щелчке по зеленому блоку появляется сообщение **Зеленый**. Ситуация проиллюстрирована на рис. 8.37.

Во всех рассмотренных случаях важно не только то, что появляется сообщение **Зеленый**, но и то, что оно появляется первым. Объясняется все просто. Например, пользователь выполнил щелчок по красному блоку.

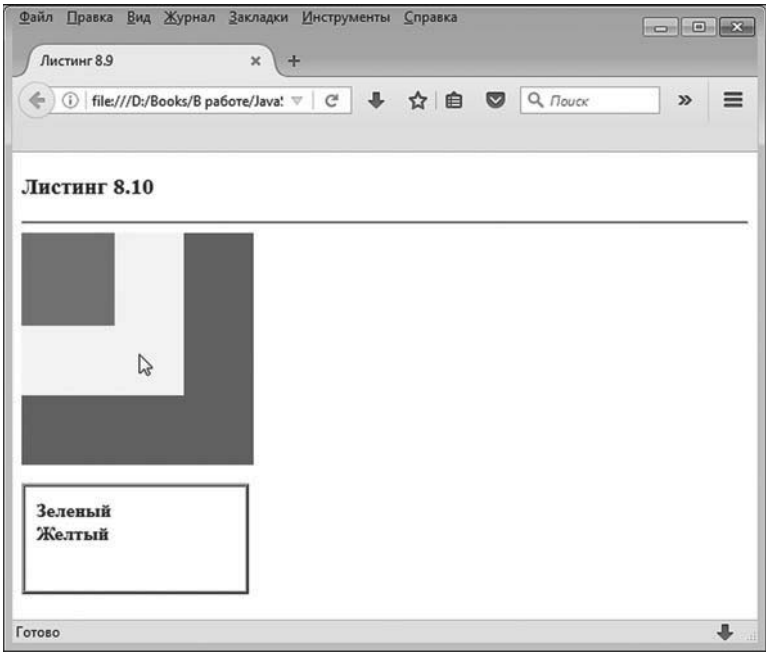

Рис. 8.36. При щелчке по желтому блоку появляются сообщения **Зеленый и Желтый**

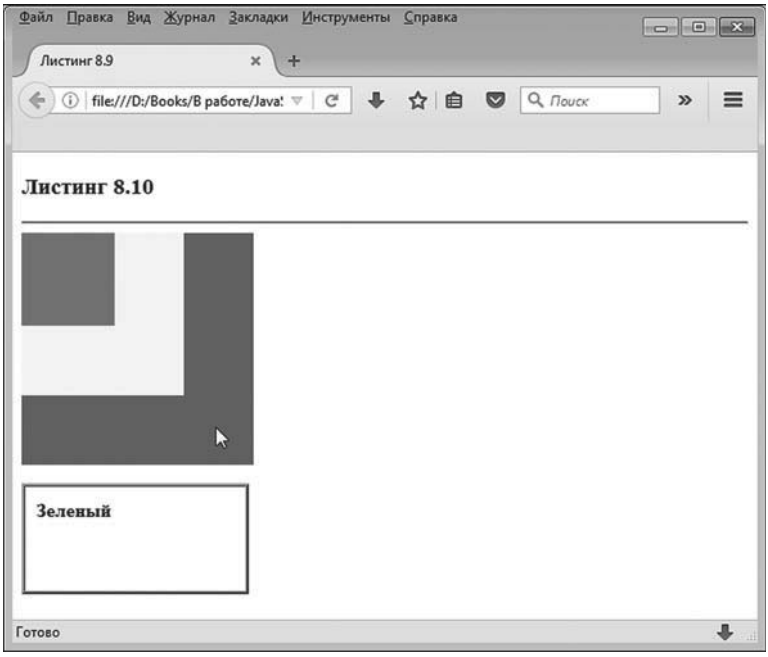

Рис. 8.37. При щелчке по зеленому блоку появляется сообщение **Зеленый**

Соответствующее событие последовательно передается от зеленого блока через желтый в красный блок. Но обработчик зеленого блока описан так, что событие обрабатывается уже на этом этапе, то есть при прямой передаче. Поэтому появляется сообщение **Зеленый**. У желтого блока обработчик описан иначе, поэтому желтый блок при прямой передаче события его не обрабатывает. Событие доходит до красного блока и пытается начать свой обратный путь. Происходит обработка события красным блоком. Появляется сообщение **Красный**. Но поскольку в обработчике красного блока процесс дальнейшей передачи события блокируется, то на этом все и заканчивается.

Если пользователь щелкнул по желтому блоку, то, как и в предыдущем случае, сначала событие обрабатывается зеленым блоком, а затем передается в желтый. Происходит обработка, и событие начинает передаваться в обратном направлении. Но поскольку в зеленом блоке событие уже обработано, то второй раз сообщение **Зеленый** не отображается. Ситуация, когда пользователь выполняет щелчок по зеленому блоку, представляется тривиальной.

Резюме

> Ничего не видел, все люди братья, все должны помогать друг другу.
>
> *из к/ф «Гостья из будущего»*

Наиболее важные моменты, рассмотренные в этой главе, можно сформулировать следующим образом.

- В документе могут использоваться такие элементы управления, как кнопки, поля ввода, опции, переключатели, списки и некоторые другие. С помощью сценариев такие элементы добавляются в документ, задаются их визуальные характеристики и определяются функциональные возможности.
- При работе с элементами управления обычно определяют обработчики событий, на которые должны «реагировать» элементы. Среди наиболее актуальных событий можно выделить связанные с загрузкой документа, щелчком кнопкой мыши, нажатием и отпусканием клавиш, передачей и потерей фокуса, изменением состояния элементов управления и рядом других.

- Объект события передается аргументом функции/методу, обрабатывающим событие. Объект события позволяет получить ряд важных характеристик события, таких, например, как место положения курсора, источник события и тип события.
- При обработке событий следует учитывать механизм передачи объекта событий при наличии вложенных компонентов, на которых происходят события.
- В документе может использоваться и обрабатываться программными средствами графика, включая непосредственное рисование в графической области документа. Существуют специальные методы, которые позволяют отображать графические примитивы в документе и обрабатывать изображения, реализованные в виде графических файлов.

Глава 9
РАЗЛИЧНЫЕ ПРИМЕРЫ

Тихо, кричать не надо. Каждая ваша мысль нам известна.

из к/ф «Гостья из будущего»

В этой главе мы рассмотрим некоторые примеры использования сценариев на языке JavaScript в веб-документах. Примеры различные, и они призваны проиллюстрировать подходы и механизмы, которые обсуждались в предыдущих главах книги. Предполагается, что читатель проявит некоторую самостоятельность в разборе материала главы. Хотя, конечно, наиболее важные и сложные части программных кодов будут прокомментированы.

Триадная кривая Коха

Как говорит наш дорогой шеф, в нашем деле главное — этот самый реализм.

из к/ф «Бриллиантовая рука»

Мы рассмотрим задачу о построении триадной кривой Коха. Алгоритм построения кривой проиллюстрируем с помощью документа, созданного средствами HTML с использованием сценария, написанного на языке JavaScript. После этого проанализируем код документа и сценария.

Итак, сначала берется единичный отрезок, как, например, на рис. 9.1.

Это нулевая итерация в построении кривой. Следующая, первая итерация выполняется так: исходная прямая условно разделяется на три равные части, и на основании центральной трети строится равносторонний треугольник, после чего центральная треть отбрасывается. Что получается в итоге, показано на рис. 9.2.

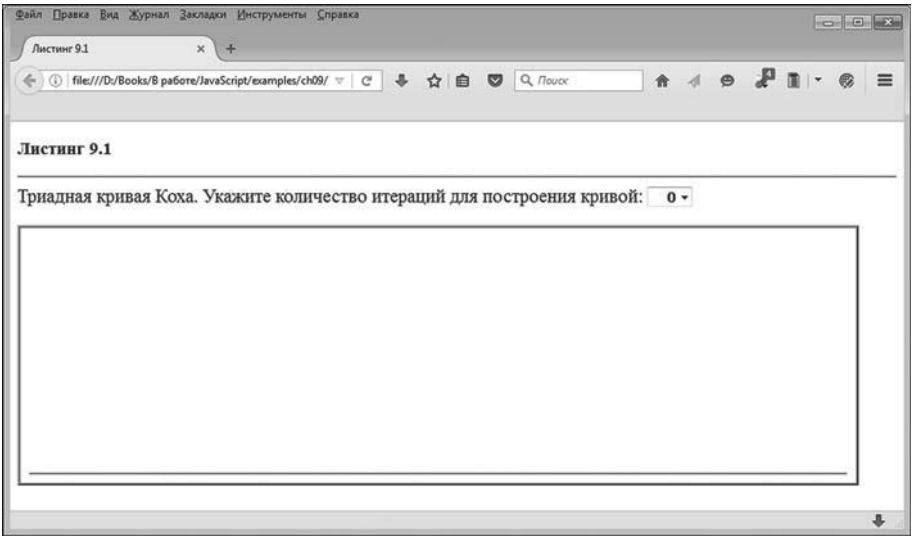

Рис. 9.1. Вид документа при загрузке: в нижней части графической области отображена прямая линия

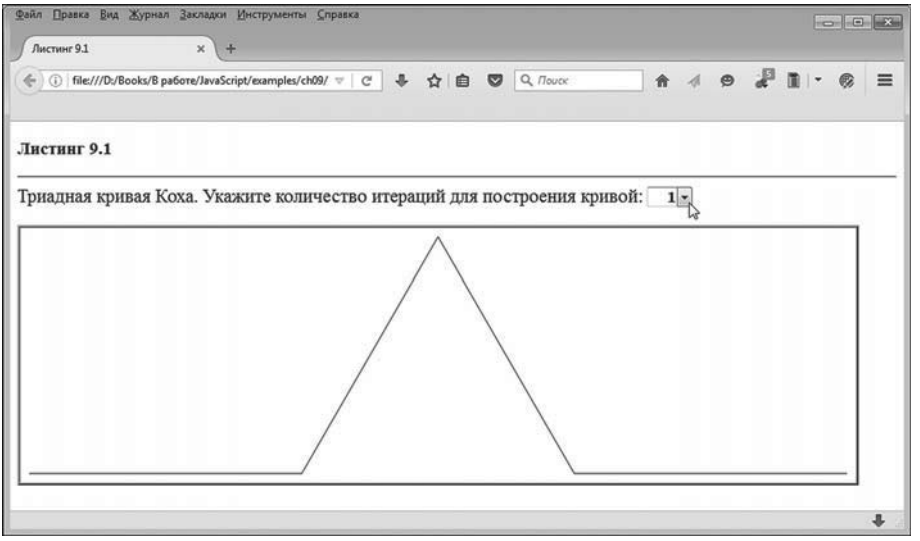

Рис. 9.2. Первая итерация при построении триадной кривой Коха

На следующей итерации каждый из четырех отрезков ломаной кривой разделяется на три равные части. Каждая центральная часть служит основанием для равностороннего треугольника, а сами центральные части отбрасываются. Результат выполнения второй итерации показан на рис. 9.3.

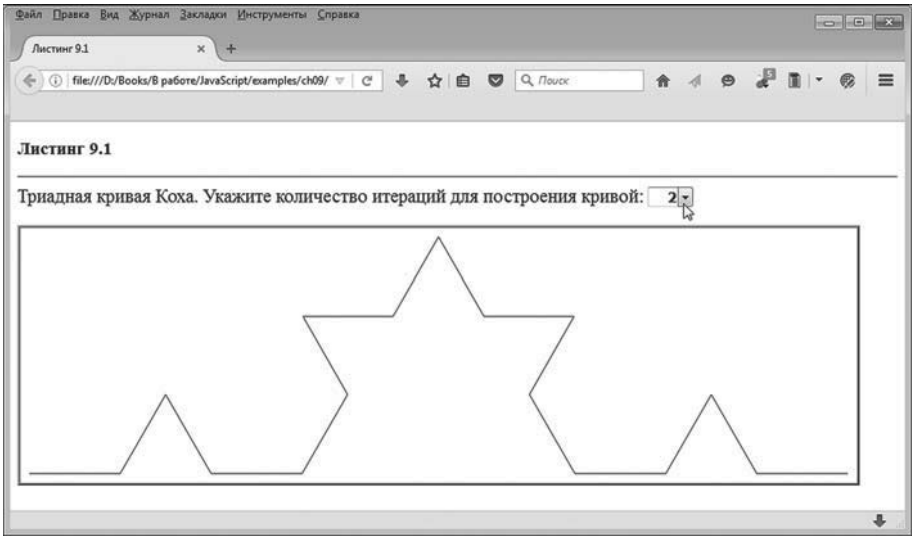

Рис. 9.3. Вторая итерация при построении триадной кривой Коха

Далее действуем по той же схеме: каждый отрезок кривой разбивается на три равные части, на основании центральных частей строятся равносторонние треугольники, а затем центральные части отбрасываются. Результат, который получаем на третьей итерации, показан на рис. 9.4. Как выглядит кривая после четырех итераций, показано на рис. 9.5.

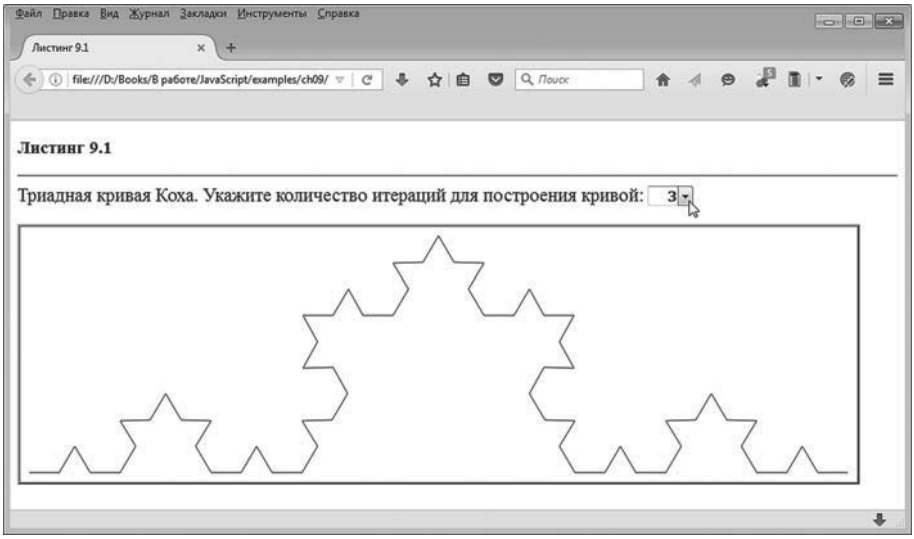

Рис. 9.4. Третья итерация при построении триадной кривой Коха

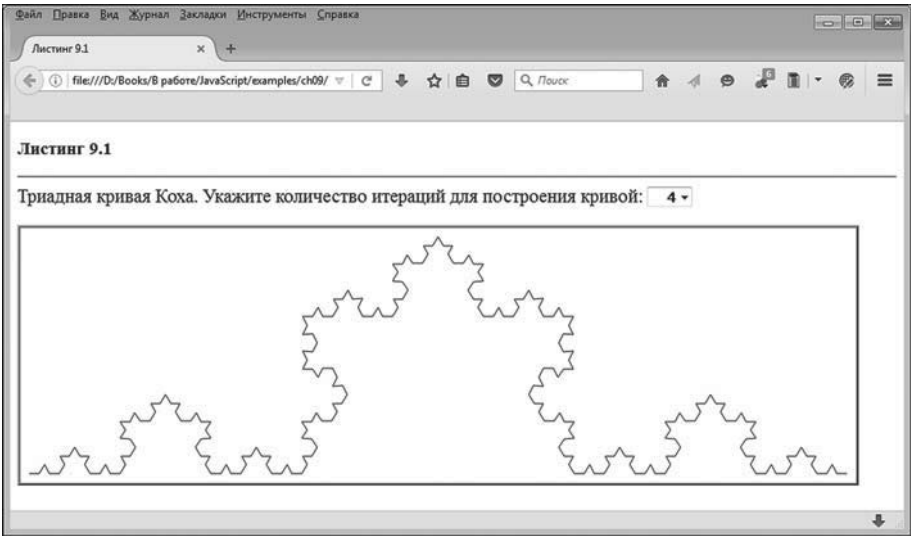

Рис. 9.5. Четвертая итерация при построении триадной кривой Коха

Пятая итерация представлена на рис. 9.6.

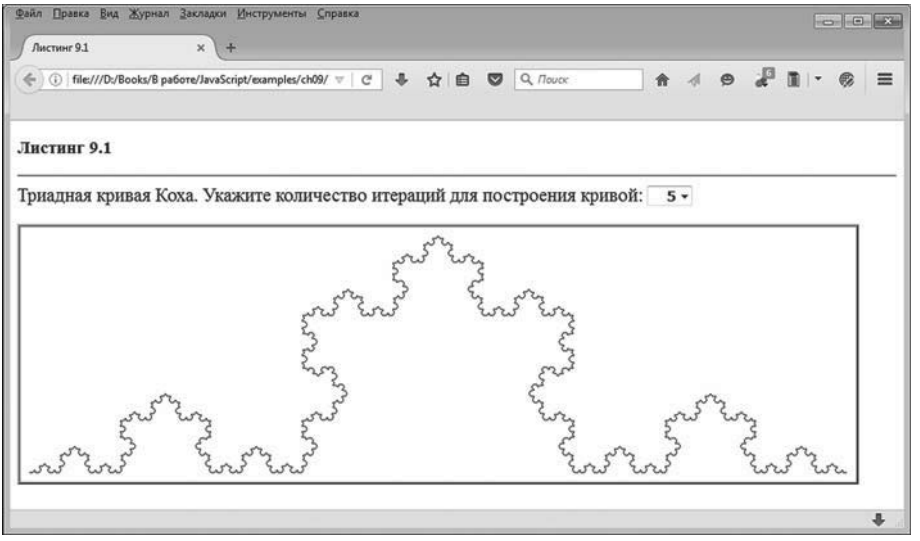

Рис. 9.6. Пятая итерация при построении триадной кривой Коха

Итерации можно продолжать и дальше, но с увеличением порядка итерации размер характерных деталей кривой становится очень маленьким. Вообще же представленный выше документ позволяет

отображать до девяти итераций. Номер итерации выбирается в раскрывающемся списке, расположенном над графической областью.

Проанализируем код документа, позволяющего проследить процесс построения триадной кривой Коха. Интересующий нас код представлен в листинге 9.1.

Листинг 9.1. Триадная кривая Коха

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 9.1</title>
<!-- Начало сценария -->
<script type="text/javascript">
  // Переменные для записи ссылок на графическую область,
  // объект графического контекста и объект
  // раскрывающегося списка:
  var cnv,ctx,sel
  // Функция для вычисления последовательности
  // точек для кривой Коха:
  function getPoints(a,b){
    // Вспомогательные переменные:
    var p,k,i,j,phi
    // Массив с двумя начальными точками:
    var pts=new Array(2)
    // Массивы - элементы:
    for(i=0;i<pts.length;i++){
      // Двумерный массив:
      pts[i]=new Array(2)
    }
    // Значения координат для начальных точек:
    for(i=0;i<2;i++){
      pts[0][i]=a[i]
      pts[1][i]=b[i]
    }
    // Считывание индекса выбранного элемента
```

```
// в раскрывающемся списке:
var n=document.getElementById("mysel").selectedIndex
// Последовательное вычисление точек для формирования
// триадной кривой:
for(k=1;k<=n;k++){
  // Создание нового массива для записи в него
  // координат точек кривой:
  p=new Array(4*pts.length-3)
  // Вычисление точек:
  for(i=0;i<pts.length-1;i++){
    // Создание двумерных массивов:
    for(j=0;j<4;j++){
      p[4*i+j]=new Array(2)
    }
    // Вычисление координат точек:
    for(j=0;j<2;j++){
      p[4*i][j]=pts[i][j]
      p[4*i+1][j]=Math.round(2*pts[i][j]/3+pts[i+1][j]/3)
      p[4*i+3][j]=Math.round(pts[i][j]/3+2*pts[i+1][j]/3)
    }
    // Угол между прямой и осью координат:
    phi=Math.atan2(p[4*i+1][1]-p[4*i+3][1],p[4*i+3][0]-p[4*i+1][0])
    // Длина отрезка:
    L=Math.sqrt(Math.pow(p[4*i+3][0]-p[4*i+1][0],2)+Math.pow(p[4*i+3][1]-p[4*i+1][1],2))
    // Координаты еще одной точки:
    p[4*i+2][0]=Math.round(p[4*i+1][0]+L*Math.cos(phi+Math.PI/3))
    p[4*i+2][1]=Math.round(p[4*i+1][1]-L*Math.sin(phi+Math.PI/3))
  }
  // Последний элемент в массиве точек:
  p[4*pts.length-4]=new Array(2)
  // Координаты точки:
  for(j=0;j<2;j++){
    p[4*pts.length-4][j]=pts[pts.length-1][j]
  }
  // Новая ссылка на массив точек:
```

```
      pts=p
    }
    // Результат функции:
     return pts
  }
  // Функция для отображения кривой в графической области:
  function drawPoints(p,ct){
    // Начало нового контура:
    ct.beginPath()
    // Синий цвет линий:
    ct.strokeStyle="blue"
    // Толщина линий:
    ct.lineWidth=1
    // Начальная точка:
    ct.moveTo(p[0][0],p[0][1])
    // Соединение точек линиями:
    for(var k=1;k<p.length;k++){
      ct.lineTo(p[k][0],p[k][1])
    }
    // Отображение контура:
    ct.stroke()
  }
  // Обработчик события, связанного с загрузкой документа:
  window.onload=function(){
    // Ссылка на графическую область:
    cnv=document.getElementById("mycanvas")
    // Ссылка на объект графического контекста:
    ctx=cnv.getContext("2d")
    // Ссылка на объект раскрывающегося списка:
    sel=document.getElementById("mysel")
    // Переменные:
    var k,Nmax=9
    // Формирование пунктов для раскрывающегося списка:
    for(k=0;k<=Nmax;k++){
      // Добавление нового пункта в список:
```

```
      sel.options[k]=new Option(k,k)
      // Индекс выбранного пункта в списке:
      sel.selectedIndex=0
    }
    // Отображение кривой:
    drawPoints(getPoints([10,cnv.height-10],[cnv.width-10,cnv.height-10]),ctx)
    // Обработчик для события, связанного с изменением
    // состояния раскрывающегося списка:
    sel.onchange=function(){
      // Очистка графической области:
      ctx.clearRect(0,0,cnv.width,cnv.height)
      // Отображение кривой:
      drawPoints(getPoints([10,cnv.height-10],[cnv.width-10,cnv.height-10]),ctx)
    }
  }
</script>
<!-- Завершение сценария -->
</head>
<body>
  <h3>Листинг 9.1</h3><hr>
  <!-- Текст -->
  <span style="font-size:16pt;">Триадная кривая Коха. Укажите количество итераций для
построения кривой:</span>
  <!-- Раскрывающийся список -->
  <select id="mysel" size="1" style="width:50px;text-align:right;font-size:11pt;font-
weight:bold;"></select><br><br>
  <!-- Графическая область -->
  <canvas id="mycanvas" width="920" height="280" style="border-style:ridge;"></canvas>
</body>
</html>
```

В теле документа описан блок с текстом, блок раскрывающегося списка (без внутренних <option>-блоков), а также <canvas>-блок, в котором происходит отображение кривой. Для раскрывающегося списка и графической области заданы размеры. Список описан со значением "mysel" для атрибута id, а у графической области значение атрибу-

та id равно "mycanvas". В сценарии описано несколько вспомогательных функций и обработчик события, связанного с загрузкой документа (в котором, кроме прочего, описан обработчик события, связанного с изменением состояния списка).

В самом начале сценария объявляются три переменные cnv, ctx, sel, в которые значениями записываем ссылки соответственно на объект графической области, объект графического контекста и объект раскрывающегося списка. Происходит это при вызове обработчика, связанного сзагрузкойдокумента(командыcnv=document.getElementById("mycanvas"),ctx=cnv.getContext("2d") и sel=document.getElementById("mysel")). После получения ссылок начинается формирование пунктов раскрывающегося списка.

Пункты списка — это числа, определяющие количество итераций, выполняемых при построении кривой Коха. Максимальное количество итераций определяется значением переменной Nmax (значение переменной равно 9).

Пункты раскрывающегося списка создаются в теле оператора цикла. Индексная переменная k пробегает значения от 0 до Nmax включительно, и за каждый цикл командой sel.options[k]=new Option(k,k) в раскрывающийся список добавляется новый пункт. По завершении оператора цикла командой sel.selectedIndex=0 в раскрывающемся списке выбирается пункт с нулевым индексом.

На данном этапе получены ссылки на объекты графической области, графического контекста и раскрывающегося списка, а также сформирован сам список и в нем выбран начальный пункт. Следующее действие — построение кривой Коха (в соответствии с заданным количеством итераций, определяемых выбранным пунктом в раскрывающемся списке). Выполняется это командой drawPoints(getPoints([10,cnv.height-10],[cnv.width-10,cnv.height-10]),ctx), в которой мы вызываем описанную в сценарии функцию drawPoints(). Функция отображает кривую по точкам, которые переданы в виде массива первым аргументом функции. Вторым аргументом функции должна быть ссылка на объект графического контекста, с использованием которого выполняется построение. В данном случае вторым аргументом передается ссылка на объект графического контекста единственной графической области в документе. Первым аргументом функции drawPoints() передается инструкция getPoints([10,cnv.height-10],[cnv.width-10,cnv.height-10]). Это вызов еще одной описанной в сценарии функции getPoints(), аргументом которой передается два массива [10,cnv.height-10] и [cnv.width-10,cnv.height-10] (по два

элемента в каждом массиве). Функция getPoints() определена таким образом, что возвращает набор точек, которые формируют кривую Коха. Аргументами функции передаются две точки, формирующие начальный отрезок, на основе которого строится кривая. Количество итераций, выполняемых при построении кривой, определяется в процессе вызова функции getPoints() на основании значения, выбранного в раскрывающемся списке.

В описании обработчика для события, связанного с изменением состояния раскрывающегося списка, командой ctx.clearRect(0,0,cnv.width,cnv.height) выполняется очистка графической области, после чего командой drawPoints(getPoints([10,cnv.height-10],[cnv.width-10,cnv.height-10]),ctx) отображается кривая Коха.

Что касается функции drawPoints(), используемой для отображения кривой по набору точек, то ее код достаточно простой. Аргументы функции обозначены как p (массив точек) и ct (ссылка на объект графического контекста). Начало нового контура инициализируется командой ct.beginPath(). Командами ct.strokeStyle="blue" и ct.lineWidth=1 задается синий цвет для линии и ее толщина, а командой ct.moveTo(p[0][0],p[0][1]) задается начальная точка для построения кривой. Затем с помощью оператора цикла точки последовательно соединяются линиями (команда ct.lineTo(p[k][0],p[k][1]) при заданном значении индекса k). Отображается кривая командой ct.stroke().

Однако центральное место сценария, без сомнения, занимает функция getPoints(), с помощью которой реализуются все наиболее важные вычисления.

Как отмечалось выше, аргументами функции передаются две точки (обозначены как a и b), определяющие отрезок, на основе которого строится кривая. Сначала кратко остановимся на общем алгоритме, а затем проследим, как он реализуется в программном коде.

Допустим, что мы получили последовательность точек, определяющих кривую Коха, на какой-то определенной итерации. Далее нам необходимо выполнить следующую итерацию. Процедура выполнения новой итерации сводится к добавлению новых точек к той последовательности, которая получена для текущей итерации. Принимая во внимание процесс построения кривой, легко понять, что между каждыми двумя точками для текущей итерации добавляется по три точки (рис. 9.7).

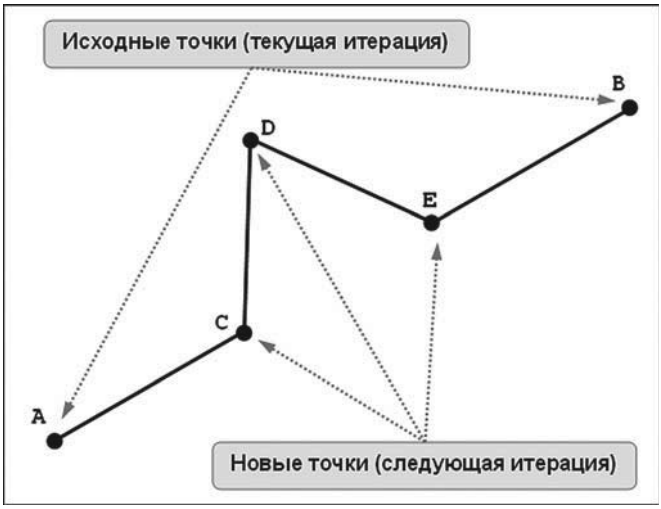

Рис. 9.7. Добавляемые при выполнении новой итерации точки

Если для текущей итерации соседними являются точки **A** и **B** (см. рис. 9.7), то на следующей итерации между ними появляются точки **C**, **D** и **E**. Поэтому если на данной итерации имеется m точек, то на следующей итерации будет $4m - 3$ точек.

Объяснение простое: кривая, соединяющая m точек, содержит $m - 1$ отрезков, на каждом из которых появляется по 3 точки.

Таким образом, появляется $3(m - 1)$ новых точек. Если к ним добавить еще m уже существующих точек, получаем всего $4m - 3$ точки.

Наша задача сводится к тому, чтобы вычислить координаты новых добавляемых точек (**C**, **D** и **E**) на основании координат исходных точек **A** и **B**. Проще всего вычислить координаты точек **C** и **E**. Они делят отрезок **AB** на три равные части (рис. 9.8).

Если точка **A** имеет координаты (x_A, y_A), а точка **B** имеет координаты (x_B, y_B), то горизонтальные координаты x_C и x_E точек **C** и **E** делят отрезок от x_A до x_B на три равные части. Сходная ситуация имеет место и для вертикальных координат. Поэтому справедливы соотношения $x_C = x_A + \frac{1}{3}(x_B - x_A) = \frac{2}{3}x_A + \frac{1}{3}x_B$, $x_E = x_A + \frac{2}{3}(x_B - x_A) = \frac{1}{3}x_A + \frac{2}{3}x_B$. Аналогичные соображения позволяют записать $y_C = \frac{2}{3}y_A + \frac{1}{3}y_B$ и $y_E = \frac{1}{3}y_A + \frac{2}{3}y_B$.

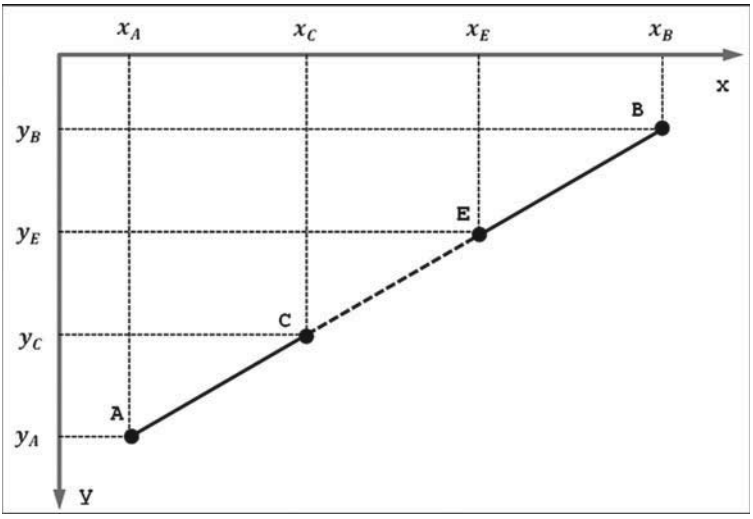

Рис. 9.8. Определение координат двух новых точек

Немного сложнее вычисляются координаты центральной точки **D** (см. рис. 9.7). Если координаты точек **C** и **E** уже известны, то координаты точки **D** можно вычислить на основе координат этих точек. Соответствующие геометрические построения показаны на рис. 9.9.

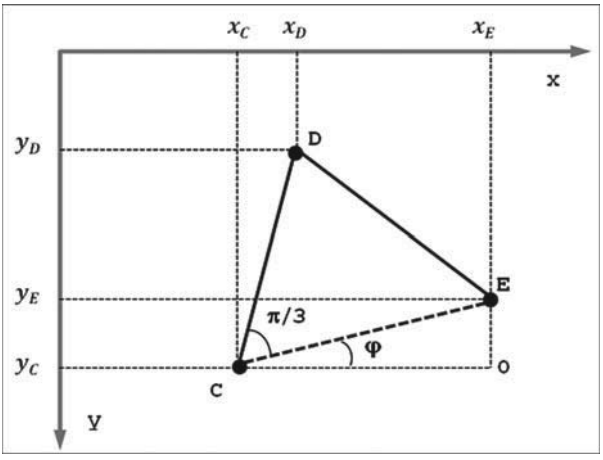

Рис. 9.9. Определение координаты новой центральной точки

Если через L обозначить длину ребра равностороннего треугольника **CDE**, то из приведенных построений (см. рис. 9.9) видно, что

горизонтальная координата точки **D** может быть вычислена как $y_D = y_C + L \cdot \cos(\varphi + \pi/3)$, а вертикальная координата вычисляется как $y_D = y_C - L \cdot \sin(\varphi + \pi/3)$. Через φ обозначен угол **ECO**, для которого имеет место соотношение $\tan(\varphi) = (y_C - y_E)/(x_E - x_C)$. Длина ребра треугольника вычисляется как $L = \sqrt{(x_E - x_C)^2 + (y_E - y_C)^2}$.

Приведенные выше соотношения используются в сценарии при вычислении координат точек для кривой Коха. В частности, в теле функции getPoints() на основе аргументов a и b, определяющих координаты двух начальных точек, формируется массив pts, состоящий из двух элементов (две начальные точки). Каждый из этих элементов является, в свою очередь, массивом из двух элементов (у каждой точки по две координаты). Соответствующие массивы создаются с помощью оператора цикла, в теле которого выполняется команда pts[i]=new Array(2). После создания массивов они заполняются. Для этого запускается еще один оператор цикла (индексная переменная i принимает значения 0 и 1), в теле которого командами pts[0][i]=a[i] и pts[1][i]=b[i] определяются координаты начальных точек. Это точки для нулевой итерации. Количество выполняемых итераций определяется командой n=document.getElementById("mysel").selectedIndex. В данном случае в переменную n записывается индекс выбранного в раскрывающемся списке пункта меню.

НА ЗАМЕТКУ

Так получается, что количество итераций совпадает с индексом выбранного пункта меню. Это же значение равно отображаемому в выбранном пункте числу и равно значению свойства value объекта раскрывающегося списка.

Для выполнения итераций в построении набора точек для кривой Коха запускается оператор цикла, в котором количество циклов определяется значением переменной n, а индексная переменная k принимает значения от 1 до n включительно. За каждый цикл командой p=new Array(4*pts.length-3) создается массив, в который будут заноситься координаты точек для новой итерации. Здесь следует учесть, что в конце каждого цикла выполняется команда pts=p, поэтому pts — это фактически массив точек с предыдущей итерации. После того как массив для новых точек создан, запускается внутренний оператор цикла, в котором индексная переменная i перебирает значения индексов эле-

ментов из массива pts. При этом для каждого фиксированного значения переменной i вычисляются координаты для четырех точек в массиве p. Соответствие здесь такое: если точка имеет индекс i в массиве pts, то в массиве p эта точка будет иметь индекс 4*i. Здесь имеет место простое копирование значений. Координаты точек с индексами 4*i+1, 4*i+2 и 4*i+3 вычисляются. Но перед копированием и вычислением координат необходимо создать соответствующие элементы (массивы из двух элементов). Для этого запускается еще один внутренний оператор цикла (индексная переменная j пробегает значения от 0 до 3), в котором командой p[4*i+j]=new Array(2) создаются массивы. В следующем операторе цикла (переменная j принимает значения 0 и 1) командами p[4*i][j]=pts[i][j], p[4*i+1][j]=Math.round(2*pts[i][j]/3+pts[i+1][j]/3) и p[4*i+3][j]=Math.round(pts[i][j]/3+2*pts[i+1][j]/3) определяются координаты для трех точек. При вычислениях мы использовали математический метод round() для округления чисел, поскольку графические координаты должны быть целочисленными.

Остается вычислить координаты еще одной точки, для чего командой phi=Math.atan2(p[4*i+1][1]-p[4*i+3][1],p[4*i+3][0]-p[4*i+1][0]) рассчитывается угол φ, а командой L=Math.sqrt(Math.pow(p[4*i+3][0]-p[4*i+1][0],2)+Math.pow(p[4*i+3][1]-p[4*i+1][1],2)) вычисляется длина ребра L. После этого командами p[4*i+2][0]=Math.round(p[4*i+1][0]+L*Math.cos(phi+Math.PI/3)) и p[4*i+2][1]=Math.round(p[4*i+1][1]-L*Math.sin(phi+Math.PI/3)) вычисляются координаты точки.

ДЕТАЛИ

В процессе вычислений мы использовали некоторые математические методы: sqrt() для вычисления квадратного корня, pow() для возведения в степень (показатель степени — второй аргумент метода), sin() для вычисления синуса, cos() для вычисления косинуса. Также при определении угла φ использовали метод atan2(). Значением выражения Math.atan2(y,x) возвращается угол между горизонтальной координатной осью и лучом, направленным на точку с координатами x и y.

После завершения оператора цикла (с индексной переменной i) незаполненным остается еще один, последний элемент в массиве p. Поэтому командой p[4*pts.length-4]=new Array(2) значением этому элементу присваивается ссылка на вновь созданный массив из двух элементов. Значения элементам этого внутреннего массива присваиваются с помощью оператора цикла (индексная переменная j принимает значения 0 и 1) командой p[4*pts.length-4][j]=pts[pts.length-1][j].

После того как массив p сформирован и заполнен, командой pts=p ссылка на него записывается в переменную pst. Все это, напомним, происходит на каждой итерации в процессе построения точек для триадной кривой. По завершении всех итераций командой return pts ссылка на созданный массив pts возвращается результатом функции getPoints().

Калькулятор

Ну, у каждого, знаете ли, свой крест!

из к/ф «Покровские ворота»

Следующий пример, который мы рассмотрим, связан с созданием калькулятора. Идея состоит в том, чтобы отобразить в рабочем документе калькулятор, с кнопками и полем отображения результатов вычислений. Мы будем рассматривать наиболее простой случай, когда калькулятор умеет выполнять только основные арифметические операции: сложение, вычитание, умножение и деление. Как именно выглядит калькулятор в области документа, показано на рис. 9.10.

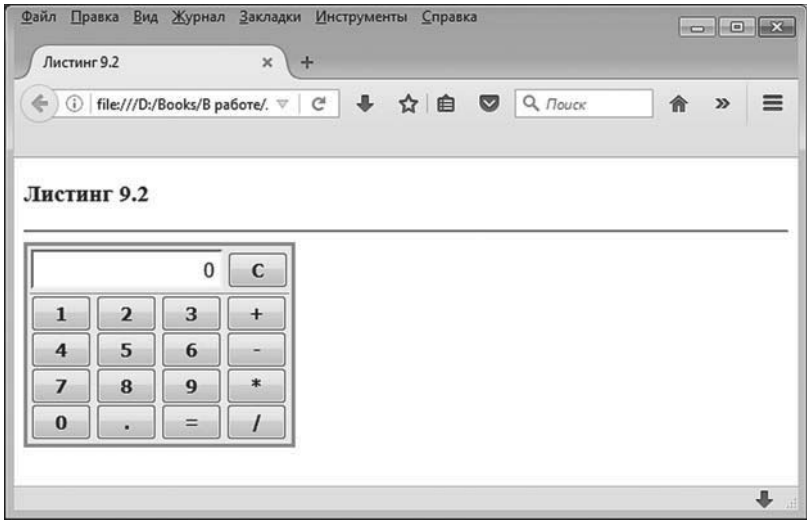

Рис. 9.10. Документ с калькулятором

Калькулятор оформлен в виде небольшой прямоугольной области серого цвета. В верхней части находится текстовое поле и кнопка

сброса (кнопка с литерой **C**). Снизу, под тонкой серой линией, расположено четыре ряда кнопок, по четыре кнопки в каждом ряду. Там представлены кнопки для ввода цифр от **0** до **9** включительно, кнопка для ввода десятичной точки (точка, разделяющая целую и дробную части числа), а также кнопки для ввода операторов **+** (сложение), **-** (вычитание), ***** (умножение), / (деление) и **=** (равенство). Код, с помощью которого реализован документ с калькулятором, представлен в листинге 9.2.

Листинг 9.2. Калькулятор

```
<!DOCTYPE HTML>
<html>
<head>
<title>Листинг 9.2</title>
<!-- Начало сценария -->
<script type="text/javascript">
  // Ссылка на текстовое поле:
  var output
  // Переменные для записи первого операнда и символа
  // оператора:
  var operand,operator
  // Переменные для идентификации режима ввода числа и
  // десятичной точки:
  var numState,pntState
  // Массив с символами операторов:
  var ops=["+","-","*","/","="]
  // Ссылка на кнопку сброса:
  var btn
  // Функция вызывается при щелчке на кнопке с цифрой:
  function btnPress(k){
    // Если вводится число, но еще ни одна цифра
    // не введена:
    if(numState){
      // Значение в текстовом поле:
      output.value=k
```

```
      // Изменение значения переменной, определяющей
      // режим ввода числа:
      numState=false
    }
    // Если при вводе числа цифры уже вводились:
    else{
      // Если в текстовом поле нулевое значение:
      if(output.value=="0"){
        // Новое значение в текстовом поле:
        output.value=k
      }
      // Если в текстовом поле уже есть цифры:
      else{
        // Цифра дописывается в конец текущего
        // значения в текстовом поле:
        output.value+=k
      }
    }
  }
  // Обработка события, связанного с загрузкой документа:
  window.onload=function(){
    // Ссылка на кнопку сброса:
    btn=document.getElementById("btnC")
    // Ссылка на текстовое поле:
    output=document.getElementById("myoutput")
    // Текстовое поле заблокировано:
    output.disabled=true
    // Обработчик события, связанного с щелчком
    // на кнопке сброса:
    btn.onclick=function(){
      // Обнуление значения в текстовом поле:
      output.value="0"
      // Значение переменной, определяющей режим
      // ввода десятичной точки:
```

```
  pntState=true
  // Значение переменной, определяющей режим
  // ввода числа:
  numState=true
  // Начальное значение переменной, определяющей
  // операцию:
  operator=""
  // Начальное значение для первого операнда:
  operand=""
}
// Обработчик события, связанного с щелчком
// на кнопке ввода десятичной точки:
document.getElementById("btnDot").onclick=function(){
  // Если разрешен ввод десятичной точки:
  if(pntState){
    // Если при вводе числа цифры еще не вводились:
    if(numState){
      // Точка добавляется вместе с нулем:
      output.value="0."
      // Изменение значения переменной,
      // определяющей режим ввода числа:
      numState=false
    }
    // Если при вводе числа цифры уже вводились:
    else{
      // Добавление точки к текущему представлению
      // числа:
      output.value+="."
    }
    // Новое значение для переменной, определяющей
    // режим ввода точки (запрещение ввода точки):
    pntState=false
  }
}
```

```
// Обработчики событий, связанных с щелчком на
// кнопках для ввода цифр:
for(var k=0;k<=9;k++){
  document.getElementById("btn"+k).onclick=function(){
    // Вызов функции btnPress() с аргументом,
    // значение которого определяется свойством
    // value соответствующей кнопки:
    btnPress(this.value)
  }
}
// Обработчики для кнопок ввода символа оператора:
for(var p=0;p<ops.length;p++){
  document.getElementById("btn"+ops[p]).onclick=function(){
    // Если первый операнд существует (не пустая
    // строка) и если это не текст "Ошибка!":
    if(operand!=""&&operand!="Ошибка!"){
      // Результат вычисления выражения:
      var res=eval(operand+operator+output.value)
      // Если произошла ошибка:
      if(res=="Infinity"||res=="-Infinity"||isNaN(res)){
        // Символ оператора:
        operator=""
        // Первый операнд:
        operand=""
        // Значение в текстовом поле:
        output.value="Ошибка!"
      }
      // Если ошибки нет:
      else{
        // В текстовом поле отображается
        // результат вычислений:
        output.value=res
        // Если щелчок на кнопке ввода
        // оператора равенства:
```

```
      if(this.value=="="){
        // Значение первого операнда:
        operand=""
      }
      // Если щелчок не на кнопке ввода
      // оператора равенства:
      else{
        // Вычисленное значение становится
        // первым операндом:
        operand=res
        // Новое значение для символа
        // оператора:
        operator=this.value
      }
    }
  }
  // Если первый операнд не существует (пустая
  // строка) или если это текст "Ошибка!":
  else{
    // Если была нажата не кнопка для ввода
    // оператора равенства:
    if(this.value!="="){
      // Новое значение для символа оператора:
      operator=this.value
      // Содержимое текстового поля становится
      // значением первого операнда:
      operand=output.value
    }
  }
  // Режим ввода нового числа:
  numState=true
  // Разрешено введение десятичной точки:
  pntState=true
}
```

```
}
// Обработка события формы, связанного с нажатием
// клавиши для ввода символа
// (цифры, точки или символа оператора):
document.getElementById("mycalc").onkeypress=function(evt){
  // Индексная переменная:
  var k
  // Массив для записи символов цифр:
  var nums=new Array(10)
  // Заполнение массива:
  for(k=0;k<nums.length;k++){
    // Значение элемента массива:
    nums[k]=""+k
  }
  // Определение символа по коду нажатой клавиши:
  var symb=String.fromCharCode(evt.which)
  // Если нажата клавиша с цифрой:
  if(symb in nums){
    // Вызов функции для обработки нажатия
    // кнопки с цифрой:
    btnPress(symb)
  }
  // Если нажата не клавиша с цифрой:
  else{
    // Если вводится точка:
    if(symb=="."){
      // Программный щелчок на кнопке ввода точки:
      document.getElementById("btnDot").click()
    }
    // Если вводится не точка:
    else{
      // Перебор символов операторов:
      for(k=0;k<ops.length;k++){
        // Если вводится символ оператора:
```

```
            if(symb==ops[k]){
              // Программный щелчок на кнопке ввода
              // соответствующего оператора:
              document.getElementById("btn"+symb).click()
              // Завершение всех операций
              // по обработке события:
              return
            }
          }
        }
      }
    } // Завершение обработчика события, связанного
      // с нажатием кнопки.
    // Программный щелчок на кнопке сброса:
    btn.click()
  }
</script>
<!-- Завершение сценария -->
<!-- Описание стилей -->
<style type="text/css">
  /* Стиль для таблицы в форме mycalc */
  #mycalc table{
    border-style: solid;
    border-width: 3px;
    border-color: #a0a0a0;
    background-color: #f0f0f0;
  }
  /* Стиль для кнопок в форме mycalc */
  #mycalc input[type="button"]{
    font-size: 15px;
    font-weight: bold;
    height: 30px;
    width: 50px;
  }
```

```
  /* Стиль для текстового поля в форме mycalc */
  #mycalc input[type="text"]{
    height: 25px;
    width: 145px;
    background-color: white;
    color: black;
    font-size: 16px;
    text-align: right;
    padding: 2px 5px;
  }
</style>
<!-- Завершение описания стилей -->
</head>
<body>
<h3>Листинг 9.2</h3><hr>
<!-- Форма для реализации калькулятора -->
<form id="mycalc">
<!-- Таблица в форме -->
<table>
  <!-- Первая строка таблицы -->
  <tr>
    <!-- Первый столбец в первой строке -->
    <td>
      <!-- Текстовое поле -->
      <input type="text" id="myoutput">
      <!-- Кнопка сброса -->
      <input type="button" value="C" id="btnC">
    </td>
  </tr>
  <!-- Вторая строка таблицы ->
  <tr>
    <!-- Первый столбец во второй строке -->
    <td style="border-top:solid 1px #a0a0a0;">
      <!-- Кнопка с цифрой 1 -->
```

```
<input type="button" value="1" id="btn1">
<!-- Кнопка с цифрой 2 -->
<input type="button" value="2" id="btn2">
<!-- Кнопка с цифрой 3 -->
<input type="button" value="3" id="btn3">
<!-- Кнопка с оператором сложения -->
<input type="button" value="+" id="btn+">
<br>
<!-- Кнопка с цифрой 4 -->
<input type="button" value="4" id="btn4">
<!-- Кнопка с цифрой 5 -->
<input type="button" value="5" id="btn5">
<!-- Кнопка с цифрой 6 -->
<input type="button" value="6" id="btn6">
<!-- Кнопка с оператором вычитания -->
<input type="button" value="-" id="btn-">
<br>
<!-- Кнопка с цифрой 7 -->
<input type="button" value="7" id="btn7">
<!-- Кнопка с цифрой 8 -->
<input type="button" value="8" id="btn8">
<!-- Кнопка с цифрой 9 -->
<input type="button" value="9" id="btn9">
<!-- Кнопка с оператором умножения -->
<input type="button" value="*" id="btn*">
<br>
 <!-- Кнопка с цифрой 0 -->
<input type="button" value="0" id="btn0">
<!-- Кнопка с десятичной точкой -->
<input type="button" value="." id="btnDot">
<!-- Кнопка с оператором равенства -->
<input type="button" value="=" id="btn=">
<!-- Кнопка с оператором деления -->
<input type="button" value="/" id="btn/">
```

```
    </td>
  </tr>
</table>
</form>
</body>
</html>
```

Здесь мы имеем HTML-код, с помощью которого реализуется интерфейс калькулятора, а также собственно программный код сценария, определяющего функциональные возможности калькулятора. Сначала кратко рассмотрим особенности HTML-код документа. Нас будут интересовать инструкции в <body>-блоке.

Для создания калькулятора мы используем форму (элемент, который описывается с помощью дескрипторов <frame> и </frame>). Большой необходимости использовать в данном случае форму нет. Но как элемент, предназначенный для размещения элементов управления, она в рассматриваемом примере вполне уместна.

Форма описана со значением "mycalc" атрибута id. Форма содержит таблицу, которая состоит из двух строк, и в каждой строке находится всего по одному столбцу. В первой строке расположено текстовое поле и кнопка сброса. И поле, и кнопка реализованы через <input>-блоки, просто для поля значение атрибута type равно "text", а для кнопки значение этого атрибута равно "button". Поле описано со значением "myoutput" для атрибута id, а значение атрибута id для кнопки сброса равно "btnC".

Вторая строка таблицы содержит столбец, в котором размещено 16 кнопок. Однако размещены они в четыре ряда. Для перехода к новой строке при отображении кнопок использована инструкция
. В частности, там представлены кнопки для цифр от 0 до 9, для операторов +, -, *, /, = и десятичной точки .. Каждая кнопка описана в виде <input>-блока со значением "button" для атрибута type. Для кнопок с цифрами значением атрибута value указано текстовое представление для соответствующей цифры. Для кнопок с символами операторов значение атрибута value совпадает с символом оператора. Значение атрибута id для всех упомянутых кнопок получается дописыванием значения атрибута value к тексту "btn". Кнопка для ввода десятичной точки описана со значением "." для атрибута value, а значение атрибута id для этой кнопки равно "btnDot".

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Инструкция style="border-top:solid 1px #a0a0a0;" в описании <td>-блока во второй строке таблицы означает, что верхняя граница столбца отображается в виде тонкой (толщина в 1 пиксель) сплошной линии серого цвета (код цвета #a0a0a0). Именно благодаря этой инструкции под текстовым полем и кнопкой отмены отображается тонкая линия (см. рис. 9.10).

Часть параметров, связанных с оформлением интерфейса калькулятора, определяется в <style>-блоке. Блок содержит три секции, в которых описываются стили для таблицы в форме, кнопок в форме и текстового поля в форме (речь идет о форме со значением "mycalc" для атрибута id). В частности, при отображении таблицы применяется сплошная линия (инструкция border-style:solid) толщиной в 3 пикселя (инструкция border-width:3px). Цвет рамки определяется кодом #a0a0a0 (серый), а цвет фона таблицы определяется кодом #f0f0f0 (серый, но немного светлее цвета рамки).

Для кнопок в форме применяется такой стиль: шрифт жирный (инструкция font-weight:bold) размера 15 пикселей (инструкция font-size:15px), высота кнопки равна 30 пикселям (инструкция height:30px), а ширина каждой кнопки равна 50 пикселям (инструкция width:50px).

Наконец, для текстового поля в форме определена высота в 25 пикселей (инструкция height:25px), ширина поля составляет 145 пикселей (инструкция width:145px), цвет фона белый (инструкция background-color:white), содержимое поля отображается черным цветом (инструкция color:black), шрифтом размера 16 пикселей (инструкция font-size:16px) и выравнивается по правому краю (инструкция text-align:right), а внутренние отступы составляют 2 пикселя сверху и снизу и 5 пикселей справа и слева (инструкция padding:2px 5px).

Теперь мы обсудим код сценария, благодаря которому калькулятор реализует свое прямое назначение — производит вычисление арифметических выражений.

В сценарии достаточно часто используется несколько элементов, поэтому ссылки на них записываются в специальные переменные. Переменная output предназначена для записи ссылки на текстовое поле. Эта ссылка записывается в переменную при выполнении команды output=document.getElementById("myoutput") в обработчике события, связанно-

го с загрузкой документа. Там же выполняется команда btn=document.getElementById("btnC"), которой в переменную btn записывается ссылка на кнопку сброса.

Кроме этих переменных, мы используем переменные operand и operator. Дело в том, что при вычислении значения арифметического выражения нужно знать значения двух операндов и оператор, который к ним применяется. Второй операнд — это то число, которое записано в текстовом поле. А вот первый операнд (который должен быть введен ранее) будет записываться в переменную operand. Символ оператора, который применяется к операндам, планируется запоминать с помощью переменной operator.

Еще одна проблема технического характера связана с вводом чисел. В алгоритме, использованном в сценарии, задействованы две переменные numState и pntState, которые принимают значения логического типа и влияют на процесс ввода числа. Их использование мы обсудим позже.

Помимо этих переменных мы используем массив ops, элементами которого являются текстовые значения с символами операторов, включая оператор равенства (всего пять элементов в массиве).

Помимо объявления переменных, весь сценарий состоит фактически из описания обработчика события, связанного с загрузкой документа, а также функции btnPress(), которая (как мы увидим далее) вызывается при щелчке по кнопке, предназначенной для ввода цифры. Проанализируем код этой функции.

Функции передается один аргумент — предполагается, что это текстовое представление для цифры, которая вводится. Многое зависит от того, вводились ли уже цифры при вводе текущего числа. Проблема в том, что если цифры еще не вводились, то текстовое поле содержит число от предыдущих вычислений. В таком случае его следует из поля убрать и ввести туда первую цифру нового числа. Если цифры уже вводились, то новую цифру следует дописать к уже содержащимся цифрам. Но и здесь не так все просто. Если пользователь ввел первый ноль, то при вводе следующей цифры этот ведущий и математически ничего не значащий ноль должен быть удален. Чтобы реализовать обработку процесса, мы используем переменную numState. Предполагается, что значение этой переменной равно true, если при вводе числа цифры еще не вводились. Если значение переменной равно false, это означает, что

хотя бы одна цифра уже была введена. С учетом этого обстоятельства тело функции btnPress() состоит из условного оператора, в котором проверяется значение переменной numState. Если значение переменной равно true, то выполняется команда output.value=k, которой значение аргумента функции k присваивается свойству value объекта поля output. Проще говоря, введенная цифра отображается в текстовом поле. Поскольку теперь первая цифра при вводе числа введена, то выполняется команда numState=false, и режим ввода числа изменяется.

Если при проверке значения переменой numState оказывается, что оно равно false, то в дело вступает еще один условный оператор. В нем проверяется условие output.value=="0", означающее, что поле содержит нулевое значение. В таком случае командой output.value=k введенная цифра отображается в поле (а ноль, соответственно, там больше не отображается). Наконец, если поле содержит значение, отличное от нуля, выполняется команда output.value+=k, которой введенная цифра дописывается к текущему значению в поле.

В обработчике события, связанного с загрузкой документа, после выполнения ссылок на поле и кнопку сброса командой output.disabled=true поле делается неактивным (блокируется). Такое поле нельзя выделить мышью и поместить туда курсор для ввода значения. Зато в него можно вносить значения программными методами.

НА ЗАМЕТКУ

По умолчанию заблокированное поле отображается серым цветом, и серым же цветом (но немного темнее фона) отображается текст в заблокированном поле. Поэтому в настройках стиля мы задали для поля белый фон и черный цвет для текста в поле.

Несмотря на то что поле заблокировано, в сценарии содержится обработка событий, связанных с нажатием клавиш на клавиатуре (при условии, что в форме есть кнопка с переданным фокусом). Поэтому при работе с калькулятором можно использовать клавиатуру для ввода цифр и символов операторов.

Также обработчик события загрузки документа содержит определение нескольких других обработчиков. Обработчик события, связанного со щелчком по кнопке сброса, содержит команду output.value="0", которой выполняется обнуление значения в текстовом поле. Командами pntState=true и numState=true задаются начальные значения для переменных, определяющих режим ввода десятичной точки и числа.

Командами operator="" и operand="" в переменные, определяющие оператор и первый операнд выражения, записываются пустые текстовые строки. Таким образом, щелчок по кнопке сброса переводит все переменные и текстовое поле в «начальное» состояние. В самом конце кода обработчика события, связанного с загрузкой документа, есть команда btn.click(). Этой командой выполняется программный щелчок по кнопке сброса. То есть физического щелчка нет, но последствия такие, как если бы он был.

Проблема с вводом десятичной точки связана с тем, что десятичная точка в числе может быть только одна. Если пользователь при вводе числа одну точку уже ввел, то ввод другой точки должен быть заблокирован. Как индикатор возможности ввода десятичной точки используется переменная pntState. Если значение переменной равно false, то точку вводить нельзя. Обработчик события, связанного со щелчком по кнопке ввода десятичной точки, начинается с проверки значения переменной pntState в условном операторе. Все перечисленные далее действия выполняются, только если переменная имеет значение true.

НА ЗАМЕТКУ

Если значение переменной pntState равно false, то щелчок по кнопке ввода десятичной точки ни к чему не приведет.

Но, даже если точку можно вводить, ситуация неоднозначная. Так, если при вводе числа цифры еще не вводились, то следует не просто ввести точку (дописать ее к текущему представлению числа), но и добавить нуль в целой части. Для этого в условном операторе проверяется значение переменной numState, и если оно равно true, то командой output.value="0." в текстовое поле точка добавляется вместе с ведущим нулем. После этого командой numState=false изменяется значение переменной, определяющей режим ввода числа (новое значение означает, что при вводе числа уже введена по крайней мере одна цифра).

Если значение переменной numState при проверке оказывается равным false, то командой output.value+="." десятичная точка дописывается к текущему представлению числа в поле. После завершения выполнения внутреннего условного оператора командой pntState=false блокируется возможность повторного ввода десятичной точки.

Для создания обработчиков событий, связанных со щелчками по кнопкам с цифрами, запускается оператор цикла. Индексная пере-

менная k в операторе цикла пробегает значения от 0 до 9. При каждом заданном значении индексной переменной свойству onclick элемента, ссылку на который получаем с помощью инструкции document.getElementById("btn"+k), значением присваивается анонимная функция. При получении ссылки на объект мы учли, что для кнопок с цифрами значения их атрибута id получаются объединением текста "btn" и цифры, отображаемой на кнопке. Что касается анонимной функции, то в ней всего одна команда btnPress(this.value), которой вызывается функция btnPress(), а аргументом ей передается выражение this.value. Поскольку функция будет вызываться из объекта кнопки, то ключевое слово this возвращает ссылку на кнопку, из которой осуществляется вызов. Таким образом, выражение this.value является значением свойства value объекта кнопки — а это цифра, написанная на кнопке.

НА ЗАМЕТКУ

Может возникнуть соблазн вместо выражения this.value использовать индексную переменную k. Однако такой код будет некорректным. В данном случае функция-обработчик (поскольку это внутренняя функция) получила бы аргументом ссылку на локальную переменную k внешней функции. Значение переменной после выполнения оператора цикла равно 10. Как следствие, при нажатии любой из кнопок с цифрами функция btnPress() будет вызываться с аргументом 10.

При создании обработчиков для кнопок с символами операторов также используется оператор цикла, но на этот раз индексная переменная p пробегает значения индексов элементов массива ops, который, напомним, содержит символы операторов (в том числе и оператора равенства). Ссылку на объект кнопки получаем с помощью инструкции document.getElementById("btn"+ops[p]). Здесь мы также учли, что для обрабатываемых кнопок значение атрибута id получается объединением текста "btn" и символа оператора. Символ оператора можно получить из массива ops. Свойству onclick объекта кнопки присваивается анонимная функция. В теле функции в условном операторе проверяется условие operand!=""&&operand!="Ошибка!". Условие истинное, если в переменной operand, куда записывается значение первого операнда, содержится не пустая строка и не текст "Ошибка!". Пустая текстовая строка в переменную operand записывается, например, при нажатии на кнопку сброса, а текст "Ошибка!", как мы увидим это далее, может оказаться значением первого операнда как следствие возникновения ошибки при предыдущих вычислениях. Фактически истинность указанно-

го выше условия означает, что первый операнд существует. Если так, то командой res=eval(operand+operator+output.value) производится вычисление арифметического выражения, и результат вычислений записывается в переменную res. Результатом выражения operand+operator+output.value является текстовая строка, получающаяся объединением первого операнда, символа оператора и значения из текстового поля. Это есть не что иное, как выражение, которое следует вычислить, но только оно записано в виде текстовой строки. Чтобы вычислить данное выражение, мы передаем текстовую строку аргументом встроенной функции eval(). Теоретически результатом мы должны получить число. Но здесь возможны некоторые неприятные ситуации. Например, если выполняется попытка деления на ноль ненулевого числа, то результатом будет бесконечность ("Infinity" или "-Infinity" в зависимости от знака делимого числа). Если, например, попытаться поделить ноль на ноль, то получим объект NaN (не число). В таком случае результатом выражения isNaN(res) будет true. Мы хотим обработать все означенные ситуации, поэтому используем условный оператор, в котором проверяется условие res=="Infinity"||res=="-Infinity"||isNaN(res). При истинном условии (произошла ошибка) командами operator="" и operand="" «сбрасываются» значения оператора и первого операнда, а командой output.value="Ошибка!" задается значение в текстовом поле. Как при этом выглядит документ с калькулятором, показано на рис. 9.11.

Рис. 9.11. Вид документа с калькулятором при возникновении ошибки при вычислениях

Если описанные ошибки не случились, то командой output.value=res вычисленное значение отображается в текстовом поле. Далее проверяется условие this.value=="=". Условие истинно, если свойство value объекта кнопки (по которой выполнен щелчок) равно "=" — то есть если речь идет о кнопке ввода оператора равенства. Если пользователь щелкнул по кнопке с оператором равенства, то командой operand="" в переменную, хранящую значение первого операнда, записывается пустая текстовая строка. Если условие ложно (щелчок выполнен не по кнопке ввода оператора равенства), то с помощью команды operand=res вычисленное значение становится первым операндом.

После выполнения команды operator=this.value в переменную operator записывается символ оператора, соответствующего нажатой кнопке. Все описанные выше действия (с учетом разных условий) происходят, если существует первый операнд. Если же первый операнд отсутствует (равен пустой строке или тексту "Ошибка!"), то последовательность действий несколько иная. А именно сначала проверяется условие this.value!="=" (нажата не кнопка для ввода оператора равенства). При истинном условии командой operator=this.value определяется новое значение для символа оператора (новое значение определяется по кнопке, которая нажата), а командой operand=output.value содержимое текстового поля присваивается переменной, в которой хранится значение первого операнда (то есть значение в поле становится первым операндом). Если условие this.value!="=" ложно (нажата кнопка ввода оператора равенства), то описанные выше две команды не выполняются. Но в любом случае (при истинном и при ложном условии this.value!="=") выполняются команды numState=true и pntState=true, которыми дается «зеленый свет» для ввода нового числа и десятичной точки.

Для формы определяется обработчик события, связанного с вводом символа с клавиатуры. Ссылку на объект формы получаем с помощью инструкции document.getElementById("mycalc"). Свойству onkeypress объекта присваивается анонимная функция, аргумент которой evt отождествляется с объектом события. В теле функции командой nums=new Array(10) создается массив из десяти элементов. Значениями элементам массива присваиваются текстовые представления для цифр от 0 до 9. Для этого запускается оператор цикла, в котором при заданном значении индексной переменной выполняется команда nums[k]=""+k. В правой части объединяется пустая текстовая строка с числом, в результате чего выполняется конкатенация текстовых строк (число приводится к тестовому представлению), и мы получаем текстовую строку, содер-

жащую данное число. Объект события используется для определения введенного символа. Для этого сначала определяется код введенного символа, а затем по этому коду вычисляется символ. Код введенного символа вычисляется на основе объекта события инструкцией evt.which. Здесь использовано свойство which объекта события. Для определения символа по его коду используем метод fromCharCode() объекта String. В результате получаем выражение String.fromCharCode(evt.which). Полученное значение записываем в переменную symb.

НА ЗАМЕТКУ

Свойство which возвращает значением код символа, если речь идет об обработке события onkeypress. Если обрабатываются события onkeydown или onkeyup, то значением свойства является код нажатой клавиши. Различие между кодом символа и кодом клавиши состоит в том, что код символа представляет собой код в таблице ASCII, в то время как код клавиши идентифицирует клавишу на клавиатуре. Поэтому, например, если вводить одну и ту же букву, но сначала строчную, а затем прописную, то коды символов в этих случаях будут разными, а код клавиши один и тот же.

Для получения кода клавиши также можно использовать свойство keyCode. Получить код символа можно с помощью свойства charCode. Следует также отметить, что свойство which не поддерживается старыми версиями браузера Internet Explorer. Если нет уверенности в том, что конечный пользователь не будет использовать такой браузер, вместо выражения evt.which можно использовать конструкцию evt.which||evt.keyCode. Результатом такого выражения является значение evt.which, если свойство which поддерживается браузером. Если свойство браузером не поддерживается, то результатом возвращается значение выражения evt.keyCode.

Кроме перечисленных свойств, существует рекомендуемое к использованию свойство key, возвращающее название нажатой клавиши/символа. Правда, поддерживается свойство только в наиболее популярных браузерах последних версий.

После того как введенный символ определен, начинается последовательная проверка на предмет выбора корректного способа обработки события. Сначала в условном операторе проверяется условие symb in nums, истинное в том случае, если значение переменной symb совпадает со значением одного из индексов в массиве nums. Если речь идет о вводе цифры с клавиатуры, то значением переменной symb является текстовое представление цифры. Массив nums, в свою очередь, состо-

ит из десяти цифр, и индексы элементов массива совпадают со значениями элементов. Поэтому если в переменную symb записана цифра (текстовое представление), то условие symb in nums истинно. Напротив, если переменная symb не содержит цифру, то условие symb in nums ложно, поскольку массив не содержит нечисловых индексов.

При истинном условии symb in nums для обработки события выполняется команда btnPress(symb). Здесь мы вызываем ту же функцию, которая вызывается при нажатии кнопки с цифрой непосредственно в форме калькулятора.

Если условие symb in nums ложно, то вступает в действие другой условный оператор. В нем проверяется условие symb==".". Его истинность означает, что с клавиатуры вводится точка. В этом случае выполняется команда document.getElementById("btnDot").click(), которой инициируется программный щелчок по кнопке калькулятора с точкой. Доступ к объекту кнопки калькулятора мы получили с помощью инструкции document.getElementById("btnDot"), а потом вызвали через полученную ссылку метод click(), сымитировав тем самым щелчок по кнопке.

Если и этот вариант не реализовался, то запускается оператор цикла, в котором индексная переменная k последовательно перебирает значения индексов элементов массива ops (массив с символами операторов). За каждый цикл проверяется условие symb==ops[k] (введенный символ совпадает с символом оператора из массива ops). Если совпадение найдено, то задействуется команда document.getElementById("btn"+symb).click(). Мы, как и в предыдущем случае, выполняем программный щелчок по кнопке калькулятора. Для доступа к кнопке использовано значение ее атрибута id, которое получается объединением текста "btn" и символа оператора. Чтобы избежать дальнейших ненужных проверок, используется инструкция return, завершающая работу метода обработки события.

Таким образом, основные принципы функционирования калькулятора следующие.

- При загрузке документа (см. рис. 9.10) в поле отображается нулевое значение, первый операнд отсутствует (пустая текстовая строка), оператор для выполнения операции отсутствует (пустая текстовая строка). К такому же состоянию калькулятор приводится при щелчке по кнопке сброса.
- При вводе чисел можно вводить десятичную точку. Если точка вводится в самом начале ввода числа, то автоматически добавля-

ется ведущий ноль. Например, если после загрузки документа или щелчка по кнопке сброса нажать кнопку с точкой, то в поле отображается значение 0.. Такое же значение появится в поле, если после нажатия кнопки с символом оператора нажать кнопку с точкой.

- При щелчке по кнопке с символом оператора (если это не оператор равенства) возможны два варианта. Если первый операнд существует, то вычисляется соответствующее выражение, которое становится значением первого операнда и отображается в поле. При вычислении выражения используется старое значение для оператора (который был введен на предыдущем этапе). После вычисления выражения старое значение оператора заменяется на то, что было введено при щелчке по кнопке. Если же при вводе оператора первый операнд отсутствует, то текущее значение в поле становится значением первого операнда, а текущим значением оператора становится то, что было введено при нажатии кнопки. Например, если мы вводим число 12, оператор +, число 3 и оператор *, то в поле отображается число 15, а следующей операцией будет выполняться умножение. Если мы продолжим и введем число 2 и оператор -, то в поле будет значение 30, а следующая операция — вычитание. А если число 2 не вводить, а сразу ввести оператор -, то в поле появляется значение 225. Объяснение такое: перед вводом оператора - был введен оператор *. Поэтому выполняется умножение. Первый операнд равен 15, и такое же значение записано в поле. Второй операнд определяется по содержимому поля. В результате 15 умножается на 15, что и дает 225.

- При вводе оператора равенства, если первый операнд существует, вычисляется значение выражения, и оно отображается в поле. Первый операнд получает значением пустую строку. Если при вводе оператора равенства первый операнд отсутствует, то в поле остается то значение, что было до ввода оператора. Например, мы последовательно вводим числа 2, оператор * и 3, а затем оператор равенства. Как следствие, в поле отображается значение 6. Далее по логике следует ввести оператор и число. Скажем, если ввести +, затем 4 и знак равенства, то в поле будет отображаться значение 10. Если после знака равенства начать вводить число, то это фактически будет как если бы мы начали вводить число после нажатия на кнопку сброса — значение, которое было до этого в поле, теряется.

- Если при вычислениях возникает ошибка (деления на ноль), то в поле отображается сообщение Ошибка!. Чтобы увидеть в поле та-

кое значение, достаточно выполнить деление на ноль. При этом следующие операции могут выполняться без предварительного нажатия кнопки сброса. Так, если после появления сообщения об ошибке вводить число, то это эквивалентно вводу числа после нажатия кнопки сброса. Если вести оператор, то внешнего эффекта это иметь не будет.

- Цифры, десятичную точку и операторы можно вводить с клавиатуры. Но для этого в форме калькулятора должен быть активный элемент (кнопка). Такой активный элемент появляется в форме после первого щелчка по кнопке калькулятора (например, это может быть кнопка сброса).

НА ЗАМЕТКУ

Документ вполне рабочий, но все же надо понимать, что пример — учебный, поэтому некоторые возможности в калькуляторе не реализованы. Мы, например, не предусмотрели ввод отрицательных чисел. Неплохо было бы реализовать и возможность выполнения некоторых вычислений — например, возведение в степень или вычисление синуса и косинуса. Желающие могут попробовать свои силы и усовершенствовать предложенный выше код.

Также следует иметь в виду, что в разных браузерах некоторые горячие комбинации клавиш совпадают с комбинациями, используемыми при наборе символов для операторов. Так что «суммарный» эффект может быть несколько неожиданным.

Бегущий текст

Понимаете, можно и зайца, конечно, научить курить...

из к/ф «Служебный роман»

Следующий пример иллюстрирует способ создания в документе бегущего текста. Документ достаточно простой. При его загрузке в окно браузера появляется прямоугольная белая область, выделенная серой рамкой, как показано на рис. 9.12.

Сразу после загрузки документа в этой области начинает отображаться текст. Внешне эффект такой, что текст отображается буква за буквой. Интервал между отображением каждой следующей буквы составляет порядка половины секунды. На рис. 9.13 показан документ в процессе отображения текста.

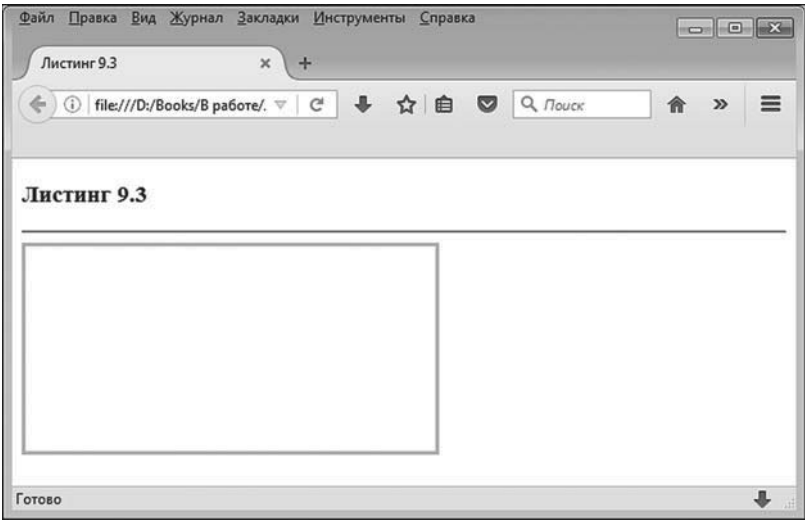

Рис. 9.12. Вид документа сразу после загрузки: документ содержит область для вывода текста

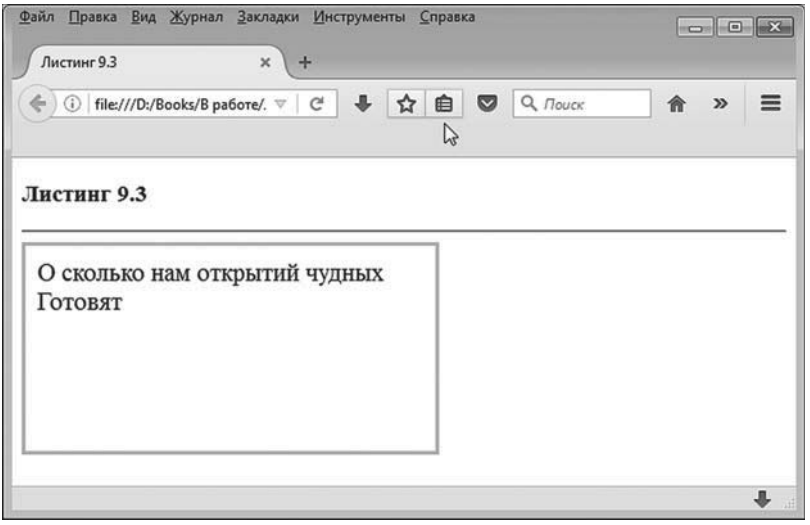

Рис. 9.13. После загрузки в области вывода отображается текст. Интервал времени между отображением букв составляет половину секунды

После того как текст полностью отображается в области вывода, делается пауза длительностью в 4 секунды.

На рис. 9.14 показан документ с полностью отображенным текстом. После паузы область вывода очищается, и текст отображается еще

раз. Такое периодическое отображение текста выполняется фактически до бесконечности (в реальности — пока пользователь не закроет документ).

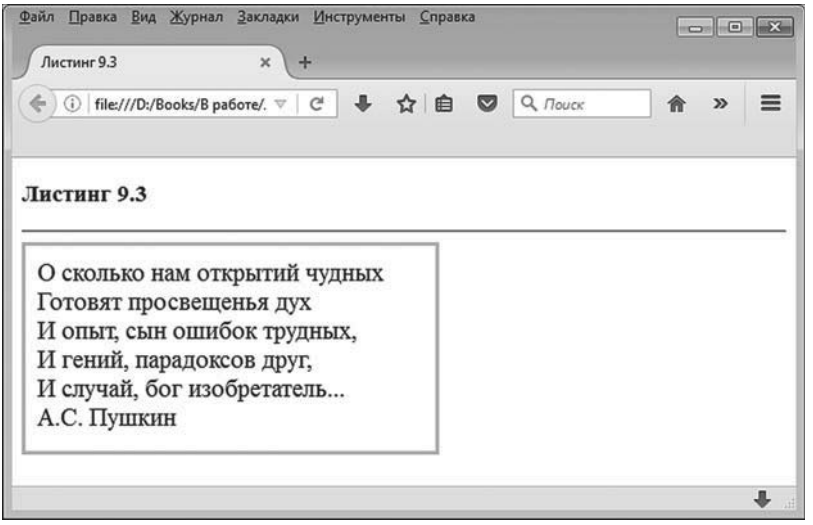

Рис. 9.14. После отображения всего текста делается пауза в 4 секунды, после чего текст начинает отображаться заново

Программный код, с помощью которого реализуется эффект бегущего текста, представлен в листинге 9.3.

Листинг 9.3. Бегущий текст

```
<!DOCTYPE HTML>
<html>
<head>
<title>Листинг 9.3</title>
<!-- Начало сценария -->
<script type="text/javascript">
  // Текстовое значения для отображения в документе:
  var txt="О сколько нам открытий чудных<br>"
    txt+="Готовят просвещенья дух<br>"
    txt+="И опыт, сын ошибок трудных,<br>"
    txt+="И гений, парадоксов друг,<br>"
    txt+="И случай, бог изобретатель…<br>"
```

```
    txt+="А.С. Пушкин"
    // Длина (в символах) отображаемого текста:
    var n=0
    // Переменная для записи ссылки на объект области,
    // в которую выводится текст:
    var ref
  // Функция, которая вызывается при отображении текста:
  function writeText(){
    // Переменная для записи значения временной задержки:
    var time
    // Подстрока текста записывается в область вывода:
    ref.innerHTML=txt.substring(0,n)
    // Если не достигнут конец текста:
    if(n<txt.length){
      // Длина отображаемой подстроки увеличивается
      // на один символ:
      n++
      // Задержка (в миллисекундах) перед вызовом
      // функции:
      time=500
    }
    // Если достигнут конец текста:
    else{
      // Начальное значение для длины подстроки,
      // отображаемой в области вывода:
      n=0
      // Задержка (в миллисекундах) перед вызовом
      // функции:
      time=4000
    }
    // Рекурсивный вызов функции с временной задержкой:
    setTimeout(writeText,time)
  }
  // Обработчик события, связанного с загрузкой документа:
```

```
  window.onload=function(){
    // Ссылка на объект области вывода:
    ref=document.getElementById("mytext")
    // Вызов функции, отображающей текст:
    writeText()
  }
</script>
<!-- Завершение сценария -->
<!-- Описание стиля для области вывода  -->
<style type="text/css">
  #mytext{
    height: 150px;
    width: 320px;
    border: solid;
    border-color: silver;
    border-width: 3px;
    font-family: Times New Roman;
    font-size: 16pt;
    padding: 10px;
  }
</style>
<!-- Завершение описания стиля -->
</head>
<body>
  <h3>Листинг 9.3</h3><hr>
  <!-- Область вывода -->
  <div id="mytext"></div>
</body>
</html>
```

В сценарии объявляется переменная txt, значением которой (в несколько приемов) присваиваются строки стихотворения. Это именно тот текст, который предназначен для отображения в документе. Использованный в сценарии алгоритм базируется на том, что из исходного теста отображается подстрока и длина отображаемой подстро-

ки каждый раз увеличивается на один символ. Так происходит, пока длина подстроки не сравняется с длиной исходного текста. В таком случае весь процесс повторяется с самого начала.

Длина подстроки, отображаемой в области вывода, будет записываться в переменную n. Начальное значение переменной равно нулю. Также мы используем переменную ref, в которую записывается ссылка на область вывода. Область вывода реализована с помощью -блока, атрибут id которого имеет значение "mytext".

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Стиль для области вывода устанавливает следующие характеристики: высота области составляет 150 пикселей (инструкция height:150px), ширина области равна 320 пикселям (инструкция width:320px), рамка вокруг области сплошная (инструкция border:solid), цвет рамки — серебряный (инструкция border-color:silver), ширина рамки равна 3 пикселям (инструкция border-width:3px), используется шрифт типа Times New Roman (инструкция font-family:Times New Roman) размера 16 (инструкция font-size:16pt), а внутренние отступы составляют 10 пикселей (инструкция padding:10px).

Ссылка на объект области вывода выполняется командой ref=document.getElementById("mytext") в теле функции, используемой в качестве обработчика события, связанного с загрузкой документа. Еще одной командой в обработчике вызывается функция writeText(), которая и выполняет всю «работу» по отображению текста.

В теле функции объявляется переменная time, в которую будет записано числовое значение для задержки (в миллисекундах), выполняемой при отображении текста. Поскольку на момент первого вызова функции переменная ref уже содержит ссылку на область вывода, то мы можем использовать эту переменную для доступа к области. Командой ref.innerHTML=txt.substring(0,n) в области вывода отображается текст. Текст — это подстрока, которая извлекается из текстового значения, записанного в переменную txt.

Для извлечения подстроки использован встроенный метод substring(). Первым аргументом методу передается индекс символа, начиная с которого выполняется извлечение подстроки. Второй аргумент метода — индекс первого не входящего в подстроку символа. Таким образом, выражение txt.substring(0,n) возвращает результатом подстроку из

текста в переменной txt, и эта подстрока начинается с символа с индексом 0 и заканчивается символом с индексом n-1.

Чтобы извлечь из переменной txt весь текст в виде подстроки, вторым аргументом следует указать значение, совпадающее с длиной текста в переменной txt.

НА ЗАМЕТКУ

Учитывая, что подстрока извлекается с самого начала строки, значение переменной n в данном конкретном случае можно интерпретировать как длину подстроки.

После отображения подстроки в области вывода в условном операторе проверяется условие n<txt.length. Условие истинно, если длина отображенной подстроки меньше длины текста в переменной txt. Если так, то командой n++ значение переменной n увеличивается на единицу, и, следовательно, в следующий раз при отображении подстроки ее длина будет на один символ больше. Значение переменной time присваивается командой time=500.

Если же условие n<txt.length ложно, то длина подстроки сравнялась с длиной исходного текста. В таком случае командой n=0 длина подстроки возвращается к нулевому начальному значению. Значение переменной time в этом случае определяется командой time=4000. После завершения выполнения условного оператора выполняется команда setTimeout(writeText,time). Этой командой в теле функции writeText() вызывается эта же функция writeText(). То есть имеет место рекурсия. Однако вызов выполняется с временной задержкой. Интервал задержки определяется значением переменной time. Если речь идет об отображении подстроки, пока ее длина меньше длины исходного текста, то интервал между вызовами функции writeText() составляет 500 миллисекунд. Если длина строки сравнялась с длиной исходного текста, то следующий вызов функции writeText() произойдет через 4000 миллисекунд.

НА ЗАМЕТКУ

Таким образом, мы организовали бесконечный периодический процесс. Базируется он на том, что функция writeText() рекурсивно вызывает сама себя, но каждый такой вызов выполняется с определенной задержкой во времени.

Игра «Жизнь»

> Огонь тоже считался божественным, пока Прометей не выкрал его. Теперь мы кипятим на нем воду.
>
> *из к/ф «Формула любви»*

В следующем примере мы смоделируем определенный процесс. Фактически речь идет о создании анимации (правда, очень скромной). Мы рассмотрим документ, в котором симулируется игра, называющаяся «Жизнь». Правила игры очень простые. Имеется массив из клеток. Каждая клетка может находиться в одном из двух состояний: она может быть живой, и она может быть мертвой. Причем живые клетки могут умирать, а мертвые клетки могут оживать. Правила превращений клеток такие.

- Если вокруг мертвой клетки ровно три живые, то мертвая клетка оживает.
- Если вокруг живой клетки находится две или три живые клетки, то такая клетка продолжает оставаться живой. В противном случае (живых соседей больше трех или меньше двух) живая клетка погибает.

Наша задача состоит в том, чтобы смоделировать описанный процесс с учетом означенных выше правил. Более конкретно, стоящая перед нами задача формулируется следующим образом. Имеется прямоугольная графическая область заданных размеров. Она разбивается на клеточки маленького размера (в несколько пикселей). Эти маленькие клеточки отождествляются с клетками в игре. Мертвые клетки отображаются белым цветом, а живые клетки отображаются синим цветом. Вначале задается случайное распределение мертвых и живых клеток. После этого на основе начальной конфигурации рассчитываются следующие поколения клеток (с учетом правил игры). Таким образом, получаем последовательность «кадров», каждый из которых представляет собой картинку с изображением мертвых и живых клеток. Эти кадры последовательно отображаются в графической области.

НА ЗАМЕТКУ

Каждая внутренняя клетка имеет ровно восемь соседних клеток. Что касается клеток, которые расположены на границах области, то мы используем периодические граничные условия: для клеток

на левой границе соседними, кроме смежных внутренних клеток, будут клетки на правой границе. Для клеток на правой границе соседними являются клетки на левой границе. Соседними считаются клетки, размещенные на верхней и нижней границах. Таким образом, задача имеет топологию тора: то есть фактически процесс происходит на поверхности, напоминающей поверхность бублика.

Мы в первую очередь рассмотрим код документа, представленный в листинге 9.4.

Листинг 9.4. Игра «Жизнь»

```
<!DOCTYPE HTML>
<html>
<head>
<title>Листинг 9.4</title>
<!-- Начало сценария -->
<script type="text/javascript">
  // Переменная для записи идентификатора процесса,
  // запускаемого с помощью функции setInterval():
  var timerID
  // Количество клеток по горизонтали и вертикали:
  var Nx=200,Ny=100
  // Размер клетки (в пикселях):
  var dz=3
  // Переменная для записи ссылки
  // на раскрывающийся список:
  var list
  // Переменные для записи ссылок на кнопки:
  var start,conf
  // Два названия для одной из кнопок:
  var nameA="Начать"
  var nameB="Остановить"
  // Переменные для записи ссылок на объект графической
  // области и объект графического контекста:
  var cnv,ctx
  // Переменная для записи ссылки на массив, через который
```

```
// реализуется конфигурация клеток:
var A
// Функция для определения начального распределения
// живых и мертвых клеток:
function init(B){
  // Локальные переменные:
  var i,j,p
  // Вероятность того, что клетка живая:
  p=list.value
  // Перебор строк массива:
  for(i=0;i<B.length;i++){
    // Перебор элементов в строке:
    for(j=0;j<B[i].length;j++){
      // Если случайное число меньше вероятности:
      if(Math.random()<p){
        // Живая клетка:
        B[i][j]=1
      }
      // Если случайное число не меньше вероятности:
      else{
        // Мертвая клетка:
        B[i][j]=0
      }
    }
  }
}
// Функция для подсчета живых клеток, являющихся
// соседними для заданной клетки:
function getState(B,i,j){
  // Локальные переменные:
  var im,ip,jm,jp,r
  // Если клетка не в первой строке:
  if(i>0){
    // Первый индекс для соседней клетки сверху:
```

```
  im=i-1
}
// Если клетка в первой строке:
else{
  // Первый индекс для соседней клетки "сверху":
  im=B.length-1
}
// Если клетка не в последней строке:
if(i<B.length-1){
  // Первый индекс для соседней клетки снизу:
  ip=i+1
}
// Если клетка в последней строке:
else{
  // Первый индекс для соседней клетки "снизу":
  ip=0
}
// Если клетка не в первом столбце:
if(j>0){
  // Второй индекс для соседней клетки слева:
  jm=j-1
}
// Если клетка в первом столбце:
else{
  // Второй индекс для соседней клетки "слева":
  jm=B[i].length-1
}
// Если клетка не в последнем столбце:
if(j<B[i].length-1){
  // Второй индекс для соседней клетки справа:
  jp=j+1
}
// Если клетка в последнем столбце:
else{
```

```
    // Второй индекс для соседней клетки "справа":
    jp=0
  }
  // Вычисление количества живых соседних клеток:
  r=B[i][jm]+B[i][jp]+B[im][j]+B[ip][j]+B[im][jm]+B[ip][jp]+B[im][jp]+B[ip][jm]
  // Результат функции:
  return r
}
// Функция для вычисления новой конфигурации
// (нового поколения клеток):
function recalc(B){
  // Локальная переменная:
  var nbs
  // Локальный массив для записи новой конфигурации:
  var C=new Array(B.length)
  // Перебор элементов локального массива:
  for(var i=0;i<C.length;i++){
    // Создание новой строки:
    C[i]=new Array(B[i].length)
    // Перебор элементов в строке:
    for(var j=0;j<C[i].length;j++){
      // Количество живых соседних клеток для
      // исходной конфигурации:
      nbs=getState(B,i,j)
      // Если текущая клетка в исходной
      // конфигурации мертвая:
      if(B[i][j]==0){
        // Если ровно три живых соседних клетки:
        if(nbs==3){
          // В новой конфигурации клетка оживает:
          C[i][j]=1
        }
        // Если живых клеток не три:
        else{
```

```
            // В новой конфигурации клетка остается
            // мертвой:
            C[i][j]=0
          }
        }
        // Если текущая клетка в исходной
        // конфигурации живая:
        else{
          // Если живых соседних клеток две или три:
          if((nbs==2)||(nbs==3)){
            // В новой конфигурации клетка остается
            // живой:
            C[i][j]=1
          }
          // Если живых соседних клеток
          // не три и не две:
          else{
            // В новой конфигурации клетка
            // становится мертвой:
            C[i][j]=0
          }
        }
      }
    }
    // Результат функции:
    return C
}
// Функция для отображения картинки на основе
// конфигурации живых и мертвых клеток:
function show(B){
    // Очистка графической области:
    ctx.clearRect(0,0,cnv.width,cnv.height)
    // Синий цвет для выполнения заливки области
    // с живыми клетками:
```

```
    ctx.fillStyle="blue"
    // Перебор строк массива:
    for(var i=0;i<B.length;i++){
      // Перебор элементов в строке:
      for(var j=0;j<B[i].length;j++){
        // Если клетка живая:
        if(B[i][j]==1){
          // Заливка области клетки цветом:
          ctx.fillRect(dz*j,dz*i,dz,dz)
        }
      }
    }
  }
  // Функция для вычисления новой конфигурации
  // и отображения картинки:
  function showNext(){
    // Новая конфигурация:
    A=recalc(A)
    // Отображение новой картинки:
    show(A)
  }
  // Функция для вызова при генерировании начальной
  // конфигурации клеток:
  function config(){
    // Начальное распределение живых и мертвых клеток:
    init(A)
    // Отображение картинки:
    show(A)
  }
  // Обработчик события, связанного с загрузкой документа:
  window.onload=function(){
    // Получение ссылки на объект графической области:
    cnv=document.getElementById("mycanvas")
    // Ширина графической области:
```

```
cnv.width=Nx*dz
// Высота графической области:
cnv.height=Ny*dz
// Ширина формы с элементами управления:
document.getElementById("myform").style.width=cnv.width-10+"px"
// Получение ссылки на объект графического контекста:
ctx=cnv.getContext("2d")
// Получение ссылки на объект кнопки выбора
// начальной конфигурации живых и мертвых клеток:
conf=document.getElementById("conf")
// Кнопка выбора конфигурации в активном состоянии:
conf.disabled=false
// Получение ссылки на объект кнопки, используемой
// для запуска и остановки вычислений:
start=document.getElementById("start")
// Название для кнопки запуска вычислений:
start.value=nameA
// Получение ссылки на объект раскрывающегося списка:
list=document.getElementById("prob")
// Список в активном состоянии:
list.disabled=false
// Индекс выбранного в списке пункта:
var index=1
// Выбор пункта в раскрывающемся списке:
list.selectedIndex=index
// Массив для реализации конфигурации клеток:
A=new Array(Ny)
// Создание каждой отдельной строки:
for(var k=0;k<A.length;k++){
  // Новая строка:
  A[k]=new Array(Nx)
}
// Начальная конфигурация:
config()
```

```
    // Обработчик щелчка на кнопке выбора конфигурации:
    conf.onclick=config
    // Обработчик для события, связанного с изменением
    // состояния раскрывающегося списка:
    list.onchange=conf.onclick
    // Обработчик события, связанного со щелчком
    // на кнопке запуска/остановки вычислений:
    start.onclick=function(){
      // Если у кнопки первое название:
      if(this.value==nameA){
        // Блокировка раскрывающегося списка:
        list.disabled=true
        // Блокировка кнопки выбора конфигурации:
        conf.disabled=true
        // Изменение названия кнопки:
        this.value=nameB
        // Запуск процесса вычислений:
        timerID=setInterval(showNext,100)
      }
      // Если у кнопки другое название:
      else{
        // Отмена блокировки раскрывающегося списка:
        list.disabled=false
        // Отмена блокировки кнопки
        // выбора конфигурации:
        conf.disabled=false
        // Изменение названия кнопки:
        this.value=nameA
        // Остановка процесса вычислений:
        clearInterval(timerID)
      }
    }
  }
</script>
```

```
<!-- Завершение сценария -->
<!-- Описание стилей -->
<style type="text/css">
  /* Стиль для формы myform */
  #myform{
    height: 30px;
    border-style: outset;
    padding: 5px;
    background-color: #f0f0f0;
    font-size: 17px;
  }
  /* Стиль для кнопок формы myform */
  #myform input[type="button"]{
    font-weight: bold;
    height: 30px;
  }
  /* Стиль для раскрывающегося списка формы myform */
  #myform select{
    height: 30px;
    border-color: silver;
    font-weight: bold;
  }
</style>
<!-- Завершение описания стилей -->
</head>
<body>
  <h3>Листинг 9.4</h3><hr>
  <!-- Форма с элементами управления -->
  <form id="myform">
    <!-- Текст -->
    <b>Вероятность</b>
    <!-- Раскрывающийся список -->
    <select size="1" id="prob">
      <option value="0.1">0.10</option>
```

```
      <option value="0.2">0.20</option>
      <option value="0.3">0.30</option>
      <option value="0.5">0.50</option>
      <option value="0.75">0.75</option>
    </select>
    <!-- Кнопка выбора конфигурации -->
    <input type="button" id="conf" value="Новая конфигурация">
    <!-- Кнопка начала и остановки вычислений -->
    <input type="button" id="start" style="width:120px;">
  </form>
  <!-- Графическая область -->
  <canvas id="mycanvas" style="border:ridge;"></canvas>
</body>
</html>
```

Документ состоит из формы и графической области. В графической области отображается конфигурация живых и мертвых клеток (живые клетки отображаются синим цветом, а мертвые клетки формируют белый фон). Форма содержит несколько элементов управления: раскрывающийся список, в котором выбирается вероятность для клетки быть живой в первом (начальном) поколении, кнопку **Новая конфигурация**, щелчок по которой приводит к генерированию новой начальной конфигурации, а также кнопка **Начать**, щелчок по которой запускает процесс вычислений и приводит к последовательному отображению конфигураций мертвых и живых клеток. Как выглядит документ при загрузке, показано на рис. 9.15.

Если в раскрывающемся списке выбрать новое значение, начальная конфигурация будет автоматически пересчитана. К автоматическому пересчету начальной конфигурации приводит и щелчок по кнопке **Новая конфигурация**.

НА ЗАМЕТКУ

При генерировании начального распределения для клеток живые и мертвые клетки распределяются случайно. Значение в раскрывающемся списке определяет вероятность того, что произвольно выбранная клетка является живой. Более конкретно, новая конфигурация рассчитывается так: перебираются все клетки, и статус

каждой клетки (живая или мертвая) определяется случайным образом. Технически процесс заключается в следующем: при определении статуса каждой клетки генерируется случайное число в диапазоне значений от 0 до 1, и если это число меньше вероятности, выбранной в раскрывающемся списке, то клетка считается живой. В противном случае (если случайное число не меньше вероятности в раскрывающемся списке) клетка считается мертвой. Значение в раскрывающемся списке можно интерпретировать как примерную долю живых клеток в начальной конфигурации.

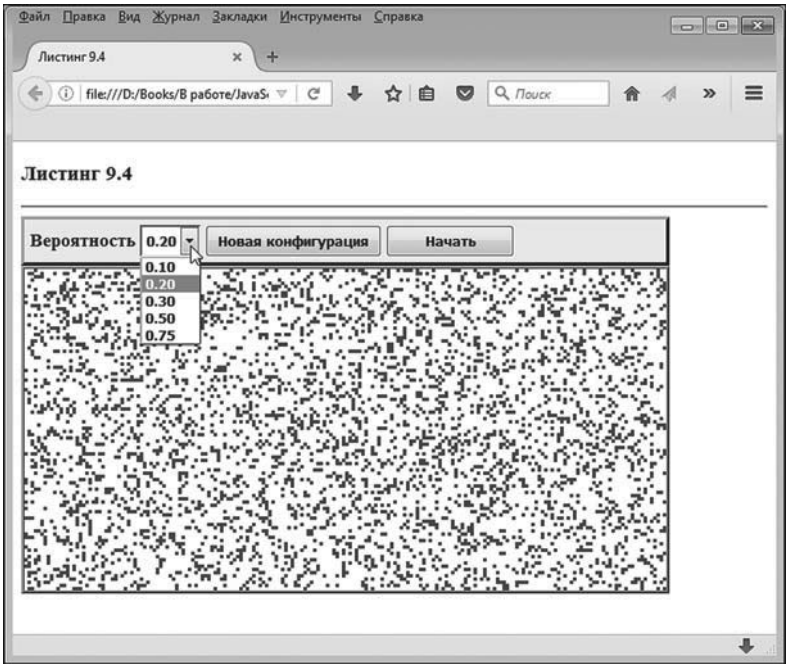

Рис. 9.15. Документ содержит графическую область и форму с элементами управления

По умолчанию, когда документ только загружается, начальная конфигурация клеток рассчитывается при значении 0.2 для доли живых клеток. На рис. 9.16 показано, как может выглядеть начальная конфигурация живых клеток, если их доля составляет величину порядка 0.5.

Если щелкнуть по кнопке **Начать**, то запускается анимация, в результате которой последовательно отображаются поколения клеток. При этом раскрывающийся список и кнопка **Новая конфигурация** блокируются (становятся неактивными), а кнопка **Начать** меняет название на **Остановить**. Ситуация проиллюстрирована на рис. 9.17.

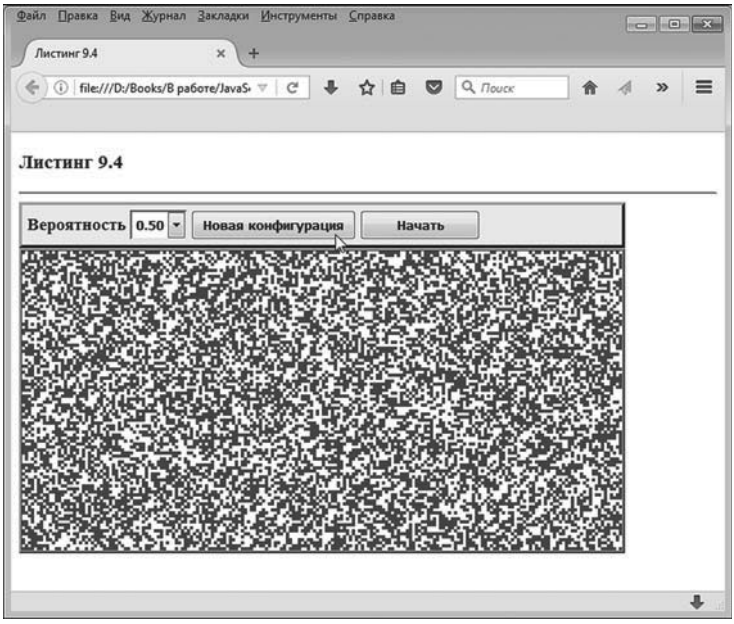

Рис. 9.16. Изменение значения в раскрывающемся списке приводит к автоматическому пересчету начальной конфигурации клеток

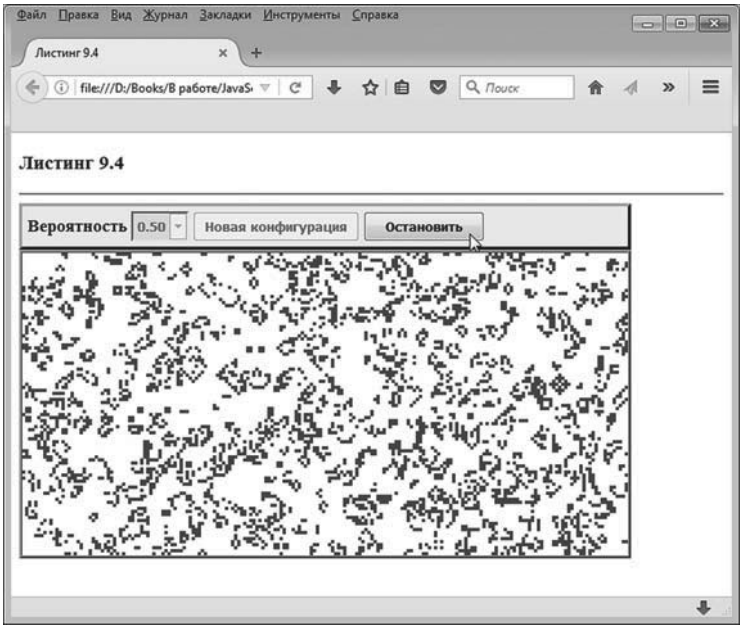

Рис. 9.17. После щелчка по кнопке **Начать** она меняет название на **Остановить**, раскрывающийся список и кнопка **Новая конфигурация** блокируются, а в графической области отображается анимация

Если щелкнуть по кнопке **Остановить**, то анимация останавливается, кнопка **Остановить** меняет название на исходное **Начать**, а также отменяется блокировка раскрывающегося списка и кнопки **Новая конфигурация**, как это показано на рис. 9.18.

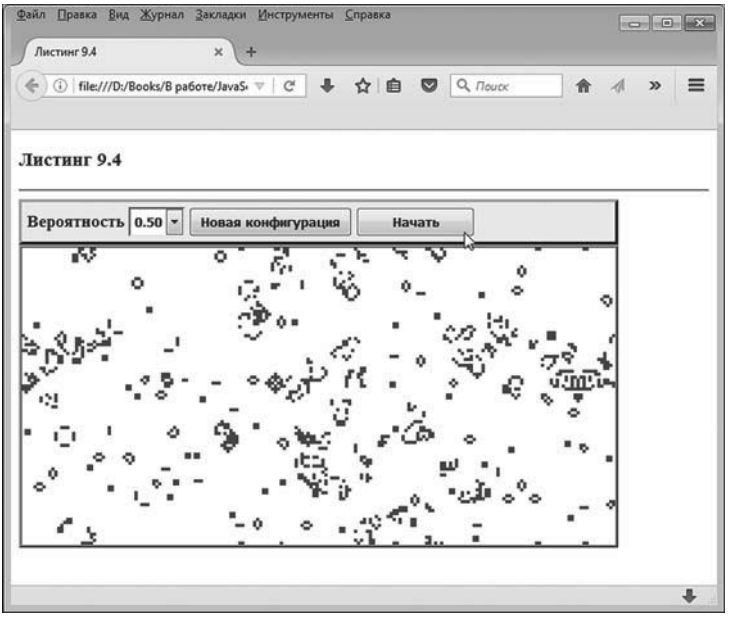

Рис. 9.18. После щелчка по кнопке **Остановить** отменяется блокировка раскрывающегося списка и кнопки **Новая конфигурация**, а процесс отображения анимации останавливается

Если еще раз щелкнуть по кнопке **Начать**, то процесс отображения поколений клеток будет продолжен с той конфигурации, на которой он был остановлен. Также пользователь может сгенерировать новую начальную конфигурацию с помощью кнопки **Новая конфигурация** и раскрывающегося списка.

НА ЗАМЕТКУ

Вне зависимости от начальной конфигурации процесс моделирования обычно сводится к тому, что в графической области образуются стационарные или периодически повторяющиеся во времени конфигурации из живых и мертвых клеток. Картинки могут быть довольно занимательными. Однако анализ этой стороны вопроса в наши планы не входит. Желающие могут обратиться к специальной литературе, посвященной данной тематике.

Теперь проанализируем программный код документа. В <body>-блоке размещается форма и графическая область. Форма реализуется с помощью <form>-блока, а графическая область представлена в виде <canvas>-блока. Последний описан достаточно скромно: для атрибута id графической области указано значение "mycanvas", а инструкцией style="border:ridge;" устанавливается трехмерная ребристая рамка вокруг области (размеры графической области вычисляются и применяются при выполнении сценария).

Форма описана со значением "myform" для атрибута id. В форме есть текст, раскрывающийся список и две кнопки. Раскрывающийся список содержит пять пунктов со значениями вероятности.

Значения атрибута value пунктов списка фактически совпадают с отображаемым текстом. Доступ к списку из сценария осуществляется по значению "prob" атрибута id.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

В блоке описания стилей для раскрывающегося списка формы myform заданы такие параметры, как высота, составляющая 30 пикселей (инструкция height:30px), серебристый цвет для рамки (инструкция border-color:silver) и жирный шрифт для отображения содержимого (инструкция font-weight:bold).

Что касается самой формы myform, то ее стиль устанавливает высоту в 30 пикселей (инструкция height:30px), выступающую трехмерную рамку (инструкция border-style:outset), внутренние отступы в 5 пикселей с каждой стороны (инструкция padding:5px), светло-серый цвет для фона (инструкция background-color:#f0f0f0) и размер в 17 пикселей для шрифта (инструкция font-size:17px).

Кнопки в форме реализованы через <input>-блоки со значением "button" для атрибута type. Кнопка для генерирования начальной конфигурации описана со значением "conf" для атрибута id. Значение "Новая конфигурация" атрибута value кнопки определяет ее название. Для кнопки запуска/остановки вычислений значение атрибута id равно "start". Название для кнопки (атрибут value) не указано. Оно определяется при выполнении сценария. Инструкция style="width:120px;" задает фиксированную ширину в 120 пикселей для кнопки. Такое «персональное внимание» обусловлено тем, что название кнопки меняется в процессе выполнения сценария. Если ширина кнопок не указана явно, то она подбирается автоматически в соответствии с длиной названия кноп-

ки. Чтобы ширина кнопки не изменялась при изменении ее названия, мы зафиксировали ширину кнопки.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

Стиль кнопок, размещаемых в форме myform, предусматривает использование жирного шрифта (инструкция font-weight:bold) и высоту в 30 пикселей (инструкция height:30px).

Использованный в документе сценарий содержит несколько глобальных переменных и вспомогательных функций, с помощью которых выполняются операции по расчету конфигураций клеток и представлению полученных результатов в графическом виде. В сценарии реализуется следующая идея. Мы отождествляем массив клеток с двумерным числовым массивом. Живой клетке соответствует значение 1, а мертвой клетке соответствует значение 0. На начальном этапе в соответствии с настройками системы (значение вероятности в раскрывающемся списке) задается начальная конфигурация клеток: двумерный массив из нулей и единиц. Этот массив используется для расчета следующего поколения клеток, которое реализуется в виде нового двумерного массива из нулей и единиц. Затем генерируется еще одно поколение клеток и так далее. Каждый раз мы имеем дело с двумерным числовым массивом. Графическое представление результата выполняется достаточно просто: если в двумерном числовом массиве некоторый элемент равен единице, то соответствующая клеточка в графической области отображается синим цветом. В противном случае клеточка отображается фоновым (белым) цветом.

Данный подход удобен еще и в плане подсчета живых соседних клеток для заданной клетки. Для этого достаточно в двумерном массиве подсчитать сумму значений соседних элементов: поскольку мертвой клетке соответствует элемент с нулевым значением, а живой клетке соответствует элемент с единичным значением, то сумма элементов совпадает с количеством живых клеток среди этих элементов.

Итак, в сценарии объявляется глобальная переменная timerID. В эту переменную записывается результат вызова функции setInterval(), с помощью которой запускается процесс последовательного отображения конфигураций клеток. Через переменную timerID мы будем прерывать процесс вычислений.

Переменные Nx и Ny со значениями соответственно 200 и 100 определяют количество столбцов и строк в массиве клеток. Переменная dz со значением 3 определяет размер квадрата в графической области, который отождествляется с клеткой. Другими словами, каждая клетка в графической области отображается в виде квадрата со стороной dz.

Группа переменных используется для записи в них ссылок на объекты элементов документа: в переменную list записывается ссылка на раскрывающийся список, в переменные start и conf записываются ссылки на кнопки запуска/остановки вычислений и выбора начальной конфигурации соответственно, переменная cnv используется для записи ссылки на графическую область, а переменная ctx нужна для записи ссылки на объект графического контекста. Ссылка на массив с числовыми значениями, который мы отождествляем с клетками, записывается в переменную A. Еще две глобальные переменные nameA со значением "Начать" и nameB со значением "Остановить" определяют названия для кнопки запуска/остановки вычислений.

Для вычисления начального распределения живых и мертвых клеток используется функция init(). Ее аргумент, обозначенный как B, отождествляется с массивом, который в процессе выполнения функции следует заполнить нулями и единицами.

В теле функции объявляются локальные переменные i, j и p. Первые две используются в операторах цикла как индексные переменные, а в переменную p командой p=list.value записывается значение, выбранное в раскрывающемся списке. Далее с помощью вложенных операторов цикла перебираются элементы массива B, и при заданных индексах i и j выполняется условный оператор. В условном операторе проверяется условие Math.random()<p. Слева от оператора сравнения указано выражение Math.random(), результатом которого является случайное число в диапазоне возможных значений от 0 (включительно) до 1 (не включая). Если условие истинно, то командой B[i][j]=1 элементу присваивается единичное значение (клетка живая). В противном случае командой B[i][j]=0 элементу присваивается нулевое значение (клетка мертвая).

> **НА ЗАМЕТКУ**
>
> Если B является массивом, элементами которого являются массивы (а мы используем именно такой подход), то результатом выражения B.length является количество элементов в массиве B — то есть коли-

чество строк. Значением выражения B[i].length является количество элементов в строке с индексом i.
Метод random() возвращает случайное равномерно распределенное (это важно) на интервале от 0 до 1 число. Поскольку число распределено равномерно, то вероятность того, что выражение Math.random()<p истинно, равна p. Таким образом, при присваивании значения элементу массива B он с вероятностью p получает значение 1 и с вероятностью 1-p получает значение 0.

Для подсчета живых соседних клеток предлагается специальная функция. Функция называется getState(), и у нее три аргумента: двумерный массив B, в котором выполняется подсчет, а также индексы i и j элемента (клетки), для которого подсчитываются «живые соседи». Небольшая сложность здесь связана с тем, что для элементов, расположенных у границы, соседи находятся на противоположной границе. Чтобы облегчить вычисления, мы вводим ряд локальных переменных и с помощью условных операторов присваиваем им значения.

В принципе соседние узлы для данного узла определяются просто: один или оба индекса соседнего узла отличаются на единицу (может быть больше или меньше) от соответствующего индекса заданного узла. Для внутренних узлов (тех, что не в крайней строке и не в крайнем столбце) так и есть. Но если узел находится, например, в первой строке, то его первый индекс нулевой. При вычислении первого индекса для «соседа сверху» следовало бы отнимать единицу. Но такая операция некорректна. Как ранее было условлено, мы в таком случае считаем, что соседним является элемент в том же столбце в последней строке массива. Аналогично, если исходный элемент находится в последней строке, то первый индекс «нижнего соседа» должен был бы быть на единицу больше первого индекса данного элемента. Такая операция тоже некорректна. Для элемента в последней строке соседним является элемент в том же столбце, но в первой строке (с нулевым индексом). Аналогично все происходит и для элементов в первом и последнем столбцах, только там речь идет о вычислении второго индекса.

В теле функции getState() объявляются четыре локальные переменные im, ip, jm и jp, в которые мы записываем индексы соседних элементов: первый индекс элемента сверху, первый индекс элемента снизу, второй индекс элемента слева и второй индекс элемента справа. Для вычисления значений индексов, как отмечалось выше, используются

условные операторы. После того как значения индексов вычислены, в еще одну локальную переменную r с помощью команды r=B[i][jm]+B[i][jp]+B[im][j]+B[ip][j]+B[im][jm]+B[ip][jp]+B[im][jp]+B[ip][jm] записывается сумма значений соседних элементов. Это и есть результат функции getState().

Для вычисления новой конфигурации (нового поколения клеток) предназначена функция recalc(). Аргумент функции B отождествляется с двумерным массивом, через который реализована текущая конфигурация клеток. В теле функции объявляется локальная переменная nbs, в которую впоследствии будет записываться количество живых клеток для каждой из тестируемых клеток. Также в функции создается локальный массив C, размеры и структура которого совпадают со структурой и размерами массива B, переданного аргументом функции. А именно сначала командой C=new Array(B.length) создается массив строк, а затем с помощью оператора цикла (с индексной переменной i) командой C[i]=new Array(B[i].length) создаются собственно строки массива. Для присваивания значений элементам массива запускается еще один (внутренний) оператор цикла (с индексной переменной j).

Вычисление значения элемента с индексами i и j в массиве C происходит с учетом значения соответствующего элемента в массиве B и количества «живых соседей» вокруг него. В данном случае имеет место реализация правил игры, касающихся переходов клеток между разными состояниями. Так, для элемента с индексами i и j в массиве B командой nbs=getState(B,i,j) вычисляется количество живых соседних клеток. Если текущая клетка в исходной конфигурации мертвая (истинно условие B[i][j]==0), проверяется условие nbs==3. Его истинность означает, что имеются ровно три живые соседние клетки. В таком случае в новой конфигурации исходная мертвая клетка оживает (команда C[i][j]=1). Если условие nbs==3 ложно, то в новой конфигурации клетка остается мертвой (команда C[i][j]=0).

Если в исходной конфигурации клетка живая (условие B[i][j]==0 ложно), проверяется условие (nbs==2)||(nbs==3). Истинность этого условия означает, что по соседству расположены две или три живые клетки. Если так, то живая в исходной конфигурации клетка остается живой и в новой конфигурации (команда C[i][j]=1). Если условие (nbs==2)||(nbs==3) не выполнено, то живая клетка в новой конфигурации становится мертвой (команда C[i][j]=0).

После того как массив C создан и заполнен, командой return C ссылка на него возвращается результатом функции recalc().

Для отображения картинки на основе массива, определяющего конфигурацию клеток, в сценарии описана функция show(). Аргументом функции передается массив с конфигурацией клеток (обозначен как B). При вызове функции сначала выполняется очистка графической области командой ctx.clearRect(0,0,cnv.width,cnv.height).

НА ЗАМЕТКУ

Напомним, что в переменную cnv записывается ссылка на объект графической области, а в переменную ctx записывается ссылка на объект графического контекста. Ссылки присваиваются при вызове обработчика, связанного с загрузкой документа (описывается далее). Ширина графической области определяется свойством width, а для определения высоты графической области используется свойство height.

Командой ctx.fillStyle="blue" устанавливается синий цвет для выполнения заливки области с живыми клетками. После этого начинается собственно процесс заливки. Для этого запускаются вложенные операторы цикла, в которых перебираются элементы массива B. При заданных значениях индексов i и j элемента в условном операторе проверяется условие if(B[i][j]==1, означающее (при истинности), что соответствующая клетка живая. Если это действительно так, то командой ctx.fillRect(dz*j,dz*i,dz,dz) выполняется заливка синим цветом области, соответствующей данной клетке.

НА ЗАМЕТКУ

Следует учесть, что в массиве B первый индекс определяет строку, в которой находится элемент (клетка), а второй индекс определяет столбец. Нам при рисовании картинки необходимо перевести индексы в графические координаты. Строка — это вертикальная координата, а столбец — горизонтальная. Поэтому первый индекс соответствует вертикальной координате, а второй индекс соответствует горизонтальной координате. Далее, если первый индекс равен i и каждая клетка отображается в графической области квадратом со стороной dz, то вертикальная координата левого верхнего угла такого квадрата равна dz*i. Аналогично, если второй индекс равен j, то горизонтальная координата левого верхнего угла квадрата равна dz*j.

Функция showNext() не имеет аргументов. Она предназначена для вычисления новой конфигурации и отображения картинки. В теле фун-

кции командой A=recalc(A) на основе текущего значения массива A вычисляется массив для новой конфигурации, и ссылка на этот массив записывается в переменную A. Далее командой show(A) для этой новой конфигурации отображается картинка.

Похожие операции выполняет функция config(), но только теперь вместо вычисления новой конфигурации на основе предыдущей производится генерирование случайного начального распределения клеток. А именно сначала командой init(A) задается начальное распределение живых и мертвых клеток (и эта информация записывается в массив A). Затем командой show(A) начальное распределение отображается в графической области.

Нам осталось проанализировать лишь код обработчика события, связанного с загрузкой документа. В первую очередь там вычисляются ссылки на объекты элементов в документе.

Командой cnv=document.getElementById("mycanvas") получаем ссылку на объект графической области.

Ширина графической области определяется командой cnv.width=Nx*dz, а высоту области вычисляем командой cnv.height=Ny*dz. После того как размеры геометрической области вычислены, командой document.getElementById("myform").style.width=cnv.width-10+"px" определяется ширина формы с элементами управления. Здесь мы к числовому значению дописываем суффикс "px" (размер в пикселях) и учитываем наличие внутренних отступов в 5 пикселей справа и слева.

Для получения ссылки на объект графического контекста использована команда ctx=cnv.getContext("2d"). Ссылку на объект кнопки выбора начальной конфигурации получаем с помощью команды conf=document.getElementById("conf"). Командой conf.disabled=false для данной кнопки отменяется режим блокирования.

НА ЗАМЕТКУ

Если перезагрузка документа будет выполнена в момент, когда кнопка выбора начальной конфигурации заблокирована, то кнопка может остаться в таком состоянии и после перезагрузки. Чтобы этого избежать, в обработчике события загрузки документа использована инструкция conf.disabled=false, которая в явном виде отменяет режим блокирования кнопки. Аналогичная процедура проделывается и с раскрывающимся списком.

Командой start=document.getElementById("start") вычисляется ссылка на кнопку запуска/остановки вычислений. Поскольку в HTML-коде кнопки ее название не указано, командой start.value=nameA задаем название кнопки. Ссылку на объект раскрывающегося списка получаем с помощью команды list=document.getElementById("prob"). Аналогично кнопке выбора начальной конфигурации, для списка отменяем режим блокировки. Поэтому в сценарий добавлена инструкция list.disabled=false. Командой list.selectedIndex=index задается индекс выбранного в раскрывающемся списке пункта. Предварительно локальной переменной index присвоено единичное значение (то есть в раскрывающемся списке выбран второй пункт).

Далее командой A=new Array(Ny) создается массив, и ссылка на него записывается в глобальную переменную A. Каждый элемент такого массива — строка с элементами (которые отождествляются с клетками). Но эти строки необходимо создать. Для создания строк запускается оператор цикла, в котором при заданном значении индексной переменной k командой A[k]=new Array(Nx) собственно создается строка. После создания массива его необходимо заполнить. Мы вызываем функцию config(), что приводит не только к заполнению массива, но и отображению начальной конфигурации в графической области.

Далее определяются некоторые обработчики событий. Так, командой conf.onclick=config в качестве обработчика события, связанного со щелчком по кнопке выбора начальной конфигурации, регистрируется функция config(). Проще говоря, при щелчке по указанной кнопке будет вызываться функция config(). Фактически эта же функция используется при обработке события, связанного с изменением состояния раскрывающегося списка. Правда, формально команда list.onchange=conf.onclick означает, что для события, связанного с изменением состояния раскрывающегося списка, значением присваивается ссылка на обработчик события, связанного со щелчком по кнопке выбора начальной конфигурации. Но поскольку для этого обработчика мы регистрировали функцию config(), то именно она будет вызываться при выборе нового пункта в раскрывающемся списке.

Обработчик события, связанного со щелчком по кнопке запуска/остановки вычислений, описан явно в виде анонимной функции. При обработке события определяющее значение имеет название кнопки. В условном операторе проверяется условие this.value==nameA. Ссылка this в данном случае выполняется на объект, на котором произошло со-

бытие — то есть это кнопка. Поэтому условие this.value==nameA истинно, если название кнопки совпадает с текстом, содержащимся в переменной nameA. Именно такое название кнопка получает при загрузке документа. Если так, то командой list.disabled=true блокируется раскрывающийся список, командой conf.disabled=true блокируются кнопки выбора начальной конфигурации, командой this.value=nameB изменяется название кнопки (на которой произошло событие — то есть кнопки запуска/остановки вычислений), и, наконец, командой timerID=setInterval (showNext,100) запускается процесс вычислений, при котором с интервалом 100 миллисекунд вызывается функция showNext(). Каждый раз при вызове функции showNext() на основе текущей конфигурации рассчитывается новая и для нее отображается картинка.

Если при щелчке по кнопке запуска/остановки вычислений название кнопки отлично от текста в переменной nameA (условие this.value==nameA ложно), командой list.disabled=false отменяется блокировка раскрывающегося списка, командой conf.disabled=false отменяется блокировка кнопки выбора новой конфигурации, командой this.value=nameA изменяется название кнопки запуска/остановки вычислений, а командой clearInterval(timerID) останавливается процесс вычисления новых конфигураций для распределения клеток.

Динамические рисунки

— Господин Калиостро, а как насчет портрета?
— Погодите вы, голубчик, с портретом! Дайте ему со скульптурой разобраться.

из к/ф «Формула любви»

Еще один пример, который мы рассмотрим далее, касается двух аспектов: во-первых, это создание и копирование изображений, и, во-вторых, это динамическое создание элементов документа. Прежде чем анализировать программный код, мы остановимся на функциональных возможностях соответствующего документа. На рис. 9.19 показано, как выглядит интересующий нас документ при загрузке в окно браузера.

Документ содержит большое изображение, которое разбито на равноотстоящие блоки одинакового размера. Вся область «большого» изображения выделена серой (точнее, серебристой) рамкой. Каждый блок внутри изображения также выделен рамкой такой же толщины и цве-

та. Справа от «большого» изображения имеется область, выделенная рамкой. Ее цвет и ширина такие же, как и у рамки вокруг изображения. При загрузке документа область справа от изображения пустая.

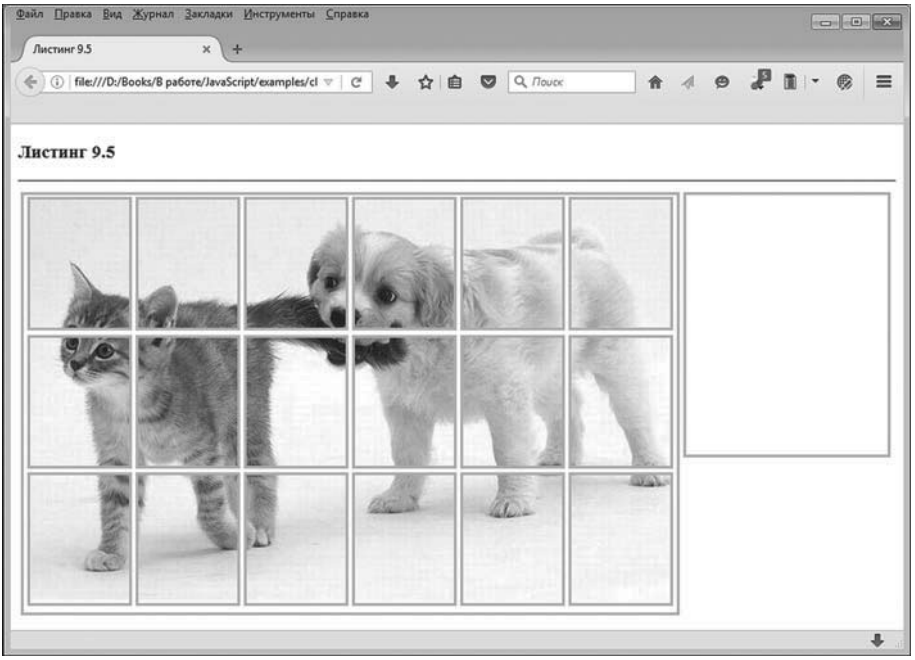

Рис. 9.19. При загрузке документа отображается разделенное на одинаковые блоки исходное изображение и пустая графическая область

Ситуация меняется, если пользователь наводит курсор мыши на одну из областей внутри «большого» изображения. В этом случае блок, на который наведен курсор мыши, выделяется рамкой более темного цвета. Такой же рамкой выделяется область справа. Еще в этой области отображается в увеличенном виде фрагмент изображения из того блока, на который наведен курсор мыши. Ситуацию иллюстрирует рис. 9.20.

При перемещении курсора мыши на другой блок с изображением его рамка выделяется темным цветом, а в области справа отображается увеличенное изображение из этого блока. При этом рамка вокруг того блока, на который курсор мыши был наведен ранее, отображается стандартным цветом (как и для прочих не выделенных блоков). На рис. 9.21 показана ситуация, когда в области «большого» рисунка выделен (наведением курсора мыши) другой блок по сравнению с тем, какой блок был выделен ранее (см. рис. 9.20).

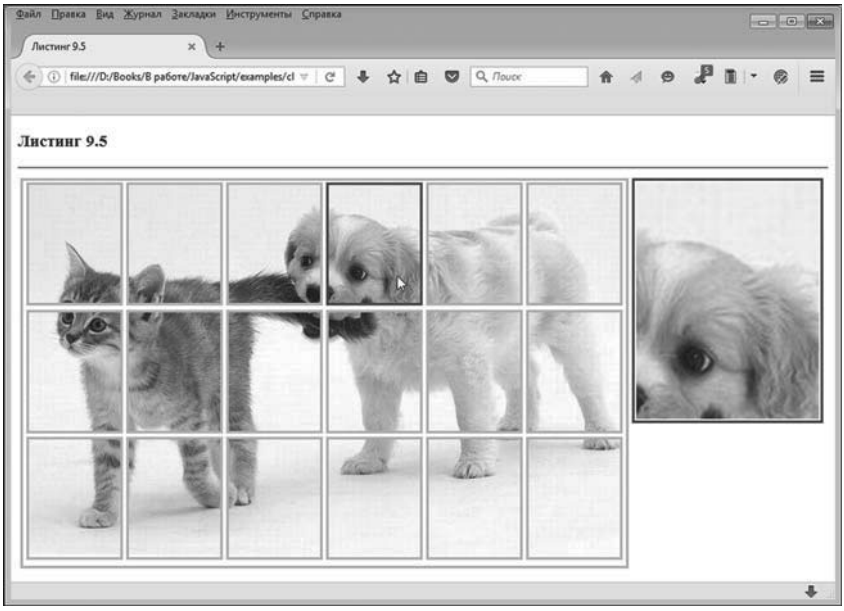

Рис. 9.20. При наведении курсора на блок с изображением он выделяется рамкой, а соответствующее изображение отображается в увеличенном виде в области справа

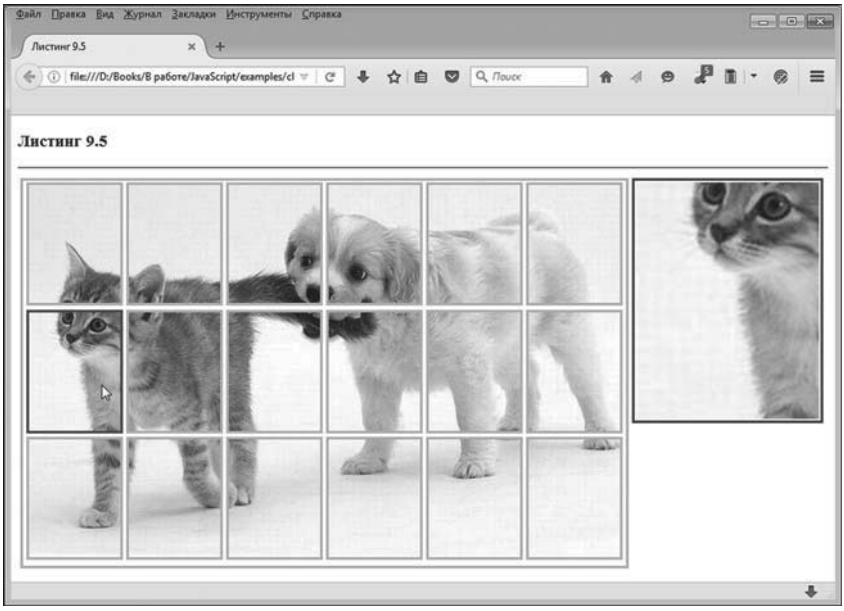

Рис. 9.21. При перемещении курсора мыши в графической области в правой части документа отображается в увеличенном виде блок, на который в данный момент наведен курсор мыши

Если курсор мыши переместить за пределы области с «большим» изображением, то область справа будет очищена, и документ возвращается к своему начальному состоянию, как при загрузке (см. рис. 9.19).

В листинге 9.5 представлен программный код документа, который рассматривался выше.

Листинг 9.5. Динамические рисунки

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Листинг 9.5</title>
<!-- Начало сценария -->
<script type="text/javascript">
  // Количество строк и количество столбцов, на которые
  // разбивается исходное изображение:
  var rows=3,cols=6
  // Полная ссылка на файл с изображением:
  var file="../images/pets.jpg"
  // Коэффициент масштабирования для отображения
  // отдельного фрагмента изображения:
  var R=2
  // Переменные для записи ссылок на массив графических
  // областей и на объект графической области, в которой
  // отображается увеличенный фрагмент изображения:
  var Cnv,outCnv
  // Переменные для записи ссылок на массив объектов
  // графических контекстов и на объект графического
  // контекста области, в которой отображается фрагмент
  // исходного изображения:
  var Ctx,outCtx
  // Переменная для записи ссылки на таблицу, содержащую
  // ячейки с графическими областями:
  var mytab
  // Переменная для записи ссылки на объект исходного
```

```
// изображения:
var myimg
// Обработчик события, связанного с загрузкой документа:
window.onload=function(){
  // Создание объекта изображения:
  myimg=new Image()
  // Ссылка на файл с изображением:
  myimg.src=file
  // Получение ссылки на таблицу для размещения
  // ячеек с графическими областями:
  mytab=document.getElementById("mytable")
  // Получение ссылки на графическую область,
  // предназначенную для отображения фрагмента
  // исходного изображения:
  outCnv=document.getElementById("output")
  // Получение ссылки на объект графического контекста
  // области, предназначенной для отображения фрагмента
  // исходного изображения:
  outCtx=outCnv.getContext("2d")
  // Создание массива для объектов
  // графических областей:
  Cnv=new Array(rows)
  // Создание массива для объектов графических
  // контекстов графических областей:
  Ctx=new Array(rows)
  // Перебор строк в массивах:
  for(var i=0;i<rows;i++){
    // Добавление в таблицу строки:
    mytab.insertRow(i)
    // Создание строки для массива объектов
    // графических областей:
    Cnv[i]=new Array(cols)
    // Создание строки для массива объектов
    // графических контекстов:
```

```
Ctx[i]=new Array(cols)
// Перебор элементов в строке:
for(var j=0;j<cols;j++){
  // Добавление в строку таблицы новой ячейки:
  mytab.rows[i].insertCell(j)
  // Создание элемента массива - объекта
  // графической области:
  Cnv[i][j]=document.createElement("canvas")
  // Создание элемента массива - объекта
  // графического контекста:
  Ctx[i][j]=Cnv[i][j].getContext("2d")
  // Добавление графической области
  // в ячейку таблицы:
  mytab.rows[i].cells[j].appendChild(Cnv[i][j])
  // Обработчик события, связанного с наведением
  // курсора мыши на область ячейки таблицы:
  mytab.rows[i].cells[j].onmouseover=function(){
    // Локальные индексные переменные:
    var m,n
    // Вычисление индекса строки, в которой
    // находится ячейка:
    m=this.parentElement.rowIndex
    // Вычисление индекса столбца, в котором
    // находится ячейка:
    n=this.cellIndex
    // Цвет рамки вокруг графической области,
    // связанной с ячейкой:
    Cnv[m][n].style.borderColor="#505050"
    // Цвет рамки вокруг графической области,
    // предназначенной для отображения фрагмента
    // исходного изображения:
    outCnv.style.borderColor="#505050"
    // Отображение фрагмента исходного
    // изображения:
```

```
        outCtx.drawImage(Cnv[m][n],0,0,Cnv[m][n].width,Cnv[m][n].height,0,0,outCnv.
        width,outCnv.height)
      }
      // Обработчик события, связанного
      // с перемещением курсора мыши за пределы
      // области ячейки:
      mytab.rows[i].cells[j].onmouseout=function(){
        // Локальная переменная для записи ссылки
        // на графическую область, размещенную
        // внутри ячейки таблицы:
        var cnv
        // Получение ссылки на графическую область
        // внутри ячейки таблицы:
        cnv=this.getElementsByTagName("canvas")[0]
        // Цвет рамки вокруг графической области
        // внутри ячейки таблицы:
        cnv.style.borderColor="silver"
        // Цвет рамки вокруг графической области, в
        // которой отображается фрагмент
        // исходного изображения:
        outCnv.style.borderColor="silver"
        // Очистка графической области,
        // предназначенной для отображения фрагмента
        // исходного изображения:
        outCtx.clearRect(0,0,outCnv.width,outCnv.height)
      }
    }
  }
  // Обработчик события, связанного с загрузкой
  // исходного изображения:
  myimg.onload=function(){
    // Локальные переменные для записи значений ширины
    // и высоты изображений, отображаемых внутри
    // ячеек таблицы:
```

```
    var width,height
    // Вычисление ширины изображения в ячейке:
    width=Math.floor(this.width/cols)
    // Вычисление высоты изображения в ячейке:
    height=Math.floor(this.height/rows)
    // Высота графической области, предназначенной
    // для отображения отдельного фрагмента
    // исходного изображения:
    outCnv.height=Math.round(R*height)
    // Ширина графической области, предназначенной
    // для отображения отдельного фрагмента
    // исходного изображения:
    outCnv.width=Math.round(R*width)
    // Перебор строк в массиве графических областей:
    for(var i=0;i<Cnv.length;i++){
      // Перебор элементов в строке массива:
      for(var j=0;j<Cnv[i].length;j++){
        // Ширина графической области в ячейке:
        Cnv[i][j].width=width
        // Высота графической области в ячейке:
        Cnv[i][j].height=height
        // Сплошная рамка вокруг области:
        Cnv[i][j].style.border="solid"
        // Толщина рамки для области:
        Cnv[i][j].style.borderWidth="3px"
        // Цвет рамки для области:
        Cnv[i][j].style.borderColor="silver"
        // Отображение фрагмента исходного
        // изображения внутри ячейки таблицы:
        Ctx[i][j].drawImage(this,j*width,i*height,width,height,0,0,width,height)
      }
    }
  }
}
```

```
</script>
<!-- Завершение сценария -->
<!-- Описание стилей -->
<style type="text/css">
  /* Стиль таблицы mytable */
  #mytable{
    padding:2px;
    border-style:solid;
    border-width:3px;
    border-color:silver;
    border-spacing:0px;
  }
  /* Стиль ячейки таблицы mytable */
  #mytable td{
    padding:0px 2px;
  }
  /* Стиль графической области для вывода отдельного
  фрагмента изображения */
  #output{
    border-style:solid;
    border-color:silver;
    border-width:3px;
    padding: 2px;
  }
</style>
<!-- Завершение описания стилей -->
</head>
<body>
  <h3>Листинг 9.5</h3><hr>
<!-- Внешняя таблица -->
<table>
  <!-- Строка внешней таблицы -->
  <tr>
    <!-- Первая ячейка в строке внешней таблицы -->
```

```
    <td style="vertical-align:top;">
      <!-- Внутренняя таблица mytable -->
      <table id="mytable"></table>
    </td>
    <!-- Вторая ячейка в строке внешней таблицы -->
    <td style="vertical-align:top;">
      <!-- Графическая область для отображения фрагмента
           исходного изображения -->
      <canvas id="output"><canvas>
    </td>
  </tr>
</table>
</body>
</html>
```

Анализ начнем с <body>-блока документа. Его основу составляет таблица (будем называть ее *внешней*). Таблица состоит из одной строки, в которой две ячейки. В первой ячейке размещена таблица (будем называть ее *внутренней*), а во второй ячейке размещается графическая область. Внешняя таблица формирует структуру документа: первая ячейка нужна для отображения «большого» изображения, а во второй ячейке размещается графическая область для отображения увеличенного фрагмента изображения.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

В описании ячеек использована инструкция style="vertical-align:top;", благодаря чему выравнивание вдоль вертикали в ячейках выполняется по верхнему уровню. Благодаря этому графическая область и основное изображение по верхней границе находятся на одной линии.

Внутренняя таблица в первой ячейке описана с атрибутом id, значение которого равно "mytable". Однако описание таблицы состоит всего из дескрипторов <table> и </table>. То есть структура таблицы (количество строк и столбцов) не заданы. Количество строк и столбцов в таблице определяется в сценарии. Соответственно, строки и столбцы в таблицу также добавляются в сценарии.

Графическая область во второй ячейке внешней таблицы создается с помощью <canvas>-блока. Атрибут id этого элемента имеет значение "output". Кроме перечисленных элементов <body>-блок больше ничего не содержит. Поэтому имеет смысл приступить к анализу кода сценария.

ЯЗЫК ГИПЕРТЕКСТОВОЙ РАЗМЕТКИ HTML

В документе имеется <style>-блок с описанием нескольких стилей. Для стиля внутренней таблицы (описана со значением "mytable" атрибута id) задаются внутренние отступы в 2 пикселя (инструкция padding:2px), устанавливается сплошная линия для рамки (инструкция border-style:solid), толщина рамки составляет 3 пикселя (инструкция border-width:3px), цвет рамки определяется инструкцией border-color:silver, а инструкция border-spacing:0px устанавливает нулевые отступы между рамками в таблице.

Стиль для ячейки внутренней таблицы содержит всего одну инструкцию padding:0px 2px,в которой внутренние отступы сверху и снизу установлены нулевые, а отступы справа и слева составляют величину в 2 пикселя.

Описание стиля для графической области, в которую выводится отдельный фрагмент изображения, содержит четыре инструкции, которыми задается рамка в виде сплошной линии (инструкция border-style:solid), устанавливается цвет рамки (инструкция border-color:silver), определяется толщина рамки в 3 пикселя (инструкция border-width:3px) и устанавливаются внутренние отступы в 2 пикселя со всех сторон (инструкция padding: 2px).

В сценарии используется следующий подход. Сначала в соответствии с указанным количеством строк и столбцов формируется внутренняя таблица. В каждую ячейку этой внутренней таблицы помещается графическая область. В этой графической области отображается часть исходного изображения. Само исходное изображение («большая» картинка) разбивается на такое же количество строк и столбцов, как и внутренняя таблица. Для каждой ячейки внешней таблицы регистрируются обработчики событий, связанных с наведением курсора мыши на ячейку и перемещением курсора мыши за пределы области ячейки. Также для удобства графические области, которые добавляются в ячейки внутренней таблицы, объединяются в двумерный массив.

Для реализации означенного подхода в сценарии объявляются переменные rows (со значением 3) и cols (со значением 6). Значения этих

переменных определяют соответственно количество строк и столбцов во внутренней таблице. Переменная file значением содержит путь к файлу с изображением. Значение "../images/pets.jpg" этой переменной означает, что файл с исходным изображением называется pets.jpg и находится в папке images, размещенной на том же уровне иерархии, что и папка с файлом веб-документа.

ДЕТАЛИ

Напомним, что двоеточие в пути к файлу означает переход на один уровень вверх. Это следует учесть при «расшифровке» пути "../images/pets.jpg" к файлу с изображением. Допустим, что файл с веб-документом находится в некоторой папке — для определенности пускай это будет папка examples. Тогда файл с изображением pets.jpg должен находиться в папке images, причем сама папка images должна находиться в той же папке, в которой находится папка examples.

Как мы отмечали ранее, в графической области, расположенной в правой части документа, фрагмент изображения отображается в увеличенном виде. Коэффициент увеличения определяется значением переменной R (мы используем значение 2). В переменную outCnv будет записана ссылка на объект графической области, предназначенной для отображения фрагмента исходного изображения. Ссылка на объект графического контекста этой области будет записываться в переменную outCtx.

В сценарии создается два двумерных массива: один для объектов графических областей, размещенных в ячейках, а другой — для объектов графического контекста этих областей. Ссылка на первый массив будет записываться в переменную Cnv, для записи ссылки на второй массив предназначена переменная Ctx.

Наконец, нам понадобится получить доступ к внутренней таблице и объекту исходного изображения. Ссылка на объект таблицы запишется в переменную mytab, а ссылка на объект изображения запишется в переменную myimg.

Кроме объявления перечисленных выше переменных, сценарий содержит описание обработчика события, связанного с загрузкой документа. В теле обработчика командой myimg=new Image() создается объект изображения. Ссылка на созданный объект записывается в переменную myimg. Далее командой myimg.src=file для созданного изображения

определяется ссылка на файл. Таким образом, изображение связывается с конкретным файлом на диске.

Для получения ссылки на внутреннюю таблицу используем команду mytab=document.getElementById("mytable"). Ссылку на графическую область в правой части документа получаем посредством инструкции outCnv=document.getElementById("output"). Графический контекст для этой области определяется командой outCtx=outCnv.getContext("2d").

Далее наступает черед создания двумерных массивов. Массив для объектов графических областей создаем командой Cnv=new Array(rows). Командой Ctx=new Array(rows) создаем еще один массив такого же размера. Это будет массив объектов графических контекстов. Но данные массивы пока еще одномерные. Каждый элемент обоих массивов должен стать строкой. Для этого запускаем оператор цикла, в котором индексная переменная i принимает значения от 0 до rows-1 включительно. За каждый цикл командами Cnv[i]=new Array(cols) и Ctx[i]=new Array(cols) создаются строки для указанных массивов. Но перед этим командой mytab.insertRow(i) во внутреннюю таблицу добавляем новую строку.

ДЕТАЛИ

Для вставки строки в таблицу используется метод insertRow(). Метод вызывается из объекта таблицы, в которую добавляется строка. Аргументом методу передается индекс добавляемой строки в коллекции строк таблицы. Индекс i перебирает строки в массивах и одновременно определяет индекс строки, которую следует вставить в таблицу. Именно в этой строке должны находиться ячейки, в которые добавляются графические области из строки массива Cnv с индексом i.

Добавленная в таблицу строка является пустой, а созданные строки массивов не заполнены (элементам массивов в строке не присвоены значения). Поэтому запускается еще один, внутренний оператор цикла, в котором индексная переменная j пробегает значения от 0 до cols-1 включительно. В теле внутреннего оператора цикла командой mytab.rows[i].insertCell(j) в строку таблицы с индексом i добавляется новая ячейка.

ДЕТАЛИ

Коллекцию строк таблицы можно получить с помощью свойства rows объекта таблицы. Указав индекс, получаем объект соответствую-

щей строки. Поэтому инструкция mytab.rows[i] дает ссылку на объект строки с индексом i таблицы mytab. Для вставки в эту строку ячейки используем метод insertCell(). Метод вызывается из объекта строки, в которую добавляется ячейка. Аргументом методу передается индекс столбца, в который вставляется ячейка.

При заданных значениях индексов i и j командой Cnv[i][j]=document.createElement("canvas") создается элемент массива, который является объектом графической области. В этой команде мы передали методу createElement() название дескриптора "canvas". Название дескриптора определяет тип создаваемого объекта. После того как объект графической области создан, командой Ctx[i][j]=Cnv[i][j].getContext("2d") создается объект графического контекста для этой области. Ссылка на созданный объект присваивается значением элементу массива Ctx с индексами i и j.

Для добавления графической области в ячейку таблицы используем выражение mytab.rows[i].cells[j].appendChild(Cnv[i][j]). Инструкция mytab.rows[i].cells[j] является ссылкой на объект ячейки таблицы mytab, которая находится в строке с индексом i, а индекс ячейки в этой строке равен j.

ДЕТАЛИ

Свойство rows объекта таблицы возвращает ссылку на коллекцию строк, входящих в таблицу. Выражение mytab.rows[i] дает ссылку на строку таблицы mytab с индексом i. Свойство cells объекта строки возвращает ссылку на коллекцию ячеек, входящих в строку. Поэтому выражение mytab.rows[i].cells[j] представляет собой ссылку на ячейку с индексом j в строке с индексом i в таблице mytab.

Для ячеек таблицы регистрируются два обработчика событий, связанных с наведением курсора мыши на область ячейки и перемещением курсора мыши за пределы области ячейки. При описании обработчика события, связанного с наведением курсора мыши, свойству onmouseover объекта mytab.rows[i].cells[j] ячейки таблицы присваивается функция. В теле функции объявляются локальные переменные m и n. В локальные переменные мы записываем значения индексов ячейки таблицы, для которой регистрируется обработчик. Первый индекс вычисляется командой m=this.parentElement.rowIndex, для вычисления второго индекса использована команда n=this.cellIndex. В команде m=this.

parentElement.rowIndex ключевое слово this означает ссылку на объект ячейки, для которой регистрируется обработчик.

Свойство parentElement возвращает ссылку на родительский элемент для ячейки. Таким элементом является строка, содержащая ячейку. Свойство rowIndex дает значение индекса строки в таблице. В команде n=this.cellIndex свойство cellIndex дает значение индекса ячейки в строке.

ДЕТАЛИ

В теле обработчика мы фактически вычисляем и записываем в переменные m и n значения, которые на момент регистрации обработчика имеют переменные i и j соответственно. Но если мы используем в программном коде функции-обработчика эти переменные, то получим в обработчике ссылку на локальные переменные внешней функции. При вызове обработчика будут использоваться не те значения, которые были у переменных i и j на момент регистрации обработчика, а значения, которые будут у переменных после окончания выполнения операторов цикла. Как бы там ни было, вариант с использованием переменных i и j не подходит. Поэтому мы в теле обработчика вычисляем индексы ячейки на основе ссылки this на объект ячейки. Далее мы увидим, что это далеко не единственный путь решения проблемы.

После того как индексы вычислены, командой Cnv[m][n].style.borderColor="#505050" задается цвет рамки вокруг графической области, связанной с ячейкой.

Цвет рамки вокруг графической области, предназначенной для отображения фрагмента исходного изображения, определяется командой outCnv.style.borderColor="#505050". Наконец, командой outCtx.drawImage(Cnv[m][n],0,0,Cnv[m][n].width,Cnv[m][n].height,0,0,outCnv.width,outCnv.height) отображается фрагмент исходного изображения. В этой команде из объекта outCtx вызывается метод drawImage(). У метода девять аргументов.

Первый аргумент Cnv[m][n] является ссылкой на графическую область, содержащуюся внутри ячейки таблицы (на которую наведен курсор). Вообще первый аргумент метода drawImage() определяет изображение, предназначенное для отображения. В данном случае таким изображением является картинка, содержащаяся в графи-

ческой области Cnv[m][n]. Следующие два аргумента определяют точку в области, начиная с которой выполняется копирование фрагмента. Они нулевые. Это означает, что копирование фрагмента начинается с левого верхнего угла.

Четвертый и пятый аргументы определяют размеры копируемого изображения. Мы четвертым аргументом указали значение Cnv[m][n].width, равное ширине графической области внутри ячейки. Пятым аргументом передается значение Cnv[m][n].height, равное высоте графической области внутри ячейки.

Следовательно, копированию подлежит все содержимое графической области внутри ячейки. Следующие два аргумента определяют точку в графической области, в которую выводится изображение. Аргументы нулевые, и поэтому в графической области справа от «большого» изображения фрагмент отображается, начиная с левого верхнего угла. Еще два аргумента определяют ширину и высоту подобласти для вывода изображения.

В приведенной выше команде два последних аргумента outCnv.width и outCnv.height — это не что иное, как ширина и высота области справа от «большой» картинки. Поэтому фрагмент исходного изображения масштабируется по размерам этой области.

Обработчик события, связанного с перемещением курсора мыши за пределы области ячейки, описывается немного иначе (по сравнению с обработчиком события, связанного с наведением курсора мыши на область ячейки).

В рассмотренном выше обработчике мы по объекту ячейки определили индексы, задающие положение ячейки в таблице, а затем с использованием этих индексов получали доступ к объекту графической области и объекту графического контекста, которые являются элементами в глобальных массивах. Здесь мы меняем стратегию и на основе объекта ячейки получаем доступ к содержащейся в ячейке графической области напрямую, без вычисления индексов и обращения к массивам. Для этого в анонимной функции, которая присваивается значением свойству onmouseout ячейки (ссылку на которую дает выражение mytab.rows[i].cells[j]), объявляется локальная переменная cnv. Значением переменной присваивается ссылка на графическую область, размещенную внутри данной ячейки. Для этого мы используем команду cnv=this.getElementsByTagName("canvas")[0]. В этой команде

мы вызываем метод getElementsByTagName(), возвращающий коллекцию элементов. Тип элементов определяется аргументом, переданным методу. Мы передаем методу аргумент "canvas", поэтому в результате получаем коллекцию ссылок на объекты графических областей. Но пикантность ситуации в том, что метод getElementsByTagName() вызывается не из объекта документа document, а из объекта ячейки, ссылка на которую возвращается ключевым словом this. Поэтому возвращаемая коллекция состоит только из тех объектов графических областей, которые находятся в данной ячейке. А в ячейке, как мы знаем, находится только одна такая область. Чтобы получить ссылку на нее, в конце инструкции указываем в квадратных скобках нулевой индекс.

После того как ссылка на объект графической области внутри ячейки вычислена, командой cnv.style.borderColor="silver" задается цвет рамки вокруг этой графической области. Цвет рамки вокруг графической области справа от «большого» рисунка задаем командой outCnv.style.borderColor="silver". Очистка этой графической области выполняется с помощью инструкции outCtx.clearRect(0,0,outCnv.width,outCnv.height).

Еще один обработчик определяет действия, выполняемые при загрузке исходного изображения. Напомним, что ссылка на объект изображения записывается в переменную myimg. Поэтому для определения обработчика, связанного с загрузкой данного изображения, мы свойству onload объекта myimg присваиваем значением анонимную функцию. В теле функции объявляются локальные переменные width и height, значения которых вычисляются командами width=Math.floor(this.width/cols) и height=Math.floor(this.height/rows).

Первый параметр — это ширина изображения, поделенная на количество столбцов, а второй параметр — высота изображения, поделенная на количество строк. Оба параметра округляются до целого числа и определяют соответственно ширину и высоту фрагментов, на которые разбивается исходное изображение.

НА ЗАМЕТКУ

Поскольку речь идет об описании обработчика для события, состоящего в загрузке изображения, то ключевое слово this в теле обработчика возвращает ссылку на объект изображения.

Для округления полученных значений до целых чисел используется метод round() встроенного объекта Math.

После вычисления параметров width и height они используются в командах outCnv.height=Math.round(R*height) и outCnv.width=Math.round(R*width) при вычислении ширины и высоты графической области, предназначенной для отображения отдельного фрагмента исходного изображения.

Размеры определяются просто, умножением на коэффициент масштабирования (значение переменной R) соответствующих размеров фрагмента изображения. Что касается последних, то, хотя значения переменных width и height вычислены, они еще не применены к графическим областям в ячейках таблицы. Поэтому запускаются вложенные операторы цикла, в которых для каждого набора значений индексных переменных i и j командами Cnv[i][j].width=width и Cnv[i][j].height=height задается ширина и высота графической области в ячейке, командой Cnv[i][j].style.border="solid" устанавливается сплошная рамка вокруг области. Толщина рамки для области определяется командой Cnv[i][j].style.borderWidth="3px". Цвет рамки определяем командой Cnv[i][j].style.borderColor="silver". Наконец, с помощью инструкции Ctx[i][j].drawImage(this,j*width,i*height,width,height,0,0,width,height) в графической области внутри ячейки таблицы рисуется фрагмент исходного изображения.

ДЕТАЛИ

При вызове метода drawImage() ему передается несколько аргументов. Первым аргументом указано ключевое слово this, являющееся ссылкой на объект, на котором произошло событие (ведь речь идет о программном коде обработчика события). Данный объект — это объект загруженного изображения. Следовательно, отображается именно оно, но, как мы поймем далее, не все, а только фрагмент. Следующие два аргумента определяют точку, начиная с которой отображается изображение. Напомним, что переменная j определяет индекс столбца ячейки, а переменная i определяет индекс строки ячейки, в которой находится графическая область и куда выводится фрагмент изображения. Параметр width задает ширину отдельного фрагмента изображения, а параметр height содержит значение высоты фрагмента. Поэтому левый верхний угол отображаемого фрагмента имеет координату j*width по горизонтали и координату i*height по вертикали. Именно эти значения переданы вторым и третьим аргументами методу drawImage(). То есть это координата точки, начиная с которой выполняется отображение фрагмента. Размеры отображаемого фрагмента определяются параметрами width и height. Следующие два аргумента метода нулевые. Они определяют точку в графической области (внутри ячейки), начиная с которой отображается картинка. Наконец, два последних аргумента

(указаны переменные width и height) определяют размеры подобласти для вывода изображения — то есть в данном случае фрагмент занимает всю графическую область в ячейке.

Резюме

Все свободны! Да, конвой тоже свободен. Конвой свободен!

из к/ф «Иван Васильевич меняет профессию»

В этой главе мы рассмотрели несколько примеров. Краткое резюме может быть таким.

- Язык JavaScript позволяет успешно решать самые различные задачи. Он удобен не только для проведения вычислений, но и для визуализации данных.
- Как правило, существуют разные способы решения одной и той же задачи. Выбор того или иного подхода является не менее важным аспектом решения задачи, чем составление самого программного кода.
- Для успешного овладения навыками программирования недостаточно усвоить только теорию. Нужна постоянная практика.

Заключение
НЕМНОГО О ВЕБ-ПРОГРАММИРОВАНИИ

Всё время думать одну и ту же мысль нельзя.
Это очень вредно.

из м/ф «38 попугаев»

Веб-программирование по сравнению с исторически традиционными программными подходами имеет существенные особенности. И эти особенности не сводятся к чисто техническим моментам, хотя они, конечно же, присутствуют. Типична ситуация, когда разрабатываемый программный продукт должен быть в значительной степени универсальным в плане требований к вспомогательному программному обеспечению. Если более конкретно, то, например, при разработке веб-страниц и сценариев приходится принимать в расчет, что конечный пользователь программного продукта может использовать браузеры разных типов и разных версий. Данное обстоятельство имеет вполне определенные последствия и влияет на стратегию реализации программного кода. Поэтому для успешной работы в области веб-программирования мало овладеть в совершенстве навыками собственно программирования. Необходимо четко представлять, какие программные средства популярны и востребованы на данный момент. Это подразумевает постоянное отслеживание и системный анализ новых тенденций на рынке программных продуктов.

Веб-программирование не ограничивается только языком JavaScript, хотя на сегодня данный язык один из самых перспективных. Существуют иные языки, достойные внимания. Более того, для современного стиля программирования характерно комплексное использование различных технологий, когда создание программного продукта требует применения большого количества методик и программных утилит. Здесь от программиста потребуется наличие определенного программного кругозора.

Какой вывод следует из всего этого? Нельзя замыкаться на достигнутом. Нужно постоянно расширять свои навыки, поднимать профессиональный уровень. Ясно, что процесс сложный, требующий усилий и времени. С другой стороны, на таком пути можно рассчитывать на значительный успех, поскольку востребованность веб-программистов в ближайшее время будет только расти. Так что дорогу осилит идущий. И хочется верить, что данная книга поможет читателю в полной мере реализовать себя на выбранной стезе.

Предметный указатель

Производственно-практическое издание

РОССИЙСКИЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Васильев Алексей Николаевич

JAVASCRIPT В ПРИМЕРАХ И ЗАДАЧАХ

Директор редакции *Е. Капьёв*
Ответственный редактор *Е. Истомина*
Художественный редактор *В. Брагина*

ООО «Издательство «Э»
123308, Москва, ул. Зорге, д. 1. Тел. 8 (495) 411-68-86.

Өндіруші: «Э» АҚБ Баспасы, 123308, Мәскеу, Ресей, Зорге көшесі, 1 үй.
Тел. 8 (495) 411-68-86.
Тауар белгісі: «Э»
Қазақстан Республикасында дистрибьютор және өнім бойынша арыз-талаптарды қабылдаушының
өкілі «РДЦ-Алматы» ЖШС, Алматы қ., Домбровский көш., 3«а», литер Б, офис 1.
Тел.: 8 (727) 251-59-89/90/91/92, факс: 8 (727) 251 58 12 вн. 107.
Өнімнің жарамдылық мерзімі шектелмеген.
Сертификация туралы ақпарат сайтта Өндіруші «Э»

Сведения о подтверждении соответствия издания согласно законодательству РФ
о техническом регулировании можно получить на сайте Издательства «Э»

Өндірген мемлекет: Ресей
Сертификация қарастырылмаған

Подписано в печать 21.06.2017. Формат 70x100$^{1}/_{16}$.
Печать офсетная. Усл. печ. л. 58,33.
Тираж экз. Заказ

ISBN 978-5-699-95459-9

JavaScript — язык, популярность и востребованность которого сегодня стремительно растет. В своем самоучителе Алексей Васильев, автор множества учебников по программированию для начинающих, простым и понятным языком знакомит читателей с основами ООП и веб-программирования на JavaScript. Разработанная им авторская методика позволяет в кратчайшие сроки освоить все базовые принципы этого языка и приступить к самостоятельной разработке первых проектов. Множество примеров с подробным разбором и разъяснениями автора делает процесс обучения увлекательным и эффективным.

Дополнительные материалы можно скачать по адресу:
https://eksmo.ru/files/JavaScript_Vasilyev.zip

Самое главное:

- Объектно-ориентированное программирование и JavaScript
- Принципы веб-разработки и сценариев
- Доступные разъяснения и разбор примеров
- Легкий и понятный новичкам стиль изложения
- Использована методика обучения, многократно проверенная на практике

Об авторе

Алексей Николаевич Васильев — доктор физико-математических наук, профессор кафедры теоретической физики физического факультета Киевского национального университета имени Тараса Шевченко. Автор более 15 книг по программированию на языках C++, Java, C#, Python и математическому моделированию.

Книга написана очень простым языком и будет интересна людям, не владеющим программированием, но желающим расширить свой кругозор и прикоснуться к миру веб-разработки. Она расскажет, как использовать JavaScript для создания собственных скриптов, и позволит понять назначение этого языка в окружении браузеров. Отличным дополнением к содержанию является масса примеров, которые сделают ваше обучение интересным и нескучным.

И. О. Борисов,
Zend Certified Engineer, преподаватель
центра «Специалист» при МГТУ им. Баумана

ISBN 978-5-699-95459-9