

А. А. Дуванов

WEB-КОНСТРУИРОВАНИЕ. DHTML

Санкт-Петербург
«БХВ-Петербург»

2003

УДК 681.3.06
ББК 32.973.26-018.2
Д79

Дуванов А. А.

Д79 Web-конструирование. DHTML. — СПб.: БХВ-Петербург, 2003. — 512 с.: ил.

ISBN 5-94157-334-0

Это вторая книга серии "Web-конструирование". Она пополняет навыки создания гипертекстовых страниц на базе "чистого" HTML правилами построения каскадных стилевых таблиц (CSS) и основами программирования сценариев на языке JavaScript.

Подробно рассматривается объектная модель браузера, на конкретных примерах объясняется методика построения собственных программных объектов, интегрируемых в общую среду "браузер-гипертекстовое приложение". Большое внимание уделяется структуре программного кода, дисциплине программирования, тестированию и отладке сценариев.

Для учащихся 10—11 классов и преподавателей школ

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	<i>Екатерина Кондукова</i>
Зам. главного редактора	<i>Людмила Еремеевская</i>
Зав. редакцией	<i>Григорий Добин</i>
Редактор	<i>Елена Михальчук</i>
Компьютерная верстка	<i>Татьяны Олоновой</i>
Корректор	<i>Вера Александрова</i>
Дизайн обложки	<i>Игоря Цырульников</i>
Зав. производством	<i>Николай Тверских</i>

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 25.06.03.

Формат 70×100^{1/16}. Печать офсетная. Усл. печ. л. 41,28.

Тираж экз. Заказ №

"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02 от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН
199034, Санкт-Петербург, 9 линия, 12.

Содержание

Предисловие	1
Напутствие читателю	1
Гипертекстовые учебники Роботландского университета	2
Поддержка книги (коды примеров)	2
Инструментарий занятий	3
Устройство книги	4
Благодарности	6
Введение	7
Что такое DHTML?	7
Что такое CSS?	8
ЧАСТЬ I. ОСНОВЫ CSS	11
Урок 1. CSS в HTML-коде	13
Стиль для отдельного тега	13
Стиль для отдельного HTML-файла	14
Стиль для нескольких HTML-файлов	17
Комбинирование стилей	17
За что мы любим CSS?	21
Зачетный класс	21
Задания	22
Урок 2. Обзор свойств CSS	24
Единицы измерения	24
Шрифт	25
Цвет	28
Текст	29
Поля и рамки	31
Вид	32

Зачетный класс	33
Задания	34
Урок 3. Основы построения CSS	35
Наследование	35
Контекстные селекторы	37
Классы	38
Теги <i><DIV></i> и <i></i>	42
Абсолютное позиционирование	45
<i>Z-index</i>	47
Каскадирование	48
Зачетный класс	49
Задания	52
Урок 4. Позиционирование, <i>z-index</i>	54
Позиционирование	54
Иерархия кода страницы	54
Абсолютное позиционирование	55
Относительное позиционирование	59
Смешанное позиционирование	61
<i>Z-index</i>	69
Зачетный класс	70
Задания	71
ЧАСТЬ II. ОСНОВЫ JavaScript	75
Урок 5. Первый скрипт	77
Классический HTML болен статичностью	77
JavaScript? Что это такое?	77
Как скрипт «монтируется» в HTML?	78
Зачетный класс	81
Задания	82
Дополнительный материал	82
Урок 6. Переменные, константы, выражения	87
Переменные и константы	87
Описание переменных	88
Выражения	89
Команды	89
Зачетный класс	91
Задания	94
Дополнительный материал	95

Урок 7. Ветвления.....	105
Ввод информации.....	105
Условная команда.....	106
Переключатель.....	108
Логические операции.....	110
Блок.....	110
Конструируем небольшой экзамен.....	111
Арифметические и строковые выражения как условия.....	113
Значения логического типа.....	114
Трудные условия.....	114
Ввод альтернативы.....	115
Зачетный класс.....	117
Задания.....	120
Дополнительный материал.....	120
Урок 8. Повторения	125
Команды цикла.....	125
Команды <i>break</i> и <i>continue</i>	127
Зачетный класс.....	129
Задания.....	131
Дополнительный материал.....	132
Урок 9. Функции.....	136
Описание и вызов функции.....	136
Проектирование «сверху вниз».....	138
Оборонительное программирование.....	145
Как браузер разбирается в типах переменных?.....	147
Зачетный класс.....	148
Задания.....	151
Дополнительный материал.....	152
Урок 10. Объект = методы + свойства	159
Понятие объекта.....	159
Доступ к свойствам и методам.....	161
Встроенные объекты и объекты пользователя.....	161
Зачетный класс.....	167
Дополнительный материал.....	169
ЧАСТЬ III. ОБЪЕКТНАЯ МОДЕЛЬ БРАУЗЕРА	193
Урок 11. Объекты и события браузера.....	195
Объект <i>window</i>	195
Объект <i>document</i>	198

События	200
Зачетный класс	205
Задания	205
Дополнительный материал.....	207
Урок 12. Формы (кнопки, строки ввода)	212
Принцип программного управления	212
Блок <i><Form></i>	213
Элемент <i><INPUT></i>	216
Зачетный класс	221
Задания	223
Дополнительный материал.....	225
Урок 13. Формы (флажки, радиокнопки)	231
Флажки	231
Радиокнопки	237
Зачетный класс	241
Задания	243
Дополнительный материал.....	245
Урок 14. Формы (меню, многострочное поле).....	248
Меню <i><SELECT></i>	248
Многострочное поле <i><TEXTAREA></i>	254
Зачетный класс	259
Задания	265
ЧАСТЬ IV. ПЕРВЫЙ ПРАКТИКУМ НА «КУХНЕ» СИДОРОВА	267
Урок 15. Конструирование объектов	269
Объект для хранения данных.....	269
Построение сложного объекта.....	277
Задания	288
Урок 16. Меню	289
Меню ссылок на UL	290
Меню ссылок на <i><SELECT></i>	292
Меню на стандартных кнопках	298
Меню на графических кнопках	300
Задания	305
Урок 17. Роботландский Плюсик	307
Плюсик.....	307
Техническое задание на гипертекстового Плюсика.....	309

Программирование стека	311
Протокол. Откатка. Накатка	316
Плюсик работает!	327
ЧАСТЬ V. CSS + JAVASCRIPT	341
Урок 18. Программное управление стилями	343
Принцип программного управления стилями	343
Доступ к элементам стиля	343
Построение динамического меню	347
Движение на экране	351
Задания	354
Урок 19. Управление содержимым страницы	355
Свойство <i>innerText</i>	355
Свойство <i>outerText</i>	356
Свойство <i>innerHTML</i>	356
Свойство <i>outerHTML</i>	357
«Крестики-нолики»	357
Модель _«плавающего»_ фрейма	363
Задания	366
ЧАСТЬ VI. ВТОРОЙ ПРАКТИКУМ НА «КУХНЕ» СИДОРОВА	369
Урок 20. Второй практикум на «кухне» Сидорова: Конюх	371
Техническое задание	372
Урок 21. Стиливые решения	376
Урок 22. Скрипты	389
Перетаскивание элементов по экрану	389
Модальные окна	398
Полный код Конюха	405
ЧАСТЬ VII. ПРИЛОЖЕНИЯ	407
Приложение 1. Справочник CSS	409
Единицы измерения	409
Структура справочных таблиц	410

Приложение 2. Справочник JavaScript	424
Зарезервированные слова	424
Конструкции	424
Операции	435
Встроенные функции	444
Встроенные объекты	446
Приложение 3. Справочник. Объекты и события браузера	467
<i>window</i>	467
Примеры работы с объектом <i>window</i>	469
<i>document</i>	475
<i>event</i>	482
<i>history</i>	485
<i>location</i>	486
<i>navigator</i>	489
<i>screen</i>	491
События браузера	493
Приложение 4. Коды испытателей	495
Испытатель 1. Сравнения (<i>Урок 9</i>)	495
Испытатель 2. Открытие окна (<i>Урок 11</i>)	496
Испытатель 3. Кнопка с секретом (<i>Урок 11</i>)	498
Испытатель 4. <i>value</i> и <i>defaultValue</i> (<i>Урок 12</i>)	500
Испытатель 5. Угадай пароль (<i>Урок 12</i>)	502
Литература	504

Посвящается Александру Домнину, моему верному
другу со школьных лет, и Олегу Какалину
в память о первых совместных работах

Предисловие

Это вторая книга серии «Web-конструирование»¹. Она дополняет навыки создания гипертекстовых страниц на базе «чистого» HTML (HyperText Markup Language — язык разметки гипертекста) правилами построения каскадных стилевых таблиц (CSS, Cascading Style Sheets — каскадные таблицы стилей) и основами программирования сценариев на языке JavaScript.

Технология построения динамических гипертекстов, описанная в книге, условно изображается формулой:

DHTML = HTML 4.0 + CSS + JavaScript + Internet Explorer (4 и старше)

Расшифровывается это так: внедрение в гипертекстовые документы (HTML, версия 4.0) стилевых свойств (CSS) и сценариев (JavaScript). В качестве программы просмотра документов предполагается браузер Internet Explorer (версии 4 и старше).

Эта книга, как и первая, отражает многолетний опыт Роботландского университета — сетевой школы, в которой обучаются команды (группы школьников во главе с руководителем — школьным учителем) из разных регионов России и ближнего зарубежья.

Напутствие читателю

Вам надоели статичные гипертекстовые документы? Хочется, чтобы ваша страница следила за мышью, клавиатурой, меняла экран в ответ на действия пользователя? Может быть, вы даже мечтаете создать в гипертексте игру или другую полезную программу?

Книга вам поможет. Ведь DHTML означает динамический HTML, т. е. речь пойдет о технологии, которая позволяет создавать «живые» гипертексты, практически неотличимые от обычных компьютерных программ.

Но эта книга для вас, если только вы:

- ☐ построили несколько гипертекстовых страниц, честно записывая теги HTML в текстовом редакторе;
- ☐ написали пару программ на каком-нибудь языке программирования и поняли, что это занятие вам по душе.

¹ Первая: Web-конструирование. HTML. СПб.: «БХВ-Петербург», 2003.

Что означают эти требования?

1. Если вы построили сайт в визуальном режиме, используя Word, FrontPage или другой визуальный редактор, книга вам не поможет. Для изучения DHTML нужно владеть «анатомией» гипертекстового конструирования: понимать структуру HTML-документа, правила записи тегов и их атрибутов. Можете считать себя вполне подготовленным в этом вопросе, если уже поработали с первой книгой этой серии «Web-конструирование. HTML».
2. Динамика гипертекстовых страниц и способность их к активной работе с пользователем появляются благодаря программным кодам, которые дополняют обычную разметку HTML. Книга не опирается на ваш опыт программирования, но предполагает склонность к этому роду деятельности. Если вы по принуждению написали несколько программных строк для отчета, мало что в них поняли, а главное — не почувствовали никакого душевного трепета от этих «магических заклинаний», «оживляющих» железный ящик с экраном на вашем столе, — книга не для вас. Вы только потеряете деньги и время. Посоветуйте книгу своим увлеченным друзьям.

Гипертекстовые учебники Роботландского университета

Эта книга не просто отражает опыт сетевой школы Роботландии, она является бумажной версией двух гипертекстовых учебников.

Учебники университета особенные: они больше похожи на электронные лаборатории. На их страницах можно «дергать за веревочки» многочисленных испытателей, сдавать экзамен в Зачетном классе, запускать собственный код.

Бумажная книга, конечно, лишена интерактивности, зато читать ее гораздо комфортнее, чем тексты с экрана компьютера.

Хотя бумажная книга и построена самодостаточным образом, идеальным представляется вариант, при котором в распоряжении пользователя окажутся обе версии. Бумажный носитель вы уже держите в руках, а электронные учебники можно заказать на сайте www.botik.ru/~robot или в письме автору по адресу kurs@robotland.botik.ru.

Теперь можно читать, удобно расположившись на диване, а компьютер включать только для выполнения заданий, работы с Испытателями, Зачетными классами и для создания собственных проектов.

Поддержка книги (коды примеров)

Исходные коды примеров и испытательных стендов этой книги можно скопировать с адреса: <ftp://ftp.botik.ru/rented/robot/univer/dbook.zip>.

Инструментарий занятий

Браузер: Microsoft Internet Explorer (4 или старше)

Прежде всего для практической работы необходим браузер. В нем происходит отладка программ и интерпретация построенных приложений.

Когда создается сайт для Интернета, разработчику нужно иметь под рукой как можно больше разных браузеров и уж обязательно два основных: Microsoft Internet Explorer (IE) и Netscape Navigator (NN). Ведь пользователь сети может «заехать» к нам на любом транспорте! И нужно знать, что он увидит.

К сожалению (и большой головной боли Web-программистов), браузеры разных фирм интерпретируют HTML-коды, программы на JavaScript и стили CSS по-разному. Принципы построения динамических документов остаются практически одинаковыми, но в деталях появляются досадные различия.

Разветвлять начальный курс по особенностям разных браузеров кажется не слишком правильным, так же как и ограничивать изложения одними общими идеями. Поэтому автором в качестве «рабочей лошадки» выбран браузер Microsoft Internet Explorer (версии 4 или старше).

Такой выбор сделан, исходя из следующих соображений:

- ☐ браузер IE самый популярный сегодня;
- ☐ IE не надо устанавливать пользователям Windows (а таких пользователей большинство);
- ☐ с точки зрения программирования IE (как конструктор объектной модели) устроен более логично, чем, например, NN, и тем самым на нем легче учить новичка;
- ☐ если приложение создается не для Интернета, а для работы на отдельном компьютере или в локальной сети, то его можно ориентировать только на один браузер, прописав это в сопроводительной документации;
- ☐ используя полученные в книге навыки, можно легко перейти к созданию приложений, ориентированных на другие браузеры.

Текстовый редактор

Для написания программ необходим какой-нибудь текстовый редактор. Вообще говоря, годится стандартный Блокнот Windows. Сам автор предпочитает МикроМир, а многие его коллеги с удовольствием работают в текстовом редакторе, встроенном в популярную оболочку FAR.

Устройство книги

Материал книги изложен в шести частях.

Сначала в книге излагаются стилевые свойства тегов (*Часть I. Основы CSS*), затем основы языка JavaScript (*Часть II. Основы JavaScript*), наконец, эти два механизма суммируются, что позволяет разработчику активно воздействовать на объектную модель документа и достигать необходимого уровня динамичности и интерактивности гипертекстового приложения.

Рассматривается объектная модель браузера, на конкретных примерах объясняется методика построения собственных программных объектов, интегрируемых в общую среду «браузер-гипертекстовое приложение». Большое внимание уделяется структуре программного кода, дисциплине программирования, тестированию и отладке сценариев.

Детально, начиная с технического задания, конструируются избранные исполнители системы «Роботландия» (курс информатики для малышей). Определяется дизайн изделия, строятся стилевые таблицы, разрабатываются сценарии.

Материал книги разбит на уроки. Урок — это одна тема. Он, как правило, завершается Зачетным классом (набором контрольных вопросов по изученной теме) и пакетом заданий для выполнения практических работ. Изучение книжного урока может занять один или несколько школьных уроков (если работать с книгой в школе): это зависит от уровня подготовки школьников и времени, отводимого на обучение. В некоторых уроках кроме основного (минимального) слоя присутствует дополнительный материал (выделяемый в отдельный подраздел). Он более подробно раскрывает тему и затрагивает методические вопросы. Дополнительный материал ориентирован в первую очередь на учителя, но будет полезен и любому другому заинтересованному читателю.

Часть I. Основы CSS содержит четыре урока, в которых рассматриваются стилевые свойства тегов, способы построения и внедрения стилевых определений.

Это самый простой раздел книги. В нем нет программ на JavaScript, обращений к объектной модели документа, но и этот простой стилиевой инструментарий открывает разработчику новые возможности визуального представления документа на экране, которые несвойственны «чистому» HTML.

Часть II. Основы JavaScript включает шесть уроков, в которых излагаются основы программирования и правила записи программ на языке JavaScript.

Часть III. Объектная модель браузера содержит четыре урока. Вы познакомитесь с программной моделью, которую строит браузер для каждого гипер-

текстового документа, и изучите способы воздействия на эту модель при помощи сценариев на JavaScript.

Главной иллюстрирующей темой здесь выступают способы управления элементами гипертекстовых форм (кнопки, меню, поля ввода).

Часть IV. Первый практикум на «кухне» Сидорова состоит из трех уроков, в которых молодой программист Иван Сидоров вместе со своими коллегами излагает принципы ООП (объектно-ориентированного программирования). Теория подкреплена многочисленными примерами, которые имеют практическую ценность и могут быть использованы при создании динамических гипертекстовых приложений. Разговор на «кухне» завершается построением Плюсика — роботландского исполнителя. Это достаточно сложное приложение. Подобные примеры авторы пособий обычно «анатомируют», показывая устройство готового продукта. Сидоров нарушает эту традицию. Он выступает не в роли лектора, а в роли партнера читателя. Проект создается с чистого листа, на который сначала записывается техническое задание. Показаны все этапы работы проектировщика, в том числе построение специальных отладочных стендов и интерфейсные улучшения продукта на завершающей стадии работы.

Часть V. CSS + JavaScript включает рассмотрение в двух уроках способов программного управления стилевыми свойствами тегов и содержимым гипертекстовой страницы.

Часть VI. Второй практикум на «кухне» Сидорова состоит из трех уроков, в которых детально, начиная с технического задания, конструируется роботландский исполнитель Конюх. Определяется дизайн изделия, строятся стилевые таблицы, разрабатываются сценарии. Отладка продукта выполняется на испытательных стендах, специально построенных для этой цели.

Приложения 1, 2, 3 содержат достаточно подробные (но далеко не полные) справочники по свойствам CSS, синтаксису и объектам языка JavaScript, объектной модели браузера. Помимо справочного материала, в разделе собраны многочисленные примеры, которые помогают понять смысл описываемых конструкций.

В *Приложение 4 (Коды испытателей)* собраны коды испытателей, на которые есть ссылки со страниц книги. Рабочие файлы испытателей (вместе с файлами примеров) можно скопировать с адреса: **<ftp://ftp.botik.ru/rented/robot/univer/dbook.zip>**.

В списке *Литературы* приводятся ссылки на дополнительные источники для тех, кто собирается продолжить практику конструирования динамических гипертекстов.

Благодарности

Автор с радостью вспоминает тот счастливый миг, когда его посетила идея попросить Якова Наумовича Зайдельмана, своего давнего товарища, прочесть рукопись с «критическим пером» в руке. Острота этого «фирменного» инструмента известна еще со времен совместной работы над школьным курсом «Роботландия».

Правда, количество замечаний и претензий оказалось больше, чем автор ожидал, и ему пришлось заново переписывать отдельные фрагменты книги, придумывать новые учебные примеры, переделывать программные коды.

Одним словом, Яков Наумович нагрузил автора дополнительной работой (с которой автор едва справился к сроку сдачи книги в издательство), и читатели должны почувствовать большую симпатию к этому человеку, ибо он непримиримо стоял на страже их интересов.

Введение

Что такое DHTML?

Суть построения «живых» гипертекстовых приложений достаточно проста. Браузер как интерпретатор HTML-кода реализует две функции: визуальную и моделирующую. Визуальное действие очень заметно: браузер отображает документ на экране. Но на этом работа не заканчивается. Браузер строит в памяти компьютера программную модель документа, в которой каждому тегу из HTML-файла соответствует отдельный объект.

Посмотрите на схему (рис. 1). Построение объектной модели документа (она носит иерархический характер) браузер начинает с объекта `window`. В этом объекте описаны свойства окна, в котором показан документ. В состав объекта `window` входит объект `document`, который соответствует контейнеру `<HTML>...</HTML>`. Объект `document` включает в себя (среди прочих) объект-массив `images`, который содержит информацию обо всех тегах `<IMG>` исходного HTML-файла. В частности, первая картинка описывается в этом массиве объектом `images[0]`, вторая — объектом `images[1]` и так далее.

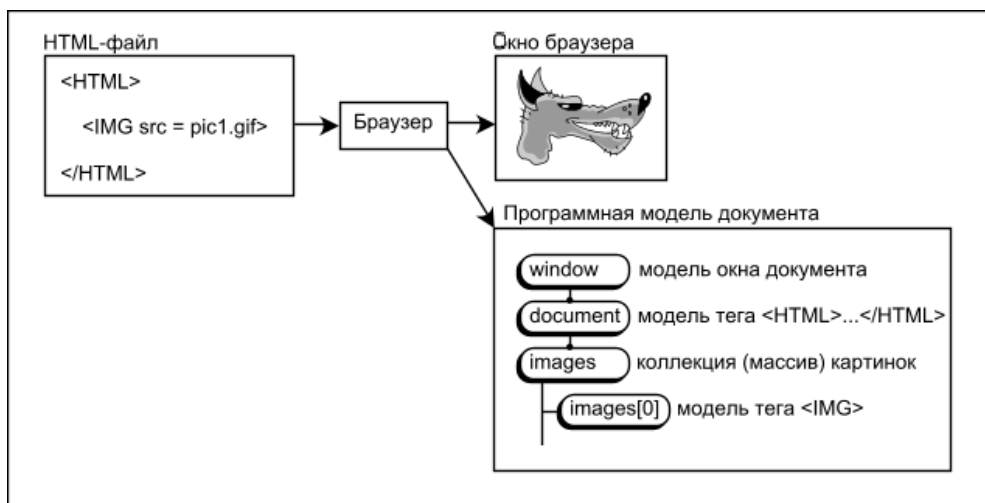


Рис. 1. Браузер строит изображение и объектную модель документа

Документ на экране станет «живым», если в HTML-коде предусмотреть не только описание тегов, но и воздействия на объектную модель при возникновении того или иного события. Можно, например, при наступлении события «курсор мыши над картинкой» изменить в объекте `images[0]` свойство `src` с `pic1.gif` на `pic2.gif` (то есть заменить одну картинку другой). И мы увидим, как волк сердится на компьютерную мышь (рис. 2).

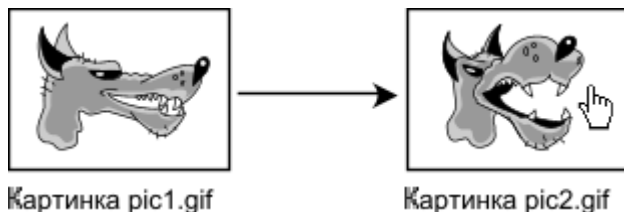


Рис. 2. Замена изображения во время просмотра документа

Программные коды, оживляющие гипертекст (их называют «скриптами», или «сценариями»), записываются на специальных языках программирования. Самым популярным языком, коды которого можно встраивать в HTML-тексты, является JavaScript. Именно этот язык рассмотрен в книге и именно он использован в ней как инструмент создания динамических интерактивных приложений.

Что такое CSS?

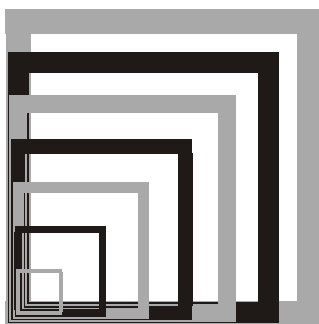
CSS — Cascading Style Sheets (каскадные таблицы стилей) — это средство, позволяющее отделить содержание гипертекстовых страниц от формы их представления на экране, а в паре со сценариями — достичь такой динамичности и интерактивности, при которых гипертекстовое приложение становится практически неотличимо от мультимедийного продукта, созданного в профессиональных программистских оболочках.

Технология CSS допускает, например, абсолютное позиционирование, т. е. позволяет выводить элементы на экран, используя реальные координаты. Программное изменение координат в объектной модели вызывает изменение положения элемента на экране.

Наряду с двумя координатами x и y при помощи CSS можно задавать еще и третью координату — z -индекс. Третья координата определяет номер слоя, в который помещается элемент. Таким образом, при движении в многослойном пространстве одни элементы могут проходить над или под другими.

Но даже при проектировании статичных гипертекстов технология CSS весьма полезна:

- ❑ CSS позволяет сократить объем HTML-файла за счет исключения из кода страницы тегов и атрибутов, ответственных за визуальное представление элементов на экране (теги: `<FONT>`, `<CENTER>`, `<B>`, `<I>`, `<S>`, `<U>` и т. д. ; атрибуты: `align`, `color`, `bgcolor`, `size`, `width`, `height` и т. д.);
- ❑ CSS предлагает гораздо больше средств для управления внешним видом элемента, чем обычные теги и атрибуты HTML;
- ❑ разработчик получает возможность менять внешний вид сотни гипертекстовых страниц, внося правки только в один файл с описаниями стилевых свойств.



Часть I

ОСНОВЫ CSS

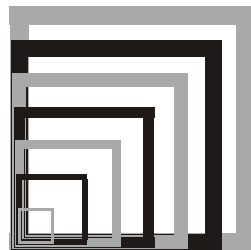
Урок 1. CSS в HTML-коде

Урок 2. Обзор свойств CSS

Урок 3. Основы построения CSS

Урок 4. Позиционирование, *z-index*

Урок 1



CSS в HTML-коде

Стиль для отдельного тега

CSS — Cascading Style Sheets (каскадные таблицы стилей) — это средство, позволяющее задавать различные визуальные свойства HTML-тегам.

Можно указать, например, как должен выглядеть на экране конкретный абзац `<P>`:

```
<P style="font-size:1.5cm; color:green">
```

К этому абзацу применено стилевое определение.

Стиль задается атрибутом **style**. В приведенном примере браузеру дано указание вывести абзац зелеными буквами размером в 1.5 сантиметра. Силевые определения задаются при помощи указаний (рис. 1.1)

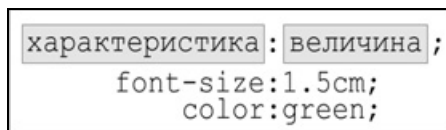


Рис. 1.1. Формат стилевых определений

Указания отделяются друг от друга символом «;».

Атрибут **style** можно использовать практически в каждом теге, задавая его специфические свойства. Посмотрите еще один пример внедрения стилей в теги.

Страница на рис. 1.2 построена таким кодом (листинг 1.1).

Листинг 1.1

```
<HTML>  
<HEAD>  
  <META http-equiv="Content-Type"
```

```
content="text/html; charset=windows-1251">
<TITLE>Стиль для отдельного тега</TITLE>
</HEAD>
<BODY bgcolor=white text=black>
  <H2>Стиль для отдельного тега</H2>
  <HR>
  <UL>
    <LI>Это обычный текст.
    <LI style="color:red;font-size:1cm;font-style:italic">
      Красные наклонные буквы высотой &nbsp;1&nbsp;сантиметр.
    <LI style="margin-left:2cm">
      Отступ слева &nbsp;2&nbsp;сантиметра.
  </UL>
</BODY>
</HTML>
```

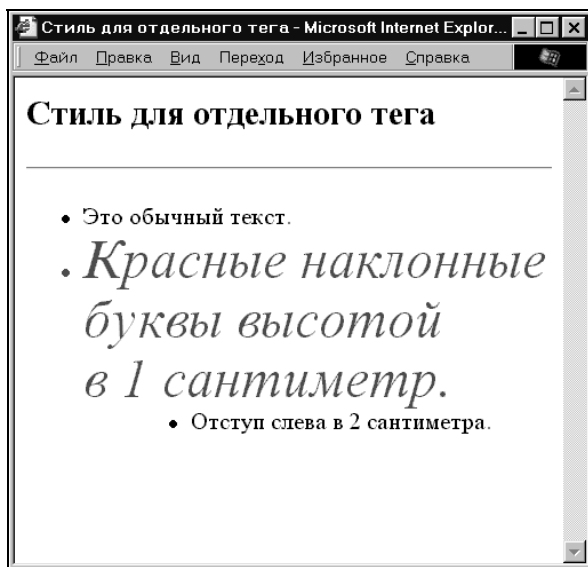


Рис. 1.2. Стиль для одного тега

Стиль для отдельного HTML-файла

Можно задавать стиль для тега или группы тегов так, чтобы определение работало на протяжении всего HTML-документа. Например, можно указать вид всех заголовков. Для этого достаточно написать определение (листинг 1.2) в головной части документа.

Листинг 1.2

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
  <TITLE>Стиль для отдельного файла</TITLE>
  <STYLE type="text/css">
    <!--
      H1, H2, H3, H4, H5, H6
      {
        text-align:right;
        color:maroon;
        font-family: "Arial Cyr", Geneva, Helvetica, sans-serif;
      }
    -->
  </STYLE>
</HEAD>
<BODY bgcolor=#DFF0D5 text=black>
  <H2>Стиль для отдельного файла</H2>
  <HR>
  <P>Это обычный текст
  <H3>Это заголовок</H3>
  <P>Это снова обычный текст
</BODY>
</HTML>
```

Браузер отображает заголовки в этом документе рубленным шрифтом цвета maroon и выравнивает их по правому краю экрана (рис. 1.3).

Такое поведение браузера соответствует инструкции (листинг 1.3).

Листинг 1.3

```
<STYLE type="text/css">
  <!--
    H1, H2, H3, H4, H5, H6
    {
      text-align:right;
      color:maroon;
      font-family: "Arial Cyr", Geneva, Helvetica, sans-serif;
    }
  -->
```

```
-->
</STYLE>
```

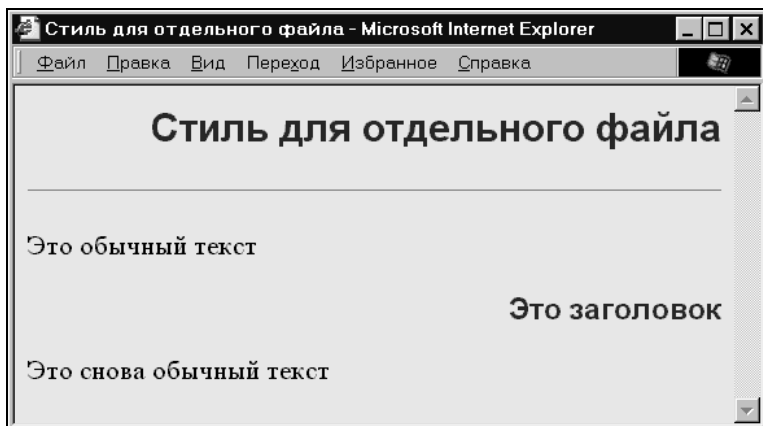


Рис. 1.3. Стиль для отдельного файла

Стилевые определения (*селекторы*) располагаются внутри блока `<STYLE>...</STYLE>` и «запаковываются» в HTML-комментарий (для браузеров, которые не поддерживают CSS).

Стилевое определение имеет вид:

Имя тега (или имена тегов через запятые)

```
{
    характеристика:величина;
    ...
    характеристика:величина;
}
```

В приведенном выше примере использованы три характеристики:

<code>text-align:right;</code>	—	задает выравнивание по левому краю;
<code>color:maroon;</code>	—	задает цвет maroon;
<code>font-family:"Arial Cyr", Geneva, Helvetica,sans-serif;</code>	—	задает рубленый шрифт.

Заголовки будут выводиться шрифтом Arial Cyr, если, конечно, этот шрифт есть в компьютере пользователя. Если шрифта нет, браузер последовательно ищет шрифты Geneva, Helvetica или, в конце концов, какой-нибудь рубленый шрифт (указание «sans-serif»). В случае полной неудачи браузер выведет текст «шрифтом по умолчанию», т. е., как правило, шрифтом Times New Roman.

Стиль для нескольких HTML-файлов

Обычной практикой является указание стилей в отдельном файле. Для таких файлов используют расширение `css`. Например, можно в файле `style.css` записать:

```
BODY {margin-left:40px;}
H1,H2,H3,H4,H5,H6
{
    text-align:right;
    color:maroon;
    font-family:"Arial Cyr",Geneva,Helvetica,sans-serif;
}
```

Для подключения этих указаний в разделе `<HEAD>...</HEAD>` HTML-файла нужно поместить ссылку:

```
<LINK rel=stylesheet type="text/css" href=style.css>
```

Такой способ расположения стилевых определений очень удобен. На один и тот же стиливой файл могут ссылаться многие HTML-документы. Изменения в этом единственном файле скажутся на внешнем виде десятка (а то и сотни) страниц.

Обратите внимание на стилевое указание `BODY {margin-left:40px;}`.

Оно задает экранный отступ слева в 40 пикселей для всего документа в целом. В силу такого определения элементы не будут «наползать» на вертикальную фоновую полосу, моделирующую левое страничное поле. Как видите, для задания страничного отступа можно обойтись без таблиц и «распорок», о которых говорилось в книге «Web-конструирование. HTML».

Комбинирование стилей

Были показаны три способа внедрения стилей в HTML-коды:

- ☐ указания в отдельном теге;
- ☐ указания в заголовке HTML-файла;
- ☐ указания в отдельном CSS-файле.

А что если комбинировать эти способы? Какой из них окажется самым «сильным» для конкретного тега?

Проделаем серию опытов.

Документ без CSS-указаний

В примере, код которого приводится ниже (листинг 1.4, рис. 1,4), заголовки выводятся черным цветом по белому.

Листинг 1.4

```
<HTML>
  <HEAD>
    <TITLE>Пример 1</TITLE>
  </HEAD>
  <BODY bgcolor=white text=black>
    <H1>Заголовок 1</H1>
    <H2>Заголовок 2</H2>
    <H3>Заголовок 3</H3>
  </BODY>
</HTML>
```

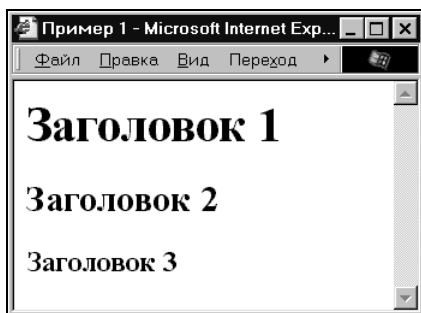


Рис. 1.4. Документ без CSS-указаний

CSS-указания в отдельном теге

Этот код (листинг 1.5) отобразит первые два заголовка черным цветом, а последний — красным.

Листинг 1.5

```
<HTML>
  <HEAD>
    <TITLE>Пример 2</TITLE>
  </HEAD>
  <BODY bgcolor=white text=black>
    <H1>Заголовок 1</H1>
```

```
<H2>Заголовок 2</H2>
<H3 style="color:red">Заголовок 3</H3>
</BODY>
</HTML>
```

CSS-указания в <HEAD>

Здесь (листинг 1.6) первые два заголовка выводятся синим цветом, а последний — красным.

Листинг 1.6

```
<HTML>
<HEAD>
  <STYLE type="text/css">
    <!--
      H1,H2,H3
      {
        color:blue;
      }
    -->
  </STYLE>
  <TITLE>Пример 3</TITLE>
</HEAD>
<BODY bgcolor=white text=black>
  <H1>Заголовок 1</H1>
  <H2>Заголовок 2</H2>
  <H3 style="color:red">Заголовок 3</H3>
</BODY>
</HTML>
```

CSS-указания в CSS-файле

Файл prim.css содержит:

```
H1,H2,H3
{
  color:green;
}
```

HTML-файл имеет вид (листинг 1.7).

Листинг 1.7

```
<HTML>
<HEAD>
  <!-- Подключение файла prim.css. -->
  <LINK rel=stylesheet type="text/css" href=prim.css>
  <STYLE type="text/css">
    <!--
      H1, H2, H3
      {
        color:blue;
      }
    -->
  </STYLE>
  <TITLE>Пример 4</TITLE>
</HEAD>
<BODY bgcolor=white text=black>
  <H1>Заголовок 1</H1>
  <H2>Заголовок 2</H2>
  <H3 style="color:red">Заголовок 3</H3>
</BODY>
</HTML>
```

Код отобразит первые два заголовка синим цветом, а последний — красным. Рассмотрим другой порядок следования тегов `<STYLE>` и `<LINK>` (листинг 1.8).

Листинг 1.8

```
<HTML>
<HEAD>
  <STYLE type="text/css">
    <!--
      H1, H2, H3
      {
        color:blue;
      }
    -->
  </STYLE>
```

```
<!-- Подключение файла prim.css. -->
<LINK rel=stylesheet type="text/css" href=prim.css>
<TITLE>Пример 5</TITLE>
</HEAD>
<BODY bgcolor=white text=black>
  <H1>Заголовок 1</H1>
  <H2>Заголовок 2</H2>
  <H3 style="color:red">Заголовок 3</H3>
</BODY>
</HTML>
```

Код отобразит первые два заголовка зеленым цветом, а последний — красным.

За что мы любим CSS?

Используя CSS можно:

- ☐ задавать поля, отступы, размер и тип шрифта, цвета текста и фона для отдельных элементов страницы (абзацев, слов, букв), оформлять красные строки, буквицы. В CSS представлен ряд свойств, с помощью которых можно создавать выпадающие меню, накладывать один элемент на другой, заставлять текст «отбрасывать тень», проявляться в «рентгеновских лучах» и использовать другие эффекты. Можно выводить элементы на экран браузера с точностью до одного пиксела, динамически перемещать их по экрану в разных слоях (одни элементы будут проходить над или под другими). Иными словами, CSS вместе с JavaScript позволяет монтировать на экране браузера приложение, практически ничем не отличающееся от компьютерной программы, написанной на профессиональном языке программирования, таком, например, как Си;
- ☐ изменять оформление целого сайта, состоящего из сотен файлов, не прикасаясь к HTML-коду, редактируя всего лишь один CSS-файл;
- ☐ уменьшать количество тегов в HTML-документе, отделяя информационное наполнение HTML-файла от его внешнего представления на экране браузера.

Зачетный класс

1. Как расшифровывается аббревиатура CSS?
2. Опишите назначение технологии CSS.
3. Каким атрибутом можно задать стиль в теге?

4. Какие из нижеприведенных записей задают красный цвет текста в абзаце?
- а) `<P color=red>`;
 - б) `<P color=#FF0000>`;
 - в) `<P color=rgb(255,0,0)>`;
 - г) `<P style="color:red">`;
 - д) `<P style="color:#FF0000">`;
 - е) `<P style="color:rgb(255,0,0)">`.
5. В каком специальном HTML-блоке `<?????>...</?????>` размещают стилевые описания?
6. Какие из нижеприведенных стилевых описаний не содержат ошибок?
- а) `P {color=red; font-size=1cm};`
 - б) `P {color=red, font-size=1cm};`
 - в) `P {color:red font-size:1cm};`
 - г) `P {color:red, font-size:1cm};`
 - д) `P {color:red; font-size:1cm}.`
7. В каком теге записывается ссылка на файл со стилевыми определениями, каким атрибутом она задается? Запишите эти имена вместо вопросительных знаков: `<???? rel=stylesheet type="text/css" ????=file.css>`
8. Какие три способа задания стилевых описаний вам известны? Сравните эти способы и укажите назначение каждого из них.

Задания

1. После внимательного изучения примеров урока ответьте на следующие вопросы:
- Какое CSS-указание главное: описанное в отдельном теге или размещенное в разделе `<HEAD>` HTML-программы?
 - Какое указание главное: описанное в разделе `<HEAD>` HTML-программы или размещенное в отдельном CSS-файле и связанное с документом при помощи тега `<LINK>`? Зависит ли результат от порядка следования этих предписаний?
2. Используя стили, постройте документ, изображенный на рис. 1.5.
- В этом документе:
- ☐ основные цвета: черный текст на белом фоне;
 - ☐ отступ всех элементов на странице слева и справа равен 2 см;

- ☐ абзацы выравниваются справа;
- ☐ заголовки выравниваются справа и записываются рубленным шрифтом красного цвета;
- ☐ термины, выделяемые курсивом, записываются зеленым цветом.

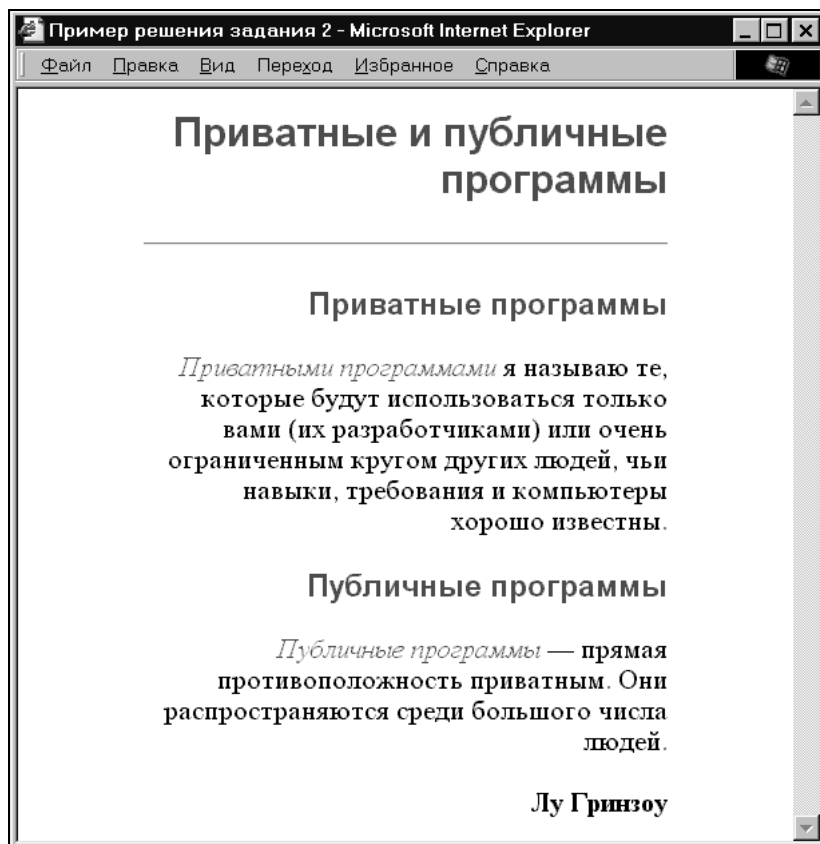
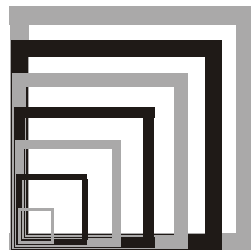


Рис. 1.5. Вид документа для задания 2

Урок 2



Обзор свойств CSS

Существует более 70 свойств, которые можно использовать для составления стилевых определений. Много! Для практической работы, конечно, необходим справочник, особенно на первых порах, и справочники приводятся в конце этой книги в *Приложениях 1—3*. Однако для начала необходимо получить общее представление о том, чем можно управлять в принципе, и выполнить несколько простых учебных заданий. Именно такому вводному (но не полному!) обзору с изрядной долей тренировочных упражнений посвящен *Урок 2* книги.

Рассмотрение свойств распределено (так же, как и в справочнике) по следующим разделам:

- ☐ шрифт;
- ☐ цвет;
- ☐ текст;
- ☐ поля и рамки;
- ☐ вид.

А начнем обзор свойств CSS с единиц измерения, которые используются в стилевых указаниях.

Единицы измерения

Единицы измерения, используемые в стилевых указаниях, представлены в табл. 2.1.

Таблица 2.1

Обозначение	Название	Пример
in	Дюйм	<code><P style="font-size:1in"></code>
cm	Сантиметр	<code><P style="font-size:1cm"></code>
mm	Миллиметр	<code><P style="font-size:1mm"></code>

Таблица 2.1 (окончание)

Обозначение	Название	Пример
px	Пиксел	<code><P style="font-size:1px"></code>
pt	Пункт	<code><P style="font-size:1pt"></code>
pc	Пики	<code><P style="font-size:1pc"></code>
%	Проценты	<code><P style="font-size:1%"></code>

Пункты и пики — это типографские единицы, которыми обозначают размер шрифта или кегль.

В Word, например, этот параметр задается в специальном окошке числами, как правило, от 8 до 72. Это величина кегля в пунктах. Один типографский пункт равен 0,375 мм. Причем это размер не самих символов шрифта, а так называемого «очка», т. е. размер по вертикали матрицы, на которой гравировается в типографии символ (литера). Этот размер больше, чем размер самой литеры. Так, в шрифте кегля 10 заглавные буквы имеют размер около 7 пунктов.

При печати книг для основного текста, как правило, выбирается кегль в 10 или 12 пунктов. Для заголовков — более крупные кегли, а для ссылок и примечаний более мелкие (обычно 8 пунктов).

Пика — более крупная единица измерения. Одна пика равна 12 пунктам.

Процентный отсчет ведется от какого-либо базового значения. В случае шрифта — от текущего размера.

Использование процентов очень привлекательно для разработчика. Когда пользователь меняет в браузере размер шрифта, размеры, заданные в стилевых указаниях в процентном виде, меняются пропорционально.

Шрифт

font-family

Это стилевое свойство задает название гарнитуры шрифта (например, Arial) или родовое имя:

- `serif` — шрифт с засечками (серифный);
- `Sans-serif` — шрифт без засечек (рубленый);
- `monospace` — моноширинный шрифт (символы имеют одинаковую ширину).

На рис. 2.1—2.3 показаны примеры шрифтов.

Это шрифт с засечками.

Рис. 2.1. Серифный шрифт

Это рубленый шрифт.

Рис. 2.2. Рубленый шрифт

Это моноширинный шрифт.

Рис. 2.3. Моноширинный шрифт

Гарнитура — это набор начертаний символов одного шрифта.

Этот набор может включать в себя прямое и курсивное начертание, различаться по степени жирности (толщине штрихов), ширине литер (сжатые, нормальные, растянутые) и кеглю.

Шрифты подразделяются на серифные (с засечками) и рубленые (без засечек). Пример шрифта с засечками — гарнитура Times, без засечек — Helvetica.

Серифные шрифты посредством засечек (серифов) в нижней части литер улучшают читаемость документа. Нижние серифы создают впечатление слитности написания слова, объединяя соседние буквы.

Рубленые шрифты, в отличие от серифных, имеют гладкие края букв и не имеют завитков.

Серифные шрифты используют для записи основного текста, а рубленые — для заголовков или мелких подписей или надписей.

Существует еще одно деление шрифтов на два класса: пропорциональные шрифты и моноширинные.

Моноширинные шрифты имитируют литеры пишущих машинок и первых матричных принтеров. В этих шрифтах ширина всех литер одинакова. Именно таким шрифтом (если шрифт не задан CSS) браузер выводит текст на экран при использовании тегов `<PRE>`, `<CODE>`, `<TT>`, `<SAMP>`, `<KBD>`.

Обычные же тексты браузер выводит пропорциональным шрифтом. В пропорциональных шрифтах каждая буква занимает столько места, сколько ей действительно нужно.

Текст, записанный пропорциональным шрифтом, более комфортен для чтения, чем текст, записанный шрифтом моноширинным. Пропорциональные шрифты используют для основного текста, а моноширинные — для специальных выделений.

Гарнитуры Times и Helvetica — пропорциональные. К моноширинным шрифтам относится гарнитура Courier.

Любая графическая операционная система имеет три набора стандартных гарнитур, для Windows это:

Times Roman — серифный шрифт;
 Arial — рубленый шрифт;
 Courier — моноширинный шрифт.

По умолчанию браузер использует шрифт Times Roman для вывода обычного текста и шрифт Courier для моноширинного.

Если гипертекстовый продукт — сайт, лучше не задавать собственную гарнитуру. Указанный шрифт на компьютере пользователя может не содержать русских букв, и текст станет похож на «древненорвежский» (рис. 2.4).

Â øðèôòà íàò ðóññêèèõ áóéâ.

Рис. 2.4. «Кракобяки» вместо букв

Для локальных гипертекстовых приложений можно использовать любые шрифты. Такое приложение передается пользователю вместе с набором шрифтов и инструкцией по их установке.

При помощи свойства `font-family` можно указать не один, а несколько шрифтов через запятые в порядке предпочтения:

h1, h2, h3, h4, h5, h6

```
{
  font-family: "Arial Cyr", Geneva, Helvetica, sans-serif;
}
```

Сначала браузер будет пытаться найти шрифт Arial Cyr, затем Geneva, потом Helvetica и, наконец, в случае полной неудачи, какой-нибудь рубленый шрифт (указание `sans-serif`).

Если название шрифта состоит из нескольких слов, оно заключается в кавычки.

font-size

Свойство `font-size` используется для задания абсолютного или относительного размера текущего шрифта.

В качестве относительного размера используется процентное задание (по отношению к размеру текущего шрифта) или ключевые слова:

larger — крупнее;
smaller — мельче.

В качестве абсолютного размера используется указание числа с одной из следующих единиц измерения: in, cm, mm, px, pt, pc. Можно использовать и такие ключевые слова:

xx-small — сверхмелкий;
x-small — очень мелкий;
small — мелкий;
medium — средний;
large — крупный;
x-large — очень крупный;
xx-large — сверхкрупный.

Следует отдавать предпочтение относительным величинам для того, чтобы пользователь, используя настройки браузера, смог установить для себя комфортный размер текста.

Цвет

Стилевые свойства поддерживают три формы указания цвета:

- ключевое слово, например, `white`;
- шестнадцатеричный RGB-код, например, `#EEE5DB`;
- десятичный RGB-код, например, `rgb(255, 64, 0)`.

Замечание

RGB-кодирование цвета подробно изложено в книге «Web-конструирование. HTML» (Урок 2).

color

Определяет цвет элемента.

background-color

Определяет цвет фона, на котором выводится элемент (рис. 2.5).

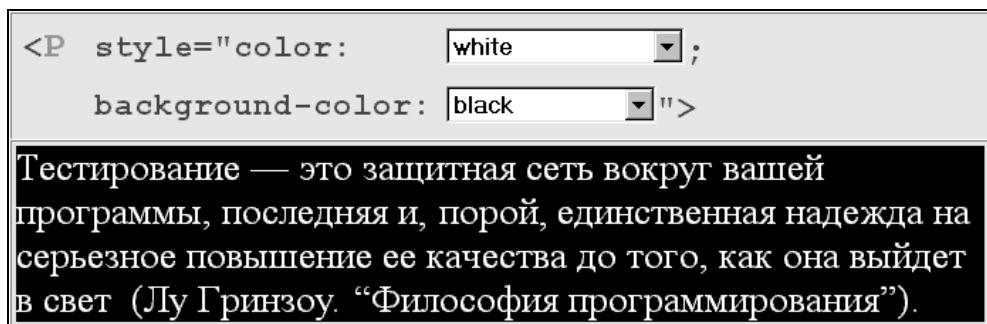


Рис. 2.5. Испытательный стенд «Цвет»

Текст

letter-spacing

Определяет добавочное расстояние между буквами. Допускается использовать ключевое слово `normal` или задавать произвольное расстояние, в том числе и со знаком минус (наложение символов). Процентная запись не допускается (рис. 2.6).

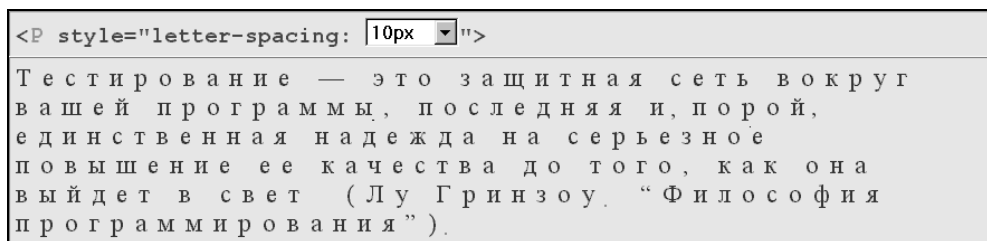


Рис. 2.6. Испытательный стенд «letter-spacing»

line-height

Определяет расстояние между строками (интерлиньяж). Можно задавать абсолютную высоту (14pt), количество строк (1.4) или проценты от текущей высоты строки (200%). Например, двойной интервал между строками задается как `line-height:2` или `line-height:200%`. Допускается задание ключевого слова `normal` (рис. 2.7).

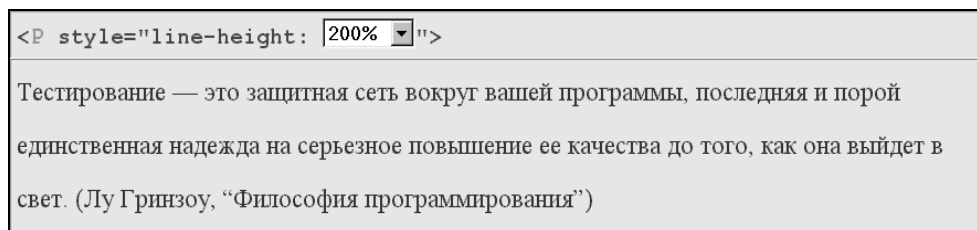


Рис. 2.7. Испытательный стенд «line-height»

text-align

Свойство определяет способ горизонтального выравнивания текста (рис. 2.8).

Допустимы следующие значения:

- left — по левому краю;
- right — по правому краю;
- center — по центру;
- justify — по ширине.

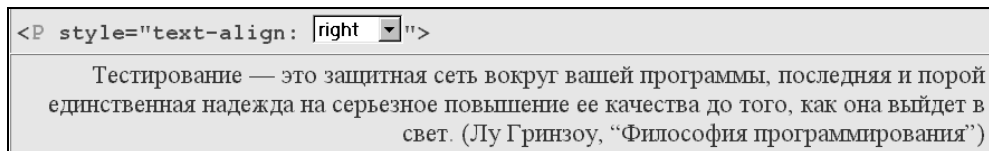


Рис. 2.8. Испытательный стенд «text-align»

К сожалению, приятным значением `justify` практически невозможно пользоваться в текстах с длинными словами (например, на русском или немецком языках). Браузер «не умеет» пока переносить слова по слогам, поэтому на экране при выравнивании `justify` часто образуются большие «дыры» (рис. 2.9).

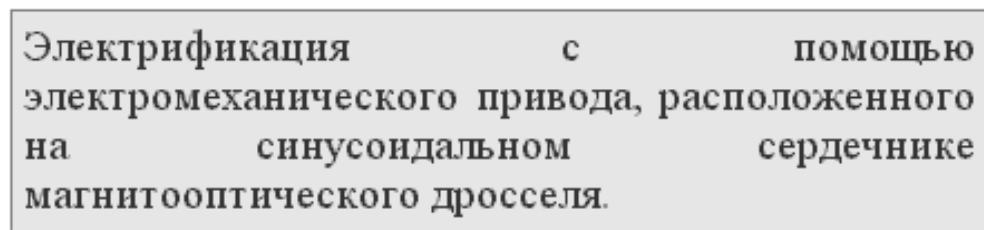


Рис. 2.9. Выравнивание `justify`

Поля и рамки

border-style

Определяет вид рамки элемента. Допустимы следующие значения:

- `none` — рамки нет (по умолчанию);
- `solid` — обычная сплошная граница;
- `double` — двойная линия;
- `groove` — «вдавленная» граница;
- `ridge` — «выпуклая» граница;
- `inset` — «вдавленный» элемент;
- `outset` — «выпуклый» элемент.

border-color

Определяет цвет рамки. Свойство работает, если задано свойство `border-style`.

border-width

Определяет толщину рамки (рис. 2.10). Свойство работает, если задано свойство `border-style`. Ширину можно указывать числовыми единицами (процентной записи нет) или при помощи следующих ключевых слов:

- `thin` — тонкая;
- `medium` — средняя;
- `thick` — толстая.

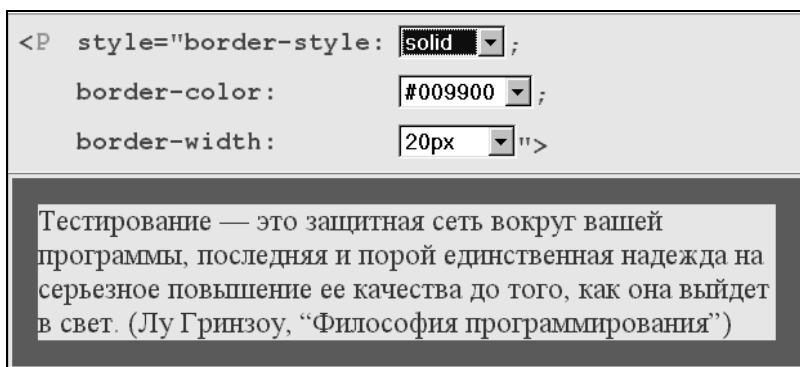


Рис. 2.10. Испытательный стенд «border»

margin

Задает ширину поля (отступа) элемента от краев блока. Допускается числовая, процентная (от ширины блока) запись или ключевое слово `auto`. В случае `auto` предполагается, что браузер будет подыскивать оптимальное значение самостоятельно.

padding

Задает расстояние между содержимым и рамкой элемента (рис. 2.11). Допускается числовая или процентная (от ширины элемента) запись.

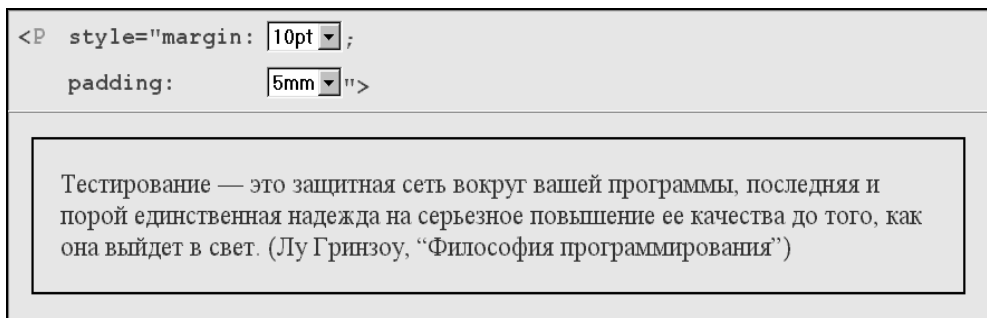


Рис. 2.11. Испытательный стенд
«margin/padding»

Вид

width

Устанавливает ширину элемента. Допускаются числовые, процентные значения и `auto`. Свойство используется для элементов `<DIV>` и `<SPAN>`. Эти теги объединяют элементы в блок. Подробнее о них рассказано в следующем уроке.

height

Устанавливает высоту элемента (рис. 2.12). Допускаются числовые значения и `auto`. Свойство используется для элементов `<DIV>` и `<SPAN>`.

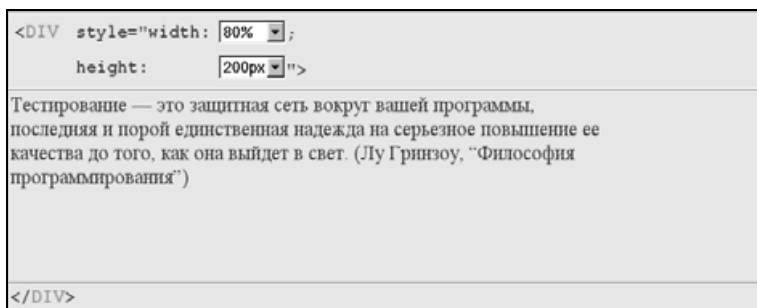


Рис. 2.12. Испытательный стенд «Вид»

Зачетный класс

1. Перечислите единицы измерения, которые используются в стилевых указаниях. Запишите обозначения этих единиц.
2. Какие единицы измерения называются типографскими? Почему?
3. Что обозначает типографский пункт?
4. Вместо знаков вопроса запишите правильные цифры:
 - а) 1 пункт = ??? мм;
 - б) 1 пика = ?? пунктам;
 - в) 1 пика = ?.? мм.
5. Каким размером шрифта обычно печатают книги?
6. Какой размер шрифта используют для примечаний?
7. По отношению к какому размеру задается процентная величина шрифта?
8. Почему использование процентных указаний для размера шрифта предпочтительнее абсолютных величин?
9. Что такое «гарнитура» шрифта?
10. Чем отличаются серифные шрифты от рубленых?
11. Чем отличаются пропорциональные шрифты от моноширинных?
12. Для каких текстов предпочтительнее серифные шрифты, а для каких рубленые?
13. Для каких текстов предпочтительнее пропорциональные шрифты, а для каких моноширинные?
14. Почему не рекомендуется для сайта задавать гарнитуру шрифта явным образом?

15. Какими свойствами CSS задаются:
- а) гарнитура, размер, цвет и фон шрифта;
 - б) добавочное расстояние между буквами;
 - в) расстояние между строками;
 - г) способ горизонтального выравнивания;
 - д) вид, цвет и толщина рамки;
 - е) ширина отступа элемента от краев блока;
 - ж) расстояние элемента до рамки;
 - з) ширина и высота элемента?

Задания

Разработайте код для вывода на экран следующего текста.

Стойкий оловянный солдатик

Было когда-то на свете двадцать пять оловянных солдатиков. Все они были сыновьями одной матери — старой оловянной ложки — и, значит, приходились друг другу родными братьями. Они были очень красивы: ружье на плече, грудь колесом, мундир красный с синим. Чудо что за солдатики!

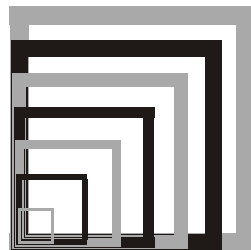
Г.-Х. Андерсен

Приведенная ниже табл. 2.2 содержит 7 заданий. Попробуйте получить красивое визуальное представление вашего документа.

Таблица 2.2

Зада- ние	Дано						
	Шрифт	Размер	Цвет	Фон	Рамка	Поля	Отступ
01	Academy	medium	Черный	Подобрать	Подобрать	Подобрать	Подобрать
02	Antiqua	large	Подобрать	Синий	То же	То же	То же
03	Arial	xx-small	Зеленый	Подобрать	—"—	—"—	—"—
04	Courier	x-small	Подобрать	Голубой	—"—	—"—	—"—
05	Garamond	x-large	Красный	Подобрать	—"—	—"—	—"—
06	New York	xx-large	Подобрать	Серый	—"—	—"—	—"—
07	Parsek	14pt	Желтый	Подобрать	—"—	—"—	—"—

Урок 3



Основы построения CSS

Наследование

HTML-код имеет иерархическую структуру. Весь документ кодируется внутри тегов `<BODY>...</BODY>`. Внутри абзаца `<P>` могут содержаться элементы, размеченные тегами `<STRONG>` и т. д.

Наследование стилей означает, что если определен какой-то стиль для тега `<BODY>`, то этот стиль будет относиться ко всем тегам документа. Соответственно, если задан стиль для некоторого тега, то все теги, расположенные внутри него в HTML-программе, также будут обладать этим стилем.

Пусть для тега `<P>` задан стиль:

```
P
{
    color: red;
    font-size:14pt;
    font-family:Arial,sans-serif;
}
```

Тогда содержимое тега `<EM>`, помещенного внутрь абзаца, будет также выведено на экран рубленным шрифтом красного цвета размером в 14 пунктов (рис. 3.1):

`<P>`

Стилевые определения обладают свойством `<EM>`наследования`</EM>`.

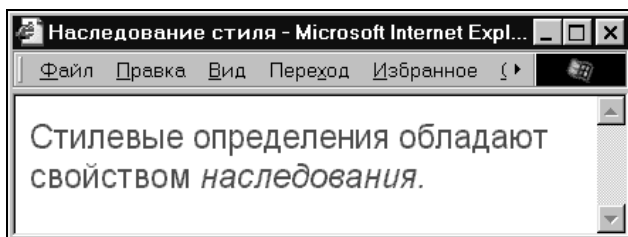


Рис. 3.1. Наследование стиля

Если нужно, например, добавить для тега `<EM>` разрядку текста, то такая модификация задается дополнительно (листинг 3.1, рис. 3.2).

Листинг 3.1

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
  <TITLE>Дополнительное стиливое указание</TITLE>
  <STYLE type="text/css">
    <!--
      P
      {color: red;
        font-size:14pt;
        font-family:Arial,sans-serif;
      }
    -->
  </STYLE>
</HEAD>
<BODY bgcolor=white text=black>
  <P>
    Стиливые определения обладают свойством
    <EM style="letter-spacing:6pt">наследования</EM>.
  </BODY>
</HTML>
```

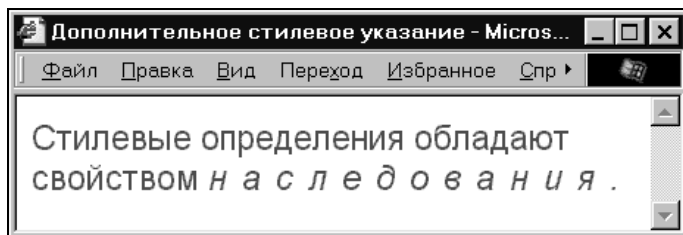


Рис. 3.2. Дополнительное стиливое указание

Как видите, содержимое тега `<EM>` выводится так же, как и содержимое тега `<P>`, внутри которого он содержится (рубленным шрифтом в 14 пунктов красного цвета), но благодаря дополнительному определению `style="letter-spacing:6pt"` между буквами появляются добавочные промежутки в 6 пунктов.

Внутри тега-потомка можно не только ввести дополнительные стилевые определения, но и переопределить стилевые свойства родителя (листинг 3.2).

Листинг 3.2

```
<HTML>
  <HEAD>
    <META http-equiv="Content-Type"
      content="text/html; charset=windows-1251">
    <TITLE>Переопределение стилевых свойств</TITLE>
    <STYLE type="text/css">
      <!--
      P
      {color: red;
        font-size:14pt;
        font-family:Arial,sans-serif;
      }
      -->
    </STYLE>
  </HEAD>
  <BODY bgcolor=white text=black>
    <P>
      Стилиевые определения обладают свойством
      <EM style="letter-spacing:6pt;color:blue">наследования</EM>.
    </BODY>
</HTML>
```

Теперь слово «наследования» будет показано синим цветом.

Контекстные селекторы

Можно написать стилевое определение, которое будет работать только при определенной комбинации вложенности тегов. Например, можно установить цвет текста в **** синим только для случая, когда этот тег расположен внутри тега **<H3>** (листинг 3.3).

Листинг 3.3

```

<HTML>
  <HEAD>
    <META http-equiv="Content-Type"
      content="text/html; charset=windows-1251">
    <TITLE> Контекстный селектор </TITLE>
    <STYLE type="text/css">
      <!--
        H3 EM { color: blue }
      -->
    </STYLE>
  </HEAD>
  <BODY bgcolor=white text=black>
    <H3>Контекстный <EM>селектор</EM></H3>
    <P>Контекстный <EM>селектор</EM></P>
  </BODY>
</HTML>

```

Слово «селектор» выводится на экран синим цветом только в первой строке (внутри тега `<H3>`), а во второй строке (внутри тега `<P>`) — черным. Обратите внимание, что в стилевом определении отсутствует запятая. Это и есть признак контекстного определения. Если записать `H3, EM { color: blue }`, то синий цвет приобретут как теги `<H3>`, так и теги `<EM>`, т. е. запятая определяет одинаковые стили для группы тегов.

Контекстные и групповые определения можно комбинировать. Запись

```
H3 EM, H2 I{ color: blue }
```

равнозначна двум таким записям:

```
H3 EM{ color: blue }
```

```
H2 I{ color: blue }
```

Классы

Стилевые определения можно описывать без указания тега. В этом случае каждому определению присваивается имя, которое можно использовать для сопоставления заданного стиля конкретному тегу. Такие стилевые определения называются классами. Класс записывается следующим образом:

```

.имя
{
    характеристика: величина;

```

```
...
характеристика: величина;
}
```

Иными словами, определение записывается как обычно, но вместо указаний на теги размещается конструкция `.имя`. Можно, например, определить стилевой класс с именем `def`:

```
.def
{
    color:red;
    font-size: 16pt;
    font-family:Geneva, Helvetica, sans-serif;
    border:solid 0.2cm blue
}
```

Замечание

Строка `border:solid 0.2cm blue` демонстрирует запись в одном стилевом указании нескольких параметров разного назначения (здесь: стиль, толщина рамки и ее цвет). Такую интеграцию допускают специальные обобщающие стилевые свойства (см. *Приложение 1*), к которым относится `border`.

Сопоставляют стилевой класс с тегом при помощи атрибута `class`:

```
<P class=def>текст</P>
```

Посмотрите, как работает этот код (рис. 3.3, листинг 3.4).

Листинг 3.4

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
  <TITLE>Определение класса</TITLE>
  <STYLE type="text/css">
  <!--
    .def
    {
      font-family: Helvetica, sans-serif;
      font-size: 14pt;
      border:solid 4pt red;
```

```
padding: 6pt;
margin-left: 5%;
margin-right: 5%;
}
-->
</STYLE>
</HEAD>
<BODY bgcolor=white text=black>
  <H2>Классы</H2>
  <P class=def>
Класс &#151; стиливое определение без привязки к тегу.
  <P>
Вместо указания тега в определении записывается имя класса.
  <P class=def>
Перед именем класса ставится точка.
</BODY>
</HTML>
```

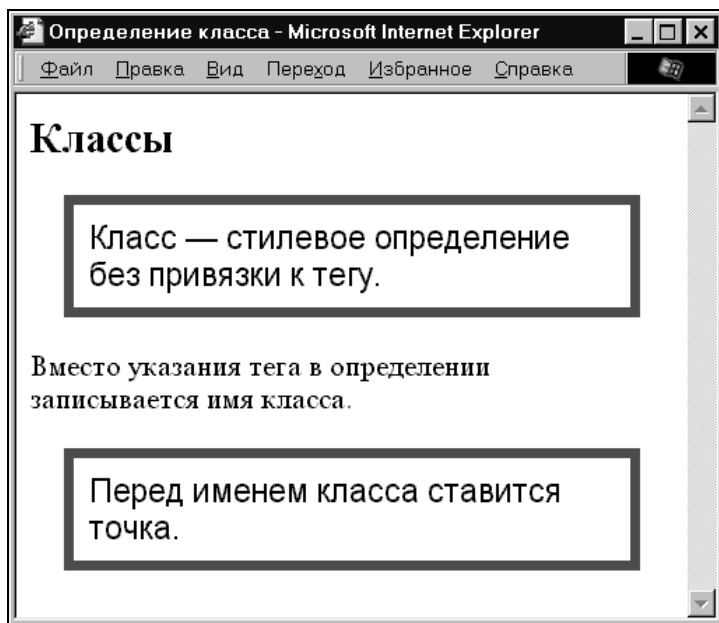


Рис. 3.3. Определение класса

Можно образовывать классы на основе существующих стилевых определений. В следующем примере определен класс `def` — за основу взято опре-

деление стиля для тега `<P>` и добавлены новые свойства (листинг 3.5, рис. 3.4).

Листинг 3.5

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
  <TITLE>Определение класса на базе существующего стиля</TITLE>
  <STYLE type="text/css">
    <!--
      P
      {
        font-family: Helvetica, sans-serif;
      }
      P.def
      {
        text-align: justify;
        background:#CFB597;
        font-size: 14pt;
        border:solid 4pt red;
        padding: 6pt;
        margin-left: 5%;
        margin-right: 5%;
      }
    -->
  </STYLE>
</HEAD>
<BODY bgcolor=white text=black>
  <H2>Классы (обычный заголовок)</H2>
  <P>
    Этот абзац использует стилевое определение для тега P (рубленый шрифт).
  <P class=def>
    Этот абзац использует стиль производного класса &#147;def&#148; (в опре-
    деление стиля для тега P добавлено: красная рамка, поля, цвет фона, вы-
    равнивание слева и справа, повышенный размер шрифта).
  </BODY>
</HTML>
```

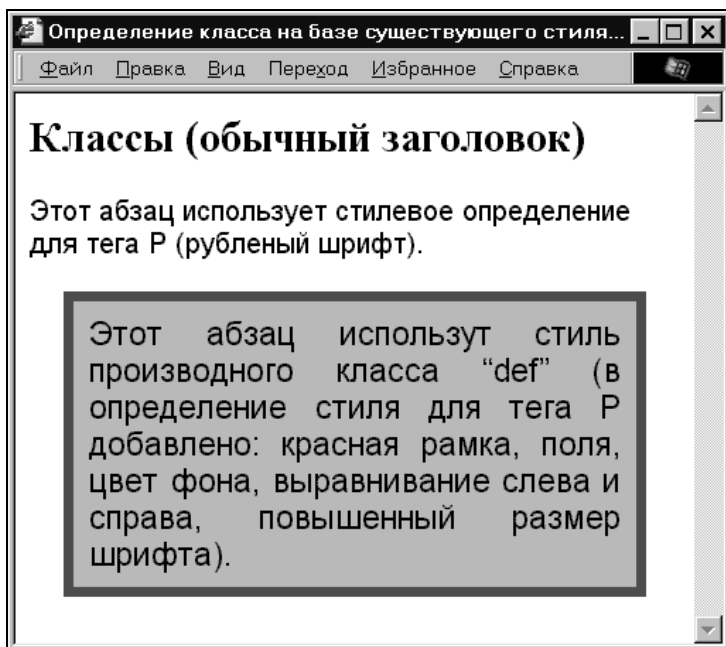


Рис. 3.4. Определение класса на базе существующего стиля

Теги `<DIV>` и `<SPAN>`

Эти теги играют особую роль для CSS. Они позволяют выделять в документе отдельные области, задавая для них специфические свойства. Все, что нужно сделать — это поместить теги, принадлежащие конструируемой области внутри `<DIV>...</DIV>` или `<SPAN>...</SPAN>`. Разница между `<DIV>` и `<SPAN>` только в одном: после блока `<DIV>` браузер переходит на новую строку, а после блока `<SPAN>` — нет. Использование тега `<SPAN>` позволяет тем самым задавать стилевые свойства даже отдельным словам и буквам.

Посмотрите примеры использования этих тегов (рис. 3.5, 3.6 и листинги 3.6, 3.7).

Пример 1. `<DIV>`

Листинг 3.6

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
```

```
<TITLE>Использование тега DIV</TITLE>
<STYLE type="text/css">
<!--
    .areal
    {
        color:red; font-weight:bolder;
        font-size:40pt; background:aqua;
    }
    .area2
    {
        color:maroon;background:#CFB597;
    }
    .area3
    {
        color:blue;background:#C0C0C0;
    }
-->
</STYLE>
</HEAD>
<BODY bgcolor=white text=black>
    <DIV class=areal><IMG src=./pic/ball.gif>Где</DIV>
    <DIV class=area2>начало того конца,</DIV>
    <DIV class=area3>которым оканчивается начало?</DIV>
</BODY>
</HTML>
```

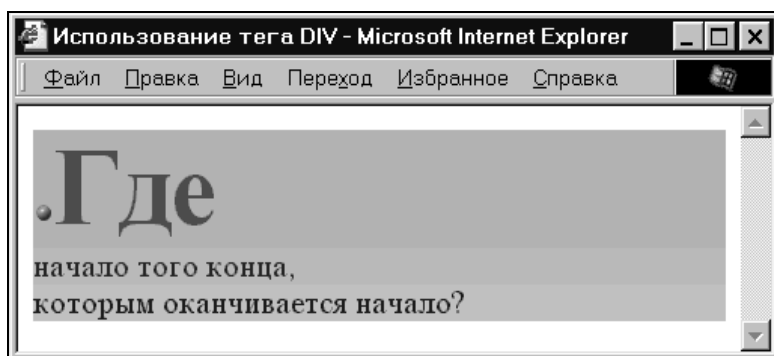


Рис. 3.5. Использование тега <DIV>

Пример 2.

Листинг 3.7

```
<HTML>

<HEAD>

  <META http-equiv="Content-Type"
        content="text/html; charset=windows-1251">

  <TITLE>Использование тега SPAN</TITLE>

  <STYLE type="text/css">

    <!--
      .area1
      {
        color:red; font-weight:bolder;
        font-size:40pt; background:aqua;
      }
      .area2
      {
        color:maroon;background:#CFB597;padding: 6pt;
      }
      .area3
      {
        color:blue;background:#C0C0C0;padding: 6pt;
      }
    -->

  </STYLE>

</HEAD>

<BODY bgcolor=white text=black>

  <SPAN class=area1><IMG src=./pic/ball.gif>Где </SPAN>

  <SPAN class=area2>начало того конца, </SPAN>

  <SPAN class=area3>которым оканчивается начало?</SPAN>

</BODY>

</HTML>
```

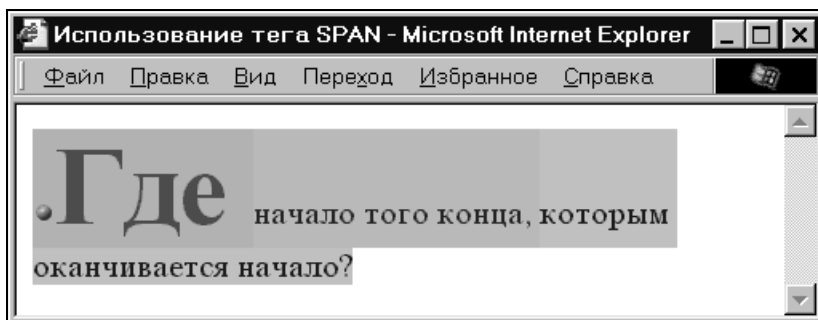


Рис. 3.6. Использование тега

Абсолютное позиционирование

При помощи CSS можно отображать элементы на экране, используя реальные координаты, отсчитываемые от левого верхнего угла окна браузера. Такую возможность предоставляет стилевое свойство `position` со значением `absolute`. Сами координаты задаются при помощи свойств `left` (горизонтальная координата) и `top` (вертикальная координата).

Применение этих свойств иллюстрирует следующий пример (листинг 3.8, рис. 3.7).

Листинг 3.8

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
  <TITLE>Абсолютное позиционирование</TITLE>
  <STYLE type="text/css">
    <!--
    .area1
    {
      position: absolute; top:10; left:10;
      color:red;font-weight:bolder;
      font-size:40pt; background:aqua
    }
  </-->
</HEAD>
<BODY>
```

```

.area2
{
    position: absolute; top:20; left:150;
    color:maroon;background:#CFB597;
    padding: 12pt;
}
.area3
{
    position: absolute; top:70; left:130;
    color:blue;background:#C0C0C0;
    padding: 12pt;
}
-->
</STYLE>
</HEAD>
<BODY bgcolor=white text=black>
    <DIV class=area1><IMG src=./pic/ball.gif>Где</DIV>
    <DIV class=area2>начало того конца,</DIV>
    <DIV class=area3>которым оканчивается начало?</DIV>
</BODY>
</HTML>

```

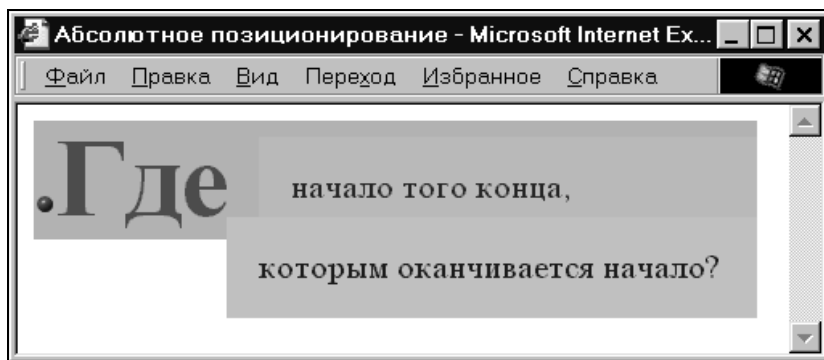


Рис. 3.7. Абсолютное позиционирование

Как видите, браузер разместил три заданные тегами `<DIV>`, области в указанные координаты. При этом области перекрывают друг друга.

Ближе к пользователю получается та область, которая следует в HTML-коде последней. Если переставить порядок следования тегов `<div>` в программе, то и порядок наложения областей друг на друга изменится. Однако CSS предоставляет программисту и другой, более гибкий инструмент для управления порядком наложения элементов. Это — `z-index`.

Z-index

Этот стилевое свойство позволяет указывать, в каком слое (на каком уровне) находится элемент на экране. Номер основного уровня (на который выводятся обычные элементы без стиливых указаний) равен нулю. Следовательно, элементы с отрицательным `z-index` расположены ниже (дальше), с положительным — выше (ближе) основного экранного слоя. Если элементы имеют одинаковый `z-index`, то они расположены в одном слое. В противном случае ближе к нам расположен слой, имеющий больший `z-index`.

Приведенный ниже пример демонстрирует свойство `z-index` (листинг 3.9, рис. 3.8).

Листинг 3.9

```
<HTML>
  <HEAD>
    <META http-equiv="Content-Type"
      content="text/html; charset=windows-1251">
    <TITLE>Z-index</TITLE>
  </HEAD>
  <BODY bgcolor=white text=black>
    <H2>Z-index</H2>
    <HR>
    <DIV style="position:absolute; top:50; left:140;
      height:130; width:100; background:red;
      font-size:60; color:white; z-index:3">
      Это ближе</DIV>
    <DIV style="position:absolute; top:50; left:360;
      height:30; width:100; background:blue;
      font-size:30; color:white; z-index:1">
      Это дальше</DIV>
```

```
<DIV style="position:absolute; top:80; left:270;  
        width:125; z-index:2">  
    <IMG src=pic/explorer.gif width=125 height=82 alt =""></DIV>  
</BODY>  
</HTML>
```



Рис. 3.8. Z-index

Каскадирование

Web-программист благодаря CSS получает гибкий инструмент для представления информации на экране компьютера. Важно не запутаться в том, как созданные стили будут взаимодействовать друг с другом, с атрибутами тегов (они тоже определяют стиль) и со стилями самого браузера (теми стилями, которыми браузер показывает документ по умолчанию).

Как было рассказано в *Уроке 1*, программист может использовать в HTML-документе три типа стилей:

- ☐ Встроенный (inline). Стил, описанный внутри тега при помощи атрибута `style`. Контролирует представление отдельного тега.
- ☐ Внедренный (embedded). Стил, описанный в заголовке HTML-файла при помощи тегов `<STYLE>...</STYLE>`. Контролирует представление отдельного HTML-документа.
- ☐ Связанный (linked). Стил, описанный в отдельном CSS-файле. Контролирует представление многих HTML-документов. Для ссылки на стилевой файл в головной части HTML-документа записывают тег `<LINK>`.

В одном документе могут применяться все описанные выше три стилевых механизма. Добавим к этому стили, задаваемые обычными атрибутами тегов, и стиль «по умолчанию» самого браузера. Как это все взаимодействует, какому стилю отдает предпочтение браузер при построении документа на экране компьютера?

Правила, которым должен следовать «послушный» браузер, называются *каскадными*. Они означают, что для браузера самыми главными являются встроенные стили, затем, по убыванию старшинства, внедренный и связанный. Как следует из опытов предыдущих уроков, внедренный и связанные стили совершенно равнозначны для браузера. Он следует последнему по порядку указанию.

Самым младшим в стилевой иерархии оказывается стиль «по умолчанию». Его браузер использует тогда, когда нет никаких стилевых указаний. В понятие «каскадирование» входит и механизм наследования, по которому к потомку без собственных стилей применяется стиль родителя. Стили, заданные обычными атрибутами тегов, работают по общему каскадному правилу (рис. 3.9).

<pre> <P style="color:blue"> Это будет показано синим. </P> </pre>	Это будет показано синим.
<pre><P style="color:blue"> Это будет показано синим, а это будет показано красным. </P></pre>	Это будет показано синим, а это будет показано красным.

Рис. 3.9. Каскадирование стилей

Зачетный класс

1. Что такое «иерархия»?
2. Для каждой из приведенных ниже схем (рис. 3.10, 3.11) запишите HTML-коды, которые им соответствуют.

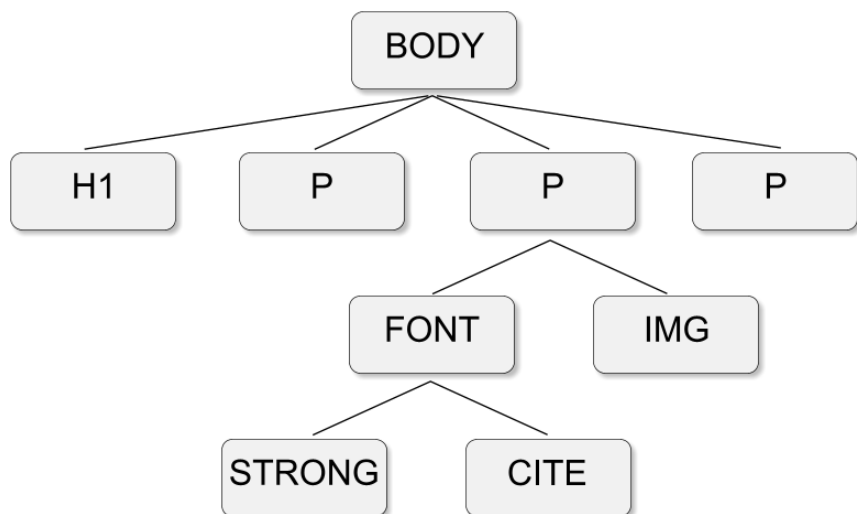


Рис. 3.10. Схема 1

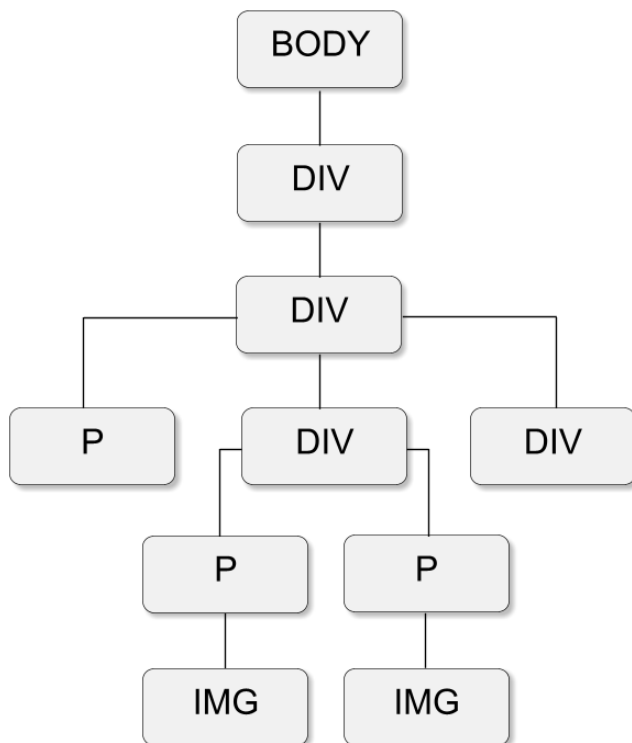


Рис. 3.11. Схема 2

3. Постройте иерархическое дерево для каждой приведенной ниже программы.

Программа 1	Программа 2
<pre> <BODY> <DIV> ... </DIV> <DIV> <DIV> <DIV> <DIV> ... </DIV> </DIV> </DIV> </DIV> <DIV> ... </DIV> <DIV> ... </DIV> </BODY> </pre>	<pre> <BODY> <DIV> <DIV> ... </DIV> <DIV> <DIV> <DIV> ... </DIV> <DIV> ... </DIV> </DIV> <DIV> <DIV> <DIV> ... </DIV> </DIV> </DIV> <DIV> ... </DIV> <DIV> ... </DIV> </BODY> </pre>

4. Опишите механизм наследования стилей.
5. Можно ли переопределить наследуемый стиль? Если да, то как?
6. Для тега `<P>` задан стиль `P {color:red}`. Каким цветом будет выведено слово «сложный» в следующих кодах:
- `<H1>Это сложный пример</H1>;`
 - `<H1>Это <SPAN style="color:blue">сложный</SPAN> пример.</H1>;`
 - `<P>Это сложный пример.</P>;`

г) `<P>Это <SPAN style="color:blue">сложный</SPAN> пример.</P>;`

д) `<P>Это <FONT color=blue>сложный</FONT> пример.</P>;`

е) `<P>Это <SPAN style="font-size:smaller">сложный</SPAN> пример.</P>.`

7. Что такое «контекстное определение»?

8. Задан стиль `P {color:red}`. Каким цветом будет выведено слово «сложный» в следующих кодах:

а) `<H1>Это сложный пример</H1>;`

б) `<H1>Это <EM>сложный</EM> пример</H1>;`

в) `<P>Это сложный пример.</P>;`

г) `<P>Это <EM>сложный</EM> пример</P>?`

9. Задан стиль `P, EM {color:red}`. Каким цветом будет выведено слово «сложный» в следующих кодах:

а) `<H1>Это сложный пример</H1>;`

б) `<H1>Это <EM>сложный</EM> пример</H1>;`

в) `<P>Это сложный пример.</P>;`

г) `<P>Это <EM>сложный</EM> пример</P>.`

10. Опишите назначение тегов `<DIV>` и `<SPAN>`. В чем сходство работы этих тегов, а в чем различие?

11. Что такое «абсолютное позиционирование», каким стилевым указанием оно задается?

12. Что принимается за начало координат при абсолютном позиционировании?

13. По какому правилу абсолютно позиционированные области накладываются друг на друга?

14. Какие значения может принимать стилевое свойство `z-index` и как оно влияет на экранный образ объекта?

15. Что понимается под каскадированием стилевых указаний?

Задания

При подготовке заданий, представленных ниже, использовались задачи к лекциям Е. П. Лилитко «Программирование для Internet». Для построения решений рекомендуется использовать справочник свойств CSS, *Приложение 1*.

1. Сделайте страницу, в которой все абзацы выровнены по левому и правому краям, и имеют красную строку в один сантиметр.

2. Создайте новый стиль `P.def`. Абзацы этого стиля должны выравниваться по ширине, иметь отступ слева 2 см и справа 1 см. Расположите на странице обычные абзацы вместе с абзацами `P.def`.
3. Определите стили для написания старой и новой цены товара. Старая цена — серого цвета, перечеркнутая. Новая цена — красного цвета, на 50% более крупного кегля, чем остальной текст. Напишите список товаров со старыми и новыми ценами.
4. Определите два стиля. В первом стиле:

- буквы должны печататься коричневым по светло-серому фону;
- расстояние между содержимым и рамкой элемента должно составлять 0,5 см;
- текст выравнивается по левому и правому краям.

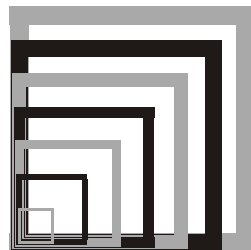
Во втором стиле:

- фон бирюзовый;
- расстояние между содержимым и рамкой элемента должно составлять 0,5 см;
- поля слева и справа от элемента по 1 см;
- рубленый шрифт.

Подготовьте документ с двумя разделами. Первый раздел определяется первым стилем, второй — вторым. Второй раздел должен быть вложен в первый, чтобы было видно наследование. Какие стилевые указания наследуются во втором разделе, а какие нет?

5. Сделайте стиль `.nb` такой, чтобы элемент заключался в рамку (бордюр), занимал (по ширине) половину окна браузера (независимо от его размера), был расположен у левого края, а остальные элементы страницы «обтекали» бы этот элемент справа.
6. Подготовьте стиль для абзаца, у которого слева и справа проводится вертикальная черта (на всю высоту абзаца).
7. Подготовьте стиль абзаца «подложка» (`.ground`) и стиль «надпись» (`.poster`). Используя созданные стили, создайте страницу с надписью поверх подложки. Подложка пишется очень крупными буквами мягкого светло-серого цвета. Надпись пишется коричневыми буквами обычного размера.
8. Используя стили, сделайте страницу, в которой текст выводится в две колонки (как в газете). Таблицы при этом не используются.
9. Постройте на экране две области с линейками прокрутки и поместите в них информационные элементы.
10. Используя свойство `z-index`, постройте на экране несколько перекрывающих друг друга областей.

Урок 4



Позиционирование, *z-index*

Позиционирование

Позиционирование — это управление координатами размещения элемента в окне браузера. CSS предлагает для позиционирования свойство `position`.

У этого свойства могут быть три значения: `absolute` (абсолютное позиционирование), `relative` (относительное позиционирование) и `static` (статическое позиционирование). Значение `static` размещает элемент на странице так, как он располагался бы без всякого позиционирования, поэтому использование этого значения не дает ничего нового. Алгоритмы работы значений `absolute` и `relative` будут рассмотрены ниже, но сначала одно важное напоминание об иерархической структуре HTML-кода.

Иерархия кода страницы

Иерархия — это структура, в которой содержимое «упаковано» по пакетикам, вложенным друг в друга наподобие матрешек.

Все элементы страницы расположены внутри блока `<BODY>`. Таким образом, элемент `<BODY>` является родителем всех других элементов страницы. Пусть, например, страница имеет следующий код (рис. 4.1).

В показанном примере элемент `<BODY>` имеет два прямых потомка `e01` и `e02`. В свою очередь эти элементы тоже имеют по два потомка: `e011`, `e012` и `e021`, `e022` соответственно.

Позиционирование элементов выполняется с учетом иерархии кода. Вот почему так важно отчетливо представлять структуру документа. Опытные программисты всегда записывают код лесенкой (как в приведенном примере), и у них возникает гораздо меньше проблем при программировании, тестировании и отладке своих страниц.

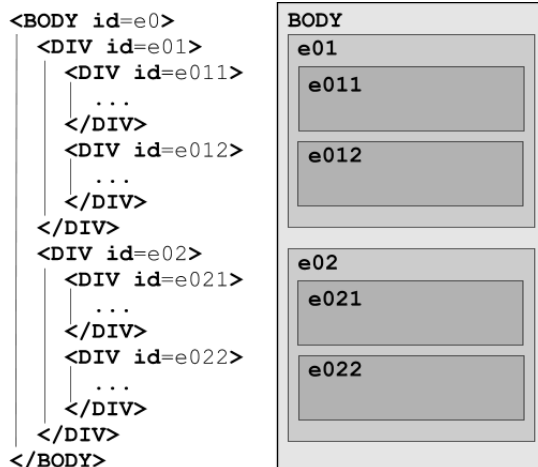


Рис. 4.1. Иерархия страничного кода

Абсолютное позиционирование

Абсолютное позиционирование задается стилевым указанием `position:absolute`. При этом начало координат элемента находится в верхнем левом углу прямого предка, при условии, что он также позиционирован (абсолютно или относительно). Если родитель не позиционирован, то берется его родитель. Если все предки не имеют указаний `position`, то в качестве точки отсчета принимается левый верхний угол экранного образа тега `<BODY>`, т. е. левый верхний угол документа.

Горизонтальная и вертикальная координаты задаются свойствами `left` и `top` соответственно.

Ниже показан пример (листинг 4.1, рис. 4.2), в котором картинка позиционируется в точку (100, 50). Началом координат браузер считает начало документа.

Листинг 4.1

```

<HTML>
  <HEAD>
    <META http-equiv="Content-Type"
      content="text/html; charset=windows-1251">
    <TITLE>Абсолютное позиционирование</TITLE>
  </HEAD>
  <BODY bgcolor=white text=black>
    <H1>Абсолютное позиционирование</H1>
  <P>

```

В этом примере картинка абсолютно позиционирована. Она располагается в 100 пикселях по горизонтали и в 50 пикселях по вертикали от начала документа.

<P>

Измените размеры окна и убедитесь, что картинка всегда остается на прежнем месте.

```
<IMG src=./pic/let.gif width=126 height=60 border=0
alt="Роботландский университет"
style="position:absolute; left:100px; top:50px;">
```

</BODY>

</HTML>

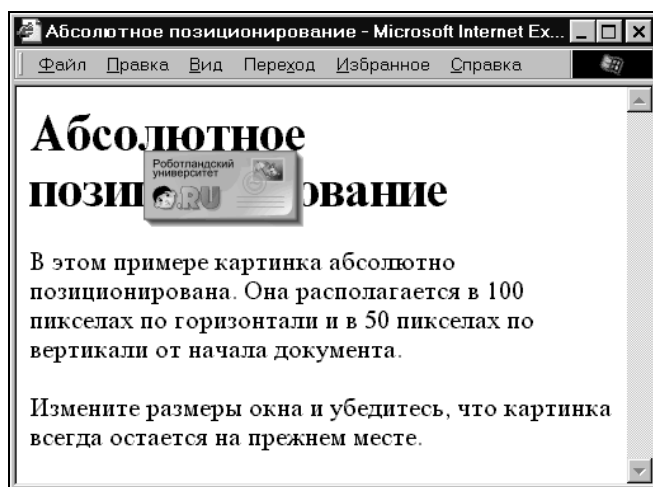


Рис. 4.2. Картинка абсолютно позиционирована в (100, 50)

В следующем примере (листинг 4.2, рис. 4.3) абсолютно позиционированы две картинки. В коде для каждой из них указаны координаты (100, 50), но для одной началом координат является начало документа, а для другой — начало таблицы.

Листинг 4.2

```
<HTML>
```

```
<HEAD>
```

```
<META http-equiv="Content-Type"
      content="text/html; charset=windows-1251">
```

```
<TITLE>Абсолютное позиционирование</TITLE>
```

```
</HEAD>
```

```
<BODY bgcolor=white text=black>
```

```
<H1>Абсолютное позиционирование</H1>
```

```
<P>
```

Обе картинки абсолютно позиционированы в точку (100, 50). Начало координат для одной картинки совпадает с началом документа, для второй — с началом таблицы.

```
<TABLE border=1 cellspacing=0 cellpadding=10
```

```
    style="position:absolute; left:50px; top:300px;">
```

```
<TR><TD>
```

Таблица абсолютно позиционирована.

```
<IMG src=./pic/let.gif width=126 height=60 border=0
```

```
    alt="Роботландский университет"
```

```
    style="position:absolute; left:100px; top:50px;">
```

```
</TD></TR>
```

```
</TABLE>
```

```
<IMG src=./pic/let.gif width=126 height=60 border=0
```

```
    alt="Роботландский университет"
```

```
    style="position:absolute; left:100px; top:50px;">
```

```
</BODY>
```

```
</HTML>
```

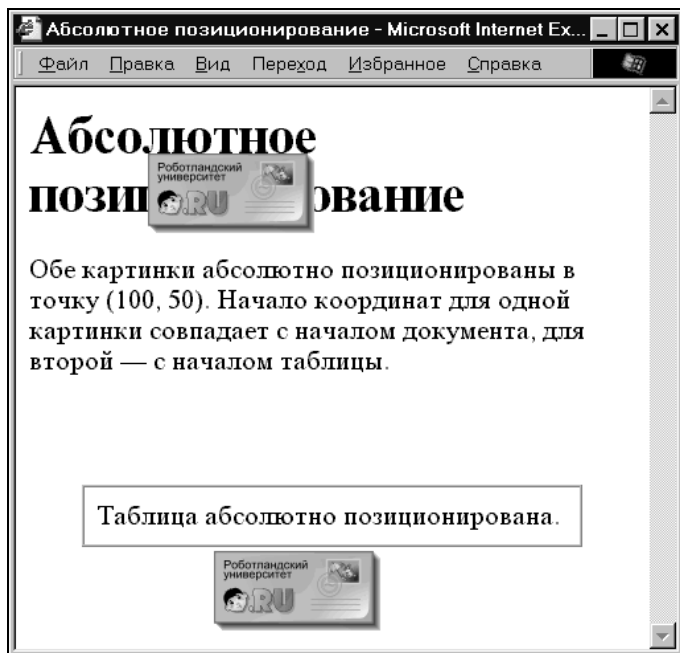


Рис. 4.3. Две абсолютно позиционированные картинки

Приведенные выше примеры наглядно показывают, что абсолютно позиционированные элементы выпадают из процесса обычного последовательного форматирования. Браузер не берет в расчет порядок следования кодов, а учитывает только вложенность для определения начала координат. Элементы выводятся в указанном месте, перекрывая при этом другие элементы страницы.

Порядок следования кодов абсолютно позиционируемых элементов становится важным, если они начинают перекрывать друг друга: «выше» оказывается тот элемент, код которого идет позже (листинг 4.3, рис. 4.4).



Рис. 4.4. Картинка, код которой расположен позже, перекрывает картинку, код которой идет раньше

Листинг 4.3

```
<HTML>

<HEAD>

<META http-equiv="Content-Type"
```

```
content="text/html; charset=windows-1251">
<TITLE>Абсолютное позиционирование</TITLE>
</HEAD>
<BODY bgcolor=white text=black>
  <H1>Абсолютное позиционирование</H1>
  <P>
```

В этом примере обе картинки абсолютно позиционированы. Так как код `“Незнайки”` идет раньше кода `“Профессора”`, то вторая картинка перекрывает первую.

```
  <IMG src=./pic/prav01.gif width=140 height=150 border=0
    alt=Незнайка
    style="position:absolute; left:100px; top:150px;">
  <IMG src=./pic/prav31.gif width=140 height=150 border=0
    alt=Профессор
    style="position:absolute; left:200px; top:250px;">
</BODY>
</HTML>
```

Относительное позиционирование

Относительное позиционирование задается стилевым указанием `position:relative`. Такой элемент участвует в обычном последовательном форматировании документа. За начало координат принимается та точка, в которой элемент был бы размещен без позиционирования.

Алгоритм относительного позиционирования можно представить следующим образом. Сначала браузер размещает элемент на странице, выполняя обычное форматирование, а затем по указаниям `left` и `top` смещает его на заданные значения.

В приведенном ниже примере (листинг 4.4, рис. 4.5) документ форматируется обычным образом.

Листинг 4.4

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
```

```

<TITLE>Обычное построение документа</TITLE>
</HEAD>
<BODY bgcolor=white text=black>
  <H1>Обычное построение документа</H1>
  <P>
    <IMG src=./pic/let.gif width=126 height=60 border=0
      alt=Университет align=left hspace=10>
    Это пример обычного форматирования документа без всякого
    позиционирования.<BR clear=left>
  </BODY>
</HTML>

```

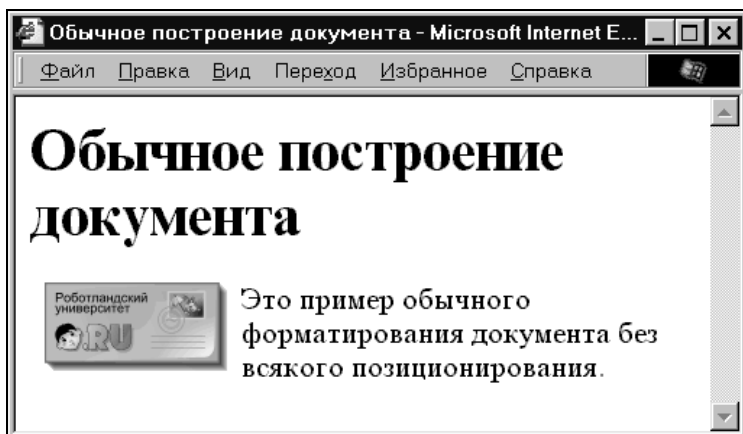


Рис. 4.5. Обычное форматирование

В следующем примере (листинг 4.5, рис. 4.6) на картинку наложено относительное позиционирование.

Листинг 4.5

```

<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
  <TITLE>Относительное позиционирование</TITLE>
</HEAD>
<BODY bgcolor=white text=black>
  <H1> Относительное позиционирование </H1>

```

<P>

```
<IMG src=./pic/let.gif width=126 height=60 border=0  
alt=Университет align=left hspace=10  
style="position:relative; left:0px; top:150px;">
```

В этом примере картинка смещена вниз на 150 пикселей относительно того места, которое она занимала бы на экране при отсутствии позиционирования. <BR clear=left>

</BODY>

</HTML>

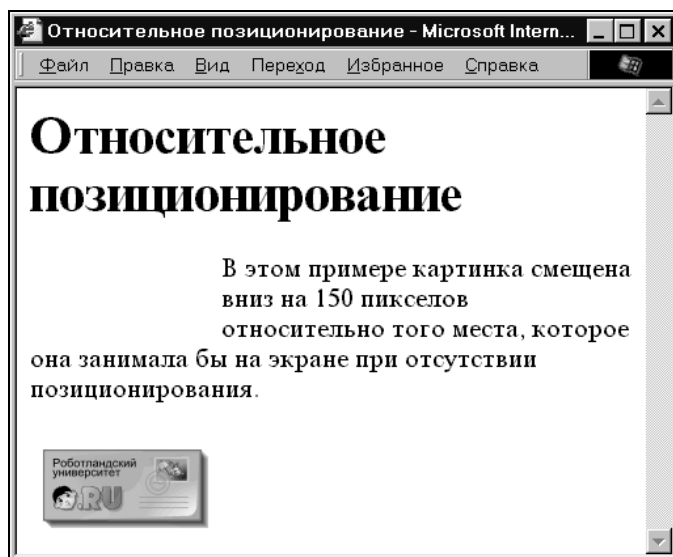


Рис. 4.6. Относительное позиционирование

Если изменить размер окна, то можно увидеть, что картинка перемещается по экрану вслед за точкой отсчета — началом абзаца, которому принадлежит тег `<img>` в коде документа. Начало абзаца меняет свое положение тогда, когда предшествующий заголовок записывается в одну или две экранные строки.

Смешанное позиционирование

Рассмотрим четыре варианта смешанного позиционирования, условно обозначенных ниже как:

- `absolute [relative]` — относительно позиционированный блок внутри абсолютно позиционированного;

- `relative [absolute]` — абсолютно позиционированный блок внутри относительно позиционированного;
- `absolute [absolute]` — абсолютно позиционированный блок внутри абсолютно позиционированного;
- `relative [relative]` — относительно позиционированный блок внутри относительно позиционированного.

Absolute [relative]

Код относительно позиционированной картинки расположен внутри кода абсолютно позиционированной таблицы. Таблица смещена от начала документа вправо на 50 пикселей, и это положение остается фиксированным при изменении размеров окна. Картинка смещена вниз на 100 пикселей относительно ее «нормального» положения внутри таблицы. При этом в таблице текст расположен так, как будто картинка находится на своем обычном месте (листинг 4.6, рис. 4.7, 4.8).

Листинг 4.6

```
<HTML>
  <HEAD>
    <META http-equiv="Content-Type"
      content="text/html; charset=windows-1251">
    <TITLE>Смешанное позиционирование</TITLE>
  </HEAD>
  <BODY bgcolor=white text=black>
    <H1>Смешанное позиционирование</H1>
    <P>
      Таблица абсолютно позиционирована в точку (50,0). Внутри кода
      таблицы расположена картинка с относительным позиционированием вниз
      на 100 пикселей.
    <P>
      <TABLE border=1 cellspacing=0 cellpadding=10
        width=80% bgcolor=#EEE5DB
        style="position:absolute;left:50px;top:0px">
        <TR><TD>
          <P>
            <IMG src=./pic/let.gif width=126 height=60 border=0
              alt=Университет align=left hspace=10
              style="position:relative; left:0px; top:100px;">
```

```

Тег <TT><B>IMG</B></TT> размещается внутри таблицы.
Для картинки записано относительное позиционирование
вниз на 100 пикселей.<BR clear=left>
</TD></TR>
</TABLE>
</BODY>
</HTML>

```

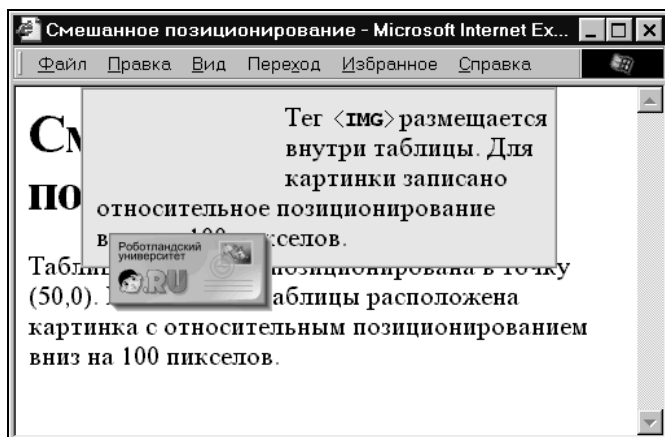


Рис. 4.7. Absolute (relative)

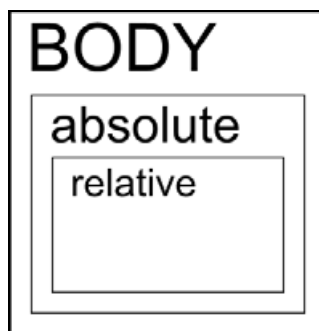


Рис. 4.8. Схема позиционирования

Relative [absolute]

Во втором примере код абсолютно позиционированной картинки расположен внутри кода относительно позиционированной таблицы. Таблица смещена относительно своего «нормального» положения вправо на 50 пиксе-

лов. Начало координат для картинки связано с левым верхним углом таблицы. Картинка смещается вниз относительно этого положения на 100 пикселей. При этом текст в таблице форматируется без учета тега `<IMG>` (листинг 4.7, рис. 4.9, 4.10).

Листинг 4.7

```
<HTML>
```

```
<HEAD>
```

```
<META http-equiv="Content-Type"
      content="text/html; charset=windows-1251">
```

```
<TITLE>Смешанное позиционирование</TITLE>
```

```
</HEAD>
```

```
<BODY bgcolor=white text=black>
```

```
<H1>Смешанное позиционирование</H1>
```

```
<P>
```

Таблица относительно позиционирована вправо на 50 пикселей. Внутри кода таблицы расположена картинка с абсолютным позиционированием (0,100).

```
<P>
```

```
<TABLE border=1 cellspacing=0 cellpadding=10
      width=80% bgcolor=#EEE5DB
      style="position:relative;left:50px;top:0px">
```

```
<TR><TD>
```

```
<P>
```

```
<IMG src=./pic/let.gif width=126 height=60 border=0
      alt=Университет align=left hspace=10
      style="position:absolute; left:0px; top:100px;">
```

Тег `<TT><B>IMG</B></TT>` размещается внутри таблицы.

Для картинки записано абсолютное позиционирование вниз на 100 пикселей. `<BR clear=left>`

```
</TD></TR>
```

```
</TABLE>
```

```
</BODY>
```

```
</HTML>
```

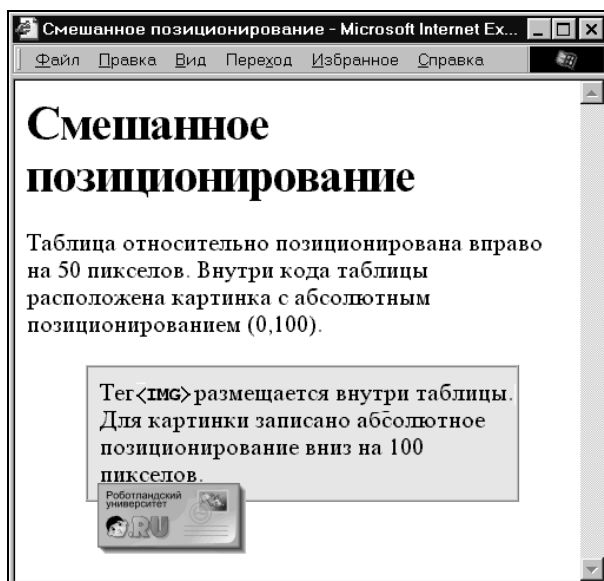


Рис. 4.9. Relative (absolute)

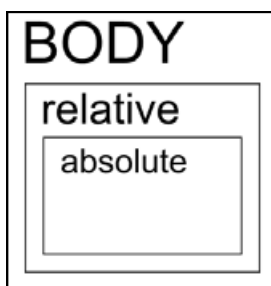


Рис. 4.10. Схема позиционирования

Absolute [absolute]

Код абсолютно позиционированной картинки расположен внутри кода абсолютно позиционированной таблицы. Картинка не меняет своего положения на экране при изменении размера окна, потому что координаты таблицы остаются неизменными (листинг 4.8, рис. 4.11, 4.12).

Листинг 4.8

```
<HTML>
  <HEAD>
```

```

<META http-equiv="Content-Type"
      content="text/html; charset=windows-1251">
<TITLE>Смешанное позиционирование</TITLE>
</HEAD>
<BODY bgcolor=white text=black>
  <H1>Смешанное позиционирование</H1>
  <P>
    Таблица абсолютно позиционирована в точку (50,0). Внутри кода
    таблицы расположена картинка с абсолютным позиционированием
    в точку (0,100).
  <P>
    <TABLE border=1 cellspacing=0 cellpadding=10
      width=80% bgcolor=#EEE5DB
      style="position:absolute;left:50px;top:0px">
      <TR><TD>
        <P>
          <IMG src=./pic/let.gif width=126 height=60 border=0
            alt=Университет align=left hspace=10
            style="position:absolute; left:0px; top:100px;">
          Тег <TT><B>IMG</B></TT> размещается внутри таблицы.
          Для картинки записано абсолютное позиционирование
          в точку (0,100).<BR clear=left>
        </TD></TR>
      </TABLE>
    </BODY>
</HTML>

```

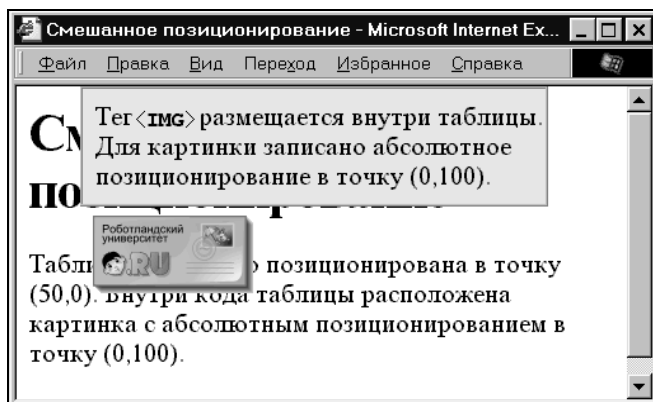


Рис. 4.11. Absolute (absolute)

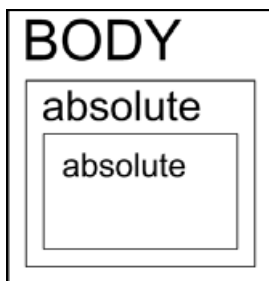


Рис. 4.12. Схема позиционирования

Relative [relative]

Код относительно позиционированной картинке расположен внутри кода относительно позиционированной таблицы. Когда положение таблицы меняется на экране из-за изменения размеров окна, меняется и положение картинке: ведь смещается начало координат, ниже которого на 100 пикселей должна быть расположена картинка (листинг 4.9, рис. 4.13, 4.14).

Листинг 4.9

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
  <TITLE>Смешанное позиционирование</TITLE>
</HEAD>
<BODY bgcolor=white text=black">
  <H1>Смешанное позиционирование</H1>
  <P>
    Таблица относительно позиционирована вправо на 50 пикселей. Внутри
    кода таблицы расположена картинка с относительным позиционированием
    вниз на 100 пикселей.
  <P>
    <TABLE border=1 cellspacing=0 cellpadding=10
      width=80% bgcolor=#EEE5DB
      style="position:relative;left:50px;top:0px">
      <TR><TD>
        <P>
          <IMG src=./pic/let.gif width=126 height=60 border=0
```

```
alt=Университет align=left hspace=10
style="position:relative; left:0px; top:100px;">
```

Тег `<TT><B>IMG</B></TT>` размещается внутри таблицы.

Для картинки записано относительное позиционирование вниз на 100 пикселей. `<BR clear=left>`

```
</TD></TR>
```

```
</TABLE>
```

```
</BODY>
```

```
</HTML>
```

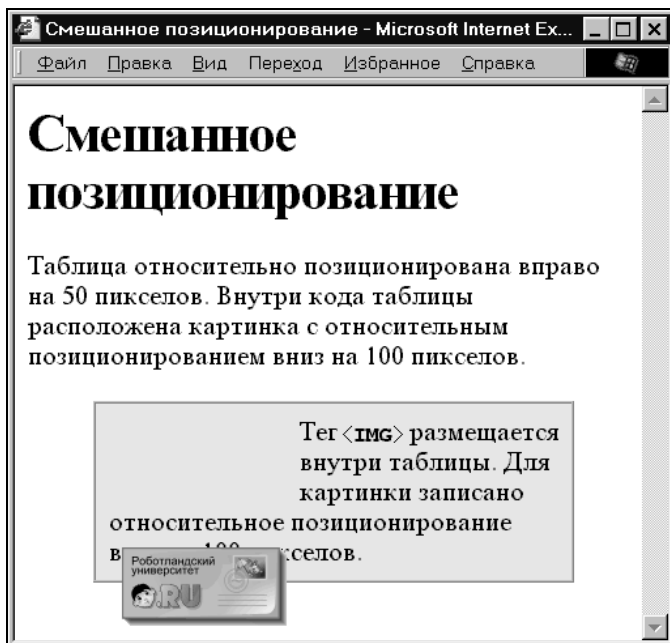


Рис. 4.13. Relative [relative]

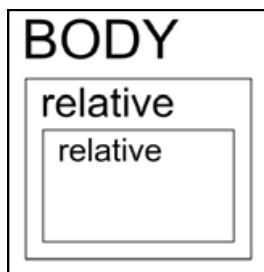


Рис. 4.14. Схема позиционирования

Приведенные выше примеры наглядно показывают, что элемент позиционируется вместе со своими потомками, независимо от того, как позиционируются последние (или не позиционируются вовсе).

Z-index

Стилевое свойство `z-index` позволяет отойти от плоского представления документа на экране. Его значением является целое число, номер плоскости, в которой будет размещаться элемент (вместе со своими потомками). Основной текст имеет нулевой уровень (`z-index:0`). Положительный `z-index` размещает элементы над основным текстом, отрицательный — под ним. Из двух плоскостей размещения та расположена выше, у которой `z-index` больше (листинг 4.10, рис. 4.15).

Листинг 4.10

```
<HTML>
  <HEAD>
    <META http-equiv="Content-Type"
      content="text/html; charset=windows-1251">
    <TITLE>Z-индекс</TITLE>
  </HEAD>
  <BODY bgcolor=white text=black>
    <H1>Z-индекс</H1>
    <P>
      Этот текст&nbsp;&#151; основной. К нему не приписано никаких
      стиливых воздействий. Значит, этот текст лежит в нулевой
      плоскости.
    <IMG src=./pic/let.gif width=126 height=60 border=0
      alt=Университет align=left hspace=10
      style="position:absolute; left:0px; top:20px; z-index=-1;">
    <IMG src=./pic/let.gif width=126 height=60 border=0
      alt=Университет align=left hspace=10
      style="position:absolute; left:50px; top:70px; z-index=-2;">
  </BODY>
</HTML>
```

Если у элементов одинаковый `z-index` (задан явно или не задан вовсе), то тот из них будет располагаться выше, чей HTML-код идет позже.

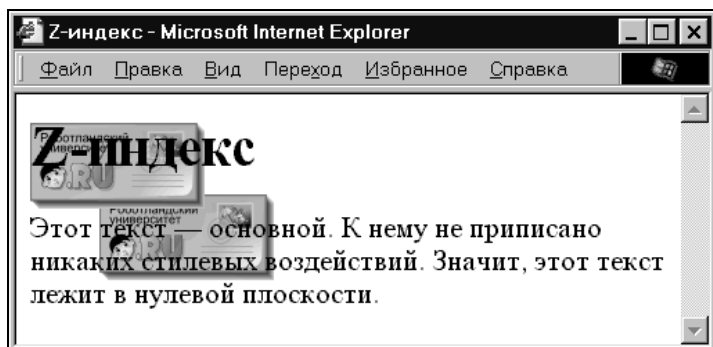


Рис. 4.15. Z-index

Зачетный класс

1. Что такое «позиционирование»?
2. Каким свойством CSS задается позиционирование элемента? Какие значения может принимать это свойство?
3. Какими стилевыми указаниями задаются координаты элемента в окне браузера?
4. Расскажите алгоритм, по которому выполняется абсолютное позиционирование.
5. Расскажите алгоритм, по которому выполняется относительное позиционирование.
6. Расскажите алгоритм выполнения смешанного позиционирования по схеме `absolute [relative]`.
7. Расскажите алгоритм выполнения смешанного позиционирования по схеме `relative [absolute]`.
8. Расскажите алгоритм выполнения смешанного позиционирования по схеме `absolute [absolute]`.
9. Расскажите алгоритм выполнения смешанного позиционирования по схеме `relative [relative]`.
10. Расскажите алгоритм вывода на экран элементов, обладающих стилевым свойством `z-index`.

Задания

1. Создайте страницу, в которой одна и та же картинка выводится несколько раз со смещением вниз и вправо так, чтобы каждая следующая копия была выше предыдущей (рис. 4.16).

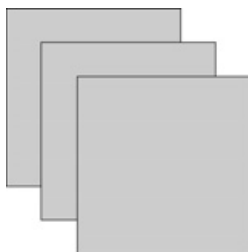


Рис. 4.16. Схема задания 1

2. Создайте страницу, в которой одна и та же картинка выводится несколько раз со смещением вниз и вправо так, чтобы каждая следующая копия была ниже предыдущей (рис. 4.17).

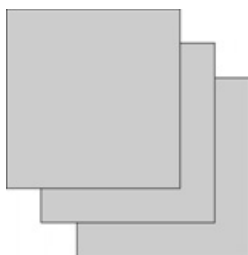


Рис. 4.17. Схема задания 2

3. Стилиевое свойство `overflow` используется для определения того, что случится, если содержимое элемента выйдет за пределы заданной области. Среди предписываемых стандартом значений нормально работает только `scroll` — содержимое прокручивается. Напишите страничку, которая демонстрирует работу свойства `overflow` (рис. 4.18).

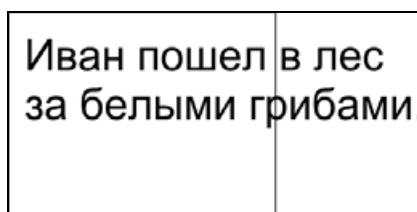


Рис. 4.18. Схема задания 3

4. Иван Мячиков решил задать для текстового блока на своей странице широкое левое поле. Он поместил блок внутрь конструкции

```
<DIV style="position:relative; left:100px;">
```

...

```
</DIV>
```

Поле получилось, но текст перестал форматироваться по правой границе окна. Помогите Мячикову исправить положение (рис. 4.19).

5. Если у абсолютно позиционированного элемента опущена одна или две координаты, то:
- ☐ если не задано свойство `left`, левый край элемента будет располагаться справа за элементом-родителем;
 - ☐ если не задано свойство `top`, верхний край элемента будет располагаться ниже элемента-родителя. При этом если родитель позиционирован относительно, то элемент будет располагаться по вертикали там, где он располагался бы, если вообще не был бы позиционирован.

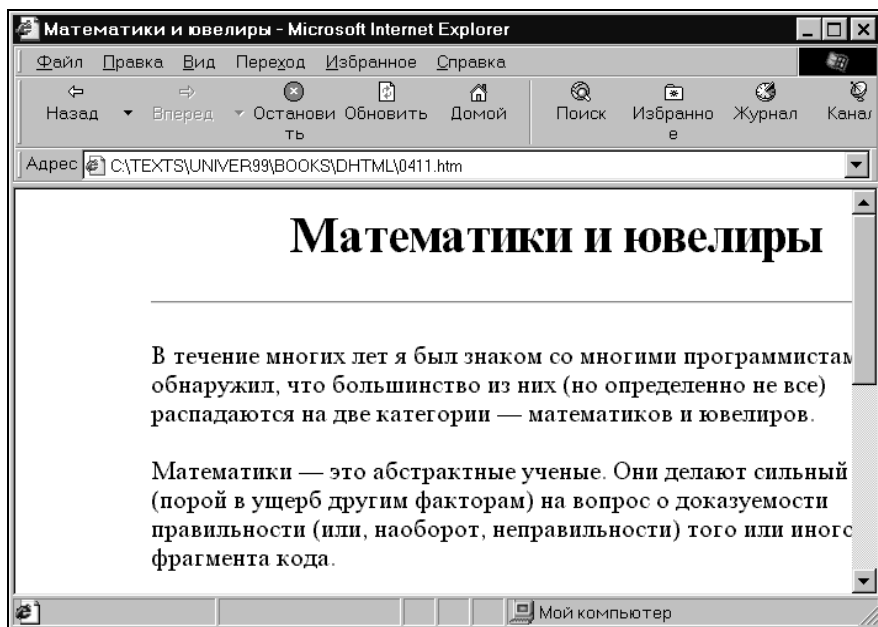


Рис. 4.19. Иллюстрация к заданию 4

Используя описанные особенности, создайте страничку с левым полем, в котором бы располагался маркер, указывающий на ошибочное слово в строке. При этом если из-за изменения размера окна ошибочное слово переместится в другую строку, маркер должен автоматически следовать за ним (рис. 4.20).

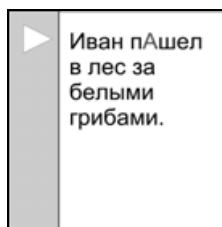


Рис. 4.20. Схема задания 4

5. Если картинку поместить в абзац, то она ведет себя как символ, перемещаясь по экрану за своим местом (при изменении размера окна браузера). Создайте страничку, в которой целый блок (с несколькими картинками, текстом, другими элементами) ведет себя подобным образом (рис. 4.21).

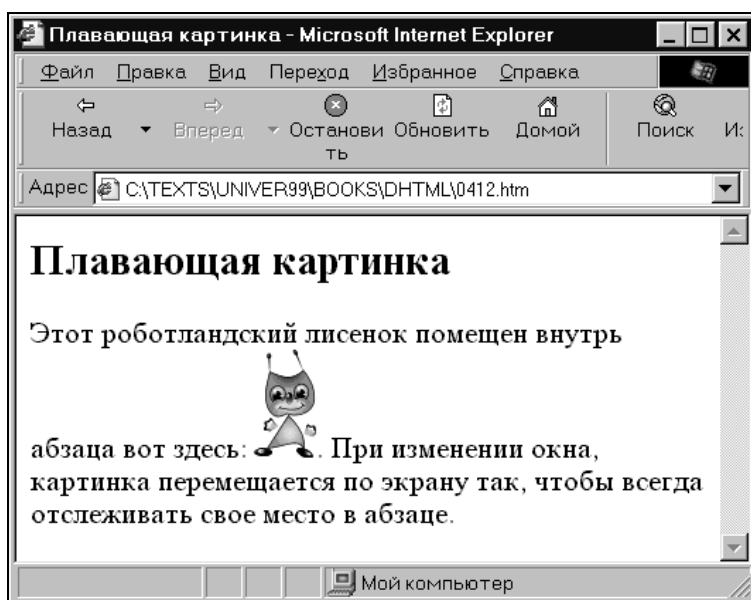
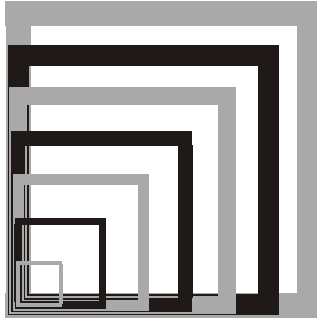


Рис. 4.21. Иллюстрация к заданию 5

6. Используя позиционирование, создайте объемную надпись из обычного текста (без картинок). Эта надпись должна быть привязана к своему месту в абзаце, подобно обычному слову (рис. 4.22).



Рис. 4.22. Иллюстрация к заданию 6



Часть II

Основы JavaScript

Урок 5. Первый скрипт

Урок 6. Переменные, константы,
выражения

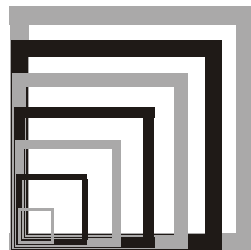
Урок 7. Ветвления

Урок 8. Повторения

Урок 9. Функции

Урок 10. Объект = методы + свойства

Урок 5



Первый скрипт

Классический HTML болен статичностью

«Чистый» HTML имеет весьма ограниченные возможности для программирования интерфейса приложений и построения интерактивных сеансов работы пользователей.

Традиционное средство, которое предлагается HTML для изменения экрана, — щелчок мышью по ссылке. Браузер загружает новый файл, и экран перерисовывается.

Традиционная динамика в документе — анимационный GIF. Но этой анимации далеко до настоящей мультипликации.

А ведь хочется? Правда, хочется! Хочется динамических меню, реакций на перемещения и щелчки мыши. Хочется полноценного диалога на страницах документа, такого, как в настоящих компьютерных программах. Хочется придумать и «выпечь» в гипертексте динамическую игру или обучающую программу с «живыми» демонстрациями, зачетными классами и испытательными стендами. Хочется больше жизни!

JavaScript — одно из тех средств, которые нужны для этой цели. С JavaScript гипертексты получают встроенный программный «моторчик», запускающий HTML на новые обороты.

JavaScript? Что это такое?

Это язык программирования, который понятен браузеру. Это означает, что браузер умеет выполнять (интерпретировать) команды этого языка.

Программу на JavaScript (ее называют *скриптом*, или сценарием) можно помещать внутрь HTML-программы или держать в отдельном файле. Этот файл браузер прочитает (по специальной команде) во время интерпретации HTML-кода.

Коды JavaScript дополняют коды HTML и «живут» вместе с ними. Даже если они расположены в отдельном файле.

Как скрипт «монтируется» в HTML?

Скрипт размещается между двумя парными тегами `<SCRIPT>...</SCRIPT>`. Обычно запись скрипта выглядит так, как показано в листинге 5.1.

Листинг 5.1. Текст программы на языке JavaScript и комментарии

<code><SCRIPT language=JavaScript></code>	Начало скрипта
<code><!--</code>	
<code>...</code>	
<code>Код на JavaScript</code>	Скрипт представлен как HTML-комментарий, чтобы не «смущать» браузеры, которые о скриптах не знают
<code>...</code>	
<code>//--></code>	
<code></SCRIPT></code>	Конец скрипта
<code><NOSCRIPT></code>	
<code>...</code>	
<code>Код для браузеров, которые не поддерживают JavaScript</code>	Блок <code><NOSCRIPT></code> — специально для браузеров, которые «не понимают» скриптов
<code>...</code>	
<code></NOSCRIPT></code>	

Пояснения к приведенному выше листингу 5.1:

- ❑ Код скрипта оформляют как HTML-комментарий. Браузеры, которые умеют работать со скриптами, об этом знают, а другие проигнорируют скрипты (как обычный комментарий).
- ❑ Закрывающий тег HTML-комментария предваряется двумя наклонными чертами `</>` (комментарий языка JavaScript) опять же в целях безопасности. Некоторые браузеры «сходят с ума», если не видят этих знаков. (Комментарием в JavaScript считается все, что располагается от символов `</>` до конца текущей строки.)
- ❑ Открывающий тег `<SCRIPT>` использует атрибут `language` для указания языка, на котором написан скрипт. Значением этого атрибута в нашем случае является ключевое слово `JavaScript`.
- ❑ Блок `<NOSCRIPT>...</NOSCRIPT>` является не обязательным, но желательным. Внутри него могут располагаться теги HTML.

Пример 1

Листинг 5.2

```
<HTML>
  <HEAD>
    <META http-equiv="Content-Type"
      content="text/html; charset=windows-1251">
    <TITLE>Проверка браузера</TITLE>
  </HEAD>
  <BODY bgcolor=white text=black link=blue alink=red vlink=purple>
    <H1>Проверка браузера</H1>
    <HR>
    <SCRIPT language=JavaScript>
      <!--
        alert("Ваш браузер поддерживает язык JavaScript!");
      //-->
    </SCRIPT>
    <NOSCRIPT>
      <H2>Предупреждение</H2>
      <P>
        Ваш браузер не поддерживает JavaScript. Этот документ будет показан не-
        верно, возможны сообщения об ошибках.
      <P>
        Версия документа без скриптов расположена <A href=doc1.htm>здесь</A>.
    </NOSCRIPT>
    <P>
      Для возврата к основному тексту нажмите кнопку <EM>Назад</EM> на инстру-
      ментальной панели браузера.
    </BODY>
</HTML>
```

Скрипт в приведенном выше примере (листинг 5.2), содержит всего одну команду на языке JavaScript — вызов функции `alert`. Функция `alert` создает информационное окно с единственной кнопкой **ОК** внутри (рис. 5.1).

Пользователь читает сообщение, нажимает кнопку и окно убирается с экрана. Теперь браузер приступает к интерпретации команд, расположенных вслед за скриптом в HTML-коде — выводит в окошко текст, показанный на рис. 5.2.

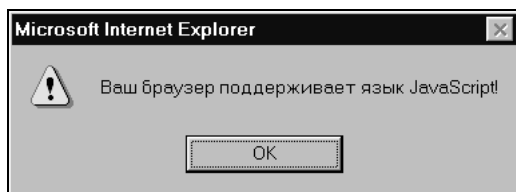


Рис. 5.1. Информационное окно alert

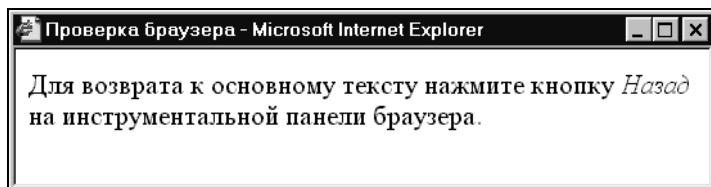


Рис. 5.2. Вид страницы в браузере, который поддерживает скрипты

Если браузер не поддерживает JavaScript или эта поддержка отключена в настройках, в окно документа выводится сообщение, показанное на рис. 5.3.



Рис. 5.3. Вид страницы в браузере, который не поддерживает скрипты

Замечание

Если текст-аргумент функции `alert` слишком длинный, его можно записывать в нескольких строках, используя знак «+» (операция соединения):

```
alert("Язык JavaScript был разработан в 1995 году фирмой"+  
      " Netscape для второй версии своего браузера");
```

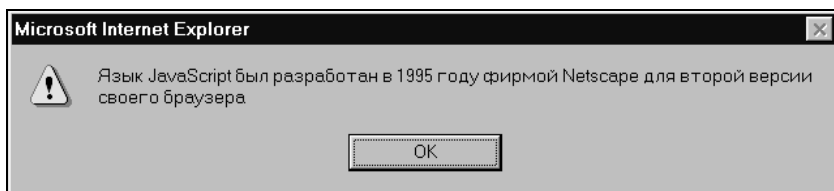


Рис. 5.4. Для записи длинной строки использована операция соединения

Операция соединения не означает разрыв строки на экране (рис. 5.4). Для перевода строк нужно использовать символы «\n»:

```
alert("Первая строка.\nСтрока вторая. ");
```

Браузеры Netscape третьих версий «не любят», когда в текстовых константах JavaScript попадают русские буквы «я» строчная и «С» прописная. «Лечится» это записью перед такими буквами обратной косой черты (рис. 5.5):

```
alert("Перва\я строка.\n\Строка втора\я. ");
```

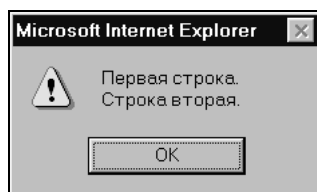


Рис. 5.5. Вывод текста в две строки

Зачетный класс

1. Отметьте верные высказывания:
 - а) JavaScript — это язык программирования;
 - б) программы на JavaScript выполняет Windows;
 - в) программы на JavaScript выполняет браузер;
 - г) JavaScript позволяет создавать динамические гипертексты;
 - д) программа на JavaScript называется скриптом;
 - е) программа на JavaScript может располагаться в отдельном файле;
 - ж) программа на JavaScript обязана располагаться в HTML-файле;
 - з) символы «//» обозначают начало комментария в JavaScript;
 - и) символы «//» обозначают начало комментария в HTML.
2. Напишите название тега HTML, который задает начало скрипта.
3. Напишите название функции JavaScript, которая создает всплывающее информационное окно.

Задания

Проведите серию опытов с функцией `alert`. Задайте вывод сообщения (рис. 5.6).

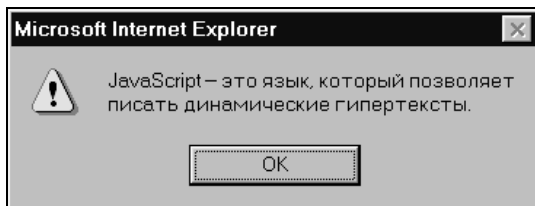


Рис. 5.6. Сообщение в информационном окне

Попробуйте другие тексты, символы перевода строк, запись одной функции на нескольких строчках, запись нескольких функций. Для опытов используйте следующую заготовку (листинг 5.3).

Листинг 5.3

```
<HTML>
  <HEAD>
    <META http-equiv="Content-Type"
      content="text/html; charset=windows-1251">
    <TITLE>Опыты с alert</TITLE>
  </HEAD>
  <BODY bgcolor=white text=black>
    <H2>Опыты с функцией alert</H2>
    <SCRIPT language=JavaScript>
      <!--
        alert("!!!");
      //-->
    </SCRIPT>
  </BODY>
</HTML>
```

Дополнительный материал

Книга, которую вы читаете, посвящена построению динамических гипертекстовых документов. Такие документы почти не отличаются по своим возможностям и внешнему виду от обычных компьютерных программ.

За счет чего на гипертекстовой странице появляется динамика и интерактивность (взаимодействие с пользователем)?

Дело в том, что браузер не просто показывает документ, он конструирует его объектную модель, а в дальнейшем следит за соответствием этой модели построенному изображению. Если что-то меняется в модели, это сразу отображается на экране.

Получается, что окно браузера можно изменить двумя способами:

- ☐ написать HTML-теги и загрузить файл в браузер;
- ☐ воздействовать на объектную модель документа.

Первый способ дает статичный гипертекстовый документ. Второй — позволяет менять уже построенное изображение.

Как же можно воздействовать на объектную модель документа? Вы, конечно, догадались, что для этого надо писать скрипты — программы на специальных языках программирования (их должен «понимать» браузер). Самый распространенный язык для этого — JavaScript.

Уроки 5–10 посвящены программированию на языке JavaScript. Объектная модель документа и способы управления этой моделью описываются в *Уроках 11–14*. Весь остальной материал книги неразрывно с этим связан.

В этом уроке рассмотрен блок `<script>...</script>`. В него «замуровываются» скриптовые программы. Показана функция `alert`, которую часто используют в скриптах.

Эта функция не является встроенной функцией языка JavaScript. Она принадлежит объектной модели документа (объекту `window`), но до *Урока 11* можно об этом не задумываться.

Хорошенько разберитесь с функцией `alert`. Она используется очень часто. Кроме простой задачи выдать пользователю сообщение, функция незаменима как средство отладки. К сожалению, браузер выдает сообщения об ошибках очень бестолково. Приходится расставлять серию вызовов `alert` для вывода промежуточных данных с целью «перехватить» ошибку в месте ее фактического размещения. «Подслеповатый» браузер не всегда выдает даже бестолковые сообщения об ошибках, а если и делает это, то указывает место *проявления* ошибки, а не место ее *расположения*.

JavaScript и Java

JavaScript — это язык программирования, который разработан специально для работы совместно с HTML.

Коды на JavaScript интерпретируются браузером.

Вспомним, что все языки программирования подразделяются на два класса: интерпретируемые и компилируемые.

Для выполнения кода на интерпретируемом языке нужна специальная программа-интерпретатор. Типичный пример интерпретируемого языка — классический Бейсик. Мы пишем программу в текстовом редакторе, затем вызываем Бейсик (интерпретатор Бейсика), и он эту программу выполняет. На некоторых компьютерах Бейсик «зашивается» прямо в постоянную память (например, так было на Ямахе и первых моделях IBM PC). Чаще же интерпретатор — это обычная компьютерная программа, расположенная на жестком или гибком диске.

Пример компилируемого языка — Си. Программа записывается в текстовом редакторе, затем вызывается компилятор, который преобразует текст программы в машинные коды. Заметьте: компилятор, в отличие от интерпретатора, не выполняет программу, а переводит ее текст в машинные коды. Результат компиляции — файл, готовый к самостоятельному исполнению на компьютере.

Для выполнения компилированной программы на компьютере нужна только операционная система. Для выполнения интерпретируемой программы дополнительно нужен еще интерпретатор.

Компилированные программы работают гораздо быстрее, чем выполняемые под управлением интерпретатора.

Следует отметить, что подразделение языков программирования на два отмеченных выше класса довольно условно. Более правильно говорить о *традиционно* компилируемых и *традиционно* интерпретируемых языках. Так, существуют компиляторы Бейсика (традиционно интерпретируемого языка) и интерпретаторы Паскаля (традиционно компилируемого языка).

С Web-программированием обычно связывают еще один язык — Java.

JavaScript и Java — это совсем разные технологии и концепции.

Объединяет их только Си-образный синтаксис языковых конструкций.

JavaScript — это интерпретируемый браузером язык, который был создан для использования в составе HTML-кода. Java — универсальный, частично компилируемый язык программирования.

JavaScript

Коды JavaScript монтируются в HTML так тесно, что часто выступают как значения специальных атрибутов HTML-тегов, реализующих реакцию на какие-то действия пользователя (например, щелчок мыши) или другие *события* (например, конец загрузки документа).

Вот пример «чистого» HTML-кода:

```
<IMG src=bonus.gif>
```

А вот код с использованием JavaScript:

```
<IMG src=bonus.gif onclick="make()">
```

В первом случае браузер просто покажет картинку, а во втором — будет реагировать на щелчок пользователем мышью по этой картинке — запускать на выполнение функцию `make`.

Как видите, запись `onclick="make()"` по форме совпадает с записью обычного атрибута HTML. «click» — это имя события, которое возникает при щелчке мыши. Приставка «on» стандартным образом образует имя «событийного» атрибута. Значением такого атрибута является скрипт, который получает управление в момент свершения события. Как только пользователь щелкнет мышью по картинке, браузер запустит на выполнение функцию `make`. Эта функция может заменить одну картинку на другую, запустить видеоклип, показать текущее время или сделать еще что-нибудь, что в ней заложено программистом.

Получается, что JavaScript расширяет HTML. Этот язык позволяет для «событийных» атрибутов «описание» описывать действия (обработчики событий), т. е. JavaScript предлагает Web-мастеру инструмент для создания «собственного» HTML, пристраивая к стандартному HTML новые «колесики», чувствительные к событиям.

Java

Это универсальный язык программирования общего назначения, который разрабатывали с расчетом на минимальную зависимость от конкретного компьютера и конкретной операционной системы.

Java-программа преобразуется сначала в промежуточный (между текстом и машинными командами) код, который затем интерпретируется (браузером или другим Java-интерпретатором). Этот код называют *байтовым кодом*.

Получается, что Java является частично компилируемым языком: текстовая программа сначала компилируется в байтовый код, а затем байтовый код интерпретируется. Такой механизм позволяет существенно увеличить скорость работы приложения по сравнению с обычной интерпретацией.

Java-программы, предназначенные для интерпретации браузером, называют *апплетами*. Апплет существенно отличается от скрипта по своему проявлению в гипертекстовом документе.

Размещение апплета на странице подобно размещению картинок. Апплет занимает на экране отдельную прямоугольную область и работает внутри нее самостоятельно, без явной связи с другими информационными элементами страницы.

С помощью языка Java легко поместить на страницу совершенно самостоятельную программу со своим интерфейсом и внешним графическим представлением (например, компьютерную игру или бухгалтерский офис). Эта экранная область будет «жить» своей собственной жизнью, отсылая, если это необходимо, данные на сервер.

Рассмотрим пример размещения Java-программы (апплета) на странице гипертекста (листинг 5.4).

Листинг 5.4

```
<HTML>
<HEAD>
<META http-equiv="Content-Type"
      content="text/html; charset=windows-1251">
<TITLE>Пример гипертекста с Java-апплетом</TITLE>
</HEAD>
<BODY bgcolor=white text=black>
  <H1>Пример гипертекста с Java-апплетом</H1>
  <APPLET code="demo.class" width="300" height="200">
  </APPLET>
</BODY>
</HTML>
```

Как видите, апплет задается почти так же, как картинка.

Задаются размеры прямоугольной области, в которой апплет будет работать, и имя файла с байтовым кодом апплета. По принятым соглашениям файл с исходным текстом Java-программы имеет расширение `java`, а байтовый код — расширение `class`.

В теге `<APPLET>` также, как и в теге `<IMG>` может быть использован атрибут `align`, выравнивающий область апплета по отношению к другим информационным элементам страницы, а также атрибут `alt`, задающий альтернативный текст.

Рассмотрим типичные задачи, которые решаются при помощи этих двух языков.

JavaScript:

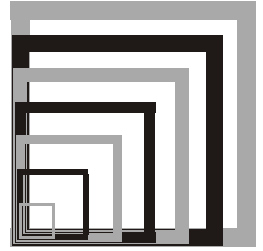
- ☐ Создание HTML-страницы «на лету», программным путем.
- ☐ Обработка простых диалоговых форм.
- ☐ Изменение внешнего вида страницы после ее вывода на экран.

Java:

- ☐ Сложные формы с вводом/выводом графической информации.
- ☐ Анимация (настоящая анимация, а не анимированные GIF-файлы).
- ☐ Трехмерные изображения и их обработка.
- ☐ Выполнение самостоятельной графической программы в среде гипертекстового документа.

Изучение языка Java выходит за рамки этой книги.

Урок 6



Переменные, константы, выражения

Мы уже познакомились с функцией `alert`.

Она предназначена для создания информационного окна. Окно «всплывает» на экране и содержит информацию для пользователя. Единственная кнопка **ОК** позволяет убрать окно с экрана и разрешает браузеру продолжить построение документа.

Аргументом функции `alert` может быть текстовая константа, она заключается в двойные или одинарные кавычки:

```
alert("Офелия, о нимфа! Помяни грехи мои в молитвах.");
```

```
alert('Птичка Божия не знает ни заботы, ни труда.');
```

Кроме того, аргументом функции `alert` может быть переменная и, в общем случае, любое выражение.

Переменные и константы

В JavaScript нет описателей типа, подобных `int`, `char` в языке Си или `Integer`, `Byte` в языке Паскаль. Тип переменной определяется ее текущим значением и может меняться при выполнении программы много раз.

Пример

```
x = "3";           // Сейчас переменная x строкового типа,  
                   // ее значение — строка «3».  
  
x = 3;             // Сейчас переменная x целого типа,  
                   // ее значение — число 3.  
  
x = "Оценка=" + x; // Сейчас переменная x строкового типа,  
                   // ее значение — строка «Оценка=3».
```

В JavaScript можно использовать константы следующих видов.

Целое число

- 255 — десятичное целое.
- 0377 — восьмеричное целое (начинается с «0»).
- 0xFF — шестнадцатеричное целое (начинается с «0x»).

Дробное число

- 3.14 — обычная запись.
- 2.3e8 — «инженерная» запись ($2.3 \cdot 10^8$).

Строка

- "Привет!"
- 'Привет!'
- 'It\'s Ok!' — для кавычек внутри строки используют указания \' и \".
- "\"ВАЗ 2104\""
- "Перва\я строка.\nСтрока вторая\я." — внутри записи можно использовать:
 - \n — перевод строки;
 - \я — буква я;
 - \С — буква С.
- 'Кинотеатр "Ударник"' — внутри одинарных кавычек можно использовать двойные без символа «\» и наоборот.
- "I'm programmer"

Описание переменных

Переменная должна быть описана до ее использования. Для описаний берется ключевое слово **var**:

```
var x; // Описание переменной с именем "x".
var y = 5; // Описание с присвоением числа.
var mes = "дядя Федор"; // Описание с присвоением строки.
```

Для образования имен переменных используются идентификаторы.

Идентификатор — это последовательность из латинских букв и арабских цифр, которая начинается с буквы. В записи разрешается использовать символ подчеркивания «_» наравне с латинскими буквами. Регистр буквы в имени переменной имеет значение. Переменные с именами «Counter» и «counter» считаются разными.

Выражения

Для арифметических операций используются следующие обозначения (табл. 6.1).

Таблица 6.1

Знак	Название	Пример
–	Унарный минус	–10
+	Сложение	$x + y$
–	Вычитание	$x - 10$
*	Умножение	$2 * x$
/	Деление	$y / 7$
%	Остаток от деления	$x \% 3$

Для строковых выражений используется операция соединения (конкатенации), обозначаемая знаком «+»:

<code>person="Кроко- дил" + "Гена";</code>	—	Переменная <code>person</code> получает значение «Крокодил Гена».
<code>x = "2" + 1;</code>	—	Переменная <code>x</code> получает значение «21». Заметьте, «21» здесь не число, а строка! Когда операция «+» связывает строку с данными другого типа, результат преобразуется в строку.

Команды

Команда завершается в JavaScript символом «;». Правила языка разрешают опускать этот знак в конце строки и перед закрывающей скобкой «}».

Комментарий

Наряду с уже известным строковым комментарием «//» можно использовать блочный комментарий «/* ... */»:

```
// Это строковый комментарий.  
/* А это  
   блочный комментарий. */
```

Вызов функции

Пока была показана только функция `alert`:

```
alert(выражение);
```

Аргументом этой функции может быть выражение, имеющее как текстовое, так и числовое значение.

```
var ex1 = "Ищите";  
var ex2 = "обращаете";  
alert(ex1 + " и " + ex2);
```

Результат выполнения скрипта показан на рис. 6.1.

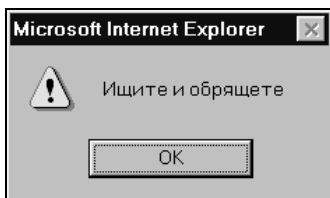


Рис. 6.1. Информационное окно alert

```
var x = 48;  
var ex1 = "попугаев";  
alert(x + " " + ex1);
```

Результат выполнения скрипта показан на рис. 6.2.



Рис. 6.2. Результат выполнения скрипта

```
var x = 2;  
var y = 3;  
alert("Результат умножения: " + x * y);
```

Результат выполнения скрипта показан на рис. 6.3.

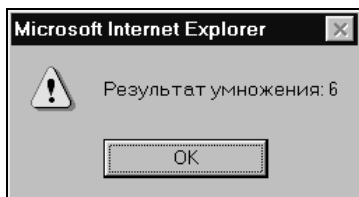


Рис. 6.3. Результат выполнения скрипта

Присваивание

Наряду с привычным присваиванием вида «переменная=выражение», можно использовать следующие записи (табл. 6.2):

Таблица 6.2

Общий вид	Пример	Что означает
переменная ++	x ++	x = x + 1
++ переменная	++ x	x = x + 1
переменная --	x --	x = x - 1
-- переменная	-- x	x = x - 1
переменная += выражение	x += "кот"	x = x + "кот"
переменная -= выражение	x -= 2	x = x - 2
переменная *= выражение	x *= 10	x = x * 10
переменная /= выражение	x /= y + z	x = x / (y + z)
переменная %= выражение	x%=9	x = x%9

Разница между командами ++ x и x ++ (-- x и x --) проявляется тогда, когда эти команды используются как составная часть других команд. В первом случае (знаки перед именем) операция выполняется до использования переменной, а во втором — после.

x = 5; — После выполнения этих команд: x и y равны 6.
y = ++ x;

x = 5; — После выполнения этих команд: x равно 6, а y равно 5.
y = x ++;

Зачетный класс

1. Что будет выведено в окно функции alert в результате выполнения следующих команд: число 10, число 35 или сообщение об ошибке?

а) // var x = 10;
 var x = 35;
 // var x = 8;
 alert(x);

- б) `/* var x = 10;`
`var x = 35;`
`/* var x = 8;`
`alert(x);`
- в) `/* var x = 10;`
`*/var x = 35;/*`
`var x = 8;`
`*/alert(x);`
- г) `/* var x = 10;`
`var x = 35;`
`var x = 8;`
`*/alert(x);`
2. Что будет выведено в окно `alert` в результате выполнения следующих команд?
- а) `var x = "60";`
`var y = 20;`
`alert(x + y);`
- б) `var x = "60";`
`var y = 20;`
`alert(x - y);`
- в) `var x = "60";`
`var y = 20;`
`alert(x / y);`
- г) `var x = "с";`
`var y = "ка";`
`alert(x + "оба" + y);`
- д) `var x = 13;`
`var y = 3;`
`y = ++ x / -- y;`
`alert(y);`
- е) `var x = 13;`
`var y = 3;`
`y = -- x / ++ y;`
`alert(y);`

```
ж)  var x = 13;
    var y = 3;
    y = x ++ % y ++;
    alert(y);
з)  var x = "";
    x += "обед";
    alert("п" + x + "а");
и)  var x = "м";
    x += "а";
    x = "рам" + x;
    var y = "пор";
    alert("п" + y + x);
к)  var str1 = "Все мы немножко лошади,\n";
    var str2 = "Каждый из нас по-своему лошадь.";
    alert(str1 + str2);
л)  var str1 = "Результат вычислений:\n";
    var num1 = 35;
    var num2 = 8;
    alert(str1 + num1 / num2);
м)  var str1 = "Результат вычислений:\n";
    var num1 = 35;
    var num2 = 8;
    alert(str1 + num1 % num2);
н)  var str1 = "Результат вычислений:\n";
    var num1 = 35;
    var num2 = 8;
    alert(str1 + (num1 ++)*(num2 ++));
о)  var str1 = "Результат вычислений:\n";
    var num1 = 35;
    var num2 = 8;
    alert(str1 + (num1 ++ % (num2 ++)));
п)  var str1 = "Результат вычислений:\n";
    var num1 = 12;
    var num2 = 8;
    alert(str1 + num1 + "+" + num2 + "=" + num1 + num2);
```

```
p) var str1 = "Результат вычислений:\n";
    var num1 = 12;
    var num2 = 8;
    alert(str1 + num1 + "+" + num2 + "=" + (num1 + num2));

c) var str1 = "Результат вычислений:\n";
    var num1 = 12;
    var num2 = 8;
    alert(str1 + num1 + "*" + num2 + "=" + num1 * num2);

т) var str1 = "Результат вычислений:\n";
    var str2 = 4;
    var num1 = 12;
    var num2 = 8;
    alert(str1 + num1 + "*" + num2 + "+" + str2 + "=" + num1 * num2 + str
2);

y) var str1 = "Результат вычислений:\n";
    var str2 = 4;
    var num1 = 12;
    var num2 = 8;
    alert(str1 + num1 + "*" + num2 + "+" + str2 + "=" + (num1 * num2 + st
r2));

ф) var str1 = "10";
    var str2 = "5";
    alert(str1 / str2);

х) var str1 = "Этажерка";
    var str2 = 2;
    alert(str1 / str2);
```

Задания

Проверьте на практике ваши ответы на вопросы зачетного класса. Используйте в качестве испытательного стенда следующий образец (листинг 6.1).

Листинг 6.1

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
```

```
<TITLE>Испытательный стенд</TITLE>
</HEAD>
<BODY bgcolor=white text=black>
  <H2>Испытательный стенд</H2>
  <SCRIPT language=JavaScript>
    <!--
      var str1="Этажерка";
      var str2=2;
      alert(str1/str2);
    //-->
  </SCRIPT>
</BODY>
</HTML>
```

Дополнительный материал

Урок знакомит читателя с базовыми элементами языка.

☐ Данные

- переменные
- константы

☐ Выражения

☐ Команды

- комментарий
- команда присваивания
- вызов функции

Переменная

Переменная — это именованная область памяти для хранения данных. Данные могут быть разных типов: целое число, десятичная дробь, логическая константа, текстовая строка. В компилируемых языках (таких как Си, Паскаль) перед использованием переменной она обязательно описывается (либо явно программистом, либо по умолчанию транслятором). Ведь данные каждого типа по-своему кодируются двоичными нулями и единицами в памяти компьютера, и обработка каждого типа происходит по своим алгоритмам.

В интерпретируемом языке JavaScript нет описателей типов данных (таких, как `int`, `float`, `char` в Си). Переменная описывается ключевым словом `var`, которое не вносит никакой типизации. Описатель `var` просто фиксиру-

ет именованную область для хранения данных, не приписывая этой области никакого типа. Это не означает, конечно, что JavaScript одинаково работает с любыми данными. Это показатель того, что тип переменной определяется по типу присваиваемого значения и может много раз меняться в процессе работы программы:

```
var x;    // Описана переменная (фиксируется область для хранения данных).  
x = 10;    // Сейчас переменная хранит целое число 10.  
x = "10";  // Сейчас переменная хранит строку "10".  
x = 30 + 10; // Сейчас переменная хранит число 40.  
x = 30 + "10"; // Сейчас переменная хранит строку "3010".  
x = 30 - "10"; // Сейчас переменная хранит целое число 20.  
x = "30" - 10; // Сейчас переменная хранит целое число 20.  
x = "30" - "10"; // Сейчас переменная хранит целое число 20.
```

Заметьте, что в выражении `30 + "10"` число 30 преобразуется в строку, а операция «+» рассматривается как операция конкатенации (соединение двух строк). В выражении `30 - "10"` строка «10» преобразуется в число 10, после чего выполняется вычитание. Аналогично в выражении `"30" - 10` строка «30» преобразуется в число. Видно, что браузер производит преобразование типов данных по контексту, и иногда не совсем так, как ожидает программист. Нужно быть очень внимательным и не следует полагаться на автоматическое преобразование типов при выполнении реальных программ. Там, где нет полной уверенности, лучше пользоваться явным преобразованием при помощи встроенных в язык функций `parseInt` и `parseFloat` (Урок 9):

```
var y = "30";  
x = parseInt(y, 10) - 10;
```

Описание переменных

Несмотря на то, что JavaScript позволяет описывать переменные в любом месте и даже не описывать их вовсе, описания нужно делать обязательно и собирать их в одном месте (в начале функции, скрипта). Если переменная не описана программистом явно, браузер описывает ее (заводит под нее область памяти) самостоятельно при первом упоминании. При этом переменная считается *глобальной*.

Глобальная переменная доступна в любой части кода текущего HTML-файла. А это очень плохо. Легко случайно испортить ее значение.

Примите на вооружение полезное правило: *переменные нужно описывать всегда*.

Если действительно нужна глобальная переменная, опишите ее явно в первом блоке `<SCRIPT>...</SCRIPT>`, разместив этот блок между тегами

`<HEAD>...</HEAD>`. Глобальные описания должны быть самыми первыми в этом блоке и обязательно вне функций (листинг 6.2).

Листинг 6.2

```
<HTML>
<HEAD>
  <SCRIPT language=JavaScript>
  <!--
    var bonus; // Это глобальная переменная. Она видна во всем файле,
                // а не только в этом скриптовом блоке.
    function ShowBonus()
    {
      var sum;    // Это локальная переменная, видна только в функции
                  // ShowBonus().

      ...

      bonus = ... // Присвоение значения глобальной переменной.

      ...
    }
  //-->
</SCRIPT>
</HEAD>
<BODY bgcolor=white text=black>
...
  <SCRIPT language=JavaScript>
  <!--
    function CalcBonus()
    {
      var sum;    // Это локальная переменная, видна только в функции
                  // CalcBonus().

      ...

      bonus = ... // Это присвоение значения глобальной переменной.

      ...        // Переменная видна в этом скриптовом блоке, хотя
                  // описана в другом.

    }
  //-->
</SCRIPT>
...
```

```
</BODY>
```

```
</HTML>
```

Цепочка из нескольких блоков `<SCRIPT>...</SCRIPT>` равнозначна одному большому скрипту (в смысле видимости переменных и функций). Переменные, описанные вне функций, видны всюду. Такие описания считаются глобальными.

Имена переменных

Выбор имени очень важен. Вот несколько советов.

1. Не именуйте «с потолка».
2. Не именуйте одной буквой.
3. Избегайте аббревиатур и сокращений.
4. Не именуйте зарезервированными словами языка.

Из приведенных правил 2 и 3 есть очевидные исключения:

i, j	—	счетчики циклов;
col	—	индекс столбца;
row	—	индекс строки;
cur	—	текущее значение;
max	—	максимум (обычно в качестве префикса или суффикса: <code>maxCol</code>);
min	—	минимум (обычно в качестве префикса или суффикса: <code>valueMin</code>);
s, str	—	строка (если нельзя ее назвать более содержательно);
buf	—	буфер (временное хранилище);
len	—	длина (обычно как префикс или суффикс: <code>lenStr</code>);
ret	—	возвращаемое из функции значение;
sum	—	сумма.

Из правил 1 и 4 исключений быть не может!

Достаточно сложно уследить за всеми ключевыми словами языка и легко попасть в ловушку. Однажды Сидоров назвал функцию, обрабатывающую щелчок на кнопке «Старт», очень подходящим именем `start`, и в браузерах четвертых версий все работало прекрасно. После появления Microsoft Internet Explorer 5 на Сидорова посыпались рекламации: гипертекстовое приложение не работало. Причина оказалась именно в имени `start`, которое в IE-5 стало ключевым словом. С тех пор Сидоров принял для себя правило: добавлять к «нормальным» именам один и

тот же предохранительный суффикс. Он переименовал `start` в `startRU`, `pointer` в `pointerRU` и все стало прекрасно работать.

5. Не бойтесь длинных имен, хотя они и увеличивают объем программы.

Мера, конечно, нужна во всем. Но я за понятные программы в ущерб некоторой эффективности.

Как именовать Горбункова Семена Семеновича?

Есть два способа:

а) ГорбунковСеменСеменыч (горбунковСеменСеменыч);

б) горбунков_семен_семеныч.

Представьте, что мы оставили бы от Семена Семеновича одну аббревиатуру «гсс» — никто бы Горбункова и не признал. С трудом бы узнали его и за шифром «ГорСеСе».

Какой способ именования из а) и б) вам больше по душе? Выбирайте! Но старайтесь придерживаться одной нотации на протяжении всей программы, а лучше — всего программирования.

6. Избегайте ненужных имен.

Это вариант уродлив и избыточен:

```
var ret = "ok!";  
if (...хорошо...) return ret;
```

Лучше записать так:

```
if (...хорошо...) return true;
```

7. Избегайте магических чисел.

Константы в программах нельзя оставлять без имен.

При суммировании всех элементов массива `set` лучше всего воспользоваться свойством `set.length`, указывающим длину массива:

```
// Суммирование всех элементов массива.  
// set.length — свойство массива: его длина.  
var sum = 0;  
for (var i = 0; i < set.length; i++) sum += set[i];
```

Суммирование первой сотни элементов можно задать так (листинг 6.3).

Листинг 6.3

```
var bonus = new Array(...); // Массив оценок.  
var important_length = 100; // Число важных оценок (они идут первыми).  
var middle_important_bonus; // Средняя оценка среди важных.  
// Суммирование важных оценок.
```

```
middle_important_bonus = 0;
for (var i = 0; i < important_length; i ++)
    middle_important_bonus += bonus[i];
// Нахождение средней оценки среди важных.
middle_important_bonus /= important_length;
```

Что это дает?

Если программа почему-то работает с первыми 100 числами массива, то, вероятно, в этом есть сермяжная правда. И это число «100» будет встречаться в программе много раз (у нас уже два раза).

Если через месяц важными окажутся не 100 элементов, а 200, придется кропотливо перечитывать уже забытый код, чтобы всюду заменить 100 на 200. Много шансов, что некоторые замены не будут выполнены (там, например, где стоит 99, которое означает `important_length-1`), а некоторые будут выполнены неправильно (например, число 100, означающее проценты, заменится на 200).

Таких неприятностей можно легко избежать, если именовать константы — замена будет производиться только в одном месте, да и читать такую программу будет гораздо легче.

Магических чисел в программе быть не должно, но 0 и 1 — очевидные исключения.

В этом суммировании массива:

```
var sum = 0;
for (var i = 0; i < set.length; i ++) sum += set[i];
```

И в этом нахождении произведения:

```
var num = 10;
var factorial = 1;
for (var i = 1; i <= num; i ++) factorial *= i;
```

константы 0 и 1 — в самом деле, числа магические! Именовывать их не надо!

Существует практика именования переменных русскими словами, записанными латинскими буквами. С одной стороны, в этом нет ничего плохого:

```
// Возвращает НОД чисел a и b.
function Nod(a,b)
{
    .....
}

var summa = 0; // Сумма элементов массива.
```

```
var dlina = 10; // Длина актуальной части массива.  
var poz;      // Текущая позиция в строке.
```

Однако в таком виде удастся написать гораздо меньше красивых и понятных имен, чем на английском языке.

Сравните:

```
var counter; // Счетчик чего-то...  
var schetchik; // Счетчик чего-то...
```

```
var bonus; // Оценка за урок.  
var ozenka; // Оценка за урок.
```

```
function number_of_elements() // Число элементов.  
function chislo_elementov() // Число элементов.
```

```
function error_message() // Сообщение об ошибке.  
function soobchenie_ob_osibke() // Сообщение об ошибке.
```

Совет такой: на этапе написания кода держите под рукой маленький русско-английский словарь и пользуйтесь им при выборе подходящего имени. Заодно потихоньку пойдет обучение словам, которые все равно нужно знать по-английски тем, кто занимается программированием и компьютерами.

И еще: хорошей практикой является комментирование имен при их описании.

Конечно, не надо комментировать переменную цикла! Описание остальных имен надо помещать в одном месте (в начале) и сопровождать максимально подробными комментариями.

Описание функции нужно также предварять подробными комментариями, в которых обязательно указывать:

- ☐ назначение функции;
- ☐ аргументы и их допустимые значения;
- ☐ возвращаемые значения;
- ☐ особенности работы функции.

```
// Вычисление НОД(a,b).  
// Вход: a и b – натуральные числа.  
// Выход: НОД(a,b) или 0, если a и b не являются допустимыми.
```

```
function Nod(a,b)  
{  
    ...  
}
```

Но даже несмотря на комментарии, для выбора содержательных, понятных и красивых имен не надо жалеть времени. Ведь, если одна переменная называется «kfgm», а другая «krfgm», то читать программу практически невозможно, даже если эти переменные подробным образом откомментированы в начале скрипта.

Команда присваивания

Команда присваивания имеет вид: `переменная = выражение`.

В любых языках программирования эта конструкция работает так:

1. Вычисляется `выражение`.
2. Результат становится значением `переменной`.

Рассмотрим сложный пример из задания 2 е) зачетного класса урока:

```
var x = 13;  
var y = 3;  
y = x ++ % y ++;
```

Работает это так:

1. Вычисляется арифметическое выражение `x ++ % y ++`
 - сначала вычисляется `x % y` (операции `++` здесь работают после использования переменных `x` и `y`): `13 % 3` равно 1;
 - выполняются операции `++` (ведь они в составе арифметического выражения, значит, должны быть выполнены *до* присваивания): значение `x` теперь равно 14, а значение `y` — 4.

2. Выполняется присваивание `y = 1` (прежнее значение 4 пропадает).

Теперь переменная `x` имеет значение 14, а переменная `y` — значение 1.

Операция присваивания

Сюрприз для тех, кто пишет на Паскале.

Оказывается, в Си и вслед за ним в JavaScript команда присваивания `переменная = выражение` одновременно является бинарной операцией. Результат этой операции — значение, полученное `переменной`.

Если мы пишем:

```
x = 1;  
y = x + 1;
```

— то можно и не думать о второй ипостаси команды. Но эти две команды можно кодировать как одну:

```
y = (x = 1) + 1;
```

Работает это так:

1. Сначала выполняется $(x = 1)$. Переменная x получает значение 1. Но этого мало! Вместо скобки $(x = 1)$ в выражение подставляется результат выполнения операции, т. е. число 1.
2. Теперь вычисляется выражение $1 + 1$ и переменная y получает значение 2.

А что, если опустить скобки?

```
y = x = 1 + 1;
```

Что произойдет теперь?

1. Браузер выдаст сообщение о синтаксической ошибке?
2. Переменные x и y получают значение 2?
3. Переменная x получит значение 1, а y — значение 2?

Правильный ответ дает второй вариант, т. е. x и y получают значение 2.

Из таблицы приоритетов операций JavaScript (см. справочный раздел в *Приложении 2*) следует:

- приоритет операции « $=$ » — самый низкий;
- операции « $=$ » выполняются справа налево.

Это означает, что в последнем примере:

1. Сначала выполнится операция сложения.
2. Результат 2 присвоится переменной x .
3. Переменная y получит значение результата предыдущей операции присваивания, т. е. число 2.

Команды:

```
Переменная += выражение;
```

```
Переменная -= выражение;
```

```
Переменная *= выражение;
```

```
Переменная /= выражение;
```

```
Переменная %= выражение;
```

тоже имеют статус операции. Значение этой операции совпадает со значением, которое получает переменная.

В следующем примере:

```
var x = 1;
```

```
var y;
```

```
y = (x += 2) + 1;
```

переменная y получит значение 4, а переменная x — значение 3.

Наиболее популярно использование операции « $=$ » в цепочке присваиваний:

```
x = y = z = t = выражение;
```

Здесь нескольким переменным присваивается одно и тоже значение.

Можно увидеть и такие записи:

1. `y = f(x = 1);`
2. `z = k(x += 1);`
3. `x = 1; y = 2; y += x += 1 + y;`
4. `y = !(x = 2);`

Подумайте, что они означают!

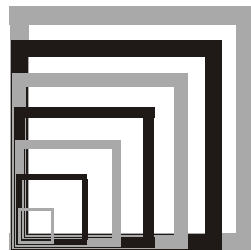
А вот настоящее «безумие»:

```
y = ++ (x = 1);
```

К счастью, оно не работает (браузер выдает сообщение об ошибке).

Операции «++» и «--» применимы только к переменным, но не к выражениям.

Урок 7



Ветвления

Функция `alert` предназначена для передачи информации пользователю. А как пользователь может передать информацию в документ?

Ввод информации

Для ввода информации можно использовать функцию `prompt`:

```
prompt("надпись", "значение строки ввода по умолчанию");
```

Функция работает так. На экран выводится окно с двумя кнопками внутри (рис. 7.1).



Рис. 7.1. Вид окна `prompt`

Пользователь записывает сообщение в строке ввода и нажимает кнопку **ОК**. Окно убирается с экрана, а введенный текст становится значением функции `prompt`. Это значение можно, например, присвоить переменной и использовать затем в скрипте. Если пользователь нажимает кнопку **Отмена** (Cancel), функция `prompt` возвращает специальное значение `null` (это не строка `"null"`, не пустая строка `""`, а значение специального служебного типа данных).

Пример

```
var str = prompt("Сколько будет 2 x 2?", "5");  
if (str == "4") alert("Гениально! Ответ и правда 4!");  
else alert("Шутка хороша, когда ее понимают!");
```

При запуске этого скрипта на экран выводится окно, изображенное на рис. 7.2.



Рис. 7.2. Сколько будет 2 x 2?

Если ввести число 4 и нажать кнопку **ОК**, скрипт продолжит работу так, как показано на рис. 7.3.

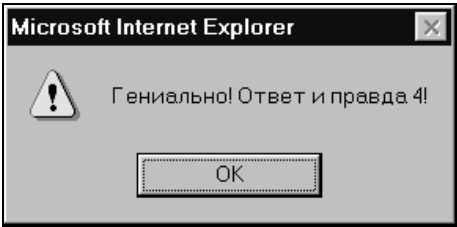


Рис. 7.3. Ответ и правда 4

Если оставить в поле ввода число 5 (или ввести любое значение, не равное 4), на экране появится сообщение, показанное на рис. 7.4.

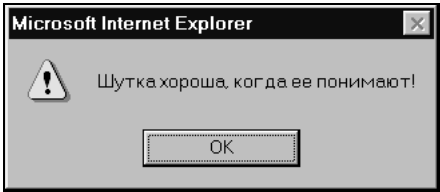


Рис. 7.4. Шутка хороша, когда ее понимают!

Условная команда

Общий вид условной команды приведен в табл. 7.1.

Таблица 7.1

Общий вид	Пример
<code>if (условие) команда1</code>	<code>if (a > b) c = a;</code>
<code>else команда2</code>	<code>else c = b;</code>

Условная команда работает как обычно:

1. Проверяется условие.
2. Если условие истинно, выполняется команда1, иначе — команда2.

Команда1 и команда2 никогда не выполняются вместе — работает только одна из них. После выполнения условной команды управление передается на команду, которая следует за условной.

После выполнения кода:

```
x = 1;
if (x == 1) y = 10;
else      y = 20;
```

```
x += y;
```

переменная *x* получит значение 11. А после выполнения кода:

```
x = 1;
if (x != 1) y = 10;
else      y = 20;
```

```
x += y;
```

переменная *x* получит значение 21.

Условную команду разрешается записывать без ветви **else** (табл. 7.2).

Таблица 7.2

Общий вид	Пример
if (условие) команда1	if (x < 0) x = -x;

Работает эта конструкция так:

1. Проверяется условие.
2. Если условие истинно, выполняется команда1.

После выполнения кода:

```
x = 1;
y = 10;
if (x == 1) y += 10;
```

```
x += y;
```

переменная *x* получит значение 21. А после выполнения кода:

```
x = 1;
y = 10;
if (x != 1) y += 10;
```

```
x += y;
```

переменная *x* получит значение 11.

Конструкции команда1 и команда2 могут быть любыми командами языка, в том числе и снова условными.

Переключатель

Часто используются конструкции следующего вида (переключатели) (табл. 7.3).

Таблица 7.3

Общий вид		Пример
<code>if</code> ком1;	(условие1)	<code>if</code> (Day == 1) NameDay = "Понедельник";
<code>else</code> ком2;	<code>if</code> (условие2)	<code>else if</code> (Day == 2) NameDay = "Вторник";
<code>else</code> ком3;	<code>if</code> (условие3)	<code>else if</code> (Day == 3) NameDay = "Среда";
...		<code>else if</code> (Day == 4) NameDay = "Четверг";
		<code>else if</code> (Day == 5) NameDay = "Пятница";
<code>else</code> комN;		<code>else if</code> (Day == 6) NameDay = "Суббота";
		<code>else if</code> (Day == 7) NameDay = "Воскресенье";
		<code>else</code> NameDay = "ошибка";

Пример

```
var str = prompt("Сколько будет 2 x 2?", "5");
if (str == null) alert("Не стали вычислять?.. Зря!");
else if (str == "4") alert("Гениально! Ответ и правда 4!");
else alert("Шутка хороша, когда ее понимают!");
```

Разработчики JavaScript добавили в язык специальную конструкцию `switch`, которую «понимают» браузеры Netscape Navigator и Internet Explorer, начиная с четвертых версий. Работает эта структура аналогично подобной конструкции в языках Си и Java (табл. 7.4).

Таблица 7.4

Общий вид	Пример
<code>switch</code> (выражение)	<code>switch</code> (Day)
{	{
case вариант1:	case 1:
break;	NameDay = "Понедельник";

Таблица 7.4 (окончание)

Общий вид	Пример
<pre>case вариант2: команды break; ... default: команды }</pre>	<pre>break; case 2: NameDay = "вторник"; break; case 3: NameDay = "среда"; break; case 4: NameDay = "четверг"; break; case 5: NameDay = "пятница"; break; case 6: NameDay = "суббота"; break; case 7: NameDay = "воскресенье"; break; default: NameDay = "ошибка"; }</pre>

Управление передается на команды, расположенные за первым по порядку **case**, в котором значение варианта совпадает со значением выражения, записанного в заголовке **switch** (выражение).

Команда **break** передает управление на первую команду, следующую за конструкцией **switch**. Если команда **break** отсутствует, проверка вариантов продолжается. Вариант **default** (он может отсутствовать) срабатывает тогда, когда значение выражения не соответствует ни одному из вариантов **case**.

Логические операции

В качестве условия можно записывать логические выражения. При этом разрешается использовать следующие логические операции (табл. 7.5).

Таблица 7.5

Обозначение	Описание	Пример
==	Равно	<code>x + 1 == 8</code>
!=	Не равно	<code>str != "да"</code>
>	Больше	<code>x * y > 5</code>
>=	Больше или равно	<code>d >= 0</code>
<	Меньше	<code>num < 10</code>
<=	Меньше или равно	<code>bonus <= 5</code>
&&	Логическое И	<code>1 < x && x < 10</code>
	Логическое ИЛИ	<code>x == 1 x == 10</code>
!	Логическое отрицание	<code>!(1 < x && x < 10)</code>

Блок

Цепочка команд, заключенная в фигурные скобки {команда1; команда2; ...}, рассматривается как одна команда. Такая сложная команда называется составной, или блоком. Символ «;» за скобкой «}», закрывающей блок, не ставится. Условная команда с использованием блока может выглядеть так (табл. 7.6):

Таблица 7.6

Возможный общий вид	Пример
<code>if (условие)</code>	<code>if (d > 0)</code>
<code>{</code>	<code>{</code>
...	<code> x1 = -(b + Math.sqrt(d)) / (2 * a);</code>
команды	<code> x2 = -(b - Math.sqrt(d)) / (2 * a);</code>
...	<code> mes = "два корня";</code>
<code>}</code>	<code>}</code>

Таблица 7.6 (окончание)

Возможный общий вид	Пример
<code>else</code> <code>{</code> ... команды ... <code>}</code>	<code>else if (d == 0)</code> <code>{</code> <code>x1 = -b / (2 * a);</code> <code>mes = "один корень";</code> <code>}</code> <code>else mes = "нет корней";</code>

Конструируем небольшой экзамен

Экзамен состоит из пяти вопросов. За каждый верный ответ начисляется один балл. В итоге выставляется оценка от 2 до 5 по правилу:

- ☐ оценка равна 2, если верных ответов меньше 3;
- ☐ оценка равна числу верных ответов в противном случае.

Программа экзамена по логическим выражениям приводится в листинге 7.1.

Листинг 7.1

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
  <TITLE>Экзамен по логическим выражениям</TITLE>
</HEAD>
<BODY bgcolor=white text=black>
  <H1>Экзамен по логическим выражениям</H1>
  <HR>
  <SCRIPT language=JavaScript>
  <!--
    var bonus = 0; // Счетчик верных ответов.
    var num = 1;  // Номер вопроса.
    var question; // Вопрос.
    // 1-й вопрос.
    question = "Вопрос " + num + ". Верных ответов = " + bonus +
```

```

        ".\nЕсли x=5, то 1<x &&&& x<10 равно true или false?";
    if(prompt(question, "true") == "true") bonus++;
    num++;
    // 2-й вопрос.
    question = "Вопрос " + num + ". Верных ответов = " + bonus +
        ".\nЕсли x=5, то !(1<x &&&& x<10) равно true или false?";
    if(prompt(question, "true") == "false") bonus++;
    num++;
    // 3-й вопрос.
    question = "Вопрос " + num + ". Верных ответов = " + bonus +
        ".\nЕсли x=5, то x != 5 равно true или false?";
    if(prompt(question, "true") == "false") bonus++;
    num++;
    // 4-й вопрос.
    question = "Вопрос " + num + ". Верных ответов = " + bonus +
        ".\nЕсли x=5, то x++ != 5 равно true или false?";
    if(prompt(question, "true") == "false") bonus++;
    num++;
    // 5-й вопрос.
    question = "Вопрос " + num + ". Верных ответов = " + bonus +
        ".\nЕсли x=5, то ++x != 5 равно true или false?";
    if(prompt(question, "true") == "true") bonus++;

    // Показ результата в окне alert.
    if(bonus < 2) bonus = 2;
    alert("Ваша оценка за работу: " + bonus);
    //-->
</SCRIPT>
<P>Экзамен закончен!
</BODY>
</HTML>

```

Построенный экзамен будет иметь вид, показанный на рис. 7.5.

Замечание

Для браузеров NN и IE четвертых версий для вывода на экран одного символа «&» приходится записывать «&&». Конструкция «\&» не помогает.

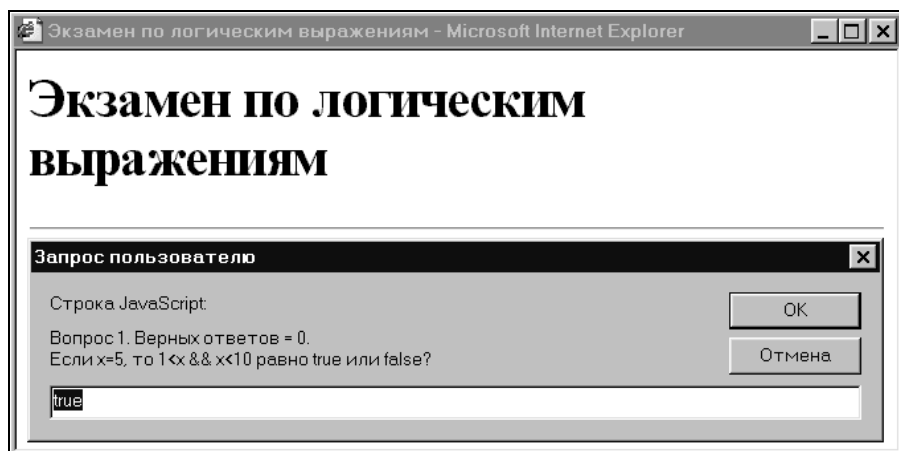


Рис. 7.5. Экзамен по логическим выражениям

Арифметические и строковые выражения как условия

В качестве условия в команде `if` можно записывать не только логические, но и арифметические, и даже строковые выражения. При этом считается, что условие ложно, если значение выражения равно 0 или пустой строке "", и истинно в противном случае (табл. 7.7).

Таблица 7.7

Пример	Пояснение
<code>var x = 20;</code> <code>if (20 - x) x ++;</code>	Условие ложно, значение x не изменится
<code>var x = 20;</code> <code>if (2 * x) x ++;</code>	Условие истинно, значение x увеличится на 1
<code>var x = "";</code> <code>if (x) x += "киса";</code>	Условие ложно, значение x не изменится
<code>var x = "";</code> <code>if (x + "?") x += "киса";</code>	Условие истинно, x получит значение «киса»
<code>var x = "";</code> <code>if (!x) x += "киса";</code>	Условие истинно, x получит значение «киса»

Значения логического типа

В языке JavaScript переменная может принимать значение логического типа. Этот тип имеет две константы, обозначаемые как **true** и **false** (табл. 7.8).

Таблица 7.8

Пример	Пояснение
<code>var x = true;</code> <code>var y = 2;</code> <code>if(x) y ++;</code>	После выполнения команд значение y равно 3
<code>var x = true;</code> <code>var y = 2;</code> <code>if(!x) y ++;</code>	После выполнения команд значение y равно 2
<code>var x = true;</code> <code>var y = 2;</code> <code>if(x !x) y ++;</code>	После выполнения команд значение y равно 3
<code>var x = true;</code> <code>var y = 2;</code> <code>if(x && !x) y ++;</code>	После выполнения команд значение y равно 2

Трудные условия

Условия, в состав которых входят арифметические и строковые выражения, на первых порах вызывают трудности. Рецепт их преодоления очень прост. Нужно помнить, что арифметические и строковые выражения, входящие в состав условия, сначала вычисляются, а затем их значения сопоставляются с логическими значениями **true** и **false**.

Строковые константы «волк» и «крокодил», а также числовые константы 25 и 3.14 эквивалентны одной и той же логической константе **true**. Следовательно, логические отрицания этих величин равны **false** (табл. 7.9).

Таблица 7.9

Пример	Пояснение
<code>if("волк")</code>	То же самое, что <code>if(true)</code>
<code>if("крокодил")</code>	То же самое, что <code>if(true)</code>

Таблица 7.9 (окончание)

Пример	Пояснение
<code>if (25)</code>	То же самое, что <code>if (true)</code>
<code>if (3.14)</code>	То же самое, что <code>if (true)</code>
<code>if (!"волк")</code>	То же самое, что <code>if (false)</code>
<code>if (!"крокодил")</code>	То же самое, что <code>if (false)</code>
<code>if (!25)</code>	То же самое, что <code>if (false)</code>
<code>if (!3.14)</code>	То же самое, что <code>if (false)</code>

Несколько более сложных примеров (табл. 7.10):

Таблица 7.10

Пример	Пояснение	
<code>"волк" && 25</code>	Эквивалентно <code>true && true</code>	Равно <code>true</code>
<code>!"волк" && 25</code>	Эквивалентно <code>!true && true</code>	Равно <code>false</code>
<code>!"волк" 25</code>	Эквивалентно <code>!true true</code>	Равно <code>true</code>
<code>!"" 22 - 22</code>	Эквивалентно <code>true false</code>	Равно <code>true</code>
<code>!("" + "кот") 22 - 22</code>	Эквивалентно <code>false false</code>	Равно <code>false</code>

Поясним последний пример цепочкой преобразований:

```
!("" + "кот") || 22 - 22
!"кот" || 0
!true || false
false || false
false
```

Ввод альтернативы

Если нужно, чтобы пользователь выбрал один из двух вариантов ответа на вопрос, то можно воспользоваться функцией `prompt` (рис. 7.6).

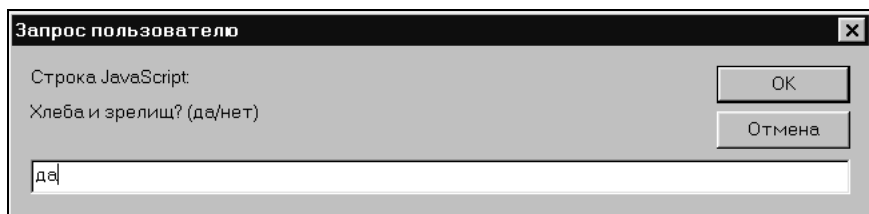


Рис. 7.6. Запрос при помощи окна `prompt`

Существует, однако, более простой и красивый способ узнать желание пользователя — функция `confirm`:

```
confirm("надпись");
```

Эта функция работает так. На экран выводится окно с двумя кнопками (рис. 7.7):

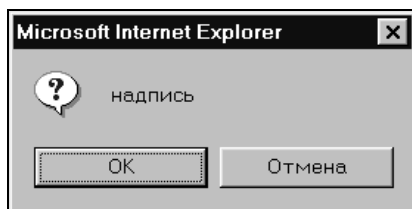


Рис. 7.7. Вид окна `confirm`

Если пользователь нажимает кнопку **ОК**, функция `confirm` возвращает значение `true`, если кнопку **Отмена** (Cancel) — значение `false`. Можно использовать эквивалентные клавиши `<Enter>` и `<Esc>`.

Пример

```
if(confirm("Хлеба и зрелищ?")) alert("Смотри сейчас, а кушай после!");  
else alert("И, словом, тот хотел арбуза,\n" +  
        "А тот соленых огурцов.");
```

При запуске этого кода на экран выводится окно, показанное на рис. 7.8:

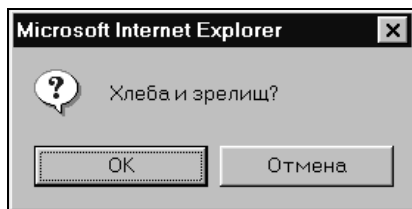


Рис. 7.8. Хлеба и зрелищ?

Если нажать кнопку **ОК**, скрипт продолжит работу так, как показано на рис. 7.9.

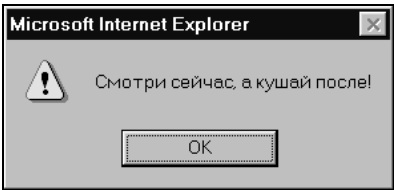


Рис. 7.9. После выбора **ОК**

Если нажать на кнопку **Отмена**, то так, как на рис. 7.10:

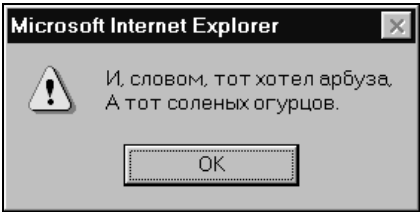


Рис. 7.10. После выбора **Отмена**

Зачетный класс

Какое значение будет иметь переменная *z* после выполнения скрипта (табл. 7.11)?

Таблица 7.11

Пример 1	Пример 2
<pre>var x = "каша"; var y = "лот"; var z = "акула"; if (x) z = x + y;</pre>	<pre>var x = "каша"; var y = "лот"; var z = "акула"; if (!x) z = x + y;</pre>
Пример 3	Пример 4
<pre>var x = "каша"; var y = "лот"; var z = "акула"; if (x && y) z = x + y;</pre>	<pre>var x = "каша"; var y = "лот"; var z = "акула"; if (!x && y) z = x + y;</pre>

Таблица 7.11 (продолжение)

Пример 5	Пример 6
<pre> var x = "каша"; var y = "лот"; var z = "акула"; if(!x y) z = x + y; </pre>	<pre> var x = "каша"; var y = "лот"; var z = "акула"; if(!(x y)) z = x + y; </pre>
Пример 7	Пример 8
<pre> var x = -10; var y = 10; var z = x + y; if(z) z = 20; </pre>	<pre> var x = -10; var y = 10; var z = x + y; if(!z) z = y - x; </pre>
Пример 9	Пример 10
<pre> var x = -10; var y = 10; var z = x + y; if(x < y) z = y - x; </pre>	<pre> var x = -10; var y = 10; var z = x + y; if(x <= y) z = y - x; </pre>
Пример 11	Пример 12
<pre> var x = -10; var y = 10; var z = x / y; if(x + y x < 0) z = y - x; </pre>	<pre> var x = -10; var y = 10; var z = x / y; if(x + y x < 0) z = y - x; </pre>
Пример 13	Пример 14
<pre> var x = -10; var y = 10; var z = x / y; if(x + y && x < 0) z = y - x; </pre>	<pre> var z; var x = -10; var y = 10; if(x + y && y < 0) z = y - x; else z = x + y; </pre>

Таблица 7.11 (продолжение)

Пример 15	Пример 16
<pre> var z; var x = -10; var y = 10; if(x + y y < 0) z = y - x; else z = x + y; </pre>	<pre> var z; var x = -10; var y = 10; if(x + y == 0 y < 0) z = y - x; else z = x + y; </pre>
Пример 17	Пример 18
<pre> var x = 1; var z = 2; if(x --) z = 5; </pre>	<pre> var x = 1; var z = 2; if(-- x) z = 5; </pre>
Пример 19	Пример 20
<pre> var x = 1; var y = x ++; var z = 0; if(x > y) z = x; </pre>	<pre> var x = "эта"; var y = "жерка"; var z = x + y; if(z == "этажерка") z = x + "полка"; </pre>
Пример 21	Пример 22
<pre> var x = "эта"; var y = "жерка"; var z = x + y; if(z != "этажерка") z = x + " полка"; </pre>	<pre> var x = true; var y = false; var z = 1; if(x && y) z = 2; </pre>

Таблица 7.11 (окончание)

Пример 23	Пример 24
<pre>var x = true; var y = false; var z = 1; if(x && !y) z = 2;</pre>	<pre>var x = true; var y = false; var z = 1; if(x y) z = 2;</pre>
Пример 25	Пример 26
<pre>var x = true; var y = false; var z = 1; if (!x && !y) z = 2;</pre>	<pre>var x = 10; var y = 5; var z = 1; if(x < 1 && x < 20) z = y;</pre>

Задания

1. Проверьте на практике ваши ответы на вопросы зачетного класса.
2. Создайте экзамен для проверки знаний по теме «условная команда». Среди заданий экзамена возможны такие:
 - а) `var x = 5; if(x != 5) x ++;` Теперь `x = ?`
 - б) `var x = 5; if(x == 5) x ++;` Теперь `x = ?`
 - в) `var x = 5; var y = x ++; if(y > x) x++;` Теперь `x = ?`
 - г) `var x = 5; var y = ++x; if(y > x) x++;` Теперь `x = ?`
 - д) `var x = 5; var y = x - 5; if(y == 0 || x > 0) x++;` Теперь `x = ?`

Дополнительный материал

Функции *prompt* и *confirm*

Эти функции, как и функция `alert`, не являются встроенными функциями JavaScript. Это — методы объекта `window`. Однако до *Урока 11* на это можно не обращать внимания.

Переключатель

Запись:

```
if      (условие1) команда1
else if (условие2) команда2
....
else if (условиеN) командаN
else
      команда0
```

— гарантирует, что при выполнении *i*-го условия будет выполнена *i*-я команда и только она. Если ни одно из условий не выполняется, будет выполнена команда0. Можно использовать и сокращенную форму:

```
if      (условие1) команда1
else if (условие2) команда2
....
else if (условиеN) командаN
```

Хочется предостеречь от типичной ошибки. Начинающие программисты пишут для переключателя вместо вложенных друг в друга условных команд их последовательную цепочку:

```
if (условие1) команда1
if (условие2) команда2
...
if (условиеN) командаN
```

Они считают, например, что после выполнения кода:

```
x = 1;
if (x == 1) x += 1;
if (x == 2) x += 2;
if (x == 3) x += 3;
if (x == 4) x += 4;
```

переменная *x* получит значение 2. На самом деле переменная *x* получит значение 8.

Этого не произойдет, если написать:

```
x = 1;
if      (x == 1) x += 1;
else if (x == 2) x += 2;
else if (x == 3) x += 3;
else if (x == 4) x += 4;
```

Теперь переменная *x* действительно получит значение 2.

Блок

Обычно блок применяют там, где по правилам синтаксиса разрешается использовать только одну команду, а нужно больше (ведь блок — это одна команда!).

Новичку кажется, что после выполнения кода:

```
x = 1
```

```
if (x != 1) x += 1; x = -x;
```

значение переменной `x` останется равным 1. На самом деле `x` получит значение -1. Для того чтобы обе команды относились к ветви **true**, их надо «погрузить» в блок:

```
x = 1
```

```
if (x != 1) { x += 1; x = -x; }
```

Условия

Эта одна из самых сложных тем для тех, кто ранее не работал на языке Си.

Запись `if (x + y && !z) z = y - x;` для программиста, работающего на Паскале, очень непривычна. Эта команда переписывается в эквивалентном виде так: `if (x + y != 0 && z == 0) z = y - x;` и означает следующее: «Если сумма `x + y` не равна нулю, а `z` равно нулю, то `z` присваивается значение `y - x`».

Но давайте разбираться по порядку.

Операция `&&` — (логическое «и») является бинарной операцией и приводит к **true**, когда оба операнда имеют значение **true**.

Выражение `a && b` — есть **true** тогда и только тогда, когда и `a` есть **true**, и `b` есть **true**.

В этом смысле запись `x + y != 0 && z == 0` становится совершенно понятной, если с учетом приоритета операций, переписать ее в виде:

```
(x + y != 0) && (z == 0)
```

Самое сложное место здесь — это соотнесение выражения `x + y` с выражением `x + y != 0`. Ведь когда в качестве условия используется арифметическое выражение, то его значение расценивается как **true**, если оно отлично от нуля, и как **false**, если оно равно нулю. Иными словами, логическое значение арифметического выражения определяется после его вычисления и сравнения результата с нулем.

Пусть `ав` — любое арифметическое выражение. Тогда запись `if (ав)` совершенно эквивалентна записи `if (ав != 0)`.

Вы можете мысленно заменять `ав` в условиях на `ав != 0` до тех пор, пока не привыкнете.

Число 0, используемое в качестве условия, дает значение **false**, а числа 1, -1, 28, -5.78 и любые другие (кроме нуля) — значение **true**.

Вот пример бесконечного цикла:

```
while(1) { ... }
```

А вот пример цикла, тело которого не выполнится ни разу:

```
while(0) { ... }
```

Выражение `x + y`, используемое в качестве условия, означает совершенно то же, что `x + y != 0`.

Рассмотрим теперь условие `!z`. Оно эквивалентно условию `!(z != 0)`, а последнее — `z == 0`.

Запись `!z` означает отрицание логического значения переменной `z`. Если `z` равно нулю (**false**), то `!z` есть **true**. Если `z` не равно нулю (**true**), то `!z` есть **false**.

0 есть **false**, а `!0` есть **true**

1 есть **true**, а `!1` есть **false**

25 есть **true**, а `!25` есть **false**

В качестве условий можно использовать не только арифметические выражения, но и строковые. Логическое значение строки определяется после ее сравнения с пустой строкой (пустая строка ведет себя в этом смысле как ноль).

Пусть `str` — любое строковое выражение. Тогда запись `if(str)` совершенно эквивалентна записи `if(str != "")`.

Вы можете мысленно заменять `str` в условиях на `str != ""` до тех пор, пока не привыкнете.

Вот пример бесконечного цикла:

```
while("Дядя" + "Вася") { ... }
```

А вот пример цикла, тело которого не выполнится ни разу:

```
while("") { ... }
```

Типичная ошибка при записи условия

С этой ошибкой постоянно сталкиваются как новички, так и опытные программисты. Она «вылезает» в кодах как бы сама собой. Но опытный программист находит эту ошибку сразу, а новичок может потратить на ее поиски дни.

После выполнения кода:

```
x = 1;  
if (x = 2) x ++;
```

значение переменной `x` не останется равным единице, а станет равным 3. Ведь в условие записана не операция сравнения (она записывается как `«==»`), а операция присваивания.

Таблица приоритетов операций

В табл. 7.12 указан приоритет операций.

Таблица 7.12

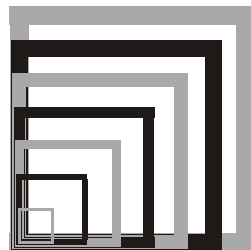
Операция	Название	Порядок выполнения
()	Группирование	От внутренних к внешним
! - + ++ --	Логическое отрицание, унарные минус и плюс, увеличение и уменьшение на 1	Справа налево
* / %	Умножение и деление	Слева направо
+ -	Сложение и вычитание	Слева направо
< <= >= >	Операции сравнения	Слева направо
== !=	Сравнение на равенство и неравенство	Слева направо
& &	Логическое «и»	Слева направо
	Логическое «или»	Слева направо

Приоритет понижается сверху вниз.

Например, выражение `+++ x` равносильно выражению `+(++ x)`, но такие записи лучше не использовать. Когда возникают сомнения в порядке выполнения операций, применяются круглые скобки.

Более полная таблица приоритетов операций приводится в справочном разделе *Приложения 2*.

Урок 8



Повторения

Напомним команды, которые рассматривались на предыдущих уроках (табл. 8.1).

Таблица 8.1

Команда	Пример
Команда присваивания	<code>result = x * y;</code>
Условная команда	<code>if (x) z ++;</code>
Блок (составная команда)	<code>{ c = x; x = y; y = c; }</code>

Продолжим знакомство с командами JavaScript.

Команды цикла

Цикл *while*

В табл. 8.2 показан общий вид цикла *while* и пример его использования.

Таблица 8.2

Общий вид	Пример
<code>while (условие) команда;</code>	<pre>var i = 1; var sum = 0; while (i <= 100) { sum += i; i ++; } alert("Сумма 1 + 2 +...+ 100 = " + sum);</pre>

Результат работы примера показан на рис. 8.1.



Рис. 8.1. Суммирование чисел от 1 до 100

Цикл работает так. Сначала проверяется условие. Если оно истинно, выполняется команда (тело цикла). И эти действия повторяются, т. е. снова проверяется условие, и если оно истинно, выполняется тело цикла, и т. д.

Цикл заканчивает работу, когда при очередной проверке условие оказывается ложным. Так как условие проверяется перед выполнением тела цикла, то команды, входящие в цикл, могут не выполниться ни разу.

Цикл *for*

В табл. 8.3 показан общий вид цикла *for* и пример его использования.

Таблица 8.3

Общий вид	Пример
for (начало; условие; приращение) команда;	var i; var sum = 0; for (i = 1; i <= 100; i ++) sum += i; alert("Сумма 1 + 2 + ... + 100 = " + sum);

Результат работы приведенного примера тот же, что показан на рис. 8.1.

Цикл работает так. Команда, помещаемая в начало, выполняется до циклического повторения (в примере это команда `i = 1;`), а сам цикл образуется следующими действиями:

- проверка условия (в примере `i <= 100`);
- выполнение тела цикла (в примере `sum += i`);
- выполнение команды, записанной в разделе приращение (в примере `i ++`).

Как и в команде **while**, тело цикла может не выполниться ни разу, если условие ложно с самого начала. Не выполнится при этом и команда из раздела *приращение*. А вот команда из раздела *начало* выполняется всегда, независимо от условия, и выполняется ровно один раз.

Замечание

В заголовке цикла **for** любая из трех конструкций *начало*, *условие*, *приращение* может быть опущена, при этом соответствующую точку с запятой опускать нельзя. Когда опущено *условие*, считается, что оно имеет значение **true**. Таким образом, цикл превращается в бесконечный: **for(;;)** команда. Этот цикл не остановится, если только не будет содержать внутри себя команду **break**.

Команды **break** и **continue**

Эти команды используют в теле цикла для изменения последовательного хода выполнения команд (рис. 8.2).

Команда **continue** заставляет браузер пропустить выполнение всех команд после нее и до конца тела цикла. Но цикл продолжается.

Команда **break** еще радикальнее — она заставляет браузер немедленно прекратить выполнение цикла.

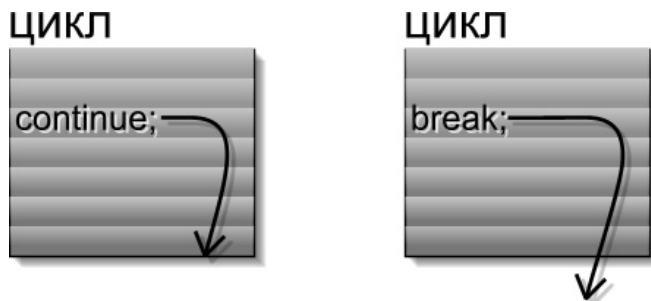


Рис. 8.2. Схема работы команд **continue** и **break**

Пример 1 (**continue**)

Найти сумму 5 четных чисел, случайным образом взятых из диапазона [1,20]

```
var len      = 5;  // Количество чисел.
```

```
var a        = 1;  // Левая граница интервала.
```

```
var b        = 20; // Правая граница интервала.
```

```
var sum      = 0;  // Сумматор.
```

```
var counter  = 0;  // Счетчик чисел.
```

```
var number;           // Случайное число.  
var str = "";         // Строка для вывода.  
while(counter < len)  
{  
    number = Math.floor(a + (b - a + 1) * Math.random());  
    if(number%2) continue;  
    sum += number;  
    str += number;  
    if(counter < len - 1) str += " + ";  
    else                str += " = ";  
    counter++;  
}  
str += sum;  
alert(str);
```

Результат работы примера показан на рис. 8.3.

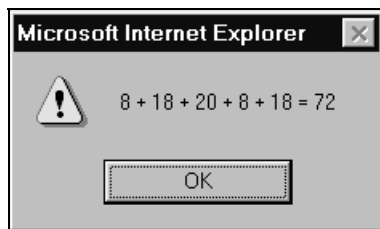


Рис. 8.3. Результат работы примера 1

Замечание

Стандартная функция `Math.random()` генерирует случайное число из отрезка $(0, 1)$ («случайное» число строго больше нуля и строго меньше единицы). Стандартная функция `Math.floor(num)` возвращает ближайшее целое число, меньшее или равное аргументу.

Пример 2 (break)

Целые числа случайным образом генерируются из диапазона $[1, 20]$. Требуется суммировать эти числа до тех пор, пока очередное случайное число не станет равным 10.

```
var a          = 1;  // Левая граница интервала.  
var b          = 20; // Правая граница интервала.  
var special = 10;    // Критическое значение случайного числа.  
var sum = 0;         // Сумматор.
```

```
var number;           // Случайное число.  
var str = "";         // Строка для вывода.  
for(;;) // Бесконечный цикл.  
{  
    number = Math.floor(a + (b - a + 1) * Math.random());  
    sum += number;  
    str += number;  
    if(number == special) break;  
    str += " + ";  
}  
str += " = " + sum;  
alert(str);
```

Результат работы примера может быть таким, как на рис. 8.4, или таким, как на рис. 8.5.

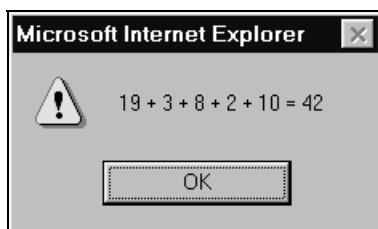


Рис. 8.4. Первый вариант работы примера 2

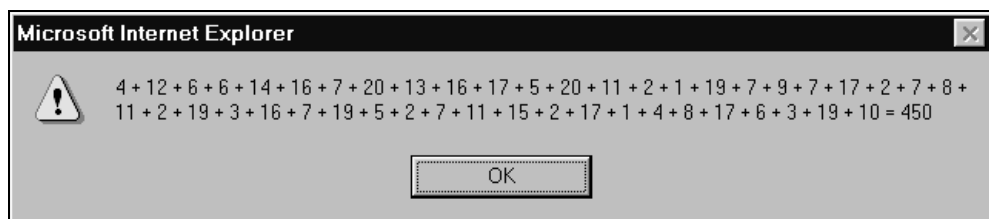


Рис. 8.5. Второй вариант работы примера 2

Зачетный класс

1. Какое сообщение будет выведено в окошко `alert` при выполнении кода (табл. 8.4)?

Таблица 8.4

Пример 1	Пример 2
<pre>var x = 5; var s = 0; while(x) {s += x; x --;} alert(s);</pre>	<pre>2. var x = 5; var s = 0; while(-- x) s += x; alert(s);</pre>
Пример 3	Пример 4
<pre>3. var x = 5; var s = 0; while(x--) s += x; alert(s);</pre>	<pre>4. var x = 5; var s = 0; while(s) s += x; alert(s);</pre>
Пример 5	Пример 6
<pre>var x = 5; var s = 0; while(!x) s += x; alert(s);</pre>	<pre>var x = 5; var s = 0; while(--x) s += x; s ++; alert(s);</pre>
Пример 7	Пример 8
<pre>var x = 5; var s = 0; while(-- x && s < 10) s += x; alert(s);</pre>	<pre>var x = 5; var s = 0; while(-- x s < 10) s += x; alert(s);</pre>
Пример 9	Пример 10
<pre>var x = 5; var s = 0; while(-- x !s) s += x; alert(s);</pre>	<pre>var x = 5; var s = 0; while(-- x && s) s += x; alert(s);</pre>
Пример 11	Пример 12
<pre>var s = 0; for(var i = 1; i < 10; i ++) s += i; alert(s);</pre>	<pre>var s = 0; for(var i = 1; i < 10; i += 2) s += i; alert(s);</pre>
Пример 13	Пример 14
<pre>var s = 0; for(var i = 10; i; i --) s += i; alert(s);</pre>	<pre>var s = 0; for(var i = 10; -- i;) s += i; alert(s);</pre>

Таблица 8.4 (окончание)

Пример 15	Пример 16
<pre>var s = 0; for(var i = 10; i --;) s += i; alert(s);</pre>	<pre>var s = 0; for(var i = 10;;) {if(-- i) break; s += i;} alert(s);</pre>
Пример 17	Пример 18
<pre>var s = 0; var i = 10; for(;;) {if(!(-- i)) break; s += i;} alert(s);</pre>	<pre>var s = 0; var i = 10; for(;;) {if(s > 6) break; s += -- i;} alert(s);</pre>
Пример 19	Пример 20
<pre>var s = 0; for(var i = 10;-- i; s += i); alert(s);</pre>	<pre>var s = 0; for(var i = 10;-- i;) {if(i > 5) continue; s += i;} alert(s);</pre>
Пример 21	Пример 22
<pre>var s = 0; var i = 10; while(-- i) {if(i > 5) continue; s += i;} alert(s);</pre>	<pre>var s = 0; var i = 10; while(-- i) {if(i < 5) break; s += i;} alert(s);</pre>

Задания

Во всех заданиях под числом понимается натуральное число.

1. Напишите скрипты с циклом **while** для решения следующих задач:

- найти $10! = 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10$;
- найти сумму цифр числа;
- найти произведение цифр числа;
- найти наибольшую цифру числа;
- найти наименьшую цифру числа;
- найти количество цифр «3» в числе;
- найти количество цифр «3» и «5» в числе;
- вывести цифры числа в обратном порядке;

- и) каждую цифру числа заменить дополнением до 9;
 - к) между цифрами числа записать знак «+»;
 - л) удвоить каждую цифру числа.
2. Напишите скрипты с циклом **for** для решения следующих задач:
- а) найти $10! = 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10$;
 - б) найти сумму 10 случайных чисел из диапазона [1, 100];
 - в) найти сумму 10 случайных четных чисел из диапазона [1, 100];
 - г) найти пятый член последовательности
$$a_1 = 2; a_n = a_{n-1}^2 + 1;$$
 - д) перевести двоичное число из 8 единиц в десятичную систему;
 - е) перевести шестнадцатеричное число из 8 единиц в десятичную систему;
 - ж) перевести троичное число из 8 двоек в десятичную систему;
 - з) построить 10-значное число, в котором цифра на *i*-м месте определяется как последняя цифра числа i^2 ;
 - и) построить 10-значное число, в котором цифра на *i*-м месте определяется как последняя цифра члена последовательности
$$a_i = 11 * i + 13;$$
 - к) проверить, выдает ли функция `math.random()` числа 0 и 1.
 - л) подсчитать число чисел, которые попадут в интервале (0,4, 0,6) за 1000 обращений к функции `math.random()`.
3. Напишите программу «Угадайка». Программа задумывает число из диапазона [1, 50], а пользователь старается отгадать его за наименьшее число проб. В каждой пробе пользователь вводит число, а «Угадайка» выдает одно из следующих сообщений: «число угадано», «задуманное число меньше», «задуманное число больше».
4. Создайте приложение для обучения малышей устному счету. Программа должна автоматически генерировать примеры типа «число1 + число2 =», проверять ответы, реагировать на ошибки, выставлять оценку в конце работы.

Дополнительный материал

Оптимизация: программирование циклов с конца

Подсчитать сумму элементов массива `set` можно так (нумерация элементов массива в JavaScript начинается с нуля; `set.length` — свойство массива — его длина):

```
var sum = 0;
for(var i = 0; i < set.length; i++) sum += set[i];
```

А можно и так:

```
var sum = 0;
for(var i = set.length; -- i >= 0; ) sum += set[i];
```

Перепишем этот вариант суммирования на «русском» языке:

```
var s = 0;
var i = set.length;
ПОВТОРЯТЬ
{
    уменьшить i на 1.
    проверить i: если i >= 0 то s += set[i];
                иначе цикл закончить
}
```

Второй способ предпочтительнее.

Сравнение с нулем обычно более эффективно, чем сравнение с другим числом, тем более сравнение с переменной, которая является собственностью объекта `set` (до нее еще нужно добраться).

Иными словами «-- i >= 0» работает быстрее, чем «i < set.length».

Что касается «i = set.length» во втором варианте, то это присваивание не «крутится» в цикле, а работает ровно один раз до входа в него.

Общее правило: в цикле со счетчиком значение счетчика должно по возможности уменьшаться.

Циклы — это как раз то место, где имеет смысл оптимизировать программу. Понятно почему: ведь выполнение тела цикла повторяется многократно, и малое улучшение умножается на большое «число оборотов». Если цикл выполняется 1000 раз, то такая оптимизация дает тысячекратный выигрыш.

Заметим, что такой вариант:

```
var sum = 0;
for(var i = set.length; (i --) >= 0; ) sum += set[i];
```

является ошибочным, ведь в нем уменьшение `i` на единицу происходит после проверки условия, а не до, как в правильном варианте.

Описание переменной цикла

Можно написать так:

```
for (var i = 100; i; i --) ...
```

— или так:

```
var i;
for (i = 100; i; i --) ...
```

Для браузера эти записи эквивалентны. Вспомним, что в конструкции цикла:

```
for(начало; условие; приращение) команда;
```

команда *начало* выполняется один раз перед первым оборотом цикла. Поэтому ясно, что переменная *i* не определяется 100 раз описателем **var**.

Рекомендации такие:

- ❑ счетчик цикла *i* можно описывать там же, где описываются другие переменные в начале функции или скрипта;
- ❑ счетчик обязательно нужно описывать в общем месте, если он используется в нескольких циклах;
- ❑ если в функции цикл единственный, то можно записывать **for (var i = ...; ...; ...)** — это красиво, и подчеркивает вспомогательный характер переменной *i*, недостойной более «солидных» описаний.

А что происходит, когда описание переменной помещается внутрь цикла **while**?

Если написать в C++ так:

```
while(...)  
{  
    int digit = ...  
    ...  
}
```

то переменная *digit* будет видна только внутри цикла и не доступна вне его. Дело здесь не в цикле, а в блоке {...}. Описания внутри блока локальны. Это означает, что переменная *digit* будет создаваться при входе в цикл и уничтожаться при выходе из него. Такая инкапсуляция переменных имеет смысл, как и инкапсуляции в программировании вообще. Программисты всегда стараются сделать область видимости переменной как можно более узкой. Это обычная практика оборонительного программирования.

Если переменная, подобная *digit*, используется как рабочая внутри цикла и не имеет смысла вне его, то разумнее объявить ее локальной внутри этого цикла, чем объявлять ее в начале функции с комментарием «рабочая переменная цикла такого-то».

Однако, к сожалению, здесь C++ расходится с JavaScript.

Для JavaScript запись:

```
while(...)  
{  
    var digit = ...  
    ...  
}
```

эквивалентна такой:

```
var digit;  
  
...  
while (...)  
{  
    digit = ...  
}
```

С этим печальным фактом приходится считаться, но в обоих случаях переменная `digit` создается один раз. Ведь описание — это не команда, а декларация. Браузер просматривает скрипт и выделяет под все переменные память, а только потом начинает выполнять команды.

Конструкция, в которой совмещены декларация и команда:

```
var digit = 10;
```

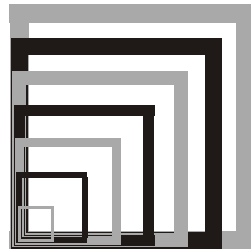
выполняется браузером так: сначала под переменную отводится память, а затем в нее помещается число 10.

Если браузер не может выполнить инициализацию переменной при ее создании:

```
function fun(t)  
{  
    var x = t + 1;  
    ...  
}
```

— то он отделяет создание переменной от ее вычисления. Переменная создается один раз при первом просмотре функции (подготовка к работе), затем работают команды функции и среди них команда `x = t + 1`.

Урок 9



Функции

В программировании часто бывает нужно повторять несколько раз одну и ту же группу команд. Если повторение происходит «на месте» — используют цикл (**while**, **for**). Если код нужно повторять в разных местах программы — его оформляют в виде функции.

Описание и вызов функции

В самом деле, если уж нужно вычислить длину (количество цифр) целого положительного числа, то логичнее проделать это один раз, а затем спокойно писать, как в математике $F(num)$, не задумываясь над «анатомией» функции F и не повторяя ее кода заново. Пусть, например, пользователь последовательно вводит три числа, а программа вычисляет общее количество цифр.

```
// Количество цифр в целом положительном числе num.
```

```
// Вход: num (целое положительное число).
```

```
// Выход: количество цифр в num.
```

```
function F(num)
{
    var len = num ? 0 : 1;
    while(num)
    {
        num = (num - num % 10) / 10;
        len ++;
    }
    return len;
}

var sum = 0;
var num1, num2, num3;
num1 = prompt("Введите первое число", "");
```

```
sum += F(num1);  
num2 = prompt("Введите второе число", "");  
sum += F(num2);  
num3 = prompt("Введите третье число", "");  
sum += F(num3);  
alert("Общее число введенных цифр: " + sum);
```

Браузер выполняет этот скрипт так. Запись:

```
function F(num)  
{  
    ...  
}
```

принимается во внимание, но команды в ней не выполняются. Браузер делает пометку: «было описание функции с именем `F`». Так происходит потому, что ключевое слово **function** и следующий за ним блок `{...}` являются не командой, а декларацией (описанием).

Команды, следующие за описанием функции, выполняются как обычно. При этом каждый раз, когда в коде попадает обращение к `F`, браузер возвращается к описанию функции и выполняет ее команды, подставляя вместо формального аргумента `num` фактические переменные `num1`, `num2`, `num3`.

На самом деле буквальной подстановки не происходит. Браузер перед вычислением функции `F` выполняет команду присваивания:

- `num = num1`; — при обращении `F(num1)`;
- `num = num2`; — при обращении `F(num2)`;
- `num = num3`; — при обращении `F(num3)`.

В общем случае описание функции на языке JavaScript выглядит так:

```
function ИмяФункции(список формальных аргументов через запятые)  
{  
    ...  
    тело функции  
    ...  
    return значение;  
}
```

Команда **return**, возвращающая значение функции, может быть не одна, а может и вовсе отсутствовать. В последнем случае функция не возвращает никакого значения, и ее вызов нельзя использовать в выражениях. В частности, вызов функции, не возвращающей значения, не может быть использован в команде присваивания.

Команда вызова функции имеет вид:

ИмяФункции(список фактических аргументов через запятые)

Фактическим аргументом функции может быть константа, переменная, выражение и, в частности, вызов другой функции.

Имена в программировании играют значительную роль. Они позволяют сделать код более понятным и, наоборот, способны запутать читателя, если выбраны как попало.

Имя `F` нельзя считать хорошим для описанной выше функции. В примере это имя использовано для «смягчения» перехода от математики к программированию. Ведь в математике функции обозначаются, как правило, одной буквой. Для программирования однобуквенное имя — редкое исключение. По одной букве трудно догадаться о назначении функции. В нашем случае имя `LenOfNumber` (длина числа) вместо `F` было бы гораздо лучше.

Длина имен функций и переменных в JavaScript не ограничивается, но имя должно быть одним словом (без пробелов). Имя строится из латинских букв, арабских цифр, знаков подчеркивания и некоторых других (не рекомендуемых) специальных знаков. Русские буквы использовать нельзя. Первым символом имени должна быть латинская буква или знак подчеркивания. Регистр букв имеет значение. Например, имена «`LenOfNumber`» и «`LenOfnumber`» считаются разными.

Замечание

Приведенный выше скрипт не проверяет правильность данных, которые вводит пользователь. В реальных программах проверка данных перед обработкой обязательна. Об оборонительном стиле программирования рассказано ниже в этом уроке.

Проектирование «сверху вниз»

Ранее говорилось, что функция используется тогда, когда нужно повторять один и тот же код в разных местах программы. Понятно, что функция удобна и тогда, когда предполагается использовать ее в разных программах.

В программировании популярны наборы библиотечных функций. Ввод и вывод данных, математические вычисления, создание интерфейсных окон, обработка щелчков мыши и перемещений — все это требуется практически в любой прикладной программе. Ясно, что есть смысл хорошо запрограммировать эти функции один раз. Так и делают. Некоторые наиболее популярные функции становятся даже элементами систем программирования (как функции `alert` и `prompt` они закодированы в самом браузере).

Однако самая важная область применения функций — технология построения программ, при которой сначала проектируется иерархическая схема

решения поставленной задачи, а каждому иерархическому узлу сопоставляется отдельная функция. Программирование начинается с корня иерархии и постепенно продвигается к ее листьям.

Такой способ проектирования называется «нисходящей технологией», или методом «сверху вниз». Он следует за особенностью человеческого мышления: мы не можем продумать проект сразу во всех тонкостях и деталях.

Задача

Написать программу, которая формирует HTML-код клетчатого поля, в котором чередуются темные и светлые клетки (рис. 9.1).

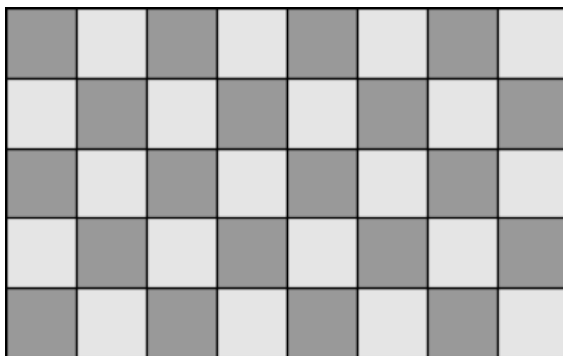


Рис. 9.1. Клетчатое поле

Решение методом «сверху вниз»

Построим изображение поля при помощи HTML-таблицы. Код таблицы будем формировать как значение строковой переменной.

Этап 1

Сначала опишем функцию, которая должна решать поставленную задачу целиком.

```
// Функция field(row, column, bcolor, wcolor)
// возвращает HTML-код таблицы, изображающей
// клетчатое поле.
// Аргументы:
//         row      - число строк.
//         column   - число столбцов.
//         bcolor   - цвет темной клетки.
//         wcolor   - цвет светлой клетки.
function field(row, column, bcolor, wcolor)
```

```
{
    // Формируем код начала таблицы.
    var str = "<TABLE border = 1 cellpadding = 10 cellspacing = 0 >";

    // Добавляем коды табличных строк.
    str += lines(row, column, bcolor, wcolor);

    // Добавляем код конца таблицы.
    str += "</TABLE>";

    return str;
}
```

Этап 2

Подумаем о том, как должен быть устроен формирователь табличных строк.

```
// Формирователь табличных строк.
// Функция lines(row, column, bcolor, wcolor)
// возвращает HTML-код всех табличных строк.
// Аргументы:
//         row      - число строк.
//         column   - число столбцов.
//         bcolor   - цвет темной клетки.
//         wcolor   - цвет светлой клетки.
function lines(row, column, bcolor, wcolor)
{
    var str      = ""; // Накопитель кода.
    var numpair = Math.floor(row / 2); // Число пар строк.

    // Строим коды строк таблицы парами.
    for(var i = 0; i < numpair; i++)
    {
        // Формируем код нечетной табличной строки.
        str += lineOdd(column, bcolor, wcolor);
        // Формируем код четной табличной строки.
        str += lineEven(column, bcolor, wcolor);
    }
}
```

```
// Строим дополнительно "нечетную" строку, если число строк нечетно.  
if(row%2) str += lineOdd(column, bcolor, wcolor);  
  
return str;  
}
```

Этап 3

Теперь можно заняться проектированием функций `lineOdd` и `lineEven`, которые строят соответственно нечетную и четную табличные строки. Функция `lineOdd` должна начинать строку с темной клетки, а функция `lineEven` — со светлой.

```
// Функция lineOdd(column, bcolor, wcolor)  
// возвращает HTML-код нечетной строки таблицы.  
// Аргументы:  
//         column - число ячеек в строке.  
//         bcolor  - цвет темной клетки.  
//         wcolor  - цвет светлой клетки.  
function lineOdd(column, bcolor, wcolor)  
{  
    // Формируем код начала табличной строки.  
    var str = "<TR>";  
    var numpair = Math.floor(column / 2); // Число пар клеток.  
  
    // Формируем коды клеток парами.  
    for(var i = 0; i < numpair; i ++)  
    {  
        // Строим код нечетной табличной клетки.  
        str += cell(bcolor);  
        // Строим код четной табличной клетки.  
        str += cell(wcolor);  
    }  
  
    // Строим дополнительную клетку, если число клеток нечетно.  
    if(column%2) str += cell(bcolor);  
  
    // Формируем код конца табличной строки.  
    str += "</TR>";  
  
    return str;  
}
```

```
// Функция lineEven(column, bcolor, wcolor)
// возвращает HTML-код четной строки таблицы.
// Аргументы:
//         column - число ячеек в строке.
//         bcolor - цвет темной клетки.
//         wcolor - цвет светлой клетки.
function lineEven(column, bcolor, wcolor)
{
    // Формируем код начала табличной строки.
    var str = "<TR>";
    var numpair = Math.floor(column / 2); // Число пар клеток.

    // Формируем коды клеток таблицы парами.
    for (var i = 0; i < numpair; i++)
    {
        // Строим код нечетной табличной клетки.
        str += cell(wcolor);
        // Строим код четной табличной клетки.
        str += cell(bcolor);
    }
    // Строим дополнительную клетку, если число клеток нечетно.
    if (column % 2) str += cell(wcolor);

    // Формируем код конца табличной строки.
    str += "</TR>";

    return str;
}
```

Этап 4

Остается только описать функцию `cell`, которая строит HTML-код отдельной табличной клетки.

```
// Функция cell(color)
// возвращает HTML-код табличной клетки.
// Аргументы:
//         color - цвет клетки
function cell(color)
{
    return "<TD bgcolor = "+color+">&nbsp;</TD>";
}
```

Замечание

Функции `lineOdd` и `lineEven`, формирующие код цепочки табличных клеток, легко заменяются одной универсальной функцией `line`. Читателю предлагается получить удовольствие от такого улучшения кода в рамках второй задачи блока заданий урока. Задачи 3, 4 и 5 также посвящены разным модификациям этого упражнения.

Построенная программа описывается иерархической схемой, которая соответствует этапам нисходящей разработки (рис. 9.2).

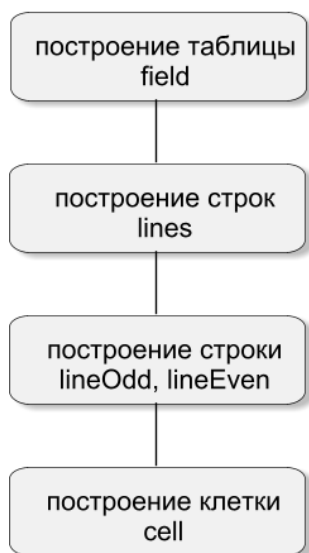


Рис. 9.2. Иерархическая схема программы

Решение построено, и его можно использовать. Например, вызов `var kodField = field(3, 2, "black", "white");` запишет в переменную `kodField` HTML-код такой таблицы (листинг 9.1).

Листинг 9.1

```
<TABLE border=1 cellspacing=0 cellpadding=10>
  <TR>
    <TD bgcolor=black>&nbsp;</TD><TD bgcolor=white>&nbsp;</TD>
  </TR>
  <TR>
    <TD bgcolor=white>&nbsp;</TD><TD bgcolor=black>&nbsp;</TD>
  </TR>
```

```

<TR>
  <TD bgcolor=black>&nbsp;  </TD><TD bgcolor=white>&nbsp;  </TD>
</TR>
</TABLE>

```

Построенный код будет записан в одну строку и без лишних пробелов, но браузер, конечно, на это «не обидится»!

Остается интересным вопрос: а как показать изображение, соответствующее этому коду на гипертекстовой странице?

Как будет показано в *Уроке 11*, это можно сделать при помощи функции `document.write` (метод объекта `document`).

Если описанный выше скрипт записан в файле `field.js`, то изображение двух таблиц на экране (рис. 9.3) можно получить при помощи такого HTML-документа (листинг 9.2).

Листинг 9.2

```

<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
  <SCRIPT language="JavaScript" src="field.js"></SCRIPT>
  <TITLE>Клетчатые поля</TITLE>
</HEAD>
<BODY bgcolor=white text=black>
<H2>Клетчатые поля</H2>
  <P>Первое поле (3x6)
  <SCRIPT language=JavaScript>
  <!--
    document.write(field(3,6,"#404040", "#F0F0F0"));
  //-->
  </SCRIPT>
  <P>Второе поле (4x13)
  <SCRIPT language=JavaScript>
  <!--
    document.write(field(4,13,"gray", "#F0F0F0"));
  //-->
  </SCRIPT>
</BODY>
</HTML>

```

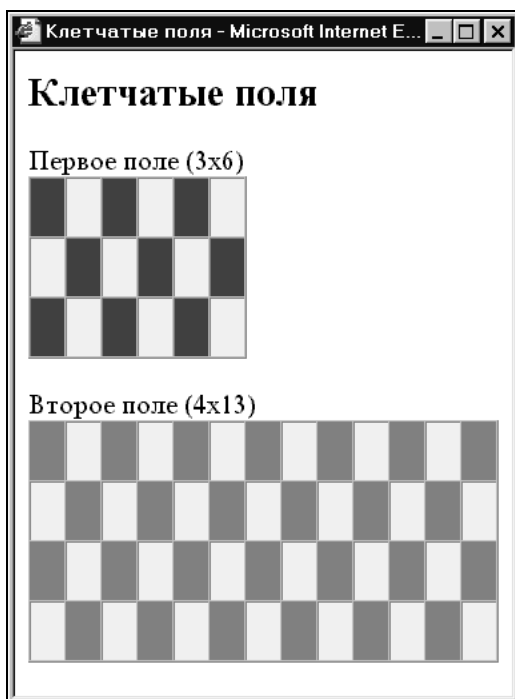


Рис. 9.3. Построено два клетчатых поля

Оборонительное программирование

Любая система, рассчитанная на реальную эксплуатацию, должна иметь защиту от «дурака». Этим обидным словом программисты обозначают потребителей своей продукции. На самом деле употребляется это слово в шутку. Пользователь имеет право на любую ошибку, и программа обязана сохранять работоспособность и правильно реагировать на любые действия потребителя, даже на такие, которые не предполагались постановкой решаемой задачи.

«Угадайка»

Петр Мячиков с удовольствием запрограммировал «Угадайку» (задание 3 Урока 8) и принес показать работу Ивану Сидорову.

— А заботится ли твоя программа о правильности данных, которые ввел пользователь? — спросил его Сидоров.

Загрузили программу в браузер и стали проверять (рис. 9.4).



Рис. 9.4. Запрос «Угадайки»

— Первое, что может сделать пользователь, — сказал Сидоров, — это нажать <Enter>, не введя никакого числа (рис. 9.5). Как реагирует программа?



Рис. 9.5. Пользователь нажал <Enter>

— «Угадайка» говорит, что теперь число находится в интервале [,50]. Странное заявление! Так кто здесь больше не прав? Пользователь или программист?

Можно ввести оборонительную проверку на ввод пустой строки:

```
if(answer == "") alert ("Ошибка при вводе числа");
```

Понятно, что пользователь может также:

- ☐ ввести вместо числа любой текст;
- ☐ ввести не целое число;
- ☐ ввести число не из рабочего диапазона.

Все это надо проверить и не пропустить неверные данные в алгоритм, который их обрабатывает.

Напишем специальную функцию для проверки правильности числа, полученного от пользователя.

```
// Проверка num на целое число из [a,b].  
// Результат: true, если проверка привела к успеху  
// и false в противном случае.  
function IsNum(num, a, b)
```

```
{  
    var ret = true; // Возвращаемое значение.  
    // Проверка на пустую строку.  
    if(num == "") ret = false;  
    else // Проверка на цифры.  
    {  
        var len = num.length;  
        for(var i = 0; i < len; i ++)  
        {  
            if(num.charAt(i) < "0" || num.charAt(i) > "9"){ret = false; break;}  
        }  
    }  
    // Проверка на интервал.  
    ret = ret && (a <= parseInt(num,10) && parseInt(num,10) <= b);  
    return ret;  
}
```

Замечание

Стандартная функция `parseInt(num,10)` преобразует явным образом значение переменной `num` в десятичное число. Второй аргумент этой функции указывает основание системы счисления: 2, 8, 10 или 16. Так, вызов `parseInt(num,16)` будет преобразовывать значение переменной `num` в 16-теричное число. Если функция `parseInt` сталкивается с неверным символом, она возвращает числовое значение, основанное на части строки до этого символа. Например, вызов `parseInt("48 попугаев",10)` вернет значение 48. Если неверный символ в строке самый первый — функция возвращает `NaN` — специальную константу (от Not a Number).

Как браузер разбирается в типах переменных?

Плохо разбирается, очень плохо! Не полагайтесь на автоматическое преобразование типов. Используйте функцию `parseInt` или аналогичную ей функцию `parseFloat` (преобразование в десятичную дробь).

Петр Мячиков написал первую «Угадайку» без использования явных преобразований типа. Сначала он дал этой программе на пробу число 49. «Угадайка» сообщила, что теперь задуманное число находится в интервале [1, 49]. Петр ввел число 222. К его удивлению «Угадайка» не выдала сообщения об ошибке, а вместо этого «сказала», что, задуманное число находится в интервале [1, 222]!

Почему так произошло? Почему число 222 было пропущено, и интервал из [1, 49] превратился в [1, 222]?

Дело в том, что проверка "222" < "49" привела к значению `true`! Такое странное поведение браузера объясняется очень просто: он сравнивал не числа, а строки. Строка «49» действительно «больше» строки «222» в смысле лексикографического порядка. Проверая строки, браузер сравнивает их по-символьно. Сначала сравнивает первые символы, потом — вторые и т. д. Если при очередном сравнении символ из первой строки располагается в кодовой таблице дальше символа из второй — проверка заканчивается и браузер считает первую строку «больше» второй. Так у него и получилось, что «49» > «222». Ведь символ «4» располагается в кодовой таблице «дальше» символа «2».

В *Приложении 4* приводится код небольшого испытателя (Испытатель 1). Загрузите его в браузер и проделайте опыты по сравнению:

- ☐ числа с числом;
- ☐ строки со строкой;
- ☐ числа со строкой.

Зачетный класс

Какое сообщение будет выведено в окошко `alert` при выполнении кода (табл. 9.1)?

Таблица 9.1

Пример 1	Пример 2
<pre>function f(x) { return alert(x + 1); }</pre>	<pre>function f(x) { return alert(x + 1); } f(2);</pre>
Пример 3	Пример 4
<pre>function f(x) { return x + 1; }</pre>	<pre>function f(x) { return x + 1; } alert(f(2));</pre>

Таблица 9.1 (продолжение)

Пример 5	Пример 6
<pre>function f(x) { return x ++; } alert(f(2));</pre>	<pre>function f(x) { return ++ x; } alert(f(2));</pre>
Пример 7	Пример 8
<pre>function f(x) { return 3 * x; x ++; } alert(f(2));</pre>	<pre>function f(x) { return 3 * x; } alert(f(2) * f(5));</pre>
Пример 9	Пример 10
<pre>function f(x) { return 3 * x; } alert(f(f(2)));</pre>	<pre>function f(x) { return !x; } alert(f(0));</pre>
Пример 11	Пример 12
<pre>function f(x) { return !x; } alert(f(2));</pre>	<pre>function f(x) { return !x; } alert(f(""));</pre>

Таблица 9.1 (окончание)

Пример 13	Пример 14
<pre>function f(x) { return !x; } alert(f("киса"));</pre>	<pre>function f(x) { return x + 1; } alert(f("киса"));</pre>
Пример 15	Пример 16
<pre>function f(x) { return x + 1; } alert(f("2"));</pre>	<pre>function f(x) { return x - 1; } alert(f("2"));</pre>
Пример 17	Пример 18
<pre>function f(x) { return x - 1; } alert(f(f("2")));</pre>	<pre>function f(x) { return x - 1; } alert(f("25" - "5"));</pre>
Пример 19	Пример 20
<pre>function f(x) { var ret = 1; if (x > 1) ret = x * f(x - 1); return ret; } alert(f(5));</pre>	<pre>function f(x) { var ret = 0; if (x) ret = x + f(x - 1); return ret; } alert(f(5));</pre>

Задания

1. Постройте испытательный стенд для проверки ответов на задания зачетного класса.
2. В программе построения кода клетчатого поля замените отдельные функции для построения нечетных и четных строк на одну универсальную функцию.
3. Напишите программу, которая строит HTML-код клетчатого поля, с указанием внутри каждой клетки ее координат в формате «строка, столбец» (рис. 9.6).

1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4
3,1	3,2	3,3	3,4
4,1	4,2	4,3	4,4

Рис. 9.6. Пример клетчатого поля с координатами клеток

4. Добавьте в решение предыдущей задачи стилевые указания, которые позволят выводить на экран клетки правильной квадратной формы (а не вытянутые по вертикали).
5. Клетчатое поле без HTML-таблиц. Напишите функцию:

```
// Функция field(row, column, bcolor, wcolor, size)
// возвращает HTML-код клетчатого поля.
// Аргументы:
//          row      - число строк.
//          column   - число столбцов.
//          bcolor   - цвет темной клетки.
//          wcolor   - цвет светлой клетки.
//          size     - размер клетки в пикселах.
function field(row, column, bcolor, wcolor, size)
{
    ...
}
```

Функция должна строить код клетчатого поля по следующей схеме (листинг 9.4).

Листинг 9.4

```

<DIV style = "
    относительно позиционирование;
    ширина: column*size;
    высота: row*size;
">

<SPAN абсолютное позиционирование для клетки 1,1>1,1</SPAN>
<SPAN абсолютное позиционирование для клетки 1,2>1,2</SPAN>
...
<SPAN абсолютное позиционирование для клетки 2,1>2,1</SPAN>
...
</DIV>

```

Границы поля задаются блоком `<DIV>...</DIV>`, а каждая клетка — блоком `<SPAN>...</SPAN>`, позиционированным в нужное место родительского блока.

6. Проверьте вашу программу обучения малышей устному счету (задание прошлого урока) на устойчивость к ошибкам пользователя. Включите в ее состав функцию проверки данных.
7. Напишите оборонительную «Угадайку».
8. Напишите диалоговую программу, которая предлагает пользователю:
 - задумать число;
 - умножить его на 2;
 - прибавить к результату 10;
 - разделить на 2;
 - сообщить результат.

Затем программа «отгадывает» задуманное число. На малышей это производит большое впечатление.

Дополнительный материал

Формальные и фактические аргументы функции

Часто начинающие программисты испытывают трудности в понимании различия между формальными и фактическими аргументами функции.

Можно считать описание функции:

```
function Sum(a, b)
```

```
{  
    return a + b;  
}
```

«документацией» на устройство этой функции. Аргументы *a* и *b* здесь формальные.

Когда записывается вызов функции $x = \text{Sum}(10, 20)$, то задаются фактические аргументы (в данном случае — константы 10 и 20).

Что происходит дальше? Браузер обращается к «документации» на функцию *Sum* и начинает выполнение. Локальным (для функции) переменным *a* и *b* (формальным аргументам) браузер присваивает значения 10 и 20. Затем вычисляет сумму и возвращает число 30.

Сколько *return* может иметь функция?

Команда **return** прерывает выполнение кода функции немедленно. Пусть, например, функция описана так:

```
function F(a)  
{  
    return 0;  
    if (a < 0) return -a;  
}
```

Вызов *F(x)* при любом значении *x* всегда вернет число 0.

Функция, описанная ниже, вычисляет абсолютное значение своего аргумента:

// Функция возвращает модуль числа *a*.

```
function abs(a)  
{  
    if (a < 0) return -a;  
    else      return a;  
}
```

Эту функцию лучше записать так:

// Функция возвращает модуль числа *a*.

```
function abs(a)  
{  
    var ret = a;  
    if (ret < 0) ret = -ret;  
    return ret;  
}
```

Надо стараться делать один выход из функции и записывать его самым последним (рис. 9.7, 9.8).

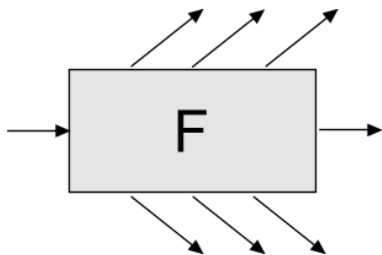


Рис. 9.7. Функция с несколькими выходами похожа на ежа



Рис. 9.8. Функцию с одним выходом легче понять и проверить

Функцию `abs` можно записать и так:

// Функция возвращает модуль числа `a`.

```
function abs(a)
{
    if (a < 0) a = -a;
    return a;
}
```

Переменная `a` внутри функции — это локальная переменная, ее создает браузер при входе в функцию. Поэтому такая запись допустима. Параметр, который передается в функцию, испорчен не будет. Поясним это на примере:

```
var x = -2;
var y = abs(x);
```

Выполнение по шагам:

1. В переменную `x` помещается число `-2`.
2. При входе в функцию `abs` браузер создает локальную (для функции) переменную `a` и копирует в эту переменную значение `x`, т. е. `a` получает значение `-2`.
3. После выполнения условной команды переменная `a` получает значение `2`.

4. Браузер записывает результат функции — число 2 и уничтожает переменную `a`.
5. Переменная `y` получает значение 2 (результат работы функции), а переменная `x` остается по-прежнему равной -2.

На самом деле самая красивая (и понятная!) запись для функции `abs` выглядит так:

```
// Функция возвращает модуль числа a.
```

```
function abs(a)
{
    return a > 0 ? a : -a;
}
```

Необходимости в самодельной функции `abs` нет. В JavaScript есть встроенный объект `Math` (см. Приложение 2). Среди методов этого объекта есть метод `abs`. Поэтому вычисление модуля `x` запишется как `Math.abs(x)`.

Функции и процедуры (для тех, кто работает на Паскале)

В Паскале под процедурой понимается именованный код, а под функцией — именованный код, возвращающий значение.

В Си и JavaScript процедур нет, есть только функции. Функция может возвращать значение, и тогда она похожа на функцию Паскаля. Функция может не возвращать значение, и тогда она похожа на процедуру Паскаля.

Функция возвращает значение, когда ее вычисление заканчивается командой **return**. Вызов такой функции можно использовать в выражениях:

```
function f(x)
{
    return ++ x;
}

var y = f(2) + 10;
```

В этом примере браузер вычислит `f(2)` и полученное значение 3 подставит вместо `f(2)` в выражение `f(2) + 10`. Таким образом, переменная `y` получит значение 13.

Функция может не возвращать значения, и тогда ее нельзя использовать в выражениях, она, подобно процедуре Паскаля, должна стоять в коде отдельной командой:

```
function g(x)
```

```
{  
    alert(x);  
}  
  
var str = "Чем мучиться и не жить, лучше жить и не мучиться!";  
g(str);
```

А вот если функция возвращает значение, то ее можно использовать и как процедуру Паскаля. Вычисленное значение при этом просто теряется:

```
function w(x,str)  
{  
    alert(str);  
    return ++ x;  
}
```

В этом коде:

```
var str = "Любил я женщин и проказы."  
var y = w(2,str);
```

на экран выводится сообщение, и переменная `y` получает значение 3.

А в этом коде:

```
var str = "Наш Федя с детства связан был с землею."  
w(2,str);
```

на экран выводится сообщение, а вычисленное значение 3 просто теряется.

Попробуйте предсказать, что будет выведено на экран в результате работы такого кода:

```
function f(x)  
{  
    return x ++;  
}  
  
var y = f(2) + 10;  
alert(y);
```

Функции с переменным числом аргументов

Это очень полезная возможность JavaScript. Аргументы, которые передаются в функцию, браузер хранит в виде глобального массива `имя_функции.arguments`. Первый аргумент, как водится в массивах JavaScript, имеет индекс 0. Число аргументов хранится в свойстве `length`. Обратиться к нему можно так:

```
имя_функции.arguments.length
```

Внутри функции работают более простые обращения:

`arguments[0]` — первый аргумент функции.

`arguments.length` — число аргументов, переданных в функцию.

Такой механизм дает приятную возможность передавать в функцию любое число аргументов. Но, конечно, функция должна быть правильно написана.

Рассмотрим пример. Построим функцию, которая вычисляет сумму переданных ей аргументов.

```
function sum()
{
    var s = 0;
    for(var i = arguments.length; -- i >= 0;)
        s += arguments[i];
    return s;
}

var x, y, z, t;
x = sum();           // x получит значение 0.
y = sum(1);          // y получит значение 1.
z = sum(1,2);         // z получит значение 3.
t = sum(1,2,3);       // t получит значение 6.
```

В приведенном выше примере функция может иметь любое число аргументов, в том числе и ноль.

Несколько первых аргументов можно сделать обязательными.

// Функция выполняет заданную операцию `op` над своими аргументами.

```
function calc(op, arg1, arg2)
{
    var str = arg1 + op + arg2;
    var len = arguments.length;
    for(var i = 3; i < len; i ++ )
        str += op + arguments[i];
    return eval(str);
}

var x, y, z, t;
x = calc("+", 1);           // Ошибка.
x = calc("+", 1, 2);        // x получит значение 3.
y = calc("*", 2,3,4);       // y получит значение 24.
z = calc("-", 10,5,3,1);    // z получит значение 1.
t = calc("/", 100,5,4);     // t получит значение 5.
```

Замечание

Стандартная функция `eval(str)` рассматривает свой аргумент `str` как выражение JavaScript, вычисляет его и возвращает полученное значение.

Оборонительное программирование

Лу Гринзоу в книге «Философия программирования для Windows 95/NT» написал: «По своей сути, оборонительное программирование — это минимизация количества опасных предположений, которые делает ваш код».

Данные

Если код работает с данными, которые поступают в него извне (например, их вводит пользователь, как в программе «Угадайка»), то он не должен предполагать, что полученные данные допустимы.

Оборонительный код должен проверить данные перед их «впрыском» в логику обработки. Проверить и принять адекватные меры: исправить ошибки, если это возможно, или отказать в обработке, выдав диагностику, оговоренную в документации.

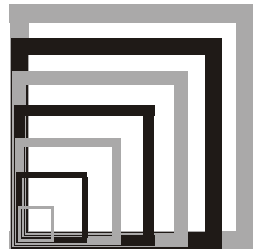
Функции

Когда код передает управление другому, внешнему коду (вызывает стандартную, библиотечную функцию или функцию, написанную вашим партнером), вы не можете быть уверены, что результат обработки будет именно таким, как это описано в сопроводительной документации или зафиксировано устно в момент обсуждения проекта на кухне за чашкой чая. В какой-то степени уберечь свой код от краха, следующего за вызовом чужих функций, можно путем тщательного тестирования того, что вы используете.

Проекты

Ваш собственный код должен быть хорошо спроектирован, документирован и отлажен. Если программные комментарии и внешние описания не соответствуют тому, что делает код на самом деле, то это — ловушка, в которую первым обычно попадает сам автор, решивший через какое-то время изменить алгоритм работы кода. Месяц — это предельный срок, в течение которого программист еще помнит, как работает его программа.

Урок 10



Объект = методы + свойства

Понятие объекта — ключевое в современном программировании. Идея объекта проста — рассматривать данные и функции, которые их обрабатывают, как одно целое.

Понятие объекта

Объект — это единая конструкция, содержащая в себе набор данных и функций, или, в терминологии JavaScript, набор свойств и методов.

Функция = метод.

Переменная = свойство.

Инкапсуляция

Этим термином обозначают сокрытие внутреннего устройства объекта, который рассматривается как «черный ящик». Известны свойства объекта, т. е. доступные извне переменные. Известны методы объекта, т. е. доступные извне функции. Но как устроены эти функции, по каким алгоритмам они работают, программисту не сообщается. Программист, пользователь объекта не знает и о том, есть ли у объекта дополнительные внутренние вспомогательные функции и переменные и как они связаны с доступными свойствами и методами.

Вот телевизор — это сложное устройство. Но зачем пользователю знать, что внутри этого объекта? Достаточно уметь нажимать кнопки на пульте, чтобы заставить «ящик» показывать нужную программу с подходящей громкостью.

Инкапсуляция удобна, правда? Она позволяет слабому человеческому разуму справиться с огромными информационными потоками, закрывая глаза на несущественные детали.

Кто-то делает микросхемы, кто-то собирает из них телевизор, кто-то смотрит программу «Время». И все довольны. Каждый хорошо знает свое дело. Представьте, что для просмотра телевизора необходимо было бы знать в деталях устройство этих паукообразных штук, которые называют микросхемами, а также технологию изготовления люминесцентных трубок и все такое прочее.

Объект и экземпляр объекта

Вообще говоря, конкретный телевизор — это в терминологии JavaScript не объект, а экземпляр объекта. Объектом является комплект документации, по которой на заводе изготавливается продукция. Все телевизоры, которые сходят с конвейера, имеют одни и те же *свойства* изображения и одни и те же *методы* управления этими свойствами.

То же и в программировании. Объект — это шаблон, документация. Экземпляр объекта — это рабочая копия.

Интерфейс объекта и внутреннее устройство объекта

Программист Сидоров придумал объект `Rectangle`. Он сообщил коллегам следующую информацию (табл. 10.1).

Таблица 10.1

Объект <code>Rectangle</code>	
Объект служит для работы с прямоугольниками. Для создания экземпляра объекта нужно написать:	
<pre>var x = new Rectangle(a,b); // Переменная x теперь является экземпляром // объекта Rectangle. Значения a и b задают // ширину и высоту прямоугольника. // Конструкция new используется для // создания экземпляра.</pre>	
Когда экземпляр объекта создан, можно использовать следующие методы и свойства:	
Свойства	Описание
<code>width</code>	Ширина прямоугольника.
<code>height</code>	Высота прямоугольника.
Методы	Описание
<code>square()</code>	Площадь прямоугольника.
<code>perimeter()</code>	Периметр прямоугольника.
<code>radius()</code>	Радиус описанной окружности.
Пример использования:	
<pre>var p = x.perimeter();</pre>	<pre>// Вычисление периметра.</pre>
<pre>var r = x.radius();</pre>	<pre>// Вычисление радиуса описанной окружности.</pre>

Таблица 10.1 (окончание)

Объект Rectangle

```
// Вычисление максимального размера.  
var m = x.width;  
if(x.height > m) m = x.height;
```

Что сообщил Сидоров коллегам? Он сообщил *интерфейс* своего объекта, т. е. информацию, необходимую для взаимодействия с объектом.

Сидоров не показал друзьям программные коды функций `square`, `perimeter`, `radius`. Не сказал он и о вспомогательной функции `sqrt`, которая была написана для вычисления квадратного корня (эту функцию использует у Сидорова метод `radius`). Не сказал Сидоров и про переменную `temp`, которая потребовалась объекту для промежуточных вычислений.

Другими словами, Сидоров не показал внутреннее устройство своего объекта. Да и зачем коллегам знать это? Сидоров — программист хороший, все свои коды тщательно проверяет, значит, можно пользоваться только интерфейсом, не заглядывая к Сидорову на «кухню».

Интерфейс объекта — это доступные для использования переменные и функции объекта.

Внутреннее устройство объекта — это описание переменных и функций внутри объекта на языке программирования («кухня» Сидорова).

Аналогия с телевизором: интерфейс — это пульт управления. Внутреннее устройство — это электронные «потроха» в ящике.

Доступ к свойствам и методам

Обратите внимание на то, как в JavaScript записывается обращение к свойствам и методам экземпляра `x` объекта `Rectangle`:

```
x.height;  
x.perimeter();
```

В общем виде формат обращения можно записать так:

имя_экземпляра.свойство_или_метод_объекта

Точка используется как признак иерархической принадлежности: она отделяет потомка от его предка.

Встроенные объекты и объекты пользователя

В языке JavaScript много встроенных объектов. Их не надо программировать, они заложены в сам язык. Это означает, что коды этих объектов вхо-

дят в состав программных кодов браузера. Программист должен знать интерфейсы этих объектов, уметь создавать экземпляры, и тогда он сможет использовать встроенные объекты по своему усмотрению.

JavaScript позволяет программировать новые объекты и менять алгоритм работы встроенных. Это очень удобно и привлекательно для программиста. Далее в книге будет много примеров на эту тему, но сначала рассмотрим работу нескольких встроенных объектов (более подробный список см. в *Приложении 3*).

Объект *Date*

Начнем практическое рассмотрение встроенных объектов языка JavaScript с очень полезного объекта *Date*. Этот объект используется для работы с датой и временем.

Для создания экземпляра объекта (не только объекта *Date*, но и любого другого) в языке JavaScript используется ключевое слово **new**:

```
var now = new Date();
```

Теперь переменная *now* является экземпляром объекта *Date* и содержит текущую дату и время.

В общем случае экземпляр образуется следующим образом:

```
var переменная = new Date(параметры);
```

Возможно указание следующих параметров (табл. 10.2).

Таблица 10.2

Параметры	Пример	Описание
Пусто	<code>var now = new Date();</code>	Экземпляр <i>now</i> содержит текущую дату и время
Целочисленные значения для года, месяца и дня	<code>var birthday = new Date(1954, 1, 8);</code>	Экземпляр <i>birthday</i> содержит дату, равную 8 февраля 1954 года (нумерация месяцев начинается с нуля)
Целочисленные значения для года, месяца, дня, часа, минуты, секунды	<code>var bell = new Date(2003, 0, 14, 12, 20, 0);</code>	Экземпляр <i>bell</i> содержит дату и время, равные 12 часам 20 минутам 14 января 2003 года

После того как экземпляр объекта `Date` создан, можно узнать записанные в нем данные, а можно их и менять. Для этого существуют многочисленные методы, список которых приводится в *Приложении 3*.

Имя метода объекта (вспомним, что метод в терминологии JavaScript — это функция) отделяется от имени экземпляра точкой.

Можно написать:

```
var year = bell.getYear();
```

Переменная `year` получит значение 2003.

Рассмотрим несколько простых скриптов с объектом `Date`.

Текущая дата и время

```
var now = new Date();  
alert("Сегодня: " + now.getDate() + "/" + (now.getMonth() + 1) + "/" +  
      now.getYear() + "\nСейчас: " + now.getHours() +  
      ":" + now.getMinutes());
```

В момент написания этих строк выполнение скрипта привело к появлению сообщения, показанного на рис. 10.1.

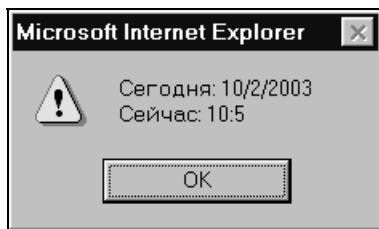


Рис. 10.1. Текущая дата и время

Число дней с начала года

```
var now = new Date(); // Текущая дата и время.  
var begin = new Date(now.getYear(), 0, 1); // Начало текущего года.  
// Число миллисекунд от начала года:  
var num = now.getTime() - begin.getTime();  
var msPerDay = 24 * 60 * 60 * 1000; // Число миллисекунд в сутках.  
num /= msPerDay; // Число дней с начала года.  
// Покажем результат:  
alert("Прошло " + Math.floor(num) + " дней с начала года");
```

На момент написания этих строк выполнение скрипта привело к появлению сообщения, показанного на рис. 10.2.


```
// Название дней недели:
var dayNames = new Array("воскресенье", "понедельник", "вторник",
                           "среда", "четверг", "пятница", "суббота");
var today = new Date(); // Текущая дата и время.
// Начало формирования строки вывода:
var str = "Сегодня: ";
// Добавим день месяца:
str += today.getDate() + " ";
// Добавим название месяца:
str += monthNames[today.getMonth()] + " ";
// Добавим год:
str += today.getFullYear() + " года, ";
// Добавим день недели:
str += dayNames[today.getDay()] + "\n";
// Добавим время:
str += "Сейчас: " + today.getHours() + ":" +
        today.getMinutes() + ":" +
        today.getSeconds();

// Покажем результат:
alert(str);
```

На момент написания этих строк выполнение скрипта привело к появлению сообщения, показанного на рис. 10.3.

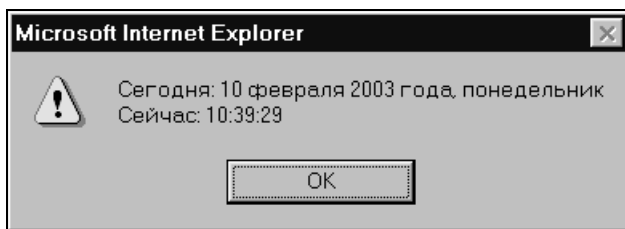


Рис. 10.3. Текущая дата и время

Длина массива (число элементов в нем) может меняться во время работы программы:

```
var f = new Array(); // Сейчас массив пустой, элементов в нем нет.
f[0] = 1;             // В массиве один элемент.
f[1] = 1;             // В массиве два элемента.
f[2] = f[0] + f[1];    // В массиве три элемента.
f[5] = 8;             // В массиве шесть элементов f[0], ..., f[5].
```

Пример: максимальный элемент массива (рис. 10.4).

```
// Создадим массив из num случайных чисел, каждое из которых входит
// в диапазон [a,b].
var i;
var num = 10; // Количество случайных чисел.
var a = 1;    // Левая граница интервала.
var b = 100;  // Правая граница интервала.
var set = new Array(); // Создан пустой массив.
// Заполнение массива случайными числами:
for(i = 0; i < num; i ++){
    set[i] = Math.floor(a + (b - a + 1) * Math.random());
}
// Найдем максимальный элемент.
var max = a;
for (i = 0; i < num; i ++){
    if(set[i] > max) max = set[i];
}
// Покажем массив и найденный максимум:
alert("Массив: " + set + "\nМаксимум: " + max);
```

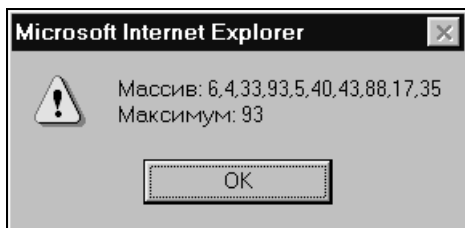


Рис. 10.4. Максимальный элемент массива

Замечание

Два цикла рассмотренного примера можно объединить в один:

```
var max = a;
for (i = 0; i < num; i ++){
    {
        set[i] = Math.floor(a + (b - a + 1) * Math.random());
        if(set[i] > max) max = set[i];
    }
}
```

До сих пор мы пользовались методами объектов и еще ни разу — их свойствами. Вспомним, что свойства объекта — это в терминологии JavaScript те интерфейсные переменные, которые доступны пользователю объекта. На самом деле, конечно, речь не об объекте, а о конкретном экземпляре объек-

та. Для обращения к свойствам указывается имя экземпляра объекта и через точку — имя свойства, например (рис. 10.5):

```
var set = new Array("Горбунков", "Семен", "Семенович");  
alert(set.length);
```



Рис. 10.5. Длина массива

Свойство `length` содержит число элементов массива — его длину.

Язык JavaScript допускает много «вольностей» при обращении с объектами. Так, например, можно задать экземпляр массива без ключевого слова `new` и имени объекта `Array`:

```
var set = [1, 4, 9, 16, 25, 36];
```

Конечно, браузер, просматривая такую запись, все равно создает экземпляр объекта `Array` точно такой же, как и при записи:

```
var set = new Array(1, 4, 9, 16, 25, 36);
```

В предыдущих уроках мы часто строили экземпляры объекта `String`, даже не подозревая об этом. Такая запись:

```
var title = "Старик, упавший с каланчи";
```

совершенно эквивалентна записи:

```
var title = new String("Старик, упавший с каланчи");
```

Для работы со встроенными объектами JavaScript удобно иметь под рукой справочный материал, который приводится в *Приложении 3*.

Зачетный класс

Какое значение будет иметь переменная `s` после выполнения скрипта?

```
a) var a = new Array(1,2,3,4,5,6,7,8,9,10);  
   var s = 0;  
   for(var i = 0; i < a.length; i ++)  
       s += a[i];
```

```
б) var a = new Array(1,2,3,4,5,6,7);
    var s = 0;
    for(var i = a.length; -- i >= 0;)
        s += a[i];
    s /= a.length;

Б) var a = new Array(6,2,3,4,7,1,5);
    var s = a[0];
    for(var i = 0; i < a.length; i ++)
        if(a[i] > s) s = a[i];

Г) var a = new Array(6,2,3,4,7,1,5);
    var d = a[0];
    var s = 0;
    for(var i = 0; i < a.length; i ++)
        if(a[i] > d) { d = a[i]; s = i; }

Д) var a = new Array("2","48","000","99");
    a.sort(Compare);
    var s = a[0];
    function Compare(a,b)
    {
        return b.length-a.length;
    }

е) var a = new Array("34","81","234","6789");
    a.sort(Compare);
    var s = a[0];
    function Compare(a,b)
    {
        if(a < b)         return 1;
        else if(a > b) return -1;
        else               return 0;
    }

Ж) var str = new String("3481234627892");
    var s = 0;
    for(var i = 0; i < str.length; i ++)
        if(str.charAt(i) == "2") s ++;

з) var str = new String("3481234627892");
    var s = 0;
```

```
for(var i = str.indexOf("2"); i >= 0;
    i = str.indexOf("2",i + 1)) s ++;
и) var str = new String("Шалтай-Болтай");
var sub = str.substring(3,5);
var s = 0;
for(var i = str.indexOf(sub); i >= 0;
    i = str.indexOf(sub,i + 1)) s ++;
```

Дополнительный материал

Объектно-ориентированное программирование (ООП) — современный способ создания программных кодов, который пришел на смену структурному программированию. Вернее будет сказать, что ООП не заменило структурное программирование, а довело его до логического совершенства.

Структурное программирование

Суть структурного программирования — в построении иерархического дерева решения и максимальной изоляции программного кода, выполняющего единую задачу. Структурные программисты достигают своей цели, собирая процедуры и данные, относящиеся к одной логической единице кода, в отдельный файл, конструируя собственные типы данных при помощи таких средств языка, как структура (**struct** в Си).

Пример 1

Программа Сидорова работает с прямоугольниками. Логично сконструировать новый тип данных `Rectangle`. Вот как это описывается в C++:

```
// Документация на новый тип данных Rectangle.
```

```
struct Rectangle
{
    int a;    // Ширина прямоугольника.
    int b;    // Высота прямоугольника.
}
```

Всякий раз, когда нужен новый прямоугольник, он определяется так:

```
Rectangle rec;
```

Эта запись очень привычна. Все выглядит так же, как и описание:

```
int a;
```

Можно присвоить переменной `rec` значение:

```
rec.a = 10; // Задали высоту прямоугольника rec.
```

```
rec.b = 20; // Задали ширину прямоугольника rec.
```

Можно использовать значение переменной `rec` в выражениях:

```
perimeter = 2 * (rec.a + rec.b); // Периметр прямоугольника.
```

Можно использовать новый тип данных как аргумент функции:

```
int function Perimeter(Rectangle x)
{
    return 2 * (x.a + x.b);
}
```

Объектно-ориентированное программирование

В ООП-языках можно конструировать не только новые данные, но объединять данные с процедурами (функциями), которые их обрабатывают. В JavaScript это называется объектом (в C++ классом).

Пример 2

```
//-- Документация на объект Rectangle. --
// Конструктор объекта.
function Rectangle(a, b)
{
    // Описание свойств объекта (его данных).
    this.a = a; // Ширина прямоугольника.
    this.b = b; // Высота прямоугольника.
}
//-- Конец документации на объект Rectangle.
```

В примере 2 описан объект `Rectangle` почти так, как это сделано в примере 1 (`struct Rectangle`). В этом объекте есть только данные, но нет функций. Теперь всякий раз, когда нужен новый прямоугольник, он определяется в JavaScript так:

```
var rec = new Rectangle(10,20); // Создан экземпляр объекта rec.
```

Можно использовать значение переменной `rec` в выражениях:

```
var perimeter = 2 * (rec.a + rec.b); // Периметр прямоугольника.
```

Можно использовать экземпляр объекта как аргумент функции:

```
function Perimeter(x)
{
    return 2 * (x.a + x.b);
}

var p = Perimeter(rec);
```

Добавим в построенный объект функции (методы), которые обрабатывают данные объекта.

Пример 3

```
//-- Документация объекта Rectangle. --  
// Конструктор объекта.  
function Rectangle(a, b)  
{  
    // Описание свойств объекта (его данных).  
    this.a = a; // Ширина прямоугольника.  
    this.b = b; // Высота прямоугольника.  
    // Описание методов объекта (его функций).  
    this.perimeter = _perimeter; // Ссылка на функцию _perimeter,  
                                // вычисляющую периметр.  
}  
// Описание метода perimeter.  
function _perimeter()  
{  
    return 2 * (this.a + this.b);  
}  
/---Конец документации объекта Rectangle
```

Теперь объект `Rectangle` содержит не только данные, но и функции. Можно писать такие коды:

```
// Создадим первый прямоугольник:  
var p1 = new Rectangle(10,20);  
// Создадим второй прямоугольник:  
var p2 = new Rectangle(35,70);  
// Найдём сумму периметров:  
var sum = p1.perimeter() + p2.perimeter();
```

Обратите внимание на строку, определяющую метод `perimeter`:

```
this.perimeter = _perimeter;
```

В ней определен метод с именем `perimeter` и указано, что реализует этот метод функция с именем `_perimeter`. Имена метода и функции, которая его реализует, могут быть любыми. Но для того, чтобы было меньше путаницы, рекомендуется имя функции образовывать от имени метода приставкой «`_`».

Из жизни программиста Сидорова

Понятие объекта — несложное, но к нему надо привыкнуть. Давайте еще раз посмотрим на пример с объектом `Rectangle`, описанный в основной части урока, и заглянем на «кухню» Ивана.

На «кухне» Сидорова

Объект в JavaScript так же, как и обычная функция, описывается ключевым словом **function**:

```
//-- Документация объекта Rectangle. --  
// Конструктор объекта.  
function Rectangle(a,b)  
{  
    // Свойства.  
    this.width = a;    // Ширина прямоугольника.  
    this.height = b;   // Высота прямоугольника.  
    // Методы.  
    this.square = _square;        // Площадь прямоугольника.  
    this.perimeter = _perimeter;  // Периметр прямоугольника.  
    this.radius = _radius;        // Радиус описанной окружности.  
}  
// Реализация методов объекта.  
function _square()  
{  
    return this.width * this.height;  
}  
function _perimeter()  
{  
    return 2 * (this.width + this.height);  
}  
function _radius()  
{  
    var temp = this.width * this.width + this.height * this.height;  
    return Math.sqrt(temp) / 2;  
}  
/-- Конец документации объекта Rectangle. --
```

Описание:

```
function Rectangle(a,b)
```

```
{  
    ...  
}
```

называется *конструктором* объекта. Сидоров использовал внутри этого описания переменные с приставкой **this**:

```
this.width = a; // Ширина прямоугольника.  
this.height = b; // Высота прямоугольника.
```

Ключевое слово **this** является указателем на экземпляр объекта, который будет создан при помощи конструктора. То есть оно задает смысл переменным и функциям как свойствам и методам будущих экземпляров объекта.

Условно можно сказать, что ключевое слово **this** «превращает» в описании конструктора переменную в свойство, а ссылку на функцию — в метод объекта.

Пользуйтесь простым правилом, позволяющим отличить обычную функцию от конструктора объекта: «если внутри **function** {...} вместо описателя **var** использовано **this** — это конструктор объекта».

```
var r1 = new Rectangle(3,4); // Первый экземпляр прямоугольника.  
var r2 = new Rectangle(10,20); // Второй экземпляр прямоугольника.
```

По эти двум командам и «документации» на объект браузер создаст два экземпляра прямоугольника **r1** и **r2**. У каждого экземпляра будут свои переменные **width** и **height**. Обратиться к ним можно так:

```
var x = r1.width; // x получает значение 3.  
var y = r2.height; // y получает значение 20.
```

Вот полный текст примера, который вы можете проверить в своем браузере (листинг 10.1):

Листинг 10.1

```
<HTML>  
  <HEAD>  
    <META http-equiv="Content-Type"  
      content="text/html; charset=windows-1251">  
    <TITLE>Проверка объекта</TITLE>  
  </HEAD>  
  <BODY bgcolor=white text=black link=blue alink=red vlink=purple>  
    <H1>Проверка объекта</H1>  
    <HR>
```

```
<SCRIPT language=JavaScript>
<!--
/-- Документация объекта Rectangle. --
/-- Конструктор объекта.
function Rectangle(a,b)
{
    // Свойства.
    this.width  = a; // Ширина прямоугольника.
    this.height = b; // Высота прямоугольника.
    // Методы.
    this.square   = _square;    // Площадь прямоугольника.
    this.perimeter = _perimeter; // Периметр прямоугольника.
    this.radius   = _radius;    // Радиус описанной окружности.
}
// Реализация методов объекта.
function _square()
{
    return this.width * this.height;
}
function _perimeter()
{
    return 2*(this.width + this.height);
}
function _radius()
{
    var temp = this.width*this.width + this.height*this.height;
    return Math.sqrt(temp)/2;
}
/-- Конец документации объекта Rectangle. --

// Примеры работы с построенным объектом.
var r1 = new Rectangle(3,4);
alert("Первый прямоугольник:\nШирина="+r1.width+"\nВысота="+
      r1.height);
var r2 = new Rectangle(10,20);
alert("Второй прямоугольник:\nШирина="+r2.width+"\nВысота="+
      r2.height);
alert("Площадь первого прямоугольника="+r1.square());
```

```
    alert("Периметр второго прямоугольника="+r2.perimeter());  
    alert("Радиус для первого прямоугольника="+r1.radius());  
    //-->  
</SCRIPT>  
</BODY>  
</HTML>
```

Попробуйте этот скрипт в работе.

Наследование

Захотел Сидоров создать новый объект `Kvadrat`. Квадрат, то есть. Но по лени своей природной, решил он не утруждать себя новыми описаниями, а использовать уже построенный объект `Rectangle`. Ведь квадрат — это тоже прямоугольник, только квадратный! Пусть все свойства и методы прямоугольника перейдут по наследству к квадрату, решил Сидоров. А так как серая жизнь надоела Ивану, он решил добавить для квадрата новое свойство «цвет»:

```
// Описание конструктора (a - сторона квадрата, c - его цвет).
```

```
function Kvadrat(a,c)  
{  
    this.parent = Rectangle; // Определили родительский объект.  
    this.parent(a,a);        // Вызвали конструктор родителя.  
    this.color = c;          // Определили свойство "цвет"  
}
```

Теперь Сидоров смело писал:

```
var kv = new Kvadrat(100, "красный"); // Создал квадрат!  
var s = kv.square();                  // Нашел его площадь.  
var p = kv.perimeter();               // Нашел периметр квадрата.  
alert("Этот квадрат " + kv.color + "\nЕго площадь=" + s +  
      "\nА периметр =" + p + "\nСторона квадрата=" + kv.width);
```

Описанный механизм очень важен для объектно-ориентированного программирования. Он носит название *наследование*.

Изменение «документации» на объект

Сидел однажды Иван на кухне и чесал свой бугристый затылок. Не нравится ему встроенный объект `Date`. Хочется Ивану, чтобы этот объект выдавал день недели по-русски. Учил Иван, конечно, в школе английский, но по-русски читать сообщения все же приятней!

И написал Иван с горя такой код:

```
var d = new Date();
d.mygetDay = _mygetDay;
// Вернуть день недели по-русски.
function _mygetDay()
{
    var dayNames = new Array("воскресенье", "понедельник", "вторник",
                              "среда", "четверг", "пятница", "суббота");
    return dayNames[this.getDay()];
}
alert(d.mygetDay());
```

— Мама, родная! — подумал Иван. — Что ж это такое я натворил!

Но раз уж написано, отступать некуда. Иван запустил скрипт и... получил день недели по-русски!

Дело в том, что команда:

```
d.mygetDay = _mygetDay;
```

добавляет к методам экземпляра `d` новый метод с именем `mygetDay` и сообщает браузеру, что этот метод описан функцией `_mygetDay`.

Теперь экземпляр `d` обладает всеми свойствами и методами объекта `Date` и еще одним методом `mygetDay`, который придумал Иван.

Конечно, другой экземпляр, например,

```
var d1 = new Date();
```

ничего про особенности экземпляра `d` не знает, и если написать:

```
var x = d1.mygetDay();
```

то браузер выдаст сообщение об ошибке.

Давайте вспомним аналогии урока. Объект — это документация на изготовление телевизора. Экземпляр объекта — это отдельный телевизор, Иван «приделал» к своему конкретному телевизору `d` дополнительную «ручку». При этом, конечно, у соседа телевизор `d1` остался в прежнем состоянии.

А можно ли изменить не экземпляр объекта, а документацию на заводе? Так, чтобы с новой «ручкой» стали сходиться с конвейера все телевизоры? Нет проблем! В языке JavaScript есть ключевое слово `prototype`. Оно как раз и указывает на «заводскую документацию» объекта. Пишем так:

```
Date.prototype.mygetDay = _mygetDay;
```

```
// Вернуть день недели по-русски.
```

```
function _mygetDay()
```

```
{  
    var dayNames = new Array("воскресенье", "понедельник", "вторник",  
                              "среда", "четверг", "пятница", "суббота");  
    return dayNames[this.getDay()];  
}  
  
var d = new Date();  
var d1 = new Date();  
alert(d.mygetDay());  
alert(d1.mygetDay());
```

Первая строка в этом коде вводит в «документацию» объекта `Date` новый метод с именем `mygetDay`. Теперь все экземпляры объекта `Date` будут иметь этот новый метод, придуманный Иваном на кухне за кружкой крепкого чая. Правда, это будет работать только внутри того HTML-файла, в котором расположен скрипт Ивана, монтирующий новую «ручку».

Статическое и динамическое наследование

Иван зачем-то создал такой объект:

```
//-- Документация на объект Num. --  
function Num(a)  
{  
    this.number = a;  
    this.mul2 = _mul2;  
}  
function _mul2()  
{  
    return this.number * 2;  
}  
//-- Конец документации на объект Num. --
```

Объект может хранить число и умножать его на 2.

Затем Иван написал:

```
var x = new Num(10);  
var y = x.mul2();  
alert(y);
```

Как он и ожидал, на экране появилось число 20.

Иван выпил кружку чая и создал для объекта `Num` наследника:

```
//-- Документация на объект Numa (наследован от Num) --  
function Numa(a)
```

```
{
    this.parent = Num; // Родителем объявлен Num.
    this.parent(a);    // Вызван конструктор родителя.
    this.put = _put;   // Объявлен новый метод у потомка.
}
function _put()
{
    alert("Исходное число=" + this.number);
}
//-- Конец документации на объект Numa. --
```

Для проверки сына Иван написал:

```
var z = new Numa(100);
z.put();
```

На экране появилась надпись: «Исходное число=100».

Иван выпил еще чаю.

— Добавлю-ка я к объекту Num новый метод, — осенило вдруг Ивана.

Он написал:

```
Num.prototype.mul3 = _mul3;
function _mul3()
{
    return this.number * 3;
}
```

Проверил:

```
var t = new Num(10);
alert(t.mul3());
```

На экране появилось число 30.

— Интересно, — подумал Иван, — а потомок понимает новый метод своего родителя?

Он написал:

```
var k = new Numa(10);
alert(k.mul3());
```

Браузер выдал сообщение об ошибке.

Такое наследование называется *статическим*. При создании наследника в его документацию копируются все свойства и методы родителя, после чего родственная связь теряется. Если родитель приобретает новый метод, потомок об этом ничего не знает.

Можно создать наследника и не разрывать «пуповины». Это достигается при помощи *динамического* наследования. Если объект создан способом динамического наследования, то копируется не «документация» родителя, а *ссылка* на нее. И если у родителя происходят изменения, это также будет касаться всех «динамических» наследников.

Синтаксис создания объекта с динамическим наследованием выглядит так:

```
//-- Документация на объект Numa (динамически наследован от Num). --  
function Numa(a)  
{  
    this.parent = Num; // Родителем объявлен Num.  
    this.parent(a);    // Вызван конструктор родителя.  
    this.put = _put;    // Объявлен новый метод у потомка.  
}  
Numa.prototype = new Num; // Задана динамическая связь с родителем.  
function _put()  
{  
    alert("Исходное число=" + this.number);  
}  
//--- Конец документации на объект Numa. --
```

Ниже приводятся два скрипта, которые вы можете запустить в своем браузере (листинги 10.2 и 10.3). Первый для проверки статического наследования, второй — для проверки динамического.

Листинг 10.2. Скрипт со статическим наследованием

```
<SCRIPT language=JavaScript>  
<!--  
    // Проверка статического наследования.  
    //-- Документация на объект Num. --  
    function Num(a)  
    {  
        this.number = a;  
        this.mul2 = _mul2;  
    }  
    function _mul2()  
    {  
        return this.number*2;  
    }  
    //-- Конец документации на объект Num. --
```

```
// Проверка Num.
var x = new Num(10);
var y = x.mul2();
alert("Проверка объекта Num: 10*2="+y);

/-- Документация на объект Numa (наследован от Num). --
function Numa(a)
{
    this.parent = Num; // Родителем объявлен Num.
    this.parent(a);    // Вызван конструктор родителя.
    this.put = _put;   // Объявлен новый метод у потомка.
}
function _put()
{
    alert("Исходное число="+this.number);
}
/-- Конец документации на объект Numa. --

// Проверка Num.
var z = new Numa(100);
z.put();

// Определяется новый метод для родителя.
Num.prototype.mul3 = _mul3;
function _mul3()
{
    return this.number*3;
}

// Проверка нового метода для родителя.
var t = new Num(10);
alert("Для родителя 10*3="+t.mul3());

// Проверка нового метода для потомка.
var k = new Numa(10);
alert("Для потомка: 10*3="+k.mul3());
/-->
</SCRIPT>
```

Листинг 10.3. Скрипт с динамическим наследованием

```
<SCRIPT language=JavaScript>
<!--
    // Проверка динамического наследования.
    //-- Документация на объект Num. --
    function Num(a)
    {
        this.number = a;
        this.mul2 = _mul2;
    }

    function _mul2()
    {
        return this.number*2;
    }
    //-- Конец документации на объект Num. --

    // Проверка Num.
    var x = new Num(10);
    var y = x.mul2();
    alert("Проверка объекта Num: 10*2="+y);

    //-- Документация на объект Numa (динамически наследован от Num) .
    function Numa(a)
    {
        this.parent = Num; // Родителем объявлен Num.
        this.parent(a);     // Вызван конструктор родителя.
        this.put = _put;    // Объявлен новый метод у потомка.
    }
    Numa.prototype = new Num; // Задана динамическая связь с родителем.
    function _put()
    {
        alert("Исходное число="+this.number);
    }
    //-- Конец документации на объект Numa. --

    // Проверка Num.
```

```

var z = new Numa(100);
z.put();

// Определяется новый метод для родителя.
Num.prototype.mul3 = _mul3;
function _mul3()
{
    return this.number*3;
}

// Проверка нового метода для родителя.
var t = new Num(10);
alert("Для родителя: 10*3="+t.mul3());

// Проверка нового метода для потомка.
var k = new Numa(10);
alert("Для потомка: 10*3="+k.mul3());
//-->
</SCRIPT>

```

Сложный метод *sort* объекта *Array*

К Ивану пришел его приятель Петр Мячиков и задал вопрос о методе *sort* объекта *Array*.

— Я ничего не понял про сортировку массива, в частности, про функцию, которая задает правила сравнения двух элементов. Посмотри на последний пример, приведенный в *Приложении 2* (табл. 10.3):

Таблица 10.3

Метод	Описание
<code>sort(function)</code> <code>sort()</code>	Сортировка массива. Параметром является имя функции, которая задает правила сравнения двух элементов. Если параметр опущен, элементы сортируются в лексикографическом порядке: <pre> var set = new Array("zebra", "ant", "dog", "cat"); set.sort(); alert(set); </pre>

Таблица 10.3 (окончание)

Метод	Описание
	На экран будет выведено сообщение: «ant, cat, dog, zebra».
	Функция <code>function</code> должна содержать два аргумента и возвращать:
	☐ отрицательное число, когда первый аргумент считается расположенным левее второго в смысле определяемого порядка;
	☐ 0, когда аргументы считаются равнозначными;
	☐ положительное число, когда первый аргумент считается правее второго в смысле определяемого порядка.
	<pre>var set = new Array(26,71,9,1); function Compare(a,b) { return a - b; } set.sort(Compare); alert(set);</pre>
	На экран будет выведено сообщение: «1, 9, 26, 71».

— Мне не понятно, что возвращает функция `Compare`, если `a` и `b` не определены в момент обращения к ней: `set.sort(Compare)`, — сказал Мячиков.

Иван стал объяснять.

В качестве аргумента метода `sort` указывается функция для сравнения двух элементов `Compare` (имя для этой функции мы придумываем сами, оно может быть любым).

Запись `set.sort(Compare)` не является записью композиции двух функций, т. е. она не означает, что сначала работает `Compare`, а затем над результатом работает `sort`. Указание `Compare` выступает в качестве аргумента функции `sort`. Мы привыкли к тому, что аргументом является переменная, константа или выражение. Оказывается, аргументом может быть и функция, вернее ссылка на функцию. Именно роль ссылки на функцию играет ее имя.

Вызов `set.sort(Compare(a,b))` будет выполняться как композиция, т. е. сначала работает функция `Compare(a,b)`, а затем ее значение передается в функцию `sort`.

Вызов `set.sort(Compare)` означает совсем другое. Здесь в функцию `sort` передается ссылка на функцию `Compare`. Внутри функции `sort` функция `Compare` вызывается как обычно: `Compare(список фактических параметров)`.

В нашем случае функция `Compare` должна иметь два аргумента и своим возвращаемым значением показывать, какой из этих двух аргументов будет первым в смысле того порядка, который мы хотим задать:

- ☐ если в этом порядке `a` идет перед `b`, нужно вернуть отрицательное число;
- ☐ если в этом порядке `a` идет после `b`, нужно вернуть положительное число;
- ☐ если в этом порядке `a` равнозначно `b`, нужно вернуть ноль.

Как работает метод `sort`? Это неизвестно, но ясно: чтобы переставлять элементы в массиве, ему необходимо эти элементы попарно сравнивать друг с другом. Вот всякий раз, когда нужно сравнить два элемента, метод `sort` и обращается к функции `Compare(a,b)`. Если функция возвращает отрицательное число, то `a` ставится впереди `b` как меньший элемент.

Если массив состоит из чисел и мы хотим упорядочить его по возрастанию, то в качестве функции `Compare` годится такой код:

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

Тот же результат можно получить короче:

```
function Compare(a,b)
{
    return a - b;
}
```

Для упорядочивания по убыванию подойдет такой код:

```
function Compare(a,b)
{
    return b - a;
}
```

Если же мы хотим оставить массив на месте, то можно написать:

```
function Compare(a,b)
{
    return 0;
}
```

Предположим, придуман такой порядок.

Число *a* считается меньше числа *b*, если у него меньше младшая цифра. Числа считаются равными, если их младшие цифры совпадают, и число *a* больше числа *b* в противном случае.

Для сортировки массива по этому правилу можно написать такую функцию сравнения:

```
function Compare(a,b)
{
    if      (a%10 < b%10)  return -1;
    else if (a%10 == b%10) return 0;
    else                return 1;
}
```

То же самое можно задать короче:

```
function Compare(a,b)
{
    return a%10 - b%10;
}
```

Например, массив [26, 71, 9, 1] после такой сортировки преобразуется в массив [71, 1, 26, 9].

Еще пример. Отсортируем числа в порядке возрастания их синусов:

```
function Compare(a,b)
{
    return Math.sin(a) - Math.sin(b);
}
```

Для того же исходного набора [26, 71, 9, 1] получим отсортированный массив [9, 26, 1, 71].

Массив как аргумент функции

Долго беседовал Иван с Мячиковым, но Петр не вникал. Тогда Иван решил написать свою функцию сортировки, чтобы показать Мячикову работу `sort` с функцией `Compare` «изнутри».

```
// Сортировка массива методом "пузырька".
// Аргументы функции:
//   set - сортируемый массив;
//   Comp - функция, задающая порядок.
// Возвращаемое значение: отсортированный массив.
// Замечание. Функция sort обращается при своей работе к функции Comp,
// которая должна работать, как было описано выше.
```

```
function sort(set, Comp)
{
    var change = true; // Были перестановки в массиве.
    while(change)
    {
        change = false;
        for(var i = 0; i < set.length - 1; i++)
            if (Comp(set[i],set[i + 1]) > 0)
            {
                // Перестановка в правильном порядке соседних элементов.
                var x      = set[i];
                set[i]     = set[i + 1];
                set[i + 1] = x;
                change     = true;
            }
    }
    return set;
}

function Compare(a,b)
{
    return a - b;
}

// Проверка сортировки.
var a = new Array(26,71,9,1);
var b = new Array();
var b = sort(a, Compare);
alert("Исходный массив: " + a);
alert("Отсортированный массив: " + b);
```

Запустите этот скрипт и убедитесь, что он работает.

Скрипт работает, да, но... Почему-то оба сообщения `alert` дают один и тот же результат: `[1, 9, 26, 71]`.

Получается, что если мы записываем массив как аргумент функции, а функция что-то с массивом делает, то исходный массив портится!

Происходит так потому, что в функцию передается не сам массив, а ссылка на него. Как исправить функцию `sort`, чтобы она не «портила» исходный массив?

Вопрос, несмотря на внешнюю простоту, довольно коварен.

Простое решение не помогает:

```
function sort(s, Comp)
{
    var set = new Array();
    set = s; // создаем локальную копию массива?
    var change = true;
    while(change)
    {
        change = false;
        for(var i = 0; i < set.length - 1; i++)
            if (Comp(set[i], set[i + 1]) > 0)
            {
                var x      = set[i];
                set[i]      = set[i + 1];
                set[i + 1] = x;
                change      = true;
            }
    }
    return set;
}
```

Дело в том, что код `set = s`; создает копию ссылки, а не копию массива.

Для создания «настоящей» копии придется «честно» копировать все элементы массива:

```
function sort(s, Comp)
{
    // Формируем локальную копию массива s.
    var set = new Array();
    for (var i = s.length; -- i >= 0;) set[i] = s[i];
    ...
    // Всю работу выполняем в локальной копии.
    ...
    return set;
}
```

Теперь массив-аргумент `s` испорчен не будет! Правда, возникает новый любопытный вопрос. Когда пишем `return set`, что возвращает функция? Ссылку на локальный массив `set` или сам массив?

Если функция возвращает ссылку, то это катастрофа: будет возвращена ссылка на локальный массив, который уничтожается при выходе из функции.

Однако в этом месте JavaScript поступает благоразумно. По команде **return** *set* сначала копируется результат, а уж потом локальный массив *set* уничтожается.

Сортировка методом «пузырька»

Иван сказал, что приведенный выше алгоритм сортировки не оптимальный:

```
function sort(s, Comp)
{
    var i;
    // Формируем локальную копию массива.
    var set = new Array();
    for (i = s.length; -- i >= 0;) set[i] = s[i];

    var change = true; // Были перестановки в массиве.
    while(change)
    {
        change = false;
        for(i = 0; i < set.length - 1; i ++)
            if (Comp(set[i], set[i + 1]) > 0)
            {
                // Перестановка в правильном порядке соседних элементов.
                var x      = set[i];
                set[i]     = set[i + 1];
                set[i + 1] = x;
                change     = true;
            }
    }
    return set;
}
```

Работает он так.

В цикле **while** просматривается массив и сравниваются соседние элементы. Если элементы стоят неправильно, они обмениваются местами, а переменная *change* устанавливается в **true**.

Просмотры повторяются, пока значение переменной *change* равно **true**.

Прокрутим алгоритм для массива [26,71,9,1] (упорядочивание по возрасту).

Первый просмотр

26,71,9,1

26,9,71,1

26,9,1,71 «Всплыл» на свое место элемент-пузырек 71.

Второй просмотр

26,9,1,71

9,26,1,71

9,1,26,71 «Всплыл» на свое место элемент-пузырек 26.

Третий просмотр

9,1,26,71

1,9,26,71

1,9,26,71 «Всплыл» на свое место элемент-пузырек 9.

Четвертый просмотр

1,9,26,71

1,9,26,71

1,9,26,71

При последнем просмотре не было сделано ни одной перестановки, поэтому переменная `change` остается равной `false` и просмотры заканчиваются.

Каждый просмотр заставляет один элемент «всплывать» на свое место.

Именно поэтому метод и называется: «метод пузырька».

Ясно, что при каждом новом просмотре длину просматриваемой части можно сокращать на 1. Понятно также, что какая-то часть массива уже может стоять в правильном порядке, и «пузырьки» будут «всплывать» на свое место, которое не обязательно отличается на 1 от предыдущего «всплытия». Значит, длину просматриваемой части в оптимальном алгоритме нужно сокращать до места «всплытия» последнего «пузырька», что может оказаться существенно больше единицы.

Посмотрим это на примере сортировки массива [2, 5, 1, 6, 7].

Первый просмотр (смотрим пять элементов)

2, 5, 1, 6, 7

2, 1, 5, 6, 7

2, 1, 5, 6, 7 «Всплыл» на свое место элемент-пузырек 5.

Понятно, что смотреть на "хвост" за элементом 5 теперь уже не надо.

Второй просмотр (смотрим два элемента)

2, 1

1, 2 «Всплыл» на свое место элемент-пузырек 2.

Перестановка была, и теперь надо смотреть один элемент, но один элемент всегда в полном порядке. Значит, цикл можно закончить.

Руководствуясь этими идеями, Иван написал улучшенный алгоритм «пузырька». У него получился скрипт, который вы можете проверить в своем браузере (листинг 10.4):

Листинг 10.4

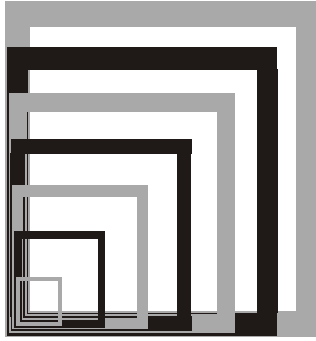
```
<SCRIPT language="JavaScript">
<!--
function sort(s, Comp)
{
    var i; // Индекс массива.
    // Формируем локальную копию массива.
    var set = new Array();
    for (i=s.length; --i>=0;) set[i]=s[i];
    var bound=set.length-1; // Верхняя граница просмотра или 0, если
                           // перестановок не было.

    while(bound)
    {
        var oldbound = bound;
        bound = 0;
        for(i=0; i<oldbound; i++)
            if (Comp(set[i],set[i+1])>0)
            {
                // Перестановка в правильном порядке соседних элементов.
                var x      = set[i];
                set[i]     = set[i+1];
                set[i+1]   = x;
                // Запомним новую границу просмотра.
                bound      = i;
            }
    }

    return set;
}
```

```
function Compare(a,b)
{
    return a-b;
}

// Проверка алгоритма.
var a = new Array(26,71,9,1);
var b = new Array();
b = sort(a, Compare);
alert("Исходный массив: "+a);
alert("Отсортированный массив: "+b);
//-->
</SCRIPT>
```

Часть III

Объектная модель браузера

Урок 11. Объекты и события браузера

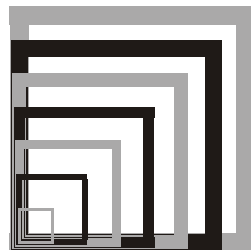
Урок 12. Формы (кнопки, строки ввода)

Урок 13. Формы (флажки, радиокнопки)

Урок 14. Формы (меню, многострочное поле)

В предыдущем разделе были изложены базовые понятия языка JavaScript, а также его синтаксис. Хотя конструкции языка и рассматривались в контексте HTML-кода (блок `<SCRIPT>...</SCRIPT>`), ничего полезного для гипертекста, кроме функций `alert`, `prompt`, `confirm`, показано не было. Но эти функции ничего не меняют на страничке, они работают в собственном окошке. А ведь было сказано, что, используя JavaScript, можно перейти к технологиям динамического HTML с его возможностями перерисовывать заново отдельные элементы документа, реагировать на действия пользователя.

Урок 11



Объекты и события браузера

Чтобы изменять уже нарисованный браузером экран или создавать новые окна, нужно проникнуть в программистскую «кухню» браузера, получить доступ к его объектам. Ведь браузер не только отображает теги на экране, он строит для них объекты, помещая их в иерархическую модель документа. Во время работы браузер постоянно «следит» за моделью. Если свойство объекта изменилось, меняется экранный образ. Менять свойства объектов можно при помощи JavaScript. Вот и динамика на экране!

Объектная модель любого документа имеет всегда одну и ту же структуру, независимо от вида конкретной страницы (рис. 11.1).

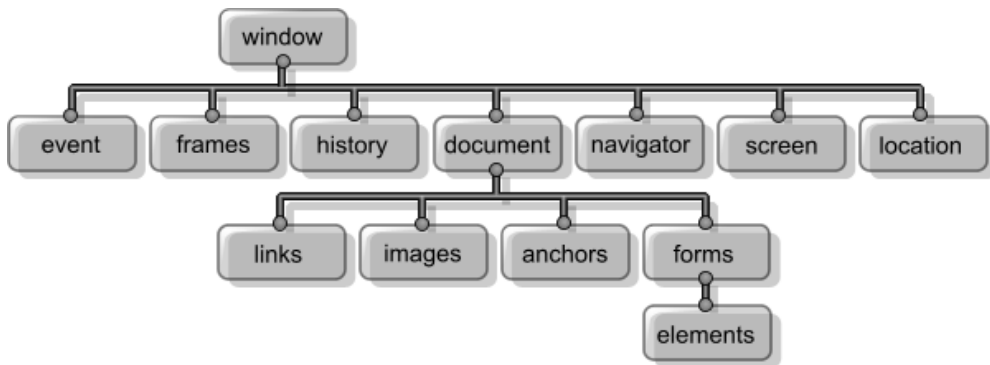


Рис. 11.1. Объектная модель документа

Строго говоря, все представленные на схеме «объекты» на самом деле являются экземплярами одноименных объектов. Эти экземпляры создаются браузером для каждого конкретного документа в индивидуальном порядке. В дальнейшем будем называть эти экземпляры просто «объекты».

Объект *window*

Объект `window` располагается на вершине иерархии. Это означает, что все остальные объекты находятся ниже его, являются его свойствами.

Объект `window` содержит также свойства и методы окна, в котором браузер показывает документ.

Три метода нам уже хорошо известны: `alert`, `prompt` и `confirm`.

Если написать:

```
window.alert("Функция alert - это метод объекта window");
```

— то браузер покажет вспомогательное окно, в которое выведет заданное сообщение (рис. 11.2).

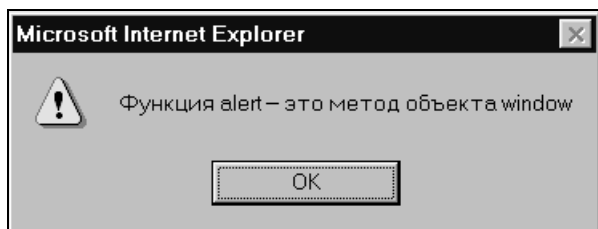


Рис. 11.2. Окно, создаваемое методом `alert`

Правда, раньше мы писали более просто:

```
alert("Функция alert - это метод объекта window");
```

Дело в том, что методы и свойства главного объекта `window` разрешается записывать без указания принадлежности. Следующие записи эквивалентны:

```
window.alert("Не все дома, но все пьют чай");
```

```
alert("Не все дома, но все пьют чай");
```

Открытие новых окон

Часто бывает удобно показать что-то пользователю в новом окне, оставляя открытым текущее. Открыть новое окно можно при помощи метода `open` объекта `window`. Вызвать метод можно одним из следующих способов:

```
var переменная = open();
```

```
var переменная = open(файл);
```

```
var переменная = open(файл, имя_окна);
```

```
var переменная = open(файл, имя_окна, параметры_окна);
```

Здесь:

- `переменная` — переменная. Указатель на экземпляр создаваемого окна;
- `файл` — строка. Имя файла, который нужно отобразить в открытом окне. Если параметр не задан, создается пустое окно;

- `имя_окна` — строка. Имя окна может использоваться для обращения к созданному окну (значение свойства `name` объекта `window`);
- `параметры_окна` — строка. Описание свойств окна. Когда параметр не задан, используются свойства окна по умолчанию.

Описание объекта `window`, его свойств и методов содержится в *Приложении 3* в конце книги. Рассмотрим здесь некоторые параметры окна. Они задаются в строке `параметры_окна` в виде:

"параметр1=значение1,параметр2=значение2,...,параметрN=значениеN"

Замечание

Не рекомендуется оставлять пробел после разделительной запятой, так как браузеру Netscape это почему-то не нравится.

Описание параметров (табл. 11.1).

Таблица 11.1

Параметр	Значение	Описание
<code>width</code>	Число	Ширина окна в пикселах. Минимальное значение — 100
<code>height</code>	Число	Высота окна в пикселах. Минимальное значение — 100
<code>scrollbars</code>	<code>yes</code> , <code>no</code> , 1, 0	Задаёт отображение линеек прокрутки
<code>resizable</code>	<code>yes</code> , <code>no</code> , 1, 0	Позволяет указать, может ли окно менять свой размер
<code>menubar</code>	<code>yes</code> , <code>no</code> , 1, 0	Позволяет указать, отображается ли полоса меню
<code>location</code>	<code>yes</code> , <code>no</code> , 1, 0	Позволяет указать, отображается ли полоса для ввода адреса
<code>status</code>	<code>yes</code> , <code>no</code> , 1, 0	Позволяет указать, отображается ли полоса статуса
<code>toolbar</code>	<code>yes</code> , <code>no</code> , 1, 0	Позволяет указать, отображается ли полоса кнопок (панель инструментов)

Приложение 4 содержит код странички с испытателем 2 для метода `open`. Чтобы лучше понять смысл описанных выше параметров, загрузите испытатель в браузер и поработайте с ним.

В качестве файла, задающего содержимого окна, можно указывать:

- HTML-файл, например, `ex1.htm`;
- картинку, например, `./pic/explorer.gif`;
- текстовый файл, например, `010401.txt`.

Метод `close` закрывает открытое окно. При этом обращение `close()` или `window.close()` закрывает текущее окно. Обращение `win.close()` или `window.win.close()` — закрывает окно с указателем `win`, которое было создано из текущего окна при помощи метода `open`:

```
var win = window.open(...);
```

Объект *document*

Объект `document` содержит объекты (они являются свойствами объекта `document`), моделирующие теги, расположенные в блоке `<HTML>...</HTML>`.

Один из самых употребительных методов объекта `document` — метод `write`. Этот метод позволяет осуществить запись в окно документа. Перед использованием метода `write`, нужно открыть запись (метод `open`), а по окончании вывода — закрыть ее (метод `close`).

Формат метода `open`:

```
переменная.document.open();
```

Здесь `переменная` — это указатель окна, которое открывается на запись.

Формат метода `write`:

```
document.write(строка); или переменная.document.write(строка);
```

Здесь:

`переменная` — указатель окна, в которое будет направлен вывод. Если `переменная` не задана, вывод направляется в текущее окно браузера;

`строка` — строка, в которой задается содержание вывода.

Формат метода `close`:

```
document.close(); или переменная.document.close();
```

Здесь `переменная` — это указатель окна, которое закрывается для записи.

Замечание

Метод `document.open` использовать необязательно, так как метод `document.write` при записи данных в закрытый документ автоматически очищает старый документ и открывает его для новой записи.

Пример. Создание страницы «на лету» (рис. 11.3).

```
var win = open("", "", "width=300,height=200");  
win.document.write("Пример. Эта страница построена ");
```

```
win.document.write("программным способом.");  
win.document.close();
```

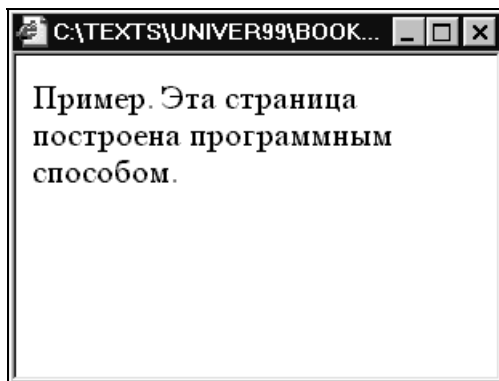


Рис. 11.3. Вывод в окно неразмеченного текста

При выводе текста в окно он размещается браузером как единый абзац независимо от того, сколько раз была использована функция `write`.

При помощи метода `write` в окно можно выводить не только обычный текст, но и текст, размеченный тегами (рис. 11.4).

```
var win = open("", "", "width=300,height=200");  
win.document.write("<h1>Пример</h1>" +  
    "<hr>" +  
    "<p>" +  
    "Эта страница построена программным способом."  
    );  
win.document.close();
```



Рис. 11.4. Вывод в окно размеченного текста

События

Браузер умеет отслеживать действия пользователя и реагировать на них. Это хорошо известно. Если щелчок мыши в окне приходится по ссылке, браузер загружает новый документ. Клавиатурный аккорд `<Alt>+<←>` возвращает прежнее содержимое окна.

Еще примеры. Пользователь подвел курсор мыши к картинке. Браузер тут же отображает в небольшом окошке `alt`-текст из соответствующего тега `<img>`. Пользователь использует клавиши со стрелками, и браузер послушно прокручивает окно.

Эти простые наблюдения показывают: мышь и клавиатура «информируют» браузер о том, что с ними делает пользователь. Кроме того, браузер «в курсе» хода загрузки документа. Если не загрузилась картинка (нет на диске, оборвалась связь с сервером), браузер замещает ее прямоугольником.

Все эти информационные сообщения называются событиями. На некоторые события браузер активно отвечает (открывает и закрывает окна, прокручивает экран, загружает документы, выводит информационные сообщения), на другие спокойно взирает, «сложив руки». Мы щелкаем мышью по обычному тексту, и ничего не происходит. Браузер не заметил щелчка? Нет, заметил, но у него нет предписаний на этот случай.

Вот обычная кнопка, которую браузер рисует при помощи HTML-кода (рис. 11.5):

```
<FORM>
  <INPUT type=button value="Обычная кнопка">
</FORM>
```

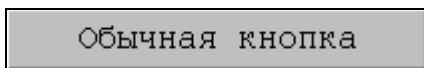


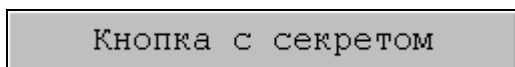
Рис. 11.5. Обычная кнопка

Кнопка задана тегом `<INPUT>` внутри тега-формы `<FORM>...</FORM>`. Атрибут `value` задает надпись на кнопке, а атрибут `type` описывает тип поля ввода. Подробный разговор о формах пойдет в следующем уроке.

Реагирует ли браузер на нажатие этой кнопки? Да, конечно, он ее перерисовывает, создавая впечатление, что кнопка вдавливаются мышью в экран. Но больше ничего не происходит.

А теперь попробуйте нажать на другую кнопку (рис. 11.6):

```
<FORM>
  <INPUT type=button value="Кнопка с секретом"
    onclick="alert('Привет!')">
</FORM>
```

**Рис. 11.6. Кнопка с секретом**

На экране появляется окно `alert` с текстом «Привет!» (рис. 11.7).

**Рис. 11.7. Сообщение выводится по нажатию на кнопку с секретом**

Что изменилось в HTML-коде? В теге `<INPUT>` появился новый атрибут `onclick`. В качестве значения атрибута в кавычках записана небольшая программа на JavaScript: `"alert('Привет!')"`.

Атрибут `onclick` сообщает браузеру, что надо делать, когда кнопка нажата! Длинную программу записывать как значение атрибута не очень удобно, поэтому часто программируют следующим образом (листинг 11.1).

Листинг 11.1

```
<SCRIPT language=JavaScript>
<!--
function Fun()
{
    var now = new Date();
    alert("Сегодня: "+now.getDate()+"/"+
        (now.getMonth()+1)+"/"+ now.getFullYear()+
        "\nСейчас: "+now.getHours()+":"+now.getMinutes());
}
//-->
</SCRIPT>
<FORM>
    <INPUT type=button value=Календарь onclick="Fun();">
</FORM>
```

На рис. 11.8 изображена кнопка, созданная с помощью приведенного выше HTML-кода.

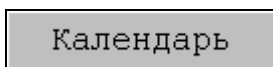


Рис. 11.8. Кнопка **Календарь**

В этом примере в атрибуте `onclick` задано короткое обращение к функции `Fun()`, а сама функция описана обычным образом в блоке `<SCRIPT>...</SCRIPT>`. Если теперь нажать на кнопку **Календарь**, на экране появится сообщение, показанное на рис. 11.9.

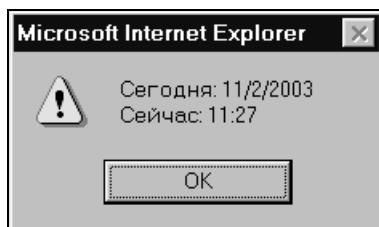


Рис. 11.9. Сообщение выводится по нажатию на кнопку **Календарь**

Функция `Fun()`, которая работает при возникновении события `onclick` — (щелчок) на кнопке, называется *обработчиком* этого события.

Загрузите в браузер испытатель 3 из *Приложения 3*. Прodelайте несколько опытов.

1. Измените надпись на кнопке. Для этого вместо слова «Календарь» задайте другой текст. Новая надпись появится на кнопке, когда поле ввода потеряет фокус. Это произойдет, если щелкнуть мышью за пределами вводимой строки.
2. Замените код функции `Fun` на другой. Например, на такой:

```
var sum = 0;  
for (var i = 100; i; i --) sum += i;  
alert("Сумма 1+2+...+100 = " + sum);
```

3. Напишите свою собственную функцию.

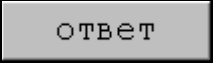
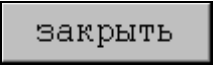
Итак, JavaScript позволяет записывать в HTML-кодах свои собственные обработчики событий, и, следовательно, делать страничку интерактивной, реагирующей на действия пользователя и другие «происшествия» (например, окончание загрузки документа).

Ключевое слово `onclick` в теге — это имя атрибута из серии других «событийных» атрибутов с приставкой «он» (см. *Приложение 3* в конце книги). Значением таких атрибутов является скриптовый код. Скрипт запускается, если происходит событие, кодируемое самим именем атрибута.

Атрибут `onclick` задает обработку события «щелчок мыши по тому месту на экране, которое занимает образ тега».

Пример интерактивного документа

На странице расположены одно поле ввода и четыре кнопки (листинг 11.2), которые имеют следующие описания:

-  — сообщает текущее время;
-  — открывает новое окно и выводит в него приветствие и кнопку, которая закрывает окно;
-  — поле ввода выражения для вычисления;
-  — вычисляет арифметическое выражение из поля ввода;
-  — закрывает окно.

Листинг 11.2. Код примера

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
  <TITLE>Интерактивная страница</TITLE>
  <SCRIPT language=JavaScript>
  <!--
    function CurTime()
    {
      var now = new Date();
      alert("Сейчас: "+now.getHours()+"."+now.getMinutes());
    }
    function Hallo()
    {
      var win=window.open("", "", "width=400,height=300");
      win.document.open();
      var str = "";
```

```

    str += '<H1>Привет!</H1>';
    str += '<HR>';
    str += '    <P>';
    str += '        Веб-программисты! Крепите свои ряды!';
    str += '    <P>';
    str += '    <FORM>';
    str += '        <INPUT type=button value=заккрыть';
    str += '            onclick="window.close()">';
    str += '    </FORM>';
    win.document.write(str);
    win.document.close();
}
//-->
</SCRIPT>
</HEAD>
<BODY bgcolor=white text=black link=blue alink=red vlink=purple>
    <H1>Интерактивная страница</H1>
    <HR>
    <FORM name=forma>
        <INPUT type=button value=время onclick="CurTime()">
        <INPUT type=button value=привет onclick="Hallo()">
        <INPUT type=text value="2*2" size=25 name=calc>
        <INPUT type=button value=ответ
            onclick="alert(eval(document.forma.calc.value))">
        <P>
        <INPUT type=button value=заккрыть onclick="window.close()">
    </FORM>
</BODY>
</HTML>

```

Примечание

Функция `CurTime` обрабатывает нажатие кнопки **время**.

Функция `Hallo` обрабатывает нажатие кнопки **привет**.

Функция `alert` обрабатывает нажатие кнопки **ответ**.

Аргументом функции `alert` является значение функции `eval`:

```
eval(document.forma.calc.value)
```

Аргументом функции `eval` является содержимое строки ввода, заданное как `document.forma.calc.value`. Здесь:

- ☐ `forma` — это объект, образ блока `<FORM name=forma ... >...</FORM>`;
- ☐ `calc` — это объект, образ тега `<INPUT name=calc ... >`;
- ☐ `value` — это свойство объекта `calc`.

Функция `window.close(...)` обрабатывает нажатие кнопки **заккрыть**.

Блок `<SCRIPT>...</SCRIPT>` с описанием функций `CurTime` и `Hallo` помещен в раздел заголовка HTML-кода. Так делают для того, чтобы браузер смог прочитать описание функций прежде, чем к этим функциям возникнет обращение: ведь заголовок документа загружается самым первым.

Зачетный класс

1. Что содержит объектная модель документа?
2. Отличаются ли по структуре модели двух разных документов?
3. Отличаются ли по содержанию модели двух разных документов?
4. На какие события реагирует браузер?
5. Какие атрибуты тегов задают обработку событий?
6. Что такое «обработчик события»?
7. Объясните принцип динамического управления гипертекстовой страницей.
8. Какой объект является корнем иерархической модели документа?
9. Какой объект содержит в себе модели всех тегов блока `<HTML>...</HTML>`?
10. Как из гипертекстового документа создать новое окно?
11. Как открыть запись в окно?
12. Как вывести в окно информацию?
13. Как закрыть запись в окно?
14. Как закрыть созданное окно?

Задания

1. Напишите приложение для вычисления площади круга (рис. 11.10):

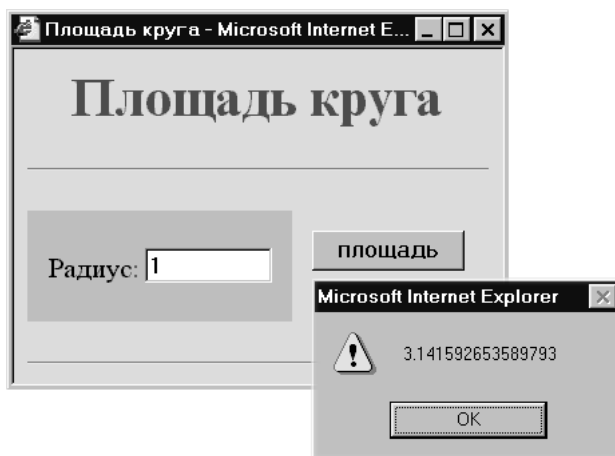


Рис. 11.10. Вычисление площади круга

2. Напишите приложение для показа решения нескольких задач, сформулированных ниже. Создайте страницу с меню, из которого нужная программа вызывалась бы для работы в новом окне браузера. Предусмотрите кнопки для закрытия вызываемых окон в них самих.
- а) Найти площадь треугольника по его сторонам a , b , c .
 - б) Найти расстояние точки (x, y) до начала координат.
 - в) Найти НОД двух чисел.
 - г) Вычислить число символов в строке, не считая пробелов.
 - д) Найти общие символы двух строк.
 - е) Найти разные символы двух строк.
 - ж) Сколько раз в числе встречается каждая его цифра?
 - з) Найти наиболее популярные цифры числа (цифры, которые встречаются в числе наибольшее число раз).
 - и) Найти число слов в тексте.
 - к) Проверить, является ли строка допустимой записью числа.
 - л) Вывести слово в обратном порядке.
 - м) Найти число дней между двумя датами.
 - н) Найти простые числа, не превосходящие заданного числа.
 - о) Проверить, можно ли представить натуральное число m в виде суммы квадратов двух натуральных чисел.
 - п) Все нулевые элементы массива переписать в его конец.
 - р) Сократить дробь m / n .

- с) Найти такие натуральные m_1 и n_1 , не имеющие общих делителей, что $m_1 / n_1 = m / n$.
- т) Получить, если это возможно, m разных случайных целых чисел из диапазона $[a, b]$.
- у) Узнать, попадет ли снаряд в цель, если известно расстояние до цели, начальная скорость снаряда и угол вылета.

Дополнительный материал

Суть динамического Web-программирования

Когда браузер загружает HTML-файл, он не просто отображает документ на экране, выполняя указания тегов, но и строит объект `window`, в котором содержится информация о документе, свойствах окна, в котором документ изображается, и информация о самом браузере. Все это вместе называется объектной моделью документа (или браузера).

Воздействие на объектную модель при помощи кодов на JavaScript позволяет менять внешний вид документа уже после того, как он построен браузером.

Построение собственных обработчиков событий наделяет документ программируемой интерактивностью.

Пример

Встретив в HTML-коде тег `<IMG src=pic.gif>`, браузер построит на экране картинку.

Если же написать `<IMG src=pic.gif onclick="alert('Привет!')">`, то браузер тоже построит картинку, но не только. Когда по картинке щелкают мышью, срабатывает функция `alert` — метод объекта `window`.

Скрипт-значение «событийного» атрибута может быть очень длинным:

```
<IMG src=pic.gif
  onclick=
    "function Compare(a,b)
      {
        return a - b;
      }
    var a = new Array(26,71,9,1);
    var b = new Array();
    for(var i = a.length; -- i >= 0;) b[i] = a[i];
    b.sort(Compare);
    alert('Исходный массив: ' + a +
      '\nОтсортированный массив: ' + b);">
```

Длинный скрипт редко записывают прямо в теге, на котором возникает событие. Обычно в теге оставляют лишь обращение к функции, а саму функцию помещают в отдельный блок `<SCRIPT>...</SCRIPT>`:

```
<SCRIPT language=JavaScript>
    function Compare(a,b)
    {
        return a - b;
    }
    function Make()
    {
        var a = new Array(26,71,9,1);
        var b = new Array();
        for(var i = a.length; -- i >= 0;) b[i] = a[i];
        b.sort(Compare);
        alert('Исходный массив: ' + a +
              '\nОтсортированный массив: ' + b);
    }
</SCRIPT>
<IMG src=pic.gif onclick="Make()">
```

Браузер строит объектную модель, в которой каждому тегу соответствует отдельный объект. Эти объекты помещаются в соответствующие места иерархической схемы документа.

Кроме того, в состав объекта `document` входят несколько специальных массивов со ссылками на объекты. Эти массивы (их называют *коллекциями*) удобно использовать для доступа к элементам модели без учета иерархии.

Так, коллекция с именем `all` содержит объекты, построенные для всех тегов документа, а коллекция `images` — объекты, построенные для тегов `<IMG>`.

Пусть документ закодирован так:

```
<HTML>
  <HEAD>
...
  <HEAD>
  <BODY>
    <FORM name=forma>
      <IMG name=mypic src=pic.gif>
    </FORM>
```

```
</BODY>
```

```
</HTML>
```

Тогда добраться до свойства `src` объекта, построенного для тега **** можно следующими способами:

`document.forma.mypic.src` — доступ с использованием иерархии кода.

`document.all["mypic"].src` — доступ через коллекцию `all`.

`document.images["mypic"].src` — доступ через коллекцию `images`.

В качестве индекса массива (коллекции) разрешается использовать имя тега — браузер автоматически преобразует такой «индекс» в число.

Рассмотрим пример:

```
<SCRIPT language=JavaScript>
```

```
function Make()
```

```
{
```

```
    document.images[0].src="pic1.gif";
```

```
}
```

```
</SCRIPT>
```

```
<IMG src=pic.gif onclick="Make()">
```

Если щелкнуть по картинке `pic.gif`, она заменится на картинку `pic1.gif`. Замена картинки задается кодом `document.images[0].src="pic1.gif"`.

Ссылки на картинки хранятся в массиве `images` следующим образом. Первой картинке соответствует ссылка `document.images[0]`, второй — `document.images[1]` и т. д.

Понятно, что использовать числовые индексы не очень удобно. Нужно высчитывать порядковый номер картинки в HTML-файле. Кроме того, вставка в HTML-файл нового тега **** (или удаление) может привести к неверной работе скрипта, так как нумерация нарушается. Вот почему на практике в качестве индекса массива используют не число, а имя тега, заданное атрибутом **name** — браузер сам преобразует это имя в правильный индекс. Следующий код не чувствителен к вставкам и удалениям картинок в HTML-файле.

```
<SCRIPT language=JavaScript>
```

```
function Make()
```

```
{
```

```
    document.images["pic"].src="pic1.gif";
```

```
}
```

```
</SCRIPT>
```

```
<IMG name=pic src=pic.gif onclick="Make()">
```

«Услужливые» браузеры (к которым относится Microsoft Internet Explorer) позволяют обратиться к объекту, построенному для картинки и еще более просто:

```
<SCRIPT>
```

```
function Make()  
{  
    document.pic.src="pic1.gif";  
}
```

```
</SCRIPT>
```

```
<IMG name=pic src=pic.gif onclick="Make()">
```

Браузер будет самостоятельно просматривать иерархические уровни модели, пока не найдет объект с именем `pic`.

Управление логикой программного кода при помощи событий

Управление логикой программного кода при помощи событий — это вторая характерная черта ООП наряду с понятием объекта.

Немного упрощенно можно так объяснить разницу между классическим и объектно-ориентированным программированием.

Программа, написанная без принципов ООП, отражает иерархию логической структуры алгоритма:

главный модуль

модуль 01

модуль 0101

модуль 010101

модуль 010102

...

...

модуль 02

модуль 0201

модуль 020101

модуль 020102

...

...

...

Данные, попадающие на обработку в главный модуль, проходят, следуя кодам программы, по ветвям логической структуры, пока не окажутся полностью обработанными в каком-нибудь листе иерархического дерева.

Можно сказать, что в классическом программировании коды первичны, а данные вторичны.

В ООП нет единой структуры управления работой программы. Есть независимые друг от друга объекты:

объект01

объект02

объект03

...

Управление выполняется посредством событий: возникает событие, и браузер передает управление в нужный объект.

Например, на картинке выполнен щелчок. Браузер передает управление объекту, построенному для соответствующего тега `<IMG>`. Атрибут `onclick="Fun ()"` является указанием: при возникновении события «щелчок мыши на теге `<IMG>`» запустить функцию `Fun ()`.

Таким образом, атрибуты `onxxxxx="скрипт"` позволяют добавлять в объекты браузера наши собственные коды. Если в объекте предусмотрена стандартная обработка события, то она выполняется после того, как закончена работа нашего обработчика.

Рассмотрим такой пример.

Когда пользователь щелкает по образу этого тега на экране:

```
<A href=f.htm>документ f</A>
```

браузер передает событие объекту, построенному для тега `<A>`. Для события «щелчок мыши» в этом объекте предусмотрен стандартный обработчик — он загружает в окно документ `f.htm`.

Давайте попробуем «перехватить» это событие:

```
<A href=f.htm onclick="alert('Хода нет?')">документ f</A>
```

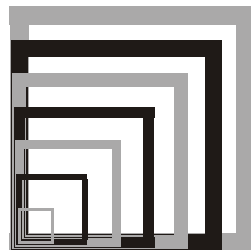
Щелкаем по ссылке, на экране возникает надпись «Хода нет?». Событие перехвачено? Закрываем окно `alert`, и видим, что браузер по-прежнему грузит документ `f.htm`.

Что произошло? При помощи атрибута `onclick` мы установили в объекте «отвод» на собственный обработчик. Но когда скрипт нашего обработчика выполнен, управление возвращается к стандартному обработчику и это вызывает загрузку документа `f.htm`.

Отключение стандартной обработки кодируется так:

```
<A href=f.htm onclick="alert('Хода нет!');return false">документ f</A>
```

Урок 12



Формы (кнопки, строки ввода)

Принцип программного управления

Для каждого тега внутри гипертекстовой страницы браузер создает объект. Доступ к этому объекту программист получает при помощи имени, которое задается атрибутом **name**. Таким образом, появляется возможность программного управления тегам.

Пусть, например, в HTML-коде задан тег для вывода на экран картинки (рис. 12.1):

```
<IMG src=./pic/risl.gif border=0 alt="" name=pic>
```



Рис. 12.1. Отображение тега `<img>` в окне браузера

Браузер строит для этого тега экземпляр стандартного объекта `Image` с именем `pic`. Получить к нему доступ можно при помощи эквивалентных конструкций:

```
window.document.images["pic"]
```

```
window.document.images.pic
```

или внутри текущего окна:

```
document.images["pic"]
```

```
document.images.pic
```

Известно, например, что у объекта `Image` есть свойство `src`, содержащее указание на файл с картинкой. Можно в программе прочитать это свойство, записав обращение:

```
document.images.pic.src
```

Можно, не только прочитать, но и изменить значение этого свойства:

```
document.images.pic.src = "/pic/ris2.gif"
```

и картинка на экране изменится соответствующим образом (рис. 12.2):



Рис. 12.2. Изображение тега `<img>` на экране изменилось

Блок `<Form>`

Работая с гипертекстовыми документами, мы не раз имели возможность вводить текст в поле ввода, выбирать пункт меню, отмечать флажком правильный ответ, нажимать кнопку. При этом мы всегда ожидали от документа осмысленной реакции.

Все эти интерфейсные элементы выводятся на экран браузера при помощи команды-контейнера `<FORM>...</FORM>`, а кнопки, поля ввода, флажки и меню задаются внутри этого блока отдельными тегами.

Блок `<FORM>...</FORM>` называется формой. В открывающем теге можно записывать следующие атрибуты (табл. 12.1).

Таблица 12.1

Атрибут	Описание
<code>action</code>	Задаёт адрес для отправки формы по сети. Можно послать форму на сервер, где она будет обработана программой (CGI-скриптом), а можно указать в качестве значения атрибута адрес электронной почты, и форма будет послана по e-mail. Например, можно написать так: <code><FORM action="mailto:myaddres@mail.ru"></code>

Таблица 12.1 (окончание)

Атрибут	Описание
enctype	Задаёт кодирование передаваемых по сети данных. В этой книге мы не рассматриваем серверных программ, которые принимают и автоматически обрабатывают полученные из формы данные. Но можно легко устроить так, что браузер пошлёт форму, заполненную пользователем, по вашему электронному адресу. Вы эти данные изучите сами или направите на обработку в собственную программу. Все будет хорошо работать, если задать значение атрибута enctype так: <pre><FORM action="mailto:myaddres@mail.ru" enctype="text/plain"></pre>
method	Задаёт способ передачи формы. Атрибут имеет два значения: get и post. Метод post более универсальный. Не вникая в детали работы этих двух методов, предлагаем «магическую» запись, которая заставит браузер послать форму по указанному адресу e-mail: <pre><FORM action="mailto:myaddres@mail.ru" enctype="text/plain" method="post"></pre>
name	Этот атрибут задаёт имя формы. Когда имя задано, можно использовать его в скриптах для доступа к полям формы. Если форму посылать куда не надо, а нужна она для создания диалога с пользователем, то из всех атрибутов потребуется только один — name: <pre><FORM name=anketa></pre> Теперь в JavaScript-коде можно обратиться к этой форме так: <code>document.anketa</code>. Если форма содержит строковое поле с именем <code>work</code>, то можно присвоить содержимое этого поля переменной: <pre>var str = document.anketa.work.value;</pre> Можно изменить значение этого поля: <pre>document.anketa.work.value = "веб-мастер";</pre> И экранное отображение поля после этого присваивания соответственно изменится.

Посылка формы по e-mail

Приведенный ниже пример (листинг 12.1) можно использовать как образец кодирования посылки формы по электронной почте.

Листинг 12.1

```

<FORM action=mailto:myaddres@mail.ru enctype="text/plain" method="post">
  Нажмите &#147;сброс&#148; и заполните заказ.<BR>
  <INPUT type=reset value=сброс><BR>
  <P>
    <TABLE border=0 cellspacing=0 cellpadding=5>
      <TR>
        <TD>Ваше имя:</TD>
        <TD>
          <INPUT type=text size=33 name=name
            value="Горбунков Семен Семенович">
        </TD>
      </TR>
      <TR>
        <TD>Ваш адрес:</TD>
        <TD>
          <INPUT type=text size=33 name=email value=gorbun@mail.ru>
        </TD>
      </TR>
      <TR>
        <TD>Прислать:</TD>
        <TD>
          <INPUT type=text size=33 name=obj
            value="Программный соковыжиматель">
        </TD>
      </TR>
    </TABLE>
    <INPUT type=submit value="послать заказ">
  </FORM>

```

Если поместить этот код в HTML-программу, то на гипертекстовой страничке возникнет картинка, подобная той, что изображена на рис. 12.3.

Пользователь нажимает кнопку **сброс**, и поля принимают значения, установленные по умолчанию. Форма заполняется, и по нажатию кнопки **послать заказ** отправляется по адресу **myaddres@mail.ru**. Обычно браузер не сам отправляет форму, а поручает эту работу почтовой программе, установленной по умолчанию в операционной системе компьютера (например, Outlook). Письмо посылается в виде:

```

Subject: Форма отправлена из Microsoft Internet Explorer
name=Горбунков Семен Семенович

```

email=gorbun@mail.ru

obj=Программный соковыжиматель

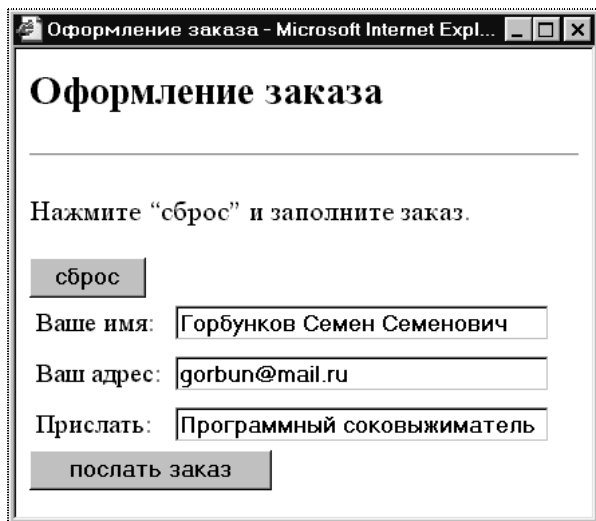


Рис. 12.3. Форма для оформления заказа

Таким образом, письмо составляется из пар *имя=значение*, которые выбираются по порядку из полей формы. В качестве обратного адреса в письме указывается адрес пользователя компьютера, с которого отправляется форма.

Вы можете провести серию опытов, посылая формы самим себе.

Замечание

Форма отправляется браузером по электронной почте лишь в том случае, если в его настройках указан соответствующий уровень безопасности.

Приведенный код наглядно показывает, что внутри блока `<FORM>...</FORM>` могут располагаться обычные теги.

Элемент `<INPUT>`

С помощью тега `<INPUT>` в форме можно задать разные элементы ввода и управления.

Атрибут *name*

Каждый тег `<INPUT>` может содержать атрибут **name**. Значение этого атрибута можно использовать как имя объекта, который браузер создает для этого тега.

Пусть, например, задана форма (ее вид на экране показан на рис. 12.4):

```
<FORM name=anketa>  
  <INPUT type=text value=крошкосшиватель name=obj>  
</FORM>
```

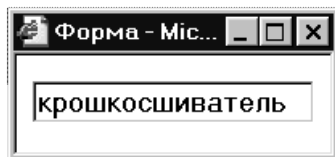


Рис. 12.4. Форма с единственной строкой ввода

В программе на JavaScript можно использовать конструкцию `document.anketa.obj.value` для доступа к свойству `value` объекта, построенного для тега `<INPUT>`. Эта запись «расшифровывается» так:

<code>document</code>	— объект <code>document</code> (родитель — объект <code>window</code>);
<code>document.anketa</code>	— объект <code>anketa</code> (родитель — объект <code>document</code>);
<code>document.anketa.obj</code>	— объект <code>obj</code> (родитель — объект <code>anketa</code>);
<code>document.anketa.obj.value</code>	— свойство <code>value</code> объекта <code>obj</code> — соответствует атрибуту <code>value</code> тега <code><INPUT></code> с именем <code>obj</code> .

Можно прочесть значение свойства `value`:

```
var str = document.anketa.obj.value;
```

А можно и изменить значение этого свойства:

```
document.anketa.obj.value = "пылеистребитель";
```

Такое присваивание приведет к мгновенной перерисовке экрана. Текст «крошкосшиватель» в поле ввода заменится на «пылеистребитель» (рис. 12.5).

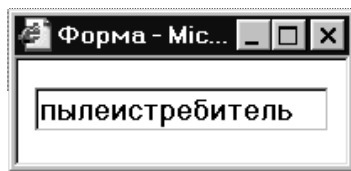


Рис. 12.5. Измененное содержимое строки ввода

Значение атрибута `name` приравнивается к имени переменной, поэтому при выборе названия нужно использовать все правила построения идентифика-

торов, рассмотренные в Уроке 6. В частности, нельзя использовать в имени русские буквы и цифру в качестве первого символа.

Имена, заданные атрибутом `name`, должны быть разными внутри одной формы, но могут совпадать с именами полей других форм — браузер не запутается, ведь формы разные и они отображаются в разные объекты.

Экранные кнопки

Конкретный вид интерфейсного элемента `<INPUT>` задается атрибутом `type`. Рассмотрим значения этого атрибута, задающие экранные кнопки (табл. 12.2).

Таблица 12.2

Значение	Описание	Пример кнопки
Reset	Задает кнопку, нажатие на которую автоматически возвращает форму в исходное состояние (все поля ввода формы принимают значения, установленные в HTML-коде). Атрибут <code>value</code> задает надпись на кнопке. <code><INPUT type=reset value=сброс></code>	
submit	Задает кнопку, нажатие на которую отправляет форму в сеть. Атрибут <code>value</code> задает надпись на кнопке. <code><INPUT type=submit value=послать></code>	
button	Задает обычную кнопку. <code><INPUT type=button value="нажми меня"></code>	

Программное управление

Объект, построенный браузером для тега `<INPUT type=button>` (или тегов `<INPUT type=submit>`, `<INPUT type=reset>`) имеет свойство `value`, которое соответствует одноименному атрибуту тега. Ниже приводится пример, демонстрирующий работу с этим свойством (листинг 12.2).

Загрузите в браузер приведенный ниже текст. Поработайте с экранными кнопками и разберитесь в алгоритме работы программы.

Листинг 12.2

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
```

```

        content="text/html; charset=windows-1251">
<TITLE>Умная кнопка</TITLE>
<SCRIPT language=JavaScript>
<!--
    var num=0;
    var name0="число нажатий: 0";
    //-->
</SCRIPT>
</HEAD>
<BODY bgcolor=white text=black>
<H2>Умная кнопка</H2>
<HR>
<FORM name=count>
    <INPUT type=button value=сброс
        onclick="document.count.key.value=name0;num=0;">
    <INPUT type=button name=key value="число нажатий: 0"
        onclick="document.count.key.value='число нажатий: '+(++num);">
</FORM>
</BODY>
</HTML>

```

Внешний вид документа показан на рис. 12.6.

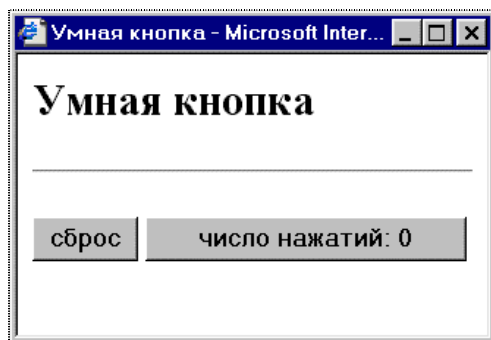


Рис. 12.6. Вид документа «Умная кнопка»

Можно упростить обращение к полям формы, если использовать указание **this** — ссылку на текущий экземпляр объекта и указание **this.form** — ссылку на объект, построенный для текущей формы:

```

<FORM>
    <INPUT type=button value=сброс
        onclick="this.form.key.value=name0;num=0;">

```

```
<INPUT type=button name=key value="число нажатий: 0"
      onclick="this.value='число нажатий: '+(++num);">
</FORM>
```

Как видите, в этом коде не потребовалось имя для формы. Синонимом имени внутри формы оказалась конструкция `this.form`. А внутри второго тега `<INPUT>` не потребовалось имя этого тега, вместо него использован синоним `this`.

Замечание

Объект, построенный для любого элемента управления или ввода, содержит свойство `form`, являющееся указателем на объект-форму, содержащий этот элемент.

Строка ввода

Значения `text` и `password` для атрибута `type` тега `<INPUT type=... >` задает строку ввода (табл. 12.3).

Таблица 12.3

Значение	Описание	Пример кнопки
text	Обычная строка ввода. Дополнительные атрибуты: ❑ Атрибут <code>value</code> задает значение поля ввода по умолчанию (именно это значение восстанавливается, когда пользователь нажимает кнопку <code><INPUT type=reset></code>); ❑ Атрибут <code>size</code> задает размер поля в символах. Если атрибут опущен, размер поля определяется длиной строки, заданной как значение атрибута <code>value</code>; ❑ Атрибут <code>maxlength</code> задает максимальную длину строки, которую можно вписать в поле ввода. Если атрибут пропущен, длина не ограничена; ❑ Атрибут <code>readonly</code> определяет строку «только для чтения». Такую строку пользователь не может редактировать. <pre><INPUT type=text value="Варвара" size=10 maxlength=30></pre>	Варвара

`<INPUT type=text value="Варвара"`

`size=10 maxlength=30>`

Таблица 12.3 (окончание)

Значение	Описание	Пример кнопки
password	Строка для «секретного» ввода. От значения <code>text</code> отличается только тем, что символы на экране при вводе отображаются звездочками. Секретная строка не копируется в буфер обмена.	
	<pre><INPUT type=password value="" size=10 maxlength=10></pre>	

Программное управление

Объект, построенный браузером для тега `<INPUT type=text>` (или `<INPUT type=password>`) имеет следующие свойства:

- `value` — текущее содержание строки ввода;
- `defaultValue` — значение атрибута `value` в соответствующем теге `<INPUT>`.

Чтобы лучше понять смысл свойств `value` и `defaultValue`, загрузите в браузер испытатель 4 из *Приложения 4* и поработайте с ним. Поработайте также на испытателе 5 и по коду испытателя разберитесь, как он устроен.

Зачетный класс

1. Нажатие на кнопку **пуск** должно вывести на экран слово **привет**. Какие коды это делают, а какие нет и почему?

а) `<FORM name=forma>`

```
<INPUT type=text value="" size=10 name=str>
```

```
<INPUT type=button value=пуск onclick="this.form.str='привет' ">
```

```
</FORM>
```

б) `<FORM name=forma>`

```
<INPUT type=text value="" size=10 name=str>
```

```
<INPUT type=button value=пуск
```

```
onclick="this.form.str.value='привет' ">
```

```
</FORM>
```

в) `<FORM name=forma>`

```
<INPUT type=text value="" size=10 name=str>
```

```
<INPUT type=button value=пуск onclick="this.str='привет' ">
```

```
</FORM>
```

г) `<FORM name=forma>`

```
<INPUT type=text value="" size=10 name=str>
```

```
<INPUT type=button value=пуск onclick="str.value='привет' ">
```

```
</FORM>
```

д) `<FORM name=forma>`

```
<INPUT type=text value="" size=10 name=str>
```

```
<INPUT type=button value=пуск
```

```
onclick="document.forma.str.value='привет' ">
```

```
</FORM>
```

е) `<FORM name=forma>`

```
<INPUT type=text value="" size=10 name=str>
```

```
<INPUT type=button value=пуск
```

```
onclick="window.document.forma.str.value='привет' ">
```

```
</FORM>
```

ж) `<FORM name=forma>`

```
<INPUT type=text value="" size=10 name=str>
```

```
<INPUT type=button value=пуск onclick="forma.str.value='привет' ">
```

```
</FORM>
```

з) `<FORM name=forma>`

```
<INPUT type=text value="" size=10 name=str>
```

```
<INPUT type=button value=пуск onclick="this.str.value='привет' ">
```

```
</FORM>
```

2. На страничке нужно предусмотреть очистку строки ввода. Какие коды это делают и почему остальные работать не будут?

а) `<FORM name=forma>`

```
<INPUT type=button value=сброс onclick="forma.str.value=''">
```

```
<INPUT type=text value=привет size=10 name=str>
```

```
</FORM>
```

б) `<FORM name=forma>`

```
<INPUT type=reset value=сброс>
```

```
<INPUT type=text value=привет size=10 name=str>
```

```
</FORM>
```

в) `<FORM name=forma>`

```
<INPUT type=button value=сброс
```

```
onclick="window.document.forma.str.value=''">
```

```
<INPUT type=text value=привет size=10 name=str>
```

```
</FORM>
```

г) `<FORM name=forma>`

```
    <INPUT type=button value=сброс onclick="str.value=''">
```

```
    <INPUT type=text value=привет size=10 name=str>
```

```
</FORM>
```

д) `<FORM name=forma>`

```
    <INPUT type=button value=сброс
```

```
    onclick="document.forma.str.value=''">
```

```
    <INPUT type=text value=привет size=10 name=str>
```

```
</FORM>
```

е) `<FORM name=forma>`

```
    <INPUT type=button value=сброс onclick="document.forma.str=''">
```

```
    <INPUT type=text value=привет size=10 name=str>
```

```
</FORM>
```

ж) `<FORM name=forma>`

```
    <INPUT type=button value=сброс onclick="this.str.value=''">
```

```
    <INPUT type=text value=привет size=10 name=str>
```

```
</FORM>
```

з) `<FORM name=forma>`

```
    <INPUT type=button value=сброс onclick="this.form.str.value=''">
```

```
    <INPUT type=text value=привет size=10 name=str>
```

```
</FORM>
```

и) `<FORM name=forma>`

```
    <INPUT type=button value=сброс onclick="str.value=''">
```

```
    <INPUT type=text value=привет size=10 name=str>
```

```
</FORM>
```

к) `<FORM name=forma>`

```
    <INPUT type=clear value=сброс>
```

```
    <INPUT type=text value=привет size=10 name=str>
```

```
</FORM>
```

Задания

1. Напишите приложение — зачетный класс, по какой-либо теме, например, по географии или информатике. Приложение может иметь вид, представленный на рис. 12.7. Постарайтесь сделать вашу программу максимально универсальной, чтобы ее можно было достаточно легко приспособить к другой теме, другому числу вопросов, другому алгоритму вычисления оценки.

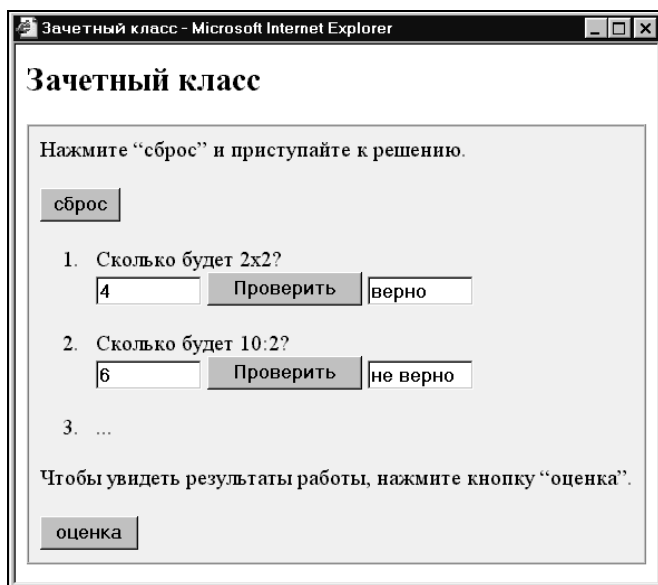


Рис. 12.7. Зачетный класс задания 1

2. Развитие предыдущей темы. Напишите приложение — зачетный класс с подсказками, которые отображаются в новом окне браузера (рис. 12.8).

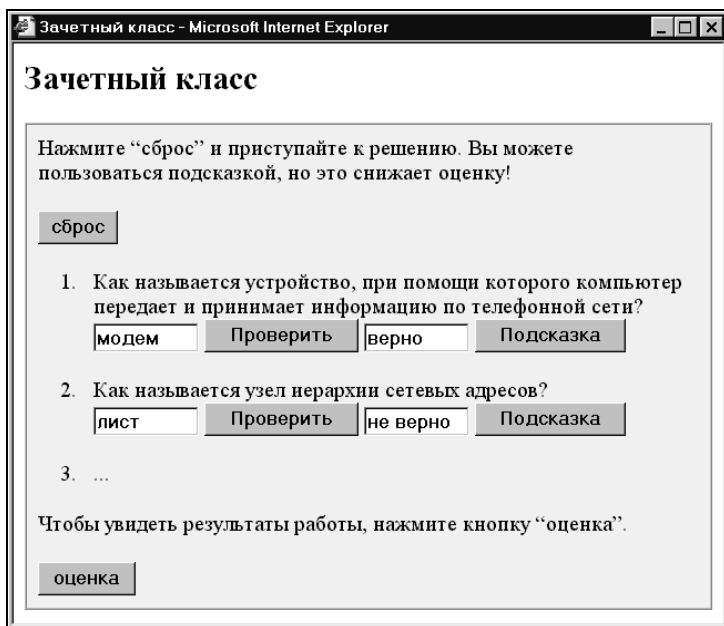


Рис. 12.8. Зачетный класс задания 2

3. Напишите экзамен по условным выражениям JavaScript. Возможный интерфейс этой программы представлен на рис. 12.9.

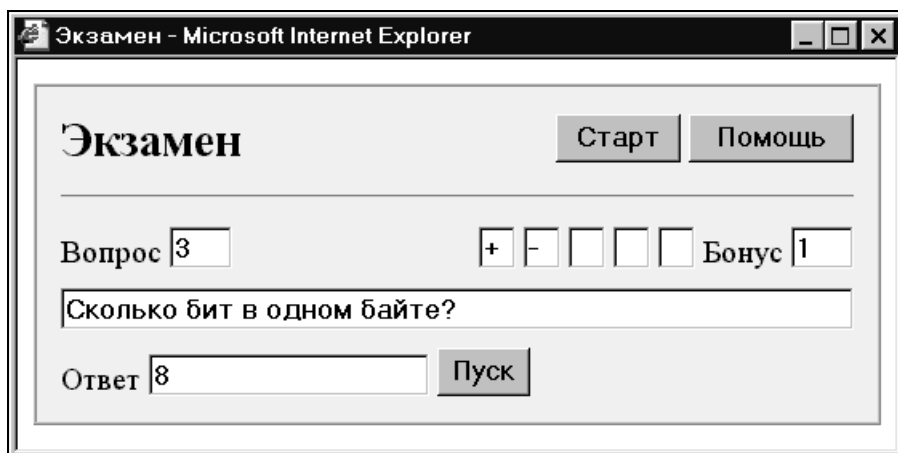


Рис. 12.9. Примерный вид приложения задания 3

4. Создайте вариант «Угадайки», используя формы. Возможный интерфейс показан на рис. 12.10.

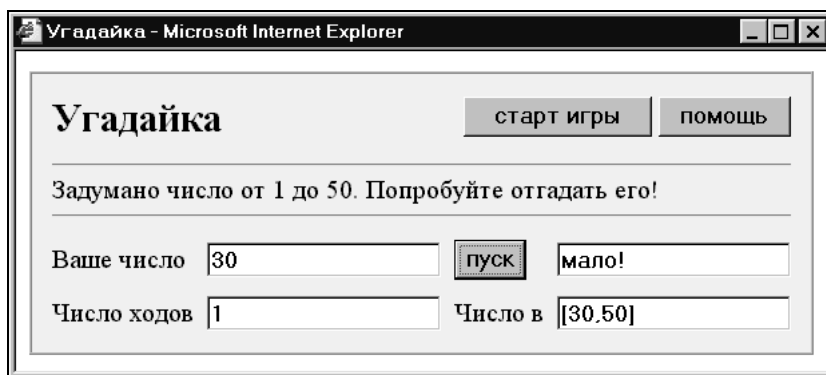


Рис. 12.10. Примерный вид приложения задания 4

Дополнительный материал

Два слоя обсуждения новых тегов

В этом и в следующих двух уроках новые теги вводятся с двух точек зрения. С одной стороны, показано, как тег записывается в HTML-коде, с другой —

рассказано, какой объект в объектной модели браузера соответствует тегу, какие свойства и методы имеет этот объект.

Первая точка зрения позволяет использовать тег для построения статичного HTML-документа, вторая — динамически менять документ, воздействуя на объектную модель при помощи набора скриптов.

Пример

Тег `<INPUT type=button value=пуск>` изображает на экране кнопку с надписью **пуск**.

Под воздействием щелчка мыши кнопка нажимается («вдавливается» в экран), но больше ничего не происходит. Если записать:

```
<FORM>
```

```
  <INPUT type=button value=пуск onclick="alert('Нажали кнопку')">
```

```
</FORM>
```

— то щелчок по кнопке выведет на экран сообщение «Нажали кнопку».

Здесь неявно использован объект, соответствующий тегу `<INPUT>`. Это неявное обращение происходит благодаря включению в состав тега специального атрибута — `onclick`. Значение этого атрибута — скрипт, который начинает работу при щелчке мыши по образу тега на экране.

Для явного обращения к объекту можно использовать имя, заданное в соответствующем теге атрибутом `name` — такое же имя получает и объект, построенный для этого тега.

В приведенном ниже примере при каждом щелчке надпись на кнопке меняется и принимает попеременно значения **пуск**, **выполнить**.

```
<SCRIPT language=JavaScript>
```

```
  var ind = 1;
```

```
  function Make()
```

```
  {
```

```
    var title = new Array("пуск", "выполнить");
```

```
    document.forma.knopka.value = title[ind];
```

```
    ind = 1 - ind;
```

```
  }
```

```
</SCRIPT>
```

```
<FORM name=forma>
```

```
  <INPUT name=knopka type=button value=пуск onclick="Make()">
```

```
</FORM>
```

Функция `Make()` явно обращается к объекту, который соответствует тегу `<INPUT>`:

`document` — доступ к объекту `document` (родитель — объект `window`);

<code>document.forma</code>	—	доступ к объекту <code>forma</code> (родитель — объект <code>document</code>);
<code>document.forma.knopka</code>	—	доступ к объекту <code>knopka</code> (родитель — объект <code>forma</code>);
<code>document.forma.knopka.value</code>	—	доступ к свойству <code>value</code> объекта <code>knopka</code> — соответствует атрибуту value тега <code><INPUT></code> с именем <code>knopka</code> .

Как видите, свойство `value` объекта, построенного браузером для тега `<INPUT>`, полностью соответствует одноименному атрибуту. Если изменить это свойство в объекте, надпись на кнопке тоже изменится, ведь браузер отслеживает соответствие объектной модели экранному образу документа.

Обратите внимание на то, что описание переменной `ind` расположено вне функции `Make()`.

Перенесем это описание внутрь:

```
function Make()
{
    var ind = 1;
    var title = new Array("пуск", "выполнить");
    document.forma.knopka.value = title[ind];
    ind = 1 - ind;
}
```

Теперь надпись на кнопке изменится один раз и больше меняться не будет. Ведь переменная, описанная внутри функции, локальна в ней, т. е., создается заново при каждом обращении к функции. Это означает, что при каждом входе в `Make()` переменная `ind` будет иметь значение 1, а команда смены значения `ind = 1 - ind` становится бессмысленной.

Оборонительная «Угадайка» на формах

Этот пример, описанный в уроке как задание 4, весьма поучителен. Он должен демонстрировать гипертекст, работающий как обычная компьютерная программа. Кроме того, пример хорошо иллюстрирует принцип ООП — управление логикой программы при помощи событий.

Код «Угадайки» может иметь следующий вид (листинг 12.3).

Листинг 12.3

```
<SCRIPT language=JavaScript>
<!--
...
// Нажата кнопка "помощь".
```

```

function help()
{
    ...
}
// Нажата кнопка "старт игры".
function newgame()
{
    ...
}
// Нажата кнопка "пуск"
function go()
{
    ...
}
// Проверка num на целое число из [a,b].
function IsNumAB(num, a, b)
{
    ...    return a <= parseInt(num,10) && parseInt(num,10) <= b;
}
//-->
</SCRIPT>

```

В этом коде нет привычной для классического программирования главной процедуры (функции), с которой программа начинает работу и которая содержит в себе иерархию вызовов всех остальных процедур и функций, задающих логику работу программы.

Все функции (кроме вспомогательной IsNumAB) совершенно независимы и не содержат вызовы друг друга. Все они являются обработчиками событий и автоматически вызываются браузером тогда, когда это событие происходит (табл. 12.4).

Таблица 12.4

Функция	Вызывается щелчком по тегу
help()	<code><INPUT type=button value=помощь onclick="help()"></code>
newgame()	<code><INPUT type=reset value="старт игры" onclick="newgame()"></code>
go()	<code><INPUT type=button value=пуск onclick="go()"></code>

Когда Иван Сидоров показал код «Угадайки» Мячикову, Петр удивился:

— Какое же это ООП? — сказал он. — Ведь в программе не создано ни одного объекта!

— Самое настоящее! — ответил Иван. — В «Угадайке» нет ни одного «самодельного» объекта, но в ней есть объекты, построенные браузером для тегов документа. «Угадайка» эти объекты нещадно использует!

Иван прав! Можно сказать больше: «Угадайка» не просто эксплуатирует объекты браузера, но пристраивает к ним свои собственные «ручки», перехватывая события и обрабатывая их собственными функциями.

Ниже показан возможный вариант функции `go()`:

```
// Нажата кнопка "пуск"
function go()
{
    // Проверка правильности ввода.
    if (IsNumAB(document.qw.num.value, a, b))
    {
        answer = parseInt(document.qw.num.value, 10);
        document.qw.num.value="";
        step++;
        // Анализ введенного пользователем числа.
        if (answer < number)
        {
            document.qw.result.value = "мало!";
            a = answer;
        }
        else if (answer > number)
        {
            document.qw.result.value = "много!";
            b = answer;
        }
        else
        {
            document.qw.result.value = "угадано!";
            a = b = answer;
        }
        document.qw.numstep.value = step;
        document.qw.interval.value = "["+a+", "+b+"]";
    }
}
```

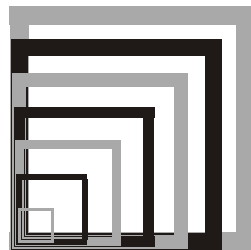
```
else document.qw.result.value = "ошибка ввода";  
}
```

Здесь:

<code>document.qw.num.value</code>	— обращения к свойству <code>value</code> объекта <code>num</code> (подобъект объекта <code>qw</code>);
<code>document.qw.result.value</code>	— обращения к свойству <code>value</code> объекта <code>result</code> (подобъект объекта <code>qw</code>);
<code>document.qw.numstep.value</code>	— обращения к свойству <code>value</code> объекта <code>numstep</code> (подобъект объекта <code>qw</code>);
<code>document.qw.interval.value</code>	— обращения к свойству <code>value</code> объекта <code>interval</code> (подобъект объекта <code>qw</code>).

Заметьте, что `qw` — это имя формы, в которую «погружена» «Угадайка», `num` — имя тега, задающего поле ввода числа, `result` — имя тега, задающего информационное сообщение, `numstep` — имя тега, задающего поле номера хода, `interval` — имя тега, задающего поле показа текущего интервала.

Урок 13



Формы (флажки, радиокнопки)

В этом уроке рассказано о двух интерфейсных элементах, которые часто используются на интерактивных страницах: флажки и радиокнопки.

Флажки

Флажки, или, как говорят, кнопки с независимой фиксацией — хорошо знакомый элемент компьютерного интерфейса (рис. 13.1).

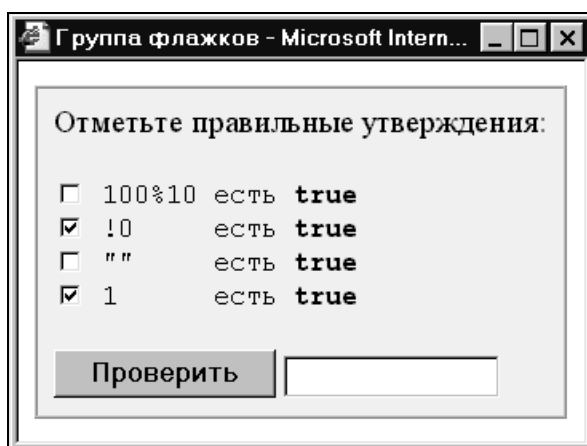


Рис. 13.1. Группа флажков на гипертекстовой странице

Флажки не зависят один от другого. Их можно устанавливать и сбрасывать в любой комбинации.

Флажок задается тегом `<INPUT type=checkbox>`.

Дополнительные атрибуты (табл. 13.1).

Таблица 13.1

Атрибут	Описание
name	Имя; используется в скриптах для доступа к объекту, созданному для тега <code><INPUT></code>
checked	Логический атрибут; для него не указывается значения. Присутствие атрибута означает, что флажок установлен по умолчанию

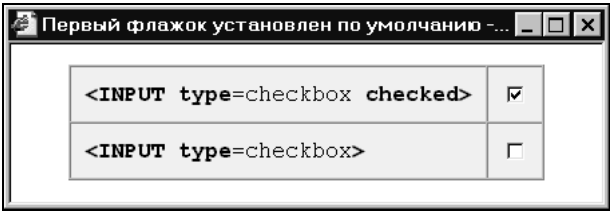


Рис. 13.2

Программное управление

Объект, построенный браузером для тега `<INPUT type=checkbox>`, имеет свойство `checked`. Это свойство принимает значение `true`, когда флажок установлен, и `false` — в противном случае.

Загрузите в браузер следующий код (листинг 13.1, вид его на странице демонстрирует рис. 13.2).

Листинг 13.1

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
  <TITLE>Программное управление флажком</TITLE>
</HEAD>
<BODY bgcolor=white text=black>
  <H2>Программное управление флажком</H2>
  <FORM>
    <INPUT type=checkbox name=flag>
    <INPUT type=button value=Проверить
      onclick="if(this.form.flag.checked)
        alert('Флажок установлен');
      else
```

```
        alert('Флажок не установлен');">
<INPUT type=button value=Изменить
        onclick="this.form.flag.checked = !this.form.flag.checked;">
</FORM>
</BODY>
</HTML>
```

Прodelайте следующие опыты.

1. Нажмите кнопку **Проверить**. Появится сообщение «Флажок не установлен». Флажок и в самом деле не установлен, ведь он задан кодом, в котором отсутствует атрибут **checked**.

```
<INPUT type=checkbox name=flag>
```

Свойство **checked** объекта, построенного для тега **<INPUT>**, имеет значение **false** как и выражение **this.form.flag.checked** в команде **if**.

2. Нажмите кнопку **Изменить**. Сработает код:

```
this.form.flag.checked = !this.form.flag.checked;
```

Свойство **checked** изменит свое значение с **false** на **true**. На экране в поле флажка появляется отметка (хотя мы ее там не ставили, а устроили ее появление на экране при помощи программного кода).

3. Нажмите кнопку **Проверить**. На экран выводится сообщение «Флажок установлен». Теперь выражение **this.form.flag.checked** имеет значение **true**.

Как видите, свойство **checked** можно не только читать, но и менять, устанавливая или сбрасывая флажок программным путем.

Обратите внимание на то, что свойство **checked** объекта имеет совсем другой смысл, чем одноименный атрибут тега **<INPUT>**. Атрибут указывает, что флажок установлен по умолчанию, а свойство хранит текущее состояние флажка, которое может меняться пользователем или программой.

Программы, которые «страдают склерозом», могут легко проверить, был ли установлен атрибут **checked** в теге **<INPUT>**, если обратятся к свойству **defaultChecked** соответствующего объекта.

Загрузите в браузер следующий код (листинг 13.2, вид его на странице демонстрирует рис. 13.3).

Листинг 13.2

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
```

```
<TITLE>checked и defaultChecked</TITLE>
</HEAD>
<BODY bgcolor=white text=black>
  <H2>checked и defaultChecked</H2>
  <FORM>
    <INPUT type=checkbox checked name=flag>
    <INPUT type=button value=checked
      onclick="alert(this.form.flag.checked);">
    <INPUT type=button value=defaultChecked
      onclick="alert(this.form.flag.defaultChecked);">
  </FORM>
</BODY>
</HTML>
```

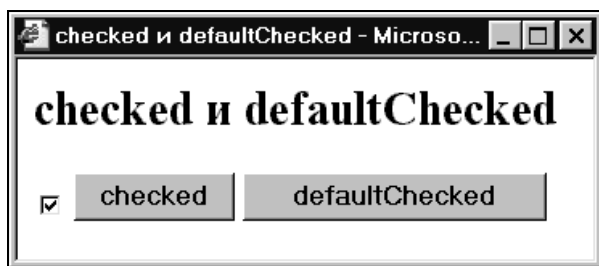


Рис. 13.3. Проверка свойств `checked` и `defaultChecked`

Проделайте следующие опыты:

1. Нажмите последовательно кнопки **checked** и **defaultChecked**. Получим сообщения «true» и «true». Это означает, что флажок установлен и отметка установлена по умолчанию.
2. Снимите отметку с флажка и снова нажмите кнопки **checked** и **defaultChecked**. Получим сообщения «false» и «true». Это означает, что флажок не установлен, но он был установлен по умолчанию.
3. Уберите из тега `<INPUT type=checkbox checked name=flag>` атрибут **checked** и снова нажмите кнопки **checked** и **defaultChecked**. Получим сообщения «false» и «false». Это означает, что флажок сейчас не установлен и не был установлен по умолчанию.
4. Отметьте флажок и нажмите кнопки **checked** и **defaultChecked**. Получим сообщения «true» и «false». Это означает, что флажок установлен, но он не был установлен по умолчанию.

Пример. Фрагмент зачетного класса.

Загрузите в браузер код (листинг 13.3, его вид на странице демонстрирует рис. 13.4).

Листинг 13.3

```
<HTML>

<HEAD>

  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">

  <TITLE>Фрагмент зачетного класса</TITLE>

</HEAD>

<BODY bgcolor=white text=black>

  <H2>Фрагмент зачетного класса</H2>

  <HR>

  <FORM>

    <INPUT type=reset value=Сброс><BR>

    Отметьте выражения со значением <TT><B>true</B></TT>:<BR>

    <INPUT type=checkbox name=task0><TT> !0 && !"" </TT><BR>

    <INPUT type=checkbox name=task1><TT> !0 || !"" </TT><BR>

    <INPUT type=checkbox name=task2><TT> 0 && !"" </TT><BR>

    <INPUT type=checkbox name=task3><TT> 0 || !"" </TT><BR>

    <INPUT type=button value=Проверить
      onclick="var ans='не верно';
        if(this.form.task0.checked &&
          this.form.task1.checked &&
            !this.form.task2.checked &&
              this.form.task3.checked
        ) ans='верно';
        this.form.result.value=ans;">

    <INPUT type=text size=15 name=result value="" readonly>

  </FORM>

</BODY>

</HTML>
```

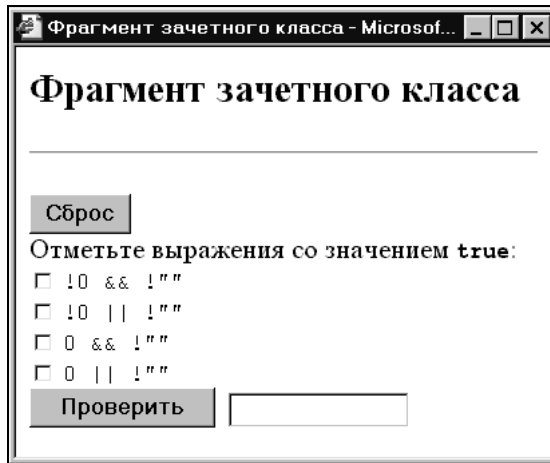


Рис. 13.4. Фрагмент зачетного класса

Поработайте с приложением и разберитесь в устройстве его кода.

В качестве значения атрибута **name** в тегах-флажках можно указывать одно и то же имя для флажков, входящих в одну группу. Доступ к отдельному флажку в группе выполняется так же, как к элементам массива при помощи индекса. Флажки в группе нумеруются в порядке их следования в HTML-коде, и нумерация начинается с нуля. Форму из предыдущего документа можно записать так:

```
<FORM>
  <INPUT type=reset value=Сброс><BR>
  Отметьте выражения со значением <TT><B>true</B></TT>:<BR>
  <INPUT type=checkbox name=task><TT> !0 && !\"'\" </TT><BR>
  <INPUT type=checkbox name=task><TT> !0 || !\"'\" </TT><BR>
  <INPUT type=checkbox name=task><TT> 0 && !\"'\" </TT><BR>
  <INPUT type=checkbox name=task><TT> 0 || !\"'\" </TT><BR>
  <INPUT type=button value=Проверить
    onclick="var ans='не верно';
      if(this.form.task[0].checked &&
        this.form.task[1].checked &&
        !this.form.task[2].checked &&
        this.form.task[3].checked
      ) ans='верно';
      this.form.result.value=ans;">
  <INPUT type=text size=15 name=result value="" readonly>
</FORM>
```

Радиокнопки

Радиокнопки (селекторные кнопки, переключатель) предназначены для выбора одного варианта из нескольких альтернативных. Подобно кнопочному переключателю диапазона волн в радиоприемнике, выбор одной кнопки автоматически снимает отметку с другой (рис. 13.5).

Радиокнопка задается тегом `<INPUT type=radio>`.

Дополнительные атрибуты следующие (табл. 13.2):

Таблица 13.2

Атрибут	Описание
name	Имя; используется в скриптах для доступа к объекту, созданному для тега <code><INPUT></code> . Кнопки, входящие в одну взаимосвязанную группу, должны иметь одинаковое имя. Именно по этому критерию (одинаковое имя) браузер распознает переключатель и организует снятие отметки с кнопки, когда выбрана другая в этой же группе. Доступ к отдельной кнопке в группе выполняется так же как к элементам массива — при помощи индекса. Кнопки в группе нумеруются в порядке их следования в HTML-коде, нумерация начинается с нуля
checked	Логический атрибут; для него не указывается значение. Присутствие атрибута означает, что отметка на кнопке установлена по умолчанию

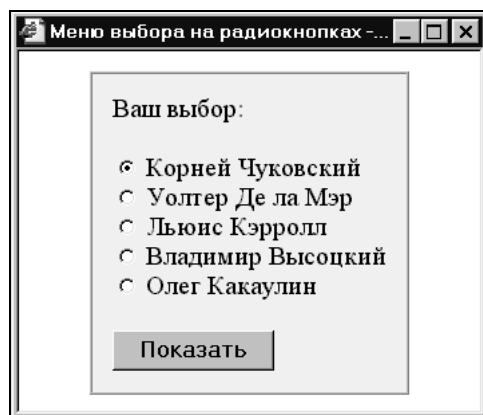


Рис. 13.5. Группа радиокнопок на гипертекстовой странице

Программное управление

Объект, построенный браузером для тега `<INPUT type=radio>`, имеет свойство `checked`. Это свойство принимает значение `true`, когда кнопка выбрана, и `false` — в противном случае.

Загрузите в браузер следующий код (листинг 13.4, вид его на странице демонстрирует рис. 13.6).

Листинг 13.4

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
  <TITLE>Программное управление радиокнопкой</TITLE>
</HEAD>
<BODY bgcolor=white text=black>
  <H2>Программное управление радиокнопкой</H2>
  <FORM>
    <INPUT type=radio name=tst>
    <INPUT type=button value=Проверить
      onclick="if(this.form.tst[0].checked)
        alert('Отметка на 1-ой кнопке установлена')
      else
        alert('Отметка на 1-ой кнопке не установлена');">
    <INPUT type=button value=Изменить
      onclick="var x = this.form.tst[0].checked;
        this.form.tst[0].checked=this.form.tst[1].checked;
        this.form.tst[1].checked=x;"><BR>
    <INPUT type=radio checked name=tst>
  </FORM>
</BODY>
</HTML>
```

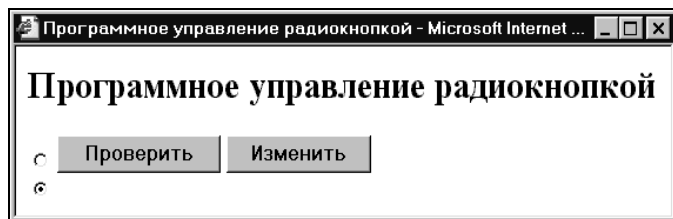


Рис. 13.6. Программное управление свойством `checked`

Прodelайте следующие опыты.

1. Нажмите кнопку **Проверить**. Появится сообщение «Отметка на 1-й кнопке не установлена». Радиокнопка и в самом деле выключена, ведь она задана кодом `<INPUT type=radio name=tst>`, в котором отсутствует атрибут `checked`. Это означает, что свойство `checked` объекта, построенного для этой радиокнопки, имеет значение `false`. , как и выражение `this.form.tst[0].checked` в команде `if`.

2. Нажмите кнопку **Изменить**. Выполнится код:

```
onclick="var x = this.form.tst[0].checked;  
        this.form.tst[0].checked=this.form.tst[1].checked;  
        this.form.test[1].checked=x;"
```

Этот код перебросит отметку со второй кнопки на первую.

4. Нажмите кнопку **Проверить**. Последует сообщение «Отметка на 1-й кнопке установлена». Это означает, что теперь выражение `this.form.tst[0].checked` имеет значение `true`.

Свойство `checked` объекта не совпадает с одноименным атрибутом тега `<INPUT>`. Атрибут `checked` устанавливает отметку на кнопке по умолчанию, а свойство `checked` отражает текущее состояние кнопки. Как и в случае с флажками, атрибуту `checked` соответствует свойство `defaultChecked`.

Изменяя активность одной кнопки переключателя, программист берет на себя ответственность правильно изменить свойство `checked` у всех его остальных кнопок (отмечена может быть только одна кнопка в группе).

Пример. Умножитель.

Умножитель, код которого представлен ниже, отображается на экране так, как показано на рис. 13.7.

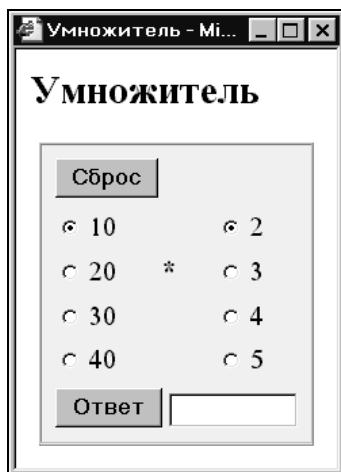


Рис. 13.7. Внешний вид Умножителя

Загрузите Умножитель в браузер, поработайте с ним и разберитесь в устройстве его кода (листинг 13.5).

Листинг 13.5

```
<HTML>

<HEAD>

  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">

  <TITLE>Умножитель</TITLE>

</HEAD>

<BODY bgcolor=white text=black>

  <H2>Умножитель</H2>

  <FORM>

    <TABLE border=1 bgcolor=#F0F0F0 cellpadding=5>
      <TR><TD>
        <TABLE border=0 cellpadding=5>
          <TR>
            <TD colspan=3><INPUT type=reset value=Сброс></TD>
          </TR>
          <TR>
            <TD><INPUT type=radio checked name=op1>10</TD>
            <TD>&nbsp;</TD>
            <TD><INPUT type=radio checked name=op2>2</TD>
          </TR>
          <TR>
            <TD><INPUT type=radio name=op1>20</TD>
            <TD><BIG>*</BIG></TD>
            <TD><INPUT type=radio name=op2>3</TD>
          </TR>
          <TR>
            <TD><INPUT type=radio name=op1>30</TD>
            <TD>&nbsp;</TD>
            <TD><INPUT type=radio name=op2>4</TD>
```

```

</TR>
<TR>
  <TD><INPUT type=radio name=op1>40</TD>
  <TD>&nbsp;</TD>
  <TD><INPUT type=radio name=op2>5</TD>
</TR>
<TR>
  <TD colspan=3>
    <INPUT type=button value=Ответ
      onclick="var op1 = new Array(10,20,30,40);
                var op2 = new Array(2,3,4,5);
                var len = 4;
                var ind1;
                var ind2;
                for(var i=len; --i>=0;)
                {
                  if(this.form.op1[i].checked) ind1=i;
                  if(this.form.op2[i].checked) ind2=i;
                }
                this.form.result.value= op1[ind1]*op2[ind2];">
    <INPUT type=text size=10 name=result value="" readonly>
  </TD>
</TR>
</TABLE>
</TD></TR>
</TABLE>
</BODY>
</HTML>

```

Зачетный класс

1. Форма на странице кодируется следующим образом:

```

<FORM name=forma>
  <INPUT type=reset value=Сброс><BR>

```

Отметьте коды, после выполнения которых, переменная у получит значение 1.

```

<INPUT type=checkbox name=ch> x=2; y = --x; <BR>
<INPUT type=checkbox name=ch> x=2; y = x--;<BR>
<INPUT type=checkbox name=ch> y=1; x = ++y;<BR>
<INPUT type=checkbox name=ch> y=1; x = y++;<BR>
<INPUT type=checkbox name=ch> y=2; x += y;<BR>
<INPUT type=button value=Проверить
      onclick='var ans="не верно";
               if (...) ans="верно";
               this.form.result.value=ans;'>
<INPUT type=text size=15 name=result value="" readonly>
</FORM>

```

Отметьте правильные условия, которые могут заместить многоточие в коде формы:

- а) `document.forma.ch[2].checked;`
- б) `document.forma.ch[0].checked;`
- в) `document.forma.ch[3].checked;`
- г) `document.forma.ch[4].checked;`
- д) `this.form.ch[0].checked;`
- е) `this.form.ch[2].checked && this.form.ch[3].checked;`
- ж) `this.form.ch[0].checked && this.form.ch[1].checked;`
- з) `this.form.ch[0].checked && this.form.ch[3].checked;`
- и) `this.form.ch[1].checked;`
- к) все условия неверные.

2. Форма на странице кодируется следующим образом:

```

<FORM name=forma>

  Заданы переменные:

<PRE>

var set = new Array(1,2,3,4,5,6,7,8,9,10);
var s = 0;
var i;
</PRE>

```

Отметьте правильный код (он один), вычисляющий сумму элементов массива:

```

<PRE>

```

```
<INPUT type=reset value=Сброс>
<INPUT type=radio name=r checked> for(;;) s+=set[i--];
<INPUT type=radio name=r> for(i=1;i<set.length;i++) s+=set[i];
<INPUT type=radio name=r> for(i=set.length;i;i--) s+=set[i];
<INPUT type=radio name=r> for(i=set.length-1;i;i--) s+=set[i];
<INPUT type=radio name=r> for(i=set.length; --i>=0;) s+=set[i];
<INPUT type=radio name=r> for(i=set.length;--i;) s+=set[i];
<INPUT type=radio name=r> for(i=set.length;i-->=0;) s+=set[i];
<INPUT type=button value=Проверить
    onclick='var answer="не верно";
              if(...) answer="верно";
              this.form.result.value=answer;'>
<INPUT type=text size=15 name=result value="" readonly>
</PRE>
</FORM>
```

Отметьте правильные условия, которые могут заместить многоточие в коде формы:

- а) `document.forma.r[2].checked;`
- б) `document.forma.r[0].checked;`
- в) `document.forma.r[3].checked;`
- г) `document.forma.r[4].checked;`
- д) `this.form.r[5].checked;`
- е) `this.form.r[4].checked;`
- ж) `this.forma.r[1].checked;`
- з) `this.forma.r[4].checked;`
- и) `this.form.r[1].checked;`
- к) все условия неверные.

Задания

1. Создайте документ с формой из первого задания зачетного класса.
2. Создайте документ с формой из второго задания зачетного класса.
3. Создайте документ с формой, изображенной на рис. 13.8.

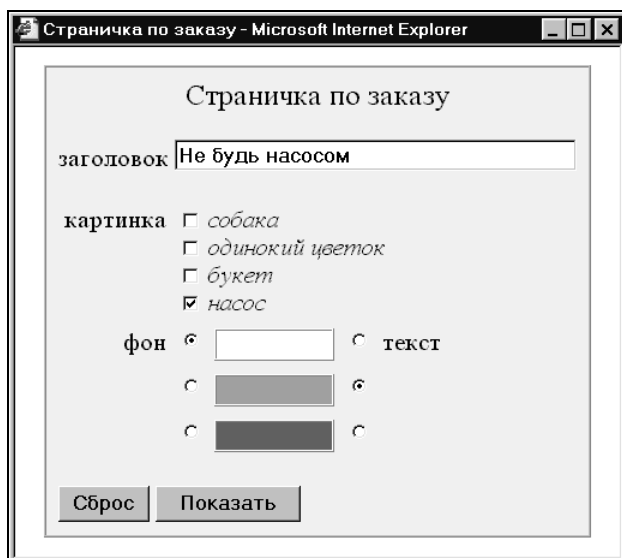


Рис. 13.8. Страничка по заказу

По нажатию на кнопку **Показать** приложение должно открыть новое окно и показать в нем заказанные картинки с короткими подписями. Новое окно должно создаваться «на лету» с использованием информации, которую ввел в форму пользователь.

Пример возможной страницы показан на рис. 13.9.

4. Создайте электронный магазин «Красное эхо» (рис. 13.10).



Рис. 13.9. Не будь насосом

Название	Описание	Количество	Цена
☐ Дядя Ваня	Биокремниевый говорящий бытовой робот (уборка, кухня, уход за детьми и домашними животными).	1	1500
☐ Курочка Ряба	Для производства золотых яиц (одно золотое яйцо в сутки).	1	800

Сумма заказа Сброс

Рис. 13.10. Электронный магазин «Красное эхо»

Дополнительный материал

Программирование магазина «Красное эхо» — достаточно полезное упражнение. Оно интересно своей простотой и правдоподобностью. Ведь электронные магазины давно уже стали привычными.

Если читатель не имеет опыта покупок в сетевых магазинах, полезно совершить экскурсию, например, в книжный магазин piter.com (рис. 13.11). Познакомьтесь с реальным интерфейсом магазина и его внешним видом. После такой экскурсии задание 4 станет более понятным.

Устройство интернет-магазина

Как правило, интернет-магазин (как программа) имеет две части: клиентскую и серверную. Клиентская часть обслуживается скриптом, вмонтированным в код страницы, которую просматривает покупатель. Пользователь листает каталоги-меню, отмечая товары, затем оформляет заказ. Заказ посылается на сервер и там обрабатывается CGI-программой — серверной частью сетевого магазина.

На рис. 13.11 приводится внешний вид книжного интернет-магазина piter.com:

Что такое CGI-скрипты

CGI-скриптом называют программу, работающую на сервере (на котором лежит www-страничка). Эта программа получает с машины клиента (той машины, с которой смотрят www-страничку) данные. Данные посылаются, например, из формы по нажатию на кнопку **Обработать заказ**

(<INPUT type=submit>). Данные поступают на сервер в программу CGI, обрабатываются и результат посылается клиенту. Гипертекстовая страничка на машине клиента выводит на страницу надпись «Заказ обработан и принят к исполнению. Ждите книги почтой через неделю».



Рис. 13.11. Книжный магазин piter.com

Обработка данных заключается в проверке корректности, занесении их в общую базу данных, отправлении сообщений бухгалтеру (принять деньги) и указаний оператору (завернуть товар в плотную бумагу и отнести на почту).

CGI расшифровывается как Common Gateway Interface (Общий шлюзовый интерфейс), т. е. под сокращением CGI понимают описание структуры информации, ее вид, в котором она следует от клиента к серверу и обратно.

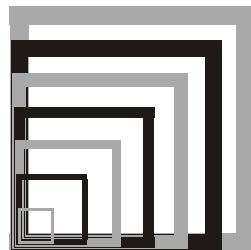
Невозможно устроить настоящий интернет-магазин средствами только скрипта, вмонтированного в гипертекстовую страницу.

Страничку (ее копии) могут одновременно просматривать сотни клиентов. Значит, на них запускаются и работают сотни копий скриптов, ничего не зная друг о друге! А ведь кладовка с книгами — одна!

Вот почему, в этой и подобных ситуациях применяют CGI-программу! Это программа одна на все клиентские копии гипертекстовой странички, с которых поступают заказы на покупки.

CGI-программа может быть написана на любом языке программирования. Это может быть Perl, Си, Паскаль, Java, Бейсик.

Урок 14



Формы (меню, многострочное поле)

Интерфейсные элементы — меню и многострочное поле — задаются внутри формы при помощи блоков `<SELECT>...</SELECT>` и `<TEXTAREA>...</TEXTAREA>`.

Меню `<SELECT>`

Блок `<SELECT>...</SELECT>` вставляет в форму меню или, как говорят, раскрывающийся список (рис. 14.1, 14.2).

Меню чаще всего используют в режиме, когда из нескольких предлагаемых вариантов можно выбрать только один, и в этом случае меню напоминает переключатель с радиокнопками.

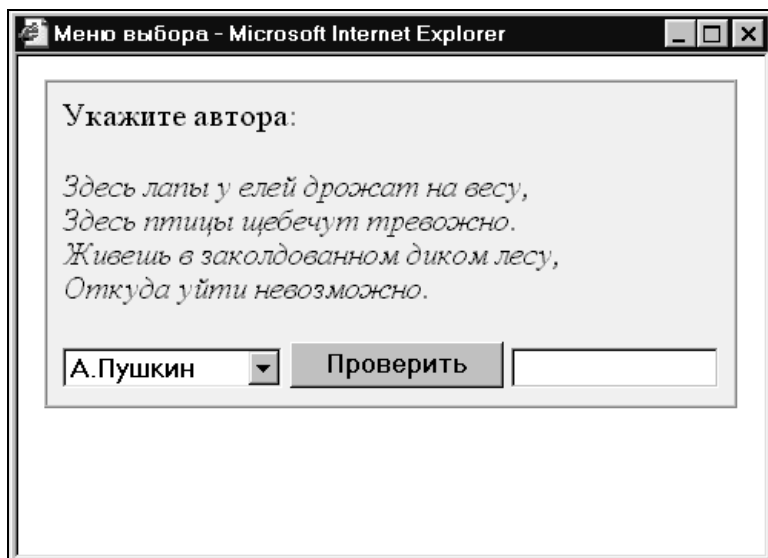


Рис. 14.1. Раскрывающийся список на гипертекстовой странице

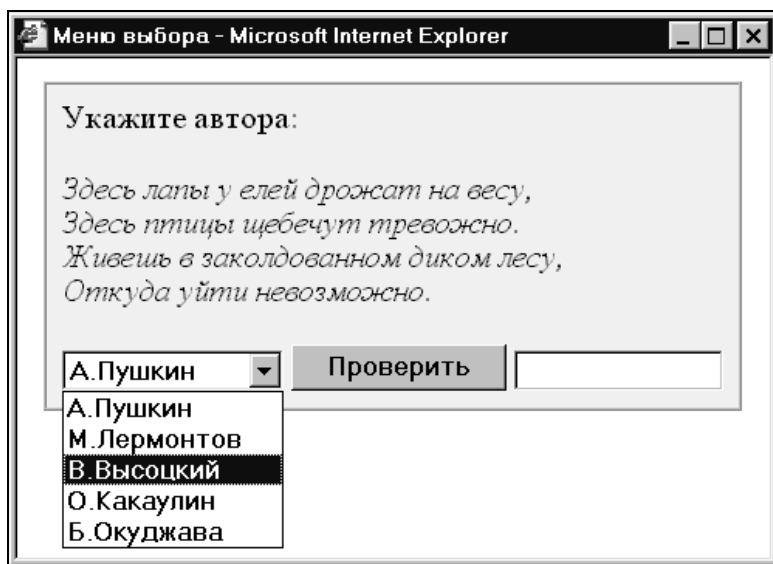


Рис. 14.2. Выбор строки в раскрывающемся списке

Меню имеет следующий формат:

```
<SELECT атрибуты>  
  <OPTION атрибуты>строка меню</OPTION>  
  ...  
  <OPTION атрибуты> строка меню</OPTION>  
</SELECT>
```

Внутри блока `<SELECT>...</SELECT>` располагаются команды `<OPTION>...</OPTION>`, каждая из которых создает одну запись — позицию меню.

Меню на рис. 14.1, 14.2 соответствует такой код (листинг 14.1):

Листинг 14.1

```
<SELECT>  
  <OPTION selected>А.Пушкин</OPTION>  
  <OPTION>М.Лермонтов</OPTION>  
  <OPTION>В.Высоцкий</OPTION>  
  <OPTION>О.Какаулин</OPTION>  
  <OPTION>Б.Окуджава</OPTION>  
</SELECT>
```

Атрибуты тега **<SELECT>**

Атрибуты тега **<SELECT>** приводятся в табл. 14.1

Таблица 14.1

Атрибут	Описание
name	Имя; используется в скриптах для доступа к объекту, созданному для тега <SELECT>
size	Число видимых строк меню. Если атрибут не задан, то его значение считается равным 1
multiple	Логический атрибут. Если он присутствует, то в меню можно выбрать не один, а несколько пунктов, и меню становится похожим на группу флажков. Этот атрибут в книге использоваться не будет

Атрибуты тега **<OPTION>**

Атрибуты тега **<OPTION>** приводятся в табл. 14.2.

Таблица 14.2

Атрибут	Описание
selected	Логический атрибут. Присутствие атрибута означает, что данная позиция меню выбрана по умолчанию

Программное управление

Объект, построенный браузером для блока **<SELECT>...</SELECT>**, имеет следующие свойства (табл. 14.3).

Таблица 14.3

Свойство	Описание
options	Массив объектов, каждый из которых описывает одну строку меню. Элементы массива, как всегда, нумеруются с 0. Пусть в форме с именем forma описан тег <SELECT name=menu> . Тогда запись document.forma.menu.options[0] указывает на первый элемент массива options . Запись document.forma.menu.options[1] — на второй и т. д.

Таблица 14.3 (окончание)

Свойство	Описание
options	Каждый элемент массива <code>options</code> является объектом и имеет следующие свойства: ☐ <code>text</code> Содержит текст соответствующей строки меню. Для приведенного выше примера прочитать первую строку меню можно при помощи конструкции: <pre>i0=document.forma.menu.options[0].text;</pre> Переменная <code>i0</code> получит значение "А.Пушкин". ☐ <code>selected</code> Равно <code>true</code>, если строка выбрана, и <code>false</code> — в противном случае. ☐ <code>Default-Selected</code> Равно <code>true</code>, если в соответствующем теге <code><OPTION></code> установлен атрибут <code>selected</code>, и <code>false</code> — в противном случае.
length	Число элементов массива <code>options</code> . Понятно, что значение этой переменной совпадает с числом строк в меню. Для доступа к этому свойству в форме с именем <code>forma</code> можно использовать запись <code>document.forma.menu.length</code> или внутри формы — запись <code>this.form.menu.length</code> (здесь <code>menu</code> — это имя тега <code><SELECT></code>)
Selected-Index	Номер выбранной строки меню. Это число совпадает с индексом того элемента массива <code>options</code> , который имеет для свойства <code>selected</code> значение <code>true</code> . Это очень полезное свойство и оно часто используется в скриптах

Чтобы читатель не запутался в перечисленных свойствах, ниже приводится краткая справочная схема. Примеры обращения к свойствам относятся к случаю, когда тег `<SELECT>` имеет имя `menu` и расположен внутри формы с именем `forma` (табл. 14.4).

Таблица 14.4

Свойства объекта	Пример доступа к свойствам
length	<code>document.forma.menu.length</code>
selectedIndex	<code>document.forma.menu.selectedIndex</code>

Таблица 14.4

Свойства объекта	Пример доступа к свойствам
options	document.forma.menu.options
text	document.forma.menu.options[0].text
selected	document.forma.menu.options[0].selected
defaultSelected	document.forma.menu.options[0].defaultSelected

Пример

На рис. 14.3 показан фрагмент зачетного класса.

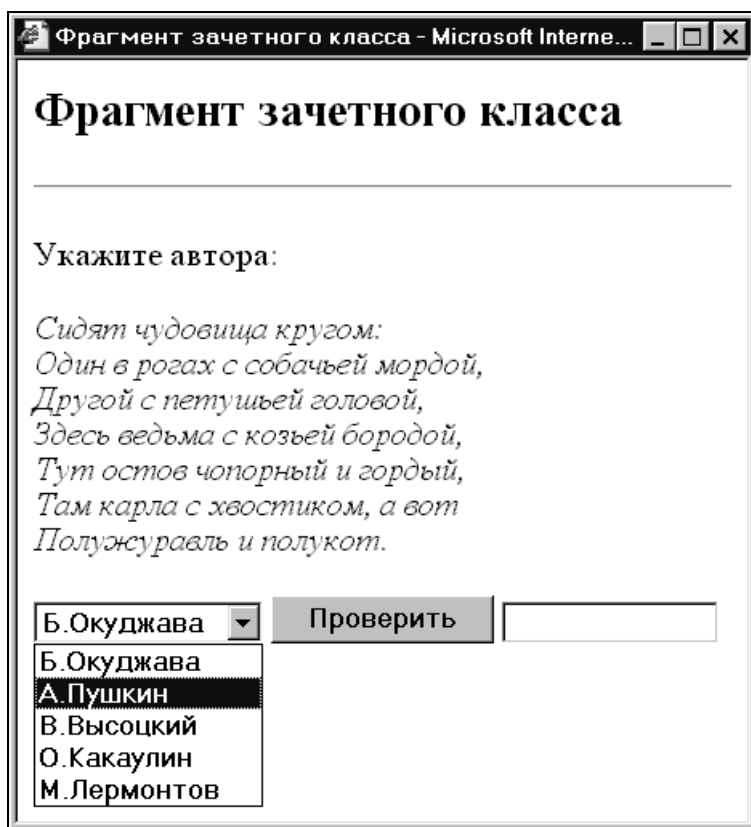


Рис. 14.3. Фрагмент зачетного класса

Загрузите код этого приложения (листинг 14.2) в браузер, поработайте с ним и разберитесь в его устройстве.

Листинг 14.2

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
  <TITLE>Фрагмент зачетного класса</TITLE>
</HEAD>
<BODY bgcolor=white text=black>
  <H2>Фрагмент зачетного класса</H2>
  <HR>
  <FORM>
    <P>Укажите автора:
    <P><CITE>
      Сидят чудовища кругом:<BR>
      Один в рогах с собачьей мордой,<BR>
      Другой с петушьей головой,<BR>
      Здесь ведьма с козьей бородой,<BR>
      Тут остов чопорный и гордый,<BR>
      Там карла с хвостиком, а вот<BR>
      Полужуравль и полукот.</CITE>
    <P>
    <SELECT name=menu>
      <OPTION selected>Б.Окуджава</OPTION>
      <OPTION>А.Пушкин</OPTION>
      <OPTION>В.Высоцкий</OPTION>
      <OPTION>О.Какаулин</OPTION>
      <OPTION>М.Лермонтов</OPTION>
    </SELECT>
    >&nbsp;<INPUT type=button value=Проверить
      onclick="this.form.result.value=
        (this.form.menu.selectedIndex==1)?'верно':'не верно'"
    >&nbsp;<INPUT type=text size=15 name=result value="" readonly>
  </FORM>
</BODY>
</HTML>
```

Многострочное поле <TEXTAREA>

В HTML-форме можно поместить многострочное поле ввода, которое представляет собой простой текстовый редактор.

Блок <TEXTAREA>...</TEXTAREA> имеет следующий формат:

```
<TEXTAREA атрибуты>
```

```
текст
```

```
</TEXTAREA>
```

Текст, записанный внутри блока, будет выводиться в многострочное поле вывода. На экране этот текст представляется точно в таком виде, в котором он записан в HTML-коде. Если текст содержит HTML-теги, то они выводятся на экран как обычные символы, лишаясь своего специального смысла.

Атрибуты тега <TEXTAREA>

Атрибуты тега <TEXTAREA> приводятся в табл. 14.5

Таблица 14.5

Атрибут	Описание
name	Имя; используется в скриптах для доступа к объекту, созданному для тега <TEXTAREA>
cols	Задаёт число столбцов текстового поля
rows	Задаёт число строк текстового поля
readonly	Определяет поле «только для чтения». Такое поле пользователь не может редактировать

Программное управление

Объект, построенный браузером для блока <TEXTAREA>...</TEXTAREA>, имеет следующие свойства (табл. 14.6).

Таблица 14.6

Свойство	Описание
value	Значением является текущее содержимое редактора. Содержимое окна размещается в переменной <i>value</i> как одна текстовая строка; концы экранных строк представляются двумя символами «\r\n» (возврат каретки к началу строки и переход на новую строку)

Таблица 14.6 (окончание)

Свойство	Описание
defaultValue	Значением является содержимое редактора по умолчанию, т. е. тот текст, который записан в HTML-коде между тегами <code><TEXTAREA>...</TEXTAREA></code>

На рис. 14.4 показан внешний вид испытательного стенда, код которого приводится ниже (листинг 14.3).

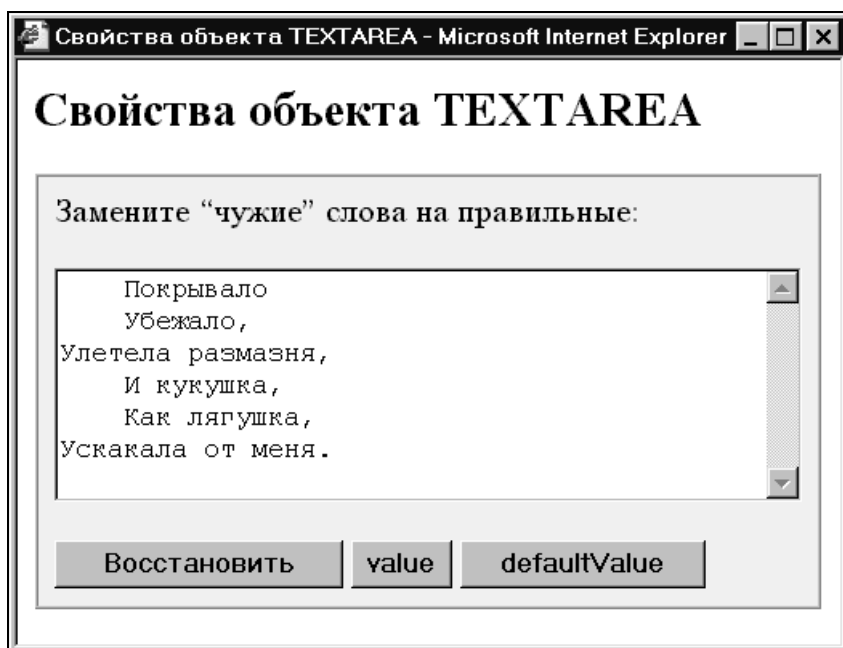


Рис. 14.4. Испытательный стенд TEXTAREA

Листинг 14.3

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
  <TITLE>Свойства объекта TEXTAREA</TITLE>
</HEAD>
<BODY bgcolor=white text=black>
```

```

<H2>Свойства объекта TEXTAREA</H2>
<FORM>
<TABLE bgcolor=#F0F0F0 border=1 cellspacing=0 cellpadding=10>
  <TR>
    <TD>
      <P>Замените &#147;чужие&#148; слова на правильные:
      <P>
      <TEXTAREA rows=7 cols=44 name=edit>
Покрывало
Убежало,
Улетела размазня,
И кукушка,
Как лягушка,
Ускакала от меня.
    </TEXTAREA>
    <P>
    <NOBR>
      <INPUT type=reset value=Восстановить>
      <INPUT type=button value=value
        onclick="alert(this.form.edit.value)">
      <INPUT type=button value=defaultValue
        onclick="alert(this.form.edit.defaultValue)">
    </NOBR>
    </TD>
  </TR>
</TABLE>
</FORM>
</BODY>
</HTML>

```

Поработайте на этом испытателе:

1. Последовательно нажмите кнопки `value` и `defaultValue` и убедитесь, что значения этих свойств совпадают и соответствуют содержимому многострочного поля.
2. Измените текст в многострочном поле. Теперь значения свойств `value` и `defaultValue` различаются: первое содержит текущее содержимое, отображаемое на экране, второе — текст, который задан в HTML-коде.

Содержимое поля можно не только читать, но и менять внутри программы-скрипта. На рис. 14.5, 14.6 показаны примеры такого приложения.

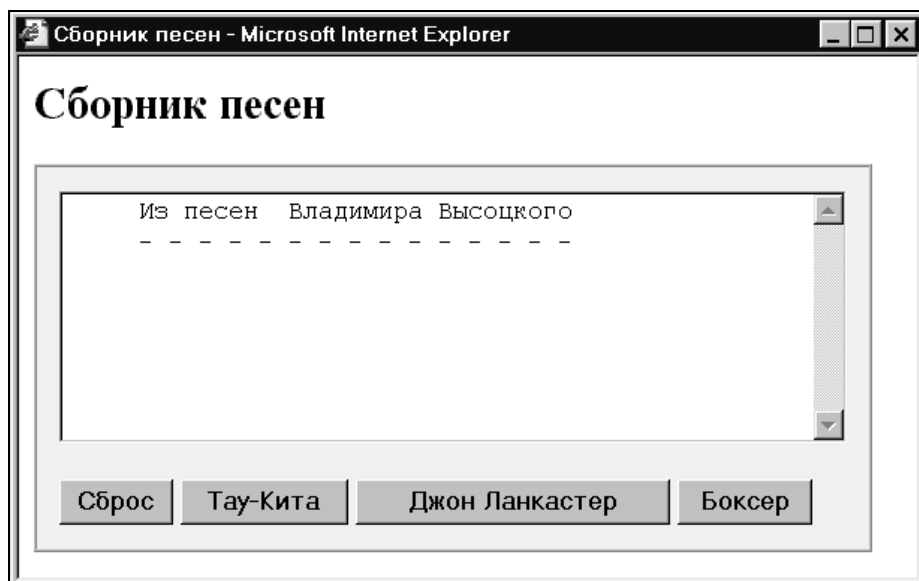


Рис. 14.5. Сборник песен. Вид приложения после загрузки

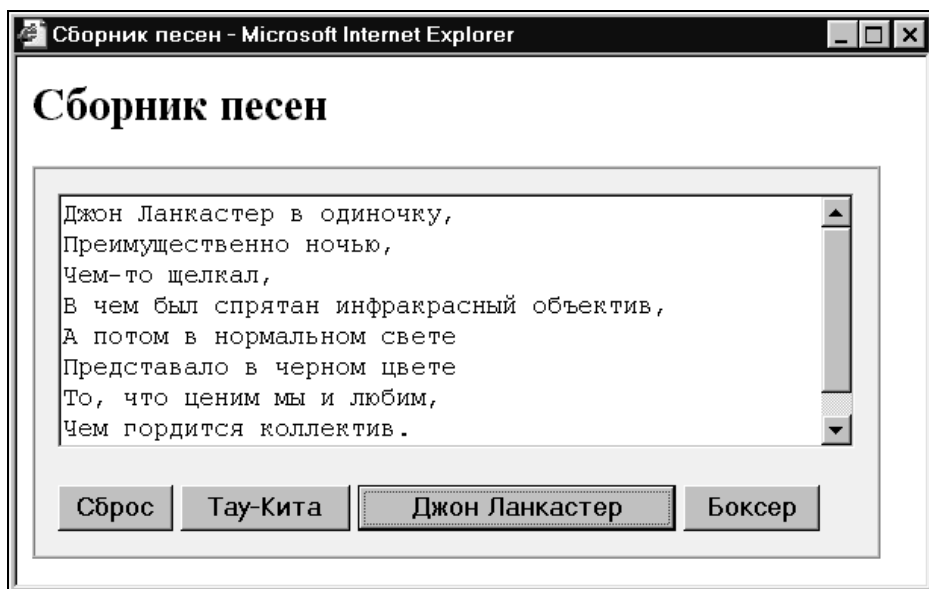


Рис. 14.6. Сборник песен. Выбран текст «Джон Ланкастер»

Загрузите в браузер код этого приложения (листинг 14.4) и поработайте с ним.

Листинг 14.4

```

<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
  <TITLE>Сборник песен</TITLE>
  <SCRIPT language=JavaScript>
    <!--
var text1 = "В далеком созвездии Тау-Кита\r\n"+
  "Все стало для нас непонятно.\r\n"+
  'Сигнал посылаем: "Вы что это там?"\r\n'+
  "А нас посылают обратно.\r\n";
var text2 = "Джон Ланкастер в одиночку,\r\n"+
  "Преимущественно ночью,\r\n"+
  "Чем-то щелкал,\r\n"+
  "В чем был спрятан инфракрасный объектив,\r\n"+
  "А потом в нормальном свете\r\n"+
  "Представало в черном цвете\r\n"+
  "То, что ценим мы и любим,\r\n"+
  "Чем гордится коллектив.\r\n";
var text3 = "Удар, удар, еще удар,\r\n"+
  "Опять удар и вот\r\n"+
  "Борис Буткеев (Краснодар)\r\n"+
  "Проводит апперкот.\r\n";

    //-->
  </SCRIPT>
</HEAD>
<BODY bgcolor=white text=black>
  <H2>Сборник песен</H2>
  <FORM>
    <TABLE bgcolor=#C0C0B0 border=1 cellpadding=10>
      <TR>
        <TD>
          <TEXTAREA rows=8 cols=50 name=edit>
            Из песен Владимира Высоцкого
            - - - - -
          </TEXTAREA>

```

```
<P>
<NOBR>
  <INPUT type=reset value=Сброс>
  <INPUT type=button value="Тай-Кита"
    onclick="this.form.edit.value=text1;">
  <INPUT type=button value="Джон Ланкастер"
    onclick="this.form.edit.value=text2;">
  <INPUT type=button value="Боксер"
    onclick="this.form.edit.value=text3;">
</NOBR>
</TD>
</TR>
</TABLE>
</FORM>
</BODY>
</HTML>
```

Обратите внимание на то, что концы текстовых строк, предназначенных для вывода в окошко редактора `<TEXTAREA>`, необходимо явно помечать символами «`\r\n`».

Зачетный класс

1. Приложение имеет вид, показанный на рис. 14.7.

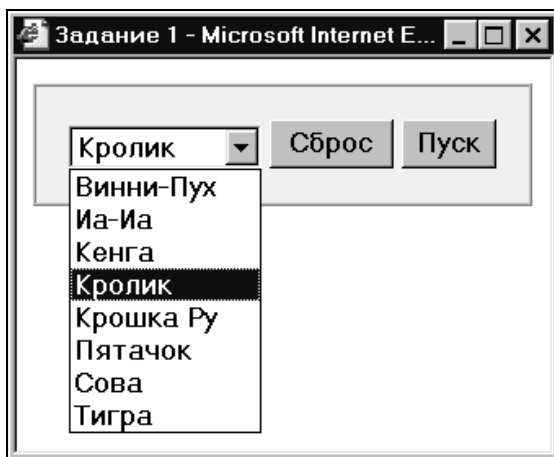


Рис. 14.7. Вид приложения к заданию 1

Кнопка **Пуск** выводит на экран окно alert с текстом, который выбран в окошке в меню. Кнопка **Сброс** устанавливает меню в положение «Кролик».

Какая из приведенных ниже программ соответствует этому интерфейсу?

а)

```
<SELECT name=forma>
  <OPTION>Винни-Пух</OPTION>
  <OPTION>Иа-Иа</OPTION>
  <OPTION>Кенга</OPTION>
  <OPTION selected>Кролик</OPTION>
  <OPTION>Крошка Ру</OPTION>
  <OPTION>Пятачок</OPTION>
  <OPTION>Сова</OPTION>
  <OPTION>Тигра</OPTION>
</SELECT>
<INPUT type=reset value=Сброс>
<INPUT type=button value=Пуск
  onclick= "var ind=this.form.menu.selectedIndex;
            alert (this.form.menu.options[ind].text)">
```

б)

```
<SELECT name=menu>
  <OPTION>Винни-Пух</OPTION>
  <OPTION>Иа-Иа</OPTION>
  <OPTION>Кенга</OPTION>
  <OPTION selected>Кролик</OPTION>
  <OPTION>Крошка Ру</OPTION>
  <OPTION>Пятачок</OPTION>
  <OPTION>Сова</OPTION>
  <OPTION>Тигра</OPTION>
</SELECT>
<INPUT type=reset value=Сброс>
<INPUT type=button value=Пуск
  onclick= "var ind=this.form.menu.selectedIndex;
            alert (this.form.menu.options[ind].text)">
```

в)

```
<SELECT name=menu>
  <OPTION>Винни-Пух</OPTION>
  <OPTION>Иа-Иа</OPTION>
  <OPTION>Кенга</OPTION>
  <OPTION selected>Кролик</OPTION>
  <OPTION>Крошка Ру</OPTION>
```

```

<OPTION>Пятачок</OPTION>
<OPTION>Сова</OPTION>
<OPTION>Тигра</OPTION>
</SELECT>
<INPUT type=reset value=Сброс>
<INPUT type=button value=Пуск
      onclick= "var ind=this.form.menu.selected;
                alert (this.form.menu.options[ind].text)">
г)
<SELECT name=menu>
  <OPTION selected>Винни-Пух</OPTION>
  <OPTION>Иа-Иа</OPTION>
  <OPTION>Кенга</OPTION>
  <OPTION>Кролик</OPTION>
  <OPTION>Крошка Ру</OPTION>
  <OPTION>Пятачок</OPTION>
  <OPTION>Сова</OPTION>
  <OPTION>Тигра</OPTION>
</SELECT>
<INPUT type=reset value=Сброс>
<INPUT type=button value=Пуск
      onclick= "var ind=this.form.menu.selectedIndex;
                alert (this.form.menu.options[ind].text)">

```

2. Приложение имеет вид, показанный на рис. 14.8.

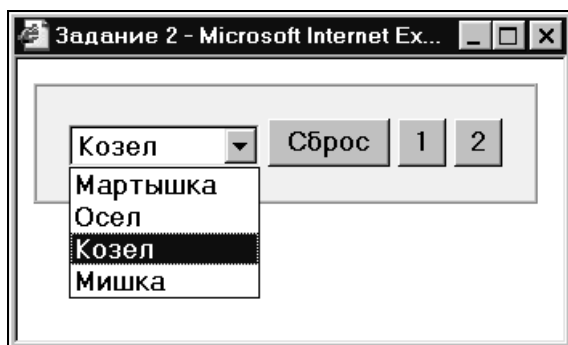


Рис. 14.8. Вид приложения к заданию 2

Кнопка **Сброс** устанавливает меню в положение «Козел». Кнопки **1** и **2** показывают сообщения, которые зависят от выбора строки в меню (табл. 14.7).

Таблица 14.7

Что выбрано в меню	Сообщение на кнопке «1»	Сообщение на кнопке «2»
Мартышка	0	false
Осел	1	false
Козел	2	true
Мишка	3	false

Какая из приведенных ниже программ соответствует этому интерфейсу?

а)

```
<SELECT name=m>
  <OPTION>Мартышка</OPTION>
  <OPTION>Осел</OPTION>
  <OPTION>Козел</OPTION>
  <OPTION selected >Мишка</OPTION>
</SELECT>
<INPUT type=reset value=Сброс>
<INPUT type=button value=1 onclick="alert(this.form.m.selected)">
<INPUT type=button value=2
  onclick="var ind=this.form.m.selected;
            alert(this.form.m.options[ind].defaultSelected)">
```

б)

```
<SELECT name=m>
  <OPTION>Мартышка</OPTION>
  <OPTION>Осел</OPTION>
  <OPTION selected>Козел</OPTION>
  <OPTION>Мишка</OPTION>
</SELECT>
<INPUT type=reset value=Сброс>
<INPUT type=button value=1 onclick="alert(this.form.m.selected)">
<INPUT type=button value=2
  onclick="var ind=this.form.m.selected;
            alert(this.form.m.options[ind].defaultSelected)">
```

в)

```
<SELECT name=m>
  <OPTION>Мартышка</OPTION>
  <OPTION>Осел</OPTION>
  <OPTION selected>Козел</OPTION>
  <OPTION>Мишка</OPTION>
</SELECT>
```

```
<INPUT type=reset value=Сброс>
<INPUT type=button value=1 onclick="alert(this.form.m.selectedIndex)">
<INPUT type=button value=2
      onclick="var ind=this.form.m.selected;
                alert(this.form.m.options[ind].defaultSelected)">
  г)
<SELECT name=m>
  <OPTION>Мартышка</OPTION>
  <OPTION>Осел</OPTION>
  <OPTION selected>Козел</OPTION>
  <OPTION>Мишка</OPTION>
</SELECT>
<INPUT type=reset value=Сброс>
<INPUT type=button value=1 onclick="alert(this.form.m.selectedIndex)">
<INPUT type=button value=2
      onclick="var ind=this.form.m.selectedIndex;
                alert(this.form.m.options[ind].selected)">
```

3. Приложение имеет вид, показанный на рис. 14.9.

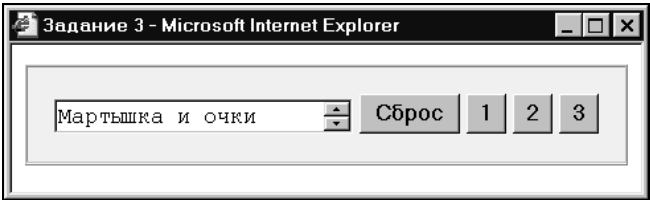


Рис. 14.9. Вид приложения к заданию 3

Сообщение в поле меняется в зависимости от того, какая кнопка нажата (табл. 14.8).

Таблица 14.8

Какая кнопка нажата	Сообщение
Кнопка 1	Волк на псарне
Кнопка 2	Пловец и море
Кнопка 3	Совет Мышей
Кнопка Сброс	Мартышка и очки

Какая из приведенных ниже программ соответствует этому интерфейсу?

а)

```
<TEXTAREA rows=1 cols=20 readonly name=edit>
Мартышка и очки
</TEXTAREA>
```

```

<INPUT type=reset value=Сброс>
<INPUT type=button value=1
    onclick="this.form.edit.value='Совет мышей\r\n'">
<INPUT type=button value=2
    onclick="this.form.edit.value='Пловец и море\r\n'">
<INPUT type=button value=3
    onclick="this.form.edit.value='Волк на псарне \r\n'">

```

б)

```

<TEXTAREA rows=1 cols=20 readonly name=edit>
Мартышка и очки
</TEXTAREA>
<INPUT type=reset value=Сброс>
<INPUT type=button value=1
    onclick="this.form.edit.value='Волк на псарне\r\n'">
<INPUT type=button value=2
    onclick="this.form.edit.value='Пловец и море\r\n'">
<INPUT type=button value=3
    onclick="this.form.edit.value='Совет Мышей\r\n'">

```

в)

```

<TEXTAREA rows=1 cols=20 readonly name=edit>
Мартышка и очки
</TEXTAREA>
<INPUT type=reset value=Сброс>
<INPUT type=button value=1
    onclick="this.edit.value='Волк на псарне\r\n'">
<INPUT type=button value=2
    onclick="this.edit.value='Пловец и море\r\n'">
<INPUT type=button value=3
    onclick="this.edit.value='Совет Мышей\r\n'">

```

г)

```

<TEXTAREA rows=20 cols=1 readonly name=edit>
Мартышка и очки
</TEXTAREA>
<INPUT type=reset value=Сброс>
<INPUT type=button value=1
    onclick="this.form.edit.value='Волк на псарне\r\n'">
<INPUT type=button value=2
    onclick="this.form.edit.value='Пловец и море\r\n'">
<INPUT type=button value=3
    onclick="this.form.edit.value='Совет Мышей\r\n'">

```

Задания

1. Создайте меню выбора на радиокнопках, имеющее интерфейс, показанный на рис. 14.10.

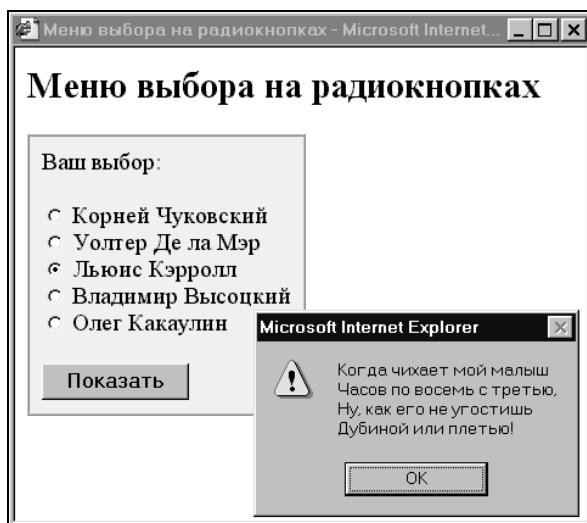


Рис. 14.10. Вид приложения к заданию 1

2. Создайте меню выбора на радиокнопках, имеющее интерфейс, показанный на рис. 14.11.

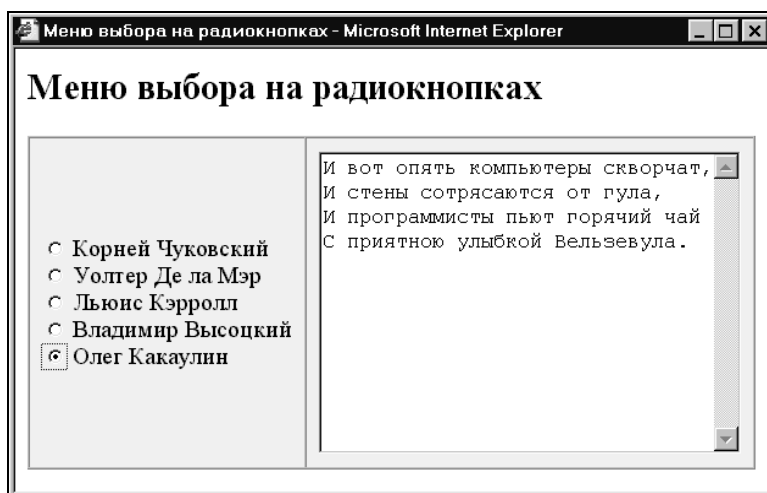


Рис. 14.11. Вид приложения к заданию 2

3. Создайте меню выбора на теге `<SELECT>`, имеющее интерфейс, показанный на рис. 14.12.



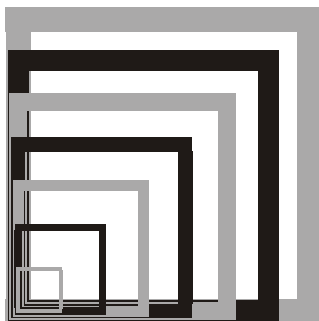
Рис. 14.12. Вид приложения к заданию 3

4. Электронный контролер. Разработайте приложение, которое можно будет использовать для проведения диктантов и контрольных работ.

Диктант. Ученик записывает текст под диктовку в экранный редактор, затем нажимает кнопку **Проверить**. Приложение сравнивает написанный текст с заранее заготовленным образцом. Слова с ошибками демонстрируются при помощи функции `alert`. Когда все ошибки исправлены, приложение выставляет оценку.

Контрольная работа. Ученик заполняет в тексте на экране пропущенные места (они могут быть представлены, например, символами подчеркивания), затем нажимает кнопку **Проверить**. Приложение сравнивает написанный текст с заранее заготовленным образцом. Слова с ошибками показываются при помощи функции `alert`. Когда все ошибки исправлены, приложение выставляет оценку.

Электронный контролер может быть снабжен кнопкой **Помощь**, нажатие на которую создает дополнительное окно и выводит в него справочник по тем правилам, на которые ориентирован диктант или контрольная работа.



Часть IV

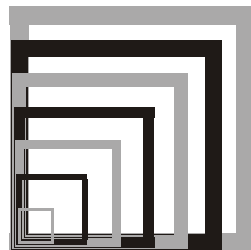
Первый практикум на «кухне» Сидорова

Урок 15. Конструирование объектов

Урок 16. Меню

Урок 17. Роботландский Плюсик

Урок 15



Конструирование объектов

Объект для хранения данных

Однажды Иван Сидоров задумал написать приложение для просмотра своих любимых картинок. Он решил сделать меню и окошко для демонстрации в нем графики. Пользователь выбирает нужную позицию, и картинка появляется на экране (рис. 15.1).

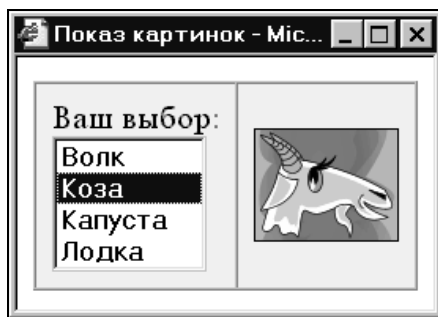


Рис. 15.1. Приложение для просмотра картинок

— Удобная организация данных в любой программе — это половина успеха! — справедливо заметил Иван. С каждой строкой меню должны быть связаны две величины: название картинки (например, «Волк») и путь к ней на диске (например, «./pic/wolf.gif»). Названия нужно выводить на экран, а путь к соответствующему файлу присваивать свойству `src` объекта, построенного браузером для тега `<IMG>`.

Для задания одной строки своего меню, Иван построил такой объект:

```
// Объект для описания одной строки меню.  
// Конструктор объекта item.  
// Аргументы конструктора:  
//   n - название картинки;  
//   f - путь к картинке;
```

```
function item(n, f)
{
    this.name=n; // Название картинки.
    this.file=f; // Путь к картинке.
}
```

Как видите, описание конструктора объекта очень похоже на описание обычной функции! О том, что это конструктор, говорит ключевое слово **this**, отделяемое точкой от переменных `name` и `file`.

Если записать так:

```
function fun(n, f)
{
    var name = n;
    var file = f;
}
```

— то получится обычная функция, а не конструктор объекта, причем, функция совершенно бесполезная. Ведь переменные `name` и `file` существуют только внутри функции и не сохраняют свои значения после завершения ее работы.

Указания **this** сообщают браузеру, что построен объект, значит, от него можно образовывать экземпляры и внутри каждого экземпляра работать со своими переменными `name` и `file`, которые теперь уже называются не переменными, а свойствами объекта `item`.

```
// Создали первый экземпляр объекта item.
var obj1 = new item("Волк", "./pic/wolf.gif");
// Создали второй экземпляр объекта item.
var obj2 = new item("Коза", "./pic/goat.gif");
// Создали третий экземпляр объекта item.
var obj3 = new item("Капуста", "./pic/cabbage.gif");
// Создали четвертый экземпляр объекта item.
var obj4 = new item("Лодка", "./pic/boat.gif");

// Создали массив из построенных экземпляров:
var items = new Array(obj1, obj2, obj3, obj4);
```

Созданный массив `items` очень удобно использовать. Можно программным путем (при помощи метода `document.write()`) построить меню выбора на теге **<SELECT>** (листинг 15.1).

Листинг 15.1

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
  <TITLE>Меню построено программой</TITLE>
  <SCRIPT language=JavaScript>
    <!--
      // Объект для описания одной строки меню.
      // Конструктор объекта item.
      // Аргументы конструктора:
      //   n - название картинки;
      //   f - путь к картинке;
      function item(n, f)
      {
        this.name=n; // Название картинки.
        this.file=f; // Путь к картинке.
      }
      //-- Конеч описание объекта.
      var obj1 = new item("Волк", "./pic/wolf.gif");
      var obj2 = new item("Коза", "./pic/goat.gif");
      var obj3 = new item("Капуста", "./pic/cabbage.gif");
      var obj4 = new item("Лодка", "./pic/boat.gif");
      // Создаем массив из построенных экземпляров:
      var items = new Array(obj1, obj2, obj3, obj4);
    //-->
  </SCRIPT>
</HEAD>
<BODY bgcolor=white text=black>
  <H2>Меню построено программой</H2>
  <SCRIPT language=JavaScript>
    <!--
      // Построение меню.
      document.write("<SELECT size="+items.length+">");
      for (var ind=0; ind<items.length; ind++)
        document.write("<OPTION>"+ items[ind].name+"</OPTION>");
      document.write("</SELECT>");
```

```
//-->  
</SCRIPT>  
</BODY>  
</HTML>
```

Внешний вид этой странички показан на рис. 15.2.

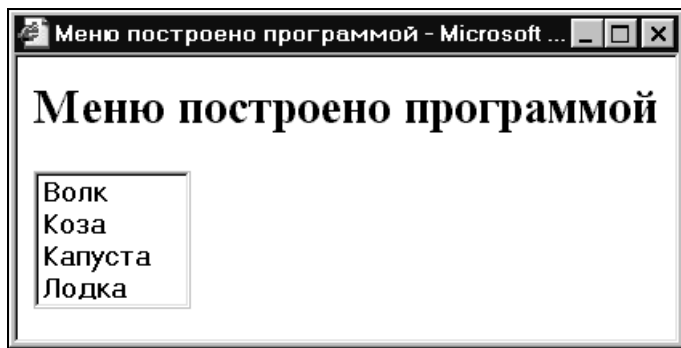


Рис. 15.2. Страничка, построенная «на лету»

Первая команда `document.write("<SELECT size="+items.length+">")` строит тег `<SELECT>`, в котором атрибут `size` (число видимых строк меню) получает значение `items.length`, т. е. становится равным числу элементов в массиве `items` — все строки меню видны на экране.

Позиции меню `<OPTION>...</OPTION>` строятся при помощи цикла:

```
for(var ind=0; ind<items.length; ind++)  
    document.write("<OPTION>"+ items[ind].name+"</OPTION>");
```

В позицию меню на экране с номером `ind` записывается значение `items[ind].name`. Здесь:

<code>items</code>	— массив экземпляров объектов <code>item</code> ;
<code>items[ind]</code>	— экземпляр объекта <code>item</code> с номером <code>ind</code> в массиве <code>items</code> ;
<code>items[ind].name</code>	— свойство <code>name</code> экземпляра объекта <code>item</code> с номером <code>ind</code> в массиве <code>items</code> .

Последняя команда `document.write("</SELECT>")` строит закрывающий тег `</SELECT>`.

Таким образом, приведенный выше скрипт строит HTML-код, равнозначный следующему (листинг 15.2).

Листинг 15.2

```
<SELECT size=4>
  <OPTION>Волк</OPTION>
  <OPTION>Коза</OPTION>
  <OPTION>Капуста</OPTION>
  <OPTION>Лодка</OPTION>
</SELECT>
```

Зачем программировать этот простой код? Почему бы просто не записать его на HTML? На самом деле смысла в проделанной работе гораздо больше, чем кажется на первый взгляд: мы сделали построение HTML-кода зависимым от набора данных в массиве `items`. Это означает, что для нового меню изменится только `items`, но не код, который строит меню. Мы выделили в задаче «подвижную» часть — набор данных, а код построения «отлили» в скрипте.

Страничка-иллюстратор, построенная Сидоровым (рис. 15.1), задается следующим кодом (листинг 15.3).

Листинг 15.3

```
<HTML>
  <HEAD>
    <META http-equiv="Content-Type"
      content="text/html; charset=windows-1251">
    <TITLE>Показ картинок</TITLE>
    <SCRIPT language=JavaScript>
    <!--
      // Объект для описания одной строки меню.
      // Конструктор объекта item.
      // Аргументы конструктора:
      //   name - название картинки;
      //   file - путь к картинке.
      function item(name, file)
      {
        this.name=name;
        this.file=file;
      }
      //-- Конец описания объекта.
      var items = new Array( // Данные для меню выбора.
        new item("Волк", "./pic/wolf.gif"),
```

```

    new item("Коза", "./pic/goat.gif"),
    new item("Капуста", "./pic/cabbage.gif"),
    new item("Лодка", "./pic/boat.gif")
    );

var mark=0; // Отметка выбора позиции в меню.
// Обработка изменения отметки выбора в меню
function Make()
{
    var ind = document.menu.sel.selectedIndex;
    if (mark != ind)
    {
        document.images.pic.src=items[ind].file;
        mark = ind;
    }
}
//-->
</SCRIPT>
</HEAD>
<BODY bgcolor=white text=black>
<H2>Показ картинок</H2>
<FORM name=menu>
<TABLE border=1 bgcolor=#F0F0F0 cellpadding=10>
<TR>
<TD>
    Ваш выбор:<BR>
    <SCRIPT language=JavaScript>
    <!--
    // Построение меню.
    document.write("<SELECT name=sel onchange='Make()' "+
        " size="+items.length+">");
    for(var ind=0; ind<items.length; ind++)
        document.write("<OPTION "+
            (ind == mark ? ' selected' : '')+">"+
            items[ind].name+"</OPTION>");
    document.write("</SELECT>");
    Make();
    //-->
    </SCRIPT>
</TD>

```

```
<TD>
  <SCRIPT language=JavaScript>
    <!--
    document.write("<IMG name=pic src="+items[mark].file+
                  " width=90 height=70 border=1 alt=''>");
    //-->
  </SCRIPT>
</TD>
</TR>
</TABLE>
</FORM>
</BODY>
</HTML>
```

Функция `Make()` срабатывает, когда пользователь выбирает новую позицию в меню. Сначала определяется индекс отмеченной позиции:

```
var ind = document.menu.sel.selectedIndex;
```

Затем на экран выводится соответствующее изображение:

```
document.images.pic.src=items[ind].file;
```

Для доступа к свойству `src` объекта `IMG` используется коллекция `images` (она входит в состав объекта `document` и содержит ссылки на объекты, построенные для всех тегов `<IMG>` документа).

Пользуясь преимуществом разделения данных и кода, Иван легко сделал новую страничку с другими картинками (рис. 15.3).

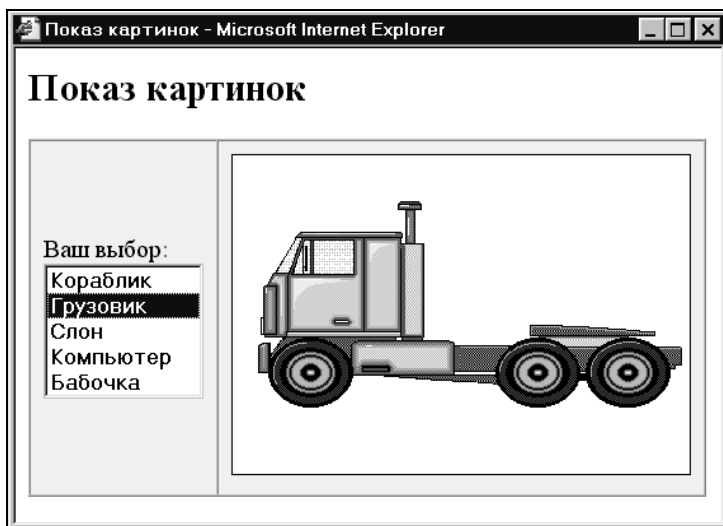


Рис. 15.3. Иллюстратор Сидорова с другими картинками

Что изменил Сидоров в старом файле? Очень немного. Прежнее описание массива `items` заменилось новым:

```
var items = new Array(  
    new item("Кораблик", ". /pic/ship.gif"),  
    new item("Грузовик", ". /pic/truck.gif"),  
    new item("Слон",      ". /pic/ elephant.gif"),  
    new item("Компьютер", ". /pic/ibm.gif"),  
    new item("Бабочка",   ". /pic/butterfl.gif")  
);
```

Еще Ивану пришлось изменить код построения атрибутов, задающих размер картинки. Команда:

```
document.write("<IMG name=pic src="+items[0].file+  
    " width=90 height=70 border=1 alt=''>");
```

заменилась на

```
document.write("<IMG name=pic src="+items[0].file+  
    " width=360 height=250 border=1 alt=''>");
```

Эти изменения Ивана расстроили. Нарушилась его задумка «неприкосновенности» кода построения страницы. Однако Иван легко вышел из положения, определив наряду с массивом `items` дополнительные переменные `width` и `height`. Конечно, пришлось немного переделать программный код. Данные для меню стали определяться так:

```
var width  = 360;  
var height = 250;  
var items = new Array(  
    new item("Кораблик", ". /pic/ship.gif"),  
    new item("Грузовик", ". /pic/truck.gif"),  
    new item("Слон",      ". /pic/elephant.gif"),  
    new item("Компьютер", ". /pic/ibm.gif"),  
    new item("Бабочка",   ". /pic/butterfl.gif")  
);
```

На код построения страницы теперь наложено табу! Читатель может вслед за Иваном изменить программу иллюстратора так, чтобы она учитывала значения переменных `width` и `height`.

* * *

На «кухне» Сидорова сегодня случилось много интересного и поучительного. Главное блюдо — это собственный объект `item`, «испеченный» на JavaScript. Подведем «кулинарные» итоги.

- Объект описывается при помощи конструкции **function**. Это описание называется конструктором объекта (у нас конструктор объекта: `item`).

- В конструкторе свойства объекта задаются при помощи конструкции `this.имя_свойства` (у нас: `this.name`, `this.file`).
- Экземпляр объекта образуется при помощи конструкции `new` (например, `var obj = new item("Слон", "../pic/elephant.gif")`).

Построение сложного объекта

Размышляя над построенной программой, дотошный Иван отметил в ней следующий недостаток.

Построить на одной странице несколько меню не просто. Ведь программа работает с глобальными переменными `items`, `width`, `height` и `mark`. Для нового меню придется ввести новые переменные с другими именами, например, `items1`, `width1`, `height1` и `mark1`. Кроме того, имена тегов `<FORM>`, `<SELECT>`, `<IMG>` тоже нужно менять с `menu`, `sel` и `pic` на `menu1`, `sel1` и `pic1`. Для нового меню придется скопировать все скрипты и вносить соответствующие изменения в именах.

Вся эта возня и рутина совсем испортили Ивану настроение. Наконец, он придумал красивый выход: надо сделать меню при помощи объекта! Тогда от этого объекта можно будет образовать сколько угодно экземпляров, и никаких конфликтов с именами не будет.

Не откладывая дело в долгий ящик, Иван записал для этого объекта такой интерфейс (листинг 15.4).

Листинг 15.4

```
<SCRIPT language=JavaScript>
<!--
// Конструируем первое меню.
var menu1 = new Menu("menu1", 90, 70,
    "Волк",    "../pic/wolf.gif",
    "Коза",    "../pic/goat.gif",
    "Капуста", "../pic/cabbage.gif",
    "Лодка",   "../pic/boat.gif");

// Показываем меню на экране.
menu1.Show();

// Конструируем второе меню.
var menu2 = new Menu("menu2", 360, 250,
    "Кораблик", "../pic/ship.gif",
    "Грузовик", "../pic/truck.gif",
    "Слон",      "../pic/elephant.gif",
    "Компьютер", "../pic/ibm.gif",
    "Бабочка",   "../pic/butterfl.gif");
```

```
// Показываем меню на экране.
menu2.Show();
//-->
</SCRIPT>
```

Как видите, Иван назвал новый объект `Menu`. В приведенном коде создаются два экземпляра этого объекта с именами `menu1` и `menu2`. Каждый экземпляр выводит меню на экран при помощи метода `Show()`, который, конечно, должен быть включен в описание объекта.

Вызов конструктора объекта `Menu` у Ивана получился таким:

```
Menu(имя_экземпляра, ширина_картинки, высота_картинки,
    имя1, файл1,
    ....
    имяN, файлN);
```

Среди аргументов конструктора не очень понятно назначение первого: «имя_экземпляра». Иван объяснил присутствие этого аргумента так. Функция `Make()` должна работать тогда, когда пользователь меняет отметку в меню, т. е. вызов этой функции должен быть расположен в теге `<SELECT>`:

```
<SELECT ... onchange='Make()'>
```

Но для экземпляра `menu1` вызов должен быть записан как `menu1.Make()`, а для экземпляра `menu2` — как `menu2.Make()`. Получается, что для построения HTML-кода нужно знать имя экземпляра объекта `Menu`. Кроме того, имя экземпляра, передаваемое конструктору как первый его аргумент, можно будет использовать для построения уникальных имен тегов `<FORM>`, `<SELECT>` и `<IMG>`, принимающих участие в построении экранного образа.

Программу с объектом `Menu` Иван поместил в отдельный файл `menu.js` (листинг 15.5).

Листинг 15.5

```
//--- файл menu.js -----
// Меню выбора на теге SELECT:
// Демонстрация картинок одного размера
// -----
// Использование
// -----
// Меню представляет собой объект. Необходимо:
// 1. Создать экземпляр объекта
// var имя_экземпляра = new Menu(имя_экземпляра, ширина, высота,
//                                название1, картинка1,
//                                ...);
```

```
//      Здесь:
//      имя_экземпляра - имя созданного экземпляра;
//      ширина         - ширина картинок;
//      высота         - высота картинок;
//      название1      - текст для первой строки меню;
//      картинка1      - путь к файлу с первой картинкой.
// 2. Настроить внешний вид меню, используя свойства объекта:
//      имя_экземпляра.bgcolor = цвет фона подложки меню
//      имя_экземпляра.border  = толщина рамки вокруг картинки
//      имя_экземпляра.mark    = номер отмеченной строки (нумерация с 0)
// 3. Показать меню на экране
//      имя_экземпляра.Show();
//-----
// Описание одной строки меню (название, файл).
function item(name, file)
{
    this.name=name; // Название картинки.
    this.file=file; // Путь к картинке.
}
//-----
// Конструктор главного объекта.
function Menu(name, width, height)
{
    // Свойства меню для пользователя.
    this.bgcolor = "white"; // Цвет фона подложки меню.
    this.border  = 0;       // Толщина рамки вокруг картинки.
    this.mark    = 0;       // Отмеченная строка в меню.
    // Методы меню для пользователя.
    this.Show = _Show;     // Вывести меню на экран.

    // "Внутренние" свойства.
    this.items = new Array(); // Массив строк меню.
    this.name  = name;       // Имя экземпляра объекта.
    this.width = width;      // Ширина картинки.
    this.height= height;     // Высота картинки.
    // "Внутренние" функции.
    this.Make  = _Make;      // Обработка щелчка на строке меню.

    var len = arguments.length; // Число аргументов конструктора
    if(!len%2) len--;           // должно быть нечетным.
```

```

// Заполним массив items (строки меню).
var j = 0; // Это индекс по массиву items.
for(var i=3; i<len; i+=2)
    this.items[j++] = new item(arguments[i],arguments[i+1]);
}
//-----
// Обработка изменения отметки выбора в меню.
function _Make()
{
    // Вычислить индекс отмеченной строки.
    var ind = document[this.name+"Form"][this.name+"Select"].selectedIndex;
    // Если отмечена новая строка, показать новую картинку.
    if (this.mark != ind)
    {
        document.images[this.name+"Img"].src=this.items[ind].file;
        this.mark = ind;
    }
}
//-----
// Генерация HTML-кода для показа меню на экране.
function _Show()
{
    var str="<FORM name="+this.name+"Form">"+
        "<TABLE border=1 bgcolor="+this.bgcolor+
        " cellspacing=0 cellpadding=10">"+
        "<TR><TD>"+
        "Ваш выбор:<BR>"+
        "<SELECT name="+this.name+"Select"+
        " onchange='"+this.name+".Make()'"+
        " size="+this.items.length+">";
    for(var ind=0; ind<this.items.length; ind++)
        str += "<OPTION"+(ind==this.mark ? " selected":"" )+ ">"+
            this.items[ind].name+"</OPTION>";
    str += "</SELECT>";
    str += "</TD><TD>";
    str += "<IMG name="+this.name+"Img"+
        " src="+this.items[this.mark].file+
        " width="+this.width+ " height="+this.height+

```

```
" border="+this.border+" alt=''>;  
str +="</TD></TR></TABLE></FORM>";  
document.write(str);  
this.Make();  
}
```

Замечание

Вызов `this.Make()` приводит в соответствие отмеченный пункт меню и картинку при перезагрузке содержимого окна (кнопка **Обновить** браузера) и переходах **Вперед/Назад**, так как в этом случае элементы формы сохраняют значения, установленные пользователем, а это может привести к неверному отображению построенного иллюстратора.

Для проверки работы объекта `Menu` Иван сконструировал страничку `menu.htm`, на которой построил сразу четыре экземпляра объекта (рис. 15.4).

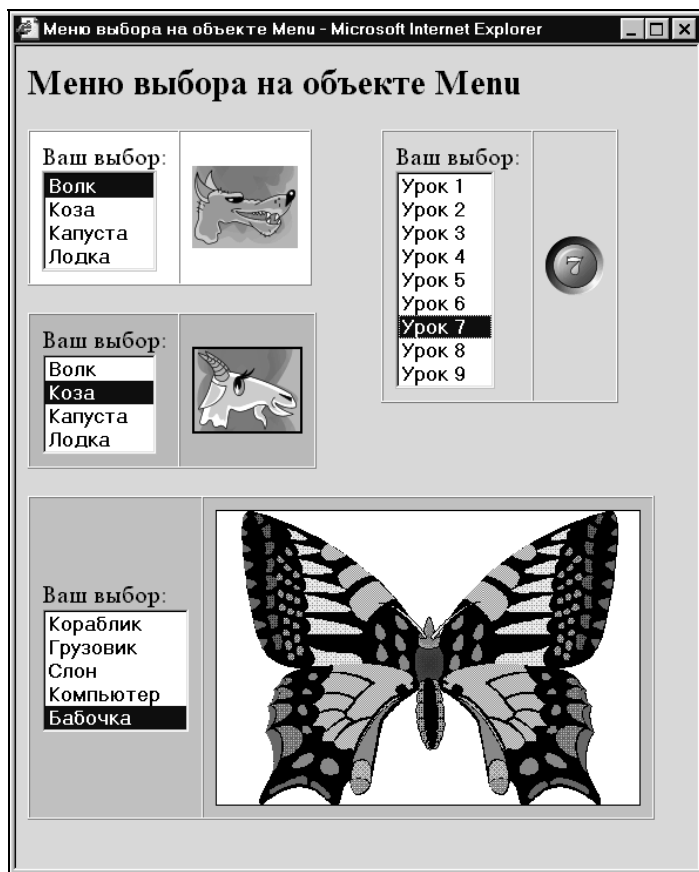


Рис. 15.4. Четыре иллюстратора на одной страничке

Код этой странички выглядит так (листинг 15.6).

Листинг 15.6

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
  <SCRIPT language="JavaScript" src="menu.js"></SCRIPT>
  <TITLE>Меню выбора на объекте Menu</TITLE>
</HEAD>
<BODY bgcolor=#DFD8C5 text=black>
  <H2>Меню выбора на объекте Menu</H2>
  <TABLE border=0 cellpadding=0 cellspacing=0>
    <TR valign=top>
      <TD>
        <SCRIPT language=JavaScript>
          <!--
            var menu1 = new Menu("menu1", 90, 70,
                                "Волк",    ".\pic\wolf.gif",
                                "Коза",    ".\pic\goat.gif",
                                "Капуста", ".\pic\cabbage.gif",
                                "Лодка",   ".\pic\boat.gif");

            menu1.Show();
            var menu2 = new Menu("menu2", 90, 70,
                                "Волк",    ".\pic\wolf.gif",
                                "Коза",    ".\pic\goat.gif",
                                "Капуста", ".\pic\cabbage.gif",
                                "Лодка",   ".\pic\boat.gif");

            menu2.bgcolor="#CFB597";
            menu2.border=2;
            menu2.mark=1;
            menu2.Show();
          //-->
        </SCRIPT>
      </TD>
      <TD>
        <SCRIPT language=JavaScript>
          <!--
```

```
var menu3 = new Menu("menu3", 50, 50,
    "Урок 1", "./pic/js011.gif",
    "Урок 2", "./pic/js021.gif",
    "Урок 3", "./pic/js031.gif",
    "Урок 4", "./pic/js041.gif",
    "Урок 5", "./pic/js051.gif",
    "Урок 6", "./pic/js061.gif",
    "Урок 7", "./pic/js071.gif",
    "Урок 8", "./pic/js081.gif",
    "Урок 9", "./pic/js091.gif");

menu3.bgcolor="#DFD8C5";
menu3.mark=6;
menu3.Show();
//-->
</SCRIPT>
</TD>
</TR>
<TR>
<TD colspan=2>
<SCRIPT language=JavaScript>
<!--
var menu4 = new Menu("menu4", 360, 250,
    "Кораблик", "./pic/ship.gif",
    "Грузовик", "./pic/truck.gif",
    "Слон",      "./pic/elephant.gif",
    "Компьютер", "./pic/ibm.gif",
    "Бабочка",   "./pic/butterfl.gif");

menu4.bgcolor="silver";
menu4.border=1;
menu4.mark=4;
menu4.Show();
//-->
</SCRIPT>
</TD>
</TR>
</TABLE>
</BODY>
</HTML>
```

Пояснения к файлу menu.js

Конструктор

Конструктор объекта задается следующим кодом (листинг 15.7).

Листинг 15.7

```
01 function Menu(name, width, height)
02 {
03     // Свойства меню для пользователя.
04     this.bgcolor = "white";      // Цвет фона подложки меню.
05     this.border  = 0;           // Толщина рамки вокруг картинки.
06     this.mark    = 0;           // Отмеченная строка в меню.
07     // Методы меню для пользователя.
08     this.Show = _Show;         // Вывести меню на экран.
09
10     // "Внутренние" переменные.
11     this.items = new Array();   // Массив строк меню.
12     this.name  = name;          // Имя экземпляра объекта.
13     this.width = width;         // Ширина картинки.
14     this.height= height;        // Высота картинки.
15
16     // "Внутренние" функции.
17     this.Make  = _Make;         // Обработка щелчка на строке меню.
18
19     var len = arguments.length; // Число аргументов конструктора
20     if(!len%2) len--;           // должно быть нечетным.
21
22     // Заполним массив items (строки меню).
23     var j = 0; // Это индекс по массиву items.
24     for(var i=3; i<len; i+=2)
25         this.items[j++] = new item(arguments[i],arguments[i+1]);
26 }
```

В строках 04—06 свойства bgcolor, border, mark определены в расчете на доступ к ним извне. По умолчанию цвет фона меню сделан белым, рамка вокруг картинки отсутствует, в меню отмечена первая строка (ее номер 0).

В строке 11 определен массив `items`. В дальнейшем он заполняется объектами `item`, каждый из которых описывает пару (название, файл) для одной строки меню.

Заметьте, что в описании конструктора (строка 01) указано только три обязательных аргумента. Однако при создании экземпляра можно задавать любое число аргументов в соответствии с инструкцией по использованию объекта. Доступ к аргументам осуществляется при помощи массива `arguments`. При этом `arguments[0]` содержит первый аргумент, `arguments[1]` — второй и т. д. Это свойство JavaScript относится не только к конструкторам объектов, но и к любым функциям.

В строках 12–14 первые три аргумента конструктора заносятся в свойства `name`, `width` и `height`.

В строке 08 определяется метод `Show`. Ниже в программе будет описана функция `_Show`, которая этот метод реализует (генерация HTML-кода построения меню).

В строке 17 описан метод `Make`. Метод реализуется при помощи функции `_Make`. Видим, что метод объекта задается при помощи команды присваивания:

```
this.имя_метода = имя_функции;
```

В левой части размещается имя метода (с приставкой `this.`), а в правой — имя функции, которая этот метод реализует. Чтобы было легко отличить функцию, реализующую метод объекта от обычной функции, рекомендуется вносить в имена функций-методов какую-то регулярную особенность. Например, начинать их со знака подчеркивания, за которым следует имя метода.

В строке 19 локальной переменной `len` присваивается число аргументов, переданное конструктору при создании экземпляра. Это число должно быть нечетным. Если это не так, строка 20 отсекает последний аргумент. Пусть, например, экземпляр задан кодом (листинг 15.8).

Листинг 15.8

```
var menu1 = new Menu("menu1", 90, 70,  
    "Волк",    ". /pic/wolf.gif",  
    "Коза",    ". /pic/goat.gif",  
    "Капуста", ". /pic/cabbage.gif",  
    "Лодка");
```

Меню будет построено только на три позиции («Лодка» будет проигнорирована). Таким образом, строка 20 — это попытка Ивана поставить в свой объект защиту от неверных данных. Отметим, что этот полезный фрагмент конструктора можно было бы и усилить.

В строках 23—25 происходит заполнение массива `items` данными, извлеченными из аргументов конструктора.

Метод *Show*

Этот метод генерирует HTML-код, который отображает меню на экране. Метод реализует функция `_Show`:

```
01 function _Show()
02 {
03     var str="<FORM name="+this.name+"Form">"+
04         "<TABLE border=1 bgcolor="+this.bgcolor+
05         " cellspacing=0 cellpadding=10">"+
06         "<TR><TD>"+
07         "Ваш выбор:<BR>"+
08         "<SELECT name="+this.name+"Select"+
09         " onchange='"+this.name+".Make()'"+
10         " size="+this.items.length+">";
11     for(var ind=0; ind<this.items.length; ind++)
12         str += "<OPTION"+(ind==this.mark ? " selected":"" )+ ">"+
13             this.items[ind].name+"</OPTION>";
14     str += "</SELECT>";
15     str += "</TD><TD>";
16     str += "<IMG name="+this.name+"Img"+
17         " src="+this.items[this.mark].file+
18         " width="+this.width+ " height="+this.height+
19         " border="+this.border+ " alt=' '>";
20     str += "</TD></TR></TABLE></FORM>";
21     document.write(str);
22     this.Make();
23 }
```

Например, для экземпляра:

```
var menu1 = new Menu("menu1", 90, 70,
    "Волк",    ". /pic/wolf.gif",
    "Коза",    ". /pic/goat.gif",
    "Капуста", ". /pic/cabbage.gif",
    "Лодка",   ". /pic/ship.gif");
```

метод `Show` строит код, равнозначный следующему (листинг 15.9).

Листинг 15.9

```

<FORM name=menu1Form>
  <TABLE border=1 bgcolor=white cellspacing=0 cellpadding=10>
    <TR>
      <TD>
        Ваш выбор:<BR>
        <SELECT name=menu1Select onchange='menu1.Make()' size=4>
          <OPTION selected>Волк</OPTION>
          <OPTION>Коза</OPTION>
          <OPTION>Капуста</OPTION>
          <OPTION>Лодка</OPTION>
        </SELECT>
      </TD>
      <TD>
        <IMG name=menu1Img src=./pic/wolf.gif
          width=90 height=70 border=0 alt=''>
      </TD>
    </TR>
  </TABLE>
</FORM>

```

Посмотрите, как метод Show использует имя экземпляра, переданное через первый аргумент конструктора:

- ❑ имя формы образуется из имени экземпляра добавлением окончания «Form» (строка 03);
- ❑ Имя тега <SELECT> образуется из имени экземпляра добавлением окончания «Select» (строка 08);
- ❑ имя картинки образуется из имени экземпляра добавлением окончания «Img» (строка 16);
- ❑ в теге <SELECT> обращение к обработчику события change построено с учетом имени экземпляра: menu1.Make() (строка 09);
- ❑ значение свойств bgcolor, border, mark использовано при построении таблицы (строка 04), картинки (строка 19) и пунктов меню (строка 12).

Метод Make

Этот метод является обработчиком события change в теге <SELECT>:

```

01 function _Make()
02 {

```

```
03 // Вычислить индекс отмеченной строки.
04 var ind =
05 document[this.name+"Form"][this.name+"Select"].selectedIndex;
06 // Если отмечена новая строка, показать новую картинку.
07 if(this.mark != ind)
08 {
09     document.images[this.name+"Img"].src=this.items[ind].file;
10     this.mark = ind;
11 }
12 }
```

Для экземпляра объекта с именем `menu1` этот код будет эквивалентен такому:

```
function _Make()
{
    // Вычислить индекс отмеченной строки.
    var ind = document.menu1Form.menu1Select.selectedIndex;
    // Если отмечена новая строка, показать новую картинку.
    if(this.mark != ind)
    {
        document.images.menu1Img.src=this.items[ind].file;
        this.mark = ind;
    }
}
```

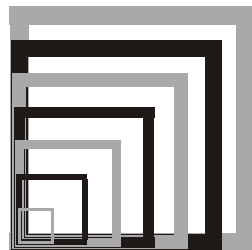
Переменная `mark` хранит номер отмеченной строки, поэтому перерисовка картинки происходит лишь тогда, когда это действительно необходимо.

В строках 04—05 определяется индекс отмеченной строки. В строке 09 задается вывод новой картинки на экран. Для доступа к свойству `src` объекта соответствующему тегу `<IMG>` используется коллекция `images`.

Задания

1. Создайте страничку для демонстрации своих любимых картинок.
2. Попробуйте улучшить объект `Menu`. Усиьте контроль правильности аргументов конструктора. Добавьте новые свойства, которые можно будет использовать для управления внешним видом меню на экране. Измените объект таким образом, чтобы каждая картинка могла иметь свои собственные размеры.

Урок 16



Меню

«Искре-1256» посвящается...

Мы снова на программистской «кухне» Ивана Сидорова. Он товарищ открытый, секретов не держит и всегда готов поделиться своими находками за чашкой крепкого чая. Так было и встарь в славном городе Благовещенске, где сиживал он с друзьями в ЛПМ БГПИ (лаборатория прикладной математики Благовещенского пединститута). Ах, что тогда вытворялось там, на технике, которая теперешним школьникам и не снилась! Взять хотя бы такого отечественного монстра персонального компьютеростроения, как «Искра-1256». Оперативной памяти — аж 4 Кбайта! Вместо дисководов — один накопитель на кассете от обычного бытового магнитофона. Язык программирования тоже отечественный, без названия. Иван с друзьями окрестили его Дуб, за жесткую непоколебимую сущность. Представьте, имена переменных нельзя было выбирать произвольно, все они были предопределены заранее конструкторами языка. Какая-то гремучая смесь из Алгола и Ассемблера. Внешний вид программ — не для слабонервных. Иван точно не помнит, но, кажется, в языке не было даже комментариев. Тем не менее было бы вдохновение, да интерес, да мускулистые руки — посадим картофель и лопатой! А уж этого добра (интереса и вдохновения) было в ЛПМ сколько угодно. И лились рекой крепкие чаи, и рождались на 4 Кбайтах удивительные вещи. Например, среда исполнителя Кукарача. В среде и сам исполнитель, и редактор программ для него, и, конечно, транслятор. Техника теперь изменилась, Сидоров уехал из Благовещенска. Пьет он теперь чай один на своей «кухне» для компьютера с 256-мегабайтовой памятью и увлечен HTML, JavaScript, CSS.

Предлагаемые ниже заметки Сидорова публикуются с его согласия и посвящаются, по его настоянию, той самой «Искре-1256», о которой у Ивана остались самые теплые чувства.

Заметки посвящены теме меню, которая была затронута на предыдущем уроке.

— Меню нужны разные! — так подумал Иван и налил первый стакан чаю.

Меню ссылок на UL

Стандартный список HTML — чем не меню? Вот вам и пример этого (рис. 16.1).

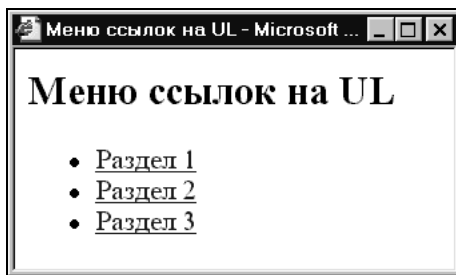


Рис. 16.1. Меню на стандартном маркированном списке

Код этого меню — проще не бывает (листинг 16.1).

Листинг 16.1

```
<UL type=disc>
  <LI><A href="item1.htm">Раздел 1</A>
  <LI><A href="item2.htm">Раздел 2</A>
  <LI><A href="item3.htm">Раздел 3</A>
</UL>
```

До того просто, что не интересно. Никакого JavaScript, CSS. Так не пойдет. Давайте создадим функцию, которая будет сама строить это меню там, где нам надо.

Пусть интерфейс функции будет таким:

```
menu1(
    "Раздел1", "item1.htm",
    "Раздел2", "item2.htm",
    "Раздел3", "item3.htm"
);
```

В аргументах функции записываются название разделов и ссылки, им соответствующие, а функция строит на экране список.

Построить список динамически можно при помощи метода `write` объекта `document`. Есть только одна особенность: число аргументов функции `menu1` заранее неизвестно. Но и это не беда: JavaScript для каждой нашей функции строит одноименный объект. Аргументы функции записываются в специальный массив этого объекта: `имя_функции.arguments`. Внутри самой функции можно писать короче: `arguments`. Длина массива, а значит, число ар-

гументов функции, как обычно, содержится в свойстве `arguments.length`. Учитывая вышесказанное, кодируем функцию так:

```
// Построение списка ссылок.  
// Обращение к функции:  
// menu1(  
//     "имя раздела меню", "ссылка",  
//     ...  
//     "имя раздела меню", "ссылка",  
//     );  
function menu1()  
{  
    var alen = arguments.length; // Число аргументов функции.  
    if (alen > 1) // Функция работает, если число аргументов больше одного.  
    {  
        // Если число аргументов нечетно, последний аргумент игнорируем.  
        if(alen%2) alen--;  
        document.write("<UL type=disc>"); // Строим начальный тег списка.  
        for(var i=0; i<alen; i+=2) // Построение списка  
            document.write("<LI><A href="+arguments[i+1]+">"+  
                            arguments[i]+"</A>");  
        document.write("</UL>");// Строим конечный тег списка.  
    }  
}
```

Функция `menu1` не использует сложных данных и глобальных переменных, поэтому заменять ее объектом особого смысла не имеет. Меню, показанное на рис. 16.1, можно отобразить следующим кодом (листинг 16.2, описание функции `menu1` расположено в файле `menu1.js`).

Листинг 16.2

```
<HTML>  
  <HEAD>  
    <META http-equiv="Content-Type"  
          content="text/html; charset=windows-1251">  
    <SCRIPT language="JavaScript" src="menu1.js"> </SCRIPT>  
    <TITLE>Меню ссылок</TITLE>  
  </HEAD>  
  <BODY bgcolor=white text=black>  
    <H2>Меню ссылок</H2>  
    <SCRIPT language=JavaScript>
```

```
<!--  
    menu1(  
        "Раздел1", "item1.htm",  
        "Раздел2", "item2.htm",  
        "Раздел3", "item3.htm"  
    );  
    //-->  
</SCRIPT>  
</BODY>  
</HTML>
```

Меню ссылок на **<SELECT>**

Второй стакан чая Иван поднял за тег **<SELECT>**.

Без JavaScript меню вроде того, что показано на рис. 16.2, интересно, но бесполезно.

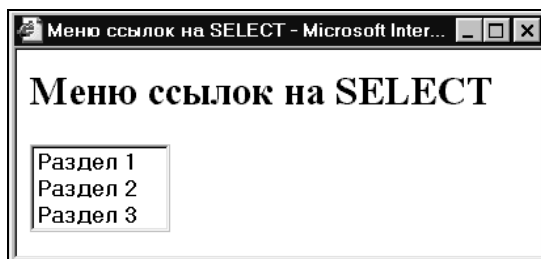


Рис. 16.2. Меню на блоке **<SELECT>**

Код этого меню представлен в листинге 16.3.

Листинг 16.3

```
<FORM>  
    <SELECT size=3>  
        <OPTION>Раздел 1</OPTION>  
        <OPTION>Раздел 2</OPTION>  
        <OPTION>Раздел 3</OPTION>  
    </SELECT>  
</FORM>
```

Ведь надо определить, что выбрал пользователь, и в соответствии с этим что-то сделать, например, загрузить новую страницу. Но JavaScript в наших руках! Для меню на три раздела можно написать такой код (листинг 16.4).

Листинг 16.4

```

<FORM>
  <SELECT size=3 name=select>
    <OPTION>Раздел 1</OPTION>
    <OPTION>Раздел 2</OPTION>
    <OPTION>Раздел 3</OPTION>
  </SELECT>
  <INPUT type=button value=Пуск onclick="menu2_go(this.form.select)">
</FORM>

```

Функция `menu2_go`, которая обрабатывает щелчок на кнопке **Пуск**, может быть такой:

```

function menu2_go(sel)
{
    var link;
    if      (sel.selectedIndex==0) link="item1.htm";
    else if (sel.selectedIndex==1) link="item2.htm";
    else                               link="item3.htm";
    location.href=link;
}

```

Да, но мы поставили задачу написать функцию с универсальным интерфейсом:

```

menu2 (
    "Раздел1", "item1.htm",
    ...
    "Разделn", "itemn.htm"
);

```

Придется опять поработать с функцией `write` объекта `document`. Вот код, который делает все, что задумано:

```

// Построение меню ссылок.
// Обращение к функции:
// menu2 (
//      "имя раздела меню", "ссылка",
//      ...
//      "имя раздела меню", "ссылка"
//      );
// -----
var links = new Array(); // В этот массив помещаются ссылки.
function menu2 (

```

```

{
    var alen = arguments.length; // Число аргументов функции
    if (alen > 1) // Функция работает, если число аргументов больше одного.
    {
        // Если число аргументов нечетно, последний аргумент игнорируем.
        if(alen%2) alen--;
        document.write("<FORM><SELECT name=select size= "+alen/2+">");
        // Построение позиций меню и заполнение массива ссылками.
        for(var i=0; i<alen; i+=2)
        {
            document.write("<OPTION>"+arguments[i]+"</OPTION>");
            links[i/2] = arguments[i+1];
        }
        document.write("</SELECT><INPUT type=button value=Пуск"+
            " onclick='menu2_go(this.form.select)'></FORM>");
    }
}
// Обработка щелчка на кнопке "Пуск"
function menu2_go(sel)
{
    location.href= links[sel.selectedIndex];
}

```

Построенным скриптом Иван остался недоволен. Функция menu2 использует глобальный массив links. Если на одной гипертекстовой странице построить несколько таких меню, неизбежно возникнут проблемы. Ведь массив links — один! Выход прост: надо для меню построить объект, а массив links спрятать внутри. Тогда для каждого экземпляра будет создаваться своя копия этого массива.

Иван построил объект MenuSelect. Скрипт он разместил в файле menu3.js:

```

// Меню ссылок (объект MenuSelect).
// Использование.
// 1. Создать экземпляр объекта:
// var имя_экземпляра=MenuSelect(имя_экземпляра,название1,ссылка1, ...);
// Здесь:
//     имя_экземпляра - имя созданного экземпляра;
//     название1      - текст для первой строки меню;
//     ссылка1        - ссылка для первой строки меню.
// 2. Настроить внешний вид меню используя свойства объекта:
// имя_экземпляра.allview = true  - видны все строки меню
// имя_экземпляра.allview = false - видна одна строка меню (по

```

```
//                                     умолчанию)
// имя_экземпляра.mark = 0 - отмеченная строка (по умолчанию первая)
// имя_экземпляра.textgo - текст на кнопке "Пуск" (по умолчанию "Пуск").
// 3. Показать меню на экране
//   имя_экземпляра.Show();
//-----
// Описание одной строки меню (название, ссылка).
function item(name, link)
{
    this.name=name; // Текст в строке меню.
    this.link=link; // Ссылка, соответствующая тексту.
}
// Конструктор главного объекта.
function MenuSelect()
{
    // Свойства объекта для пользователя.
    this.allview = false; // По умолчанию на экране видна одна строка меню.
    this.textgo = "Пуск"; // Текст на кнопке (по умолчанию "Пуск").
    this.mark = 0; // Отметка в меню(первая строка по умолчанию).
    // Методы объекта для пользователя.
    this.Show = _Show; // Вывести меню на экран.

    // Внутренние переменные.
    this.items = new Array(); // Массив строк меню.
    this.name = arguments[0]; // Имя экземпляра объекта.

    // Внутренние функции.
    this.Make = _Make; // Щелчок на строке меню.

    var alen = arguments.length; // Число аргументов конструктора
    if (!alen%2) alen--; // должно быть нечетным.
    // Заполним массив items (строки меню).
    var j = 0; // Это индекс по массиву items.
    for(var i=1; i<alen; i+=2)
        this.items[j++] = new item(arguments[i],arguments[i+1]);
}

// Обработка нажатия кнопки Пуск.
function _Make(sel)
```

```

{
    location.href = this.items[sel.selectedIndex].link;
}
// Генерация HTML-кода для построения меню на экране.
function _Show()
{
    var len = this.items.length;
    var str= "<FORM><SELECT name=select size="+
        (this.allview ? len:1)+">";
    for(var i=0; i<len; i++)
        str += "<OPTION>" + (i==this.mark ? " selected>" : ">") +
            this.items[i].name+"</OPTION>";
    str += "</SELECT><INPUT type=button value='"+this.textgo+"'" +
        " onclick='"+this.name+".Make(this.form.select)'>"+
        "</FORM>";
    document.write(str);
}

```

Пример использования объекта MenuSelect показан на рис. 16.3.

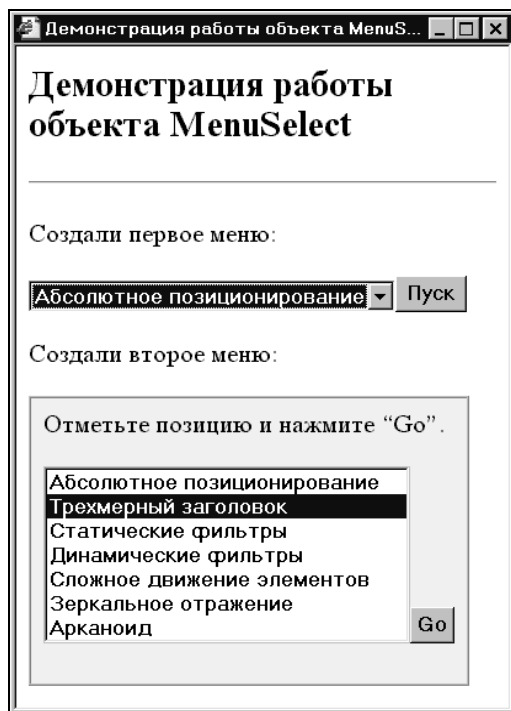


Рис. 16.3. Демонстрация работы объекта MenuSelect

Код этой страницы приводится ниже (листинг 16.5).

Листинг 16.5

[illegible]

```

        "Динамические фильтры", "css4.htm",
        "Сложное движение элементов", "css5.htm",
        "Зеркальное отражение", "css6.htm",
        "Арканоид", "css7.htm");

    menu2.allview = true;
    menu2.textgo="Go";
    menu2.mark=3;
    menu2.Show();
    //-->
</SCRIPT>
</TD>
</TR>
</TABLE>
</BODY>
</HTML>

```

Меню на стандартных кнопках

— Обычные кнопки `<INPUT type=button>` тоже годятся для меню, — подумал Иван. Нажал такую кнопку — и получил результат.

Недолго думая, Сидоров написал такую программу (листинг 16.6).

Листинг 16.6

```

<HTML>
<HEAD>
  <META http-equiv="Content-Type" content="text/html;
    charset=windows-1251">
  <TITLE>Меню на стандартных кнопках-1</TITLE>
</HEAD>
<BODY bgcolor=white text=black>
  <H2>Меню на стандартных кнопках-1</H2>
  <HR>
  <FORM>
    <INPUT type=button value="Абсолютное позиционирование"
      onclick="location.href='css1.htm'"><BR>
    <INPUT type=button value="Трехмерный заголовок"
      onclick="location.href='css2.htm'"><BR>
    <INPUT type=button value="Статические фильтры"
      onclick="location.href='css3.htm'"><BR>
    <INPUT type=button value="Динамические фильтры"

```

```
        onclick="location.href='css4.htm'"><BR>
<INPUT type=button value="Сложное движение элементов"
        onclick="location.href='css5.htm'"><BR>
<INPUT type=button value="Зеркальное отражение"
        onclick="location.href='css6.htm'"><BR>
<INPUT type=button value="Арканоид"
        onclick="location.href='css7.htm'">
</FORM>
</BODY>
</HTML>
```

На экране страничка отобразилась так, как это показано на рис. 16.4.

Кнопочки работают хорошо, но длина у них получилась разная, что, конечно, портит дизайн. Иван попробовал добиться одинаковой длины кнопок за счет пробелов в коротких названиях. Но идея подобрать число пробелов показалась Ивану не очень приятной, а главное, добиться одинаковой длины у всех кнопок так и не удалось — ведь надписи на кнопках браузер записывает пропорциональным шрифтом, т. е. таким, у которого каждый символ имеет свою ширину. Пытался Сидоров сменить шрифт на кнопках на моноширинный, но и это у него не получилось. Потом он вдруг вспомнил про CSS и написал такое стилевое определение:

```
INPUT
{
    width:300px;
    font-size:10pt;
}
```

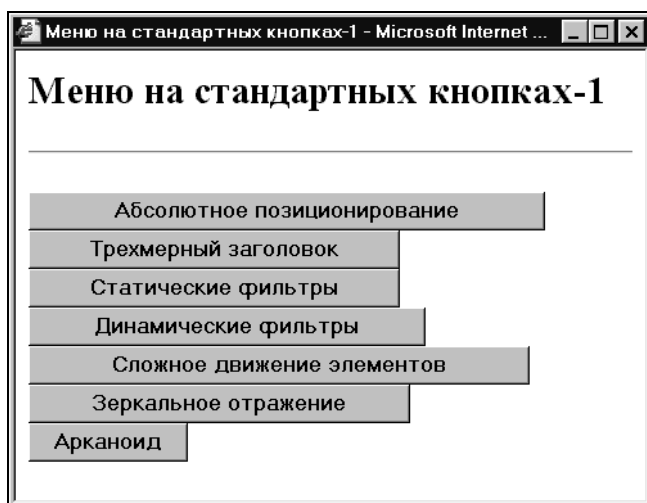


Рис. 16.4. Меню на стандартных кнопках 1

Теперь размеры всех кнопок стали одинаковыми (рис. 16.5).

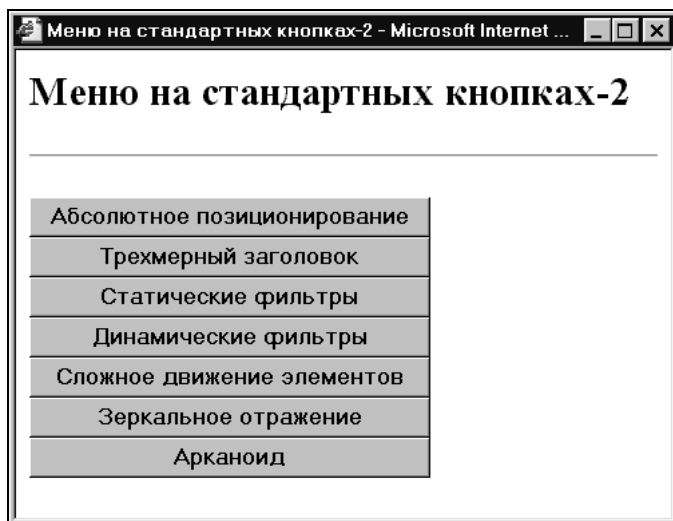


Рис. 16.5. Меню на стандартных кнопках 2

Меню на графических кнопках

Иван Сидоров сделал несколько меню на графических кнопках, которые нарисовал для него компьютерный художник Саша Мур. Саша нарисовал два экземпляра кнопок. Первый набор — основной (рис. 16.6):



Рис. 16.6. Основной набор кнопок

Второй — вид кнопок, когда над ними находится курсор мыши (рис. 16.7).



Рис. 16.7. Второй набор кнопок

Иван — большой любитель порядка. Он придумал для файлов с картинками следующие имена:

key011.gif, key021.gif, Имена файлов для первого набора кнопок.
key031.gif, key041.gif,
key051.gif, key061.gif,
key071.gif

key012.gif, key022.gif, Имена файлов для второго набора кнопок.
 key032.gif, key042.gif,
 key052.gif, key062.gif,
 key072.gif

Меню на этих кнопках получилось таким (рис. 16.8).

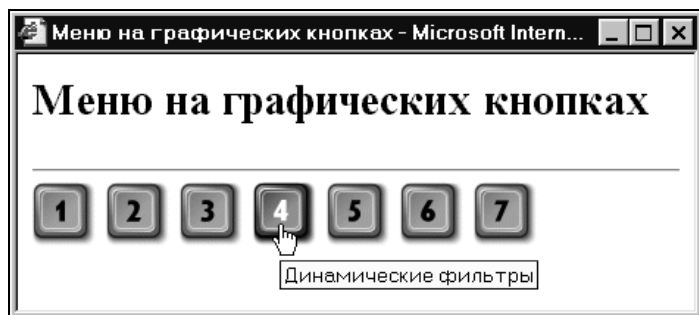


Рис. 16.8. Вид меню на графических кнопках. Курсор мыши над кнопкой 4

Для вывода такого меню на экран Иван придумал объект MenuPic:

```
//-- Файл menupic.js
// Меню ссылок на графических кнопках.
// (работает в Internet Explorer и Netscape Navigator)
// Использование
// -----
// Меню представляет собой объект MenuPic. Необходимо:
// 1. Подготовить картинки кнопки и дать им имена по правилу:
//   имя011.расширение - первая кнопка, курсор не над кнопкой;
//   имя012.расширение - первая кнопка, курсор над кнопкой;
//   ...
// Пример
// key011.gif
// key012.gif
// ...
// 2. Создать экземпляр объекта.
// var имя_экземпляра = MenuPic(
//   имя_экземпляра, основа, расширение, ширина, высота,
//   название1, ссылка1, ...);
// Здесь:
// имя_экземпляра - имя экземпляра объекта MenuPic (например, "mp");
// основа         - основа имени картинок (например, "./pic/key");
```

```
// расширение      - расширение имени картинок (например, "gif");
// ширина          - ширина картинки (например, 43);
// высота          - высота картинки (например, 43);
// название1       - alt-текст для первой кнопки (например, "Урок 1");
// ссылка          - ссылка для первой кнопки (например, "../01.htm")
// ...
// 3. Показать меню на экране.
// имя_экземпляра.Show();
//-----
// Описание одной строки меню.
// pic_over - ссылка на картинку под курсором.
// pic_out - ссылка на основную картинку.
// name - содержание alt-надписи кнопки.
// link - URL для перехода.
function item(pic_over, pic_out, name, link)
{
    // Создали два экземпляра объекта Image.
    this.pic_over = new Image();
    this.pic_out = new Image();

    this.pic_over.src=pic_over;
    this.pic_out.src=pic_out;
    this.name = name;
    this.link = link;
}
// Конструктор главного объекта.
function MenuPic(name,base,ext,width,height)
{
    this.name    = name;           // Имя экземпляра объекта.
    this.width   = width;          // Ширина картинок.
    this.height  = height;         // Высота картинок.
    this.items   = new Array();    // Массив позиций меню.
    this.Show = _Show;             // Метод для показа меню на экране.
    this.ismenu  = arguments.length > 6; // = false, если аргументов мало.

    // Заполнение массива данными
    if(this.ismenu)
```

```

{
    var j=0;
    for(var i=5; i<arguments.length; i+=2)
    {
        var num = j < 9 ? "0"+eval(j+1):eval(j+1);
        this.items[j++] = new item(base+num+"2."+ext,
                                   base+num+"1."+ext,
                                   arguments[i],
                                   arguments[i+1]);
    }
}
}
// Показать меню на экране
function _Show()
{
    if(this.ismenu)
    {
        document.write("<NOBR>");
        for(var i=0; i<this.items.length; i++)
            document.write(
                "<A href="+this.items[i].link+
                " onmouseover=\"document.images."+this.name+i+".src='"+
                    this.items[i].pic_over.src+"'\")+
                " onmouseout=\"document.images."+this.name+i+".src='"+
                    this.items[i].pic_out.src+"'\")+
                "><IMG name="+this.name+i+" border=0 src="+
                    this.items[i].pic_out.src+
                " width="+this.width+" height="+this.height+
                " alt='"+this.items[i].name+"'\">></A>");
        document.write("</NOBR>");
    }
}

```

Функцию `_Show` можно упростить, если внести «мышинные» атрибуты внутрь тега `<IMG>`:

```

function _Show()
{
    if(this.ismenu)

```

```

{
    document.write("<NOBR>");
    for(var i=0; i<this.len; i++)
        document.write(
            "<A href="+this.items[i].link+
            "><IMG border=0 src="+this.items[i].pic_out.src+
            " width="+this.width+" height="+this.height+
            " alt='"+this.items[i].name+"'" +
            " onmouseover='"+this.src='"+this.items[i].pic_over.src+"'" +
            " onmouseout='"+this.src='"+this.items[i].pic_out.src+"'" +
            "></A>");
    document.write("</NOBR>");
}
}

```

Замечание

Последний вариант не работает в браузерах NN ниже 6 версии. Первый вариант работает, начиная с NN3. Оба варианта нормально работают в IE, начиная с IE4.

Для показа меню можно написать такой HTML-код (листинг 16.7).

Листинг 16.7

```

<HTML>
<HEAD>
<META http-equiv="Content-Type" content="text/html;
    charset=windows-1251">
<TITLE>Меню на графических кнопках</TITLE>
<SCRIPT language="JavaScript" src="menupic.js"></SCRIPT>
<SCRIPT language=JavaScript>
<!--
    var menupic = new MenuPic("menupic", "./pic/key", "gif", 43, 43,
        "Абсолютное позиционирование", "css1.htm",
        "Трехмерный заголовок", "css2.htm",
        "Статические фильтры", "css3.htm",
        "Динамические фильтры", "css4.htm",
        "Сложное движение элементов", "css5.htm",
        "Зеркальное отражение", "css6.htm",

```

```
        "Арканоид", "../css7.htm" );

//-->
</SCRIPT>
</HEAD>
<BODY bgcolor=white text=black>
    <H2>Меню на графических кнопках</H2>
    <HR>
    <SCRIPT language=JavaScript>
    <!--
        menupic.Show();
    //-->
    </SCRIPT>
</BODY>
</HTML>
```

Задания

1. Наберите коды, предложенные Сидоровым, и проверьте их в работе.
2. Создайте меню на графических кнопках, в котором предусмотрена работа с тремя наборами графических файлов, изображающих кнопки в разном «состоянии». Первый набор — неактивные кнопки. Второй набор — вид кнопок, когда над ними расположен курсор. Третий набор — кнопки в момент щелчка мыши. Замена картинок выполняется обработчиками событий: `mouseover` — курсор над кнопкой, `mouseout` — курсор «ушел» с кнопки, `mousedown` — нажата кнопка мыши, `mouseup` — кнопка мыши отпущена.

Пусть разные виды одной кнопки задаются следующим набором файлов (табл. 16.1).

Таблица 16.1

Имя файла	Описание
pic0.gif	Неактивная кнопка
pic1.gif	Над кнопкой расположен курсор мыши
pic2.gif	Кнопка в момент щелчка (имитация нажатия)

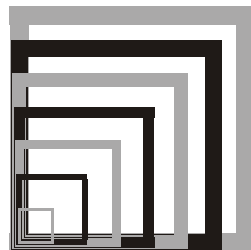
Тогда код, «обслуживающий» эту кнопку, может иметь вид (листинг 16.8).

Листинг 16.8

```
<IMG src=pic0.gif width=27 height=51 border=0 alt="Раздел 1"  
  onmouseover = "this.src='pic1.gif'"  
  onmouseout  = "this.src='pic0.gif'"  
  onmousedown = "this.src='pic2.gif'"  
  onmouseup   = "this.src='pic1.gif'">
```

Для доступа к свойству `src` объекта, построенного для тега `<IMG>`, использована не коллекция `document.images`, как в других кодах этого урока, а указание на текущий объект `this`. Это нормально работает в Internet Explorer и Netscape 6, но, к сожалению, не работает в Netscape 3 и 4.

Урок 17



Роботландский Плюсик

В уроке на примере построения приложения Плюсик показан процесс проектирования и отладки сложных систем, начиная с составления технического задания.

Плюсик

Плюсик — это модель стекового калькулятора. Эта программа входит в состав программно-методической системы «Роботландия». Ее вид показан на рис. 17.1.



Рис. 17.1. Вид Плюсика из пакета «Роботландия»

Среда исполнителя состоит из стека и вычислителя. Стек служит для хранения чисел. Числа помещаются в стек по одному, подобно тому, как кольца

детской пирамидки нанизываются на стержень. За один раз взять из стека можно только одно число с вершины числовой пирамиды. Копия числа в стеке не сохраняется, зато становится доступным следующее число, расположенное «ниже» взятого.

Система команд Плюсика

Система команд исполнителя представлена в табл. 17.1.

Таблица 17.1

Команда	Описание
ЗАПОМНИ <i>число</i>	Число, записанное в команде, отправляется на хранение в стек.
СЛОЖИ	Эти четыре команды работают одинаково: из стека последовательно извлекаются два числа и отправляются в вычислитель для выполнения соответствующей арифметической операции. Результат помещается в стек.
ВЫЧТИ	
УМНОЖЬ	
ДЕЛИ	Второе число извлеченной из стека пары вычислитель считает первым операндом операции. Ситуация отказа возникает, когда: ☐ в стеке меньше двух чисел; ☐ выполняется деление на ноль.

Примеры программ Плюсика.

1. Вычислить: $125,67 \times 82$
 ЗАПОМНИ 125.67
 ЗАПОМНИ 82
 УМНОЖЬ
2. Вычислить: $7253 - (2,356 + 23,78)$
 ЗАПОМНИ 7253
 ЗАПОМНИ 2.356
 ЗАПОМНИ 23.78
 СЛОЖИ
 ВЫЧТИ

Алгоритм написания программ для Плюсика

Для преобразования обычного выражения в программу для Плюсика можно просматривать выражение последовательно слева направо, записывая команды **ЗАПОМНИ** *число*. Если после очередной записи команды **ЗАПОМНИ** вы-

ясняется, что над последними двумя числами в стеке можно выполнить арифметическую операцию, записывается соответствующая команда.

Техническое задание на гипертекстового Плюсика

Верный принципам структурного проектирования, Сидоров начал разработку Плюсика с построения технического задания. Пусть, решил Сидоров, среда исполнителя будет выглядеть так, как на рис. 17.2.

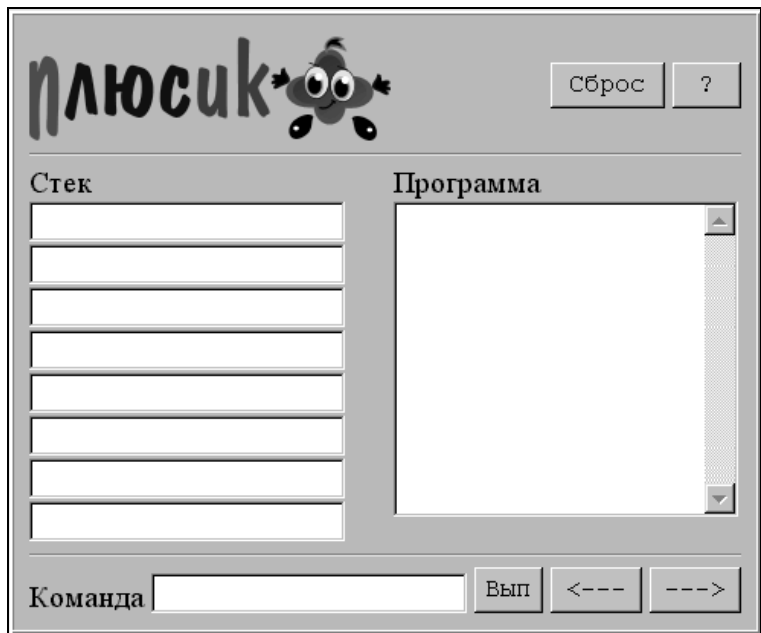


Рис. 17.2. Вид Плюсика на гипертекстовой странице

Пояснения к интерфейсным элементам приведены в табл. 17.2

Таблица 17.2

Элемент интерфейса	Описание
	Кнопка, которая приводит среду исполнителя в исходное состояние: очищает стек, поле ввода команды и поле программы (протокол работы пользователя)
	Кнопка, которая выдает справочную информацию по интерфейсу приложения и системе команд исполнителя

Таблица 17.2 (продолжение)

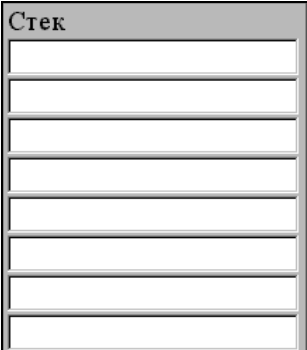
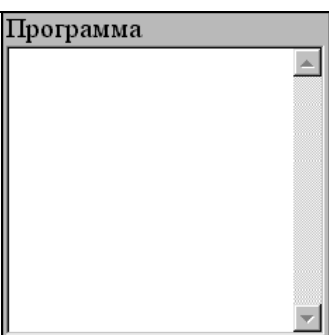
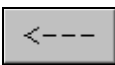
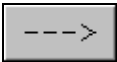
Элемент интерфейса	Описание
	Модель стека исполнителя. На экране будут видны только 8 ячеек (верхушка стека, естественно, верхняя ячейка). Однако ограниченный на максимальную «глубину» стека не предполагается
	Поле протокола. Все команды, вводимые пользователем, будут отображаться в этой области
	Поле для ввода команды исполнителя
	Кнопка, которая заставляет Плюсик выполнить команду, введенную в поле команды. После успешного выполнения поле команды очищается, и копия команды появляется в поле протокола. Если выполнение невозможно, выдается сообщение «Не понимаю» (ошибка в записи команды) или «Не могу» (команду невозможно выполнить). При этом поле команды и поле протокола сохраняют свой прежний вид
	Откатка. Среда возвращается в положение, которое она имела перед выполнением последней команды. Повторные нажатия на эту кнопку приводят к дальнейшей откатке, на два, три и более шагов, вплоть до начального состояния среды

Таблица 17.2 (окончание)

Элемент интерфейса	Описание
	Накатка. Накатка — это движение по истории изменения среды в сторону, противоположную откатке. Если пользователь, выполняя откатку, «зашел» слишком далеко, он всегда может исправить положение, выполняя накатку

Программирование стека

Программирование Сидоров начал с построения стека. Он описал на JavaScript такой объект:

```
//-- Файл stack.js --
//-- Начало описания объекта Stack.
// Конструктор
function Stack()
{
    // Свойства.
    this.stack = new Array(); // Массив stack - модель стека.
    this.top    = 0;           // Индекс "заверхушки" стека (свободного места
                               // в массиве) или - число элементов в стеке.

    // Основные методы.
    this.Put = _Stack_Put;      // Положить на стек.
    this.Pop = _Stack_Pop;      // Взять со стека.

    // Вспомогательные методы
    this.Cls  = _Stack_Cls;     // Очистка стека.
    this.Element = _Stack_Element; // Копия элемента в стеке с
                                   // номером i отверхушки.
}

//-- Описание методов объекта.
// Положить элемент на стек.
// Элемент element располагается на свободном месте массива, а индекс
// "заверхушки" наращивается на 1.
function _Stack_Put(element)
{
    this.stack[this.top++] = element;
}
```

```
// Взять элемент со стека.  
// Если стек не пуст, возвращается верхний элемент, а индекс свободного  
// места уменьшается на 1.  
// Если стек пуст, возвращается аварийное значение null.  
function _Stack_Pop()  
{  
    var ret = null;  
    if (this.top) ret = this.stack[--this.top];  
    return ret;  
}  
// Очистка стека.  
function _Stack_Cls()  
{  
    this.top = 0;  
}  
// Копия элемента в стеке с номером i от верхушки.  
// Если элемента нет, возвращается null.  
function _Stack_Element(i)  
{  
    return (i <= this.top) ? this.stack[this.top - i - 1] : null;  
}  
/-- Конец описания объекта Stack
```

Комментарии к коду объекта *Stack*

1. Стек моделируется при помощи массива `stack`. Переменная `top` указывает на свободное место в массиве. Эту переменную можно рассматривать как число элементов в массиве или как указатель на место, которое будет занимать верхушка стека после добавления очередного элемента. В этом смысле `top` — «заверхушка» стека.
2. Основные методы объекта `Put` и `Pop` реализуют возможные операции над стеком: «положить на стек элемент», «извлечь из стека элемент».
3. Дополнительные методы `Cls` и `Element` реализуют «сервисные» функции объекта, которые не имеют отношения к операциям над стеком, как над информационной структурой, но удобны для нашей программистской «кухни»: сброса среды в исходное состояние и отображения стека на экране.

Испытательный стенд

Для отладки объекта Иван соорудил такой испытательный стенд (листинг 17.1).

Листинг 17.1

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
  <SCRIPT language="JavaScript" src="stack.js"></SCRIPT>
  <TITLE>Отладка стека</TITLE>
  <SCRIPT language=JavaScript>
  <!--
  //-- Код для отладки стека
  var stack = new Stack(); // Создан экземпляр стека.
  var maxlen = 8;         // Число ячеек на экране.
  var basename = "s";     // Основа имени каждой ячейки.
  // Сброс отладочной среды.
  function Ini()
  {
    stack.Cls();
    Draw();
  }
  // Положить элемент в стек.
  function Put(a)
  {
    if (a != "")
    {
      stack.Put(a);
      Draw();
    }
    else alert("Не понимаю!");
  }
  // Взять элемент из стека.
  function Pop()
```

```

{
    if (stack.Pop() == null) alert("Не могу!");
    else Draw();
}
// Отрисовка стека на экране.
function Draw()
{
    // Отрисовка стека.
    for(var i=0; i<maxlen; i++)
    {
        var element = stack.Element(i);
        document.plus[basename+i].value = element == null ? "" : element;
    }
    // Отрисовка пустой командной строки.
    document.plus.com.value="";
    // Вывод значения "заверхушки".
    document.plus.extratop.value=stack.top;
}
//-->
</SCRIPT>
</HEAD>
<BODY bgcolor=#DFD8C5 text=black>
<FORM name=plus>
<TABLE bgcolor=#CFB597 border=2 cellpadding=10>
<TR><TD>
<TABLE border=0 cellpadding=0>
<TR valign=center>
<TD align=left>
<FONT color=red>
<BIG><STRONG>Отладка стека</STRONG></BIG>
</FONT>
</TD>
<TD align=right>
<INPUT type=reset value=Сброс onclick="Ini()">
</TD>
</TR>
<TR>
<TD colspan=2><HR></TD>

```

```

</TR>
<TR>
  <TD valign=top>
    Стек<BR>
    <INPUT type=text name=s0 value="" size=20 readonly><BR>
    <INPUT type=text name=s1 value="" size=20 readonly><BR>
    <INPUT type=text name=s2 value="" size=20 readonly><BR>
    <INPUT type=text name=s3 value="" size=20 readonly><BR>
    <INPUT type=text name=s4 value="" size=20 readonly><BR>
    <INPUT type=text name=s5 value="" size=20 readonly><BR>
    <INPUT type=text name=s6 value="" size=20 readonly><BR>
    <INPUT type=text name=s7 value="" size=20 readonly>
  </TD>
  <TD valign=top align=right>
    Элементов в стеке<BR>
    <INPUT type=text name=extratop value=0 size=20 readonly>
  </TD>
</TR>
<TR>
  <TD colspan=2><HR></TD>
</TR>
<TR>
  <TD colspan=2>
    Элемент <INPUT type=text name=com value="" size=20>
    <INPUT type=button value="В стек"
      onclick="Put (this.form.com.value) ">
    <INPUT type=button value="Из стека" onclick="Pop() ">
  </TD>
</TR>
</TABLE>
</TD></TR>
</TABLE>
</FORM>
</BODY>
</HTML>

```

Вид испытательного стенда показан на рис. 17.3.

Рис. 17.3. Вид испытательного стенда

Комментарии к коду отладочного стенда

1. Функции `Ini`, `Put` и `Pop` являются обработчиками события `click` на кнопках **Сброс**, **В стек** и **Из стека** соответственно. Эти функции сначала выполняют нужную работу над экземпляром `stack` объекта `Stack`, а затем вызывают функцию `Draw`, которая перерисовывает экранное изображение стека.
2. Функция `Put` проверяет на «пусто» элемент, который нужно положить на стек. Если элемент — пустая строка — выдается сообщение «Не понимаю».
3. Функция `Pop` проверяет, не пуст ли стек. Если стек пуст, выдается сообщение «Не могу».
4. Функция `Draw` отображает содержимое стека на экране, очищает строку ввода и выводит значение свойства `top`.

Протокол. Откатка. Накатка

После отладки стека Иван решил написать и отладить поле протокола, откатку и накатку. Фактически он создал в программных кодах всю среду Плюсика. Правда, построенный Плюсик получился очень «ленивый»: все команды он выполняет одинаково — помещает их как текст на свой стек.

При этом исполнитель совершенно не задумывается о том, что же написано в этих «командах».

Следует отметить, что Сидоров поступил очень мудро! Ведь он решил заняться откаткой и накаткой, отладить эти операции, а только затем приступить к программированию кодов, ответственных за выполнение команд Плюсика.

А пока выполнение команды Иван выделил в отдельную функцию `ExecuteCom`. Эта функция должна получать в качестве аргумента ввод пользователя, а в качестве результата выдавать значение одной из трех переменных (табл. 17.3).

Таблица 17.3

Имя переменной	Значение переменной
Ok	Строка «Ok» (команда выполнена успешно)
BadData	Строка «Не понимаю»
CantDo	Строка «Не могу»

Заметьте, что Сидоров использует переменные со значениями, которые никогда не меняются в программе. Фактически это константы. Но специального типа «константа» нет в JavaScript, поэтому приходится константы «замуровывать» в обычные переменные. Можно, конечно, не заводить переменных, а прямо в нужном месте кода записывать текст «Ok» вместо переменной `Ok`, «Не понимаю» вместо `BadData` и «Не могу» вместо `CantDo`. Но такое программирование не для Сидорова! Представьте, что через некоторое время тексты сообщений нужно будет поменять (изменились, например, условия использования Плюсика или язык его интерфейса). Если константы не занесены в переменные (один раз в начале программы!), то придется искать их употребление по всему исходному тексту. Морока! Да и легко допустить ошибку при такой замене.

Для начала вместо реальной функции `ExecuteCom` Иван поставил «заглушку»:

```
function ExecuteCom(str)
{
    stack.Put(str);
    return Ok;
}
```

«Заглушка» просто размещает ввод пользователя на стек и возвращает значение переменной `Ok`.

Протокол

Для поддержки поля протокола Плюсика Сидоров написал такой объект:

```
//-- Начало описания объекта Program.
// Конструктор.
function Program()
{
    // Свойства.
    this.program = new Array(); // Массив строк протокола.
    this.len      = 0;           // Длина протокола.
    this.maxlen   = 0;           // Максимальная длина протокола.
    // Методы.
    this.Add      = _Program_Add;    // Добавить строку.
    this.Undo     = _Program_Undo;   // Откатка.
    this.Redo     = _Program_Redo;   // Накатка.
    this.Cls      = _Program_Cls;    // Сброс протокола.
    this.Element  = _Program_Element; // i-я строка протокола.
}
//-- Описание методов объекта.
// Добавить строку.
function _Program_Add(str)
{
    this.program[this.len++] = str;
    this.maxlen = this.len;
}
// Откатка
function _Program_Undo()
{
    var ret = false;
    if (this.len) {this.len--; ret = true;}
    return ret;
}
// Накатка
function _Program_Redo()
{
    var ret = false;
    if (this.len < this.maxlen) {this.len++; ret = true;}
    return ret;
}
// Сброс протокола
```

```
function _Program_Cls()
{
    this.len = this.maxlen = 0;
}
// Возвращается i-строка протокола. Если строки нет, возвращается null.
function _Program_Element(i)
{
    var ret = null;
    if(0 <= i && i < this.len) ret = this.program[i];
    return ret;
}
//-- Конец описания объекта Program.
```

Обратите внимание: значение переменной `len` совпадает со значением переменной `maxlen` до тех пор, пока не работает откатка. После откатки `len` содержит длину протокола с учетом откатки, а `maxlen` является «упором» для накатки.

Откатка

Откатку Иван реализовал при помощи функции `Undo`:

```
// Пользователь нажал кнопку "<--"
// program - рабочий экземпляр объекта Program.
// stack    - рабочий экземпляр объекта Stack.
function Undo()
{
    if(program.Undo())
    {
        stack.Cls();
        // Выполнить заново все команды до this.len.
        for(var i=0; i<program.len; i++)
            ExecuteCom(program.Element(i));
        // Отрисовать экран.
        Draw();
    }
}
```

Алгоритм работы функции `Undo`:

- ❑ Откатка протокола: `program.Undo()`;
- ❑ Если откатка протокола выполнена:
 - сбросить стек: `stack.Cls()`;

- выполнить заново все команды протокола:
 - ♦ `for (var i=0; i<program.len; i++)`
`ExecuteCom(program.Element(i));`
- отрисовать экран: `Draw();`

Накатка

Накатка реализована при помощи функции `Redo`:

```
// Пользователь нажал кнопку "-->"
// program - рабочий экземпляр объекта Program.
function Redo()
{
    if(program.Redo())
    {
        // Выполнить команду.
        ExecuteCom(program.Element(program.len-1));
        // Отрисовать экран.
        Draw();
    }
}
```

Алгоритм накатки:

- ☐ Накатка протокола: `program.Redo();`
- ☐ Если накатка протокола выполнена:
 - выполнить команду, которая появилась в протоколе после накатки:
 - ♦ `ExecuteCom(program.Element(program.len-1));`
 - отрисовать экран: `Draw();`

Выполнение команды пользователя

Для упрощения ввода команд их имена в СКИ (системе команд исполнителя) сокращены до одной буквы (табл. 17.4).

Таблица 17.4

СКИ Плюсика	Команда
з <i>число</i>	ЗАПОМНИ <i>число</i>
в	ВЫЧТИ
с	СЛОЖИ
у	УМНОЖЬ
д	ДЕЛИ

«Заглушку» ExecuteCom Иван заменил такой функцией:

```
var DoCommand = new Array(DoSave, DoSub, DoAdd, DoMul, DoDiv);
// Выполнение команды пользователя.
function ExecuteCom(str)
{
    // Анализ ввода пользователя.
    var ret = Parse(str);
    // Переключение на выполнение команды.
    if (ret == Ok) ret = DoCommand[command]();
    return ret;
}
```

Сначала вызывается функция Parse. Задача этой функции — провести анализ строки, которую ввел пользователь. Если в строке ошибок нет, функция Parse возвращает значение переменной Ok, если ошибки есть — значение переменной BadData. В случае удачного анализа функция Parse заносит в глобальные переменные command и number нужные значения. В переменную command — одно из чисел 0, 1, 2, 3, 4, соответствующих номеру команды из СКИ Плюсика (глобальный массив ski). В переменную number — заносит число — параметр команды з.

Если функция Parse вернула значение Ok, вызывается одна из функций (табл. 17.5).

Таблица 17.5

Функция	Действие: выполняет команду
DoSave()	з число
DoSub()	в
DoAdd()	с
DoMul()	у
DoDiv()	д

Реализации этих функций весьма просты. Ниже приводятся соответствующие коды:

```
// stack - рабочий экземпляр объекта Stack.

// Команда "з".
function DoSave()
```

```
{
    stack.Put(number);
    return Ok;
}
// Команда "в"
function DoSub()
{
    var ret= CantDo;
    if(stack.top > 1)
    {
        var a = stack.Pop();
        var b = stack.Pop();
        stack.Put(b-a);
        ret = Ok;
    }
    return ret;
}
// Команда "с"
function DoAdd()
{
    var ret= CantDo;
    if(stack.top > 1)
    {
        var a = stack.Pop();
        var b = stack.Pop();
        // Используем функцию parseFloat чтобы работала операция сложения
        // чисел, а не конкатенация двух строк.
        stack.Put(parseFloat(b)+parseFloat(a));
        ret= Ok;
    }
    return ret;
}
// Команда "у"
function DoMul()
{
    var ret= CantDo;
    if(stack.top > 1)
```

```
{
    var a = stack.Pop();
    var b = stack.Pop();
    stack.Put(b*a);
    ret= Ok;
}
return ret;
}
// Команда "д"
function DoDiv()
{
    var ret= CantDo;
    if(stack.top > 1)
    {
        var a = stack.Pop();
        if(a == 0) stack.Put(a);
        else
        {
            var b = stack.Pop();
            stack.Put(b/a);
            ret= Ok;
        }
    }
    return ret;
}
```

Анализатор *Parse*

Функция `Parse` проверяет строку, введенную пользователем, на правильность. Проверка выполняется в процессе «расщепления» строки на составляющие: имя команды и возможный параметр (*число* — для команды **з**). Вот код этой функции:

```
// Анализ ввода пользователя.
function Parse(str)
{
    var ret = Ok; // Возвращаемое значение.
    var words = new Words(str); // "Извлекатель слов" из строки.
    var word = ""; // Очередное слово из str.
```

```

// Первое слово из str -- имя команды.
word = words.GetWord();
if(word == "") ret = BadData;
else
{
    // Определим имя команды.
    for(command=0; command<ski.length; command++)
        if(word.toLowerCase() == ski[command]) break;
    if(command >= ski.length) ret = BadData;
    else
    {
        // Если это первая команда ("з"), то должен быть параметр - число.
        if(command==0)
        {
            word = words.GetWord();
            if(!word || isNaN(word)) ret = BadData;
            else number = word;
        }
        if (ret == Ok)
        {
            // За командой не должно быть ничего лишнего.
            word = words.GetWord();
            if (word != "") ret = BadData;
        }
    }
}
return ret;
}

```

Функция `Parse` использует в своей работе экземпляр объекта `Words`, который Сидоров придумал и написал специально для разбора строки на элементы (слова), отделенные друг от друга пробелами. «Разбираемая» на составные части строка передается экземпляру объекта при его создании как аргумент конструктора. Единственный метод объекта `GetWord` позволяет при каждом новом запуске возвращать очередное слово строки (последовательность символов, не содержащую пробелы). Если слов в строке больше нет, метод `GetWord` возвращает пустую строку.

Коды объекта `Words` приводятся ниже:

```

/-- Начало описания объекта Words.
// Объект позволяет извлекать из переданной ему строки слова,

```

```
// т. е. последовательности символов от пробела (начала строки)
// до пробела (конца строки).
// Конструктор
// Аргумент конструктора - строка.
function Words(str)
{
    // Свойства.
    this.str = str;           // Анализируемая строка.
    this.begin = 0;           // Индекс начала слова в строке.
    this.end = 0;             // Индекс "законца" слова в строке.
    // Методы
    this.GetWord = _Words_GetWord // Выдать очередное слово строки.
}
//-- Описание методов объекта
// Получить очередное слово строки.
function _Words_GetWord()
{
    var word="";
    // Началом разбора строки является предыдущий "законец" слова.
    // Пропустим первые пробелы.
    for(this.begin=this.end;
        this.begin<this.str.length; this.begin++)
        if(this.str.charAt(this.begin) != " ") break;
    // Если остаток строки не пуст.
    if(this.begin < this.str.length)
    {
        // Найдем первый пробел за концом слова или конец строки.
        for(this.end=this.begin+1; this.end < this.str.length; this.end++)
            if(this.str.charAt(this.end) == " ") break;
        // Копируем в word найденное слово.
        word = this.str.substring(this.begin, this.end);
    }
    // Возвращаем найденное слово.
    return word;
}
//-- Конец описания объекта Words
```

Анализатор `Parse` можно упростить, если использовать для «расщепления» строки на «слова» метод `split` объекта `String` и механизм *регулярных выражений* JavaScript:

```
// Анализ ввода пользователя.
```

```
function Parse(str)
{
    var ret = Ok;

    // Преобразование строки в массив.
    // Строка расщепляется на слова (разделитель слов - цепочка пробелов),
    // которые заносятся в массив setwords.
    var setwords = str.split(/ +/);

    // Первое слово из str - имя команды.
    if(!setwords.length || setwords[0] == "") ret = BadData;
    else
    {
        // Определим имя команды.
        for(command=0; command<ski.length; command++)
            if(setwords[0].toLowerCase() == ski[command]) break;
        if(command >= ski.length) ret = BadData;
        else
        {
            // Если это первая команда ("з"), то должен быть параметр - число.
            if(!command)
            {
                if(setwords.length != 2 || isNaN(setwords[1])) ret = BadData;
                else number = setwords[1];
            }
            else
            {
                if(setwords.length != 1) ret = BadData;
            }
        }
    }
    return ret;
}
```

В этом варианте функции `Parse` объект `Words` уже не нужен.

Регулярное выражение `/ +/` задает в качестве разделителя элементов для функции `split` один или несколько пробелов (свойство метасимвола «+»). Вариант `str.split(" ")` в качестве замены `str.split(/ +/)` не годится — в нем разделителем является одиночный пробел, что приведет к нежелательному для нас преобразованию в массив, например, такой строки: "э 1". Первый элемент массива будет содержать букву «э», а второй — пустую строку.

Подробное рассмотрение регулярных выражений выходит за рамки этой книги.

Плюсик работает!

Ленивый Иван нагрузил клавиши `<Enter>` и `<F1>` в строке ввода функциями экранных кнопок **Вып** и **?**, а аккорды `<Ctrl>+<Z>` и `<Ctrl>+<Y>` закрепил за операциями «откатка» и «накатка». Теперь стало возможно работать с Плюсиком, отложив мышку в сторону. Иван провел тщательное тестирование своей программы, после чего пошел пить чай с сухариками — программирование всегда вызывало у него жажду. И он не давал себе «засохнуть»!

Ниже приводится полный код исполнителя Плюсик (листинг 17.2).

Листинг 17.2

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
  <META http-equiv="author" content="Иван Сидоров">
  <TITLE>Плюсик</TITLE>
  <SCRIPT language=JavaScript>
    <!--
/*===== Описание объектов программы=====*/
//===== Начало описания объекта Stack =====
// Конструктор
//-- Начало описания объекта Stack.
// Конструктор
function Stack()
{
  // Свойства.
  this.stack = new Array(); // Массив stack - модель стека.
  this.top   = 0;           // Индекс "заверхушки" стека (свободного места
                             // в массиве) или - число элементов в стеке.
```

```
// Основные методы.
this.Put = _Stack_Put;           // Положить на стек.
this.Pop = _Stack_Pop;          // Взять со стека.
// Вспомогательные методы
this.Cls = _Stack_Cls;          // Очистка стека.
this.Element = _Stack_Element;  // Копия элемента в стеке с
                                // номером i от верхушки.
}
//-- Описание методов объекта.
// Положить элемент на стек.
// Элемент element располагается на свободном месте массива, а индекс
// "заверхушки" наращивается на 1.
function _Stack_Put(element)
{
    this.stack[this.top++] = element;
}
// Взять элемент со стека.
// Если стек не пуст, возвращается верхний элемент, а индекс свободного
// места уменьшается на 1.
// Если стек пуст, возвращается аварийное значение null.
function _Stack_Pop()
{
    var ret = null;
    if (this.top) ret = this.stack[--this.top];
    return ret;
}
// Очистка стека.
function _Stack_Cls()
{
    this.top = 0;
}
// Копия элемента в стеке с номером i от верхушки.
// Если элемента нет, возвращается null.
function _Stack_Element(i)
{
    return (i <= this.top) ? this.stack[this.top - i - 1] : null;
}
```

```
//-- Конец описания объекта Stack
//===== Конец описания объекта Stack =====

//===== Начало описания объекта Program =====
// Конструктор
function Program()
{
    // Свойства.
    this.program = new Array(); // Массив строк протокола.
    this.len      = 0;           // Длина протокола.
    this.maxlen   = 0;           // Максимальная длина протокола.
    // Методы.
    this.Add      = _Program_Add; // Добавить строку.
    this.Undo     = _Program_Undo; // Откатка.
    this.Redo     = _Program_Redo; // Накатка.
    this.Cls      = _Program_Cls; // Сброс протокола.
    this.Element  = _Program_Element; // i-я строка протокола.
}
//-- Описание методов объекта.
// Добавить строку.
function _Program_Add(str)
{
    this.program[this.len++] = str;
    this.maxlen = this.len;
}
// Откатка
function _Program_Undo()
{
    var ret = false;
    if (this.len) {this.len--; ret = true;}
    return ret;
}
// Накатка
function _Program_Redo()
{
    var ret = false;
    if (this.len < this.maxlen) {this.len++; ret = true;}
    return ret;
}
```

```

// Сброс протокола
function _Program_Cls()
{
    this.len = this.maxlen = 0;
}
// Возвращается i-строка протокола. Если строки нет, возвращается null.
function _Program_Element(i)
{
    var ret = null;
    if(0 <= i && i < this.len) ret = this.program[i];
    return ret;
}
//===== Конец описания объекта Program =====

//===== Начало описания объекта Words =====
// Объект позволяет извлекать из переданной ему строки слова,
// т. е. последовательности символов от пробела (начала строки)
// до пробела (конца строки).
// Конструктор
// Аргумент конструктора -- строка
function Words(str)
{
    // Свойства.
    this.str    = str;        // Анализируемая строка.
    this.begin  = 0;          // Индекс начала слова в строке.
    this.end    = 0;          // Индекс "законца" слова в строке.
    // Методы
    this.GetWord = _Words_GetWord // Выдать очередное слово строки.
}
//-- Описание методов объекта
// Получить очередное слово строки.
function _Words_GetWord()
{
    var word="";
    // Началом разбора строки является предыдущий "законец" слова.
    // Пропустим первые пробелы.
    for(this.begin=this.end;
        this.begin<this.str.length; this.begin++)
        if(this.str.charAt(this.begin) != " ") break;

```

```

// Если остаток строки не пуст.
if(this.begin < this.str.length)
{
    // Найдем первый пробел за концом слова или конец строки.
    for(this.end=this.begin+1; this.end < this.str.length; this.end++)
        if(this.str.charAt(this.end) == " ") break;
    // Копируем в word найденное слово.
    word += this.str.substring(this.begin, this.end);
}
// Возвращаем найденное слово.
return word;
}

//===== Конец описания объекта Words =====

/*=====Кодирование среды Плюсика =====*/
//===== Глобальные переменные
var stack    = new Stack();    // Модель стека Плюсика.
var program = new Program(); // Модель программного поля Плюсика.
var lenStackScreen = 8;       // Число ячеек стека Плюсика на экране.
var lenProgScreen  = 10;      // Число строк программы Плюсика на экране.
var basename = "s";          // Основа имени каждой ячейки стека.
var Ok        = "Ok";        // Команда выполнена успешно.
var BadData   = "Не понимаю"; // Команда не входит в СКИ Плюсика.
var CantDo    = "Не могу";    // Команду выполнить невозможно.
var ski       = new Array(    // Система команд Плюсика.
    "з",
    "в",
    "с",
    "у",
    "д"
);
var legends = new Array(      // Надписи на экране Плюсика.
    "Команда",
    "Стек",
    "Программа",
    "Сброс",
    "?",

```

```

        "Вып",
        "<--",
        "-->"
    );

// Текст "Помощи".
var helptxt = "Исполнитель Плюсик\n\n"+
    "Система команд Плюсика:\n"+
    ski[0]+" <число> -- поместить число в стек\n"+
    ski[1]+" -- вычесть\n"+
    ski[2]+" -- сложить\n"+
    ski[3]+" -- умножить\n"+
    ski[4]+" -- делить\n\n"+
    "Среда:\n"+
    legends[0]+" " -- поле для ввода команды\n"+
    legends[1]+" " -- стек\n"+
    legends[2]+" " -- поле протокола\n\n"+
    "Управление:\n"+
    "/" + legends[3] + "/" -- среда в начальное положение\n"+
    "/" + legends[4] + "/" или F1 -- справка\n"+
    "/" + legends[5] + "/" или Enter -- выполнить команду\n"+
    "/" + legends[6] + "/" или Ctrl+Z -- откатка\n"+
    "/" + legends[7] + "/" или Ctrl+Y -- накатка";

// Коды клавиш
var key_ret = 13;
var key_z = 122;
var key_Z = 90;
var key_rus_z = 1103;
var key_rus_Z = 1071;
var key_y = 121;
var key_Y = 89;
var key_rus_y = 1085;
var key_rus_Y = 1053;
// Рабочие переменные.
var command; // Индекс команды в ski.
var number; // Число-параметр команды "з".
    //===== Конец описания глобальных переменных

// Пользователь нажал кнопку "Сброс".

```

```
function Ini ()
{
    stack.Cls();
    program.Cls();
    // Отрисовать экран
    Draw();
}
// Пользователь нажал кнопку "?"
function Help()
{
    alert(helptxt);
}
// Пользователь нажал кнопку "Вып"
function Execute(str)
{
    if(str != "")
    {
        var mess = ExecuteCom(str);
        if (mess==Ok)
        {
            program.Add(str);
            // Отрисовать экран
            Draw();
        }
        else alert(mess);
    }
    else alert(BadData);
}
var DoCommand = new Array(DoSave, DoSub, DoAdd, DoMul, DoDiv);
// Выполнение команды пользователя.
function ExecuteCom(str)
{
    // Анализ ввода пользователя.
    var ret = Parse(str);
    // Переключение на выполнение команды.
    if(ret == Ok) ret = DoCommand[command]();
    return ret;
}
```

// Анализ ввода пользователя.

```
function Parse(str)
{
    var ret = Ok;                // Возвращаемое значение.
    var words = new Words(str);  // "Извлекатель слов" из строки.
    var word = "";               // Очередное слово из str.
    // Первое слово из str -- имя команды.
    word = words.GetWord();
    if(word == "") ret = BadData;
    else
    {
        // Определим имя команды.
        for(command=0; command<ski.length; command++)
            if(word.toLowerCase() == ski[command]) break;
        if(command >= ski.length) ret = BadData;
        else
        {
            // Если это первая команда ("з"), то должен быть параметр - число.
            if(command==0)
            {
                word = words.GetWord();
                if(!word || isNaN(word)) ret = BadData;
                else number = word;
            }
            if (ret == Ok)
            {
                // За командой не должно быть ничего лишнего.
                word = words.GetWord();
                if (word != "") ret = BadData;
            }
        }
    }
    return ret;
}

// Выполнение команд из СКИ.
// stack - рабочий экземпляр объекта Stack.

// Команда "з".
function DoSave()
```

```
{
    stack.Put(number);
    return Ok;
}
// Команда "в"
function DoSub()
{
    var ret= CantDo;
    if(stack.top > 1)
    {
        var a = stack.Pop();
        var b = stack.Pop();
        stack.Put(b-a);
        ret= Ok;
    }
    return ret;
}
// Команда "с"
function DoAdd()
{
    var ret= CantDo;
    if(stack.top > 1)
    {
        var a = stack.Pop();
        var b = stack.Pop();
        stack.Put(parseFloat(b)+parseFloat(a));
        ret= Ok;
    }
    return ret;
}
// Команда "у"
function DoMul()
{
    var ret= CantDo;
    if(stack.top > 1)
    {
        var a = stack.Pop();
        var b = stack.Pop();
        stack.Put(b*a);
    }
}
```

```
        ret= Ok;
    }
    return ret;
}
// Команда "д"
function DoDiv()
{
    var ret= CantDo;
    if(stack.top > 1)
    {
        var a = stack.Pop();
        if(a == 0) stack.Put(a);
        else
        {
            var b = stack.Pop();
            stack.Put(b/a);
            ret= Ok;
        }
    }
    return ret;
}
// Пользователь нажал кнопку "<--".
function Undo()
{
    if(program.Undo())
    {
        stack.Cls();
        // выполнить заново все команды до this.len.
        for(var i=0; i<program.len; i++)
            ExecuteCom(program.Element(i));
        // Отрисовать экран
        Draw();
    }
}
// Пользователь нажал кнопку "-->"
function Redo()
{
    if(program.Redo())
```

```

{
    // Выполнить команду.
    ExecuteCom(program.Element(program.len-1));
    // Отрисовать экран.
    Draw();
}
}
// Отрисовка экрана.
function Draw()
{
    var i;
    var element;
    var str;
    // Отрисовка стека
    for(i=0; i<lenStackScreen; i++)
    {
        str ="document.plus."+basename+i+".value=";
        element = stack.Element(i);
        if(element != null) eval(str+"element");
        else eval(str+"");
    }
    // Отрисовка программы
    str="";
    for(i=0; i<program.len; i++)
        str += eval(i+1)+". "+program.program[i)+"\r\n";
    document.plus.prog.value=str;
    document.plus.prog.scrollTop=program.len>lenProgScreen ?
        (program.len-lenProgScreen)*document.plus.prog.scrollHeight/
        lenProgScreen : 0;
    // Отрисовка пустой командной строки.
    document.plus.com.value="";
}

//-->
</SCRIPT>

</HEAD>

<BODY bgcolor=#DFD8C5 text=black onload=Ini()>
    <FORM name=plus>
        <TABLE bgcolor=#CFB597 border=2 cellpadding=0 cellspacing=0>
            <TR><TD>

```

```

<TABLE border=0 cellspacing=0 cellpadding=0>
  <TR valign=center>
    <TD align=left>
      <IMG src=./pic/plus.gif align=middle width=237 height=73
        border=0 alt="">
    </TD>
    <TD align=right>
      <SCRIPT language=JavaScript>
        <!-- // Кнопки "Сброс", "?"
        document.write("<NOBR><INPUT type=reset value='"+ legends[3]+
          "' onclick=' Ini ()'>" +
          " <INPUT type=button value='"+ legends[4]+
          "' onclick=' Help()'></NOBR>");

        //-->
      </SCRIPT>
    </TD>
  </TR>
  <TR>
    <TD colspan=2><HR></TD>
  </TR>
  <TR>
    <TD valign=top>
      <SCRIPT language=JavaScript>
        <!-- // Стек
        document.write(legends[1]);

        //-->
      </SCRIPT>
      <BR>
      <INPUT type=text name=s0 value="" size=22 readonly><BR>
      <INPUT type=text name=s1 value="" size=22 readonly><BR>
      <INPUT type=text name=s2 value="" size=22 readonly><BR>
      <INPUT type=text name=s3 value="" size=22 readonly><BR>
      <INPUT type=text name=s4 value="" size=22 readonly><BR>
      <INPUT type=text name=s5 value="" size=22 readonly><BR>
      <INPUT type=text name=s6 value="" size=22 readonly><BR>
      <INPUT type=text name=s7 value="" size=22 readonly>
    </TD>
    <TD valign=top>

```

```

        <SCRIPT language=JavaScript>
        <!-- // Программа
            document.write(legends[2]);
        //-->
        </SCRIPT>
        <BR>
        <TEXTAREA name=prog rows=10 cols=25 readonly>
        </TEXTAREA>
    </TD>
</TR>
<TR>
    <TD colspan=2><HR></TD>
</TR>
<TR>
    <TD colspan=2>
        <SCRIPT language=JavaScript>
        <!-- // Команда
            document.write(legends[0]);
        //-->
        </SCRIPT>
        <INPUT type=text name=com value="" size=22 maxlength=20
            onhelp="event.returnValue=false; Help();"
            onkeypress="if(event.keyCode==key_ret)
                Execute(this.form.com.value)"
            onkeydown="if(event.ctrlKey)
                {
                    var flag=false;
                    // Ctrl+Z (откатка) ?
                    if(event.keyCode==key_z      ||
                       event.keyCode==key_Z      ||
                       event.keyCode==key_rus_z ||
                       event.keyCode==key_rus_Z)
                    {Undo(); flag=true;}
                    // Ctrl+Y (накатка) ?
                    else if(event.keyCode==key_y      ||
                           event.keyCode==key_Y      ||
                           event.keyCode==key_rus_z ||
                           event.keyCode==key_rus_Z)

```

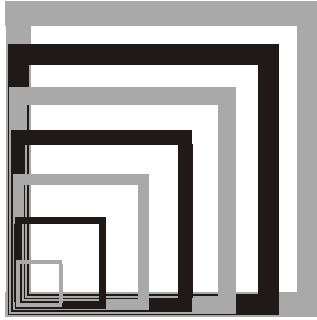
```

        {Redo(); flag=true;}
        if(flag)
        {
            event.cancelBubble=true;
            event.returnValue=false;
            return false;
        }
    }">

<SCRIPT language=JavaScript>
<!-- // Кнопки "Вып", "<---", "--->"
document.write("<NOBR><INPUT type=button value='"+ legends[5]+
    "' onclick='Execute(this.form.com.value)'">"+
    " <INPUT type=button value='"+ legends[6]+
    "' onclick='Undo()'">"+
    " <INPUT type=button value='"+ legends[7]+
    "' onclick='Redo()'"></NOBR>");

    //-->
</SCRIPT>
</TD>
</TR>
</TABLE>
</TD></TR>
</TABLE>
</FORM>
</BODY>
</HTML>

```



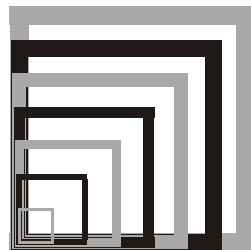
Часть V

CSS + JavaScript

Урок 18. Программное управление стилями

Урок 19. Управление содержимым страницы

Урок 18



Программное управление стилями

Принцип программного управления стилями

Когда для тега HTML применяется стиль CSS, то это отражается на объектной модели документа. Браузер предусматривает для каждого объекта-тега свойство `style`, которое само является объектом и содержит информацию о стилях тега. Эту информацию можно читать и менять, а значит, — влиять на внешнее представление (или положение) элемента на экране.

Доступ к элементам стиля

Возникает законный вопрос: как добраться до свойства `style` объекта, построенного браузером для тега? Один из способов — использовать коллекцию `document.all`.

Коллекция — это массив. Коллекция `all` содержит объекты, построенные для всех тегов документа. Так, `document.all[0]` — это объект, построенный для первого тега документа, `document.all[1]` — объект, моделирующий второй тег и т. д. Каждый элемент коллекции `all` имеет свойство `tagName` — имя тега, которому соответствует объект.

Расположенный ниже код показывает в окошке `alert` свои теги (листинг 18.1).

Листинг 18.1

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
```

```

<TITLE>Коллекция all</TITLE>
<SCRIPT language=JavaScript>
<!--
    function show()
    {
        var str="Список тегов документа:\n";
        for(var i=0; i<document.all.length; i++)
            str += "all["+i+"] - "+document.all[i].tagName+"\n";
        alert(str);
    }
//-->
</SCRIPT>
</HEAD>
<BODY bgcolor=white text=black onclick="show()">
    <H2>Коллекция all</H2>
    <HR>
    <P>

```

Для просмотра списка имен тегов этой программы щелкните мышью по документу.

```

</BODY>
</HTML>

```

Работа этой программы показана на рис. 18.1.

Замечание

На самом деле коллекция содержит не сами объекты, а ссылки на них, но именно ссылки и нужны для доступа к объектам в программных кодах.

Обращаться по индексу к элементу коллекции `all` не слишком удобно. Во-первых, этот индекс не хочется высчитывать «вручную», во-вторых, он меняется, если в документ помещаются новые теги. Последняя неприятность делает использование числового индекса совершенно невозможным. К счастью, в качестве индекса можно использовать имя или идентификатор тега. Имя мы использовали и раньше, оно вводится для тега атрибутом `name`. Идентификатор, играющий практически ту же роль, что и имя, задается атрибутом `id`.

Пусть, например, в HTML-коде задан тег:

```

<IMG id=pic src=pic/explorer.gif width=125 height=82 alt=""
    onclick="show()">

```

Добаться до соответствующего объекта, можно, заменив индекс элемента коллекции `all` на имя, заданное `id`-атрибутом: `document.all["pic"]` (используется и эквивалентная запись `document.all.pic`).

Теперь в обработчике `show` можно менять свойства картинки, например, ее размер. Именно так и сделано в следующем примере (листинг 18.2).

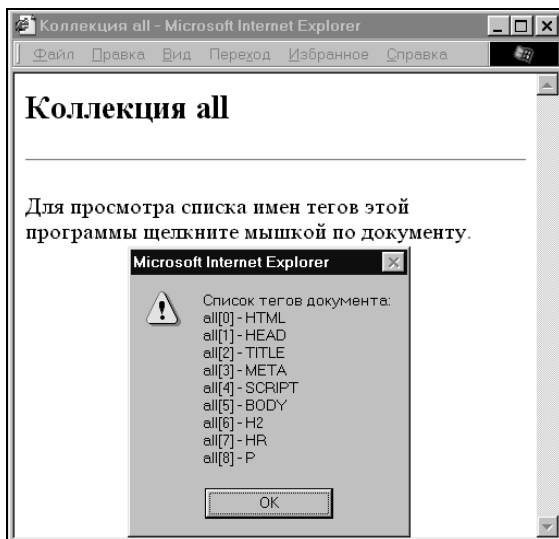


Рис. 18.1. Все теги документа

Листинг 18.2

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
  <TITLE>Доступ к элементу all по имени</TITLE>
  <SCRIPT language=JavaScript>
  <!--
    var size=true;
    function show()
    {
      if(size)
      {
        document.all["pic"].width  *= 2;
        document.all["pic"].height *= 2;
      }
    }
  </SCRIPT>
</HTML>
```

```

else
{
    document.all["pic"].width /= 2;
    document.all["pic"].height /= 2;
}
size = !size;
}
//-->
</SCRIPT>
</HEAD>
<BODY bgcolor=white text=black>
    <H2>Доступ к элементу all по имени</H2>
    <HR>
    <IMG id=pic src=pic/explorer.gif width=125 height=82 alt=""
        onclick="show()">
    <P>Щелкните по картинке мышью.
</BODY>
</HTML>

```

На рис. 18.2 показан вид этого документа после загрузки, а на рис. 18.3 — после первого щелчка по картинке. Если продолжать щелчки, изображения будут чередоваться.

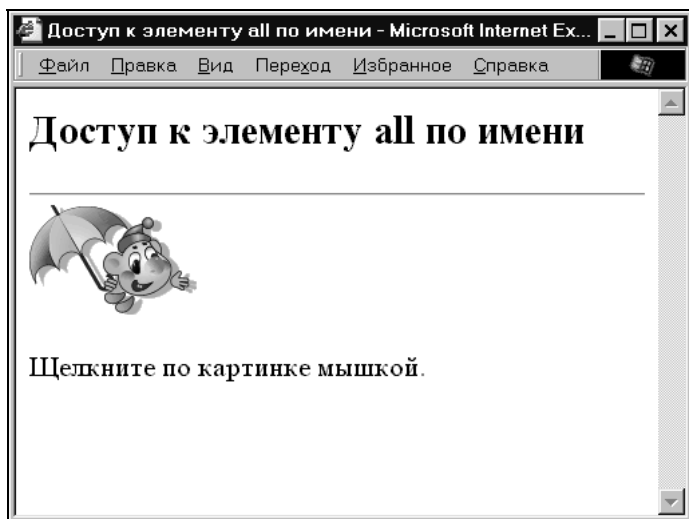


Рис. 18.2. Вид документа после «четного» щелчка

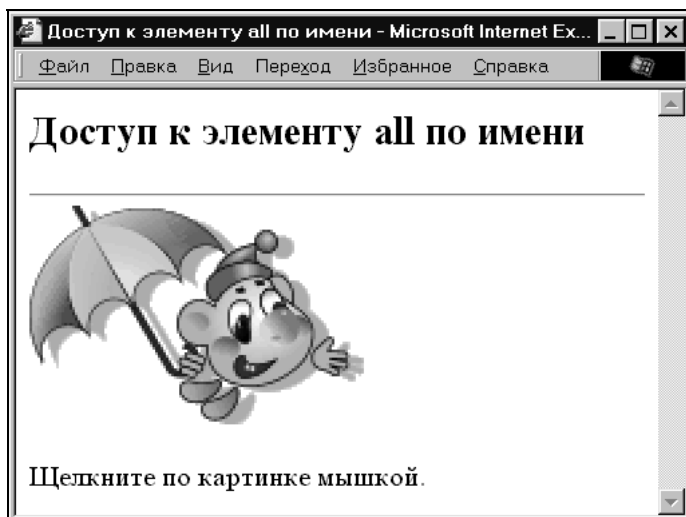


Рис. 18.3. Вид документа после «нечетного» щелчка

Нас интересует свойство `style`. Добраться до него можно по схеме `document.all["имя"].style` (или `document.all.имя.style`).

Построение динамического меню

Посмотрите на следующий HTML-код:

```
<UL type=disc>
  <LI onclick="go(0)">Раздел 0
  <LI onclick="go(1)">Раздел 1
  <LI onclick="go(2)">Раздел 2
</UL>
```

Ничего особенного! Обычный список с обработкой события `click` на каждом из трех строк списка. Функция `go` может выполнять какие-то действия, например, загружать на экран новые страницы. Ограничим пока работу функции `go` простым `alert`-сообщением (рис. 18.4).

Меню работает, но выглядит не очень симпатично. Хочется, чтобы при «наезде» курсором на строку, она инвертировала свои цвета. Давайте применим CSS! Будем отслеживать события `mouseover` (курсор мыши появился над элементом) и `mouseout` (курсор мыши ушел с элемента) и соответственно менять стиль этого элемента. Теперь список будет иметь вид:

```
<UL type=disc>
  <LI id=item0 onclick="go(0)"
    onmouseover="over(0)" onmouseout="out(0)">Раздел 0
```

```

<LI id=item1 onclick="go(1)"
    onmouseover="over(1)" onmouseout="out(1)">Раздел 1
<LI id=item2 onclick="go(2)"
    onmouseover="over(2)" onmouseout="out(2)">Раздел 2
</UL>

```

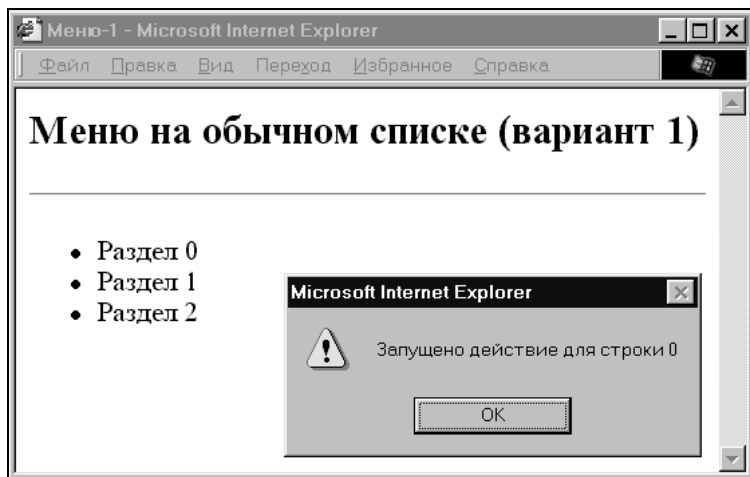


Рис. 18.4. Первый вариант меню на маркированном списке

Функция `over` должна менять стиль строки списка таким образом, чтобы он был написан белым по черному, а функция `out` должна обеспечить обычный черный текст по белому фону.

Добаться до нужных свойств первой строки списка достаточно просто:

`document.all["item0"].style.color` — цвет текста;

`document.all["item0"].style.background` — цвет фона.

Функции `over` и `out` можно записать так:

```

function over(i)
{
    document.all["item"+i].style.color="white";
    document.all["item"+i].style.background="black";
}
function out(i)
{
    document.all["item"+i].style.color="black";
    document.all["item"+i].style.background="white";
}

```

В качестве индекса коллекции `all` используется выражение `"item" + i`. Для `i = 0` получится «item0», для `i = 1` — «item1», для `i = 2` — «item2», т. е. как раз то, что нам надо.

Определим для тега `<LI>` начальный стиль:

```
LI { color:black; background:white; }
```

Что у нас получилось, можно посмотреть на рис. 18.5. Код этой страницы представлен ниже (листинг 18.3):

Листинг 18.3

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
  <TITLE>Меню-2</TITLE>
  <STYLE type="text/css">
    <!--
      LI {color:black; background:white;}
    -->
  </STYLE>
  <SCRIPT language=JavaScript>
    <!--
      // Запуск действия, соответствующего i-й строке меню.
      function go(i)
      {
        alert("Запущено действие для строки "+i);
      }
      // Курсор сместился на i-ю строку меню
      function over(i)
      {
        document.all["item"+i].style.color="white";
        document.all["item"+i].style.background="black";
      }
      function out(i)
      {
        document.all["item"+i].style.color="black";
        document.all["item"+i].style.background="white";
      }
    //-->
```

```

</SCRIPT>
</HEAD>
<BODY bgcolor=white text=black>
  <H2>Меню на обычном списке (вариант 2)</H2>
  <HR>
  <UL type=disc>
    <LI id=item0 onclick="go(0)"
      onmouseover="over(0)" onmouseout="out(0)">Раздел 0
    <LI id=item1 onclick="go(1)"
      onmouseover="over(1)" onmouseout="out(1)">Раздел 1
    <LI id=item2 onclick="go(2)"
      onmouseover="over(2)" onmouseout="out(2)">Раздел 2
  </UL>
</BODY>
</HTML>

```

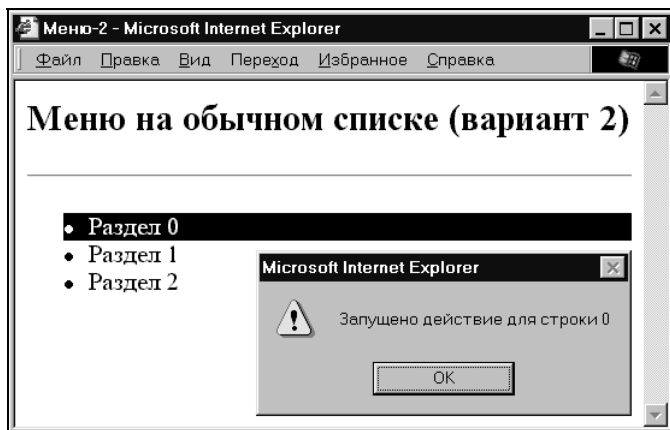


Рис. 18.5. Второй вариант меню на маркированном списке

Меню работает! Но почему-то выделение занимает всю ширину строки, а это не очень красиво. Так получается потому, что в стиле не задана ширина строк, а, значит, она наследуется от родителя — тега **<BODY>** и, тем самым, равна ширине окна браузера. Попытка задать ширину в описании стиля для тега **** терпит неудачу. Дело в том, что стилевое свойство `width` поддерживается только тегами **<DIV>** и ****. Придется поместить список внутри тега **<DIV>**:

```

<DIV style="width:200px">
  <UL type=disc>
    <LI id=item0 onclick="go(0)"

```

```
onmouseover="over(0) " onmouseout="out(0) ">Раздел 0
<LI id=item1 onclick="go(1) "
onmouseover="over(1) " onmouseout="out(1) ">Раздел 1
<LI id=item2 onclick="go(2) "
onmouseover="over(2) " onmouseout="out(2) ">Раздел 2
</UL>
</DIV>
```

И последний штрих: укажем в стиле для тега `<LI>` форму курсора такую, как на ссылке:

```
LI { color:black; background:white; cursor:hand}
```

Результат наших усилий показан на рис. 18.6.

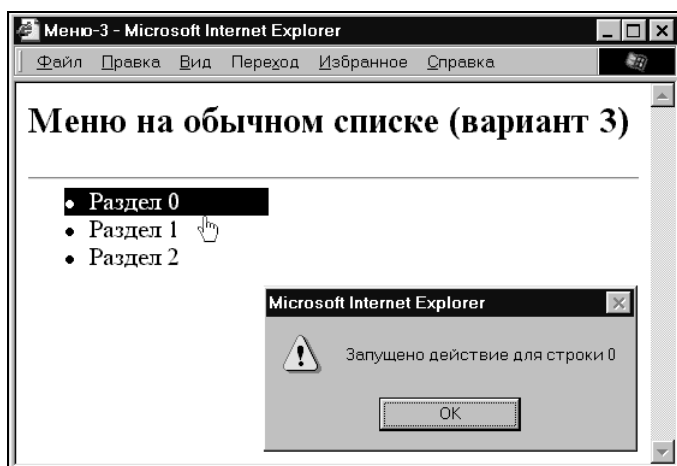


Рис. 18.6. Третий вариант меню на маркированном списке

Движение на экране

В следующем документе картинка (человечек с зонтиком) начинает двигаться по экрану сразу после загрузки документа (рис. 18.7).

Так происходит потому, что в теге `<BODY>` записан атрибут `onload`. После полной загрузки документа, выполнится JavaScript-выражение, указанное в качестве значения этого атрибута: `"tim=setInterval('Move()',20)"`.

Функция `setInterval('Move()',20)` создаст таймер, который будет запускать функцию `Move()` каждые 20 миллисекунд (более подробно о программировании таймеров см. в *Приложениях*).

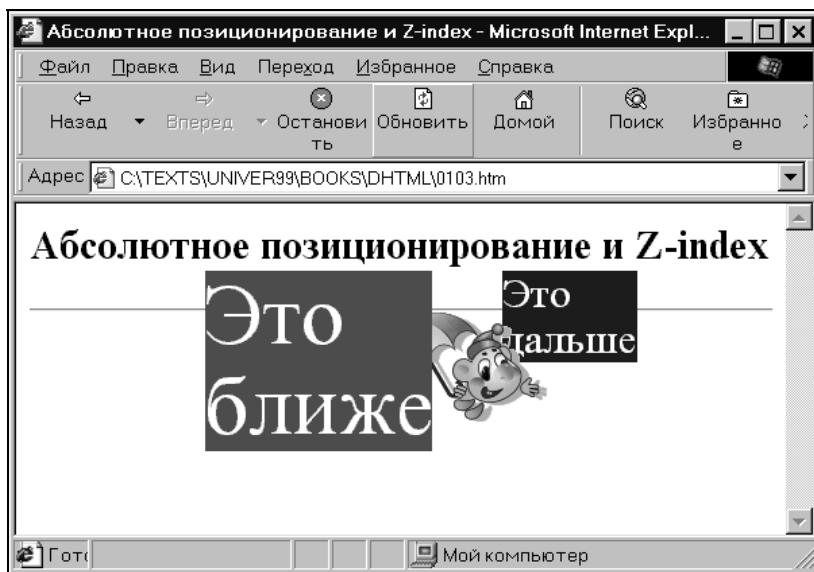


Рис. 18.7. Движение картинки по экрану

Функция `Move()` увеличивает горизонтальную координату картинки на два пиксела, если эта координата меньше 500. В противном случае функция останавливает таймер:

```
function Move()
{
    var obj = document.all["move"].style;
    if (obj.posLeft < 500) obj.posLeft +=2;
    else clearInterval(tim);
}
```

Замечание

Заметьте, что вместо стилевого свойства `left` используется свойство `posLeft`. Эти свойства совершенно равнозначны, но первое хранится с окончанием «px» (например, «200px»), и, тем самым, не удобно для использования в арифметических вычислениях, второе — обычное число.

Вот полный код приведенного выше документа (листинг 18.4).

Листинг 18.4

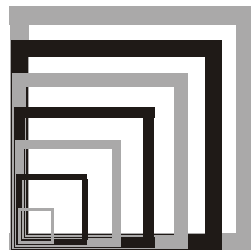
```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
```

```
<TITLE>Абсолютное позиционирование и Z-index</TITLE>
<SCRIPT language=JavaScript>
<!--
    var tim; // Таймер.
    // Шаг перемещения по экрану.
    function Move()
    {
        var obj = document.all["move"].style;
        if(obj.posLeft < 500) obj.posLeft +=2;
        else clearInterval(tim);
    }
//-->
</SCRIPT>
</HEAD>
<BODY bgcolor=white text=black
    onload="tim=setInterval('Move()',20)">
<H2>Абсолютное позиционирование и Z-index</H2>
<HR>
<DIV style="position:absolute; top:50; left:140;
    height:130; width:100; background:red;
    font-size:60; color: white; z-index:3">
    Это ближе
</DIV>
<DIV style="position:absolute; top: 50; left:360;
    height: 30; width:100; background: blue;
    font-size:30; color:white; z-index:1">
    Это дальше
</DIV>
<DIV id=move style="position:absolute; top:80; left:0;
    width:125;z-index:2">
    <IMG src=pic/explorer.gif width=125 height=82 alt ="">
</DIV>
</BODY>
</HTML>
```

Задания

1. Сделайте страничку, в которой по основному тексту «проплывает» небольшой фрагмент другого текста. Варианты алгоритмов движения описаны ниже:
 - а) текст начинает двигаться слева направо; через некоторое время текст «скачком» возвращается в исходное положение слева, и все повторяется;
 - б) текст начинает двигаться слева направо; через некоторое время текст меняет направление движения, доходит до исходного положения, и все повторяется;
 - в) текст начинает двигаться по синусоиде вправо и «скачком» возвращается в исходное положение;
 - г) текст начинает двигаться по синусоиде вправо, затем меняет направление движения; в исходном положении все повторяется.
2. Сделайте страничку, в которой по основному тексту «плавает» небольшой фрагмент другого текста. Управление движением — при помощи экранных кнопок.
3. Сделайте страничку, в которой по основному тексту «плавает» небольшой фрагмент другого текста. Управление движением — при помощи кнопок клавиатуры.
4. Сделайте страничку, в которой по основному тексту можно перетаскивать мышью картинку.
5. Сделайте страничку, в которой кнопку **Нажми меня** преследует курсор мыши.
6. Сделайте страничку, в которой кнопка **Догони меня** убегает от курсора мыши.

Урок 19



Управление содержимым страницы

Пользуясь свойствами `innerText`, `outerText`, `innerHTML`, `outerHTML`, можно динамически, т. е. уже после построения страницы браузером, менять отдельные элементы на экране. Этими четырьмя свойствами обладают объекты, построенные для большинства видимых элементов гипертекстовой страницы.

Свойство *innerText*

Свойство имеет своим значением весь текст, содержащийся в элементе, включая содержимое всех наследников, без тегов HTML. Присвоение свойству нового текстового значения приводит к замене содержимого элемента на этот текст. При этом вся старая теговая структура внутри элемента пропадает, а новые теги, если они есть, воспринимаются как обычный текст.

Пусть элемент — это абзац, показанный на рис. 19.1.



Это важное слово

Рис. 19.1

Его код имеет вид:

```
<P id=prim>Это <B>важное</B> слово.</P>
```

Тогда команда `alert(document.all["prim"].innerText)` покажет значение «Это важное слово».

Получается, что свойство `innerText` «не видит» теговую разметку внутри элемента.

После присваивания:

```
document.all["prim"].innerText="Это <I>важное</I> слово.";
```

абзац станет таким, как на рис. 19.2.



Это <I>важное</I> слово

Рис. 19.2

Таким образом, теговая разметка в новом содержимом рассматривается как обычный текст.

Свойство *outerText*

Это свойство при чтении дает тот же результат, что и `innerText`. При записи нового значения оно также работает аналогично, но заменяет не только внутренность элемента, но и теги, задающие элемент.

Пусть элементом является полужирный текст внутри абзаца (рис. 19.1):

```
<P>Это <B id=prim>важное</B> слово.</P>
```

Команда `alert(document.all["prim"].outerText)` покажет значение «важное».

После присваивания:

```
document.all["prim"].outerText="не важное";
```

абзац станет таким, как на рис. 19.3.

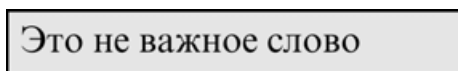


Рис. 19.3

Выделение пропало. Как видите, свойство `outerText` позволяет заменять содержимое элемента вместе с задающими его тегами.

Если команда была бы такой:

```
document.all["prim"].innerText="не важное";
```

выделение сохранилось бы (рис. 19.4).



Рис. 19.4

Свойство *innerHTML*

Это свойство имеет значением весь код, содержащийся внутри элемента.

Пусть элемент — это абзац, показанный на рис. 19.1.

Его код имеет вид:

```
<P id=prim>Это <B>важное</B> слово.</P>
```

Тогда команда `alert(document.all["prim"].innerHTML)` выдаст значение «Это важное слово».

После присваивания:

```
document.all["prim"].innerHTML="Это <I>важное</I> слово.";

```

абзац станет таким, как на рис. 19.5.

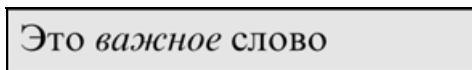


Рис. 19.5

Как видите, теги, записанные в новом значении, корректно работают.

Свойство *outerHTML*

Отличие от `innerHTML` только в том, что в расчет дополнительно берутся начальный и конечный теги, задающие элемент.

Пусть элементом является полужирный текст внутри абзаца, как на рис. 19.1.

Код этого фрагмента:

```
<P>Это <B id=prim>важное</B> слово.</P>

```

Команда `alert(document.all["prim"].outerHTML)` выдаст значение «`<B id=prim>важное</B>`».

После присваивания:

```
document.all["prim"].outerHTML="<I id=prim>не важное</I>";

```

абзац станет таким, как на рис. 19.6.

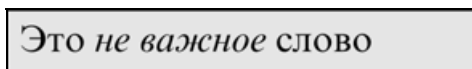


Рис. 19.6

«Крестики-нолики»

Посмотрим, как можно применить описанные свойства при проектировании игры «Крестики-нолики».

Игровое поле построим при помощи таблицы. Для пустых клеток зададим стиль:

```
.empty /* Стиль пустой клетки. */
{
    background-color:#EEE5DB;
    color:#EEE5DB;
}

```

```
width:50; height:60;
text-align:center;
}
```

Для клеток, в которых будут записаны ходы, опишем стиль:

```
.busy /* Стиль занятой клетки. */
{
background-color:#D1B284;
width:50; height:60;
text-align:center;
}
```

Первоначально игровая таблица будет иметь вид, показанный на рис. 19.7.

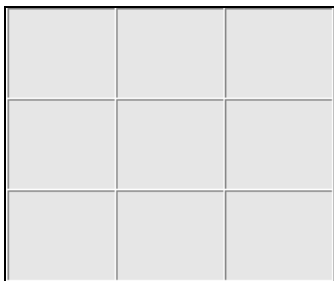


Рис. 19.7. Таблица для игры в «Крестики-нолики»

Вот код этой таблицы (листинг 19.1).

Листинг 19.1

```
<TABLE border=1 cellspacing=0 cellpadding=10>
<TR>
<TD class=empty id=game00>e</TD>
<TD class=empty id=game01>e</TD>
<TD class=empty id=game02>e</TD>
</TR>
<TR>
<TD class=empty id=game10>e</TD>
<TD class=empty id=game11>e</TD>
<TD class=empty id=game12>e</TD>
</TR>
<TR>
<TD class=empty id=game20>e</TD>
<TD class=empty id=game21>e</TD>
```

```
<TD class=empty id=game22>e</TD>
</TR>
</TABLE>
```

В «пустые» клетки записывается символ «е» (от «empty»). Так как цвет текста в пустых клетках совпадает с цветом фона, то эти символы не будут видны на экране.

Построить таблицу можно и программным путем при помощи такой функции:

```
var nameId = "game"; // Основа идентификатора табличной клетки.
// Построение игровой таблицы.
function Show()
{
    var str="<TABLE border=1 cellspacing=0 cellpadding=10>";
    for(var i=0; i<3; i++)
    {
        str += "<TR>";
        for (var j=0; j<3; j++)
            str += "<TD class=empty id="+nameId+i+j+">e</TD>";
        str += "</TR>";
    }
    str += "</TABLE>";
    document.write(str);
}
```

Щелчок мыши будем обрабатывать такой функцией:

```
var stepX = true; // Ход "X"
// Обработка щелчка мыши.
function Click()
{
    var obj = event.srcElement; // Тег, на котором щелкнули мышью.
    if(obj.className == "empty") // Если щелчок по пустой клетке.
    {
        obj.className="busy"; // Изменим стиль клетки.
        obj.innerHTML = stepX ? "X":"0"; // Запишем ход игрока.
        stepX = !stepX;
    }
}
```

Что из этого получилось, видно на рис. 19.8.

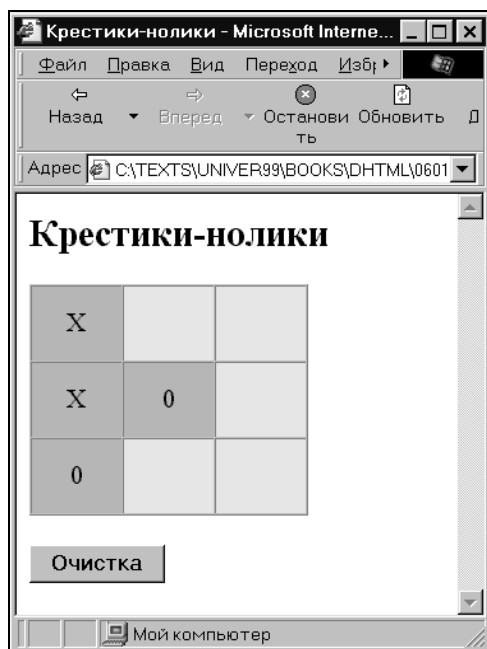


Рис. 19.8. Приложение для игры в «Крестики-нолики»

Полный код этого примера (листинг 19.2).

Листинг 19.2

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
  <TITLE>Крестики-нолики</TITLE>
  <STYLE type="text/css">
  <!--
    .empty /* Стилй пустой клетки. */
    {
      background-color:#EEE5DB;
      color:#EEE5DB;
      width:50; height:60;
      text-align:center;
    }
  </-->
```

```
.busy /* Стиль занятой клетки. */
{
    background-color:#D1B284;
    width:50; height:60;
    text-align:center;
}
-->
</STYLE>
<SCRIPT language=JavaScript>
<!--
var nameId = "game"; // Основа идентификатора табличной клетки.
var stepX = true;    // Ход "X"
// Построение игровой таблицы.
function Show()
{
    var str="<TABLE border=1 cellspacing=0 cellpadding=10>";
    for(var i=0; i<3; i++)
    {
        str += "<TR>";
        for(var j=0; j<3; j++)
            str += "<TD class=empty id="+nameId+i+j+">e</TD>";
        str += "</TR>";
    }
    str += "</TABLE>";
    document.write(str);
}
// Обработка щелчка мыши.
function Click()
{
    var obj = event.srcElement;    // Тег, на котором щелкнули мышью.
    if(obj.className == "empty")  // Если щелчок по пустой клетке.
    {
        obj.className="busy";      // Изменим стиль клетки.
        obj.innerHTML = stepX ? "X":"0"; // Запишем ход игрока.
        stepX = !stepX;
    }
}
```

```

function Clear()
{
    for(var i=0; i<3; i++)
        for(var j=0; j<3; j++)
        {
            var obj=document.all[nameId+i+j];
            obj.className="empty";
            obj.innerText = "e";
        }
    step = "0";
}
//-->
</SCRIPT>
</HEAD>
<BODY bgcolor=white text=black onclick="Click()">
    <H2>Крестики-нолики</H2>
    <SCRIPT language=JavaScript>
    <!--
        Show(); // Нарисуем игровую таблицу.
    //-->
    </SCRIPT>
    <FORM>
        <INPUT type=button value=Очистка onclick="Clear()">
    </FORM>
</BODY>
</HTML>

```

Чтобы не загромождать учебный код, программа не проверяет окончание игры, но такую проверку легко добавить, и в ней пригодятся неиспользуемые пока идентификаторы клеток таблицы.

Игра будет выглядеть более симпатично, если внутри таблицы расставлять не текстовые, а графические пометки (рис. 19.9).

Щелчок мыши теперь будет обрабатываться такой функцией:

```

// Код для вывода картинок в табличные клетки.
var PicX="<IMG src=./pic/gx.gif width=29 height=31 border=0 alt=''>";
var Pic0="<IMG src=./pic/g0.gif width=29 height=31 border=0 alt=''>";
var stepX = true;    // Ход "X"
// Обработка щелчка мыши.
function Click()

```

```
{  
    var obj = event.srcElement; // Тот, на котором щелкнули.  
    if(obj.className == "empty") // Щелчок по пустой клетке.  
    {  
        obj.className = "busy"; // Изменить стиль клетки.  
        obj.innerHTML = stepX ? PicX : Pic0; // Запишем ход игрока.  
        stepX = !stepX;  
    }  
}
```

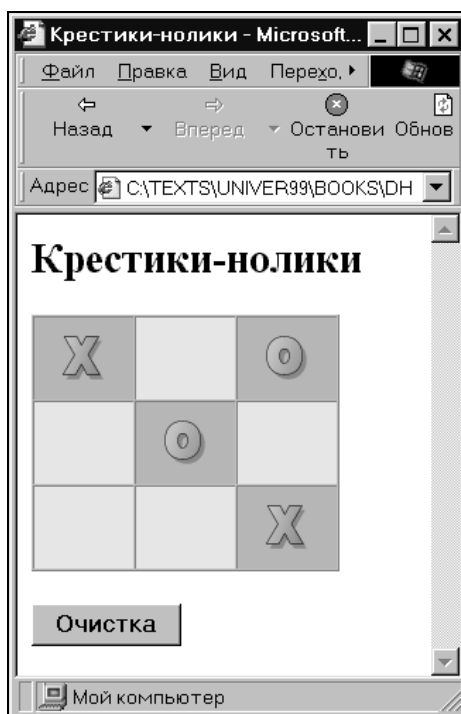


Рис. 19.9. Приложение для игры в «Крестики-нолики»
(графический вариант)

Модель «плавающего» фрейма

Рассмотрим еще один пример динамического изменения содержимого гипертекстовой страницы, показанный на рис. 19.10.

Это приложение имеет следующую структуру (листинг 19.3).

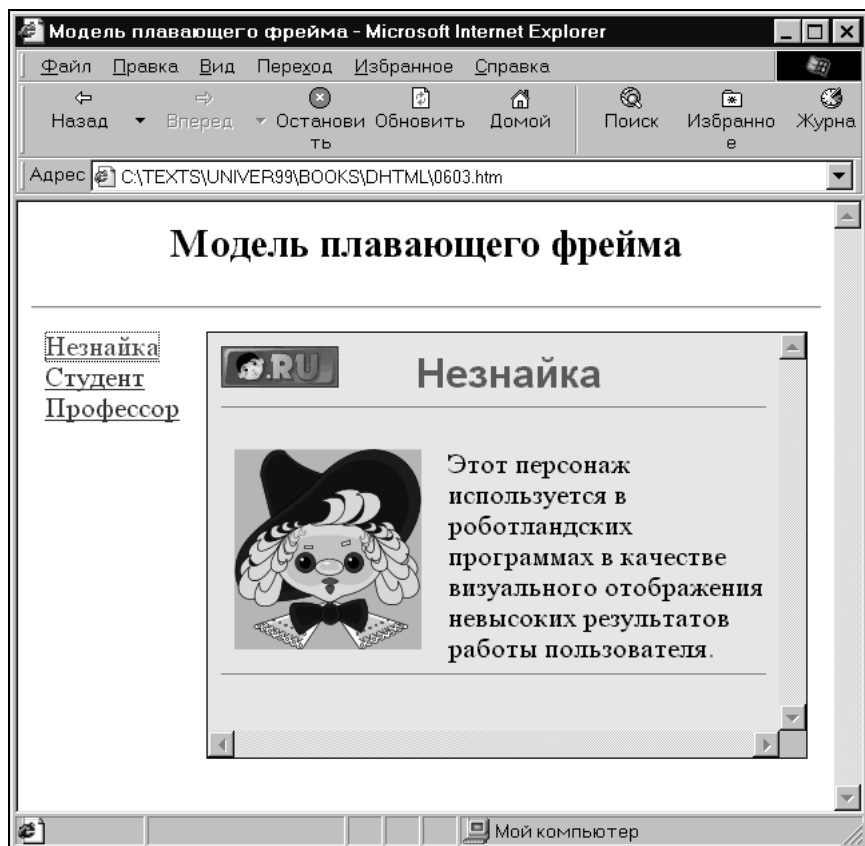


Рис. 19.10. Модель «плавающего» фрейма

Листинг 19.3

```
<HTML>
  <HEAD>
    ...
  </HEAD>
  <BODY bgcolor=white text=black link=blue alink=red vlink=purple
    onload="set (0) ">
    <H2 align=center>Модель плавающего фрейма</H2>
    <HR>
    <TABLE border=0 cellspacing=0 cellpadding=10 width=100%>
      <TR>
        <TD valign=top align=left>
```

```

        <A href="javascript:void(0) "
            onclick="set(0) ">Незнайка</A><BR>
        <A href="javascript:void(0) "
            onclick="set(1) ">Студент</A><BR>
        <A href="javascript:void(0) "
            onclick="set(2) ">Профессор</A><BR>
    </TD>
    <TD valign=top align=right>
        <SPAN id=frame
            style="position:relative;
                width:450; height:320;
                overflow:scroll;
                text-align:left;
                padding:10;
                border: 1px solid black;
                background: #EEE5DB;">
            &nbsp;
        </SPAN>
    </TD>
</TR>
</TABLE>
<DIV id=frame0 style="display:none">
    ...
</DIV>
<DIV id=frame1 style="display:none">
    ...
</DIV>
<DIV id=frame2 style="display:none">
    ...
</DIV>
</BODY>
</HTML>

```

В первой ячейке таблицы располагается гипертекстовое меню, а во второй — модель фрейма (пустого).

Загрузка гипертекстового содержимого в область-фрейм выполняется при помощи функции `set`:

```

// Базовое имя фрейма и областей загрузки.
var nameFrame = "frame";
// Загрузка содержимого n-й области во фрейм.

```

```
function set(n)
{
    eval("document.all['"+nameFrame+"'].innerHTML=document.all['"+
        nameFrame+n+"'].innerHTML");
}
```

При $n=0$ команда внутри функции равнозначна такой:

```
document.all['frame'].innerHTML = document.all['frame0'].innerHTML;
```

То есть во фрейм (тег **** внутри таблицы) загружается содержимое области с идентификатором `frame0`. Именно такая команда срабатывает после загрузки документа (`onload="set(0)"` в теге **<BODY>**).

В гипертекстовых ссылках:

```
<A href="javascript:void(0)" onclick="set(0)">Незнайка</A><BR>
```

```
<A href="javascript:void(0)" onclick="set(1)">Студент</A><BR>
```

```
<A href="javascript:void(0)" onclick="set(2)">Профессор</A><BR>
```

— вместо обычного перехода записан фиктивный `javascript:void(0)`, а щелчок обрабатывается функцией `set` с соответствующим параметром. Обращение `set(0)` загружает во фрейм область `frame0` (про Незнайку), обращение `set(1)` — область `frame1` (про Студента) и обращение `set(2)` — область `frame2` (про Профессора).

Для того чтобы области `frame0`, `frame1` и `frame2` не были видны на экране, к блокам, которые их содержат, применен стиль `display:none` (скрытие элемента).

Задания

1. Доработайте приложение «Крестики-нолики». Напишите краткую инструкцию по игре, разработайте подходящий дизайн, а главное, напишите функцию для проверки окончания игры.

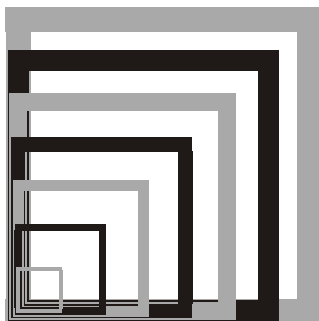
Как проверить, например, не стоит ли (в текстовом варианте игры) символ «X» в первой ячейке таблицы? Доступ к ячейке запишется как `document.all["game00"]`, а проверка на «X» будет иметь вид: `document.all["game00"].innerText == "X"`.

2. Постройте на основе Модели «плавающего» фрейма собственное приложение.
3. Разработайте приложение — версию игры «15». Игра происходит в таблице 4×4 . В клетках таблицы расположены 15 фишек с числами 1, 2, ..., 15. В одной ячейке может располагаться только одна фишка. Ее можно перемещать в соседнюю клетку, если она пуста. Игра заканчивается, когда фишки занимают свои места в порядке возрастания на них чисел (рис. 19.11).

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Рис. 19.11. Завершение игры «15»

Предусмотрите автоматическую генерацию начального состояния игры, контроль числа ходов, проверку игры на окончание. Фишка должна перемещаться в пустую соседнюю клетку щелчком мыши. Приложение будет более привлекательным, если фишки выполнить в графике.



Часть VI

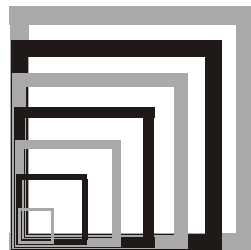
Второй практикум на «кухне» Сидорова

Урок 20. Второй практикум на «кухне»
Сидорова: Конюх

Урок 21. Стилиевые решения

Урок 22. Скрипты

Урок 20



Второй практикум на «кухне» Сидорова: Конюх

В роботландском курсе информатики для младших школьников в разделе «Алгоритмические этюды» есть программа «Конюх» (внешний вид показан на рис. 20.1). Назначение программ этой серии — закрепление на практике понятий: «исполнитель», «среда исполнителя», «система команд», «алгоритм и программа», «аварийные сообщения исполнителя».

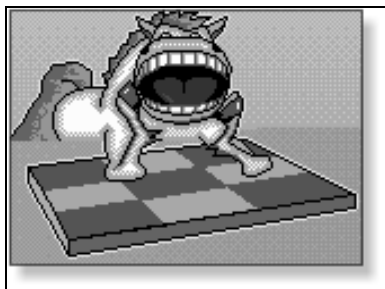


Рис. 20.1

Программа «Конюх» предлагает обучаемому решить две задачи на маленькой шахматной доске из 9 клеток. В начальный момент черные кони расположены на верхней горизонтали, а белые — на нижней. Требуется переставить фигуры так, чтобы белые кони заняли верхний ряд, а черные — нижний. При этом кони перемещаются обычным шахматным прыжком по букве «Г». В первой задаче (она проще), на поле 6 коней во второй (она сложнее) — 4.

Как выглядит среда Конюха перед решением задачи с шестью конями, показано на рис. 20.2.

На экране расположены три области:

- ☐ среда исполнителя (фрагмент шахматной доски из 9 клеток с фигурками коней);
- ☐ поле для ввода команд исполнителю (на нем виден текстовый курсор);

- поле, в котором отображается протокол работы с исполнителем (программа).



Рис. 20.2. Роботландский Конюх

Команда исполнителю формируется по правилам записи шахматных ходов. Например, ход белого коня из клетки a1 в клетку c2 задается командой:

a1-c2

Сидоров получил задание разработать Конюха в гипертекстовом виде для нового роботландского интерактивного учебника «Азы информатики».

Понятно, что работу он начал с составления технического задания.

Техническое задание

Требуется создать программу (экранную среду) для решения задач Конюха. Предполагается, что пользоваться программой будут школьники 1—5 классов в рамках начального курса информатики (раздел «Алгоритмические этюды»). Круг пользователей и учебная цель проектируемой программы накладывает на нее следующие требования:

- программа должна быть визуально привлекательной для младшего школьника;
- программа должна работать под браузером IE, начиная с четвертой версии, и отображаться без линеек прокрутки в окне 640×480.

Экран программы должен содержать:

- среду исполнителя (шахматная доска и фигуры коней);

- ☐ поле ввода команды с группой экранных кнопок:
 - для ввода команды;
 - откатки;
 - накатки;
 - сброса среды (откатки в начало).
- ☐ поле протокола (для демонстрации программы исполнителя);
- ☐ блок экранных кнопок:
 - для выполнения настроек среды:
 - ♦ число коней (6 или 4; 6 по умолчанию);
 - ♦ режим решения (визуальный или командный; командный по умолчанию);
 - показа условия задачи;
 - показа пользовательских инструкций;
 - закрытия окна.

Среда должна позволять:

- ☐ решать задачу в командном режиме при помощи записи текстовой команды в поле ввода;
- ☐ решать задачу в визуальном режиме перетаскиванием коней по экрану в нужную клетку поля;
- ☐ иметь полную откатку и накатку выполненных команд исполнителя во время решения задачи;
- ☐ выводить на экран диагностику при неверных действиях пользователя:
 - «Не понимаю» — задана неверная команда;
 - «Не могу» — команду выполнить невозможно;
- ☐ среда должна иметь контекстно зависимые подсказки;
- ☐ среда должна информировать пользователя об окончании решения задачи;
- ☐ наряду с графическим интерфейсом (экранные кнопки) должны работать «горячие» клавиатурные клавиши.

Построение технического задания Сидоров завершил рисунком, на котором изобразил макет внешнего вида программы.

Экран Конюха содержит:

- ☐ название программы, которое будет сопровождаться рисунком исполнителя;

□ блок кнопок:

- 1 — установка режимов работы программы;
- 2 — показ условия задачи;
- 3 — показ пользовательских инструкций;
- 4 — закрытие окна.

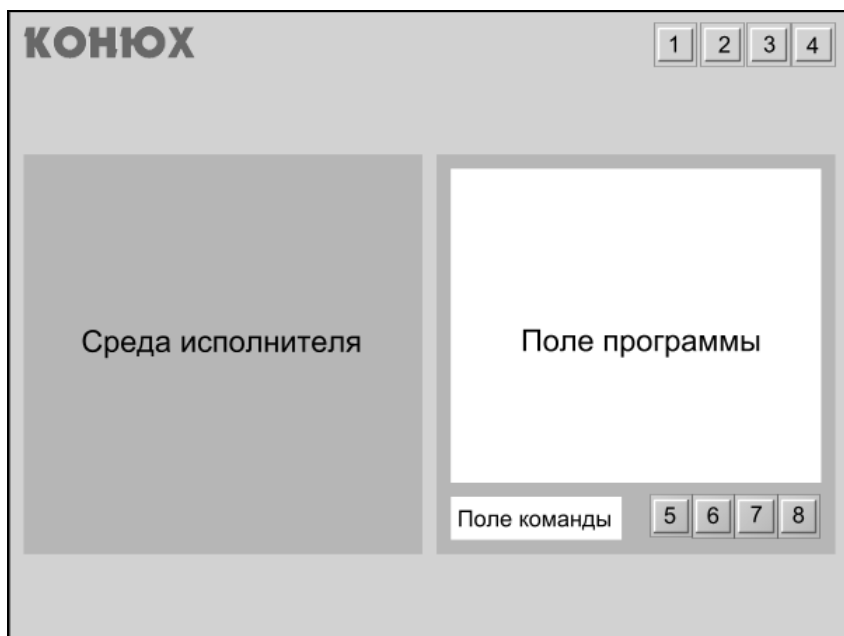


Рис.20.3. Макет внешнего вида нового Конюха.

□ среду исполнителя;

□ панель управления исполнителем, содержащую:

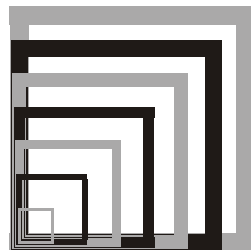
- поле ввода команды;
- поле программы;
- кнопки управления:
 - ♦ 5 — ввод команды;
 - ♦ 6 — откатка;
 - ♦ 7 — накатка;
 - ♦ 8 — сброс (откатка в начало).

Прошло около недели, и Сидоров принес в Роботландию готовый продукт, показанный на рис. 20.4.



Рис. 20.4. Новый гипертекстовый Конюх

Урок 21



Стилевые решения

Программа «Конюх» будет вызываться со страниц учебника «Азы информатики» для работы в отдельном окне. Зададим размеры этого окна как 610×460 — это (с небольшим запасом) обеспечит показ программы без ли-неек прокрутки на мониторах с разрешением 640×480.

Вызвать Конюха с гипертекстовой страницы можно будет, например, при помощи стандартной кнопки (листинг 21.1).

Листинг 21.1

```
<FORM>
```

```
  <INPUT type=button value=Конюх
    onclick="window.open('./horse/horse.htm','',
      'width=610,height=460,menubar=0,resizable=0,scrollbars=0')">
```

```
</FORM>
```

Используя CSS-позиционирование элементов, построим жесткий экранный каркас, который не будет форматироваться браузером. Все экранные элементы поместим внутрь «выпуклого» прямоугольника, который зададим следующим стилевым определением:

```
.program /* Стиль основной области. */
{
  position:absolute;
  left:10px; top:10px;
  width:590px; height:440px;
  /* "Выпуклая" граница. */
  border-style:      solid;
  border-width:      1px;
  border-left-color:  white;
  border-top-color:  white;
  border-right-color: gray;
  border-bottom-color: gray;
}
```

Размеры прямоугольника заданы как 590×440, что вместе с отступами по краям в 10 пикселей как раз дает размер окна 610×460. В HTML-файле Конюха запишем такой код (листинг 21.2).

Листинг 21.2

```
<BODY bgcolor=#EEE5DB text=black>
  <DIV class=program id=program title="Исполнитель Конюх">
    </DIV>
</BODY>
```

Как это будет выглядеть на странице браузера, показано на рис. 21.1.



Рис. 21.1. Окно Конюха с основной областью

Теперь зададим стили для среды исполнителя и панели управления:

```
.environment /* Стилъ среды исполнителя. */
{
    position:absolute;
    left: 10px; top:100px;
    width:280px; height:280px;
    padding: 10px;
    /* "Вдавленная" граница. */
    background-color:    #D1B284;
    border-style:        solid;
    border-width:        1px;
```

```

border-left-color:   gray;
border-top-color:    gray;
border-right-color:  white;
border-bottom-color: white;
}

.control /* Стиль панели управления. */
{
    position: absolute;
    left: 300px; top: 100px;
    width: 280px; height: 280px;
    padding: 10px;
    /* "Вдавленная" граница. */
    background-color:   #D1B284;
    border-style:       solid;
    border-width:       1px;
    border-left-color:  gray;
    border-top-color:   gray;
    border-right-color: white;
    border-bottom-color: white;
}

```

Эти две области будут вложены в основной блок `program`, поэтому отсчет координат в них выполняется от левого верхнего угла этого блока (листинг 21.3).

Листинг 21.3

```

<BODY bgcolor=#EEE5DB text=black>
  <DIV class=program id=program title="Исполнитель Конюх">
    <SPAN class=environment id=environment title="Среда исполнителя">
      <IMG src=./pic/horsef.gif width=258 height=258
        hspace=0 vspace=0 border=0
        alt="Среда исполнителя" title="Среда исполнителя">
    </SPAN>
    <SPAN class=control title="Панель управления">
    </SPAN>
  </DIV>
</BODY>

```

Отображение этого кода на экране показано на рис. 21.2.

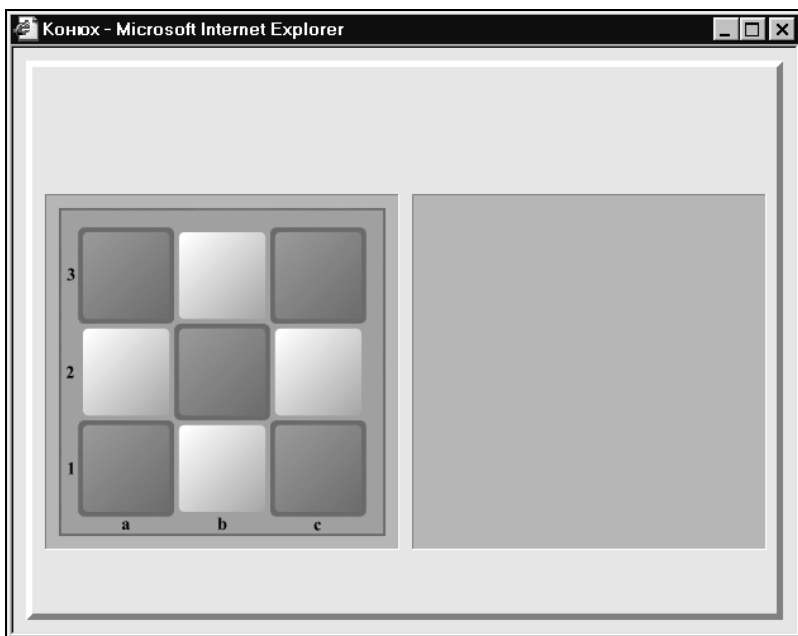


Рис. 21.2. Окно Конюха со средой и панелью управления

Зададим стили для поля программы (протокола) и поля ввода команд:

```
.list /* Стилъ протокола. */
{
    position:absolute;
    left: 10px; top:10px;
    width:258px; height:220px;
    font-size:12pt;
}
.com /* Стилъ поля ввода. */
{
    position:absolute;
    left: 10px; top:238px;
    width:90px; height:30px;
    font-size:14pt;
    font-family:"Arial Cyr",Geneva,Helvetica,sans-serif;
}
```

Поле протокола и поле ввода будут размещаться в блоке `control`, поэтому координаты полей будут отсчитываться от левого верхнего угла этой области (листинг 21.4).

Листинг 21.4

```

<BODY bgcolor=#EEE5DB text=black>
  <DIV class=program id=program title="Исполнитель Конюх">
    <SPAN class=environment id=environment title="Среда исполнителя"
      ><IMG src=./pic/horsef.gif width=258 height=258
        hspace=0 vspace=0 border=0
        alt="Среда исполнителя" title="Среда исполнителя">
    </SPAN>
    <SPAN class=control title="Панель управления">
      <FORM name=forma>
        <TEXTAREA class=list name=list cols=20 rows=10
          title="Программа" readonly>
        </TEXTAREA>
        <INPUT class=com name=com type=text size=7
          maxlength=5 title="Поле ввода команды">
      </FORM>
    </SPAN>
  </DIV>
</BODY>

```

Отображение этого кода на экране показано на рис. 21.3.

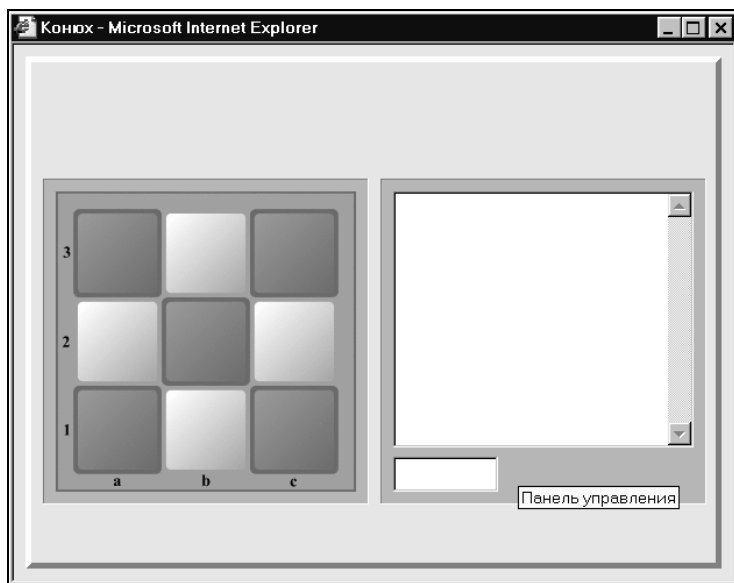


Рис. 21.3. Окно Конюха с полем протокола и строкой ввода

Введем область `programUp` — верхнюю часть окна Конюха. В нее вложим картинку с изображением исполнителя и область `programKey` с кнопками 1—4. Новые стилевые определения выглядят так:

```
/* Стилй верхней части программной области. */
.programUp
{
    position:absolute;
    left: 10px; top:10px;
}
/* Стилй блока с кнопками в верхней части. */
.programKey
{
    position:absolute;
    left: 449px;
}
```

«Чувствительность» экранных кнопок будем имитировать подменой изображения при перемещении курсора на кнопку и уходе с нее (события `mouseover` и `mouseout` листинг 21.5).

Листинг 21.5

```
<BODY bgcolor=#EEE5DB text=black>
<DIV class=program id=program title="Исполнитель Конюх">
  <SPAN class=programUp>
    <IMG src=./pic/horser.gif width=430 height=91
      border=0 hspace=0 vspace=0
      alt="Исполнитель Конюх" title="Исполнитель Конюх">
  <SPAN class=programKey>
    <NOBR><IMG src=./pic/key1.gif width=30 height=30
      border=0 hspace=0 vspace=0
      alt="Настройка программы" title="Настройка программы"
      onmouseover="this.src='./pic/key1_.gif'"
      onmouseout="this.src='./pic/key1.gif'">
    <IMG src=./pic/key2.gif width=30 height=30
      border=0 hspace=0 vspace=0
      alt="Постановка задачи" title="Постановка задачи"
      onmouseover="this.src='./pic/key2_.gif'"
      onmouseout="this.src='./pic/key2.gif'">
    <IMG src=./pic/key3.gif width=30 height=30
```

```

border=0 hspace=0 vspace=0
alt="Инструкции" title="Инструкции"
onmouseover="this.src='./pic/key3_.gif'"
onmouseout="this.src='./pic/key3.gif'">
<IMG src=./pic/key4.gif width=30 height=30
border=0 hspace=0 vspace=0
alt="Выход" title="Выход"
onmouseover="this.src='./pic/key4_.gif'"
onmouseout="this.src='./pic/key4.gif'">

</NOBR>
</SPAN> <!-- Конец области programKey -->
</SPAN> <!-- Конец области programUp -->
<SPAN class=environment id=environment title="Среда исполнителя">
  <IMG src=./pic/horsef.gif width=258 height=258
    hspace=0 vspace=0 border=0
    alt="Среда исполнителя" title="Среда исполнителя">
</SPAN><!-- Конец области environment -->
<SPAN class=control title="Панель управления">
  <FORM name=forma>
    <TEXTAREA class=list name=list cols=20 rows=10
      title="Программа" readonly>
    </TEXTAREA>
    <INPUT class=com name=com type=text size=7
      maxlength=5 title="Поле ввода команды">
  </FORM>
</SPAN> <!-- Конец области control -->
</DIV> <!-- Конец области program -->
</BODY>

```

Отображение этого кода на экран показано на рис. 21.4.

В среду исполнителя поместим фигурки коней. Было бы логично расположить соответствующие HTML-коды внутри блока `environment`, но придется располагать их «снаружи». Дело в том, что браузер IE-5 интерпретирует координаты элемента при его перетаскивании по экрану (а коней хочется перемещать мышью!) совсем не так, как браузер IE-4, когда элемент позиционируется не относительно блока `<BODY>`. Поэтому (в угоду обоим браузерам) координаты коней будем отсчитывать от левого верхнего угла рабочего поля окна (элемента `<BODY>` листинга 21.6).

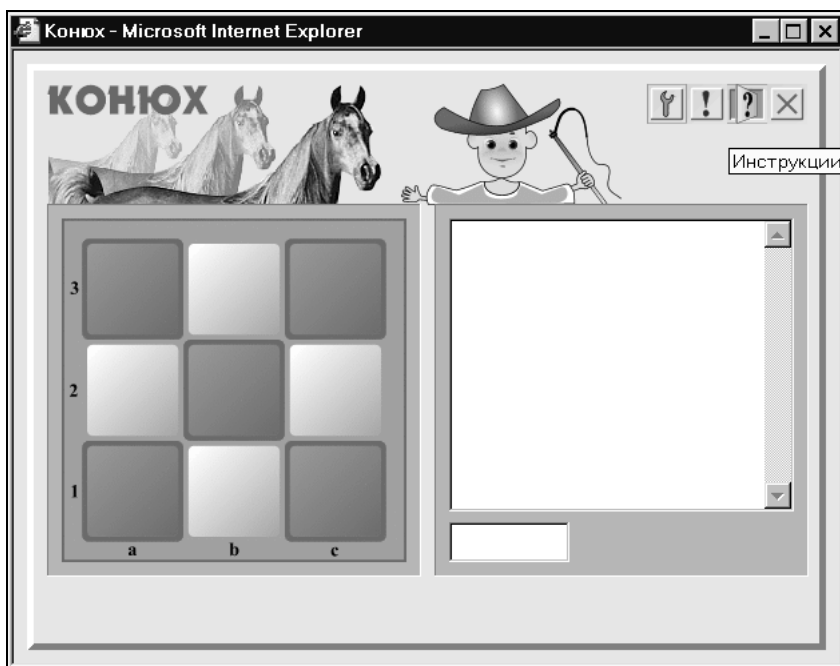


Рис. 21.4. Конюх с изображением исполнителя и кнопками окна

Листинг 21.6

```
<IMG style="position:absolute;left:60px;top:144px;"
      src=./pic/horseb.gif
      id=horse00 width=47 height=61 border=0
      alt="Черный конь" title="Черный конь">
<IMG style="position:absolute;left:136px;top:144px;"
      src=./pic/horseb.gif
      id=horse10 width=47 height=61 border=0
      alt="Черный конь" title="Черный конь">
<IMG style="position:absolute;left:212px;top:144px;"
      src=./pic/horseb.gif
      id=horse20 width=47 height=61 border=0
      alt="Черный конь" title="Черный конь">
<IMG style="position:absolute;left:60px;top:296px;"
      src=./pic/horsew.gif
      id=horse02 width=47 height=61 border=0
```

```

    alt="Белый конь" title="Белый конь">
<IMG style="position:absolute;left:136px;top:296px;"
    src=./pic/horsew.gif
    alt="Белый конь" title="Белый конь"
    id=horse12 width=47 height=61 border=0
    alt="Белый конь" title="Белый конь">
<IMG style="position:absolute;left:212px;top:296px;"
    src=./pic/horsew.gif
    id=horse22 width=47 height=61 border=0
    alt="Белый конь" title="Белый конь">

```

На рис. 21.5 показан вид окна Конюха с этими картинками.

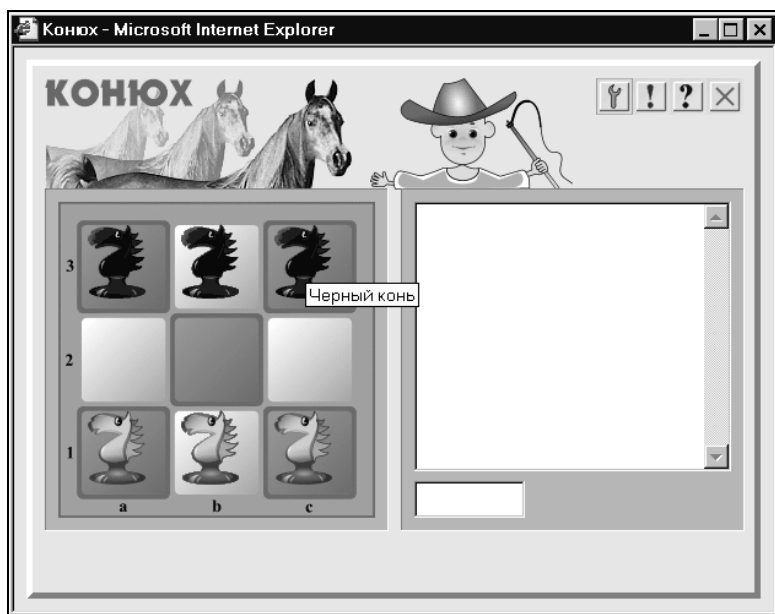


Рис. 21.5. Окно Конюха с изображением коней на рабочем поле

Теперь осталось совсем немного. Определим стиль блока с кнопками на панели управления:

```

/* Стиль кнопок на панели управления. */
.controlKey
{
    position:absolute;
    left:150px; top:238px;
    width:160px;
}

```

Разместим в этом блоке сами кнопки (листинг 21.7).

Листинг 21.7

```
<BODY bgcolor=#EEE5DB text=black>

<DIV class=program id=program title="Исполнитель Конюх">
  <SPAN class=programUp>
    ...
    <SPAN class=programKey>
      ...
    </SPAN>
  </SPAN>
  <SPAN class=environment id=environment title="Среда исполнителя">
    ...
  </SPAN>
  <SPAN class=control title="Панель управления">
    <FORM name=forma>
      ...
    </FORM>
    <SPAN class=controlKey>
      <NOBR><IMG src=./pic/key5.gif width=30 height=30
        border=0 hspace=0 vspace=0
        alt="Ввод команды" title="Ввод команды"
        onmouseover="this.src='./pic/key5_.gif'"
        onmouseout="this.src='./pic/key5.gif'">
      <IMG src=./pic/key6.gif width=30 height=30
        border=0 hspace=0 vspace=0
        alt="Откатка" title="Откатка"
        onmouseover="this.src='./pic/key6_.gif'"
        onmouseout="this.src='./pic/key6.gif'">
      <IMG src=./pic/key7.gif width=30 height=30
        border=0 hspace=0 vspace=0
        alt="Накатка" title="Накатка"
        onmouseover="this.src='./pic/key7_.gif'"
```

```

onmouseout="this.src='./pic/key7.gif'">
<IMG src='./pic/key8.gif' width=30 height=30
border=0 hspace=0 vspace=0
alt="Сброс" title="Сброс"
onmouseover="this.src='./pic/key8_.gif'"
onmouseout="this.src='./pic/key8.gif'">
</NOBR>
</SPAN> <!-- Конец области controlKey -->
</SPAN>
</DIV>
<!-- Картинки с изображением коней -->
...
</BODY>

```

На рис. 21.6 показан вид окна Конюха с этими картинками.

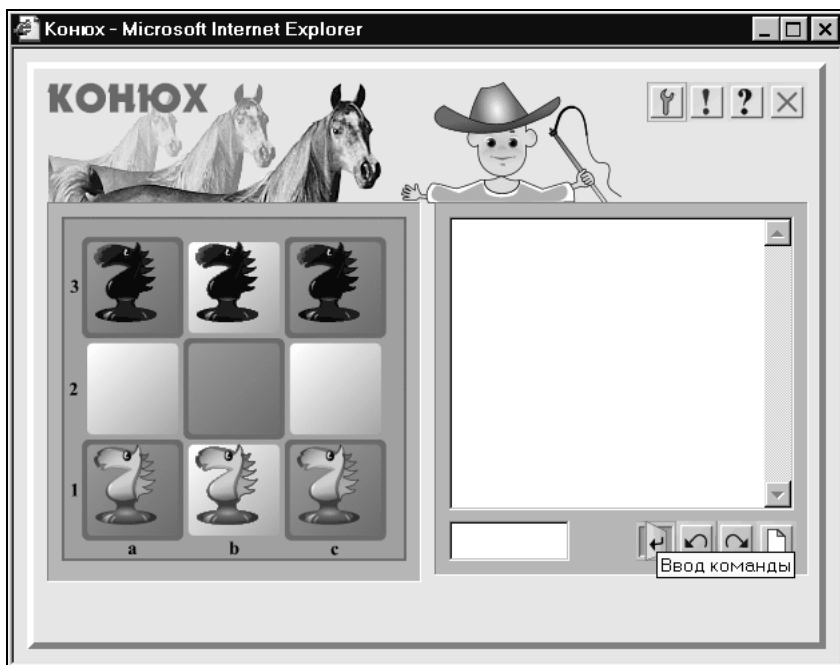


Рис. 21.6. Окно Конюха с панелью управления

Последний штрих: добавим картинку-строку с авторской подписью. Стиль новой области определим так:

```
/* Стиль нижней части программной области. */
```

```
.programDown
{
    position:absolute;
    left: 10px; top:400px;
}
```

Картинку расположим внутри этой области (листинг 21.8).

Листинг 21.8

```
<BODY bgcolor=#EEE5DB text=black>

<DIV class=program id=program title="Исполнитель Конюх">
    <SPAN class=programUp>
        ...
        <SPAN class=programKey>
            ...
        </SPAN> <!-- Конец области programKey -->
    </SPAN> <!-- Конец области programUp -->
    <SPAN class=environment id=environment title="Среда исполнителя">
        ...
    </SPAN> <!-- Конец области environment -->
    <SPAN class=control title="Панель управления">
        ...
    </SPAN> <!-- Конец области control -->
    <SPAN class=programDown>
        <IMG src=./pic/autor.gif width=421 height=11
            border=0 alt="Авторы" title="Авторы">
        </SPAN> <!-- Конец области programDown -->
    </DIV>
    <!-- Картинки с изображением коней -->
    ...
</BODY>
```

На рис. 21.7 показан вид окна Конюха с авторской подписью.

Экранный образ приложения практически построен, но это просто изображение, которое почти никак не реагирует на воздействия мыши или клавиатуры. Изображение превратится в настоящую программу тогда, когда будут

написаны коды скриптов, обрабатывающие эти события. Некоторые поучительные фрагменты данных кодов будут рассмотрены в следующем уроке.

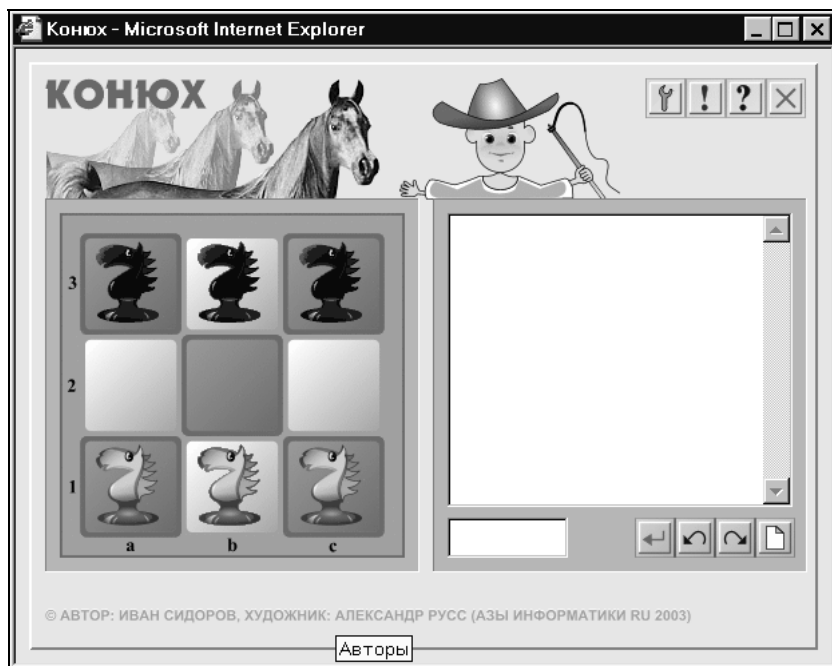
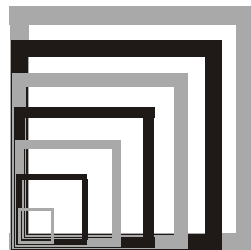


Рис. 21.7. Окно Конюха с авторской подписью

Урок 22



Скрипты

Перетаскивание элементов по экрану

Перетаскивание экранного элемента в графических системах пользователь привык выполнять следующим образом.

- ☐ Нажать кнопку мыши на элементе (захват элемента, событие `mousedown`).
- ☐ Не отпуская нажатой кнопки, перемещать курсор мыши, а вместе с ним и элемент по экрану (перемещение элемента, событие `mousemove`).
- ☐ Отпустить кнопку мыши в нужном месте экрана (освобождение элемента, событие `mouseup`).

Если нужно перемещать по экрану картинку `<IMG>`, то вызов собственных обработчиков событий можно вмонтировать в этот тег:

```
<IMG ...  
  onmousedown = "..."  
  onmousemove = "..."  
  onmouseup   = "...">
```

Когда перемещаемых элементов много, удобнее вызовы обработчиков располагать не в тегах, а прямо в объекте `document`:

```
// Назначение обработчиков для событий, связанных с перетаскиванием.  
document.onmousedown = mouseDownEvent;  
document.onmousemove = mouseMoveEvent;  
document.onmouseup   = mouseUpEvent;  
// Обработка захвата элемента.  
// -----  
function mouseDownEvent()  
{  
  ...  
}  
// Обработка перемещения элемента.
```

```
// -----
function mouseMoveEvent()
{
    ...
}
// Обработка освобождения элемента.
// -----
```

```
function mouseUpEvent()
{
    ...
}
```

Свойства `onmousedown`, `onmousemove`, `onmouseup` объекта `document` содержат ссылки на функции, которые обрабатывают соответствующие события.

Команды:

```
document.onmousedown = mouseDownEvent;
document.onmousemove = mouseMoveEvent;
document.onmouseup = mouseUpEvent;
```

записывают в эти свойства ссылки на функции `mouseDownEvent`, `mouseMoveEvent`, `mouseUpEvent`, которые предстоит запрограммировать.

Пусть, например, нужно перемещать по экрану любую картинку (тег `<img>`). Когда пользователь «захватывает» элемент, управление передается в функцию `mouseDownEvent()`. Нужно проверить, что элемент действительно является картинкой, и запомнить ссылку на объект, построенный браузером для этого элемента:

```
// Указатель на перетаскиваемый элемент или null.
var imgDragging = null;
// Обработка захвата элемента.
// -----
function mouseDownEvent()
```

```
{
    // Это картинка?
    if(event.srcElement.tagName == "IMG")
    {
        // Запомним указатель на объект.
        imgDragging = event.srcElement;
    }
    // Перетаскивание запрещено.
    else imgDragging = null;
}
```

На самом деле в момент захвата элемента нужно сделать еще две вещи: запомнить координаты курсора мыши относительно левого верхнего угла перемещаемого элемента и «приподнять» элемент над основным слоем (чтобы он перемещался «поверх» всех других элементов экрана):

```
// Указатель на перетаскиваемый элемент или null.
var imgDragging = null;
// Положение курсора относительно элемента
// в момент начала перетаскивания.
var dx;
var dy;
// Обработка захвата элемента.
// -----
function mouseDownEvent()
{
    // Это картинка?
    if(event.srcElement.tagName == "IMG")
    {
        // Запомним указатель на объект.
        imgDragging = event.srcElement;
        // Вычислим смещения мышиного курсора
        // относительно начала перетаскиваемого элемента.
        dx = event.offsetX;
        dy = event.offsetY;
        // Передвинем элемент в верхний слой.
        imgDragging.style.zIndex = 1;
    }
    // Перетаскивание запрещено
    else imgDragging = null;
}
```

Вот теперь все подготовлено к перемещению элемента по экрану!

Обработка события `mousemove` может быть такой:

```
// Обработка перемещения элемента.
// -----
function mouseMoveEvent()
{
    if(imgDragging != null)
    {
        // Вычислим новые координаты элемента.
```

```

imgDragging.style.pixelLeft = event.x - dx;
imgDragging.style.pixelTop  = event.y - dy;
// Остановим прохождение события.
event.cancelBubble = true;
event.returnValue = false;
}
}

```

Последние две команды запрещают браузеру обрабатывать событие `mousemove` стандартным образом (стандартная обработка помешает работе наших кодов).

Теперь осталось написать обработчик события `mouseup` (пользователь отпустил кнопку мыши). Код этого обработчика крайне прост: в нем устанавливается флаг запрета дальнейшего перемещения, и элемент возвращается в основной экранный слой.

```

// Обработка освобождения элемента.
// -----
function mouseUpEvent()
{
    if(imgDragging != null)
    {
        // Вернем элемент в основной слой.
        imgDragging.style.zIndex = 0;
        // Сбросим признак перемещения.
        imgDragging = null;
    }
}

```

Ниже представлен код документа (листинг 22.1), в котором сведены воедино все рассмотренные выше скрипты.

Листинг 22.1

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML lang="ru">
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
  <TITLE>Перетаскивание картинок в документе</TITLE>
  <SCRIPT language=JavaScript>
  <!--

```

```
//===== Перетаскивание картинок по экрану =====
// Назначение обработчиков для событий, связанных с перетаскиванием.
document.onmousedown=mouseDownEvent;
document.onmousemove=mouseMoveEvent;
document.onmouseup=mouseUpEvent;
// Положение курсора относительно картинки
// в момент начала перетаскивания.
var dx;
var dy;
// Указатель на перетаскиваемый элемент или null.
var imgDragging = null;
// Обработка захвата элемента.
// -----
function mouseDownEvent()
{
    // Это картинка?
    if(event.srcElement.tagName == "IMG")
    {
        // Запомним указатель на объект.
        imgDragging = event.srcElement;
        // Вычислим смещения курсора мыши
        // относительно начала перетаскиваемого элемента.
        dx = event.offsetX;
        dy = event.offsetY;
        // Элемент передвинем в верхний слой.
        imgDragging.style.zIndex = 1;
    }
    else imgDragging = null; // Перетаскивание запрещено.
}
// Обработка перемещения элемента.
// -----
function mouseMoveEvent()
{
    if(imgDragging != null)
    {
        // Вычислим новые координаты элемента.
        imgDragging.style.pixelLeft = event.x - dx;
```

```

imgDragging.style.pixelTop = event.y - dy;
// Остановим прохождение события.
event.cancelBubble = true;
event.returnValue = false;
}
}
// Обработка освобождения элемента.
// -----
function mouseUpEvent()
{
    if(imgDragging != null)
    {
        // Вернем элемент в основной слой.
        imgDragging.style.zIndex = 0;
        // Сбросим признак перемещения.
        imgDragging = null;
    }
}
//-->
</SCRIPT>
</HEAD>
<BODY bgcolor=#EEE5DB text=black>
    <H2>Перетаскивание картинок в документе</H2>
    <HR>
    <P>

```

Фигурки коней можно перетаскивать по экрану мышью. Нажмите на картинке левую кнопку мыши и, не отпуская ее, переместите изображение в другое место.

```

<IMG style="position:absolute; left:100px;top:200px"
    src=./pic/horseb.gif width=47 height=61
    border=0 alt="Черный конь">
<IMG style="position:absolute; left:100px;top:300px"
    src=./pic/horsew.gif width=47 height=61
    border=0 alt="Белый конь">
</BODY>
</HTML>

```

Вид этого документа показан на рис. 22.1.

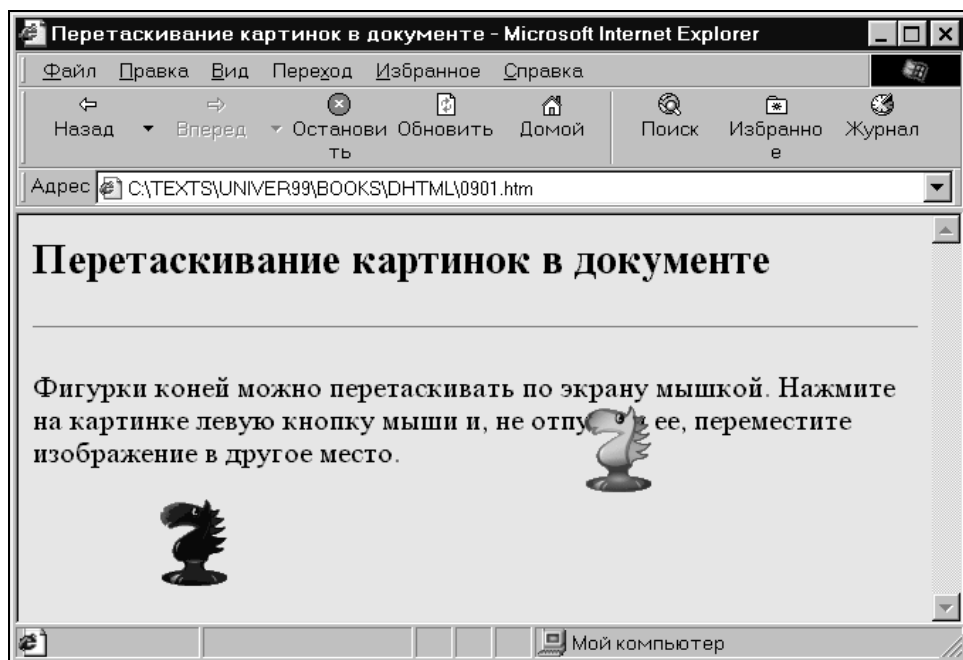


Рис. 22.1. Вид приложения «Перетаскивание картинок в документе»

Добавим в HTML-код примера еще одну картинку — изображение шахматной доски, не меняя при этом обслуживающие скрипты. Понятно, что третья картинка подвержена перетаскиванию по экрану, как и первые две (листинг 22.2).

Листинг 22.2

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML lang="ru">
  <HEAD>
    <META http-equiv="Content-Type"
      content="text/html; charset=windows-1251">
    <TITLE>Перетаскивание трех картинок</TITLE>
    <SCRIPT language=JavaScript>
    <!--
      ...
    //-->
  </SCRIPT>
</HEAD>
```

```

<BODY bgcolor=#EEE5DB text=black>
  <H2>Перетаскивание трех картинок</H2>
  <HR>
  <P>
    Все картинки на этой странице подвержены перетаскиванию.
    <IMG style="position:absolute;left:60px;top:210px;"
      src=./pic/horsef.gif width=258 height=258
      hspace=0 vspace=0 border=0
      alt="Среда исполнителя" title="Среда исполнителя">
    <IMG style="position:absolute; left:90px;top:234px;"
      src=./pic/horseb.gif width=47 height=61 border=0
      id=horse00 alt="Черный конь" title="Черный конь">
    <IMG style="position:absolute; left:90px;top:386px"
      src=./pic/horsew.gif width=47 height=61 border=0
      id=horse02 alt="Белый конь" title="Белый конь">
  </BODY>
</HTML>

```

Вид этого документа показан на рис. 22.2.

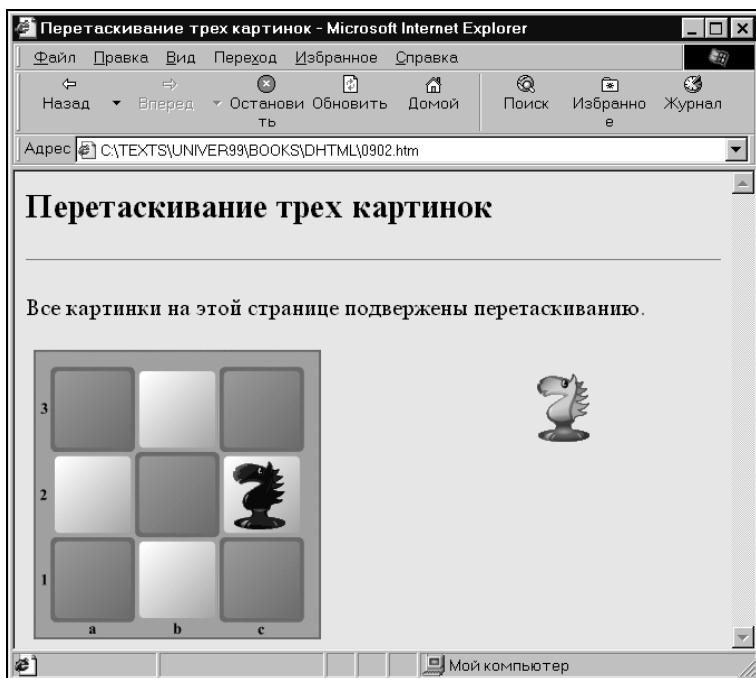


Рис. 22.2. Вид приложения «Перетаскивание трех картинок»

Поставим задачу — разрешить пользователю перемещать только коней. Запишем в каждом теге, кодирующем коня, атрибут `id` со значением `horseXY`, где `X` и `Y` соответствуют координатам клетки шахматного поля, в которой конь расположен в начальный момент. Теперь легко отличить картинку, изображающую коня, от других картинок:

```
// Обработка захвата элемента
// -----
function mouseDownEvent()
{
    // Это конь?
    if(event.srcElement.id.indexOf("horse") != -1)
    {
        // Запомним указатель на объект.
        imgDragging = event.srcElement;
        // Вычислим смещения курсора мыши
        // относительно начала перетаскиваемого элемента.
        dx = event.offsetX;
        dy = event.offsetY;
        // Элемент передвинем в верхний слой.
        imgDragging.style.zIndex = 1;
    }
    else imgDragging = null; // Перетаскивание запрещено.
}
```

Теперь картинка с шахматной доской перетаскиванию не подвержена.

Следующий шаг — ограничить перемещение коней областью шахматного поля: нужно проверять координаты курсора мыши перед тем, как присваивать коням новые координаты:

```
// Обработка перемещения элемента
// -----
function mouseMoveEvent()
{
    if(imgDragging != null)
    {
        // Вычислим новые координаты элемента.
        var left = event.x - dx;
        var top  = event.y - dy;
        if(left > fieldLeftRU+fieldDLeftRU*2)
            left = fieldLeftRU+fieldDLeftRU*2;
```

```
else if(left < fieldLeftRU)
    left = fieldLeftRU;
if(top > fieldTopRU+fieldDTopRU*2)
    top = fieldTopRU+fieldDTopRU*2;
else if(top < fieldTopRU)
    top = fieldTopRU;
// Присвоим новые координаты.
imgDragging.style.pixelLeft = left;
imgDragging.style.pixelTop = top;
// Остановим прохождение события.
event.cancelBubble = true;
event.returnValue = false;
}
}
```

Перед тем, как «освободить» коня от связи с курсором, нужно определить шахматную клетку, над которой «повис» конь; проверить допустимость хода и, если ход возможен, «подровнять» положение коня по центру клетки.

Модальные окна

В программе Конюх используются «всплывающие» сообщения двух типов:

- ☐ сообщения исполнителя «Не понимаю» и «Не могу» (они не мешают работе пользователя в среде);
- ☐ сообщения «Постановка задачи», «Инструкции» и диалог «Настройка программы» (они блокируют работу в основном окне).

Сообщения первого типа реализованы при помощи обычных картинок, которые в обычном состоянии не видны на экране (стилевое свойство `visibility` установлено в `hidden`), а в нужный момент показываются (`visibility` устанавливается в `visible`).

Сообщения второго типа показываются в так называемых *модальных* окнах. Модальное окно блокирует работу пользователя в основном окне до тех пор, пока не будет закрыто.

Модальное окно открывается при помощи метода `showModalDialog` объекта `window`. Формат метода:

```
перемен = window.showModalDialog(файл, аргумент, параметры);
```

Здесь:

- ☐ `перемен` — переменная, в которую помещается возвращаемое из модального окна значение;
- ☐ `файл` — текстовая строка — ссылка на HTML-файл, который будет отображаться в модальном окне;

- аргумент — аргумент, передаваемый в модальное окно;
- параметры — текстовая строка — параметры, задающие вид, размеры окна и его положение на экране.

Параметры окна задаются в строке `параметры` в виде:

```
"пар1=знач1;пар2=знач2;...;парN=значN"
```

Вместо знака равенства можно использовать двоеточие (как в стилевых оп-ределениях). Допустимо задание следующих параметров (табл. 22.1):

Таблица 22.1

Параметр	Значение	Описание
<code>dialogWidth</code>	Число	Ширина окна
<code>dialogHeight</code>	Число	Высота окна
<code>dialogTop</code>	Число	Вертикальная координата левого верхнего угла окна
<code>dialogLeft</code>	Число	Горизонтальная координата левого верхнего угла окна
<code>center</code>	Yes, no, 1, 0	Выравнивать ли окно по центру экрана
<code>help</code>	Yes, no, 1, 0	Показывать ли кнопку вызова подсказки в заголовке окна

Модальное окно без аргументов

Сначала покажем вызов модального окна без передачи в него аргументов. Подготовим файл `2201.htm` — он будет отображаться в модальном окне (листинг 22.3).

Листинг 22.3

```
<HTML>
...
<BODY bgcolor=#D1B284 text=#4C3300
    onkeypress="
        if(event.keyCode==27)
        { // Нажата клавиша Esc.
            window.returnValue=null;
            window.close();
        }">
<DIV style="color:maroon;">
```

```

    <B>Постановка задач Конюха</B>
  </DIV><BR>
  <P>
    <DIV class=field>
      <P>

```

Попробуйте, используя правила шахматных ходов, переставить белых и черных коней.

```

    <P>

```

Задача с 6 конями решается за 8 ходов, а задача с 4 конями — за 16.

```

  </DIV>

```

```

</BODY>

```

```

</HTML>

```

Для вызова модального окна напомним такой код (листинг 22.4).

Листинг 22.4

```

<FORM>

```

Вызов окна без передачи в него аргументов.

```

  <P>

```

```

    <INPUT type=button value=Показать
      onclick=" window.showModalDialog('2201.htm','',
        'dialogWidth:400px;dialogHeight:340px;center:yes;help:no;')">

```

```

</FORM>

```

Вид этого модального окна показан на рис. 22.3.

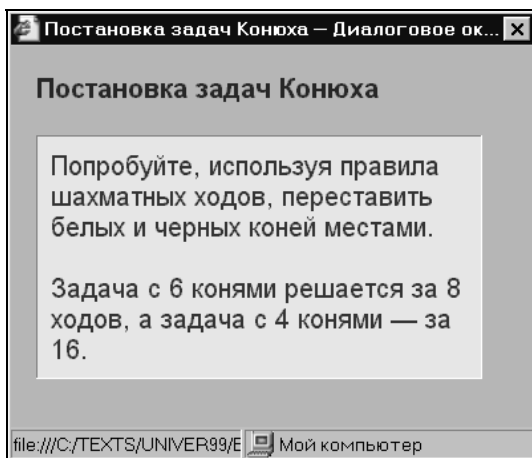


Рис. 22.3. Модальное окно «Постановка задач Конюха»

Модальное окно с одним аргументом

В следующем примере выведем аргумент модального окна на экран.

Подготовим HTML-код для вывода в окно — файл 2202.htm (листинг 22.5).

Листинг 22.5

```
<HTML>
...
<BODY bgcolor=#D1B284 text=#4C3300
    onkeypress="
        if(event.keyCode==27)
        { // Нажата клавиша Esc.
            window.returnValue=null;
            window.close();
        }">
<DIV style="color:maroon;">
    <B>
        <SCRIPT language=JavaScript>
            <!--
                // Вывод аргумента в окно.
                document.write(window.dialogArguments);
            //-->
        </SCRIPT>
    </B><BR>
</DIV>
<DIV class=field>
```

Это пример модального окна. Его можно закрыть экранной кнопкой

```
<IMG src=./pic/key4.gif width=30 height=30 border=0 hspace=0 vspace=0
alt="Выход"> или клавишей Esc (предусмотрен специальный обработчик в са-
```

мом модальном окне).

```
</DIV>
</BODY>
</HTML>
```

Вызов модального окна и передачу в него аргумента закодируем так (листинг 22.6).

Листинг 22.6

```
<FORM>
```

Введите аргумент (он будет передан в модальное окно):

<P>

```
<INPUT type=text name=arg size=50
      value="Этот текст будет выведен в модальное окно.">
<INPUT type=button value=Показать
      onclick="window.showModalDialog('2202.htm', this.form.arg.value,
      'dialogWidth:440px;dialogHeight:300px;center:yes;help:no;') ">
```

</FORM>

Вызов и вид этого модального окна показаны на рис. 22.4 и 22.5.

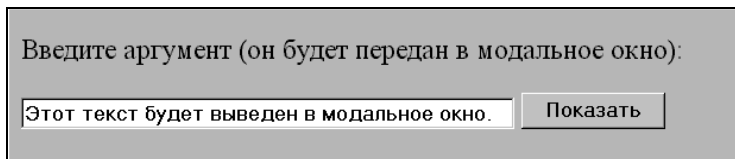


Рис. 22.4. Вызов модального окна с одним аргументом



Рис. 22.5. Модальное окно с одним аргументом

Модальное окно с несколькими аргументами

В модальное окно можно передать сколько угодно аргументов, разместив их в массиве. Подготовим код (листинг 22.7) модального окна (файл 2203.htm), которое будет выводить на экран два переданных ему аргумента.

Листинг 22.7

<HTML>

...

```
<BODY bgcolor=#D1B284 text=#4C3300
    onkeypress="
        if (event.keyCode==27)
        { // Нажата клавиша Esc.
            window.returnValue=null;
            window.close();
        }">
<DIV style="color:maroon;">
    <B>
    <SCRIPT language=JavaScript>
    <!--
        // Выводим в окно первый аргумент.
        document.write(window.dialogArguments[0]);
    //-->
    </SCRIPT>
    </B><BR>
</DIV>
<DIV class=field>
```

Это пример модального окна. Его можно закрыть экранной кнопкой

```
<IMG src=./pic/key4.gif width=30 height=30 border=0 hspace=0 vspace=0
alt="Выход"> или клавишей Esc (предусмотрен специальный обработчик в са-
мом модальном окне).
```

```
</DIV>
<DIV style="color:maroon;">
    <B>
    <SCRIPT language=JavaScript>
    <!--
        // Выводим в окно второй аргумент.
        document.write(window.dialogArguments[1]);
    //-->
    </SCRIPT>
    </B><BR>
</DIV>
</BODY>
</HTML>
```

Вызов модального окна и передачу в него аргументов закодируем так (листинг 22.8).

Модальное окно возвращает данные

Предыдущие примеры показывают, что в модальное окно можно передавать данные. Понятно, что связь документа с собственным модальным окном была бы неполноценной, если бы данные не могли проходить и в обратном направлении.

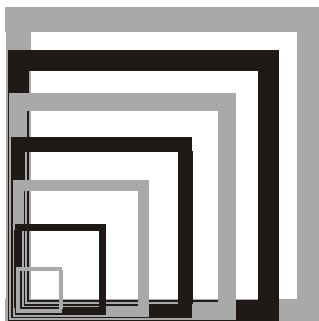
Получить данные из модального окна очень просто. Нужно поместить в нем ссылку на данные в свойство `window.returnValue`, а в документе-родителе при вызове окна воспользоваться конструкцией:

```
var ret = window.showModalDialog(...);
```

Переменная `ret` и будет содержать возвращаемые данные после закрытия модального окна. Возвращать можно одиночное значение или целый массив данных.

Полный код Конюха

Исполнитель (вместе с кодами других примеров книги) можно скопировать с адреса: <ftp://ftp.botik.ru/rented/robot/univer/dbook.zip>



Часть VII

Приложения

Приложение 1. Справочник CSS

Приложение 2. Справочник JavaScript

Приложение 3. Справочник объекты
и события браузера

Приложение 1

Справочник CSS

Единицы измерения

Типографские значения

Таблица П1.1

Единица	Описание
pt	Пункты. Один пункт равен 0,375 мм. В одном дюйме 72 пункта
pc	Пики. Одна пика равна 12 пунктам

Экранные значения

Таблица П1.2

Единица	Описание
px	Пиксели. Пиксел — это элемент экрана, размер которого зависит от установок монитора и видеокарты

Абсолютные значения длины

Таблица П1.3

Единица	Описание
in	Дюймы. Один дюйм равен 2,54 мм
cm	Сантиметры
mm	Миллиметры

Структура справочных таблиц

Описание стилевых свойств задаются в виде табл. П1.4.

Таблица П1.4

Свойство	Описание
font-size	Значения
fontSize	<абсолютный> <относительный> <размер> <процент> Здесь:
Размер шрифта	☐ <абсолютный>: xx-small x-small small medium large x-large xx-large
	☐ <относительный>: larger smaller
	☐ <размер>: в любых допустимых единицах измерения;
	☐ <процент>: процент от родительского размера шрифта.
	По умолчанию: medium Поддерживается: всеми элементами Наследование: да

Первая колонка таблицы содержит название свойства CSS, а также (выделено) название соответствующего свойства объекта style.

Пусть, например, в HTML-коде задан такой элемент:

```
<DIV id=block style="font-size:small">  
    Огромный шар, надутый паром,<BR>  
    Поднялся в воздух он не даром,<BR>  
    Наш коротышка хоть не птица,<BR>  
    Летать он все-таки годится.<BR>  
    И все доступно уж, эхма!<BR>  
    Теперь для нашего ума!<BR>
```

</DIV>

Если, в процессе интерактивного взаимодействия страницы с пользователем, необходимо изменить размер шрифта элемента block, то это достигается, например, такими строками на JavaScript:

```
var block=document.all.block.style;  
block.fontSize = "large";
```

Во второй колонке таблицы приводятся возможные значения свойства. При этом запись в угловых скобках означает целое семейство значений, а запись без угловых скобок — задает допустимое ключевое слово, которое можно записывать в качестве значения свойства. Символ «|» разделяет возможные варианты.

Примечания даются в третьей колонке.

Шрифт

Таблица П1.5

Свойство	Описание	Примечание
font-family	Значения	Можно установить несколько возможных значений в порядке предпочтения (на случай, когда у пользователя не окажется нужного шрифта). Значения-указания разделяются запятыми: font-family: "Arial Cyr", Geneva, sans-serif;
fontFamily	Название семейства шрифта (например, New York) или родовое имя (serif, sans-serif, monospace, cursive, fantasy).	
Название семейства	По умолчанию: устанавливается браузером	
	Поддерживается: всеми элементами Наследование: да	
font-size	Значения	
fontSize	<абсолютный> <относительный> <размер> <процент>	
Размер	Здесь: ☐ <абсолютный>: xx-small x-small small medium large x-large xx-large <относительный>: larger smaller <размер>: в любых допустимых единицах измерения; <процент>: процент от родительского размера шрифта По умолчанию: medium Поддерживается: всеми элементами Наследование: да	
font-style	Значения	
fontStyle	normal small-caps	
Определение стиля	По умолчанию: normal	
	Поддерживается: всеми элементами Наследование: да	

Таблица П1.5 (окончание)

Свойство	Описание	Примечание
<code>font-variant</code> <code>fontVariant</code>	<i>Значения</i> <code>normal italic, oblique</code> <i>По умолчанию:</i> <code>normal</code>	
Вариант написания	<i>Поддерживается:</i> всеми элементами <i>Наследование:</i> да	
<code>font-weight</code> <code>fontWeight</code>	<i>Значения</i> <code>normal bold bolder lighter <число от 100 до 900></code> <i>По умолчанию:</i> <code>normal</code> <i>Поддерживается:</i> всеми элементами <i>Наследование:</i> да	Если используется числовое значение, то оно должно быть кратным 100. Значение 400 — это то же, что <code>normal</code> , 700 — то же, что и <code>bold</code>
Толщина линий шрифта		
<code>font</code> <code>font</code>	<i>Значения</i> <code><font-style> <font-variant> <font-weight></code>	Процентная запись допускается только для <code><font-size></code> и <code><font-height></code> .
Свойства шрифта	<code><font-size> / <font-height> <font-family></code> <i>По умолчанию:</i> не определено <i>Поддерживается:</i> всеми элементами <i>Наследование:</i> да	Это свойство позволяет установить сразу несколько характеристик шрифта. Разделитель значений — пробел. Порядок значений важен, но любое значение может быть опущено, кроме <code><font-size></code> и <code><font-family></code> . Пример: <code>font: bold 12pt / 14pt sans-serif;</code>

Цвет и фон

Таблица П1.6

Свойство	Описание	Примечание
<code>color</code> <code>color</code>	<i>Значения</i> Название цвета или его номер. <i>По умолчанию:</i> зависит от браузера	Цвет может быть задан названием (например, <code>blue</code>) или по шкале RGB: в шестнадцатеричной системе (например, <code>#ffffff</code>), в десятичной системе (например, <code>120, 0, 255</code>), процентами (например, <code>80%, 0%, 20%</code>)
Цвет текста	<i>Поддерживается:</i> всеми элементами <i>Наследование:</i> да	

Таблица П1.6 (продолжение)

Свойство	Описание	Примечание
<code>background-attachment</code> backgroundAttachment	<i>Значения</i> <code>fixed scroll</code> <i>По умолчанию:</i> <code>scroll</code>	Определяет, будет ли фоновая картинка занимать фиксированное положение или прокручиваться вместе с содержимым
Прокрутка фоновой картинки	<i>Поддерживается:</i> всеми элементами <i>Наследование:</i> нет	
<code>background-color</code> backgroundColor	<i>Значения</i> <code>transparent <цвет></code> <i>По умолчанию:</i> <code>transparent</code> (прозрачный)	Цвет может быть задан названием (например, <code>blue</code>) или по шкале RGB: в шестнадцатеричной системе (например, <code>#ffffff</code>), в десятичной системе (например, <code>120, 0, 255</code>), процентами (например, <code>80%, 0%, 20%</code>)
Цвет фона	<i>Поддерживается:</i> всеми элементами <i>Наследование:</i> нет	
<code>background-image</code> backgroundImage	<i>Значения</i> <code>none <ссылка></code> <i>По умолчанию:</i> <code>none</code> (нет)	Определяет фоновую картинку. Синтаксис записи ссылки: <code>BODY {background-image: url(fon1.gif)}</code>
Фоновая картинка	<i>Поддерживается:</i> всеми элементами <i>Наследование:</i> нет	
<code>background-position</code> backgroundPosition	<i>Значения</i> <code>top center bottom left right <позиция></code> <i>По умолчанию:</i> <code>top left</code>	Определяет начальное положение фоновой картинки (или фоновой заливки) с помощью двух значений (сначала по горизонтали, потом по вертикали) или при помощи двух ключевых слов из возможных. Пример: <code>.fon {background-position: center left}</code>
Начальное положение фоновой картинки или заливки	<i>Поддерживается:</i> всеми элементами <i>Наследование:</i> нет	

Таблица П1.6 (продолжение)

Свойство	Описание	Примечание
<code>background-repeat</code> <code>backgroundRepeat</code> Повторение фоновой картинки	<i>Значения</i> <code>repeat</code> <code>repeat-x</code> <code>repeat-y</code> <code>no-repeat</code> <i>По умолчанию:</i> <code>repeat</code> <i>Поддерживается:</i> всеми элементами <i>Наследование:</i> нет	Определяет, повторяется ли фоновая картинка. Если используется значение <code>repeat-x</code> или <code>repeat-y</code>, то картинка повторяется только по одному направлению. Пример: <pre>BODY { background-image: url(fon1.gif); background-repeat: no-repeat; }</pre>
<code>background</code> <code>background</code> Цвет фона	<i>Значения</i> <code>transparent</code> <code><цвет></code> <code><ссылка></code> <code><повтор></code> <code><скролл></code> <code><позиция></code> Здесь: ☐ <code><цвет></code>: название цвета или его номер; ☐ <code><ссылка></code>: картинка с заданным адресом. Адрес заключается в скобки и идет непосредственно за ключевым словом <code>url</code>: <pre>BODY {background: url(fon1.gif)}</pre> Можно использовать и цвет, и картинку. Картинка будет лежать поверх цветного фона. ☐ <code><повтор></code>: <code>repeat</code> (по умолчанию) <code>repeat-x</code> <code>repeat-y</code> <code>no-repeat</code> ☐ <code><скролл></code>: <code>fixed</code> <code>scroll</code>	Порядок следования указаний не важен, разделитель — пробел. Пример: <pre>background: #C0C0C0 right no-repeat url(..\pic\pic.gif);</pre>

Таблица П1.6 (окончание)

Свойство	Описание	Примечание
	Определяет, будет ли фоновая картинка оставаться на месте или прокручиваться вместе со страницей. ☐ <позиция>: определяет положение картинки на странице. Значения могут быть процентными, абсолютными или ключевыми словами: top middle bottom left center right По умолчанию: transparent Поддерживается: всеми элементами Наследование: нет	

Текст

Таблица П1.7

Свойство	Описание	Примечание
letter-spacing letterSpacing	Значения normal <расстояние> По умолчанию: normal	Определяет добавочное расстояние между буквами
Расстояние между буквами	Поддерживается: всеми элементами Наследование: да	
line-height lineHeight	Значения normal <число> <расстояние> <процент>	Определяет высоту текущей строки. Значение <число> интерпретируется как размер шрифта текущего элемента, умноженный на соответствующее число (например, 1.2). Если используется <расстояние>, то должны применяться единицы измерения. Процентное отношение используется в сравнении с текущим размером шрифта и должно быть больше 100%
Высота строки	По умолчанию: зависит от браузера Поддерживается: всеми элементами Наследование: да	

Таблица П1.7 (продолжение)

Свойство	Описание	Примечание
<code>text-align</code> textAlign	<i>Значения</i> left right center justify	Определяет, как текст будет выровнен относительно содержащего его элемента
Выравнивание текста	<i>По умолчанию:</i> зависит от браузера <i>Поддерживается:</i> всеми элементами <i>Наследование:</i> да	
<code>text-decoration</code> textDecoration	<i>Значения</i> none overline underline line-through	Линия вдоль строки (нет, над, под, через)
Специальный вид текста	<i>По умолчанию:</i> none <i>Поддерживается:</i> всеми элементами <i>Наследование:</i> нет	
<code>text-indent</code> textIndent	<i>Значения</i> <величина> <процент> <i>По умолчанию:</i> 0	Устанавливает величину отступа первой строки (красной строки), в единицах измерения или как процент от ширины родительского элемента
Величина отступа первой строки	<i>Поддерживается:</i> всеми элементами <i>Наследование:</i> да	
<code>text-transform</code> textTransform	<i>Значения</i> none capitalize uppercase lowercase	capitalize — делает заглавной первую букву каждого слова. uppercase — делает все буквы в словах заглавными. lowercase — делает все буквы в словах строчными
Регистр символов	<i>По умолчанию:</i> none <i>Поддерживается:</i> всеми элементами <i>Наследование:</i> да	
<code>vertical-align</code> verticalAlign	<i>Значения</i> baseline sub super top text-top middle bottom text-bottom	baseline — выравнивание по образцу родительского элемента. sub — перевод в нижние индексы. super — перевод в верхние индексы. top — выравнивает верх элемента по верху самого высокого элемента текущей строки
Расположение по вертикали	<i>По умолчанию:</i> baseline <i>Поддерживается:</i> всеми элементами <i>Наследование:</i> нет	

Таблица П1.7 (продолжение)

Свойство	Описание	Примечание
		text-top — выравнивает элемент по верху текста, набранного шрифтом родительского элемента middle — устанавливает вертикальную среднюю линию элемента на основе родительского элемента плюс половина строки последнего bottom — выравнивает по низу самого низкого элемента текущей строки text-bottom — выравнивает элемент по низу текста, набранного шрифтом родительского элемента
list-style-image listStyleImage Картинка в качестве маркера списка	Значения none <ссылка> По умолчанию: none Поддерживается: всеми элементами Наследование: да	Определяет адрес картинки, используемой в качестве маркера для каждого элемента списка. Пример: LI {list-style-image: url(../pic/pic/gif);} Это свойство (как и другие, связанные со списками) имеет смысл только для списочных тегов, но может передаваться к ним по наследству от любых объемлющих элементов
list-style-type listStyleType Вид маркера списка	Значения none circle disc square decimal lower-alpha upper-alpha lower-roman upper-roman По умолчанию: disc Поддерживается: всеми элементами Наследование: да	Определяет вид маркера списка. none — нет; circle — окружность; disc — круг; square — квадрат; decimal — арабская нумерация; lower-alpha — строчные латинские буквы; upper-alpha — прописные латинские буквы; lower-roman — малые римские цифры; upper-roman — большие римские цифры;

Таблица П1.7 (окончание)

Свойство	Описание	Примечание
<code>list-style-position</code> <code>listStylePosition</code>	<i>Значения</i> <code>outside</code> <code>inside</code> <i>По умолчанию:</i> <code>outside</code>	Указывает, где внутри или вне «тела» списка должен находиться маркер
Положение маркера списка	<i>Поддерживается:</i> всеми элементами <i>Наследование:</i> да	

Поля и рамки

Стилевые свойства этого раздела поддерживаются блоковыми элементами, т. е. такими, которые отображаются на экране с переводом строки до и после элемента (`<P>`, `<DIV>`, `<TABLE>` и т. д.).

Таблица П1.8

Свойство	Описание	Примечание
<code>border-color</code> <code>border-top-color</code> <code>border-bottom-color</code> <code>border-right-color</code> <code>border-left-color</code> <code>borderColor</code> <code>borderTopColor</code> <code>borderBottomColor</code> <code>borderRightColor</code> <code>borderLeftColor</code>	<i>Значения</i> <code><цвет></code> <i>По умолчанию:</i> нет цвета <i>Поддерживается:</i> блоковыми элементами <i>Наследование:</i> нет	Определяет цвет четырех сторон рамки. Свойство работает, если задано свойство <code>border-style</code>
Цвет четырех сторон рамки		
<code>border-style</code> <code>border-top-style</code> <code>border-bottom-style</code> <code>border-right-style</code> <code>border-left-style</code> <code>borderStyle</code> <code>borderTopStyle</code> <code>borderBottomStyle</code> <code>borderRightStyle</code> <code>borderLeftStyle</code>	<i>Значения</i> <code>none</code> <code>solid</code> <code>double</code> <code>groove</code> <code>ridge</code> <code>inset</code> <code>outset</code> <i>По умолчанию:</i> <code>none</code> <i>Поддерживается:</i> блоковыми элементами <i>Наследование:</i> нет	Определяет стиль четырех сторон рамки
Стиль четырех сторон рамки		

Таблица П1.8 (продолжение)

Свойство	Описание	Примечание
border-width	Значения	Определяет ширину четырех сторон рамки
border-top-width	thin medium thick	
border-bottom-width	<размер>	
border-right-width	По умолчанию: medium	
border-left-width	Поддерживается: блоковыми элементами	
borderWidth	Наследование: нет	
borderTopWidth		
borderBottomWidth		
borderRightWidth		
borderLeftWidth		
Ширина четырех сторон рамки		
border	Значения	Определяет свойства четырех сторон рамки (объединяет все свойства рамки). Пример: <DIV style = "border:thin solid blue"> ... </DIV>
border-top	<ширина> <стиль> <цвет>	
border-bottom	По умолчанию: medium	
border-right	none <нет цвета>	
border-left	Поддерживается: блоковыми элементами	
border	Наследование: нет	
borderTop		
borderBottom		
borderRight		
borderLeft		
Свойства четырех сторон рамки		
Margin	Значения	Определяет размер отступов данного элемента. Процентная запись относительно ширины родительского элемента
margin-top	auto <ширина> <процент>	
margin-bottom	По умолчанию: 0	
margin-right	Поддерживается: блоковыми элементами	
margin-left	Наследование: нет	
margin		
marginTop		
marginBottom		
marginRight		
marginLeft		
Отступ элемента		

Таблица П1.8 (окончание)

Свойство	Описание	Примечание
<code>padding</code>	<i>Значения</i>	Определяет расстояние между содержимым и рамкой элемента. Процентная запись относительно ширины родительского элемента
<code>padding-top</code>	<code><ширина> <процент></code>	
<code>padding-bottom</code>	<i>По умолчанию:</i> 0	
<code>padding-right</code>	<i>Поддерживается:</i> блоковыми элементами	
<code>padding-left</code>	<i>Наследование:</i> нет	
<code>padding</code>		
<code>paddingTop</code>		
<code>paddingBottom</code>		
<code>paddingRight</code>		
<code>paddingLeft</code>		
Расстояние между содержимым и рамкой		

Вид

Таблица П1.9

Свойство	Описание	Примечание
<code>display</code>	<i>Значения</i>	Это свойство определяет, будет ли показан элемент. Если элемент не виден (<code>none</code>), то место на странице под него не отводится
<code>display</code>	<code>"" none</code>	
	<i>По умолчанию:</i> <code>""</code>	
Видимость элемента	<i>Поддерживается:</i> всеми элементами <i>Наследование:</i> нет	
<code>float</code>	<i>Значения</i>	Определяет «обтекание» элемента другими элементами (под, слева или справа)
<code>styleFloat</code>	<code>none left right</code>	
	<i>По умолчанию:</i> <code>none</code>	
Обтекание	<i>Поддерживается:</i> блоковыми элементами <i>Наследование:</i> нет	

Таблица П1.9 (продолжение)

Свойство	Описание	Примечание
<code>clear</code> <code>clear</code>	<i>Значения</i> <code>none left right</code> <i>По умолчанию:</i> <code>none</code>	Указывает, что следующие элементы должны быть показаны ниже элемента, выровненного по левому или правому краю. По умолчанию текст “обтекает” такие элементы
Прекращение «обтекания»	<i>Поддерживается:</i> всеми элементами <i>Наследование:</i> нет	
<code>overflow</code> <code>overflow</code>	<i>Значения</i> <code>"" scroll</code> <i>По умолчанию:</i> <code>""</code> <i>Поддерживается:</i> всеми элементами	Значение <code>scroll</code> указывает, что содержимое элемента будет прокручиваться, если оно выйдет за его границы
Прокрутка	<i>Наследование:</i> нет	
<code>visibility</code> <code>visibility</code>	<i>Значения</i> <code>visible hidden inherit</code> <i>По умолчанию:</i> <code>inherit</code> <i>Поддерживается:</i> всеми элементами	Позволяет элементу быть видимым или невидимым на странице. Невидимые (<code>hidden</code>) элементы занимают то же место и так же влияют на расположение других элементов, как и видимые (<code>visible</code>), но становятся прозрачными. Значение <code>inherit</code> означает, что элемент виден, если его родительский элемент является видимым
Видимость	<i>Наследование:</i> нет	
<code>width</code> <code>width</code> <code>pixelWidth</code> <code>posWidth</code>	<i>Значения</i> <code>auto <ширина> <процент></code> <i>По умолчанию:</i> <code>auto</code> <i>Поддерживается:</i> блоковыми элементами	Устанавливает ширину элемента. Значение возвращается как строка, включающая тип единицы измерения (px, % и т.д.). Для получения значения в виде числа используется свойство <code>posWidth</code> объекта <code>style</code> . Процентная запись относительно ширины родительского элемента
Ширина	<i>Наследование:</i> нет	
<code>height</code> <code>height</code> <code>pixelHeight</code> <code>posHeight</code>	<i>Значения</i> <code>auto <высота></code> <i>По умолчанию:</i> <code>auto</code> <i>Поддерживается:</i> блоковыми элементами	Устанавливает высоту элемента. Значение возвращается как строка, включающая тип единицы измерения (px, cm и т.д.). Для получения значения в виде числа используется свойство <code>posHeight</code> объекта <code>style</code>
Высота	<i>Наследование:</i> нет	

Позиционирование, *z-index*

Таблица П1.10

Свойство	Описание	Примечание
<code>positon</code> <code>positon</code>	<i>Значения</i> <code>absolute</code> <code>relative</code> <code>static</code>	Свойство <code>position</code> объекта <code>style</code> доступно только для чтения
Позиционирование	<i>По умолчанию:</i> <code>static</code> <i>Поддерживается:</i> всеми элементами <i>Наследование:</i> нет	
<code>left</code> <code>left</code> <code>pixelLeft</code> <code>posLeft</code>	<i>Значения</i> <code>auto</code> <code><координата></code> <code><процент></code> <i>По умолчанию:</i> <code>auto</code> <i>Поддерживается:</i> всеми элементами. <i>Наследование:</i> нет	Устанавливает горизонтальную координату элемента. Значение возвращается как строка, включающая тип единицы измерения (<code>px</code> , <code>%</code> и т.д.). Для получения значения в виде числа используется свойство <code>posLeft</code> объекта <code>style</code> . Процентная запись относительно ширины родительского элемента.
Горизонтальная координата элемента		
<code>top</code> <code>top</code> <code>pixelTop</code> <code>posTop</code>	<i>Значения</i> <code>auto</code> <code><координата></code> <code><процент></code> <i>По умолчанию:</i> <code>auto</code> <i>Поддерживается:</i> всеми элементами. <i>Наследование:</i> нет	Устанавливает вертикальную координату элемента. Значение возвращается как строка, включающая тип единицы измерения (<code>px</code> , <code>%</code> и т.д.). Для получения значения в виде числа используется свойство <code>posTop</code> объекта <code>style</code> . Процентная запись относительно высоты родительского элемента.
Вертикальная координата элемента		
<code>z-index</code> <code>zIndex</code>	<i>Значения</i> <code><число></code> <i>По умолчанию:</i> в зависимости от контекста HTML-кода <i>Поддерживается:</i> всеми элементами <i>Наследование:</i> нет	Указывает, в каком порядке элементы будут перекрывать друг друга. Элементы с более высоким <code>z-м</code> будут располагаться «сверху» элементов с более низким. Положительные значения «приподнимают» элемент над обычным текстом, отрицательные «прячут» под него.
Порядок расположения элементов по «слоям»		

Курсор

Таблица П1.11

Свойство	Описание	Примечание
<code>cursor</code> cursor	<i>Значения</i> auto crosshair default hand move e-resize ne-resize nw- resize n-resize se-resize sw-resize w-resize text wait help <i>По умолчанию:</i> auto <i>Поддерживается:</i> всеми элементами <i>Наследование:</i> нет	Определяет вид курсора мыши
Вид курсора мыши		

Приложение 2

Справочник JavaScript

Зарезервированные слова

Словарь зарезервированных слов JavaScript подобен используемому в языке Java, хотя часть слов из приведенных ниже реально в JavaScript не используются:

<code>abstract</code>	<code>boolean</code>	<code>break</code>	<code>byte</code>	<code>case</code>	<code>catch</code>
<code>char</code>	<code>class</code>	<code>const</code>	<code>continue</code>	<code>debugger</code>	<code>default</code>
<code>delete</code>	<code>do</code>	<code>double</code>	<code>else</code>	<code>enum</code>	<code>export</code>
<code>extends</code>	<code>false</code>	<code>final</code>	<code>finally</code>	<code>float</code>	<code>for</code>
<code>function</code>	<code>goto</code>	<code>if</code>	<code>implements</code>	<code>import</code>	<code>in</code>
<code>instanceof</code>	<code>int</code>	<code>interface</code>	<code>long</code>	<code>native</code>	<code>new</code>
<code>null</code>	<code>package</code>	<code>private</code>	<code>protected</code>	<code>public</code>	<code>return</code>
<code>short</code>	<code>static</code>	<code>super</code>	<code>switch</code>	<code>synchronized</code>	<code>this</code>
<code>throw</code>	<code>throws</code>	<code>transient</code>	<code>true</code>	<code>try</code>	<code>typeof</code>
<code>var</code>	<code>void</code>	<code>while</code>	<code>with</code>		

Конструкции

Команда прерывания

Вид конструкции

`break`

Описание

Команда, прерывающая последовательное выполнение команд внутри цикла (`while`, `for`) или переключателя (`switch`) и передающая управление на первую команду за циклом или переключателем.

Пример

```
// Произведение начальных членов
// массива до первого нулевого.
var set = new Array(1,2,3,4,0,5,6,7);
var p = 1;
for(var i=0; i<set.length; i++)
    if (set[i]) p *= set[i];
    else      break;
alert(p);
```

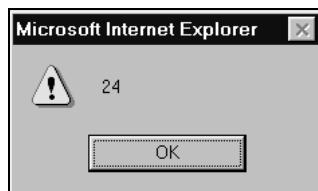


Рис. П2.1

Команда продолжения цикла

Вид конструкции

`continue`

Описание

Команда заставляет браузер пропустить все команды до конца тела цикла (`while`, `for`), но цикл продолжается.

Пример

```
// Произведение всех ненулевых членов массива.
var set = new Array(1,0,2,4,0,5,6,0);
var p = 1;
for(var i=0; i<set.length; i++)
{
    if(!set[i]) continue;
    p *= set[i];
}
alert(p);
```

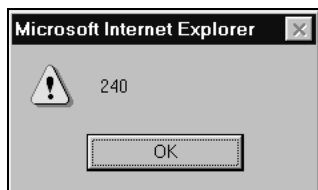


Рис. П2.2

Эту программу лучше, конечно, переписать так:

```
var set = new Array(1,0,2,4,0,5,6,0);
var p = 1;
for(var i=0; i<set.length; i++)
    if (set[i]) p *= set[i];
alert(p);
```

Команда цикла

Вид конструкции

```
for (нач; усл; приращ)  
    команда
```

Описание

Команда `нач` выполняется один раз перед входом в цикл. Цикл состоит в повторении следующих действий:

- ☐ проверка условия `усл`;
- ☐ выполнение команды `команда`;
- ☐ выполнение команды `приращ`.

Если условие оказывается ложным, цикл прекращается (`команда` и `приращ` после отрицательной проверки не работают).

Пример

```
// Произведение нечетных чисел массива.  
var set = new Array(1,2,3,4,0,5,6,8);  
var p = 1;  
for(var i=0; i<set.length; i++)  
    if(set[i]%2) p *= set[i];  
alert(p);
```

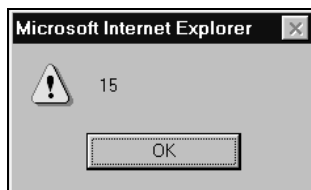


Рис. П2.3

Команда цикла

Вид конструкции

```
for (индекс in объект)  
    команда
```

Описание

Специальная форма цикла, используемая для работы с объектами. Здесь:

- индекс — имя переменной, используемой для доступа к свойствам экземпляра объекта и их значениям;
- объект — имя экземпляра объекта, к которому осуществляется доступ.

Пример

```
var str = "Свойства объекта:\n\n";
for(var i in navigator)
    str += i+"="+navigator[i]+"\\n";
alert(str);
```



Рис. П2.4

Переменная `i` принимает значение имени свойства объекта, а `navigator[i]` — значение этого свойства.

Описание функции (обычной или конструктора объекта)

Вид конструкции

```
function имя (параметры)
{
    команды
}
```

Описание

Здесь:

- имя — имя функции;
 - параметры — список формальных переменных (через запятые);
 - команды — последовательность команд, реализующих алгоритм функции.
- Среди команд функции возможны команды **return** выражение;, если функция возвращает значение.

Пример

```
// Случайное целое число из диапазона [a,b]
function rnd(a,b)
{
    return Math.floor(a + (b-a+1)*Math.random());
}
alert(rnd(1,100));
```

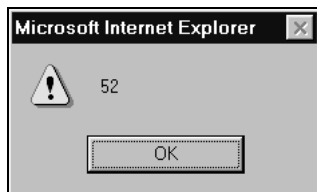


Рис. П2.5

Условная команда

Вид конструкции

```
if (условие) команда1;
else        команда2;
```

или

```
if(условие) команда1;
```

Описание

Если условие принимает значение **true**, выполняется команда1, иначе команда2. В сокращенной форме ветвь **else** отсутствует.

Пример

```
// Абсолютное значение числа.
var x = -25.6754;
if (x < 0) x = -x;
alert(x);
```

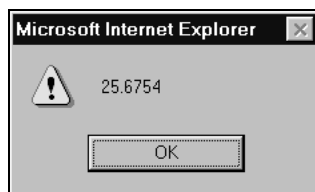


Рис. П2.6

Создание экземпляра объекта

Вид конструкции

```
var x=new объект(пар);
```

Описание

Здесь:

- `x` — имя переменной;
- `объект` — имя объекта;
- `пар` — параметры (значения) экземпляра объекта.

Пример

```
var str = new String("Рога и копыта");  
var set = new Array(str, 1, -25.6754);  
alert(set);
```

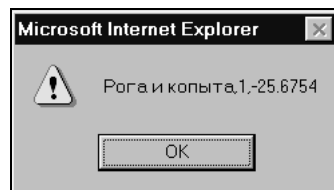


Рис. П2.7

Выход из функции

Вид конструкции

```
return;
```

Описание

Команда, возвращающая значение из функции.

Пример

```
// Сумма элементов массива.  
function Sum(a)  
{  
    var s = 0;  
    for(var i=0; i<a.length; i++) s += a[i];  
    return s;  
}  
var set1 = new Array(15, 25, 10);  
var set2 = new Array(10, 20, 20);  
alert(Sum(set1)+Sum(set2));
```

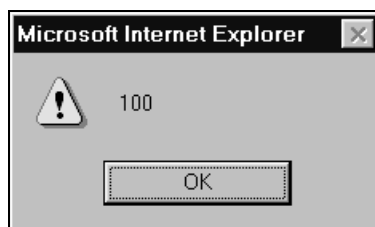


Рис. П2.8

Переключатель (выбор варианта из возможных)

Вид конструкции

```
switch (выражение)
{
    case вариант1:
        команды
        break;
    case вариант2:
        команды
        break;
    ...
    default:
        команды
}
```

Описание

Управление передается на команды, расположенные за первым по порядку **case**, в котором значение варианта совпадает со значением выражения, записанного в заголовке **switch** (выражение).

Команда **break** передает управление на первую команду, следующую за конструкцией **switch**. Если команда **break** отсутствует, проверка вариантов продолжается. Вариант **default** (он может отсутствовать) срабатывает тогда, когда значение выражения не соответствует ни одному из вариантов **case**.

Пример

```
switch (Day)
{
    case 1:
        NameDay="понедельник";
        break;
    case 2:
        NameDay="вторник";
        break;
    case 3:
        NameDay="среда";
```

```
    break;
case 4:
    NameDay="четверг";
    break;
case 5:
    NameDay="пятница";
    break;
case 6:
    NameDay="суббота";
    break;
case 7:
    NameDay="воскресенье";
    break;
default:
    NameDay="ошибка";
}
```

Текущий экземпляр объекта

Вид конструкции

`this`

Описание

Ссылка на текущий экземпляр объекта

Пример

<HTML>

<HEAD>

<SCRIPT language=JavaScript>

<!--

function Name(obj)

{

 alert ("Имя кнопки: "+obj.value);

}

//-->

</SCRIPT>

</HEAD>

<BODY bgcolor=white text=black>

```

<FORM>
  <INPUT type=button value=Go
    onclick="Name (this) ">
  <INPUT type=button value=Show
    onclick="Name (this) ">
</FORM>
</BODY>
</HTML>

```

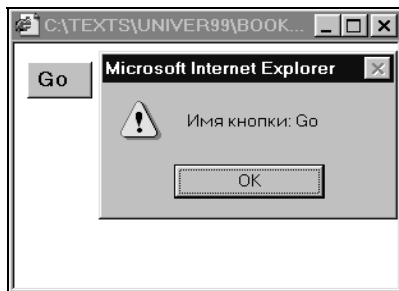


Рис. П2.9

Описание переменной

Вид конструкции

```
var имя = выражение;
```

или

```
var имя;
```

Описание

Создается переменная имя со значением выражение.

Пример

```

var str = new String("мартобря 86 числа");
var opr;
alert(str+"\n"+opr);

```

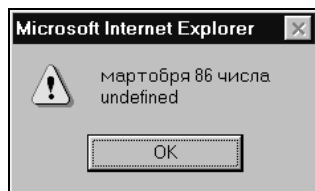


Рис. П2.10

Команда цикла

Вид конструкции

```

while (условие)
  команда;

```

Описание

Цикл состоит в повторении следующих действий:

- ☐ проверка условия условие;
- ☐ выполнение команды команда;

Если условие оказывается ложным, цикл прекращается (команда после отрицательной проверки не работает).

Пример

// Наибольший общий делитель двух чисел.

```
function Nod(a,b)
```

```
{
```

```
  while(b)
```

```
  {
```

```
    var d=a%b;
```

```
    a=b;
```

```
    b=d;
```

```
  }
```

```
  return a;
```

```
}
```

```
alert (Nod(50,70) );
```

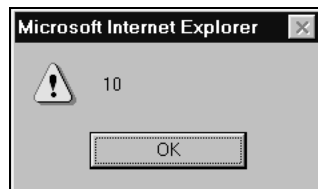


Рис. П2.11

Работа с объектом

Вид конструкции

```
with(объект)
```

```
{
```

```
  команды
```

```
}
```

Описание

Упрощение записи при многократных обращениях к объекту. В блоке {команды} имя объекта при обращении к его методам и свойствам можно не указывать.

Запись

```
var a = Math.PI/2;
```

```
var b = Math.sin(b);
```

равнозначна такой:

```
with(Math)
```

```
{
```

```
  var a = PI/2;
```

```
  var b = sin(b);
```

```
}
```

Блок (составная команда)

Вид конструкции

```
{  
    команды  
}
```

Описание

Используется для обрамления тела функции и там, где синтаксис языка требует одну команду, а надо записать несколько.

Пример

```
// Сколько раз максимальный элемент  
// массива в нем повторяется?  
var a = new Array(90, 2, 58, 90, 87, 90);  
var count = 1;  
var m = a[0];  
for(var i=1; i<a.length; i++)  
{  
    if(a[i]>m)  
    {  
        m = a[i];  
        count = 1;  
    }  
    else if(a[i]==m) count++;  
}  
alert(count);
```

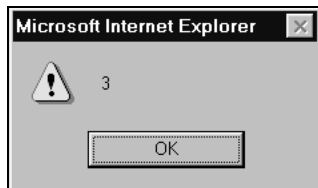


Рис. П2.12

Комментарий

Вид конструкции

```
//строка  
или  
/*строка  
строка  
...  
строка  
строка*/
```

Описание

Комментарий.

Пример

```
// Это комментарий.  
/* Это тоже комментарий. */  
/*  
    Это последний  
    комментарий.  
*/
```

Операции

Операции присваивания

Безусловная операция присваивания

Вид операции

=

Описание

Безусловная операция присваивания. В отличие от Паскаля, в JavaScript (и в языке Си) конструкция:

`переменная = выражение`

является не командой присваивания, а операцией присваивания. Разница в том, что операция имеет значение, и его можно использовать, например, в арифметическом выражении.

Пример

```
var x = 20;  
var y = 30;  
var t;  
var z = 100 + (t=x+y);  
alert("x=" + x + "\ny=" + y + "\nt=" + t +  
      "\nz=" + z);
```



Рис. П2.13

Вид операции

`+=`

`-=`

`*=`

`/=`

и т. д.

Описание

Запись вида:

переменная операция= выражение

эквивалентна следующей записи:

переменная = переменная операция (выражение)

Пример

```
var x = 20;
```

```
x += 80;
```

```
var y = 30;
```

```
y *= x/5;
```

```
var t = 8;
```

```
t %= (x+y)/100;
```

```
var z = (t += 10)+4;
```

```
alert("x=" + x + "\ny=" + y + "\nt=" + t +  
      "\nz=" + z);
```



Рис. П2.14

Условная операция

Вид операции

`?`

Описание

Используется в виде следующей конструкции:

переменная = условие ? выражение1:выражение2

Значение операции есть выражение1, если условие принимает значение **true** и выражение2 в противном случае.

Пример

```
var x = 20;
var y = 30;
var t = x < y ? x : y;
var z = x > y ? x : y;
alert("x=" + x + "\ny=" + y + "\nt=" + t +
      "\nz=" + z);
```



Рис. П2.15

Арифметические операции

Бинарные операции

+
-
*
/
%

Описание

Бинарные операции: сложение, вычитание, умножение, деление, остаток от деления.

Пример

```
var x = 20;
var y = x + 30;
var t = ((x - 10) * y) / 5;
var z = (t + 1) % x;
alert("x=" + x + "\ny=" + y + "\nt=" + t +
      "\nz=" + z);
```

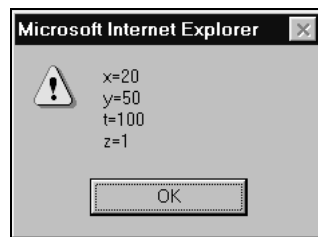


Рис. П2.16

Унарные операции

Вид операции

-
++
--

Описание

Унарные операции: смена знака, увеличение и уменьшение на один. В записях вида ++x и --x изменение значения переменной происходит до ее использования, а в записях вида x++ и x-- после.

Пример

```
var x = -20;
var y = x++;
var t = ++x(++y);
var z = ++x+y++;
alert("x=" + x + "\ny=" + y + "\nt=" + t +
      "\nz=" + z);
```

**Рис. П2.17****Операции сравнения****Операции «равно» и «не равно»****Вид операции**

==

!=

Пример

```
var x = 20;
var y = x==20;
var t = x!=20;
var z = y!=t;
alert("x=" + x + "\ny=" + y + "\nt=" + t +
      "\nz=" + z);
```

**Рис. П2.18****Операции «больше» и «больше или равно»****Вид операции**

>

>=

Пример

```
var x = 20;
var y = x>20;
var t = x>=20;
var z = ++x>20;
alert("x=" + x + "\ny=" + y + "\nt=" + t +
      "\nz=" + z);
```

**Рис. П2.19**

Операции «меньше» и «меньше или равно»

Вид операции

<

<=

Пример

```
var x = 20;
var y = x<20;
var t = x<=20;
var z = --x<20;
alert("x=" + x + "\ny=" + y + "\nt=" + t +
      "\nz=" + z);
```



Рис. П2.20

Логические операции

Логическое И

Вид операции

&&

Описание

Бинарная операция «логическое И». Таблица истинности:

операнд1	операнд2	результат
false	false	false
false	true	false
true	false	false
true	true	true

Пример

```
var x = 20;
var y = x==20;
var t = --x<20;
var z = y && t;
alert("x=" + x + "\ny=" + y + "\nt=" + t +
      "\nz=" + z);
```



Рис. П2.21

Логическое ИЛИ

Вид операции

||

Описание

Бинарная операция «логическое ИЛИ». Таблица истинности:

операнд1	операнд2	результат
false	false	false
false	true	true
true	false	true
true	true	true



Рис.П2.22

Пример

```
var x = 20;
var y = x==20;
var t = x<20;
var z = y || t;
alert("x=" + x + "\ny=" + y + "\nt=" + t +
      "\nz=" + z);
```

Логическое НЕ

Вид операции

!

Описание

Унарная операция «логическое НЕ». Таблица истинности:

операнд	результат
false	true
true	false

Пример

```
var x = 20;  
var y = x==20;  
var t = !(x==20);  
var z = !x;  
alert("x=" + x + "\ny=" + y + "\nt=" + t +  
      "\nz=" + z);
```



Рис. П2.23

Битовые операции

И

Вид операции

&

Описание

Бинарная операция «И» над битами операндов. Соответствующий разряд результата равен 1, если оба разряда операндов равны 1, и равен 0 в противном случае.

Пример

```
var x = 0xFF;  
var y = x & 0x0F;  
var t = x & 0xF0;  
var z = x & 0x00;  
alert("x=" + x + "\ny=" + y + "\nt=" + t +  
      "\nz=" + z);
```



Рис. П2.24

ИЛИ

Вид операции

|

Описание

Бинарная операция «ИЛИ» над битами операндов. Соответствующий разряд результата равен 1, если один из разрядов операндов равен 1, и равен 0 в противном случае.

Пример

```
var x = 0x0F;
var y = x | 0xFF;
var t = x | 0xF0;
var z = x | 0x00;
alert("x=" + x + "\ny=" + y + "\nt=" +
      t + "\nz=" + z);
```



Рис. П2.25

Исключающее ИЛИ

Вид операции

^

Описание

Бинарная операция «исключающее ИЛИ» над битами операндов. Соответствующий разряд результата равен 0, если разряды операндов одинаковы, и равен 1 в противном случае.

Пример

```
var x = 0x0F;
var y = x ^ 0xFF;
var t = x ^ 0xF0;
var z = x ^ 0x0F;
alert("x=" + x + "\ny=" + y + "\nt=" +
      t + "\nz=" + z);
```



Рис. П2.26

Отрицание

Вид операции

~

Описание

Унарная операция «отрицание» над битами операндов.

Пример

```
var x = 0x0F;
var y = 0xFF;
var t = ~x;
var z = ~y;
alert("x=" + x + "\ny=" + y + "\nt=" + t +
      "\nz=" + z);
```



Рис. П2.27

Сдвиг влево

Вид операции

<<

Описание

Бинарная операция «сдвиг влево». Биты первого операнда сдвигаются влево на величину, задаваемую вторым операндом. Младшие «сдвинутые» разряды первого операнда замещаются нулями.

Пример

```
var x = 0x01;
var y = x<<1;
var t = x<<2;
var z = x<<3;
alert("x=" + x + "\ny=" + y + "\nt=" + t +
      "\nz=" + z);
```



Рис. П2.28

Сдвиг вправо

Вид операции

>>

Описание

Бинарная операция «сдвиг вправо». Биты первого операнда сдвигаются вправо на величину, задаваемую вторым операндом. Старшие «сдвинутые» разряды первого операнда замещаются нулями для положительного числа и единицами для отрицательного.

Пример

```
var x = 0x10;
var y = -0x10;
var t = x>>4;
var z = y>>4;
alert("x=" + x + "\ny=" + y + "\nt=" + t +
      "\nz=" + z);
```



Рис. П2.29

Порядок выполнения операций

Имеет приоритет операция, занимающая в нижеприведенной табл. П2.1 более высокую строку. Если операции находятся в одной строке, то порядок их выполнения указан во втором столбце таблицы.

Таблица П2.1

Операция	Описание
Первичные: () [] вызов функции	Слева направо
Унарные: ! ~ ++ -- - + new	Справа налево
Мультипликативные: * / %	Слева направо
Аддитивные: + -	Слева направо
Сдвиги: << >>	Слева направо
Сравнения: < <= > >=	Слева направо
Сравнения на равенство: == !=	Слева направо
Поразрядные: &	Слева направо
Поразрядные: ^	Слева направо
Поразрядные:	Слева направо
Логические: &&	Слева направо
Логические:	Слева направо
Условное выражение: ?	Справа налево
Присваивание:	Справа налево
= += -= /= *= и т.д.	
Запятая: ,	Слева направо

Встроенные функции

Вид функции

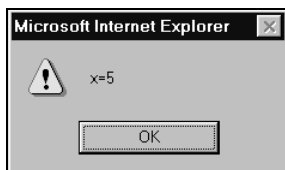
eval (строка)

Описание

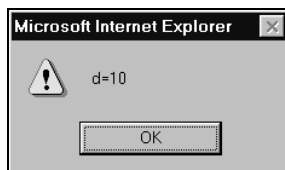
Функция выполняет переданную ей строку так, как если бы это было JavaScript-выражение или JavaScript-программа. Если выполненная строка имеет возвращаемое значение, то оно является значением функции eval.

Пример 1

```
var x = eval("100/5-15");  
alert("x=" + x);
```

**Рис. П2.30****Пример 2**

```
eval("var d = 10; alert('d=' + d);");
```

**Рис. П2.31****Вид функции**

```
parseInt(стр, осн)  
parseInt(стр)
```

Описание

Преобразование строки `стр` в целое число. В качестве параметра `осн` указывается основание системы счисления. Если параметр `осн` отсутствует или система счисления конфликтует с первыми символами, JavaScript использует систему счисления, анализируя первые символы строки (табл. П2.2).

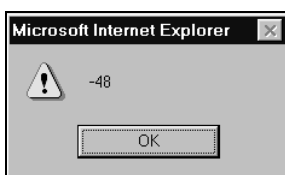
Таблица П2.2.

символы	система
0	8 (восьмиричная)
0x	16 (шестнадцатеричная)
другие	10 (десятичная)

Если `parseInt` сталкивается с недопустимым символом, то возвращает значение, основанное на подстроке, следующей до этого символа, игнорируя все последующие. Если первый же символ не допустим, `parseInt` возвращает значение `NaN`.

Пример 1

```
var num = parseInt("-48.3534");  
alert(num);
```

**Рис. П2.32****Пример 2**

```
var num = parseInt("111", 2);  
alert(num);
```

**Рис. П2.33**

Вид функции

`parseFloat(стр)`

Описание

Преобразование строки `стр` в число с плавающей точкой. Если `parseFloat` сталкивается с недопустимым символом, то возвращает значение, основанное на подстроке, следующей до этого символа, игнорируя все последующие. Если первый же символ не допустим, `parseFloat` возвращает `NaN`.

Пример

```
var num = parseFloat("3.345");  
alert(num);
```

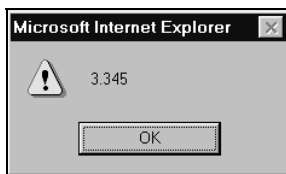


Рис. П2.34

Вид функции

`isNaN(стр)`

Описание

Если строка `стр` не является числом, возвращает `true`, иначе — `false`.

Пример

```
alert(isNaN("3.345"));
```

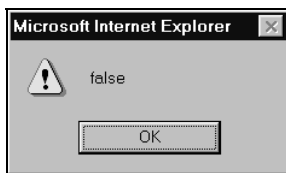


Рис. П2.35

Встроенные объекты

Date

Объект позволяет работать с датой и временем.

Создание экземпляра

```
new Date()
```

```
new Date(year, month, day)
```

```
new Date(year, month, day, hours, minutes, seconds)
```

Экземпляр, созданный без использования параметров, содержит текущую дату и время. Параметры `month`, `day`, `year`, `hours`, `minutes`, `seconds` являются целыми числами и задают соответствующие части даты и времени. Если часы, минуты и секунды не заданы, они устанавливаются в ноль (табл. П2.3).

Таблица П2.3

Параметр	Значение	Диапазон
<code>year</code>	Год	Например, 2003
<code>month</code>	Месяц	0..11 (январь..декабрь)
<code>day</code>	День месяца	1..31
<code>hours</code>	Часы	0..23
<code>minutes</code>	Минуты	0..59
<code>seconds</code>	Секунды	0..59

Методы

Вид метода

```
getFullYear()
```

Описание

Возвращает год. Значение года, принадлежащее XX веку, возвращается в виде двух последних цифр.

Пример

```
var d = new Date();
```

```
var y = d.getFullYear();
```

```
alert(y);
```



Рис. П2.36

Вид метода

```
setYear(year)
```

Описание

Устанавливает год.

Пример

```
var d = new Date();  
d.setYear(1999);  
var y = d.getYear();  
alert(y);
```

Вид метода

```
getMonth()
```

Описание

Возвращает месяц.

Пример

```
var d = new Date();  
var m = d.getMonth();  
alert(m);
```

Вид метода

```
setMonth(month)
```

Описание

Устанавливает месяц.

Пример

```
var d = new Date();  
d.setMonth(4)  
var y = d.getMonth();  
alert(y);
```

Вид метода

```
getDate()
```

Описание

Возвращает день месяца.

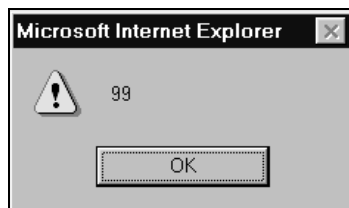


Рис. П2.37

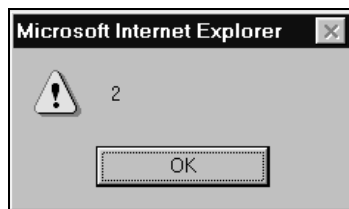


Рис. П2.38

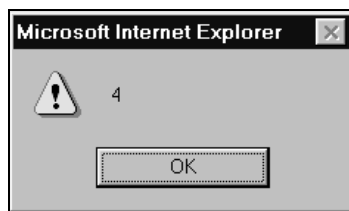


Рис. П2.39

Пример

```
var d = new Date();  
var x = d.getDate();  
alert(x);
```

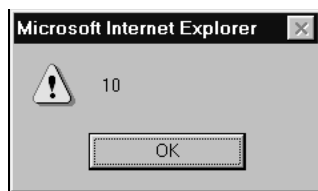


Рис. П2.40

Вид метода

setDate(day)

Описание

Устанавливает день месяца.

Пример

```
var d = new Date();  
d.setDate(2);  
var x = d.getDate();  
alert(x);
```

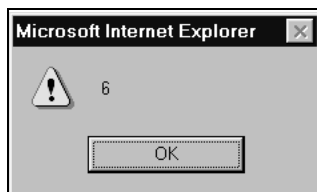


Рис. П2.41

Вид метода

getDay()

Описание

Возвращает день недели (0 — воскресенье, 1 — понедельник, ... 6 — суббота).

Пример

```
var d = new Date();  
var x = d.getDay();  
alert(x);
```

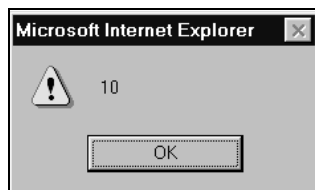


Рис. П2.42

Вид метода

getHours()

Описание

Возвращает час.

Пример

```
var d = new Date();  
var t = d.getHours();  
alert(t);
```

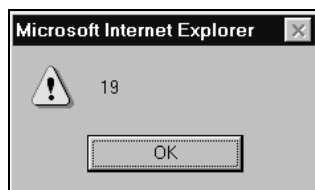


Рис. П2.43

Вид метода

```
setHours()
```

Описание

Устанавливает час.

Пример

```
var d = new Date();  
d.setHours(22);  
var x = d.getHours();  
alert(x);
```

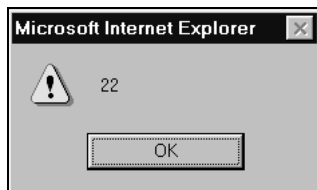


Рис. П2.44

Вид метода

```
getMinutes()
```

Описание

Возвращает минуты.

Пример

```
var d = new Date();  
var m = d.getMinutes();  
alert(m);
```

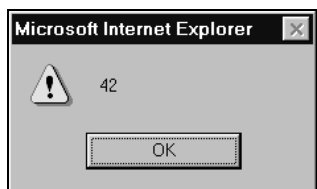


Рис. П2.45

Вид метода

```
setMinutes(minutes)
```

Описание

Устанавливает минуты.

Пример

```
var d = new Date();  
d.setMinutes(30);  
var m = d.getMinutes();  
alert(m);
```

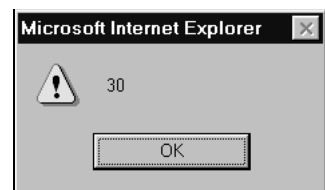


Рис. П2.46

Вид метода

```
getTime()
```

Описание

Возвращает число миллисекунд, прошедших с 0 часов 1 января 1970.

Пример

```
var d = new Date();  
var t = d.getTime();  
alert(t);
```



Рис. П2.47

Вид метода

```
setTime(time)
```

Описание

Устанавливает число миллисекунд, прошедших с 0 часов 1 января 1970.

Пример

```
var d = new Date();  
d.setTime(0);  
var t = d.getMinutes();  
alert(t + "\nYear=" + d.getYear() + "\nMonth=" +  
      d.getMonth());
```



Рис. П2.48

Array

Создание экземпляра

```
new Array()  
new Array(element1, element2, ..., elementN)
```

Параметры `element1`, `element2`, ..., `elementN` задают элементы массива. Конструктор без параметров создает пустой массив. Экземпляр объекта создается и при таком описании:

```
var set = ["скольжение", "причитание", "уможжение", "пиление"];
```

Эта запись эквивалентна следующей:

```
var set = new Array("скольжение", "причитание", "уможжение", "пиление");
```

Нумерация элементов массива начинается с нуля. Для доступа к отдельным элементам массива используют конструкцию:

```
имя_массива[индекс]
```

Например, `set[0]` — это элемент "скольжение".

Длина массива (число элементов в нем) может меняться во время работы программы.

```
set[10] = "хроматика";  
alert(set);
```

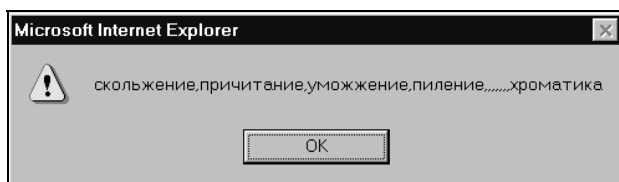


Рис. П2.49

Методы

Вид метода

`concat(array)`

Описание

Возвращает массив, полученный добавлением массива `array`. Исходный массив не меняется.

Пример

```
var set1 = new Array (1,2);  
var set2 = new Array (3,4);  
var set  = set1.concat(set2);  
alert(set1+"\n"+set2+"\n"+set);
```



Рис. П2.50

Вид метода

`join(разделитель)`

Описание

Возвращает строку элементов массива, разделенных значением разделитель. Исходный массив не меняется.

Пример

```
var set = new Array (1,2,3,4,5);  
var arrayText = set.join(" + ");  
alert(set+"\n"+arrayText);
```



Рис. П2.51

Вид метода

```
reverse()
```

Описание

Переставляет элементы в массиве так, что первый элемент становится последним.

Пример

```
var set = new Array (1,2,3);  
set.reverse();  
alert(set);
```

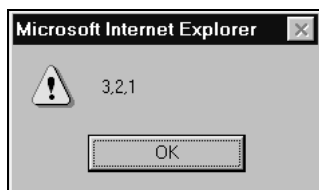


Рис. П2.52

Вид метода

```
slice(ind1,ind2)  
slice(ind1)
```

Описание

Возвращает массив, полученный «вырезанием» из исходного массива части с позиции `ind1` по позицию `ind2-1`. То есть в массиве `set` вырезаются элементы с `set[ind1]` по `set[ind2-1]`. Исходный массив не меняется.

- ❑ Если `ind2` меньше нуля, то отсчет последнего участника выделения ведется с конца массива. Вырезаются элементы с `set[ind1]` по `set[set.length-ind2-1]`.
- ❑ Если `ind2` опущен, выделение продолжается до конца массива, т. е., вырезаются элементы с `set[ind1]` по `set[set.length-1]`.

Пример

```
var set = new Array (0,1,2,3);  
var set1 = set.slice(1,3);  
var set2 = set.slice(1,-1);  
var set3 = set.slice(1);  
alert("set=" + set + "\nset1=" + set1 +  
      "\nset2=" + set2 + "\nset3=" + set3);
```



Рис. П2.53

Вид метода

```
sort(function)
sort()
```

Описание

Сортировка массива. Параметром является имя функции, которая задает правила сравнения двух элементов. Если параметр опущен, элементы сортируются в лексикографическом порядке.

Пример

```
var set = new Array("zebra", "ant", "dog", "cat");
set.sort();
alert(set);
```



Рис. П2.54

Функция `function` должна содержать два аргумента и возвращать:

- ☐ отрицательное число, когда первый аргумент считается расположенным левее второго в смысле определяемого порядка;
- ☐ 0, когда аргументы считаются равнозначными в смысле определяемого порядка;
- ☐ положительное число, когда первый аргумент считается правее второго в смысле определяемого порядка.

Пример

```
var set = new Array (26,71,9,1);
function Compare(a,b)
```

```
{  
    return a-b;  
}  
set.sort(Compare);  
alert(set);
```

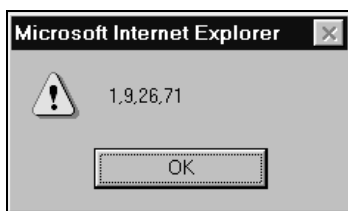


Рис. П2.55

Свойства

`length`

Длина массива (число элементов в нем).

Пример

```
var set = new Array(0,1,2,3,4,5,6,7,8,9,10);  
alert(set.length);
```

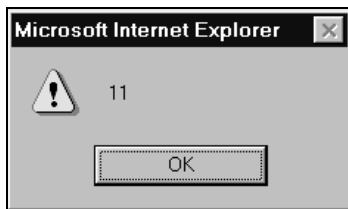


Рис. П2.56

String

Создание экземпляра

```
new String(str)
```

Параметр `str` задает строковое выражение. Экземпляр создается по умолчанию при таком описании переменной:

```
var str = "Сверкалось... Скойкие суды волчились у развел.";
```

Эта запись эквивалентна следующей:

```
var str = new String("Сверкалось... Скойкие сюды волчились у развел.");
```

Методы

Вид метода

`charAt(ind)`

Возвращает символ из указанной позиции строки (нумерация с нуля).

Пример

```
var str = new String("абвгд");  
var ind = 1;  
alert(str.charAt(ind));
```

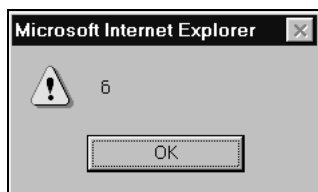


Рис. П2.57

Вид метода

`indexOf(subStr, startInd)`

`indexOf(subStr)`

Поиск подстроки `subStr`, начиная с позиции `startInd` и до конца строки. Метод возвращает позицию, с которой начинается подстрока. Если `startInd` не задан, поиск начинается с нулевой позиции. Если подстрока не найдена, метод возвращает `-1`.

Пример

```
var str = new String("чрезвычайный");  
var ind = str.indexOf("чай", 7);  
alert(ind);
```

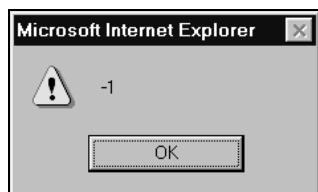


Рис. П2.58

Вид метода

```
lastIndexOf(subStr, startInd)
```

```
lastindexOf(subStr)
```

Поиск подстроки `subStr`, начиная с позиции `startInd` и до начала строки. Метод возвращает позицию, с которой начинается подстрока. Если `startInd` не задан, поиск начинается с последней позиции строки к ее началу. Если подстрока не найдена, метод возвращает `-1`.

Пример

```
var str = new String("чрезвычайный");  
var ind = str.lastIndexOf("чай", 10);  
alert(ind);
```

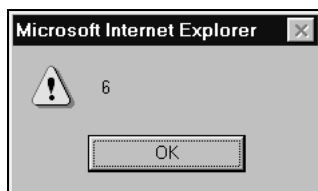


Рис. П2.59

Вид метода

```
substring(ind1, ind2)
```

```
substring(ind1)
```

Возвращает подстроку от позиции `ind1` до позиции `ind2-1`.

- ☐ если `ind1` меньше нуля, он полагается равным нулю;
- ☐ если `ind2` больше длины строки, он полагается равным длине строки;
- ☐ если `ind1` равен `ind2`, возвращается пустая строка;
- ☐ если `ind2` опущен, возвращается конец строки, начиная с позиции `ind1`.

Пример

```
var str = new String("чрезвычайный");  
var sub = str.substring(6, 9);  
alert("str="+str+"\nsub="+sub);
```



Рис. П2.60

Вид метода

`toLowerCase()`

Описание

Возвращает строку, преобразованную к нижнему регистру (исходная строка никак не меняется).

Пример

```
var str = "This is Example";  
var sub = str.toLowerCase();  
alert("str="+str+"\nsub="+sub);
```

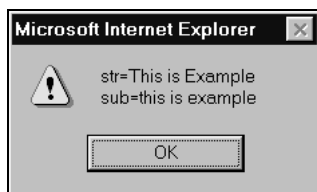


Рис. П2.61

Вид метода

`toUpperCase()`

Описание

Возвращает строку, преобразованную к верхнему регистру (исходная строка никак не меняется).

Пример

```
var str = "This is Example";  
var sub = str.toUpperCase();  
alert("str="+str+"\nsub="+sub);
```

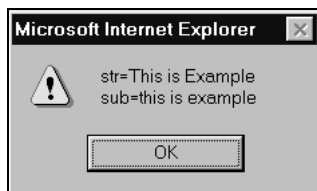


Рис. П2.62

Свойства

`length`

Описание

Длина строки.

Пример

```
var str = "Бег от н\я";  
alert(str.length);
```

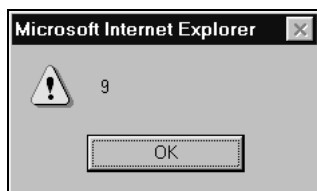


Рис. П2.63

Math

Создание экземпляра

Объект существует в единственном экземпляре. Программист не может создать свой собственный экземпляр этого объекта.

Методы

Метод

abs(arg)

Описание

Возвращает абсолютное значение аргумента.

Пример

```
var x = -25;  
var y = Math.abs(x);  
alert(y);
```

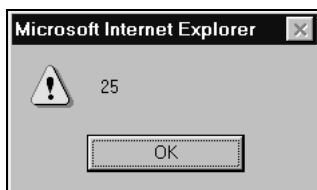


Рис. П2.64

Метод

```
sin(arg)
cos(arg)
tan(arg)
```

Описание

Стандартные тригонометрические функции, аргумент указывается в радианах.

Пример

```
var x = Math.PI/2;
var y = Math.sin(x);
alert(y);
```

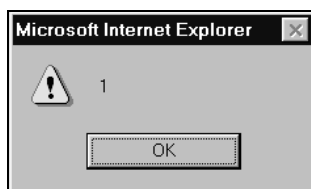


Рис. П2.65

Метод

```
asin(arg)
acos(arg)
atan(arg)
```

Описание

Обратные тригонометрические функции, возвращают значения в радианах.

Пример

```
var x = 0.5;
var y = Math.asin(x);
alert(y);
```



Рис. П2.66

Метод`exp(arg)``log(arg)`**Описание**

Экспонента и натуральный логарифм (основание e).

Пример

```
var x = 1;  
var y = Math.exp(x);  
alert(y);
```



Рис. П2.67

Метод`ceil(arg)`**Описание**

Возвращает ближайшее целое число, большее или равное аргументу.

Пример

```
var x = -2.69;  
var y = Math.ceil(x);  
alert(y);
```

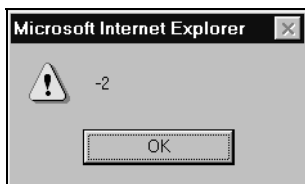


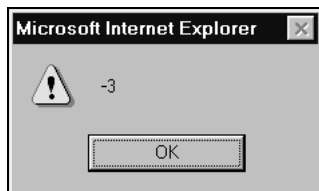
Рис. П2.68

Метод`floor(arg)`**Описание**

Возвращает ближайшее целое число, меньшее или равное аргументу.

Пример

```
var x = -2.69;  
var y = Math.floor(x);  
alert(y);
```

**Рис. П2.69****Метод**

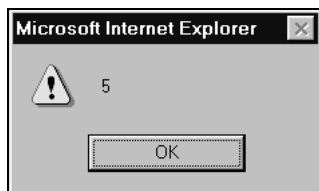
```
min(arg1,arg2)  
max(arg1,arg2)
```

Описание

Минимальное (максимальное) значение из двух числовых аргументов.

Пример

```
var x = 5;  
var y = Math.min(x,10);  
alert(y);
```

**Рис. П2.70****Метод**

```
pow(arg1,arg2)
```

Описание

Возвращает *arg1* в степени *arg2*.

Пример

```
var x = 2;  
var y = Math.pow(x,10);  
alert(y);
```

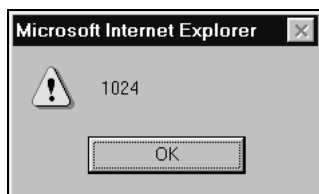


Рис. П2.71

Метод`random()`**Описание**

Возвращает случайное число из диапазона $[0,1]$.

Пример

```
alert(Math.random());
```



Рис. П2.72

Метод`round(arg)`**Описание**

Округляет аргумент до ближайшего целого.

Пример

```
var x = -2.69;  
var y = Math.round(x);  
alert(y);
```

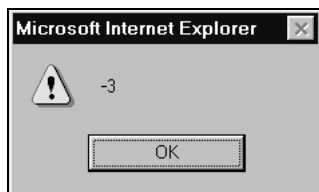


Рис. П2.73

Метод

`sqrt(arg)`

Описание

Корень квадратный из аргумента.

Пример

```
var x = 2;  
var y = Math.sqrt(x);  
alert(y);
```



Рис. П2.74

Свойства

Свойство

`E`

Описание

Основание натурального логарифма (константа Эйлера).

Пример

```
alert(Math.E);
```



Рис. П2.75

Свойство

`LN10`

Описание

Натуральный логарифм числа 10.

Пример

```
alert (Math.LN10) ;
```

**Рис. П2.76****Свойство**

LN2

Описание

Натуральный логарифм числа 2.

Пример

```
alert (Math.LN2) ;
```

**Рис. П2.77****Свойство**

LOG10E

Описание

Логарифм числа E по основанию 10.

Пример

```
alert (Math.LOG10E) ;
```

**Рис. П2.78**

Свойство

LOG2E

Описание

Логарифм числа E по основанию 2.

Пример

```
alert(Math.LOG2E);
```



Рис. П2.79

Свойство

PI

Описание

Число пи.

Пример

```
alert(Math.PI);
```



Рис. П2.80

Свойство

SQRT1_2

Описание

Квадратный корень из 1/2.

Пример

```
alert(Math.SQRT1_2);
```



Рис. П2.81

Свойство

SQRT2

Описание

Квадратный корень из 2.

Пример

```
alert(Math.SQRT2);
```



Рис. П2.82

Приложение 3

Справочник. Объекты и события браузера

window

Объект `window` находится в вершине иерархии и содержит в себе все другие объекты браузера (рис. ПЗ.1).

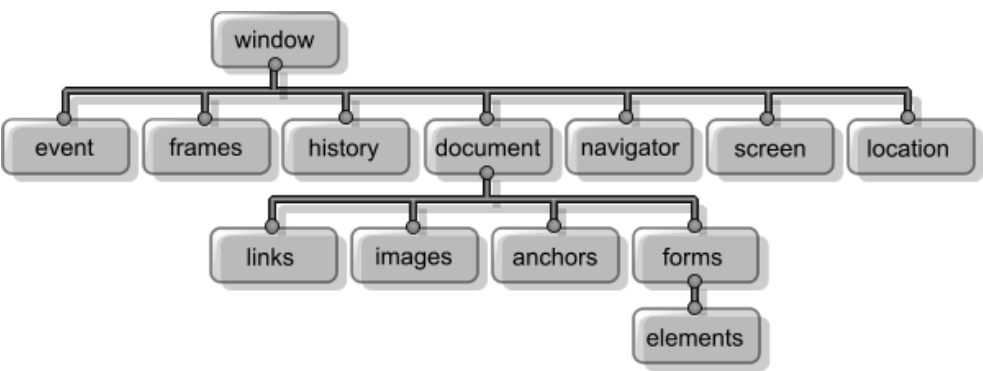


Рис. ПЗ.1. Объектная модель документа

Методы

Таблица ПЗ.1

Метод	Описание
open	Открывает новое окно браузера
close	Закрывает окно браузера

Таблица П3.1 (окончание)

Метод	Описание
alert	Стандартные диалоговые окна
prompt	
confirm	
blur	Уводит фокус из окна
focus	Переводит фокус на окно
scroll	Показывает документ в окне с заданными абсолютными смещениями от его начала
scrollBy	Перемещает документ в окне на заданные величины по отношению к текущему положению
setInterval	Указывает функции выполняться периодически через заданное количество миллисекунд
clearInterval	Отменяет действие метода setInterval
setTimeout	Запускает функцию через заданное количество миллисекунд
clearTimeout	Отменяет действие метода setTimeout

Свойства

Таблица П3.2

Свойство	Описание
self	Указатель на текущее окно
opener	Указатель на окно родителя. Используя этот указатель, можно в созданном окне работать со всем тем, что принадлежит родителю, например, использовать родительские скрипты
status	Содержат соответственно временное и постоянное содержимое статусной строки. Временное содержимое меняется при наезде курсора мыши на ссылку или при выдаче браузером диагностических сообщений
defaultStatus	
returnValue	Определяет возвращаемое из окна значение

Примеры работы с объектом window

Открыть новое окно и закрыть его

```
var переменная = open();  
var переменная = open(файл);  
var переменная = open(файл, имя_окна);  
var переменная = open(файл, имя_окна, параметры_окна);  
  
переменная.close();
```

Здесь:

- переменная — экземпляр создаваемого окна;
- файл — строка, имя файла, который нужно отобразить в созданном окне. Если параметр не задан, создается пустое окно;
- имя_окна — строка, имя окна; используется для обращения к созданному окну (значение свойства name объекта window);
- параметры_окна — строка, свойства окна. Когда параметр не задан, используются свойства окна по умолчанию.

Параметры окна задаются в строке параметры_окна в виде:

```
"параметр1=значение1,параметр2=значение2,...,параметрN=значениеN"
```

Замечание

Не рекомендуется оставлять пробел после разделительной запятой.

Описание параметров (табл. П3.3).

Таблица П3.3

Параметр	Значение	Описание
width	Число	Ширина окна в пикселах. Минимальное значение —100
height	Число	Высота окна в пикселах. Минимальное значение — 100
left	Число	Задаёт горизонтальную координату левого верхнего угла окна в пикселах
top	Число	Задаёт вертикальную координату левого верхнего угла окна в пикселах

Таблица П3.3 (окончание)

Параметр	Значение	Описание
scrollbars	yes, no, 1, 0	Позволяет задать отображение линейек прокрутки
resizable	yes, no, 1, 0	Позволяет указать, может ли окно менять свой размер
menubar	yes, no, 1, 0	Позволяет указать, отображается ли полоса меню
location	yes, no, 1, 0	Позволяет указать, отображается ли полоса для ввода адреса
status	yes, no, 1, 0	Позволяет указать, отображается ли полоса статуса
toolbar	yes, no, 1, 0	Позволяет указать, отображается ли полоса кнопок инструментального меню
fullscreen	yes, no, 1, 0	Позволяет указать, показывается ли новое окно на полный экран или как обычное окно. По умолчанию — как обычное окно

Пример

```
var win = open("./pic/dog.jpg", "",  
    "width=250,height=300,left=0,top=0,scrollbars=no,resizable=no,"+  
    "menubar=no,location=no,status=no,toolbar=no,fullscreen=no");
```

На рис. П3.2 показано окно, открытое при помощи этого кода.



Рис. П3.2. Создано новое окно

Метод `close` закрывает открытое окно. При этом обращение `close()` или `window.close()` закрывает текущее окно. Обращение `win.close()` или `window.win.close()` — закрывает окно `win`, которое было создано из текущего окна при помощи метода `open`.

Диалоговые окна

```
alert(сообщение);
```

```
var переменная = confirm(сообщение);
```

```
var переменная = prompt(сообщение);
```

```
var переменная = prompt(сообщение, ввод_по_умолчанию);
```

Здесь:

□ `сообщение` — произвольное строковое выражение;

□ `ввод_по_умолчанию` — произвольное строковое выражение.

Функция `alert` выводит на экран модальное окно с заданным сообщением (рис. П3.3). Внутри окна всего одна кнопка **ОК**. Пользователь читает сообщение и убирает окно нажатием на **ОК** или клавишей `<Esc>` на клавиатуре.

Пример

```
alert("\нЯ скачу, но \я скачу иначе,\н"+  
    "По пол\ям, по лужам, по росе...\н"+  
    "Говор\ят: он иноходью скачет.\н"+  
    "Это значит иначе, чем все.\н\н"+  
    "В.Высоцкий");
```

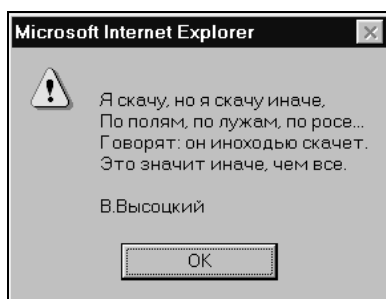


Рис. П3.3. Показ сообщения в окне `alert`

Функция `confirm` выводит на экран модальное окно с заданным сообщением-вопросом (рис. П3.4). Внутри окна — две кнопки: **ОК** и **Отмена**. Функция возвращает:

□ `true`, если нажата кнопка **ОК**;

□ `false`, если нажата кнопка **Отмена** или клавиша `<Esc>`.

Пример

```
if(confirm("Вам квасу?")) alert("Квас кончился\я!");  
else alert("А кваса все равно нет!");
```



Рис. ПЗ.4. Схема работы функции `confirm`

Функция `prompt` выводит на экран модальное окно с заданным сообщением и строкой ввода (рис. ПЗ.5). Внутри окна — две кнопки: **ОК** и **Отмена**. Функция возвращает:

- содержимое строки ввода, если нажата кнопка **ОК**;
- `null`, если нажата кнопка **Отмена** или клавиша `<Esc>`.

Пример

```
var str = prompt("Как теб\я зовут?", "Вен\я");  
if(str==null) alert("А \я хотел познакомиться\я!");  
else alert("Товарищ " + str + "!\nУ теб\я спина бела\я!");
```

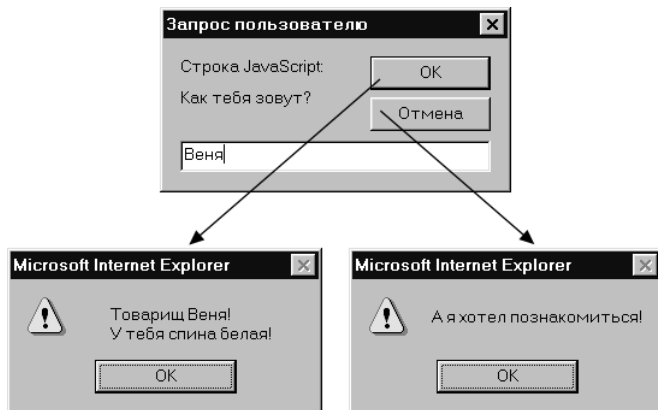


Рис. ПЗ.5. Схема работы функции `prompt`

Перемещение фокуса

Метод `focus` перемещает фокус на окно, по отношению к которому этот метод используется. Когда окно создается, оно автоматически становится активным, получает фокус. Метод `blur` действует обратным образом: уводит фокус из того окна, по отношению к которому применяется.

Скроллинг

Формат команд скроллинга:

```
scroll(координата_по_горизонтали, координата_по_вертикали);
```

```
scrollBy(смещение_по_горизонтали, смещение_по_вертикали);
```

Координаты и смещения задаются в пикселах (без указания единиц измерения).

Метод `scrollBy` производит перемещение документа в окне браузера относительно его текущего положения.

Таймеры

Метод `setTimeout` создает таймер, который выполняет заданные действия по истечении указанного числа миллисекунд. Метод `clearTimeout` выключает созданный таймер.

```
переменная = setTimeout("вызов_функции", число_миллисекунд);
```

```
clearTimeout(переменная);
```

Для выполнения действий, повторяющихся с определенным интервалом времени, используется метод `setInterval`. Метод `clearInterval` отключает действие метода `setInterval`.

```
переменная = setInterval("вызов_функции", число_миллисекунд);
```

```
clearInterval(переменная);
```

В листинге 3.1 приводится код, отображение которого в окне браузера показано на рис. ПЗ.6.

Пример. Часы.

Листинг ПЗ.1

```
<HTML>
```

```
<HEAD>
```

```
<META http-equiv="Content-Type"
```

```
content="text/html; charset=windows-1251">
```

```
<SCRIPT language="JavaScript">
```

```

<!--
var wait = false; // Флаг для включения задержки показа даты.
var tim;           // Указатель на таймер.
// Переключение показа даты на показ времени.
function pusuk()
{
    wait = false;
    showTime();
}
// Показ времени.
function showTime()
{
    if(wait) window.setTimeout("pusuk()", 2000);
    else
    {
        var time = new Date();
        var hour = time.getHours();
        var minute = time.getMinutes();
        var second = time.getSeconds();
        var temp = hour;
        temp += ((minute < 10)? ":0":":") + minute;
        temp += ((second < 10)? ":0":":") + second;
        document.forma.clock.value = temp;
    }
}
// Показ даты.
function Curdate()
{
    wait = true;
    var now = new Date();
    document.forma.clock.value =
        now.getDate()+"/"+(now.getMonth()+1)+"/"+ now.getFullYear();
}
//-->
</SCRIPT>
<TITLE>Часы</TITLE>
</HEAD>
<BODY bgcolor=white text=black

```

```

    onload="tim=setInterval('showTime()', 1000);"
    onunload="window.clearInterval(tim)">
<H1>Часы</H1>
<FORM name=forma>
    <INPUT type=button name=clock value = "*****"
        onclick="Curdate();">
</FORM>
</BODY>
</HTML>

```

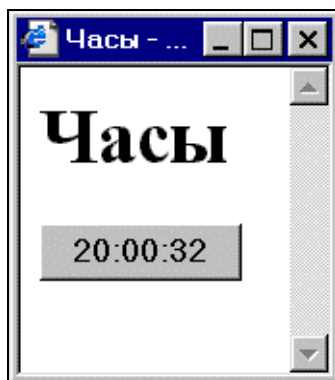


Рис. ПЗ.6. Часы

document

Методы

Таблица ПЗ.4

Метод	Описание
<code>open()</code>	Открывает запись в окне браузера. Прежнее содержимое окна очищается
<code>close()</code>	Закрывает запись в окне браузера
<code>clear()</code>	Очистка окна браузера
<code>write(str)</code>	Записывает текст и код HTML, содержащийся в строке <code>str</code> в документ

Таблица ПЗ.4 (окончание)

Метод	Описание
<code>writeln(str)</code>	Записывает текст и код HTML, заканчивающийся возвратом каретки (переходом на новую строку). Переход на новую строку на экране браузера будет замечен лишь в том случае, если он выполняется внутри тега <code><PRE></code>. Методы <code>write</code> и <code>writeln</code> удобно использовать в следующих случаях: ☐ короткий скрипт заменяет собой длинный HTML-код; ☐ документ строится, учитывая особенности браузера, разрешения экрана, даты, предпочтения пользователя и т. д.; ☐ создание документов полностью программным путем («на лету»).

Свойства

Таблица ПЗ.5

Свойство	Описание
<code>title</code>	Название документа, определенное в теге <code><TITLE></code> . В IE это свойство можно читать и менять, в NN — только читать
<code>URL</code>	Адрес страницы. В IE это свойство можно читать и менять, в NN — только читать
<code>location</code>	Адрес страницы. Это свойство можно читать и менять как в IE, так и в NN
<code>lastModified</code>	Дата последнего изменения документа
<code>bgColor</code>	Цвета фона, текста, непосещенной ссылки, активной ссылки и посещенной ссылки (аналоги атрибутов <code>bgcolor</code> , <code>text</code> , <code>link</code> , <code>alink</code> , <code>vlink</code> тега <code><BODY></code>). Эти свойства доступны для чтения и записи
<code>fgColor</code>	
<code>linkColor</code>	
<code>alinkColor</code>	
<code>vlinkColor</code>	
<code>readyState</code>	Возвращает значение <code>complete</code> после полной загрузки документа. Свойство поддерживается только в IE

Коллекции

Кроме рассмотренных выше свойств и методов (их список здесь не полон) объект `document` имеет еще *коллекции*. Коллекция — это массив ссылок на однородные объекты, каждый из которых имеет одинаковые методы и свойства.

Индексация традиционно начинается с нуля. В качестве индекса можно использовать имя объекта, если оно задано в соответствующем теге при помощи атрибута `name` (или `id`). Например, доступ к объекту, построенному браузером для тега `<IMG name=pic...>` может быть получен при помощи конструкции `document.images[0]`, если это первая картинка в HTML-файле или при помощи конструкции `document.images["pic"]` независимо от порядкового номера картинки. Последний способ, конечно удобнее. Допускается равнозначная запись `document.images.pic`.

Приведенный пример иллюстрирует использование коллекции `images` — в ней собраны объекты, построенные браузером для всех картинок, задаваемых в документе тегами `<IMG>`.

У каждой коллекции (как единого целого) существует свойство `length` — ее длина. Например, доступ к свойству `src` объекта, построенного для первого тега `<IMG>` можно задать при помощи конструкции `document.images[0].src`. Для последнего тега `<IMG>` — `document.images[document.images.length-1].src`.

В общем случае индексом коллекции может быть выражение. В приведенном ниже примере в переменную `widthpic` будет записана ширина картинки, заданной тегом `<IMG>` с именем `mypic`:

```
var ind = "pic";  
var widthpic = document.images["my"+ind].width;
```

Ниже приводится описание четырех наиболее употребительных коллекций, которые принадлежат объекту `document`.

all

Описание

Коллекция всех тегов в теле документа.

Пример

Страница, код которой приводится ниже (листинг П3.2), показывает в окне `alert` все свои теги.

Листинг П3.2

```
<HTML>
<HEAD>
  <TITLE>Теги документа</TITLE>
  <SCRIPT language=JavaScript>
  <!--
    function ShowTags()
    {
      var str = "";
      for(var i=0; i<document.all.length; i++)
        str += document.all[i].tagName+"\n"
      alert(str);
    }
  //-->
</SCRIPT>
</HEAD>
<BODY bgcolor=white text=black>
  <FORM>
    <INPUT type=button value="Показать теги"
      onclick="ShowTags()" ">
  </FORM>
</BODY>
</HTML>
```



Рис. П3.7

forms

Описание

Коллекция всех форм на странице.

Каждый элемент этой коллекции сам содержит коллекцию `elements` — список элементов данной формы (теги `<INPUT>`, `<SELECT>`, `<TEXTAREA>`).

Для доступа к форме с именем `forma` используется конструкция `document.forms["forma"]` (или `document.forms.forma`). Для доступа к элементу формы с именем `element` — конструкция `document.forms["forma"].elements["element"]` или `document.forms.forma.elements.element`

Пример

Пусть поле ввода задается такой формой:

```
<FORM name=f>
  <INPUT name=t type=text
    value="Книги про Незнайку" size=40>
</FORM>
```

При помощи следующего скрипта и формы можно менять содержимое этого поля:

```
<SCRIPT language=JavaScript>
<!--
  var titl = new Array(
    "Приключения Незнайки и его друзей",
    "Незнайка в Солнечном городе",
    "Незнайка на Луне");
  var ind=0;
  function ChangeTitl()
  {
    document.forms.f.elements.t.value = titl[ind];
    ind = (ind+1) %3;
  }
//-->
</SCRIPT>
<FORM>
  <INPUT type=button value=Изменить onclick="ChangeTitl()">
</FORM>
```

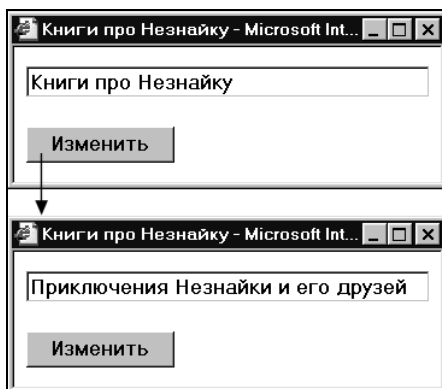


Рис. П3.8

images

Описание

Коллекция всех картинок на странице.

Пример

Пусть картинка в документе задается следующим тегом:

```
<IMG id=pic src=./pic/ispla0.gif
      border=0 alt="" width=89 height=72>
```

При помощи следующего скрипта и формы можно менять экранное изображение:

```
<SCRIPT language=JavaScript>
<!--
var img = new Array (new Image(), new Image());
img[0].src="./pic/ispla0.gif";
img[1].src="./pic/isp2a0.gif";
var ind=1;
function ChangeImg()
{
    document.images.pic.src=img[ind].src;
    ind=1-ind;
}
//-->
</SCRIPT>
<FORM>
  <INPUT type=button value=Изменить onclick="ChangeImg () ">
</FORM>
```

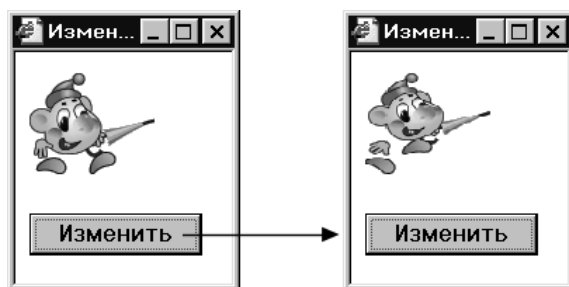


Рис. П3.9

links

Описание

Коллекция всех ссылок на странице.

Пример

Список всех ссылок на странице можно получить при помощи такого кода:

```
<SCRIPT language=JavaScript>
<!--
function LinksList()
{
    var str = "Список ссылок этой страницы:\n ";
    for(var i=0; i<document.links.length; i++)
        str +=document.links[i].href+"\n";
    alert(str);
}
//-->
</SCRIPT>
<FORM>
    <INPUT type=button value=Ссылки onclick="LinksList()" >
</FORM>
```

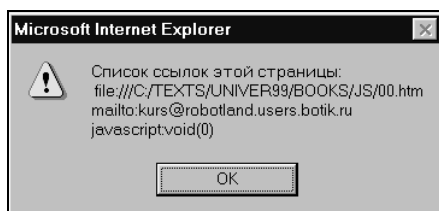


Рис. П3.10

event

Объект `event` позволяет получить детальную информацию о произошедшем событии и выполнить необходимые действия.

Объект `event` доступен только во время возникновения события. Более того, обращаться к нему можно только из обработчиков событий.

Свойства

srcElement

Описание

Позволяет узнать источник события. Это свойство является объектом и само содержит огромное число свойств, описывающих источник, на котором произошло событие. Например, `srcElement` содержит такие свойства:

`srcElement.tagName` — имя тега;

`srcElement.type` — значение атрибута `type`;

`srcElement.value` — значение атрибута `value`.

Пример

Кнопка задается следующим кодом:

```
<FORM>
```

```
<INPUT type=button value="Свойства источника события"
      onclick='alert("tagName="+event.srcElement.tagName+
                    "\ntype="+event.srcElement.type+
                    "\nvalue="+event.srcElement.value) '>
```

```
</FORM>
```

Щелчок мыши по этой кнопке выводит в окно `alert` сообщение, показанное на рисунке.



Рис. П3.11

type

Описание

Строка, содержащая тип события.

Пример

Кнопка задается следующим кодом:

```
<FORM>
```

```
  <INPUT type=button value=type  
        onclick="alert(event.type)">
```

```
</FORM>
```

Щелчок мыши по этой кнопке выводит в окно `alert` сообщение, показанное на рисунке.

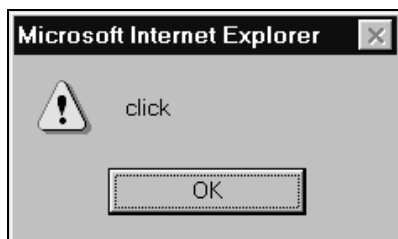


Рис. ПЗ.12

clientX, clientY

Описание

Координаты события относительно начала окна, в котором браузер показывает документ.

screenX, screenY

Описание

Координаты события относительно начала экрана монитора.

offsetX, offsetY

Описание

Координаты события относительно тега, на котором произошло событие.

x, y

Описание

Координаты события относительно начала документа в окне, в котором браузер показывает этот документ. Начало документа может быть существенно «выше» верхней границы окна и «левее» его левой границы, если работают линейки прокрутки.

button

Описание

Число, указывающее нажатую кнопку мыши:

- ☐ 0 — кнопки не нажаты;
- ☐ 1 — нажата левая (основная) кнопка;
- ☐ 2 — нажата правая кнопка;
- ☐ 3 — нажаты одновременно левая и правая кнопки;
- ☐ 4 — нажата средняя кнопка;
- ☐ 5 — нажаты левая и средняя кнопки;
- ☐ 6 — нажаты правая и средняя кнопки;
- ☐ 7 — левая, средняя и правая кнопки.

keyCode

Описание

Код нажатой клавиши.

Примеры

<Enter>	— 13	<ВЛЕВО>	— 37
<Esc>	— 27	<ВПРАВО>	— 39
<Bs>	— 8	<ВВЕРХ>	— 38
<ПРОБЕЛ>	— 32	<ВНИЗ>	— 40

altKey, ctrlKey, shiftKey

Описание

Содержат значения `true` (соответствующая клавиша удерживается) или `false` (соответствующая клавиша не нажата).

toElement, fromElement

Описание

Элемент, находящийся под курсором мыши, и элемент, вышедший из под курсора мыши. Эти свойства являются объектами и сами содержат различные свойства, среди которых:

- ☐ tagName — имя тега;
- ☐ name — значение соответствующего атрибута.

returnValue

Описание

Позволяет запретить или разрешить стандартное действие, предписанное элементу.

Пример

Ссылка задана следующим кодом:

```
<A href=event01.htm  
    onClick="event.returnValue=confirm('Перейти?') "  
>Переход на другой документ</A>
```

Щелчок по ссылке выводит на экран сообщение:

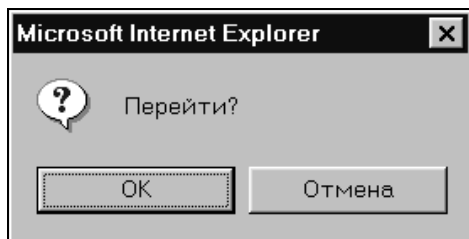


Рис. ПЗ.13

Если пользователь нажимает кнопку **Отмена**, перехода не происходит.

history

Объект `history` содержит информацию об адресах страниц, которые посещались в текущей сессии браузера. Эта информация хранится в списке `History`, длина которого содержится в свойстве `length`.

Методы

Таблица ПЗ.6

Метод	Описание
<code>back()</code>	Загружает предыдущую страницу из списка <code>History</code>
<code>forward()</code>	Загружает следующую страницу из списка <code>History</code>
<code>go(n)</code>	Загружает <i>n</i> -ю страницу из списка <code>History</code> по отношению к текущей. Целое число <i>n</i> может быть положительным, отрицательным или нулем: ❑ $n < 0$ — загружаются предыдущие страницы из <code>History</code>; ❑ $n = 0$ — перегружается текущая страница; ❑ $n > 0$ — загружаются следующие страницы из <code>History</code>. Например, <code>history.go(-1)</code> эквивалентно <code>history.back()</code> , а <code>history.go(1)</code> эквивалентно <code>history.forward()</code> .

Свойства

Таблица ПЗ.7

Свойство	Описание
<code>length</code>	Содержит число элементов списка <code>History</code> , т. е. число разных страниц, которые загружал браузер в свое окно во время текущего сеанса

location

Объект `location` содержит информацию об адресе текущей страницы и обеспечивает методы для ее перезагрузки или загрузки новой страницы.

Свойства

href

Описание

Полный URL-адрес. Свойство доступно для чтения и записи. При записи нового адреса, браузер загружает в окно новую страницу.

Пример

Пусть кнопка задается следующим кодом:

```
<FORM>
```

```
  <INPUT type=button value="location.href"  
        onclick="alert(location.href) ;">
```

```
</FORM>
```

Щелчок мыши по этой кнопке выводит в окно `alert` сообщение, показанное на рисунке.



Рис. ПЗ.14

hash

Описание

Строка, следующая в URL за символом «#» (локальная метка в файле).

host

Описание

Содержит часть URL-адреса: «hostname:port».

hostname

Описание

Содержит часть URL-адреса: «hostname».

pathname

Описание

Содержит часть URL-адреса — путь к файлу после знаков «//» в полном URL).

Пример

Пусть кнопка задается следующим кодом:

```
<FORM>
```

```
  <INPUT type=button value="location.pathname"  
        onclick="alert(location.pathname) ;">
```

```
</FORM>
```

Щелчок мыши по этой кнопке выводит в окно `alert` сообщение, показанное на рис. 3.15.



Рис. П3.15

port

Описание

Содержит часть URL-адреса — номер порта.

protocol

Описание

Содержит начальную часть URL-адреса — протокол (способ загрузки текущей страницы).

Пример

Пусть кнопка задается следующим кодом:

```
<FORM>
```

```
  <INPUT type=button value="location.protocol"
        onclick="alert(location.protocol) ;">
```

```
</FORM>
```

Щелчок мыши по этой кнопке выводит в окно `alert` сообщение, показанное на рис. 3.16.



Рис. П3.16

search

Описание

Содержит часть полного URL-адреса — строку запроса, следующую за символом "?"

Методы

Таблица П3.8

Метод	Описание
<code>assign(URL)</code>	Загружает страницу с заданным URL. Действие метода полностью эквивалентно присваиванию <code>location.href=URL</code>
<code>replace(URL)</code>	Замещает текущую страницу с заданным URL. Запись в списке <code>History</code> , соответствующая текущей странице, замещается на новую. Теперь вернуться к текущей странице по кнопке браузера Назад будет невозможно
<code>reload()</code>	Перегружает текущую страницу. Этот метод можно использовать с аргументом <code>true</code> , и тогда браузер обязательно перегружает страницу с сервера. Если же параметр отсутствует или равен <code>false</code> , браузер сам решает, откуда перегружать страницу — из кеша или сервера

navigator

Объект `navigator` содержит информацию о браузере, в котором выполняется программа.

Свойства

Свойство

`appCodeName`

Описание

Содержит кодовое имя браузера.

Пример

```
alert(navigator.appCodeName);
```

**Рис. П3.17**

Свойство

appName

Описание

Содержит название браузера. Для основных браузеров это:

☐ Microsoft Internet Explorer;

☐ Netscape.

Пример

```
alert(navigator.appName);
```



Рис. П3.18

Свойство

appVersion

Описание

Данное свойство позволяет определять версию браузера.

Пример

```
alert(navigator.appVersion)
```

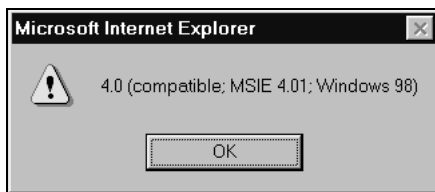


Рис. П3.19

Определение типа браузера

Ниже приводится код функции, которая возвращает:

☐ -ver, для браузера Netscape;

☐ +ver, для браузера Microsoft Internet Explorer;

☐ 0, для других браузеров.

Здесь: «ver» — десятичная дробь, версия браузера.

```
function defBrowser()  
{  
    var ver = parseFloat(navigator.appVersion);  
    var nav = navigator.appName;  
    if(nav == "Netscape") ver = -ver;  
    else if(nav != "Microsoft Internet Explorer") ver = 0;  
    return ver;  
}
```

screen

Объект `screen` содержит информацию о режимах экрана, на котором работает браузер.

Свойства

Свойство

`width`

Описание

Ширина экрана в пикселах.

Пример

```
alert(screen.width);
```



Рис. ПЗ.20

Свойство

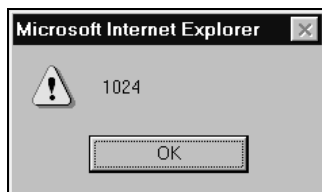
`height`

Описание

Высота экрана в пикселах.

Пример

```
alert(screen.height);
```

**Рис. П3.21****Свойство**

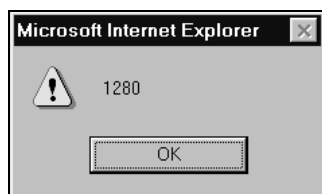
availWidth

Описание

Ширина доступной области экрана в пикселах.

Пример

```
alert(screen.availWidth);
```

**Рис. П3.22****Свойство**

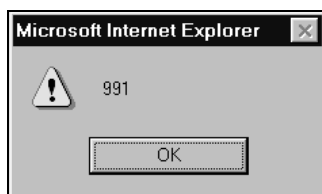
availHeight

Описание

Высота доступной области экрана в пикселах.

Пример

```
alert(screen.availHeight);
```

**Рис. П3.23**

Свойство

colorDepth

Описание

Максимальное число цветов поддерживаемое в системе (число бит на пиксел).

Пример

```
alert (screen.colorDepth);
```



Рис. ПЗ.24

События браузера

Таблица ПЗ.9

Событийный атрибут	Описание
onchange	Событие <code>change</code> происходит при потере элементом фокуса ввода, если его значение было изменено с момента получения фокуса. Этот атрибут может использоваться в тегах <code><INPUT></code> , <code><SELECT></code> и <code><TEXTAREA></code>
onclick	Событие <code>click</code> происходит при однократном щелчке мыши по элементу
ondblclick	Событие <code>dblclick</code> происходит при двойном щелчке мыши по элементу
ondragstart	Событие <code>dragstart</code> происходит, когда пользователь начинает перетаскивание мышью
onfocus onblur	События <code>focus</code> и <code>blur</code> происходят, когда элемент получает или соответственно теряет фокус ввода
onhelp	Событие <code>help</code> происходит, когда нажата клавиша <code><F1></code>

Таблица ПЗ.9 (окончание)

Событийный атрибут	Описание
<code>onkeydown</code> <code>onkeyup</code>	События <code>keydown</code> и <code>keyup</code> происходят при нажатии клавиши на клавиатуре и соответственно отпуске ее
<code>onkeypress</code>	Событие <code>keypress</code> происходит при нажатии клавиши на клавиатуре и отпуске ее
<code>onload</code>	Событие <code>load</code> происходит, когда браузер заканчивает загружать документ в окно. Этот атрибут можно использовать в теге <code><BODY></code> . Обычно в обработчике этого события выполняют какие-либо инициализационные действия: запуск таймера, установку начальных значений переменных. Иногда событие используют для увеличения счетчика посещений страницы, выдачи приветственного сообщения, показа дополнительного окна со справочным материалом
<code>onmousedown</code> <code>onmouseup</code>	События <code>mousedown</code> и <code>mouseup</code> происходят при нажатии кнопки мыши на элементе или соответственно при отпуске кнопки
<code>onmousemove</code>	Событие <code>mousemove</code> происходит, когда пользователь двигает мышь
<code>onmouseover</code> <code>onmouseout</code>	События <code>mouseover</code> и <code>mouseout</code> происходят, когда мышиный курсор перемещается на элемент или соответственно уходит за его пределы
<code>onreset</code>	Событие <code>reset</code> происходит при очистке формы. Оно используется только как атрибут тега <code><FORM></code>
<code>onresize</code>	Событие возникает в момент изменения размеров окна
<code>onscroll</code>	Событие возникает в момент прокрутки окна
<code>onsubmit</code>	Событие <code>submit</code> происходит при отправке формы. Оно используется только как атрибут тега <code><FORM></code>
<code>onunload</code>	Событие <code>unload</code> происходит, когда пользователь закрывает текущий документ в окне браузера (переходит по ссылке, по кнопкам браузера Вперед и Назад , закрывает браузер). Этот атрибут можно использовать в теге <code><BODY></code> . Обычно в обработчике этого события выполняют какие-либо завершающие действия: сообщают время, которое пользователь провел на странице, результаты работы с интерактивными элементами. Очень часто рекламные сайты в этом месте запускают новое окно, выдавая дополнительную порцию рекламных сообщений. Конечно, это раздражает пользователя и в будущем он избегает посещения таких страниц

Приложение 4

Коды испытателей

Испытатель 1. Сравнения (Урок 9)

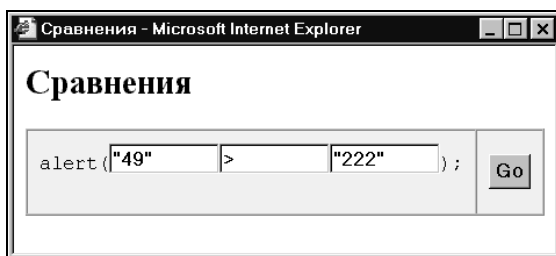


Рис. П4.1. Вид испытателя

Листинг П4.1

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
  <TITLE>Сравнения</TITLE>
</HEAD>
<BODY bgcolor=white text=black>
  <H2>Сравнения</H2>
  <P>
  <FORM>
    <TABLE bgcolor=#F0F0F0 border=1 cellspacing=0 cellpadding=10>
      <TR>
        <TD>
          <PRE>
alert(<INPUT type=text value='"49"' size=10 maxlength=20 name=str1
```

```

    <<INPUT type=text value='' size=10 maxlength=20 name=str2
    <<INPUT type=text value='"222"' size=10 maxlength=20 name=str3>;
</PRE>
    </TD>
    <TD align=center valign=center>
        <INPUT type=button value=Go
            onclick='alert(eval(this.form.str1.value+
                this.form.str2.value+this.form.str3.value))'>

    </TD>
</TR>
</TABLE>
</FORM>
</BODY>
</HTML>

```

Испытатель 2. Открытие окна (Урок 11)

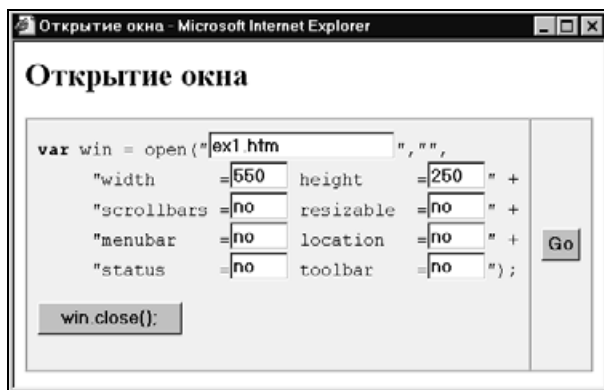


Рис. П4.2. Вид испытателя

Листинг П4.2

```

<HTML>
<HEAD>
    <META http-equiv="Content-Type"
        content="text/html; charset=windows-1251">
    <TITLE>Открытие окна</TITLE>
    <SCRIPT language=JavaScript>
    <!--

```

```

var win = null;
function closeDemo(w)
{
    if(w != null && typeof(w)=="object" && !w.closed) w.close();
}
//-->
</SCRIPT>
</HEAD>
<BODY bgcolor=white text=black>
    <H2>Открытие окна</H2>
    <P>
    <FORM>
        <TABLE bgcolor=#F0F0F0 border=1 cellpadding=10>
            <TR>
                <TD>
                    <PRE>
<B>var</B> win=open("<INPUT type=text value=ex1.htm size=20 name=Turl>",
    "",
    "width      =<INPUT type=text value=550 size=5 name=Twidht
> height      =<INPUT type=text value=250 size=5 name=Theight>" +
    "scrollbars =<INPUT type=text value=no   size=5 name=Tscroll
> resizable   =<INPUT type=text value=no   size=5 name=Tres>" +
    "menubar    =<INPUT type=text value=no   size=5 name=Tmenu
> location    =<INPUT type=text value=no   size=5 name=Tloc>" +
    "status     =<INPUT type=text value=no   size=5 name=Tstatus
> toolbar     =<INPUT type=text value=no   size=5 name=Ttool>");
<INPUT type=button value="win.close();" onClick="closeDemo(win)">
</PRE>
                </TD>
                <TD align=center valign=center>
                    <INPUT type=button value=Go onclick=
                        "win =    open(this.form.Turl.value, '',
                        'width='      +this.form.Twidth.value+', '+
                        'height='     +this.form.Theight.value+', '+
                        'scrollbars=' +this.form.Tscroll.value+', '+
                        'resizable='  +this.form.Tres.value+', '+
                        'menubar='    +this.form.Tmenu.value+', '+
                        'location='   +this.form.Tloc.value+', '+
                        'status='     +this.form.Tstatus.value+', '+
                        'toolbar='    +this.form.Ttool.value);">
                    
```

```

        </TD>
    </TR>
</TABLE>
</FORM>
</BODY>
</HTML>

```

Испытатель 3. Кнопка с секретом (Урок 11)

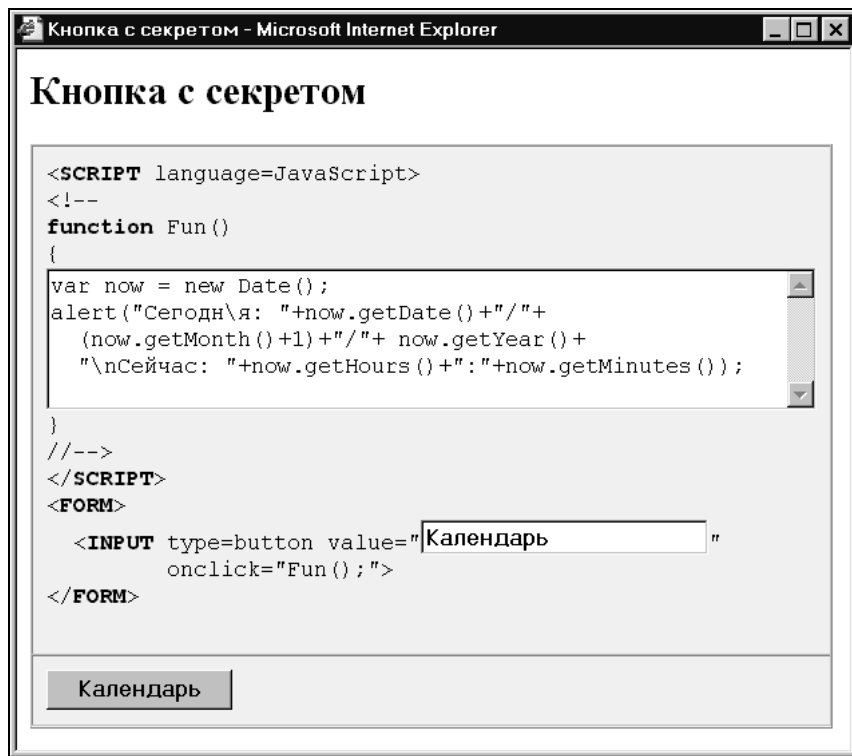


Рис. П4.3. Вид испытателя

Листинг П4.3

```

<HTML>
<HEAD>
    <META http-equiv="Content-Type"
        content="text/html; charset=windows-1251">
    <TITLE>Кнопка с секретом</TITLE>

```

```

</HEAD>
<BODY bgcolor=white text=black>
  <H2>Кнопка с секретом</H2>
  <P>
  <FORM>
    <TABLE bgcolor=#F0F0F0 border=1 cellpadding=10>
      <TR>
        <TD>
          <PRE>
&lt;<B>SCRIPT</B> language=JavaScript&gt;
&lt;!--
<B>function</B> Fun()
{
  <TEXTAREA name=fun rows=5 cols=55>
var now = new Date();
alert("Сегодня: " + now.getDate() + "/" +
  (now.getMonth() + 1) + "/" + now.getYear() +
  "\nСейчас: " + now.getHours() + ":" + now.getMinutes());
</TEXTAREA>
}
//--&gt;
&lt;/<B>SCRIPT</B>&gt;
&lt;<B>FORM</B>&gt;
  &lt;<B>INPUT</B> type=button value="<INPUT type=text value=Календарь
                                size=25 name=T1
                                onchange="this.form.name.value=
                                this.value">"
                                onclick="Fun();"&gt;
&lt;/<B>FORM</B>&gt;
</PRE>
        </TD>
      </TR>
      <TR>
        <TD align=left valign=top>
          <INPUT name=name type=button value=Календарь
            onclick="eval('function Fun(){'+this.form.fun.value+'} Fun();')">
        </TD>
      <TR>
    </TABLE>
  </TD>

```

```
</FORM>
</BODY>
</HTML>
```

Испытатель 4. *value* и *defaultValue* (Урок 12)



Рис. П4.4. Вид испытателя

Листинг П4.4

```
<HTML>
<HEAD>
  <META http-equiv="Content-Type"
    content="text/html; charset=windows-1251">
```

```

<TITLE>value и defaultValue</TITLE>
</HEAD>
<BODY bgcolor=white text=black>
  <H2>value и defaultValue</H2>
  <P>
    <FORM>
      <TABLE bgcolor=#F0F0F0 border=1 cellpadding=10>
        <TR>
          <TD>
            <PRE>
&lt;FORM&gt;
  &lt;INPUT name=str type=text size=40
    value="Почтальон Печкин"&gt;
  &lt;INPUT type=reset value=Сброс&gt;
  &lt;INPUT type=button value=value
    onclick="alert(this.form.str.value);"&gt;
  &lt;INPUT type=button value=defaultValue
    onclick="alert(this.form.str.defaultValue);"&gt;
&lt;/FORM&gt;
</PRE>
          </TD>
        </TR>
        <TR>
          <TD>
            <INPUT name=str type=text size=40 value="Почтальон Печкин"><BR>
            <INPUT type=button value=Сброс
              onclick="this.form.str.value=this.form.str.defaultValue">
            <INPUT type=button value=value
              onclick="alert(this.form.str.value)">
            <INPUT type=button value=defaultValue
              onclick="alert(this.form.str.defaultValue);">
          </TD>
        </TR>
      </TABLE>
    </FORM>
  <P>
    Проделайте с испытателем следующие опыты:
    <UL type=disc>
      <LI>

```

Нажмите последовательно кнопки `<EM>value</EM>` и `<EM>defaultValue</EM>`. Вы увидите значения одноименных свойств объекта `<TT>str</TT>`. Эти значения сейчас совпадают и равны `<TT>"Почтальон Печкин"</TT>`.

```
<LI>
```

Введите в строку новый текст `“Крокодил Гена”`. Теперь значения свойств `<TT>value</TT>` и `<TT>defaultValue</TT>` не совпадают. Почему?

```
</UL>
```

```
</BODY>
```

```
</HTML>
```

Испытатель 5. Угадай пароль (Урок 12)

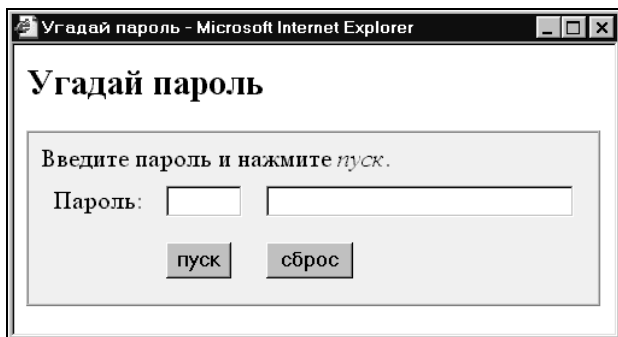


Рис. П4.5. Вид испытателя

Листинг П4.5

```
<HTML>
```

```
<HEAD>
```

```
<META http-equiv="Content-Type"
```

```
content="text/html; charset=windows-1251">
```

```
<TITLE>Угадай пароль</TITLE>
```

```
<SCRIPT language=JavaScript>
```

```
<!--
```

```
var key = "dhtml";
```

```
var mess = new Array("пароль не введен!",  
"пароль правильный",  
"пароль неверный");
```

```
function Ch(forma)
```

```
{
```

```
var ind;
```

```

        if(!forma.pass.value)          ind = 0;
        else if(forma.pass.value==key) ind = 1;
        else                           ind = 2;
        forma.mess.value=mess[ind];
    }
    //-->
</SCRIPT>
</HEAD>
<BODY bgcolor=white text=black>
    <H2>Угадай пароль</H2>
    <FORM>
        <TABLE border=1 bgcolor=#F0F0F0 cellspacing=0 cellpadding=10>
            <TR><TD>
                Введите пароль и нажмите <EM>пуск</EM>.
                <TABLE border=0 cellspacing=0 cellpadding=10>
                    <TR>
                        <TD>Пароль:</TD>
                        <TD>
                            <INPUT type=password size=6 value="" name=pass>
                        </TD>
                        <TD>
                            <INPUT type=text size=30 value="" name=mess>
                        </TD>
                    </TR>
                    <TR>
                        <TD>&nbsp;   </TD>
                        <TD>
                            <INPUT type=button value=пуск onclick="Ch(this.form);">
                        </TD>
                        <TD>
                            <INPUT type=reset value=сброс>
                        </TD>
                    </TR>
                </TABLE>
            </TD></TR>
        </TABLE>
    </FORM>
</BODY>
</HTML>

```

Литература

1. Бранденбау Д. JavaScript: сборник рецептов. СПб., Питер, 2000. 400 с.

В книге приводится серия примеров построения динамических гипертекстовых приложений, имеющих практическую ценность. Коды примеров анализируются построчно с выделением типичных приемов, которыми читатель может пополнить свою программистскую «копилку». Среди описываемых построений: поиск на стороне клиента, электронный экзамен, интерактивная презентация, электронный магазин, контекстная справка.

2. Гудман Дэнни. JavaScript: Библия пользователя. М.—СПб.—Киев, Диалектика, 2002. 950 с.

В этой книге действительно есть все о языке JavaScript и объектной модели документа (в том числе подробно изложена работа со стилируемыми объектами).

Книгу дополняет компакт диск с текстами дополнительных 23 глав (на русском языке), справочными листами (на английском языке) и исходными кодами книжных примеров.

Книга недешевая, но если вы решили всерьез заняться конструированием динамических гипертекстов, придется накопить на нее деньги — покупка стоит того.

3. Хоумер А., Улмен К. Dynamic HTML: справочник. СПб., Питер. 2000. 500 с.

В книге приводятся удобные справочники по свойствам таблиц стилей, объектной модели документа, языкам JavaScript и VBScript.

Книга не ограничивается справочными разделами. В восьми главах, предваряющих справочник, доступно и ясно излагаются основы построения динамических документов. Приводится много иллюстрирующих примеров на языках JavaScript и VBScript.