Вадим Дунаев

Web программирование для всех

bhv

Вадим Дунаев

Web программирование для всех

Санкт-Петербург

«БХВ-Петербург»

2008

УДК 681.3.068
ББК 32.973.26-018.1
Д83

Дунаев В. В.

Д83 Web-программирование для всех. — СПб.: БХВ-Петербург, 2008. — 560 с.: ил.

ISBN 978-5-9775-0197-2

В книге в доступной форме в виде диалогов между Простаком, Занудой и Профессором показано, что такое Web-программирование и в каких случаях его необходимо применять. Изложены основы языков JavaScript и PHP. Описано создание клиентских сценариев на JavaScript, начиная с создания новых окон браузера и заканчивая применением технологии AJAX. Рассмотрены серверные сценарии на языке PHP для работы с файлами и папками, взаимодействия с базами данных и многие другие. Материал сопровождается простыми практическими примерами, которые поддерживаются всеми современными браузерами, такими как Internet Explorer, Mozilla Firefox и Opera.

Для Web-разработчиков

УДК 681.3.06
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Леонид Кочин*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 30.04.08.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 45,15.
Тираж 2500 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.002108.02.07 от 28.02.2007 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0197-2

© Дунаев В. В., 2008
© Оформление, издательство "БХВ-Петербург", 2008

Оглавление

Введение **1**

Пролог **3**

Глава 1. Радости и горести программирования **7**

1.1. Когда Web-странице нужны программы? 7
1.2. С чего и как начать? 10
1.3. Почему программировать интересно? 23
1.4. Неприятности в программировании 24

Глава 2. О языках программирования вообще **31**

2.1. Переменные и типы данных 32
2.2. Массивы данных 39
2.3. Функции 43
2.4. Классы и объекты 49
2.5. Операторы и выражения 57
2.6. Специальные термины 59
2.7. Резюме и напутствие 61

Глава 3. Основы языка JavaScript **65**

3.1. Немного истории о версиях и стандартах 65
3.2. Общая характеристика языка 70
3.3. Как создавать и отлаживать сценарии на JavaScript 71
3.3.1. Вставка сценариев в HTML-документ 72
3.3.2. Подготовка, запуск и отладка сценариев 78
3.4. Ввод и вывод данных 81
3.4.1. Метод *alert — окно предупреждения* 82
3.4.2. Метод *confirm — окно подтверждения* 83
3.4.3. Метод *prompt — окно запроса* 84
3.4.4. Метод *document.write()* 85

3.5. Типы данных ... 86
3.5.1. Примитивные типы данных ... 88
3.5.2. Составные типы данных ... 90
3.5.3. Автоматическое преобразование типов данных ... 92
Преобразование строк (*String*) ... 94
Преобразование чисел (*Number*) ... 95
Преобразование логических значений (*Boolean*) ... 95
Преобразование пустого значения (*null*) ... 95
Преобразование неопределенного значения (*undefined*) ... 95
3.5.4. Принудительное преобразование типов данных ... 96
3.6. Переменные и оператор присваивания ... 100
3.6.1. Имена переменных ... 100
3.6.2. Создание переменных ... 101
3.6.3. Операторы присваивания ... 110
3.6.4. Проверка типа переменной ... 112
3.7. Операторы ... 113
3.7.1. Комментарии ... 113
3.7.2. Арифметические операторы ... 114
3.7.3. Дополнительные операторы присваивания ... 117
3.7.4. Операторы сравнения ... 118
3.7.5. Логические операторы ... 121
3.7.6. Операторы условия ... 122
Оператор *if* ... 123
Оператор условия *?:* ... 126
Оператор *switch* ... 127
3.7.7. Операторы цикла ... 129
Оператор *for* ... 129
Оператор *while* ... 132
Оператор *do-while* ... 134
3.7.8. Об условиях в операторах условия и цикла ... 135
3.7.9. Побитовые операторы ... 135
3.7.10. Другие операторы ... 137
3.7.11. Приоритет операторов ... 137
3.8. Функции ... 139
3.8.1. Встроенные функции ... 140
3.8.2. Пользовательские функции ... 142
3.8.3. Объект *Function* ... 146
3.9. Строки ... 150
3.9.1. Кавычки и специальные символы ... 151
3.9.2. Объект *String* ... 153

3.9.3. Функции вставки и замены подстрок ... 161
3.9.4. Функции удаления ведущих и заключительных пробелов ... 163
3.10. Массивы ... 164
3.10.1. Создание массива ... 165
3.10.2. Многомерные массивы ... 167
3.10.3. Копирование массива ... 168
3.10.4. Объект *Array* ... 169
3.10.5. Функции обработки числовых массивов ... 175
3.11. Числа ... 176
3.11.1. Числа целые и с плавающей точкой ... 176
3.11.2. Объект *Number* ... 180
3.11.3. Объект *Math* ... 182
3.11.4. Функции для решения некоторых математических задач ... 184
Решение квадратного уравнения ... 185
Вычисление интеграла ... 186
Вычисление производной ... 189
Поиск экстремума ... 190
3.12. Дата и время ... 191
3.12.1. Создание объекта *Date* ... 192
3.12.2. Методы объекта *Date* ... 193
3.13. Объекты ... 206
3.13.1. Создание объекта ... 206
3.13.2. Свойства и методы объекта *Object* ... 212
3.13.3. Объектные операторы ... 214
3.14. Операторы обработки исключительных ситуаций ... 217

Глава 4. Клиентские сценарии на JavaScript ... 221

4.1. Об объектной модели браузера и документа ... 221
4.1.1. Общие сведения ... 221
4.1.2. Объект *window* ... 227
Свойства объекта *window* ... 227
Методы объекта *window* ... 228
4.1.3. Объект *document* ... 229
Свойства объекта *document* ... 229
Коллекции объекта *document* ... 231
Методы объекта *document* ... 231
4.1.4. Объект *location* ... 232
Свойства объекта *location* ... 232
Методы объекта *location* ... 232
14.1.5. Объект *history* ... 233
Свойства объекта *history* ... 233
Методы объекта *history* ... 233

4.1.6. Объект *navigator* ... 233
Свойства объекта *navigator* ... 234
Коллекции объекта *navigator* ... 234
Методы объекта *navigator* ... 234
4.1.7. Объект *screen* ... 235
4.2. Доступ к объектам браузера и документа ... 235
4.3. Обработка событий ... 243
4.3.1. Привязка обработчиков событий ... 243
4.3.2. Программный вызов обработчика события ... 247
4.3.3. Изменение поведения элементов по умолчанию ... 249
4.3.4. Прохождение событий ... 250
4.3.5. Информация о событии: объект *event* ... 253
4.4. Окна и фреймы ... 255
4.4.1. Создание новых окон браузера ... 255
4.4.2. Работа с фреймами ... 257
4.4.2. Работа с "плавающими" фреймами ... 263
4.5. Работа с каскадными таблицами стилей ... 263
4.6. Управление во времени ... 269
4.7. Работа с cookie ... 272
4.8. Работа с таблицами ... 278
4.9. Работа с формами ... 281
4.9.1. Проверка данных перед отправкой ... 281
4.9.2. Создание баннера ... 283
4.9.3. Переходы между полями по клавише <Enter> ... 284
4.10. Создание меню ... 286
4.10.1. Меню на основе раскрывающегося списка ... 286
4.10.2. Двухуровневое меню на основе таблиц ... 288
4.11. Перемещение элементов мышью ... 292
4.12. Динамическое изменение содержимого документа ... 297
4.12.1. Изменение свойств, ассоциированных с атрибутами элементов, и свойств стиля ... 298
4.12.2. Предварительная загрузка изображений ... 299
4.12.3. Использование изображения для парольной защиты страницы ... 302
4.12.4. Применение свойства *innerHTML* ... 304
4.12.5. Применение технологии AJAX ... 307
4.13. Распознавание типа браузера ... 317

Глава 5. Основы языка PHP ... 321

5.1. Предварительные сведения ... 322
5.1.1. Где писать сценарии на PHP ... 322
5.1.2. Сообщения об ошибках ... 324

5.1.3. Принудительный выход из сценария ... 324
5.1.4. Справка по PHP ... 325
5.2. Вывод и типы данных ... 325
5.3. Переменные и оператор присваивания ... 330
5.3.1. Имена переменных ... 330
5.3.2. Создание переменных ... 331
5.3.3. Отображение значений переменных ... 333
5.3.4. Переменные переменные ... 337
5.3.5. Область действия переменных ... 338
5.3.6. Проверка существования переменных и их типов ... 340
5.4. Константы ... 341
5.5. Операторы ... 342
5.5.1. Комментарии ... 342
5.5.2. Арифметические операторы ... 343
5.5.3. Строковый оператор ... 345
5.5.4. Дополнительные операторы присваивания ... 345
5.5.5. Операторы сравнения ... 346
5.5.6. Логические операторы ... 347
5.5.7. Побитовые операторы ... 349
5.5.8. Операторы условного перехода ... 349
Оператор *if* ... 349
Оператор *switch* ... 350
Оператор условия *?:* ... 351
5.5.9. Операторы цикла ... 352
Оператор *for* ... 352
Оператор *while* ... 355
Оператор *do-while* ... 356
5.6. Строки ... 357
5.6.1. Двойные и одинарные кавычки ... 357
5.6.2. Склейка строк ... 361
5.6.3. Преобразование строк ... 361
5.6.4. Форматирование строк ... 366
5.7. Числа ... 371
5.7.1. Математические функции ... 371
5.7.2. Математические константы ... 372
5.7.3. Представление чисел в различных системах счисления ... 373
5.7.4. Форматирование чисел ... 375
5.8. Дата и время ... 377
5.9. Массивы ... 381
5.9.1. Создание массива ... 381
5.9.2. Многомерные массивы ... 384

5.9.3. Отображение массивов ... 386
5.9.4. Операции над массивами ... 387
Копирование массивов ... 387
Сортировка массивов ... 387
Перемещение по массиву ... 390
Запись значений элементов массива в переменные ... 393
Преобразование массива в текстовую строку ... 394
Преобразование текстовой строки в массив ... 394
Другие операции над массивами ... 395
5.10. Глобальные предопределенные переменные ... 397
5.11. Функции ... 399
5.11.1. Пользовательские функции ... 399
5.11.2. Переменные функции ... 405
5.11.3. Встроенные функции ... 406
5.11.4. Как узнать, есть ли такая функция ... 406
5.12. Классы и объекты ... 406
5.12.1. Определение класса ... 407
Свойства и методы ... 408
Конструктор ... 409
5.12.2. Применение объектов ... 411
5.12.3. Ограничение доступа к свойствам и методам ... 412
5.12.4. Клонирование и удаление объектов ... 414
5.12.5. Использование методов несозданных объектов ... 415
5.12.6. Обработка исключений ... 416
5.12.7. Пример класса формы ... 417
5.13. Выполнение PHP-кода в текстовых строках ... 419

Глава 6. Основы создания серверных сценариев на PHP ... 421

6.1. Получение данных из форм клиента ... 421
6.1.1. Получение данных из элементов форм ... 422
6.1.2. Передача файлов на сервер ... 430
6.2. Переходы между Web-страницами ... 433
6.2.1. Вывод ссылок ... 434
6.2.2. Применение форм ... 434
6.2.3. Переадресация с помощью функции *header()* ... 435
6.2.4. Добавление информации к URL-адресу ... 436
6.2.5. Применение cookie ... 437
6.2.6. Сеансы ... 439
Создание сеанса ... 439
Особенности сеансов ... 441
Пример организации сеанса ... 442
Защита страниц паролем ... 444

6.3. Работа с графикой ... 447
6.4. Работа с файлами ... 453
6.4.1. Открытие файла ... 454
6.4.2. Закрытие и удаление файлов ... 455
6.4.3. Чтение файла ... 456
Чтение файла в переменную ... 456
Чтение файла в массив ... 457
Чтение файла с удалением тегов HTML ... 457
6.4.4. Запись в файл ... 459
6.4.5. Работа с папками ... 459
6.4.6. Простой счетчик посещений страницы ... 461
6.4.7. Работа с таблицами в текстовых файлах ... 462
Чтение CSV-файла ... 463
Функции для работы с табличными данными ... 465
Сложный счетчик посещений страницы ... 470
Создание баннера ... 475
Создание гостевой книги ... 479
6.5. Работа с базами данных ... 486
6.5.1. Что такое база данных ... 486
6.5.2. Основные средства PHP для взаимодействия с базой данных ... 488
Подключение к базе данных ... 488
Передача запросов к базе данных ... 490
Обработка данных в сценарии ... 491
6.5.3. Создание гостевой книги на основе базы данных ... 492
Создание базы данных ... 493
Создание таблицы для хранения данных ... 493
Определение регистрационного имени и пароля пользователя ... 494
Определение прав пользователя ... 495
Сценарии для взаимодействия с посетителем ... 496
Сценарии для владельца гостевой книги ... 499
6.5.4. Применение SQLite ... 500
6.6. Другие возможности PHP ... 503

ПРИЛОЖЕНИЯ ... 505

Приложение 1. Краткий справочник по HTML и CSS ... 507

П1.1. Теги HTML ... 507
П1.2. Таблицы стилей ... 517
П1.2.1. Единицы измерения ... 517
П1.2.2. Параметры и свойства CSS ... 518
Свойства шрифта ... 518

Свойства цвета и фона ... 521
Свойства списков ... 524
Свойства текста ... 525
Блочные свойства ... 528
Основные свойства печати ... 536
Свойства фильтров ... 537
Свойство *cursor* ... 540

Приложение 2. Установка Web-сервера и PHP ... 541

П2.1. Установка Web-сервера ... 541
П2.2. Установка PHP ... 542
П2.2.1. Установка модуля PHP ... 542
П2.2.2. Настройка модуля PHP ... 543
П2.2.3. Установка расширений PHP ... 545
П2.2.4. Проверка работоспособности Web-сервера и обработчика PHP ... 546

Литература ... 547

Предметный указатель ... 548

Введение

Данная книга посвящена основам программирования клиентских и серверных сценариев для Web-сайтов на языках JavaScript и PHP соответственно. Она рассчитана на широкий круг читателей, впервые столкнувшихся с подобными задачами. Для понимания предлагаемого материала требуется лишь некоторое знакомство с языком HTML для разметки Web-страниц.

Общие темы программирования излагаются в форме диалогов в *главах 1* и *2*. В *главах 3* и *4* рассматриваются язык JavaScript и способы решения основных задач посредством клиентских сценариев. В *главах 5* и *6* рассказывается о языке PHP и создании серверных сценариев. В приложениях даны краткие справочные сведения об HTML и CSS, а также по установке Web-сервера и PHP.

Примеры сценариев на языке JavaScript разработаны таким образом, что выполняются всеми современными браузерами, такими как Internet Explorer 5+, Mozilla Firefox 2+ и Opera 8+.

Из-за ограниченности объема в книге рассматриваются далеко не все задачи создания сценариев и возможности языков JavaScript и PHP. Главная цель книги состоит в том, чтобы ввести читателя в увлекательный мир программирования для Web, и не только. Ваши отзывы и замечания я буду рад принять в своей гостевой книге по адресу **http://dunaevv1.narod.ru**.

Пролог

Простак. Привет, Зануда! Куда это ты спешишь с озабоченным видом?

Зануда. А вот решил обратиться к нашему знакомому Профессору, чтобы он помог мне стать программистом. Я давно ходил кругами около этой темы: разговаривал с друзьями, пробовал читать соответствующие книжки и даже писать кое-какие программки. Однако, признаюсь, кроме разочарования ничего из этого не вышло. Если хочешь, пошли вместе.

Простак. А зачем тебе это? Ты хочешь стать программером, чтобы поступить в какую-нибудь "крутую" программистскую фирму типа Microsoft и зарабатывать много денег?

Зануда. Нет, я пришел к идее заняться программированием, решая обычные задачи, связанные с созданием своего Web-сайта.

Простак. Не понимаю, как сейчас, при современном уровне развития софта можно было озаботиться программированием. Ты, похоже, просто не знаешь о существовании программных систем разработки пользовательских приложений, например, Dreamweaver, FrontPage, Flash, C++-Builder и множества других. Используя эти системы, ничего не придется программировать, достаточно просто нажимать кнопки, соответствующие твоим потребностям разработчика. Все, что получается в результате, появляется на экране компьютера почти в конечном виде, а программный код, заставляющий компьютер все это выполнять, создается и сохраняется автоматически. Ты, конечно, можешь из любопытства посмотреть на него, но зачем?

Зануда. Я знаю, что настоящее взаимодействие с компьютером может осуществить только программист. Чтобы подкрепить это высказывание, попробую воспользоваться следующей аналогией. Так, ты можешь изучать окружающий мир по книгам, туристическим буклетам и рассказам

очевидцев, но вся эта опосредованная информация не способна пробудить в тебе те чувства, которые возникают при непосредственном соприкосновении с явлением, объектом или проблемой, а значит, ты не сможешь мобилизовать свою потенциальную энергию, чтобы решить возникшие задачи. Что бы там мне не рассказывали, я чувствую себя неуверенно или дискомфортно, пока не выясню досконально лично сам, что к чему.

Ты ведь знаешь, что я люблю математику, хотя и не все в ней понимаю достаточно глубоко. Мне кажется, что программирование чем-то похоже на математическое творчество и исследование. Именно эта связь, не вполне ясная для меня, разбудила во мне интерес к программированию. Чтобы программировать, нужно, каждый скажет, знать языки программирования. Я не знаю ни одного из них. Пробовал читать справочники и учебники, но ничего путного из этого не получилось. Понятия плохо запоминаются, а с их применением вообще "труба". Казалось бы, мне все понятно, но мои первые, даже очень примитивные программки, просто не работают. Ситуация похожа на ту, когда я знаю несколько сотен слов иностранного языка, но не могу составить ни одного предложения, которое бы понял иностранец.

П р о с т а к. Я тоже не знаю языков, но не горюю об этом. Заметь, что я даже сделал пару сайтов, заказанных мне коммерческими фирмами, и при этом знания языков программирования мне совершенно не понадобились. Я просто воспользовался одним из известных пакетов для визуальной разработки сайтов. Согласен, что иногда приходится применять чужие программки с открытым кодом. Однако в этих случаях можно не разбираться ни в их деталях, ни, тем более, в языках, на которых они написаны. Если ты где-то нашел какую-то чужую программу, и она работает, то все в порядке. В противном случае, если ее исходный код доступен, ты можешь экспериментально, по интуиции, кое-что чуть-чуть изменить, чтобы твой компьютер ее "заглотил" и выполнил без выдачи непонятных для тебя сообщений об ошибках. Почти все так и делают. Если что-то остается непонятным, можно поискать советы на форумах в Интернете. Согласись, в этом есть что-то романтичное или, лучше сказать, "заводное".

З а н у д а. Итак, как выяснилось, мы оба не знаем языков программирования, но различие между нами в том, что я хочу понять программирование, а ты — нет. Знания, как говорят, плечи не оттягивают.

П р о с т а к. Но еще говорят, что от многих знаний много печали. Ладно, я тоже не прочь немного расширить границы своей эрудиции. Поэтому могу составить тебе компанию. Однако предупреждаю, если беседы с Профессором окажутся слишком занудными, то я ретируюсь.

З а н у д а. Хорошо. Насколько я знаю, программирование — очень широкая область деятельности, и его технологии существенно зависят от того, какого рода приложения (программы) требуется создавать. Чтобы не распыляться, предлагаю ограничиться созданием приложений для Интернета, точнее для Всемирной паутины, т. е. для Web. Другими словами, давай сначала познакомимся с программированием Web-приложений. Эта тема актуальна и для новичков, и для более опытных разработчиков. Я хотел бы понять, действительно ли можно взяться за это дело с нулевой предварительной подготовкой и довольно быстро достичь более или менее хороших результатов, или же требуются долгие годы и недюжинные усилия, чтобы стать профессиональным разработчиком Web-приложений. Попутно мне хотелось бы узнать, насколько полученные знания могут пригодиться для разработки программ, непосредственно не связанных с Web.

Глава 1

Радости и горести программирования

Профессор. Уважаемые Зануда и Простак! Что ж, я готов поговорить с вами о программировании, как вообще, так и применительно к задачам создания Web-приложений, в частности. А начну я, пожалуй, с нескольких замечаний о том, почему желательно уметь хотя бы немного программировать, на какие жертвы ради этого предстоит пойти, и какова будет награда. Затем мы рассмотрим, как лучше и быстрее этому научиться.

1.1. Когда Web-странице нужны программы?

Профессор. Вы, должно быть, уже знаете, что Web-страницу или даже многостраничный сайт можно сделать, вообще не прибегая к программированию, т. е. составлению вручную текстов программ на специальных языках. Другими словами, для создания Web-страницы в настоящее время нет прямой необходимости заниматься написанием кодов на таких языках, как HTML, JavaScript и др.

Простак. Да, широко известно, что для этой цели существует не одна система визуальной разработки сайтов, например, Dreamweaver, FrontPage, CofeeCap и др. Об этом я неоднократно твердил Зануде, но он хотел чего-то большего.

Зануда. Однако подобные системы-конструкторы (системы визуальной разработки) автоматически генерируют программные коды на языке HTML, соответствующие нашим манипуляциям мышью и клавиатурой во время "рисования" Web-страницы на экране монитора. Мы можем просмотреть и при желании вручную откорректировать эти коды (если мы знаем язык HTML и JavaScript), а затем увидеть результат на экране. Тем не менее, я слышал, что вполне можно обойтись без этих специальных средств, а просто написать коды HTML и других языков в обычном текстовом редакторе, сохранить их в файле с расширением htm или html, а затем открыть этот файл

в Web-браузере, например, Microsoft Internet Explorer, Opera, Mozilla Firefox и др. Если все правильно, то в результате в окне браузера отобразится соответствующая Web-страница. Впрочем, любой современный браузер предоставляет возможность взглянуть на код загруженной в него страницы. Таким образом, можно избежать использования систем-конструкторов, а делать все вручную, сохраняя полный контроль над своим проектом.

Простак. Конечно, можно и отказаться от технологических оснасток, но зачем? Если что-то можно сделать более наглядным, простым и быстро достижимым с помощью специальных инструментов, то по-чему бы ими не воспользоваться?

Профессор. Вы оба правы. Профессионала от любителей отличает то, что он в работе всегда пользуется наиболее подходящим средством, а в своем арсенале держит только хорошие инструменты. Что-то он делает вручную, "колдуя" над кодами, а что-то — с помощью инструментов визуальной разработки. Тем не менее, мы собрались здесь для обсуждения не того, как и с помощью каких оснасток следует разрабатывать сайты вообще, а как при этом создавать программы, т. е. встраиваемые в HTML-документы сценарии (скрипты). Уточню, говоря о встраиваемых в HTML-код программах, я имею в виду программные коды, написанные не на HTML, а на языках программирования, подобных тем, на которых пишутся настоящие приложения.

Как известно, "несущую конструкцию" Web-страницы обычно создают посредством языка HTML. С помощью HTML в Web-страницу (Web-документ) вставляют обычные тексты, гиперссылки, таблицы, графические изображения и другие элементы, например, звук, видео, Flash-ролики, кнопки, поля ввода данных и т. п. Вместе с тем, в HTML-коде возможны программные вставки, написанные на совсем другом языке программирования, например, JavaScript. Эти вставки, называемые еще сценариями или скриптами (scripts), предназначены главным образом для обеспечения интерактивности Web-страниц — обработки событий, вызванных манипуляциями мышью и клавиатурой, и данных, введенных пользователем. Ведь пользователь может ввести, "тарабаня" по клавиатуре, какие угодно глупости. Кроме того, с помощью сценариев можно изменить часть или все содержимое страницы, идентифицировать браузер пользователя с целью адаптации к нему загружаемого документа, прочитать и записать cookie, чтобы сохранить на пользовательском компьютере какие-то сведения, а также многое другое. Такие задачи, как создание гостевой книги, счетчика количества посещений Web-страницы, форума и электронного магазина, вообще невозможно решить без сценариев. Более того, эти функции выполняют не клиентские, а серверные сценарии.

Так что если вы замахнулись на решение клиент-серверных задач, то вам просто необходимо заняться не только языками, но и технологиями Web-программирования. Если HTML — язык разметки гипертекстовых документов — довольно прост, то язык сценариев несколько сложнее и требует для освоения особого подхода. Нередко HTML вообще не относят к языкам программирования, но это лишь особое мнение программистов, которые привыкли к большему могуществу, чем простое форматирование текста или верстка отображаемого в браузере документа, содержащего помимо обычного текста еще и другие мультимедиавставки.

З а н у д а. А почему HTML — это не язык программирования? Ведь и в нем есть команды, пусть даже и называемые тегами или инструкциями, а также параметры команд — атрибуты. Такие категории присущи всем языкам программирования. Разумеется, не все, что хочешь, можно сделать с помощью HTML, но и другие языки также имеют какие-то ограничения.

П р о ф е с с о р. Наверное из-за того, что программы, написанные на HTML, имеют очень простую (линейную) структуру. Интерпретатор HTML, т. е. Web-браузер, выполняет загружаемый HTML-код последовательно, команду за командой, как они написаны в исходном коде. Иногда, возможно, браузер делает это несколько иначе, но автор HTML-программы все равно может считать, что ее выполнение происходит именно так, и в подавляющем большинстве случаев будет действительно прав. В языке HTML нет ни переменных для хранения в оперативной памяти компьютера промежуточных и окончательных значений, ни команд управления вычислительным процессом, ни многого другого, что свойственно большинству настоящих языков программирования, хотя в нем много команд и параметров. Я бы сказал, что язык HTML лексически (словарно) богат, а средствами выражения (синтаксическими структурами) беден. Но нет худа без добра. Именно незатейливость HTML-программ в структурном отношении и обеспечила всеобщую доступность программирования на языке HTML. Представьте себе очень примитивного человека. Его словарный запас не более 200 слов. Однако мы, обладая запасом более 2000 слов, и он, весьма стесненный в средствах выражения, все же сохраняем способность общения друг с другом. Это происходит из-за того, что мы можем построить свои тексты в примитивно-линейном порядке, сократив объем удерживаемого контекста до минимума (избегая ссылок на ранее сказанное), а также не используя логические конструкции вида "если … то" и т. п. Другими словами, мы сможем кое-как общаться на основе весьма примитивного языка. Но если нам потребуется более изощренная коммуникация, лучше приспособленная для выражения нетривиальных мыслей, то понадобится и более сложный язык, более гибкий не столько лексически, сколько грамматически. Поэт использует обыкновенную

лексическую базу и почти стандартную грамматику языка, а поэтического эффекта достигает игрой лексических подстановок (метафорами) в стандартные синтаксические конструкции. Если этого мало, то он модифицирует и грамматические шаблоны, достигая мелодической и ритмической выразительности своей рифмованной речи. Я не поэт и не литературовед. Сказанное мною предназначено лишь для того, чтобы посредством аналогии акцентировать ваше внимание на некоторых важных языковых аспектах программирования.

Не следует забывать, что HTML (Hyper Text Markup Language — язык разметки гипертекстовых документов) был задуман как язык форматирования текста со ссылками на другие документы. Вслед за этим стало очевидным, что гипертекстовый документ должен иметь возможность включать в себя не только тексты, но и внешние ресурсы другого сорта, такие как графические изображения, видео, звук и т. д. Идея гипертекстов была предложена Ванневаром Бушем в 1945 году, т. е. на заре компьютерной техники, но для широкого внедрения этой концепции в жизнь потребовалось более 40 лет. Сначала всего этого было более чем достаточно, а простота HTML делала его доступным широким массам, желающим что-либо опубликовать во Всемирной сети. Позднее стало очевидным, что кроме разметки публикуемого текстового документа и вставки в него чего-то инородного, требуется делать еще кое-что, по крайней мере, организовывать каким-то особым образом взаимодействие с сервером. Прежде чем отправить на сервер какие-то данные, введенные пользователем в поля формы, настоятельно требовалось их проверить. Действительно, если пользователь ничего не ввел и нажал на кнопку **Отправить**, то зачем передавать "пустые" данные? Эти первоочередные задачи и потребовали подключения к HTML-коду программ обработки данных, способных не только их визуально представлять, но и анализировать, а при необходимости и преобразовывать.

1.2. С чего и как начать?

Простак. Насколько я правильно Вас понял, уважаемый Профессор, настоящие программы пишутся на более сложных, чем HTML, языках. Не соприкасаясь вплотную с программированием, я это и так понимал, а потому всячески сторонился такой деятельности. Я считал, что программирование, будучи сродни математике и другим точным наукам, чуждо мне. Оно кажется мне сухим и весьма занудным делом, в котором требуется точное соблюдение предопределенных правил, заучивание специальных слов и особенностей их употребления, а также понимание многих нюансов со-

ставления программ Я же люблю быстро получать ощутимые результаты, а не копаться во всяких там "крючках".

З а н у д а. А я не знаю, получится из меня специалист или нет, но, тем не менее, хотел бы поближе познакомиться с тем, чем программисты занимаются, в чем суть их проблем и как они пытаются их разрешить. Если я не ошибаюсь, то к освоению программирования следует приступить, начав с изучения какого-нибудь конкретного языка. При этом правила языка лучше осваивать на примерах и выполняя контрольные задания. Верно?

П р о ф е с с о р. Чтобы заниматься практическим программированием нужно, разумеется, изучить язык, на котором вы будете писать свои программы. Кроме этого, необходимо знать еще и среду программирования — операционную систему, под управлением которой будет выполняться ваша программа, имеющиеся и подключаемые при необходимости уже готовые библиотеки функций, а также возможности конкретной исполнительной системы более низкого уровня, такой, например, как интернет-браузер. Исполнительная система предоставляет вам доступ ко многим объектам: своим собственным элементам, загруженным документам, к конструкциям собственно языка и, возможно, операционной системы. Все это довольно сложное для понимания "хозяйство", однако его постепенно осваивают даже люди изначально малоподготовленные, но искренне желающие стать программистами.

Бытует мнение, что для понимания программирования непременно требуется освоение конкретного языка. Однако я хочу сказать, что суть умений программиста, как и обычного писателя, составляет отнюдь не доскональное знание языка. Язык лишь необходимое формообразующее средство, сам по себе он не обеспечивает автору достаточные условия создания хорошего произведения — конечного продукта его творчества.

П р о с т а к. Вот уж никак не ожидал услышать, что для программирования изучение языка не является необходимым. А как же я буду писать программы? Ведь я могу такое написать, что не только компьютер, но и Вы, Профессор, не поймете.

П р о ф е с с о р. Попробуем разобраться в этом кажущемся парадоксе. Обратите внимание, что дети примерно к пяти годам почти в совершенстве пользуются родным языком в устной речи, не имея при этом никакого понятия о его теории: орфографии, морфологии, синтаксисе и т. д. Дошкольники учатся говорить в процессе коммуникации, на практических примерах, а не по учебникам. Представьте себе, что бы получилось, если обучение языку годовалого ребенка начиналось с разъяснения того, что является именем существительным, а что — глаголом. Дети учатся примитивному словоупотреблению, начиная с называния окружающих предметов и постепенно

переходя к несложным предложениям, отображающим отношения между видимыми и уже названными предметами. Эти навыки формируются у детей удивительно быстро и как будто без нашего, взрослого сознательного вмешательства. Во всяком случае, наше взрослое вмешательство кажется нам лишь корректирующим, а не креативным, т. е. принудительно формирующим. Эффективность данного процесса особенно высока, если родители, бабушки и дедушки не используют в своей речи "птичий язык", т. е. сленг, обильно приправленный междометиями и словами-паразитами. Настоящие проблемы у детей возникают при изучении уже практически освоенного устного языка в школе, где требуется не только знать правила правописания, но и следовать им в письменной речи. И сложность не только в переходе от устной речи к письменной. Трудность в правильном использовании доставшегося нам от предков родного языка вообще, независимо от целей его применения и среды. Когда я слышу фразу, начинающуюся со слова "короче", то подозреваю, что краткости изложения мысли мне не стоит ждать, несмотря на декларированную претензию говорящего быть кратким. Желающие высказаться сжато (нередко обозначающие это стремление вводным словом "короче"), в действительности выражают лишь свое желание выразить суть, но не могут (по разным причинам), а потому за подобным "ключевым" словом следует весьма длительное и невразумительное словоблудие. А в чем причина? Трудно дать исчерпывающий ответ, но на некоторые вопиющие, грубые ошибки я легко могу указать. Прежде всего, речь должна следовать за мыслью, а не наоборот. Если у вас это не всегда получается, то просто не спешите говорить (не тараторьте), заполняя требуемые процессом мышления паузы чем попало. Не забывайте при этом, что высказанное утверждение имеет значительно более далеко идущие последствия, чем мы обычно это имеем в виду, если действительно об этом задумываемся. Недаром говорят, что слово — не воробей, вылетит — не поймаешь. Мысли материализуются, даже если вы их не записали ни в обычном письме, ни в программном коде. Помните, что "сначала было слово" и только потом все остальное — реализации и видимые представления. Программист, пишущий свою речь на искусственном языке для компьютера, должен быть точен в выражении своих мыслей. Возможные ошибки не только искажают мысль, но и прерывают само общение.

Создатель художественных произведений стремится представить свою мысль в виде образов, атакующих эмоциональную сферу, т. е. неточных и расплывчатых изначально (по своему определению). Его задача заключается не в том, чтобы научить кого-то и чему-то конкретно, а в том, чтобы возбудить в реципиенте некое чувство, которое поможет ему что-то осознать или даже сотворить. Пусть, как думает автор, читатель или зритель произведения сам сотворит свой мир на заданную тему. Программист, однако, не может огра-

ничиться только этими целями и технологиями, он должен донести свою идею и сопутствующие ей образы сначала до "холодного" компьютера, а через него и "одухотворенному" конечному пользователю.

Общаясь друг с другом, мы часто даем указания или команды, высказываем какие-то требования и пожелания. Особенно мы расположены к поучениям в виде советов, хотя их выдача — большой грех, даже при искреннем желании помочь ближнему. Это обусловлено тем, что рекомендации "благожелателей" ограничивают, так или иначе, ваше личное движение по собственному пути, т. е. вашу творческую активность, которую вы просто обязаны проявить в отпущенный вам срок. А советчики хотя и готовы помочь, но помните, что "благими намерениями вымощена дорога в ад". Поэтому советы следует давать лишь тем, кто их просит (не следует отказывать в милосердии тем, кто стучит в вашу дверь). Вместе с тем, преподавательская деятельность в различных формах (занятия в учебных заведениях, учебники и всяческие пособия) вполне оправдана в человеческом общежитейском плане тем, что разумно передать накопленный жизненный опыт, по крайней мере, следующему поколению. Однако преподаватель не должен ничего навязывать своим ученикам в качестве истины в последней инстанции. Доступные нам здесь и сейчас знания эфемерны и неоднозначны, а подлинная, вечная и незыблемая истина все время впереди нас, светит нам подобно маяку, указывающему лишь направление движения, вселяя в нас надежду когда-нибудь пристать к берегу. Но так ли нам важен вожделенный берег? А что будет потом, после достижения цели? Это сакраментальный вопрос, на который мало кто отваживался ответить серьезно, а полученные ответы в итоге оказались неудовлетворительными. Я для себя понял лишь, что продвижение к чему-то, что называют истиной, важнее ее самой, быть может недоступной изначально в нашем мире. Этот процесс и составляет, по моему мнению, содержание и смысл жизни.

Итак, мы должны быть готовы к преодолению всех возможных препятствий на пути к истинному знанию, пусть даже без надежды когда-нибудь пристать к берегу. Никто из смертных не обладает и не может постичь истину абсолютно, но отказ от стремления к ней, подобно самоубийству, лишает содержания и смысла саму жизнь. Жизнь — плавание, а не отстой у причала. С другой стороны, ученик не должен оправдывать свои неудачи тем, что кто-то его так-то и тому-то научил, в результате чего он заблудился, напортачил или просто получил двойку на экзамене. Учение — дело не столько преподавателя, сколько самого ученика, который должен сам искать и выбирать, чему, как и у кого учиться. Я в этом убежден, что бы там ни говорила официальная пропаганда и площадная молва об устройстве системы нашего образования.

Простак. Но, уважаемый Профессор, учителя в школах обязаны дать базовые знания своим ученикам по основным областям знаний: родному языку, истории, географии, математике, физике и т. д. Они должны приобщить детей к сложившейся культуре своего народа, познакомить с другими цивилизациями и, наконец, сообщить основные сведения о достижениях современной общечеловеческой науки и технологии. Ребенок, согласитесь, не может самостоятельно понять и сформулировать, из какой области знаний ему нужно получить какую-то конкретную информацию. Кроме того, ни ребенок, ни его родители не могут подчас определить, хороши ли учителя. Все это так сложно.

Профессор. Общий рецепт прост. Если вам не нравится учитель, перейдите к другому. Смешно требовать от учителя научить чему-то, особенно в том случае, если ученик не хочет учиться.

Зануда. Большинство детей, насколько я знаю, не очень-то любят учиться. Но с высшей точки зрения (государства или нации) мы же должны как-то позаботиться об уровне образования следующего поколения. Ведь мы не можем в этом важном деле пустить все на самотек или положиться на природу или Провидение. Бог не любит, если мы опускаем руки.

Простак. Как же не требовать от учителя, если мы отдаем своих детей в школу, чтобы их там научили всему, что надо. Я хотел бы, чтобы мой ребенок закончил школу в том состоянии, которое позволило бы ему продолжить образование в университете, например. Если не быть требовательным к учителям, то они просто спишут все свои огрехи на нерадивость учеников, а их успехи, если такое произойдет, непременно отнесут к своим заслугам.

Профессор. Не будем излишне драматизировать ситуацию. Дети обладают различными от природы способностями и интересами. От преподавателя можно ожидать в лучшем случае не более того, что он сам знает и умеет. Разумеется, школьные учителя отличаются от специалистов-предметников (пусть даже нобелевских лауреатов) тем, что они владеют методикой обучения, т. е. технологией освоения знаний. Все это прекрасно, но если учитель не имел собственного опыта научных исследований, то никакая изощренная методика изложения знаний не поможет их передать. Без исследований нет и откровенных, т. е. настоящих знаний. Ученики рано или поздно распознают, что их учитель просто цитирует не им написанный учебник, подчас интерпретируя изложенные в нем истины весьма убого, скучно и некреативно, а в предмете ориентируется лишь формально. Детей трудно обмануть, поскольку их вводит в жизнь в первую очередь природа, а только после этого — человеческих рук культура. Детская способность отличить фальшь

от правды значительно сильнее, чем у нас, взрослых, наученных жизнью терпимости и беспринципности.

Вы говорите, что учитель должен дать основные знания по фундаментальным наукам и искусствам, которые были сформированы человечеством в течение нескольких веков, если не тысячелетий? Вы полагаете, что это возможно? Преподаватель может лишь поведать своим ученикам, как лично он, а также многочисленные предшествующие поколения пробирались сквозь тернии к истине, как она (истина) трансформировалась с годами и под влиянием ее творцов — героев-исследователей, — как удавалось в этом процессе отделить субъективное (личное и эфемерное) от объективного (не зависящего от испытателя и перманентно воспроизводящегося), и наконец, каково текущее состояние изучаемой предметной области.

Аналогичные проблемы возникают и при попытках изучения искусственных языков, созданных человеком для определенных целей и ориентированных на работу с компьютерами, т. е. при изучении языков программирования.

П р о с т а к. Уважаемый Профессор, Вы намекаете на то, что, следуя великой и мудрой природе, не нужно загружать учеников громадой знаний о языках, а лучше рассмотреть хорошие практические примеры. При этом наиболее трудные для понимания места Вы, как мастер, можете прокомментировать, предостеречь от грубых ошибок и прочих глупостей, обычно свойственным новичкам. Я знаю, что большинство как начинающих, так и опытных разработчиков программ больше всего ценят не общие разглагольствования, а профессиональные советы, способствующие решению вполне конкретных задач. Мне кажется, что нам с Вашей, разумеется, помощью следует поставить ряд актуальных задач, а Вы затем дадите развернутые рекомендации для их решения.

П р о ф е с с о р. Предлагаемый Вами подход — лишь один из возможных вариантов. Изучение типичных примеров, а также конкретных приемов и рецептов — хороший способ подготовки исполнителей низшего звена. Таков нормальный путь обучения ремеслам, но не наукам. Освоение моделей различных областей мира, технологий их изучения и проектирования, способов применения теории к жизни — задача более высоких ступеней образования. Ремесло живо лишь в течение довольно короткого периода, пока конъюнктурно востребовано и использует конкретные и доступные в данный момент инструменты и технологические приемы. Более высокий уровень образования востребован значительно дольше, именно он постигается в высшей школе, хотя нередко в число ее дисциплин (я думаю, из меркантильных соображений) вводят прикладные курсы, не имеющие под собой никакого научного фундамента. Изучение, например, способов замены электрических

предохранителей практически очень важно, но не является задачей курса электротехники в высшем учебном заведении. Главное в том, что получившие более высокое образование имеют бесценную способность наращивать свои первоначальные знания. Иначе говоря, они знают, как учиться, изучать уже известное и исследовать еще непознанное.

Правильный подход к обучению во многом определяется степенью предварительной подготовленности учеников, а также целями, которые они перед собой ставят. Для выбора эффективного способа изучения программирования мне, как вашему преподавателю, очень важно знать, имел ли ученик предварительное знакомство с предметом изучения или нет. Это необходимо для построения оптимального плана обучения не только программированию, но и вообще всем наукам и ремеслам. Можно выделить три уровня знакомства с программированием: нулевой, предварительный и профессиональный.

Само собой разумеется, нулевой уровень соответствует полному отсутствию каких бы то ни было знаний, которые применимы на практике или, на худой конец, в "светских" разговорах о компьютерах. Человек с данным уровнем подготовленности либо шарахается от программирования, как черт от ладана, либо, ведомый природными любопытством и целеустремленностью, затратив некоторые усилия, переходит на следующую ступень.

Предварительный уровень предполагает некоторое, обычно весьма поверхностное, знакомство, по крайней мере, с основными терминами и понятиями, такими как переменная, оператор, функция, объект, тип данных и т. п., которое все же остается неточным и неполным настолько, что не позволяет уверенно решать даже относительно простые задачи, не говоря уж об эффективности принятых решений. Представители данного уровня стремятся найти и вставить в свои конструкции уже готовые коды программ. При этом они пытаются их как-то изменить, приспособив к своим конкретным задачам. Иногда это удается, но чаще вызывает неудовлетворенность из-за множества сообщений об ошибках. В поисках помощи такие люди мечутся по просторам Интернета "без руля и ветрил", обретая дурные привычки или окончательно отвращаясь от программистских амбиций. Подобным ученикам для дальнейшего продвижения в области программирования нужен хороший учебник или руководитель.

Профессиональный уровень может иметь несколько градаций. Однако для него характерно владение, по меньшей мере, одним языком и одной технологией программирования, а также опыт участия в реализации хотя бы одного конкретного проекта. Профессионал довольно легко переходит от одного языка к другому и приспосабливается в случае необходимости к соответствующей этому языку технологии. Владея одним языком программирования,

он уже приблизительно знает, с чем ему придется встретиться при изучении другого. Если он уже овладел одной технологией программирования, то при изучении другой он понимает, чем она хороша или плоха при решении задач интересующего его класса. Профессионала обычно отличает то, что он знает, с какой стороны подойти к проекту даже незнакомой ему предметной области, как его провести через все перипетии реализации и чем закончить. И это знание является, пожалуй, важнейшим среди всей информации, которой он владеет. Тем не менее, профессионал все время читает новые публикации не только на узкую специальную тему, но и о сопредельных областях. Он всегда "держит руку на пульсе" и обеспокоен тем, чтобы не упустить из виду современные тенденции развития его науки и ремесла. Профессионалам обычно хватает справочников или кратких руководств по новому языку, чтобы с минимальными усилиями перейти к практической деятельности. Программисты обычно держат в тайне освоенные приемы работы и полученные знания. Чаще всего они не могут поведать о своем мастерстве не только из-за предосторожности разбазарить свое "ноу-хау", но и просто органически: они не знают с чего начать и чем закончить в своем изложении. Возможно поэтому всем нам, обычным смертным, программисты представляются в облике "гуру", слова и звуки которых следует ловить, боясь что-либо пропустить, а затем внимательно, не считаясь с трудозатратами, пытаться как-то интерпретировать с пользой для решения своих меркантильных задачек. Чаще всего гуру — это просто очень опытный программист, сталкивавшийся в своей жизни и с разными задачами, и с различными приемами их решения. Реже встречаются специалисты, которые не только могут решать кем-то поставленные задачи, но и ставить (формулировать) их с учетом известных технологических средств.

На заре компьютерной эры программисты в большинстве своем были математиками. До середины 1980-х годов их так и называли. И недаром, поскольку они умели представить научные, инженерные и другие задачи в форме алгоритмов их решения, а затем и в виде программ для компьютера. Тех программистов, кто не обладал алгоритмическим способом мышления и не мог трансформировать постановку задачи в программу для ее решения, называли кодировщиками. Кодировщики досконально знали язык программирования или системы машинных команд, но не владели способами решения пользовательских задач. А алгоритмы решения прикладных задач в большинстве случаев следовало искать не в области программирования, а в математике. Нередко возникали вполне осмысленные и практически востребованные ситуации, которые, как казалось на первый взгляд, можно было легко решить с помощью компьютера. Малообразованный программист с радостью хватался за выгодный заказ и с энтузиазмом начинал писать соответствую-

щую программу. Затем возникали проблемы, связанные не только с ограниченными ресурсами компьютера. Программист долго мучился в поисках выхода из создавшейся затруднительной ситуации и нередко находил его путем изменения постановки самой задачи. Данный процесс мог иметь не одну безуспешную итерацию. А дело порой оказывалось в том, что задача в первоначальной своей постановке оказывалась алгоритмически неразрешимой. Иначе говоря, было принципиально невозможно построить алгоритм (а значит и программу), который приводил бы к решению. В таких случаях говорят, что задача алгоритмически неразрешима. А как распознать, разрешима ли алгоритмически данная задача? Для этого необходимо выполнить некоторое исследование, которое чаще всего лежит в области математики, а не программирования.

З а н у д а. Если я Вас правильно понял, существуют задачи, которые нельзя решить в принципе, даже с использованием суперкомпьютеров. Было бы неплохо, если бы Вы привели какой-нибудь пример для иллюстрации этого парадоксального обстоятельства.

П р о ф е с с о р. В качестве примера рассмотрим практическую задачу обмена валют, так или иначе решаемую с помощью программного обеспечения компьютера в обменном пункте. Валюты, как хорошо известно, имеют различные наборы денежных купюр. Например, существуют купюры (бумажные деньги) достоинством 5, 10, 50, 100, 500, 1000 и 5000 рублей, но нет купюр достоинством 3 и 4 рубля. Другие валюты могут иметь отличающиеся наборы купюр. Кроме того, в пункте обмена в данный момент может не оказаться некоторых из принципиально существующих купюр, а количество имеющихся очень ограничено. Клиент желает обменять по курсу некоторую сумму, например, в рублях на деньги в другой валюте. Очевидно, что возможна ситуация, когда предъявленная сумма не может быть полностью переведена в другую валюту из-за ограниченного набора купюр в обменном пункте. Чтобы все же выполнить операцию обмена, клиент либо должен согласиться получить сдачу в рублях, либо добавить некоторую сумму в рублях к предъявленной для обмена. Естественно желать, чтобы эта дополнительная сумма сдачи или добавки была как можно меньше. Можно потребовать, чтобы эта сумма была минимальной. Именно в такой постановке задача может заинтересовать математиков.

З а н у д а. Очевидно, что сначала нужно с помощью коэффициента, выражающего переводной курс валют, вычислить сумму, которую обменный пункт должен, в первом приближении, отдать клиенту.

П р о с т а к. За вычетом комиссионных, разумеется.

З а н у д а. Да, но сейчас это не столь важно, потому не будем учитывать комиссионные. Далее следует подобрать купюры так, чтобы сложившаяся сумма оказалась как можно ближе к вычисленной по курсу.

П р о с т а к. Лучше сказать, чтобы разница между вычисленным значением и суммой номиналов купюр была минимальной. Поскольку, вообще говоря, одну и ту же сумму можно составить с помощью различных комбинаций купюр, то придется рассматривать множество вариантов, чтобы выбрать наилучший в смысле минимальности сдачи (или добавки).

П р о ф е с с о р. Должен заметить, что ни один обменный пункт не решает задачу в такой постановке. Я имею в виду требование минимальности. Дело в том, что, как показали специальные математические исследования, длина пути поиска такого решения (т. е. число перебираемых вариантов) слишком велика.

П р о с т а к. В обменные пункты следует просто поставить более производительные компьютеры. Наконец, то, что нам недоступно сейчас, станет обычным завтра.

П р о ф е с с о р. К сожалению, загвоздка имеет принципиальный, а не технологический характер. Дело в том, что рассматриваемая задача с требованием минимальности расхождения вычисленной и составленной из купюр сумм имеет, как говорят специалисты, экспоненциальную сложность. Это значит, что сложность (например, число шагов или время вычисления) зависит экспоненциально от объема исходных данных. Так, если n — объем исходных данных, то сложность экспоненциально сложных задач определяется как e^n (где $e \approx 2{,}72$ — основание натуральных логарифмов), т. е. очень быстро возрастает с увеличением n. Например, при $n = 50$ значение e^n превосходит количество атомов в видимой части вселенной. Только перечисление их с помощью самого быстродействующего компьютера потребует практически неприемлемо большого времени. Относительно рассматриваемой задачи доказано, что она разрешима и, более того, найден алгоритм ее решения. Вместе с тем, доказано, что эта задача имеет экспоненциальную сложность и, следовательно, соответствующий алгоритм в интересных с точки зрения практики случаях нереализуем. При очень малых объемах данных его вполне можно реализовать и достаточно быстро получить результаты, но для пользователя это может быть не интересно. Таким образом, существуют теоретически разрешимые задачи с практически нереализуемыми алгоритмами.

П р о с т а к. Сказанное Вами, Профессор, звучит парадоксально. Но ведь формулировка задачи для пункта обмена валют выглядит вполне разумно с точки зрения потенциального пользователя, не говоря уж о ее

практической важности. Быть может, ее следует немного подкорректировать, сохраняя смысл?

Профессор. Это хорошая мысль. Сохранить "потребительский" смысл решения задачи и в то же время обеспечить его компьютерную реализацию очень важно, но далеко не всегда практически осуществимо.

Зануда. Мне кажется, что в формулировке задачи нужно убрать требование минимальности, как слишком жесткое.

Простак. Но что же тогда останется? Ты хочешь, чтобы пункт обмена решал свою задачу как попало? Например, клиент хочет обменять 1000 рублей на доллары, а ему предлагают 1 доллар, дополняя остальное рублями в виде сдачи.

Профессор. Убрать требование минимальности — хорошая идея. Мы можем построить алгоритм решения задачи обмена валют, который не обеспечивает во всех случаях минимальную разницу между вычисленной суммой и составленной из купюр, а лишь стремится ее уменьшить. Возможно, иногда ему удастся достичь минимального результата, но гарантировать это во всех случаях нельзя. Мы надеемся, что такое поведение алгоритма обеспечит если не наилучший, то хороший с практической точки зрения результат во многих случаях, хотя и не во всех. Замечу, что зачастую так и поступают.

Зануда. Как это формальный алгоритм может стремиться к чему-либо? Любой алгоритм просто что-то делает, а что-то — нет, он либо решает задачу, либо — нет, без всяких эмоций.

Профессор. Я поясню свою мысль об алгоритме, который стремится к наилучшему результату, на следующем примере. Пусть из пункта A в пункт B можно добраться различными путями по дорогам с перекрестками, на которых установлены стрелочные указатели расстояний до следующего перекрестка. Как, начиная движение из пункта A, дойти до пункта B кратчайшим путем?

Самый простой по своей идее алгоритм решения этой задачи основывается на методе прямого перебора всех возможных путей достижения пункта B из пункта A и выбора из них наименьшего по своей протяженности. Однако всех путей может оказаться так много, что практически заняться их перебором невозможно.

Однако можно поступить и иначе. На каждом перекрестке будем выбирать то направление движения, в котором следующий перекресток ближе. Если случится забрести в тупик, то можно вернуться на предыдущий перекресток и выбрать другое направление. Очевидно, что на каждом перекрестке мы

разумно выбираем направление, используя всю доступную на данном этапе движения информацию. Если бы направление движения на каждом перекрестке выбиралось случайно, то посторонний наблюдатель не мог бы сказать, что мы стремимся сократить маршрут. Поэтому мы говорим, что наш алгоритм стремится к сокращению маршрута, хотя и не гарантирует, что весь пройденный путь будет кратчайшим. Выбор наилучшего локального решения не гарантирует, что глобальное решение, составленное из локальных, тоже будет наилучшим. Действительно, сэкономив в начале пути, мы можем лишить себя таких последующих решений, которые вкупе с предыдущими оказались бы лучше.

Итак, рассмотренный алгоритм лишь стремится сократить весь маршрут, но не гарантирует его минимальной протяженности. Зато он не требует полного перебора всех возможных путей, а следовательно, работает быстрее. Впрочем, я должен заметить, что существуют эффективные алгоритмы поиска кратчайших маршрутов, основанные не на полном переборе. Наиболее известный из них — алгоритм Дейкстры, но его мы не будем сейчас рассматривать.

П р о с т а к. Мне кажется, что простой алгоритм, который лишь стремится сократить маршрут, не гарантируя при этом наилучшего результата, можно усовершенствовать. Так, прежде чем окончательно выбрать направление движения на данном перекрестке, можно сначала проверить все направления из него. Иначе говоря, следует сбегать на все смежные перекрестки, чтобы по указателям на них узнать расстояния до следующих перекрестков. Получив эту информацию и вернувшись на исходный перекресток, можно принять решение о движении с учетом протяженности не одного, а двух переходов. Разумеется, эта стратегия потребует больше времени, но зато конечный результат будет лучше.

З а н у д а. Можно заглянуть не только на непосредственно смежные, но и на последующие перекрестки. Если продолжить эту аналогию, то мы придем, в конце концов, к полному перебору всех маршрутов и выбору самого короткого из них.

П р о ф е с с о р. Вы оба мыслите в верном направлении. Но давайте оставим эту задачу. Подведем итоги обсуждения так называемых алгоритмических задач. Обратите внимание, что их "головоломный" характер обусловлен тем, что решение требует изобретения плана действий, т. е. алгоритма. Однако попытки решить задачу в более общей постановке, или даже только некоторая модификация условия задачи обнаруживают необходимость математического анализа. Возможно, вам будет небезынтересно узнать, что рассмотренные нами примеры так или иначе были использованы в теории оптимального кодирования и теории информации, в так называемой

транспортной задаче оптимального планирования перевозок, впервые поставленной в начале второй мировой войны, и в ряде других областей. Некоторые алгоритмы, придуманные для решения частных задач, оказывались применимыми и для других.

Простак. Я, признаюсь, — новичок с нулевым уровнем познаний, но не прочь их несколько расширить. Однако я опасаюсь, что это мне будет нелегко даваться. Учась в школе, я не любил точные науки из-за их чрезмерной абстрактности.

Зануда. А я делал попытки написать простейшие программки. Например, я хотел, чтобы щелчок на кнопке формы на Web-сайте приводил сначала к проверке корректности данных, введенных пользова-телем в поля формы, а затем — к их отправке по электронной почте. Однако даже эта тривиальная задача вызвала у меня немало вопросов. Я также пытался приспособить для своих нужд программные коды, заимствованные из книг или Интернета, но нередко они не работали, а разобраться в причинах неудач мне было трудно. Я считаю, возможно и слишком самонадеянно, что обладаю предварительным уровнем подготовки, но очень хотел бы освоить программирование более обстоятельно, хотя бы в пределах, обеспечивающих решение первостепенных практических задач в области сайтостроительства.

Профессор. Итак, вы оба занимаете некоторое положение между нулевым и предварительным уровнями подготовленности. Поэтому я считаю, что лучше всего начать с изучения языка программирования, а не с конкретных рецептов решения типовых задач. Но с учетом вашего уровня познаний я бы приступил к изучению языка программирования так, как это делается на уроках родного языка в начальных классах общеобразовательных школ. Другими словами, я хочу начать с пропедевтики.

Простак. А что означает последнее слово?

Профессор. Пропедевтика (греч. propaideio — предваряю) — сокращенное изложение какой-либо науки, т. е. подготовительный, вводный курс в какую-либо область знаний, предшествующий более глубокому и детальному ее изучению. Мы начнем с уяснения базовых понятий, иллюстрируя их использование на специальных (хотя и, возможно, малополезных) примерах. Я знаю, что большинству новичков больше импонирует изучение языка на примерах, максимально приближенных к практическим задачам. Однако такого рода примеры-рецепты обычно недостаточно просты, чтобы научить основам программирования. Элементарные же примеры, хотя и малополезны в качестве конкретных практических рецептов, важны для понимания фундаментальных понятий, которые пригодятся при решении любых задач. Вам придется пройти этот путь, даже если он покажется невыносимо

скучным. Необходимо получить устойчивые навыки обращения с основными терминами и соответствующими им понятиями. В дальнейшем будет и легче, и интереснее.

1.3. Почему программировать интересно?

Профессор. Программирование увлекает многих людей видимо потому, что только оно наиболее явно воплощает в жизнь принцип "сначала было слово …". Программист пишет свой "роман" на некотором языке, понятном компьютеру, и его произведение оживает, действует и работает: отображает на мониторе сменяющие друг друга картинки, печатает отчеты, управляет различного рода машинами и устройствами. Материал, с которым имеет дело программист, чрезвычайно податлив, по сравнению с деревом, сталью и прочими физическими и химическими субстанциями. Программисту нужно только представить свою мысль в виде текста — сообщения для компьютера, который должен его понять и выполнить содержащиеся в нем инструкции. В отличие от программиста скульптору приходится затрачивать огромную энергию для придания некой формы куче глины или куску гранита, а инженер-конструктор должен ограничивать полет своей фантазии современными ему материалами, технологией их переработки и производства окончательного продукта. Программист во многом похож на поэта или писателя-беллетриста. Как и последние, он творит тексты, содержащие его мысли. Однако в отличие от литературных произведений, программные тексты не могут быть метафоричными, семантически неточными и содержать обычные орфографические и синтаксические ошибки, поскольку адресованы не людям, а бездушным компьютерам, лишенным поэтического воображения и общечеловеческих способностей к ассоциациям, разрешению неопределенностей и вскрытию потаенного смысла. Пропуск буквы или запятой, лишний пробел или скобка — все это будет отвергнуто исполнителем (интерпретатором или компилятором) программного кода, хотя человек может легко понять, о чем идет речь.

Программиста постоянно искушает иллюзия, что путь от мысли к окончательному и полезному результату довольно короток, стоит только придать своему творению надлежащую форму, отшлифовав ее, удалив все заусеницы и погрешности. Он уверен в этом потому, что может, стремясь к совершенству, несколько раз в день перекраивать реализации своего замысла. Нередко ему удается существенно улучшить качество своего продукта, изменив код программы всего на 1%. Однако бывает и так, что незначительные, на первый взгляд, изменения приводят к краху всей его многотрудной и многоэтажной конструкции. Ему кажется, что он творит постоянно, паря в заоб-

лачных высях чистых идей, постепенно спускаясь к насущной цели, а не просто перерабатывает горы руды или тупо просеивает песок в надежде обнаружить золотой слиток. Кроме того, опытный программист обычно знает, что если проблема не решается легко и быстро, то, скорее всего, в этом виновата плохая основополагающая идея, а не просто его нерадивость, невнимательность и лень. Он также чувствует, что хорошая программа не может быть громоздкой и запутанной, а чрезмерная ее сложность обычно связана с недостаточной проработанностью исходного замысла. Здесь свобода абстрактного творчества и результат его материализации чрезвычайно близки. Кто еще в науке, искусстве или ремеслах может делать или хотя бы ощущать то же самое?

Объективным основанием возможности программистского творчества является то, что одна и та же задача может быть решена программно различными способами. Выбор наилучшего или хотя бы просто хорошего способа — настоящее дело программиста, зависящее напрямую от его образованности, опыта и таланта. Данная ситуация напоминает положение вещей в шахматах. В самом деле, небольшой набор эффективных вариантов начала шахматной партии известен давным-давно, но талант шахматистов проявляется главным образом в ее середине и заключительной части (эндшпиле). Правила игры здесь четко определены, число фигур ограничено, а на протяжении веков накоплен большой опыт розыгрыша конкретных партий. Однако шахматы попрежнему интересуют многих людей, в том числе и ученых, занимающихся проблемами искусственного интеллекта, а также программистов. Астрономически большое количество допустимых вариантов развития шахматных партий, из которых даже выдающиеся шахматисты каким-то образом реально рассматривают лишь некоторые (наиболее перспективные), отсекая огромные множества других! Вот удивительный феномен, который можно наблюдать также и в мире программирования.

З а н у д а. Если я Вас правильно понял, программирование увлекательно, поскольку очень соответствует природной тяге человека к творчеству. Но ведь не бывает чистого, приносящего наивысшее удовлетворение, творчества без большой и кропотливой, порой изнуряющей подготовительной и заключительной (доводочной) работы?

1.4. Неприятности в программировании

П р о ф е с с о р. Вы правы, Зануда. За удовольствия и радости приходится платить. Радость творчества программиста омрачается, в первую очередь, необходимостью отладки практически любой программы. А это нередко требует в несколько раз больше времени и умственных усилий, чем исходный замысел и последующая разработка, и представляет собой в основном кропотливый процесс поиска и устранения ошибок, которых при на-

писании кода программы очень трудно избежать. Если листинг вашей только что написанной программы занимает более четверти страницы, то почти наверняка в нем есть хотя бы одна ошибка. Даже если вам удалось устранить как будто все ошибки, вероятность существования еще не вскрытых все же остается большей нуля. Как говорят программисты, "последняя обнаруженная ошибка в действительности является предпоследней". Наиболее простым по своей сути является элементарное нарушение синтаксиса языка: пропуск символа, запятой, скобки и т. п. В обычных текстах на естественном языке подобного рода ошибки называют опечатками, причиной которых является небрежность писателя, редактора или корректора. Большинство опечаток устраняют при подготовке рукописи к публикации. Программист, в отличие от обычного писателя, не может рассчитывать на помощь редактора и корректора. Он должен самостоятельно отыскать и исправить подобного рода ошибки. В противном случае его программа просто не будет работать. Поэтому программы подвергают испытаниям или, как еще говорят, тестированию.

П р о с т а к. А чем отличается отладка от тестирования программы? Как я понимаю, и то и другое проводят для проверки наличия ошибок.

П р о ф е с с о р. Отличие состоит в том, что отладку выполняет программист (автор программы), а тестирование — обычно другие люди. Дело в том, что к тестированию приступают после того, как программист устранил, как он считает, основные ошибки (главным образом, синтаксические) и программа стала работоспособной, т. е. способной выполнять все заданные функции хотя бы при определенных условиях. Чтобы это стало возможным, необходимо устранить, по крайней мере, все синтаксические ошибки. Далее следует проверить, что программа действительно способна выполнить все возложенные на нее функции, пусть даже не при всех возможных, а только при благоприятных обстоятельствах. Однако программист, проверяя свою программу, вольно или невольно действует в соответствии с правильной логикой ее использования, не допуская грубых смысловых и логических ошибок. Иначе говоря, при проверке своей программы он избегает заведомо "глупых" действий пользователя. Например, если поле предназначено для ввода чисел, то при тестировании программист введет в него какое-нибудь число, а не буквы. Но пользователь, в отличие от автора программы, может действовать как угодно и набрать вместо числа произвольную последовательность символов, получив в результате либо что-то бессмысленное, либо (в лучшем случае) сообщение об ошибке, либо "зависание" (аварийную остановку) программы. Самый неблагоприятный вариант с точки зрения программиста состоит в зависании или, иначе говоря, в "падении" его программы, а худший исход с точки зрения пользователя — получение семантически неверных данных без каких-либо сообщений об аномалиях. Программист может заранее предусмотреть некоторые подобные ситуации, но не все из них

в состоянии проверить при отладке. Поэтому и проводят тестирование, выступая на стороне "дикого" пользователя, а не "умного" автора программы. Лучше, если это сделает кто-то другой (беспристрастный), а не разработчик. Однако при создании Web-приложений часто случается, что и дизайнер, и программист, и тестировщик — один и тот же человек. Разумеется, тестирование не ограничивается только выявлением оставшихся ошибок. Нередко следует еще проверить, как ваша программа будет работать на других платформах, при разных мониторах, в различных браузерах и т. п.

Простак. Я слышал, что есть такие уникальные люди, которые умудряются все испортить, даже не желая этого. Наверное, из них получились бы идеальные тестировщики.

Профессор. Действительно, такие феномены встречаются. При этом их нельзя заподозрить в какой бы то ни было злонамеренности. Так известно, что лишь одно появление в лаборатории знаменитого физика-теоретика Вольфганга Паули приводило к срыву до этого успешных экспериментов и выходу из строя высокоточного оборудования. Этот феномен назвали эффектом Паули. Я думаю, что для успешного тестирования своих программ автору достаточно просто стать злонамеренным, вести себя преднамеренно против своего детища, стремясь спровоцировать появление ошибок, или, по крайней мере, действовать как можно более нелогично и глупо.

Многие теоретики и практики программирования положили немало труда, чтобы увеличить производительность программистов и качество их продукции. С этой целью разрабатывались различные языки высокого уровня. С одной стороны, их стремились как-то приблизить к естественным (точнее, к английскому), а с другой, — обеспечить поддержку технологий программирования, которые способствовали бы уменьшению количества дефектов кода, облегчали бы сопровождение программных продуктов, а также их разработку большими коллективами людей.

Как оказалось, идея сделать языки программирования более доступными широкому кругу людей, приблизив их к естественным, не оправдала себя, поскольку такое приближение касалось главным образом лексики, а не синтаксиса и грамматики. Например, никто из говорящих на русском не испытал бы облегчения при замене в языке программирования ключевого слова *if* на *если*, а *begin* — на *начало*. Ключевые слова, как и слова любого иностранного естественного языка, довольно легко понимаются и запоминаются при достаточно частом их употреблении в речи. Несколько сложнее дело обстоит со словоупотреблением, когда из известных слов необходимо составлять правильные выражения. Если выражение неправильно грамматически и/или синтаксически, то оно, скорее всего, не будет понято правильно, т. е. так, как этого хотел бы говорящий.

З а н у д а. Однако в повседневной речи мы часто слышим грамматически неправильную речь, но все же достаточно адекватно воспринимаем ее.

П р о ф е с с о р. Так происходит, пока примитивные выражения выражают столь же простые мысли типа "дай", "хорошо", "я тебя люблю", "я в шоке", "пора смываться" и т. п. Простые лингвистические конструкции предназначены в основном для передачи оценок ситуации, отношений к окружающим предметам и соответствующих реакций. Примерно для этих же целей служит сигнальная система высших животных. Задумайтесь над тем, как следует построить языковое выражение, чтобы что-нибудь объяснить, доказать или просто рассказать, как, например, из пункта А попасть в пункт Б. Здесь вам примитивными сигналами не обойтись.

Итак, ошибки даже у опытных программистов встречаются настолько часто, что их появление можно считать почти неизбежным. Поэтому программист должен писать свои программы в таком виде, чтобы было легко их отлаживать, а в дальнейшем — сопровождать, т. е. устранять вновь обнаруженные дефекты, вносить изменения, включать в состав более сложных проектов и т. п. Дело в том, что нередко автор с трудом разбирается в тексте своей же программы, если не занимался ею месяц или более. Поэтому при написании текстов программ не следует скупиться на достаточно развернутые комментарии к ним. К сожалению, новички обычно пренебрегают этой рекомендацией и вспоминают о ней, когда приходится обращаться к своей старой программе, даже если она написана неделю тому назад. Кроме того, необходимо стремиться к тому, чтобы программа имела достаточно понятную структуру, состоящую из элементов программного кода (функций, объектов, блоков выражений и т. п.), относительно каждого из которых достаточно ясно, что он делает и с какими другими программными элементами связан. Это важно как для локализации ошибок, так и при модификации программы. Любая корректировка уже отлаженной программы, направленная на ее усовершенствование, как правило, чревата внесением новых ошибок. Так что обычно подтверждается принцип "лучшее — враг хорошего". Если исходная программа хорошо структурирована, то при внесении изменений вам легче будет проследить, куда и как распространилось влияние модифицированного элемента, содержащего ошибки.

Следует также отметить, что коллективная работа над созданием сложного программного продукта тоже может привести к определенным неприятностям.

П р о с т а к. А мне нравится работать в компании. Где как не в хорошей команде единомышленников можно ощутить "чувство локтя", воспользоваться взаимовыручкой и взаимным оплодотворением идеями?

З а н у д а. Хотя я предпочитаю работать в одиночку, но в принципе согласен с Простаком. К тому же не следует упускать из виду, что су-

ществуют настолько сложные задачи, которые либо просто не под силу одному человеку, либо требуют для своего решения неприемлемо много времени. К примеру, я умею писать HTML-код для сайтов, но слабо владею компьютерной обработкой графических изображений, да и в дизайне я далеко не профессионал. Поэтому при разработке сайта было бы разумно скооперироваться с художником-дизайнером. Вместе мы выполним проект и быстрее, и качественнее.

Профессор. Это верно, но не сбрасывайте со счета то очень важное обстоятельство, что при коллективной работе растут накладные расходы на всяческие согласования. Чтобы проект мог быть выполнен группой разработчиков качественно и в срок, необходимы как предварительная тщательная проработка самого проекта, так и его грамотный руководитель — главный конструктор или архитектор. Привлечение к разработке проекта нескольких программистов — довольно непростое дело. Представьте себе, что получилось, если бы мы решили создать "суперроман", поручив это дело десяти самым выдающимся писателям?

Простак. Скорее всего, они передрались бы друг с другом. По крайне мере, увязли в бесконечной взаимной критике. Однако дело не совсем безнадежное, просто среди них нужно назначить руководителя.

Зануда. А ты уверен, что кто-либо из выдающихся авторов согласится писать по чьей-то указке? Писатель не может написать некоторую часть общего романа, если не знает его общего замысла, драматургии и стиля исполнения. Наконец, он просто не сможет писать так, как это делает кто-то другой.

Простак. Но ведь существуют превосходные художественные произведения, написанные в соавторстве, например, книги Ильфа и Петрова, братьев Стругацких и братьев Вайнеров.

Зануда. Это, скорее, исключение из общего правила. Даже если авторов художественного произведения всего лишь двое, то и в этом случае остается только удивляться, как им удалось сотрудничать. Другое дело — создание увесистого справочника, словаря, энциклопедии или учебника. Здесь очевидно, что всю работу можно разделить между многими коллегами, каждый из которых напишет свою статью более или менее автономно. В конце концов, главный редактор "причешет" под единый стиль "многоцветное и разношерстное" произведение коллектива авторов, придав ему композиционную и стилевую целостность.

Профессор. Программисты в большинстве своем творческие люди, во многом похожие на писателей, но результат их работы все же существенно отличается от литературного художественного произведения,

и, главным образом, тем, что носит в большей мере технологический, чем психологический или эстетический характер. Поэтому для программ применимы методы разработки и производства, близкие к методам создания технических проектов с привлечением труда не только единиц, но и десятков специалистов различной квалификации. При этом руководителям проектов очень важно учитывать специфику их труда. Программист может просидеть над решением своей задачи несколько часов или даже дней, не написав ни строчки кода. А в один прекрасный день он придет и за полчаса создаст превосходную рабочую версию заданной ему программы. Важно понимать, что программист — это не просто производитель кодов, подобный, например, токарю, из под рук которого выскакивают болты. Чем дольше работает токарь и чем выше его квалификация, тем больше болтов он выпустит. Программист же, напротив, больше всего времени тратит на обдумывание проекта своей программы и отладку ее реализации, и меньше всего — на написание самого кода, т. е. текста из выражений на языке программирования. Со стороны может показаться, что бóльшую часть времени он ничего не делает. Так уж повелось, что за обдумывание никто не платит, и это время расценивается как праздное. Иногда разработчик затратит целую неделю, чтобы заменить многостраничный листинг первоначальной версии своей программы всего лишь несколькими строками более эффективного кода. Другой раз, чтобы получить всего лишь несколько окончательных строк кода, программисту нередко приходится исписать груды бумаги или экранных страниц. Поэтому нелепо оценивать производительность программиста количеством операторов (команд), написанных в единицу времени, а объем выполненной работы — длиной окончательного листинга.

П р о с т а к. А как же тогда оценивать его работу?

П р о ф е с с о р. Сейчас мне хотелось лишь обратить ваше внимание на нетривиальность оценки труда программистов. Мы не будем углубляться в данную тему. Отмечу только, что на этот и многие другие вопросы организации производства программ обратил внимание Фредерик Брукс в своей знаменитой книге "Мифический человеко-месяц, или как создаются программные системы". Указанную книгу обязан прочитать каждый менеджер программного проекта, а также специалист, который в том или ином качестве вовлечен в процесс промышленного производства программных продуктов. Вы же, друзья, находитесь лишь только в начале пути к профессии программиста. Возможно, кто-то из вас свернет с него. Так что сначала взгляните на предстоящее путешествие с высоты птичьего полета. Сверху, возможно, вы не все, что надо для практической и профессиональной работы, разглядите достаточно отчетливо, но сможете хотя бы немного ориентироваться в "непролазных джунглях" программирования.

Глава 2

О языках программирования вообще

Профессор. Давайте сначала рассмотрим язык программирования не в узкопрофессиональном, а в более широком плане. Другими словами, я предлагаю вам сначала ознакомиться с самыми общими категориями, так или иначе практически воплотившимися во всех современных языках программирования. Без их понимания невозможно приступить к изучению ни первого в вашей жизни, ни последующих конкретных языков. Более того, не овладев общими понятиями, нельзя эффективно продвигаться в решении конкретных задач, усваивать уже устоявшиеся, а также новые технологии программирования.

Все языки программирования включают в себя, по меньшей мере, следующие понятия:

- переменные и типы данных;
- массивы данных;
- функции;
- объекты;
- операторы;
- выражения.

Эти краеугольные составляющие могут воплощаться или, другими словами, поддерживаться различными языками по-разному. Далее мы попытаемся уяснить собственно базовые понятия, но прежде обратимся к аналогиям с естественными языками.

Естественные языки базируются на таких категориях, как имя существительное, имя прилагательное, глагол и т. п. Слова, которые мы встречаем в обычной живой речи, принадлежат к той или иной категории языка. Так, слово "стол" — имя существительное, "красный" — прилагательное, а "писать" — глагол.

Категории естественного языка воплощаются в речи, исполняя ту или иную роль члена предложения (выражения языка), например, подлежащего, сказуемого, определения, дополнения, обстоятельства.

При этом исходные слова адаптируются согласно правилам построения предложений. Так, существительные и прилагательные склоняются по падежам, а глаголы спрягаются по временам. Вместе с тем слово одной и той же категории в предложении может выполнять различные функции. Например, существительное может фигурировать в предложении и как подлежащее (главный член предложения), и как дополнение (второстепенный член), в зависимости от структуры предложения. Итак, слова различных категорий языка, т. е. составляющих его базу, могут выполнять различные роли в предложении, или реализации словоупотребления. Как и в естественном языке, примерно то же самое происходит и в языках программирования.

Если вы новичок, то первое, на что вы должны обратить внимание при изучении языка программирования, — как реализуются на практике перечисленные понятия. Если вы уже знакомы с каким-нибудь языком, то при освоении нового следует лишь уточнить особенности реализации базовых понятий.

2.1. Переменные и типы данных

Профессор. Важнейшим в программировании является понятие переменной, с уяснения которого и следует начать. Оно изначально возникло в математике задолго до появления первых вычислительных устройств и программ для них. В математике под переменной понимают то, что может быть заменено конкретным значением. В уравнениях, известных вам еще со школы, фигурируют переменные, обозначаемые чаще всего символом "x". Разумеется, обозначать переменные можно и другими буквами или их последовательностями (например, "s", "myvar" или "сумма"). Вот типичный пример уравнения, встречающегося в математике: $x^2 - 2 = 0$.

Здесь x — символ, вместо которого можно подставлять различные значения, но не совсем произвольные, а именно числа. При этом говорят также, что x пробегает значения из множества чисел. Заметьте, что данная переменная не может принимать значения в виде строк, составленных из букв, а также дат (например, 16.12.2007) или других данных нечислового типа. Иначе говоря, переменная в математике — это символьное обозначение некоторого числового значения, но не любого, а лишь принадлежащего определенному множеству, например, всех чисел, или только действительных, или только целых, или каких-то других. Таким образом, переменная — это заместитель в некотором выражении (формуле) конкретного значения из наперед заданного множества или, как еще говорят, типа данных.

Итак, переменная может фигурировать в некотором математическом выражении. Что она там делает и каков смысл данного выражения с ее участием?

Простак. Переменная выполняет, как Вы нам напомнили, исключительно "представительские" функции. Она, так сказать, представляет некоторое, а лучше сказать, любое значение из заранее определенного множества, которое Вы, профессор, назвали типом значений.

Профессор. Совершенно верно, переменная представляет значения определенного типа. Вы достаточно аккуратно охарактеризовали понятие переменной, а потому, прошу Вас, продолжите рассуждение относительно участия переменной в выражении.

Простак. Попытаюсь, насколько смогу. В самом деле, переменная x участвует в некотором выражении вкупе с другими символами. Те, кто учился в школе, легко могут понять, что приведенное Вами, Профессор, выражение констатирует тот факт, что x в квадрате за вычетом двух равно нулю. Сам собой напрашивается вопрос: а каково конкретное значение переменной, которое обеспечивает выполнение данного равенства?

Зануда. Вот именно, приведенное ранее выражение просто констатирует, что некоторое число, пока нам неизвестное, удовлетворяет математическому уравнению. Вопрос о конкретном значении переменной x, удовлетворяющем данному равенству, кажется вполне естественным, он напрашивается сам собой.

Профессор. Уважаемые коллеги, столкнувшись с некоторым уравнением, вы сразу же формулируете некую математическую задачу, решение которой должны дать математики, хотя в данном случае любой школьник скажет, что только два числа, $+\sqrt{2}$ и $-\sqrt{2}$ (в десятичном выражении это 1,4142… и –1,4142…), удовлетворяют данному уравнению. Естественно, встреча с чем-то новым порождает попытку понять, к чему бы это. Если это математическое уравнение, то само собой речь пойдет о его решении, т. е. о вычислении тех значений переменной, при которых данное уравнение выполняется. Однако с точки зрения программирования дело совсем не в этом.

В математике, по крайней мере в школьной, обычно рассматривают всяческие уравнения и неравенства. При этом зачастую сложные и длинные выражения обозначают более компактными именами, т. е. вводят в обиход новые переменные. Этот прием позволяет в дальнейшем сократить математические выкладки, заменяя данное многосимвольное выражение более коротким.

В математике в договоре о новом обозначении или, другими словами, о вводе новой переменной, обычно говорят что-то в следующем духе:

"обозначим через y выражение $x^2 - 2$";

"пусть y равно $x^2 - 2$".

Как в математике, так и в программах подобные предложения записывают кратко: $y = x^2 - 1$.

Если в математике данное выражение является краткой записью соглашения о введении нового обозначения или констатацией равенства значений левой и правой частей равенства, то в программировании это команда, требующая присвоить переменной y значение выражения, расположенного справа от знака равенства. Чтобы эта команда могла быть выполнена компьютером, необходимо сначала вычислить значение выражения справа от знака равенства, и только потом присвоить полученное значение переменной.

Зануда. Я достаточно хорошо понял, как следует понимать в языках программирования простые выражения, например, $x = 5$ или $y = x^2 - 1$. Но тогда что означает в программировании выражение вида $x^2 - 2 = 0$? Мне непонятно, как можно левой части равенства, представляющей не просто какую-то переменную, а целое выражение, присвоить некоторое значение. Конечно, в математике такое возможно. Например, мы можем сказать так: "приравняем данное выражение нулю". А что это означает в языках программирования?

Профессор. В математике данное выражение обычно трактуется как утверждение о том, что левая часть равенства эквивалентна правой. Соответствующее предложение на естественном языке имеет изъявительное наклонение, т. е. это повествовательное предложение, утверждающее в данном случае факт равенства. В повелительном наклонении то же предложение можно сформулировать, например, так: "пусть $x^2 - 2 = 0$". В переводе на язык программирования выражение $x^2 - 2 = 0$ следует понимать не как требование присвоить левой части равенства значение правой, а как утверждение о равенстве левой и правой частей, истинность которого еще необходимо проверить. Иначе говоря, с точки зрения языка программирования данная запись соответствует операции сравнения выражений слева и справа от знака равенства. Результатом этого может быть одно из двух: ИСТИНА (если левая и правая части оказались равными) или ЛОЖЬ (в противном случае). Таким образом, выражение $x^2 - 2 = 0$ следует трактовать как "проверить, выполняется ли равенство $x^2 - 2 = 0$". Разумеется, такая проверка предполагает подстановку в его выражение конкретного значения вместо символа "x".

Зануда. Если я правильно Вас понял, то знак равенства в выражениях языка может играть две совершенно различных роли. Во-первых, в простейших выражениях вида $x = 5$ он обозначает повеление присвоить переменной x некоторое значение (в данном случае 5). Во-вторых, в выражениях вида $x^2 - 2 = 0$ тот же знак равенства следует трактовать как операцию сравнения того, что находится слева от него, с тем, что расположено справа.

Лишь по самой форме выражения мы можем определить, каков смысл употребления символа "=". Однако при изучении математики (вспомните школу!) с этим не возникало особых проблем.

Профессор. Ваша проницательность, уважаемый Зануда, выше всяких похвал. Именно потому, что один и тот же знак равенства может выполнять в математических выражениях различные роли, в языках программирования для каждой из этих ситуаций обычно предусматривают свой знак. Так например, для присваивания переменной некоторого значения обычно используют символ "=", а для сравнения значений двух выражений на предмет их равенства — двойной символ равенства "==". Например, выражение $y = x^2 - 1$ в программе следует понимать как команду присвоить переменной y значение выражения $x^2 - 1$, а выражение $y == x^2 - 1$ — как операцию сравнения значений y и $x^2 - 1$. Таким образом снимается проблема неоднозначной интерпретации операций, которые должен произвести компьютер. Замечу попутно, что в некоторых языках для присваивания употребляют символ ":=", а для сравнения — "=". Однако все это — особенности синтаксиса конкретных языков программирования, которые легко освоить с помощью справочников. Никогда синтаксис языка, каким бы вычурным он ни был, не создавал настоящих проблем для его изучения. Трудности возникают при словоупотреблении и формировании предложений. Человек, изучающий не первый в своей жизни язык программирования, легко и непринужденно усваивает его скучные нотации и спецификацию потому, что уже имеет некоторый опыт составления осмысленных предложений на искусственных языках.

А теперь в качестве тестового примера рассмотрим выражение вида $x = x + 1$. С точки зрения математики данное уравнение не имеет решения, поскольку легко преобразуется к виду $0 = 1$, что, очевидно, никогда не выполняется (данное равенство ложно). А раз так, то оно нам и не интересно. Однако с точки зрения программирования подобная запись, означающая присвоение переменной x значения выражения $x + 1$, имеет смысл. Что должен сделать компьютер, выполняя данное выражение?

Простак. Компьютер должен, как мне кажется, к текущему, т. е. ранее заданному значению переменной x прибавить единицу и полученный результат присвоить этой же переменной x. Другими словами, мы можем в качестве нового значения переменной использовать результат каких-то преобразований ее предыдущего значения.

Профессор. Совершенно верно. Так что выражение $x = x + 1$ в языке программирования представляет собой не требование (императив) решить уравнение относительно переменной x, а команду увеличить существующее ее значение на единицу.

Итак, в математике переменная, обозначаемая каким-нибудь именем из одного или нескольких символов, представляет (обозначает) собой некоторое значение из заранее оговоренного множества допустимых величин. В языках программирования понятие переменной имеет аналогичный смысл — это контейнер для хранения данных. Данные, которые могут храниться в переменной, называют значениями этой переменной. Они могут быть весьма разнородными: числами, текстовыми строками, датами, массивами и др.

Данные группируют в множества, называемые типами данных. Простейшие примеры типов данных — числа и произвольные последовательности символов (строки), встречаются и другие, например, дата, время и т. д. Переменные классифицируют в зависимости от типа данных, которые они могут хранить. Так например, если переменная предназначена для хранения чисел, то говорят, что она имеет числовой тип или, короче, что это числовая переменная.

Допустим, Вы имеете три яблока и одно из них отдали мне. Сколько яблок у Вас осталось?

Простак. Уважаемый Профессор, Вы, похоже, думаете, что мы законченные идиоты, не способные выполнить элементарные арифметические операции. Мы мало или даже ничего не знаем ни в программировании, ни в математике, но это еще не повод для унижения.

Профессор. Я и не сомневался, что сказку про Буратино Вы читали. Во-первых, никогда не обижайтесь на своего учителя, поскольку показать Ваше полное незнание или, в худшем случае, превратное знание — не главная его задача, а лишь способ акцентировать внимание на какой-то проблеме. Во-вторых, учитель всегда желает Вам добра, даже тогда, когда кажется, что он Вас недооценивает. Наконец, в-третьих, попытайтесь все же ответить на поставленный мной вопрос в контексте обсуждаемой темы о типах данных.

Простак. Ответ на Ваш вопрос очень несложен: два яблока у меня останется.

Профессор. Большинство согласится с Вами, уважаемый Простак, но программисты смогут предложить и другие варианты ответа.

Зануда. В жизни, как только встречаемся с числами, мы обычно абстрагируемся от того, к чему они относятся, к яблокам, автомобилям или еще чему-то. Поскольку в данном случае речь идет об изъятии яблока, то мы считаем, что этой операции соответствует обыкновенное вычитание одного числа из другого. Таким образом, мы абстрагируемся от яблок, оставляя в поле зрения только их количества, а передачу яблок другому лицу интерпретируем как арифметическую операцию вычитания или прибавления. Это настолько естественно, что и программист должен рассуждать аналогично.

Профессор. Браво, Зануда! Абстрагирование и интерпретацию данных, а также операций над ними в обычных и несложных житейских ситуациях наш мозг производит довольно быстро и в большинстве случаев правильно. Однако программист задает вопросы не только людям, а главным образом компьютерам, интеллектуальные способности которых пока значительно уступают человеческим. Поэтому он должен добиваться точных формулировок необходимых определений исходных данных, задач и алгоритмов их решения. Иначе он рискует остаться только на уровне благих пожеланий с нулевыми шансами что-либо сделать посредством компьютера.

В рассматриваемом нами случае программист мог выполнить все необходимые интерпретации условия задачи в уме, а машине поручить лишь рутинную операцию арифметического вычитания над числами 3 и 1, а именно вычислить значение выражения 3 – 1. Компьютер, очевидно, выдал бы в результате число 2, а программист сказал нам: "останется два яблока". Однако возможен и иной ход событий. Если в компьютер ввести данные в другой форме: "3 яблока" и "1 яблоко", то что получится в результате попытки вычислить выражение "3 яблока" – "1 яблоко"?

Простак. Я понял, что в рассмотренной ситуации делается попытка выполнить арифметическую операцию не над числами, а над символьными строками, пусть даже и содержащими числа. Это, возможно, поставит компьютер в тупик.

Зануда. Тупик будет более очевидным, если мы предложим компьютеру выполнить такую операцию: "Саша" – "Маша". Данные, участвующие в этой операции, не содержат чисел, а потому у нас нет никакого разумного основания проинтерпретировать действие, обозначенное символом "минус", как арифметическое вычитание. Что должно получиться в итоге? Каким должно быть слово, являющееся результатом данной операции?

Профессор. Я не исключаю, что можно придумать некоторую особую интерпретацию этой операции. Например, пусть это будет буквосочетание, которое в качестве фрагмента есть в первом, но отсутствует во втором слове. Однако такое определение слишком специфично или, иначе говоря, не общезначимо, а потому нам проще объявить данное и подобные выражения лишенными смысла и запретить их использовать в конструкциях языка программирования.

В жизни мы имеем дело с различными данными, относя их к разным типам по той простой причине, что для одного типа данных имеют смысл и допустимы одни операции, а для другого — другие. То, что можно делать с числами, интерпретируя операции над ними в математике, нельзя выполнять с последовательностями произвольных символов, сохраняя при этом интерпретацию

результата в той же математике. Наоборот, мы можем склеивать две последовательности символов или отсекать от них ненужные части, но результат подобных операций можно осмыслить, например, в редакторской деятельности, но только не в математике. Поэтому последовательности произвольных символов (строки) выделяют в отдельный тип данных. Далее, дату можно представить в виде набора из трех чисел (число, месяц и год), которые обычно отделяют друг от друга в одной записи точками. Очевидно, что сложение двух дат едва ли можно понять как сложение двух чисел. Набор из нескольких чисел — более сложный объект, чем отдельное число. Однако прибавление к дате некоторого обычного числа вполне разумно интерпретировать как некую новую дату, следующую за исходной спустя заданное число дней. Вычисление этой даты, т. е. представление в виде набора из трех чисел не совсем тривиально (нужно знать, сколько дней в каждом месяце и году). Иначе говоря, операции над датами отличаются от простых арифметических операций над обычными числами, а потому даты также относят к особому типу данных. В языках программирования существуют так называемые встроенные типы данных. Это общезначимые, наиболее часто употребляемые и достаточно отчетливо интерпретируемые всеми типы. Вместе с тем, современные языки программирования допускают задание программистом своих особенных типов данных. Например, вы можете определить тип данных НЕДЕЛЯ, который содержит 7 элементов, представленных целыми числами (например, 1, 2,…,7) или названиями дней (например, "понедельник", "вторник",…, "воскресенье").

Во многих языках (например, C и Pascal) переменные и их тип должны быть объявлены прежде, чем они будут использованы в программе. Если при этом переменная изначально была объявлена, например, как числовая, то уже нигде в программе ей нельзя присвоить значение какого-нибудь другого типа. Говорят, что это языки с сильным контролем типов.

Однако есть языки со слабым контролем типов (их еще называют языками со свободными или динамическими типами), типичные примеры — JavaScript и PHP. В них тип переменной не фиксируется раз и навсегда в пределах программы, одна и та же переменная в различных фрагментах программы может принимать значения то одного, то другого типа по усмотрению программиста.

Простак. Хотелось бы все это рассмотреть на примерах.

Профессор. Пока мы не будем углубляться в детали реализации идеи переменных в конкретных языках. Замечу также, что мы не затронули еще одну важную тему, связанную с переменными, а именно области их видимости. Дело в том, что переменные могут быть глобальными и локальными. Глобальные доступны (т. е. видны) из любого места программы,

локальные — только в том блоке программного кода (фрагменте программы), где они были созданы. Типичный пример блока кода — функция. Понятие функции мы рассмотрим позже, а пока лишь отметим, что она может содержать в своем коде переменные, недоступные (невидимые) из внешней по отношению к ней части программы. Мы изучим данную тему далее применительно к двум языкам — JavaScript и PHP.

2.2. Массивы данных

Профессор. Ранее мы рассмотрели в общих чертах, что такое переменные и типы данных. Среди последних выделяют простые или примитивные типы, такие как числа, строки символов, логические значения (ИСТИНА и ЛОЖЬ). Они потому и называются примитивными, что интуитивно понятны, однако наш разум требует чего-то большего. А именно он без особых усилий обнаруживает, что могут быть и более сложно устроенные данные. Такие типы называют составными. Например, даты, представляющие собой наборы из трех целых чисел, очевидно сложнее, чем просто одиночные числа. Особенно важны с практической точки зрения массивы данных, т. е. наборы не двух-трех, а произвольного количества данных. Наиболее общее определение массива — упорядоченная последовательность данных.

Простак. Иначе говоря, массив состоит из элементов, следующих друг за другом в определенном порядке. К настоящему времени мы усвоили кое-что о примитивных типах данных — числах, строках и логических значениях. А какого типа могут быть элементы, входящие в последовательность, именуемую массивом? Я легко представляю себе последовательность чисел, символьных строк и даже значений типа ИСТИНА и ЛОЖЬ. Но если допустимы данные и других, более сложных типов, то можно сотворить что-то грандиозное.

Профессор. В некоторых языках (например, в C и Pascal) массив определяется как упорядоченная последовательность данных одного и того же примитивного типа, например, массив целых чисел, массив символьных строк и т. п.

Простак. А жаль. Если бы в массив можно было "загнать" данные сложных (составных, как Вы их называете) типов, то появилась бы возможность реализовать двумерные или даже *n*-мерные таблицы.

Зануда. Мне не совсем понятно, на что намекает Простак. Меня больше интересует само понятие массива, а потому я хотел бы его уточнить. Вы говорите, Профессор, что массив — это упорядоченная последо-

вательность данных. А если изменить исходный порядок сортировки данных? Станут ли два массива, содержащие одни и те же данные, но расположенные по-разному, эквивалентными, или, лучше сказать, одинаковыми с точки зрения интерпретатора, т. е. того, кто будет выполнять наши программные коды?

Профессор. Изменение первоначального порядка следования данных (элементов массива) приводит к другому массиву. Так что массив — это не просто "куча данных", их порядок в определении массива имеет принципиальное значение. Определяя массив как последовательность данных, мы закрепляем память о нем, присваивая его некоторой переменной, т. е. назначая ему имя подобно тому, как мы это делаем с данными примитивных типов. Иначе говоря, мы назначаем по своему достаточно произвольному выбору имя переменной и присваиваем ей значение, являющееся массивом.

Массив, подобно обычной переменной, имеет имя, так что одним именем мы можем обозначить не одно, а сразу несколько простых значений. Другими словами, переменной типа массив можно присвоить набор примитивных значений, упорядоченных произвольным образом.

Простак. Массив — это последовательность данных, причем упорядоченная. На практике обычно требуется не просто обратиться к массиву в целом, а узнать конкретное значение какого-то определенного элемента этого массива. Чтобы получить доступ к значению обычной переменной (т. е. переменной примитивного типа данных), достаточно указать в программе имя этой переменной. А как получить нужное значение элемента массива?

Зануда. Поскольку массив задается не просто своими элементами, но и порядком их следования, то, наверное, можно как-то указать номер нужного элемента. Насколько я правильно понимаю, любой порядок всегда связан с возможностью нумерации целыми числами.

Профессор. Совершенно верно. Обычно элементы массива нумеруются целыми числами, начиная с нуля, а не с единицы (первый элемент массива имеет номер 0, а не 1). Номер элемента массива еще называют индексом. Так, если a — массив из n элементов, а нам требуется i-й элемент, то пишут $a[i]$. При этом если вам нужен самый первый элемент массива, то следует обратиться к $a[0]$, а не к $a[1]$. В частности, значение последнего элемента есть $a[n-1]$, а не $a[n]$. Впрочем, должен вас предупредить, что данное правило не повсеместно. Если я не ошибаюсь, то в языке VBScript это не так.

Простак. Можно сказать, что различные элементы массива обозначаются одним и тем же именем, но с различными индексами.

При этом имена элемента и массива, которому он принадлежит, должны совпадать, а индекс — быть целым числом в диапазоне между нулем и значением, на единицу меньшим количества всех элементов.

П р о ф е с с о р. Правильно. Добавлю только, что во многих языках обращение к элементам массива по индексу — не единственно возможный способ, но об этом позже.

З а н у д а. Наглядно массив можно представить себе в виде таблицы из одного столбца, в ячейках которой находятся значения элементов.

П р о ф е с с о р. Это хорошая интерпретация массива.

З а н у д а. Мне не дают покоя намеки Простака на дальнейшее развитие понятия массива. Не знаю, о чем он думал, но меня сейчас интересует, а как создать таблицу с несколькими столбцами?

П р о с т а к. Позвольте, Профессор, я сам попытаюсь ответить Зануде. Думаю, что это можно сделать, задав массив, элементы которого сами являются массивами. Последние будут определять строки или ряды прямоугольной таблицы, если, конечно, все они состоят из одинакового количества элементов. Тогда, чтобы получить данные из конкретной ячейки таблицы, потребуются два индекса: первый будет указывать на строку, а второй — на столбец (ячейка таблицы, как известно, — пересечение строки и столбца).

З а н у д а. Однако в таком случае исходный массив будет содержать не примитивные данные, а составные. Вы же, Профессор, говорили раньше, что массив должен содержать данные одного и того же примитивного типа.

П р о ф е с с о р. Я говорил лишь, что это требуется в некоторых, но не во всех языках. Так например, в JavaScript, PHP и ряде других языков массив может состоять из разнотипных и не обязательно примитивных данных.

З а н у д а. Мы много говорили о массивах как о некоторых существующих объектах. Но объект, с которым предстоит иметь дело в программе, следует, прежде всего, как-то задать. А как определить массив?

П р о ф е с с о р. Хорошо, уважаемый Зануда, что Вы вспомнили об этом. В различных языках это делается по-разному, а нередко и несколькими способами. Например, в JavaScript массив с именем, скажем, `myarray` можно задать с помощью такого выражения:

```
myarray = [150, 200, "Вася", "Петя", "Маша"].
```

В квадратных скобках перечисляют значения элементов массива, разделенные запятыми. Как я уже упоминал, такой синтаксис не универсален. В раз-

личных языках задание массива оформляется по-разному. Рассматривая приведенное выражение как задание массива в JavaScript, мы можем отметить, что здесь значением элемента, например, `myarray[1]` является число 200, значением `myarray[2]` — символьная строка "Вася".

З а н у д а. Я вижу, что понятие массива в языках JavaScript и PHP было усовершенствовано так, чтобы элементы массива могли быть разнотипными и даже составных типов. С точки зрения удобства практических применений это здорово. Однако исходное достаточно простое понятие массива оказалось размытым. Его следовало бы скорректировать хотя бы так: "массив — это упорядоченная последовательность данных". При этом мы не акцентируем внимание на типах данных, а следовательно, допускаем любые типы, предусмотренные в языке.

П р о ф е с с о р. Понятие массива в исходной и весьма ограниченной формулировке было первоначально реализовано в языках программирования как обращение к одноименным данным, расположенным рядом друг с другом в памяти компьютера. Это позволило обеспечить быстрый доступ к любому элементу из сколь угодно большого множества. Изначально здесь аппаратно-программное представление довлело над общезначимым или пользовательским. Понятно, что если заполняющие неразрывную область физической памяти данные однотипны, то поддержать технически (и средствами языка) такую конструкцию проще, чем в случае разнотипных данных.

Однако даже фундаментальные понятия подвержены изменениям с течением времени. Чтобы расширить первичное понятие массива как набора однотипных данных, в языке C была введена конструкция, называемая структурой. Структура — это набор элементов (называемых еще полями), которые предназначены для хранения данных различных типов. В языках JavaScript и PHP структур нет, но существуют массивы разнотипных данных. Это расширенное понятие массива можно реализовать в указанных языках посредством структур языка C. Мы не будем сейчас рассматривать, что такое структуры. Программист может и не знать, что и каким именно образом реализовано. Например, он может понимать массивы в JavaScript и PHP как структуры в языке C, просто называя их массивами. В действительности же расширенное понятие массива в JavaScript и PHP было реализовано посредством не структур, а объектов, т. е. еще более обобщенных понятийных и технологических образований. Объекты — обобщение структур в том смысле, что они могут хранить, как контейнеры, не только данные различных типов, но и функции (называемые методами), т. е. некие элементы, обладающие способностями выполнять некоторые действия. В частности, когда объект содержит только переменные для хранения данных, но не имеет собственных функций (методов), он практически ничем не отличается от структуры или,

если угодно, от массива с произвольными типами данных элементов. Но об объектах мы поговорим отдельно. Прежде следует выяснить, что такое функция.

2.3. Функции

Профессор. Программа представляет собой последовательность выражений языка. Нередко случается, что какая-то часть программы (блок кода) неоднократно повторяется. Чтобы устранить подобного рода избыточность кода, используют понятие функции. Функция — это поименованный блок кода, который вызывается в нужных местах программы по имени. Другими словами, функция представляет собой подпрограмму, которую можно вызвать из основной программы, причем неоднократно. Повторяющийся (да и не только) блок программного кода обычно обозначают некоторым уникальным именем, чтобы потом при необходимости обратиться к нему по этому имени. Как видно, это простая и естественная идея, направленная на облегчение реализации сложных проектов, состоящих из более простых программ.

Подпрограмма или, другими словами, функция должна быть связана (интегрирована) с основной программой, так сказать, со своим внешним окружением. С целью обеспечения взаимодействия с остальной частью программы для функции можно предусмотреть так называемые вход и выход. Вход в функцию — это передача ей аргументов — данных, полученных во внешней части программы. Получив данные из своего внешнего окружения (внешней программы), функция должна их как-то обработать: выполнить некоторые действия, вычислить какое-то значение. Выход из функции — значение, вычисленное блоком кода данной функции и передаваемое во внешнюю часть программы. Входные данные называют параметрами, а выходные — возвращаемым значением. Впрочем, функция может и не принимать никаких параметров, а также ничего не возвращать. Что принимает в качестве параметров и что возвращает функция в результате своей работы, определяет программист, т. е. автор-разработчик программного кода.

Функция, создаваемая разработчиком, должна иметь определение, а ее применение в программе называют вызовом функции. Например, в JavaScript и PHP (а также в большинстве других языков) такое определение начинается с ключевого слова `function`, за которым следуют имя функции, пара круглых скобок, внутри которых можно указать список параметров, а затем блок кода, заключенный в фигурные скобки:

```
function имя_функции(список_параметров){
      блок кода
}
```

Замечу попутно, что блок кода в фигурных скобках называют еще телом функции. В нем записывают выражения языка, которые функция должна выполнить, если будет вызвана из основной программы. Таким образом, само по себе определение функции в программе еще ничего не выполняет. Оно только указывает интерпретатору, что должно быть выполнено при обращении к данной функции.

Для вызова функции в том или ином месте программы указывают следующее выражение:

```
имя_функции(список_параметров).
```

Круглые скобки после имени функции записывают независимо от того, предусмотрены для нее входные параметры или нет.

В программе вы можете один раз задать определение функции, а затем, по мере необходимости, вызывать ее столько раз, сколько требуется. Таким образом, вам не придется многократно воспроизводить в программе код тела функции.

Простак. Верно ли я Вас понял, что основная программа может содержать как определения, так и вызовы функций?

Профессор. Да, это так. В программе, содержащей вызовы функций, должны быть доступны их определения. Простейший, но не единственно возможный способ обеспечить это заключается в том, чтобы заключить определения функций в тот же программный код, в котором они где-то вызываются. Опытные программисты обычно размещают определения функций в одном месте, например, в конце или, наоборот, в начале кода программы.

Простак. Мне понятно, что в некоторых ситуациях может потребоваться передать какой-то функции данные из основной программы. Но зачем функция должна что-то возвращать? Ведь она может выполнить предусмотренные действия, например, отправить данные по электронной почте, открыть или закрыть окно, отобразить картинку и т. п., а внешней программе не потребуется от нее какое-либо значение.

Профессор. Если вызывающей программе ничего не требуется получить от вызванной функции, то программист может определить эту функцию как ничего не возвращающую. Это разрешается в JavaScript и PHP, но может быть недопустимо в других языках. Если даже практически не важно, что именно возвращает функция, все равно желательно определить для нее некоторое возвращаемое значение. В таких случаях, часто указывают, например, пустое значение, обозначаемое как null. Возможны и другие варианты. Кроме того, функции, возвращающие какие-то значения, можно использовать в выражениях с операторами (например, арифметическими), а также в качестве параметров других функций. Если функция ничего не воз-

вращает, то ее применение в выражениях с операторами обычно чревато сообщениями об ошибках.

Чтобы функция что-то возвращала, необходимо записать в ее теле специальный оператор возврата:

```
return возвращаемое_значение;
```

Приведу пример (листинг 2.1).

Листинг 2.1

```
Оклад = 10000;
Процент = 15;
Выплата = Оклад + Премия(Оклад, Процент);
function Премия(Оклад, Процент){
return Оклад*Процент/100;
}
```

В листинге 2.1 переменным `Оклад` и `Процент` сначала присваивают некоторые числовые значения. Затем вычисляют выражение, определяющее объем выплаты некоторому сотруднику с учетом его оклада и премии. Премия рассчитывается с помощью функции `Премия(Оклад, Процент)`, принимающей два параметра: оклад и процент от оклада; данная функция возвращает величину премии (символом `*` обозначен оператор арифметического умножения). Возвращаемое значение прибавляется к значению переменной `Оклад`, а полученный результат присваивается переменной `Выплата`. Определение функции `Премия(Оклад, Процент)` размещено в конце текста программы.

З а н у д а. А на каком, собственно, языке написан данный программный код?

П р о ф е с с о р. На вымышленном. Я хотел сказать, что рассмотренный пример иллюстрирует принцип применения функции, а потому не важно, какой именно язык выбрать. Главное, что все изложенное ранее должно позволить вам понять данный код. Тем не менее, признаюсь, что приведенный мной пример написан и на языке JavaScript. Напишите в обычном текстовом редакторе, например, Блокноте Windows следующие строки (листинг 2.2).

Листинг 2.2

```
<HTML>
<SCRIPT>
Оклад = 10000;
```

```
Процент = 15;
Выплата = Оклад + Премия(Оклад, Процент);
function Премия(Оклад, Процент){
return Оклад*Процент/100;
}
alert("Выплата = "+Выплата);
</SCRIPT>
</HTML>
```

Это HTML-код с внедренным в него сценарием на JavaScript. Код сценария заключен между тегами <SCRIPT> и </SCRIPT> и отличается от рассмотренного ранее только встроенной функцией alert() для вывода сообщений в диалоговом окне. Сохраните данный код в файле с расширением htm или html, а затем откройте его в Web-браузере, например, Microsoft Internet Explorer или Mozilla Firefox. В результате появится диалоговое окно с сообщением "Выплата = 11500".

З а н у д а. Однако я не вижу особой целесообразности применять здесь функцию, поскольку нет ее повторного использования. Было бы проще написать более короткий код (листинг 2.3).

Листинг 2.3

```
<HTML>
<SCRIPT>
Оклад = 10000;
Процент = 15;
Выплата = Оклад + Оклад*Процент/100;
alert("Выплата = "+Выплата);
</SCRIPT>
</HTML>
```

П р о ф е с с о р. В данном случае Вы правы. Однако мой пример — лишь иллюстрация работы с функцией, без учета целесообразности ее конкретного применения.

Появление в 1960-х годах в языках (например, ALGOL и FORTRAN) функций позволило писать программы, более ясные в структурном отношении и компактные по объему кода. В идеале основная программа могла состоять из одних только вызовов функций, перемежаемых, возможно, операторами управления (условными переходами и/или циклами). Разрабатывать и отлаживать такие программы стало и быстрее и легче. Программист при желании

мог писать программы, широко используя вызовы функций, тела которых еще не написаны. Чтобы такая программа как-то работала без зависаний и выдачи системных сообщений об ошибках, вместо настоящего кода функции можно было временно поставить так называемые заглушки, например, выводы специальных сообщений программиста, например, "работает функция myfunc". По появлению подобных сообщений можно судить, что вызов функции произошел, и при этом в главной программе не возникло коллизий.

Создав основную программу в общем виде, можно приступить к доработке ее деталей — написанию требуемых кодов, составляющих тела настоящих функций, которые до сих пор были замещены примитивными заглушками. Это так называемый способ (или стиль) программирования "сверху вниз", при котором создаваемая программа постепенно обретает требуемые функции. Усовершенствование такой программы впоследствии сводится главным образом к модификации кодов уже существующих функций и может быть выполнено другими людьми, а не только автором программы. При этом, разумеется, необходимо сохранить прежний интерфейс (вход-выход или связи с внешним окружением), чтобы не пришлось перестраивать внешнюю программу, содержащую вызовы модифицируемой функции.

Даже если какой-то блок кода предполагается задействовать в программе всего лишь один раз, все равно следует подумать, а не оформить ли его в виде функции, чтобы сделать основной текст программы более понятным и к тому же обеспечить возможность повторного использования этого блока в перспективе.

З а н у д а. Я заметил еще одно странное обстоятельство: функции `alert()` в качестве параметра передается выражение в виде суммы разнотипных данных (символьного и числового значения):

```
"Выплата = "+Выплата.
```

Здесь `"Выплата = "` — символьная строка, а `Выплата` — переменная числового типа.

Однако Вы, уважаемый Профессор, ранее говорили, что типы данных для того и придуманы, чтобы выполнять операции над однотипными данными.

П р о с т а к. Тем не менее, сообщений об ошибках не было, а результат оказался правильным!

П р о ф е с с о р. Рассмотренный сценарий написан на JavaScript, т. е. на языке со слабым контролем типов данных. Если в выражении этого языка встречается операция над данными различных типов, то происходит такое автоматическое преобразование этих данных, при котором может быть получен более или менее осмысленный результат. Так например, если требуется к символьной строке прибавить число, то интерпретатор JavaScript сна-

чала преобразует его в соответствующую строку символов, которую затем "приклеит" к первой строке. В результате получится символьная строка. Обратите внимание, что число, записанное как последовательность цифр (возможно, с разделительной десятичной точкой и знаком "+" или "–" перед числом), с точки зрения типа данных отличается от того же числа, заключенного в кавычки. В последнем случае это будет строка, содержащая число, а не собственно число. В языках со слабым контролем типов операции над разнотипными данными выполняются с учетом некоторых правил автоматического преобразования типов (так называемые правила приведения типов), которые по умолчанию разные для различных языков. Поэтому многие программисты, чтобы не запутаться и все держать под своим полным контролем, выполняют предварительное приведение данных к нужному типу с помощью специальных встроенных функций, а не полагаются на автоматические преобразования. Однако хорошее знание особенностей языка позволяет создавать более короткие и элегантные программы.

Простак. Вы использовали в коде, вставленном в HTML-документ, функцию `alert()` для вывода на экран диалогового окна с некоторым сообщением. А откуда она взялась? В сценарии же нет ее определения.

Профессор. Если быть точным, то `alert()` — не функция языка JavaScript, а метод объекта `window`, который принадлежит объектной модели браузера. Методами называют внутренние функции объектов. Иначе говоря, этот метод, как и множество других объектов браузера и загруженного в него документа, составляют внешнее окружение сценария на JavaScript. К объектам этого окружения можно обратиться из сценария, т. е. прочитать и даже изменить значения их свойств. Тем самым обеспечивается возможность управлять внешним видом и информационным содержанием Web-страницы.

Простак. Но чтобы реализовать все это, необходимо знать устройство или, лучше сказать, объектную модель документа и браузера.

Профессор. Разумеется, модель того, с чем требуется иметь дело посредством языка программирования, необходимо знать. Но это особая тема. Сейчас же мы рассматриваем языки программирования, причем с высоты "птичьего полета" и до поры не будем слишком приближаться к "земле".

Зануда. Осмелюсь напомнить, уважаемый Профессор, что ранее вы нам что-то сказали о видимости переменных. В частности, Вы заметили, что в коде функции (ее теле) можно задать локальные переменные, которые недоступны во внешней части программы.

Профессор. Да, в теле функции можно определить переменные, невидимые в ее внешнем окружении. Даже если имена внешних

и внутренних (локальных) переменных совпадают, то их значения могут отличаться. Однако подробное изучение этой темы лучше провести на примере конкретных языков, что мы и сделаем немного позднее.

2.4. Классы и объекты

Профессор. При моделировании какой-то части реального мира мы вводим в обиход некоторые сущности (предметы) и отношения (связи) между ними. Предмет или объект, каким бы он ни был в действительности или в нашем воображении, в абстрактном виде можно представить себе как нечто, обладающее определенными свойствами. Об объектах окружающего нас мира, а также нашей мысли мы судим по их свойствам, т. е. по тому, что можем воспринимать нашими органами чувств. Иначе говоря, объект внешнего или нашего внутреннего мира представляется нам не совсем таким, каким он выступает в действительности, а лишь посредством тех свойств, которые мы способны воспринять. Здесь мы, похоже, стучимся в дверь философии. Однако пора приземлиться, коль скоро мы занимаемся прикладной наукой. Попробуйте, друзья, для начала описать такой объект, как автомобиль.

Зануда. Я сразу представляю себе с помощью зрительной памяти один или даже несколько автомобилей различных марок и окрасок. Несмотря на то, что они различаются между собой внешне и внутренне, существует нечто общее между ними, но именно это общее я и затрудняюсь сформулировать во всей исчерпывающей данное понятие полноте.

Простак. Тебе, Зануда, всегда удается сказать что-то такое, возразить чему очень трудно. Тем не менее, я тоже способен к разглагольствованиям отвлеченного порядка. В контексте нашего разговора я способен признать, что между различными автомобилями есть что-то общее, позволяющее отнести их к одному и тому же классу, а именно классу автомобилей. Осмелюсь заметить, что это, по-видимому, наличие колес и двигателя внутреннего сгорания. Не спешите поймать меня на слове и "схватить за язык", уважаемые Зануда и Профессор. Разумеется, это не все и, возможно, не самые главные особенности, с помощью которых мы можем выделить автомобиль из окружающего мира. Тем не менее, по данным признакам мы все-таки отличим автомобиль от паровоза, корабля или обезьяны, например, а это уже кое-что.

Зануда. Выделить объект или, как говорят, сущность из окружающего мира, — трудная задача, тем более при первом рассмотрении. Но мы должны это делать, чтобы как-то справиться со сложностью реального мира при попытке его смоделировать, а значит — познать.

Профессор. Вот! Уважаемый Зануда сформулировал самое главное, к чему стремятся не только программисты, создающие свои опусы на искусственных языках для компьютеров, но и математики, пишущие свои пьесы на нетленном языке мысли, хотя и снабженном специфической символикой. Все они, философы, математики, программисты, поэты и обычные люди, говорящие на языке, ставшем им доступным и родным просто от рождения, трансформируют окружающий нас многогранный, с трудом постигаемый и не объяснимый в полной мере мир в свои внутренние модели. Зачем? А понять внешний мир иначе как его отражение во внутреннем невозможно. Впрочем, я так же, как и вы, не очень-то искусный философ, а потому призываю вас сейчас вернуться к весьма практической теме нашего разговора.

Итак, объект — весьма абстрактная категория нашего сознания и языка как средства сделать эту мысль доступной не только нашему внутреннему взору, но и соседям. Мы должны как-то "приземлить" это довольно общее понятие, чтобы оно стало податливой для моделирования в довольно специфических и весьма "аскетических" искусственных языках, а значит и в программировании.

Объекты отличаются друг от друга составом и/или значениями (параметрами) свойств. Объект — это своего рода контейнер, содержащий информацию о себе самом. Какова эта информация? Очевидно, что это сведения о свойствах объекта. Но каким образом сохранить эту информацию для последующего использования?

Простак. Чтобы сохранить какую-то информацию на будущее, мы ее запоминаем. В естественных условиях это как-то делает мозг, способный в дальнейшем извлечь на поверхность нашей памяти (лучше сказать, — сознания) с помощью заранее заготовленной ссылки — имени интересующей нас информационной области. Когда же мы создаем компьютерную программу, запоминание происходит путем присвоения данных, содержащих интересующую нас информацию, некоторой переменной. Конкретно, мы просто назначаем данным символическое имя, через которое впоследствии можно получить доступ к интересующим нас данным.

Профессор. Я бы не сказал лучше, чем Вы, уважаемый Простак.

Зануда. Я удивлен, как тебе, Простак, удается сходу все это понять и даже выразить достаточно членораздельно?

Простак. А я просто стараюсь относиться к излагаемому нашим Профессором без предубеждений, не противопоставляя ему без особой надобности свой предшествующий опыт. Мне, в конце концов, нужен именно его опыт, а не подтверждение ценности своего собственного.

З а н у д а. Я привык, что ты обычно говоришь в безапелляционной манере. Отчего в твоих словах теперь проглядывает осмотрительность, которой раньше недоставало?

П р о с т а к. Правда, я не склонен излишне мудрствовать. Я живу, пока вижу цель и путь к ее достижению. Вместе с тем, я не могу быть счастлив только тем, чтобы барахтаться в грязи дороги, пусть и ведущей к сияющей чистотой цели.

З а н у д а. Чтобы не "барахтаться в грязи", как ты говоришь, надо взлететь или хотя бы подпрыгнуть над строящейся дорогой. А подпрыгнув, что ты оттуда сможешь увидеть? Ту же непроторенную грязь, пусть даже и с высоты белоснежных вершин?

П р о с т а к. Ты, Зануда, просто провоцируешь меня на то, чтобы я спустился с небес, где ты сам давно обосновался. Похоже, ты хочешь просто вытолкнуть меня оттуда, как кукушонок. Неужели из-за меня тебе там стало нестерпимо тесно?

З а н у д а. Не сердись, дорогой Простак. Мне там и в самом деле хорошо летать, но холодно и одиноко, а потому я искренне рад твоему беспокойному соседству. Однако наше сближение грозит мне потерей оппонента — того, кто не даст моим мыслям протухнуть. Поэтому, любя по большому счету, я подспудно стараюсь оттолкнуть тебя. Так что прости.

П р о ф е с с о р. Ребята, не увлекайтесь выяснением личных отношений. Истина, к которой вы стремитесь, не обязательно благотворна с точки зрения ваших субъективных представлений о настоящем и будущем. Незнание истины лишь откладывает на будущее возможные очарования (и разочарования), создавая только иллюзию предвкушения счастья (или несчастья) в настоящем. Но пренебрежение уже доступной истиной, а также отказ от ее поиска однозначно чреваты неизбывными горестями впоследствии.

Мы с вами слишком отвлеклись от главной темы обсуждения. Напомню, что речь идет об объектах, т. е. таких программных конструкциях, которые характеризуются или, лучше сказать, предстают перед нами посредством своих свойств. Мы должны признать, что наше знание об объекте в целом заключается в знании его отдельных свойств. Такой подход к пониманию целого присущ и некоторым разделам математики. В частности, в аксиоматических теориях объект изучения задается набором аксиом, которые постулируют, какими свойствами он обладает. Так что объект в математике предстает перед нами как нечто (назовите и нарисуйте в своем воображении его как угодно), обладающее определенными свойствами. Далее математическая теория развивается, обрастая теоремами, чтобы сообщить нам информацию об объекте изучения, которая логически следует из первоначальных аксиом, т. е. утвер-

ждений, принимаемых без доказательства в качестве самоочевидных или достаточно обоснованных практическими соображениями. По мере развития теории об объекте мы получаем более или менее отчетливое представление о том, что есть собственно объект. Примерно так обстоят дела и при математическом моделировании объектов.

В программировании применяется довольно абстрактное и "технологичное" понятие об объекте как о контейнере, содержащем внутренние переменные (свойства), а также внутренние функции (методы). Внутренние переменные и функции можно называть свойствами объекта. Свойства-переменные предназначены для хранения данных, а свойства-методы — для выполнения некоторых действий. Особенность и тех, и других в том, что в программе они играют не самостоятельную роль, а относятся к тому объекту, для которого были определены и с которым тесно связаны. Представьте себе какое-нибудь бытовое устройство, например, микроволновую печь или телевизор. С точки зрения пользователя его можно представить абстрактно в виде "черного ящика". Ящик называется "черным" потому, что его внутренности непосредственно недоступны ни органам чувств, ни уму. По крайней мере, мы, как пользователи, не хотим разбираться в премудростях его внутреннего устройства. Вместе с тем, к этому "ящику" подключены датчики или измерительные приборы (вольтметр, амперметр, манометр, термометр, спидометр и т. п.), показания которых мы можем наблюдать. Кроме того, имеются органы управления (кнопки, переключатели), с помощью которых мы можем заставить наше устройство выполнить ту или иную функцию. Нажимая кнопку, мы инициализируем соответствующую ей функцию, т. е. заставляем наше устройство выполнить некоторое действие. При этом показания каких-то датчиков, возможно, изменятся. Иначе говоря, индикаторы могут реагировать на воздействия органов управления. С точки зрения программирования датчики — это прообразы свойств-переменных, а органы управления — методов.

Простак. Однако я хотел бы заметить, что бытовые приборы с их индикаторами (датчиками) и кнопками (управляющими элементами) обычно снабжают инструкциями или описаниями того, как они работают и как правильно их использовать по назначению. Я обращаю внимание на то, что знание о свойствах-переменных и свойствах-методах это еще не все знание об объекте. По меньшей мере еще требуется информация о способах применения.

Профессор. Верно, описание объекта в виде инструкций и руководств для пользователя представляет не что иное, как модель его функционирования, т. е. взаимосвязь показаний датчиков и управляющих воздействий. Объект в программе существует в виде набора свойств, но чтобы его использовать, необходимо знать его модель поведения и программный

интерфейс. Модель поведения объекта обычно задается в его описании от производителя.

П р о с т а к. Программный интерфейс? А что это такое?

П р о ф е с с о р. Это описание (спецификация) взаимодействия какой-нибудь более или менее автономной программной единицы (например, объекта) с внешней программой. Он устанавливает правила работы с значениями свойств-переменных и методами. Другими словами, программный интерфейс определяет инициализацию объекта и способы взаимодействия с ним в вашей программе.

Введение в программирование понятия объекта обусловлено главным образом сложностью современных программ. Типичный пример подобных программ — игры, такие как "стратегии", "симуляторы" и "стрелялки". В них много разнообразных персонажей и объектов с нетривиальным поведением. Как хорошо известно, с проблемой очень большой сложности обычно справляются посредством классификации. Так, миллиарды представителей животного мира разделяют на классы, а последние — на подклассы (отряды, семейства, роды, виды и т. п.).

Представьте себе какой-нибудь автомобиль и попробуйте описать его, указав присущие ему свойства.

П р о с т а к. У автомобиля есть кузов, мотор, колеса и еще масса других свойств, которые можно упомянуть или нет в зависимости от конкретной задачи. Так, если нам необходимо отличить данный конкретный автомобиль от всех других, то понадобится очень много признаков.

З а н у д а. А чтобы отличить данный автомобиль от всех других существующих и мыслимых объектов, потребуется необозримо больший список признаков.

П р о ф е с с о р. Тем не менее, можно построить иерархическую систему классов интересующих нас объектов, постепенно переходя от классов к содержащимся в них подклассам и тем самым уточняя информацию об объекте. Например, класс всех автомобилей охарактеризуем наличием двигателя с приводом на колеса и системы рулевого управления. Из данного класса выделим два подкласса — легковых и грузовых автомобилей, отличая их друг от друга по особенностям устройства кузова и/или разрешенному максимальному весу. Подкласс легковых автомобилей разобьем еще на несколько подклассов следующего иерархического уровня по объему цилиндров двигателя, например. Подобным образом можно построить следующий уровень классификации. Количество уровней вложенности одних классов в другие зависит от требуемой степени детализации информации об объекте.

Признаки или, другими словами, свойства классов также могут быть теми или иными в зависимости от наших возможностей их наблюдать и целей их последующего использования. Наконец, сама система классов может быть более или менее удачной. Важно то, что мы создаем сложное описание объекта постепенно, по частям. Причем на каждом этапе мы имеем дело с относительно простым и небольшим по объему объектом (классом). Говорят, что он наследует свойства своего надкласса, называемого еще родительским. Если какая-то система классов для описания всех интересующих нас объектов уже построена, то со временем некоторые из данных классов могут служить для описания других объектов. Например, может статься, что некоторые классы верхних иерархических уровней пригодятся для описания самолетов или иных транспортных средств. Следовательно, каждый подкласс кроме своих собственных свойств имеет еще и свойства класса, которому принадлежит. Такой подкласс содержит в собственном описании лишь специфические свойства. Свойства классов-родителей при этом в нем не описываются, а просто наследуются, т. е. заимствуются из описаний родительских классов.

Простак. А как создаются классы, неужели с помощью функций и переменных?

Профессор. Эра объектно-ориентированного программирования (ООП) началась в 1990-х годах. Современные языки, поддерживающие ООП (например, C++, Object Pascal, PHP и др.), позволяют создавать классы посредством специальных синтаксических средств, например, так:

```
class имя_класса {
      определение свойств-переменных
      определение свойств-методов
}
```

Как видите, класс — это просто контейнер, содержащий описание объекта. С помощью этого описания можно создать один или несколько однотипных объектов или, другими словами, экземпляров класса. Например, пусть класс описывает такой объект как автомобиль конкретной модели (например, "Мерседес 600"), имеющий среди прочих свойств еще и цвет кузова. Тогда мы можем на основе общего описания автомобиля (т. е. класса) создать множество его экземпляров (объектов), различающихся конкретными значениями свойства *цвет_кузова* (черный, белый, красный и т. п.).

Зануда. Я совсем запутался в терминах *класс*, *экземпляр класса* и *объект*. Из Ваших слов я понял, что класс — это описание какого-то объекта, а экземпляр класса — какая-то конкретизация этого класса, причем экземпляров одного и того же класса может быть сколь угодно много. Но чем

тогда отличается подкласс от класса? Разве подкласс не конкретизирует свой родительский класс? А чем же отличается объект от экземпляра класса?

Профессор. Хорошо, давайте наведем порядок в терминологии. До сих пор мы употребляли слова "объект", "класс", "подкласс" и "экземпляр класса". Под объектом мы понимали некую реальную сущность, либо ее программную модель. Я полагал, что из контекста разговора достаточно ясно, о чем именно идет речь. Тем не менее, теперь я постараюсь более аккуратно обращаться с терминами.

Так вот, мы рассматриваем программное моделирование каких-то реальных сущностей, называемых объектами. Построить модель объекта означает составить ее описание. Такое описание может иметь иерархическую структуру, элементы которой называются классами. Иерархия классов выражает отношение родства или наследования свойств между классами. Не исключено также, что какой-то объект описывается единственным классом, который не связан ни с какими другими классами. Если объект описывается несколькими классами, связанными наследованием свойств, то каждый дочерний класс уточняет (дополняет) описание, представленное его родительским классом. Классы могут содержать как конкретные, так и неконкретные описания свойств. Например, в классе может быть объявлено свойство-переменная *цвет*, значение которой может быть присвоено или нет. Как бы то ни было, классы описывают некоторые объекты подобно справочникам, но сами по себе в программе еще не функционируют. Чтобы применить класс, его необходимо сначала, как говорят, инициализировать. Инициализация класса и есть создание его экземпляра. Она аналогична вызову обычной функции, заданной своим определением. Подобно тому, как вызываемой функции вы можете передать параметры, при создании экземпляра класса можно установить или переустановить значения некоторых или всех его свойств. Вы можете создать несколько экземпляров одного и того же класса, абсолютно одинаковых или отличающихся значениями своих свойств. Поэтому мы можем говорить, что экземпляр класса — это некая конкретизация, но лучше сказать — конкретное воплощение класса в программе, из которой вы можете обращаться к свойствам и вызывать на исполнение методы экземпляра класса.

Зануда. Теперь, кажется, все встало на свои места.

Профессор. Однако я должен заметить, что у программистов все же имеется некий разнобой в терминологии. Так, нередко экземпляры класса называют просто объектами. Кроме того, существуют объектно-ориентированные языки (например, JavaScript), в которых классы не поддерживаются, а новые объекты создаются на основе уже существующих встро-

енных объектов. В таких языках термин "экземпляр класса" заменяют на "экземпляр объекта", а "класс" — на "объект".

Простак. А как создать экземпляр класса или, как еще говорят, объекта?

Профессор. Это делается с помощью специальной функции, называемой конструктором объекта. В различных языках задание конструктора и его вызов определяют по-разному, но основные принципы очень схожи. Более подробно эту тему мы рассмотрим при изучении конкретных языков, а сейчас лишь отмечу, что создаваемый экземпляр объекта можно присвоить обычной переменной, которая будет представлять его в программе. Такая переменная будет иметь особый тип данных `object`.

Простак. Хорошо, а тогда как обратиться к свойствам и методам экземпляров классов или объектов?

Профессор. Создав экземпляр объекта (класса), можно воспользоваться его свойствами — внутренними переменными и методами. Для этого необходимо указать не только требуемое свойство или метод, но и сам объект:

```
объект => свойство;
объект =>метод();
```

Здесь `объект` — это переменная, ссылающаяся на экземпляр объекта (класса) и играющая роль его имени в программе. Таким образом, свойство объекта обозначается не просто именем переменной, а составным именем, включающим в качестве префикса имя объекта. Аналогично образуется выражение для вызова метода. При этом два компонента обозначения разделяют специальным символом. В языках C++ и PHP, например, в качестве такого символа используют стрелку, а в JavaScript — точку:

```
объект.свойство;
объект.метод();
```

Таким образом, к свойствам-переменным и свойствам-методам мы обращаемся почти как к обычным переменным и функциям. Спецификой здесь является то, что перед именем свойства-переменной или вызовом свойства-метода необходимо указать объект, к которым они относятся.

Простак. Хорошо бы все изложенное Вами, Профессор, рассмотреть на примерах решения конкретных задач. Иначе все эти тонкости трудно запомнить.

Профессор. Мы сделаем это, но немного позже, когда перейдем к изучению конкретных языков. А сейчас, завершая обсуждение объектов, я хотел бы отметить, что преимущества объектно-ориентированного

стиля программирования очевидны в случае разработки достаточно крупных проектов. При решении задач относительно небольшой сложности, таких как создание Web-сайтов, обычно обходятся традиционными средствами языка и не оформляют программы в виде множества объектов. Программист должен применять те инструменты, которые он считает удобными в конкретном случае, стараясь избегать неоправданных усложнений. Программы для Web-приложений, выполняемые как на стороне клиента (браузером), так и сервером, обычно невелики по объему и достаточно просты по структуре и логике работы, поэтому при их разработке можно отказаться от объектно-ориентированного подхода. Вместе с тем, сценарии для Web-сайтов пишутся на языках JavaScript и PHP, которые имеют встроенные объекты и предоставляют возможность работы с внешними объектами — объектами браузера и загруженного в него документа. Так, почти все конструкции JavaScript выполнены в виде объектов. Символьные строки, числа, массивы, функции — все это встроенные объекты со своими свойствами и методами. Поэтому даже если вы не собираетесь заводить в программе свои собственные объекты, все равно полезно знать, как они в принципе устроены и как с ними обращаться.

2.5. Операторы и выражения

Профессор. Программы состоят из операторов. Одно из важнейших действий, с которым мы уже встречались, — присваивание значения переменной, например, $x = 5$ или name = "Вася". Оператор присваивания обычно обозначают знаком равенства, а его результат состоит в том, что значение, указанное справа, помещается (записывается) в переменную, указанную слева.

Существуют и другие операторы, например, арифметические операторы сложения (+), вычитания (–) , умножения (*) и т. п. С их помощью составляют арифметические выражения для разнообразных вычислений, например, $x + 5$. Действие такого оператора заключается в том, что к значению переменной x прибавляется 5. Предполагается, что переменная x определена ранее и имеет некоторое числовое значение. Выражение $x + 5$ само получает значение, равное результату выполнения оператора. Говорят, что оператор (выражение с операторами) возвращает значение, полученное в результате его выполнения.

Простак. Кому возвращает?

Профессор. Программе, разумеется.

З а н у д а. А как мы можем воспользоваться полученным значением? Его ведь нужно сначала как-то запомнить.

П р о с т а к. Чтобы что-то запомнить, достаточно просто присвоить это какой-нибудь переменной.

П р о ф е с с о р. Совершенно верно. Мы можем выражение с операторами присвоить переменной: $y = x + 5$.

З а н у д а. Таким образом, Вы составили выражение с двумя операторами: присваивания и сложения. А в каком порядке они выполняются и какое значение будет возвращено данным выражением?

П р о ф е с с о р. Сначала к значению переменной *x* прибавляется число 5, а затем полученный результат помещается в переменную *y*. Это же значение и есть то, которое возвращается выражением целиком.

Теперь о порядке выполнения операторов. В общем случае различные операторы в одном и том же выражении выполняются в порядке, определяемом их приоритетами, а последние указываются в описании языка программирования. В частности, оператор присваивания имеет один из самых низших приоритетов.

З а н у д а. А есть ли операторы с одинаковыми приоритетами? Одинаковы ли приоритеты у операторов, скажем, сложения и вычитания?

П р о ф е с с о р. У многих операторов приоритеты одинаковы. В частности, сложение и вычитание имеют один и тот же приоритет, но меньший, чем у деления и умножения. В выражениях с одинаковыми по приоритету операторами последние вычисляются в порядке упоминания слева направо, если только с помощью круглых скобок не задан иной порядок. Например, значение выражения (2+3)*5 равно 25, а выражения 2+3*5 — 17. Впрочем, это вам должно быть известно еще со школы.

Операторы в программе обычно отделяются друг от друга точкой с запятой и выполняются друг за другом в порядке их следования слева направо и сверху вниз. Однако существует возможность изменить такой линейный порядок. Это обеспечивают так называемые операторы управления потоком вычислений, к которым относятся условные операторы и циклы.

Условный оператор позволяет реализовать структуру условного выражения:

```
если ..., то ..., иначе ...
```

В языках программирования этот оператор имеет следующий синтаксис:

```
if (условие)
 {код, который выполняется, если условие истинно}
else
 {код, который выполняется, если условие ложно}
```

В фигурных скобках располагается блок кода — одно или несколько выражений. Часть этой конструкции, определяемая ключевым словом `else` (иначе), необязательна. В этом случае остается только часть, определенная ключевым словом `if (если)`:

```
if (условие){
 код, который выполняется, если условие истинно
}
```

Условие, указанное в круглых скобках за ключевым словом `if`, есть некоторое выражение логического типа, которое может принимать одно из двух значений: `ИСТИНА` или `ЛОЖЬ`. Обычно это выражение содержит оператор сравнения, например,

```
if (x < 5)
 {y = 10; z = 3}
else
 {y = 0; z = -2}
```

Возможны и другие разновидности оператора условного перехода, но данный вариант базовый, присутствующий практически во всех языках.

Оператор цикла обеспечивает многократное выполнение блока программного кода до тех пор, пока не выполнится некоторое условие. Он имеет несколько вариантов, но базовыми являются операторы, начинающиеся с ключевого слова `for` (для) или `while` (до тех пор, пока…). При создании программ вполне можно обойтись каким-нибудь одним из них. Однако возникают ситуации, в которых один из операторов более удобен или естественен, чем другой. Синтаксис этих операторов и их применение мы подробнее рассмотрим позже, при изучении конкретных языков.

2.6. Специальные термины

Профессор. У программистов имеется профессиональный жаргон, с помощью которого они могут обмениваться мыслями более лаконично.

Простак. Да, я знаю, что вместо слова "программа" часто говорят "софт" или "прога". Кроме того, в Интернете сейчас довольно широко практикуется так называемый "олбанский язык".

Профессор. Вы привели примеры вульгарного жаргона, а "олбанский язык" вообще языком не является: это просто игра с изменением букв в словах так, чтобы последние звучали примерно так же, как и в нормальном языке. Например, "афтар" (автор), "привед" (привет), "йад" (яд) и т. п.

Мы сейчас познакомимся с той частью специальной терминологии, которая общепризнанна и широко применяется программистами при обсуждении языка и программ как в устной, так и письменной речи.

Вот некоторые термины.

- Регистровая зависимость — зависимость выражений языка или их частей (например, имен переменных, функций и т. д.) от того, строчными или прописными буквами они написаны. Так, в регистрозависимых языках переменные `myvar`, `MYVAR` и `myVar` различны, а в регистронезависимых это одна и та же переменная. Например, язык JavaScript полностью регистрозависимый, а PHP — регистрозависимый, но не в полной мере.
- Идентификатор (identifier) — имя переменной, функции, объекта и др.; идентификатор не должен быть ключевым или зарезервированным словом (см. далее). Программист обычно выбирает идентификаторы по своему усмотрению, но в соответствии с определенными правилами. В большинстве случаев идентификаторы состоят из печатных символов (букв, цифр и символа подчеркивания), причем идентификатор не должен начинаться с цифры и содержать символы пробела. Примеры: `x`, `myvar`, `userName`, `user_name`, `var12`.
- Ключевое слово (keyword) — слово, являющееся частью языка. Ключевые слова недопустимы в качестве идентификаторов. Примеры: `function`, `if`, `while`, `var`.
- Зарезервированное слово (reserved word) — слово, которое не рекомендуется использовать в качестве идентификатора. Зарезервированные слова могут оказаться ключевыми в более поздних версиях языка.
- Оператор (statement) — команда, предписывающая компьютеру выполнить некоторое действие; операторы обычно приводят к изменению состояния окружения (например, переменной или определения). Примеры: `x = x + 5`, `function myfunc(x,y){return x*y}`.
- Литерал (literal) — значение, содержащееся непосредственно в тексте программы. Примеры: `5`, `"Привет всем!"`, `[3,7,9,25]`, `false`.
- Лексема (token) — наименьшая и неделимая единица языка. Примеры: все идентификаторы, ключевые слова, а также литералы типа `5.2` и `"Привет всем!"`.
- Знак операции (operator) — лексемы, представляющие собой встроенные операторы языка. Примеры: `=`, `+`, `-`, `*`, `/` (оператор присваивания и арифметические операторы). Операции нередко называют операторами.
- Выражение (expression) — группа лексем (обычно литералов и идентификаторов) в сочетании со знаками операций, для которых можно вычислить

значение. Примеры: `3.141`, `"Привет!"`, `(2 + x)*5`, `myfunc()`, `x = y + myfunc()`.

Простак. Все эти определения нелегко запомнить.

Профессор. Ничего, со временем, при чтении литературы по программированию и обсуждении программ с коллегами и друзьями вы научитесь правильно употреблять перечисленные термины. Однако можно вполне обойтись и только такими понятиями, как идентификатор (имя), ключевое слово, значение и оператор (операция).

2.7. Резюме и напутствие

Профессор. Итак, мы рассмотрели наиболее важные понятия, присущие большинству современных языков программирования: переменные, типы данных, массивы, функции, классы и объекты, операторы и выражения. Давайте вспомним их краткие определения.

- Переменные — это контейнеры для хранения данных, называемых значениями переменных; переменная имеет имя или идентификатор, чтобы отличить ее от других переменных и обращаться к ней по имени для получения содержащегося в ней значения.
- Значения — могут быть различных типов: числового, символьного (строкового), логического и др. Тип — множество допустимых значений — может быть задан явным перечислением элементов либо посредством формулировки некоторого ограничения или правила. Так например, логический тип данных есть множество из двух элементов (true и false), интерпретируемых как "истина" и "ложь" соответственно. Множество чисел бесконечно и не может быть задано перечислением своих элементов. Поэтому числовой тип определяют неким правилом формирования относящихся к нему элементов: последовательность цифр, перед которой может быть указан знак, а разделителем целой и дробной частей служит точка.
- Переменная — принимает значение или, другими словами, в переменную записывается значение с помощью оператора присваивания. Доступ к значению переменной осуществляют по ее имени (идентификатору).
- Массив — упорядоченный набор (последовательность данных). Можно сказать, что конкретный массив — это значение составного типа, который также называется массивом.
- Функция — это поименованный блок программного кода (тело функции), который можно вызвать для выполнения из основной (внешней по отношению к коду функции) программы.

- Класс — описание объекта в виде набора его свойств, которые аналогичны обычным переменным и функциям. Часто свойства-переменные называют просто свойствами, а свойства-функции — методами. С формальной точки зрения класс — это контейнер для хранения данных о свойствах объекта. С помощью специальной функции (конструктора), в программе создается экземпляр класса. Свойства и методы экземпляра класса могут быть доступными из внешней программы.
- Операторы — предназначены для задания команд, которые должен выполнить компьютер. Выражения содержат один или несколько операторов. Наиболее часто встречается оператор присваивания значения переменной. Для управления ходом вычислений служат условные операторы и циклы. Если бы не эти специальные операторы, программа выполнялась только в одном направлении — сверху вниз, в порядке упоминания выражений в листинге (исходном тексте).

Далее мы рассмотрим эти и другие понятия программирования на примере конкретных языков — JavaScript и PHP. Вы увидите, что между ними очень много общего, что позволит вам освоить их одновременно. Тем не менее, между ними есть и существенные различия, которыми нельзя пренебречь. Если вам удастся без особых проблем изучить оба этих языка, то, возможно, вы обретете способность к изучению и других (более сложных) языков программирования, таких как C, Java или Pascal. Базовые понятия, с которыми вы уже познакомились в общих чертах при изучении конкретных языков или при чтении специальной литературы, могут воплощаться по-разному. Одно дело теория, другое — конкретная реализация в том или ином программном продукте, на той или иной платформе. Я лишь предупреждаю вас, чтобы вы не паниковали, обнаружив что-то такое, что не работает у вас. В действительности все программисты преодолевают, так или иначе, возникшие трудности, связанные с особенностями операционных систем, исполнительных систем и приложений. Если вы, пользователи компьютера или просто читатели, столкнулись с чем-то, что не работает на вашем компьютере, хотя вы и сделали все, что указывалось в рекомендациях, то, скорее всего, дело в мелочах, которые разрешаются легко и просто. Но иногда возможны и более существенные осложнения, устранение которых лучше поручить специалисту-диагносту. Врачи и программисты знают, как по поверхностным (клиническим) симптомам можно судить о далеко идущих внутрь организма причинах неполадок, а также о том, какое, хотя бы в общих чертах, назначить лечение (устранение неполадок или ошибок). Да, все это не просто, но вполне преодолимо со временем, было бы желание. Но для начала важно соблюсти некоторые наиболее общие условия.

Освоение языка совершенно не эффективно без практического программирования, как невозможно научиться говорить на иностранном языке вне естественной среды общения. Для вящей убедительности моего тезиса вспомните, что ребенок сначала учится говорить на примерах от своих родителей, а только потом знакомится с правилами языка, которые ему надлежит применять везде и всегда, если только он не хочет поселиться в джунглях и там продолжить свой род. Поэтому необходимо основательно подготовиться к практическому выполнению предлагаемых далее примеров.

Применительно к JavaScript и PHP это совсем не сложно. Программные коды можно писать в обычном текстовом редакторе, например, блокноте Windows.

Чтобы просматривать и отлаживать *клиентские сценарии* (т. е. сценарии, исполняемые Web-браузером) на JavaScript, в вашей системе ничего не нужно устанавливать предварительно, коль скоро у вас уже имеется современный браузер, например, Microsoft Internet Explorer, Mozilla или Opera. Однако следует учитывать, что результат выполнения сценариев может ощутимо зависеть от конкретного типа браузера и его версии.

Для создания и отладки серверных сценариев на языке PHP потребуется программное обеспечение Web-сервера, например, Microsoft Internet Information Services (IIS — Информационные службы Интернета) или Apache, а также установка на сервере модуля PHP. Разумеется, можно воспользоваться удаленным Web-сервером с уже установленным модулем PHP, но изучение языка и отладка сценариев удобнее на локальном компьютере. Установить и настроить Web-сервер с модулем PHP на локальном компьютере может обычный пользователь, а не только системный администратор. Как это сделать, рассказывается в *приложении 2*.

Глава 3

Основы языка JavaScript

В данной главе мы рассмотрим основные понятия и синтаксис языка JavaScript. На этом языке обычно пишут так называемые клиентские сценарии, выполняемые Web-браузером на компьютере пользователя. Кроме того, сценарии на JavaScript пригодны для выполнения Web-сервером, а также встроенным интерпретатором Windows Scripting Host на локальном компьютере.

3.1. Немного истории о версиях и стандартах

Современное состояние чего-либо, и, в частности, языков и технологий программирования, легче понять в исторической ретроспективе. Сначала HTML-документы содержали лишь команды (теги) форматирования текста, ссылки на другие документы и вставки графических изображений. Нередко мы сначала ограничиваем себя слишком скромными целями, то ли от неуверенности в собственных силах, то ли от непонимания перспективы. При этом мы говорим: хорошо бы реализовать хотя бы это немногое, а там посмотрим. И в самом деле, на первых порах этого немногого оказалось вполне достаточно для публикации пользователями Всемирной сети своих творений, содержащих самое главное — тексты с иллюстрациями и ссылки на другие документы того же рода. Таким образом, возможность практически неограниченного распространения чего бы то ни было была реализована.

Вспомните недавнее прошлое, когда была лишь одна возможность публикации, — напечатать сначала на механической пишущей машинке, а затем попытаться разместить свой текст на бумаге, выпускаемой каким-либо официальным издательством. На издание небольшой статьи уходило несколько месяцев. Однако, как почти всегда и бывает, освоенное не только порождает удовлетворенность или, напротив, разочарование достигнутым состоянием, но и вызывает новые безудержные желания (вспомните сказку о рыбаке

и золотой рыбке). HTML (язык разметки гипертекста) был прост и доступен даже тем, кто не имел никакого опыта программирования и, тем более, создания программных проектов, т. е. программ, предназначенных для более или менее широкой потребительской аудитории. Простота и доступность HTML сыграли свою главную роль — обеспечить доступ к всемирным информационным ресурсам всех желающих. Лиха беда начало!

Вскоре выяснилось, что добавление к уже существующему еще чего-то простого и в то же время мощного может раскрыть ворота в мир почти неограниченного могущества. Где же была предполагаемая точка атаки и последующей экспансии? Первоначальные HTML-документы содержали лишь команды представления на экране некоторых информационных ресурсов первостепенной важности: текстов, графики, а затем звуков и видео. Тогда можно было лишь надлежащим образом разместить информацию на страницах, но управлять ею или обрабатывать динамически (т. е. во время экспозиции) было невозможно. Решить эти и многие другие задачи можно было с помощью языков программирования, отличных от HTML и похожих на уже известные, посредством которых пишут программы для обычных компьютерных приложений. Таким образом, необходимо было обеспечить вставку в HTML-код фрагментов на другом языке программирования.

В 1995 г. компания Netscape впервые предложила язык LiveScript для своего браузера Netscape Navigator 2.0. Главное назначение этого языка было весьма скромным — проверка и обработка данных, введенных пользователем в форму на Web-странице, чтобы исключить передачу на сервер заведомо неправильных или пустых значений. Вскоре он был переименован: префикс Live был заменен на Java. Видимо, это произошло по коммерческим соображениям. По крайней мере, JavaScript не является какой бы то ни было упрощенной версией известного мощного, но совсем другого языка Java. Итак, Netscape ввела в обиход язык программирования для своего браузера. Однако в это время уже существовали и развивались браузеры других конкурирующих фирм. Естественно, они не могли не сделать чего-то подобного.

Вскоре за Netscape корпорация Microsoft представила свою редакцию JavaScript под немного отличающимся названием JScript для своего браузера Internet Explorer 3.0. Название JScript было выбрано, видимо, чтобы избежать проблем с авторскими правами и лицензированием. Тем не менее, идея родилась и воплотилась. Так началось развитие скриптового языка (т. е. языка сценариев), который теперь называют JavaScript, а название JScript употребляют, когда хотят подчеркнуть, что имеется в виду язык JavaScript с множеством фирменных расширений, поддерживаемых преимущественно браузерами Microsoft Internet Explorer. Microsoft немало потрудилась, чтобы обеспечить программисту как можно больше возможностей и удобств программирования.

Поэтому браузеры других фирм были вынуждены поддерживать в порядке исключения кое-что из созданного Microsoft. И в самом деле, нельзя же полностью игнорировать массу программного кода, написанного исключительно для чрезвычайно широко распространенного браузера Microsoft Internet Explorer. Заметим попутно, что браузер Opera стал поддерживать JavaScript, начиная с версии 3.0.

Со временем все основные производители браузеров создали собственные расширения языка JavaScript, позволяющие решать существенно более сложные и интересные задачи, чем первоначально поставленные. Этапы развития JavaScript фиксировались в виде его версий, которые были тесно связаны с типами и версиями браузеров. Действительно, любой язык жив, пока существует система, способная его интерпретировать. Такой системой для языков программирования является компилятор или интерпретатор. Сейчас мы не будем выяснять, в чем разница между ними. Достаточно лишь указать, что скриптовые языки воспринимаются интерпретаторами, а интерпретатор JavaScript встроен в Web-браузер, который в принципе может выполнять скрипты (сценарии). Современные браузеры это могут делать, но пока с некоторыми различиями.

В табл. 3.1 приведена информация о поддержке JavaScript некоторыми наиболее популярными браузерами.

Таблица 3.1. *Поддержка JavaScript браузерами*

Браузеры	**Версии JavaScript**
Netscape 2.*x*	JavaScript 1.0
Netscape 3.*x*	JavaScript 1.1
Netscape 4.0 — 4.05	JavaScript 1.2
Netscape 4.06 — 4.08, 4.5*x*, 4.6*x*, 4.7*x*	JavaScript 1.3
Netscape 6.*x*, 7.*x*	JavaScript 1.5
Mozilla (варианты)	JavaScript 1.5
Internet Explorer 3.0	JScript 1.0
Internet Explorer 4.0	JScript 3.0
Internet Explorer 5.0	JScript 5.0
Internet Explorer 5.5	JScript 5.5
Internet Explorer 6.0, 7.0	JScript 5.6

Нетрудно заметить, что версии и редакции (клоны) языка JavaScript плотно связаны с поддерживающими их браузерами различных фирм. Браузеры

содержали в себе встроенные интерпретаторы этого языка, а их разработчики могли видоизменять эти языки и расширять их как угодно. Однако произвол такого творчества, разумеется, ограничен. Всемирная паутина по своему замыслу должна предоставлять пользователю информационные ресурсы инвариантно относительно браузеров. Поэтому творцы языка должны были, так или иначе, стремиться к некоторому "общему знаменателю". Тем не менее, разночтение все же возникло и породило весьма запутанную проблему совместимости программных кодов (сценариев), написанных для различных браузеров.

Разумеется, синтаксис и основные объекты языка остаются незыблемыми, поддерживаемыми всеми современными браузерами, которые в принципе способны выполнять сценарии для Web-страниц. Однако в языке предусмотрены еще и объекты браузера и документа, состав и/или реализация которых варьируют от версии к версии, а также зависят от типа браузера. Именно эти объекты и приводят к несовместимости браузеров различных производителей.

Сначала Netscape и Microsoft использовали одинаковые стратегии развития своих браузеров. А именно в конкурентной борьбе они создавали свои фирменные расширения первоначальной версии языка JavaScript, которые, к сожалению, не были совместимыми. На этом пути были созданы браузеры Netscape Navigator 4.*x* и Microsoft Internet Explorer 4.*x*. Между JavaScript от Netscape и Jscript от Microsoft было много общего, но имелись и существенные различия, хотя и Netscape, и Microsoft вместе с другими крупнейшими производителями программного обеспечения участвовали в работе организации World Wide Web Consortium (W3C, Консорциум Всемирной паутины), цель которой — выработка рекомендаций и стандартов для Web. Казалось бы, они вполне могли бы договориться об "общем знаменателе", но этого не произошло.

В дальнейшем стратегии Netscape и Microsoft существенно разошлись. Компания Netscape в своих браузерах версий 6.*x* и 7.*x* отказалась от создания, поддержки в полной мере уже существующих и наращивания фирменных дополнительных возможностей JavaScript в пользу приверженности стандартам, при этом она также отказалась от принципа сохранения обратной совместимости своих новых продуктов. Иначе говоря, в новых браузерах Netscape не поддерживались в полной мере сценарии, написанные для предыдущих версий. И это в условиях, когда уже существовало огромное количество кода, написанного для прежних версий браузеров Netscape! В результате, что нетрудно было предвидеть, линейка браузеров Netscape Navigator 4.*x* стала терять популярность среди пользователей и в настоящее время почти исчезла. Вместе с тем, в 1998 г. Netscape представила исходный код браузера Mozilla. Это был важный этап реализации новой стратегии, часть которой

уже воплотилась в браузерах Netscape 6*x*, 7*x*. Название "Mozilla" сначала использовалось в качестве внутреннего имени программного кода браузеров компании Netscape. В настоящее время оно относится к браузеру как продукту, платформе и организации. Mozilla — это одновременно наименование и кроссплатформенного браузера с открытым кодом, и платформы — каркаса, на базе которого разработаны как сам браузер, так и другие кроссплатформенные приложения, и организации — независимой группы разработчиков, занимающейся поддержкой и усовершенствованием данных программных продуктов. Открытый программный код Mozilla теперь может быть использован в качестве основы (с соблюдением определенных лицензионных соглашений) для создания любых собственных браузеров теми, кто возьмется за это дело. Так например, Mozilla и Mozilla Firefox — конкретные браузеры, созданные на базе одного и того же программного кода Mozilla, но являющиеся разными продуктами, поскольку имеют некоторые отличия.

Microsoft, напротив, продолжила создавать расширения языка, дающие опытному разработчику большие возможности, а неопытному — автоматическое исправление синтаксических ошибок и отклонений от соблюдения стандартов. Так что для Internet Explorer можно позволить себе некоторую небрежность в написании программных кодов: браузер многое исправит сам. При этом, что очень важно, Microsoft сохранила для своих браузеров Internet Explorer 5.*x*, 6.0 и 7.0 обратную совместимость и в то же время обеспечила поддержку современных стандартов, хотя и не на 100%. Нередко можно услышать, что хуже всех соответствует стандартам браузер Microsoft Internet Explorer. Это, на мой взгляд, неверно. Internet Explorer поддерживает стандарты примерно так же, как и его современные конкуренты, но в отличие от них он сохраняет совместимость со своими более ранними версиями, а также поддерживает множество других, мощных и полезных возможностей, которые в стандарты не вписываются. Многие разработчики используют расширения, которые функционируют неверно или совсем не работают в других браузерах. Однако это совсем не доказывает, что Internet Explorer далек от стандартов, к которым якобы близки другие браузеры. Да, он выполняет коды, написанные вне стандартов, но это его дополнительные возможности, которые никому не навязываются насильно. Если вы хотите создавать Web-документы, инвариантные относительно всех современных браузеров, то пишите тексты программ с соблюдением стандартов, не прибегая к дополнительным фирменным возможностям браузеров. А при разработке приложений исключительно для пользователей Internet Explorer (например, для интрасети) ничто не мешает вам применить его фирменные расширения языка JScript.

Теперь немного о собственно стандартах. Рано или поздно, во избежание кризиса "вавилонского столпотворения", причиной которого было, как хорошо

известно, разноязычие строителей, приходится задуматься о соответствии пестрого разнообразия языков каким-нибудь общим рамкам.

Чтобы создать основу для обеспечения межбраузерной совместимости в отношении языка JavaScript, была выработана стандартная спецификация ECMAScript (ECMA-262) для языка сценариев, встраиваемых в Web-страницы. В настоящее время известны три редакции (edition) данного стандарта. Все современные браузеры теперь объявляют полную поддержку стандарта ECMA-262 Edition 3, но некоторые из них, как уже отмечалось ранее, имеют и дополнительные возможности (расширения). Больше всех фирменных (нестандартных) возможностей предоставляет, пожалуй, браузер Microsoft Internet Explorer 5.*x*, но и другие делают что-то в этом направлении. Так что разработчикам Web-приложений следует быть внимательным и готовым к тестированию своей продукции на нескольких наиболее популярных браузерах. Соответствие стандартам версий JavaScript (Netscape) и JScript (Microsoft) представлено в табл. 3.2.

Таблица 3.2. *Соответствие версий языка стандартам*

Netscape	Microsoft	Стандарт	Примечания
JavaScript 1.0 (Netscape 2.0), JavaScript 1.2 (Netscape 4.0 — 4.05)	JScript 1.0 (Internet Explorer 3.0)	Приблизительное соответствие ECMA-262 Edition 1	Много исключений, имеются дополнения
JavaScript 1.3 (Netscape 4.06)	JScript 3.0 (Internet Explorer 4.0)	Строгое соответствие ECMA-262 Edition 1	Имеются дополнения
JavaScript 1.5 (Netscape 6*x*, 7*x*, Mozilla)	JScript 5.5, 5.6 (Internet Explorer 5.5, 5.6)	Строгое соответствие ECMA-262 Edition 3	То же

В данной главе мы рассмотрим те конструкции языка JavaScript, которые инвариантны относительно браузеров — базовые определения и встроенные объекты, избегая, где только это возможно, обращения к объектам браузера и загруженного в него документа.

3.2. Общая характеристика языка

В данном разделе приводятся наиболее общие сведения о JavaScript, которые могут оказаться полезными для тех, кто уже имеет опыт работы с каким-нибудь другим языком. Новички при первом чтении книги могут опустить этот материал.

Отличительные особенности JavaScript:

- Исходный программный код интерпретируется.
- Операторы разделяются точкой с запятой. Вместе с тем, за завершенным (т. е. полным) оператором, расположенным в одной строке, разделитель можно не указывать: в этом случае переход на другую строку работает так же, как и точка с запятой.
- Язык регистрозависимый. Новички об этом нередко забывают, возможно, потому, что HTML, с которым JavaScript обычно используется совместно, не зависит от регистра (имена тегов и атрибутов HTML можно писать как строчными, так и прописными буквами).
- Внутристрочный комментарий (неинтерпретируемый, игнорируемый текст) выделяется символом //, а многострочный — парой символов /* и */.
- К данным применяется слабый (динамический) контроль типов. В операторах с разнотипными данными последние автоматически приводятся к требуемому типу. Типы данных могут быть примитивными и составными. Примитивные типы содержат простые однородные значения, такие данные можно передавать функциям в качестве параметров по значению, а не по ссылке. Составные типы содержат разнородные данные (в том числе и составные), их передают функциям только по ссылке.
- Язык объектно-ориентированный, однако он основан на прототипах, а не на классах. Имеются четыре типа объектов: встроенные объекты, объекты браузера (интерпретирующей системы), объекты документа и объекты пользователя (программиста).
- Ввод-вывод в основном ограничен взаимодействием с документами и пользователями. По умолчанию предполагается, что доступ к локальной файловой системе запрещен. Однако браузеры могут предоставлять специальные объекты, с помощью которых обеспечивается работа с файловой системой пользователя, хотя и с выдачей предупреждений об опасности выполнения файловых операций.

3.3. Как создавать и отлаживать сценарии на JavaScript

Сценарии (программы) на языке JavaScript взаимодействуют с объектами HTML-документа (Web-страницы). Для этого они собственно и создаются. Чтобы реализовать такое взаимодействие, их необходимо, прежде всего, внедрить (вставить) в текущий HTML-документ.

3.3.1. Вставка сценариев в HTML-документ

Сценарии можно вставить в HTML-документ несколькими способами, но чаще всего их размещают внутри контейнерного тега <SCRIPT>, т. е. между дескрипторами <SCRIPT> и </SCRIPT>. Сам контейнер <SCRIPT> находится в HTML-документе. Программный код сценария можно писать непосредственно в HTML-документе, а также в отдельных текстовых файлах, которые вызываются из него. Проще всего размещать сценарий непосредственно в HTML-документе, именно так обычно и поступают. Мы рассмотрим оба варианта.

Встречая тег <SCRIPT> в загруженном HTML-документе, браузер "понимает", что за ним начинается код сценария. Заключительный тег </SCRIPT> указывает браузеру, что код сценария закончился. Все, что находится вне этих тегов, браузер воспринимает как HTML-код. Контейнер <SCRIPT> может располагаться в любом месте HTML-документа, и даже не один раз. От его расположения в теле HTML-документа иногда может зависеть функционирование всей Web-страницы, но об этом будет сказано позже.

Контейнерный тег <SCRIPT>, объемлющий код сценария, может содержать следующие атрибуты:

- LANGUAGE — указывает язык сценария; возможные значения:
  - "JavaScript", "JScript". Однако второе значение правильно воспринимается только браузером Internet Explorer. Для сценариев, рассчитанных на различные браузеры, следует указывать LANGUAGE = "JavaScript" или "javascript";
  - "VBScript", "VBS". Эти значения воспринимаются только браузером Internet Explorer.

Пример

```
<SCRIPT TYPE = "javascript">
 ...  // код сценария
</SCRIPT>
```

ЗАМЕЧАНИЕ

Если атрибут LANGUAGE не указан, то Internet Explorer подразумевает JScript. Вместе с тем, любая опечатка в значении атрибута LANGUAGE может привести к игнорированию браузером всего содержимого тега <SCRIPT>.

Атрибут LANGUAGE не соответствует стандарту W3C, хотя он широко используется на практике. Вместо него рекомендуется применять атрибут TYPE (см. далее).

- TYPE – указывает MIME-тип используемого языка. Для JavaScript следует задавать значение "text/javascript".

Пример

```
<SCRIPT TYPE = "text/javascript">
 ...  // код сценария
</SCRIPT>
```

Замечание

В теге <SCRIPT> возможно указание одновременно и атрибута LANGUAGE, и TYPE. Однако настоятельно рекомендуется указывать атрибут TYPE.

- SRC — указывает имя или URL-адрес текстового файла, содержащего код сценария. Этот атрибут необходим в том случае, если сценарий расположен не непосредственно в HTML-документе, а в отдельном файле. Расширение файла со сценарием может быть каким угодно, но обычно используют js:

```
<SCRIPT LANGUAGE = "JScript" SRC = "myscripts.js"></SCRIPT>.
```

Замечание

Если сценарий располагается в отдельном файле, то в нем, разумеется, теги <SCRIPT> и </SCRIPT> не пишут.

Сценарий, загруженный из внешнего файла, можно представить себе просто как его вставку в HTML-документ.

Внимание!

В примерах, приводимых в настоящей книге, мы ради краткости не будем указывать атрибуты LANGUAGE и TYPE.

Старые браузеры, появившиеся раньше JavaScript, игнорируют теги <SCRIPT> и </SCRIPT>, а все, что находится между ними, интерпретируют как текстовое содержимое HTML-документа. Результат при этом может быть самым неожиданным. Чтобы уменьшить вероятность отображения кода сценария в окне старого браузера, следует заключить его в дескрипторы комментария <!-- и -->. Новые браузеры, поддерживающие сценарии, будут опускать эти символы, выполняя код сценария, а старые браузеры (не понимающие сценарии), наоборот, проигнорируют код сценария. Пример такого подхода приведен в листинге 3.1.

Листинг 3.1

```
<SCRIPT LANGUAGE = "JavaScript">
 <!--
  // код сценария
 //-->
</SCRIPT>
```

Обратите внимание на заключительные символы тега комментария, перед которыми стоят две косые черты. При их отсутствии браузер будет пытаться интерпретировать символы --> как окончание комментария HTML, а при наличии он просто их проигнорирует.

В браузерах, потенциально поддерживающих сценарии, эту функцию можно отключить. Кроме того, существуют браузеры, принципиально не поддерживающие сценарии. В данной ситуации желательно хотя бы вывести сообщение о том, что произошло. С этой целью в HTML-документе используют контейнерный тег <NOSCRIPT>, в котором размещают, например, текст предупреждающего сообщения. Все браузеры, поддерживающие сценарии, проигнорируют содержимое тегов <NOSCRIPT> кроме тех случаев, когда поддержка сценариев отключена. Браузеры, принципиально не поддерживающие сценарии, напротив, содержимое тега <SCRIPT> (если оно заключено в теги <!-- и //-->) опустят, а тега <NOSCRIPT> — отобразят. Пример приведен в листинге 3.2.

Листинг 3.2

```
<HTML>
<HEAD>
<TITLE>Пример применения тега NOSCRIPT</TITLE>
</HEAD>
<SCRIPT TYPE="text/JavaScript">
<!--
   alert("Поддержка JavaScript включена");
//-->
</SCRIPT>
<NOSCRIPT>
   <H2>Ваш браузер не поддерживает JavaScript
   или его поддержка отключена</H2>
</NOSCRIPT>
</HTML>
```

Здесь был использован метод `alert(сообщение)` для вывода сообщений в диалоговом окне. Подробнее об этом и других методах вывода сообщений будет рассказано в *разд. 3.4*.

> ***Внимание!***
>
> В примерах, приводимых в настоящей книге, мы не будем указывать символы комментария для адаптации к старым браузерам из соображений экономии времени и места.

Как уже отмечалось, в отдельных файлах обычно размещают библиотеки функций (определения функций), а также сценарии, использующиеся в нескольких HTML-документах одного или нескольких сайтов. Пример сценария из нескольких разделов иллюстрирует листинг 3.3.

Листинг 3.3

```
<HTML>
...
 <SCRIPT>
  function myfunc(){
   ...
  }
 </SCRIPT>
 <SCRIPT SRC = "myprog1.js"></SCRIPT>
 <SCRIPT SRC = "myprog2.js"></SCRIPT>
...
</HTML>
```

Здесь в HTML-документе расположены три раздела (секции) сценария, два из которых загружаются из отдельных файлов. В первом разделе сценария дано определение некоторой функции.

Сценарий можно также писать в виде строки операторов, разделенных точкой с запятой. Такая строка, взятая в кавычки, служит в качестве значения атрибута-события: `<IMG SRC="mypicture.gif" ONCLICK = "alert('Привет!')"`.

Вызовы функций и их определения могут следовать в произвольном порядке, только если они расположены в одном и том же контейнере `<SCRIPT>`. Если вы размещаете их в разных контейнерах `<SCRIPT>`, то необходимо, чтобы определение функции предшествовало ее вызову. Аналогично, если определения функций находятся в отдельном файле, то его необходимо загрузить

в HTML-документ раньше вызовов этих функций. В листинге 3.4 приведены примеры правильного и неправильного расположения сценариев в HTML-документе.

Листинг 3.4

Правильно

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

Неправильно

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

При попытке загрузить и выполнить неправильный вариант HTML-кода появится диалоговое окно с сообщением "Ошибка: Предполагается наличие объекта". Браузер интерпретирует теги HTML последовательно. Так, встретив выражение вызова функции `myfunc()` в одном контейнере `<SCRIPT>`, браузер еще не имеет в памяти определения этого объекта (функции), расположенного в другом контейнере `<SCRIPT>`. Если же определение функции и ее вызов находятся в одном и том же контейнере `<SCRIPT>`, то его содержимое сначала загружается в память, а затем анализируется. Когда интерпретатор встречает вызов функции, то он ищет ее определение в памяти и в случае удачи выполняет код этой функции. Следующие два варианта, приведенные в листинге 3.5, являются правильными, потому что определение функции и ее вызов находятся в одной секции сценария (в одном контейнере `<SCRIPT>`).

Если необходимо, чтобы сценарий оказался в браузере прежде, чем загрузятся элементы HTML-документа, то его следует расположить в верхней части HTML-кода, а еще лучше в контейнере `<HEAD>` (в заголовке документа).

Листинг 3.5

Вариант 1

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

Вариант 2

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

Если требуется, чтобы сценарий загрузился после всех элементов HTML-документа, то возможны следующие два варианта.

1. Расположить сценарий в конце HTML-документа.
2. Указать атрибут-событие `ONLOAD` в контейнерном теге `<BODY>`, который задает основную часть HTML-документа. В данном случае значением атрибута `ONLOAD` обычно является строка, содержащая имя функции. Определение этой функции, как правило, содержится в контейнере `<SCRIPT>`, размещенном в контейнере заголовка HTML-документа `<HEAD>`. Обратите внимание, что при использовании атрибута-события `ONLOAD` в теге `<BODY>` обработчик этого события выполняется по завершении загрузки всех элементов, определенных в контейнере `<BODY>`, а не в процессе их загрузки. Пример такого подхода иллюстрирует листинг 3.6.

Листинг 3.6

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

```
  </HEAD>
  ...
  <BODY ONLOAD =" myfunc()">
   <! Теги тела документа >
  </BODY>
  ...
 </HTML>
```

3.3.2. Подготовка, запуск и отладка сценариев

В данной книге мы не будем рассматривать мощные и удобные для пользователя редакторы визуальной разработки Web-страниц и сайтов, такие как Dreamweaver, FrontPage др. Как HTML-документы, так и сценарии для них можно разрабатывать с помощью обыкновенного текстового редактора, например, Блокнота Windows.

Откройте какой-нибудь текстовый редактор (например, Блокнот Windows) и создайте в нем заготовку файла, который вы будете потом редактировать, вводя экспериментальные выражения или даже целые программы. Введите в рабочее поле редактора текст, приведенный в листинге 3.7.

Листинг 3.7

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

 </SCRIPT>
</HTML>
```

По существу, это HTML-документ, определяющий пустую Web-страницу. Сохраните файл на диске, например, под именем proba.htm. Расширение имени файла должно быть htm или html, поскольку мы хотим, чтобы он исполнялся в Web-браузере. В этом файле мы написали теги языка HTML. Выражения на языке JavaScript следует записывать между тегами `<SCRIPT>` и `</SCRIPT>`. При изучении данной главы записывайте в каждой строке не более одного выражения на языке JavaScript. Заканчивайте строку нажатием клавиши <Enter>, чтобы перейти к новой строке. Если в одной строке потребуется написать несколько выражений, то их следует отделять друг от друга точкой с запятой. В листинге 3.8 приведен пример простейшей программы.

Листинг 3.8. Пример программы с использованием сценария

```
<HTML>
 <HEAD><TITLE>Примеры</TITLE></HEAD>
 <SCRIPT>
  x = 5;
  y = x + 4;
  alert(y);
 </SCRIPT>
</HTML>
```

При выполнении учебных примеров обычно требуется отобразить на экране окончательные или промежуточные данные. Это можно реализовать методом `alert()`, указав в круглых скобках то, что требуется вывести на экран. В приведенном листинге 3.8 метод `alert(y)` выведет на экран диалоговое окно, в котором появится значение переменной `y` (в данном случае — число 9), как показано на рис. 3.1. Если не вставлять строку `alert(y)`, то программа выполняется без отображения результата.

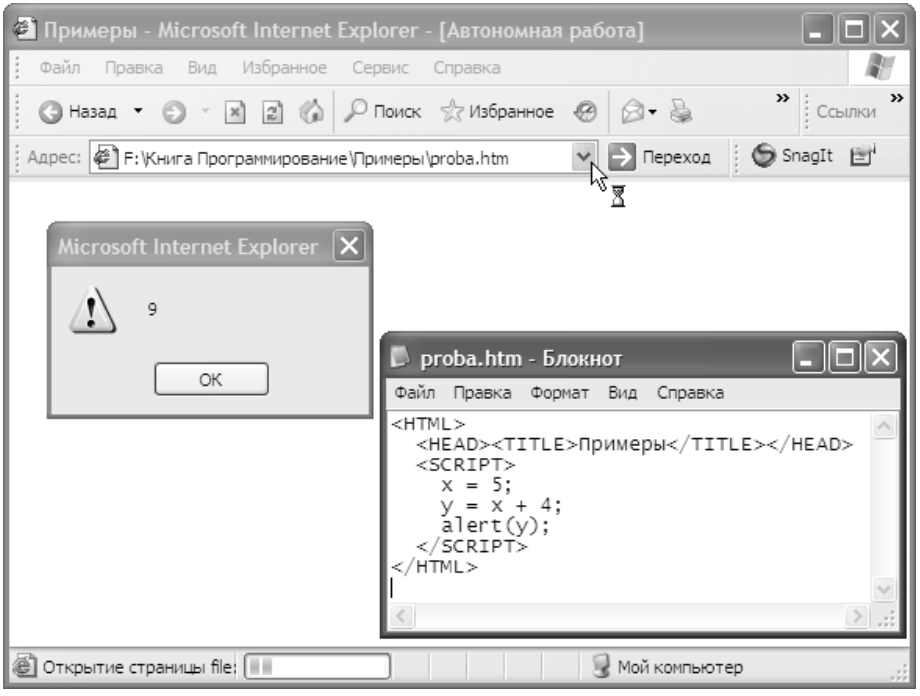

Рис. 3.1. HTML-код в текстовом редакторе и результат его выполнения браузером

Если вы хотите выводить результаты вычислений не в диалоговом окне, а непосредственно в окне Web-браузера, то вместо `alert(y)` напишите следующее выражение:

```
document.write(y).
```

Чтобы открыть файл (в нашем случае это proba.htm) в Web-браузере и выполнить содержащуюся в нем программу, достаточно просто дважды щелкнуть левой кнопкой мыши на его имени в окне Проводника Windows. Для изменения исходного кода снова откройте этот файл в текстовом редакторе, внесите требуемые коррективы и сохраните файл на диске. Чтобы Web-браузер смог отобразить измененный файл, выполните команду меню браузера **Вид | Обновить**. Таким образом, переходя от текстового редактора к браузеру и обратно, можно разрабатывать и отлаживать свои программы.

Если сценарий содержит ошибки, то реакция браузера может быть самой различной в зависимости от ситуации. Так, выполнение сценария может продолжиться после обнаружения ошибочного выражения, а может и приостановиться. Последнее, в частности, происходит при обращении к неопределенному объекту. Обращение же к переменной, которой ранее не было присвоено значение, или к отсутствующему у объекта свойству, а также деление числа на ноль не нарушают синтаксис языка и браузером не обнаруживаются. Это так называемые семантические ошибки, которые бывает трудно выявить.

Браузеры имеют средства отладки, представляющие собой окна, в которые выводятся сообщения об ошибках — их общую характеристику и место в исходном коде сценария. При возникновении ошибки в нижнем левом углу браузера Internet Explorer появляется значок с изображением треугольника с восклицательным знаком. Двойным щелчком на этой пиктограмме открывается окно с сообщением об ошибке. В Mozilla Firefox имеется Консоль ошибок, а в Opera — Консоль JavaScript, которые открываются с помощью соответствующих меню браузера.

Кроме встроенных средств отладки программисты часто используют вывод на экран сообщений о промежуточных состояниях процесса выполнения сценария: значения переменных, характеристики достигнутой точки процесса и т. п. Такого рода сообщения можно генерировать с помощью метода `alert()`, указывая в круглых скобках собственно сообщение. Данный подход оказывается особенно полезным для выявления семантических ошибок, которые интерпретатором не обнаруживаются. Листинг 3.9 поясняет эту возможность.

Листинг 3.9

```
<HTML>
<HEAD>
<TITLE>Пример ошибки</TITLE>
</HEAD>
<SCRIPT>
   x=5;
   alert("Вычисляем myfunc(x)");
   y= myfunc(x);
   alert("Результат "+y);
   x=x+y;
</SCRIPT>
</HTML>
```

В данном примере вызывается функция `myfunc(x)`, которая не определена, что является ошибкой, приводящей к остановке выполнения сценария. Для отладочных целей в код сценария вставлены два вывода сообщений: до и после вызова `myfunc(x)`. Если при выполнении сценария появилось диалоговое окно с сообщением `"Вычисляем myfunc(x)"`, значит, все предыдущие операторы были безошибочными. В точке вызова неопределенной функции `myfunc(x)` выполнение сценария прерывается и, следовательно, сообщение `"Результат …"` не появится, что и сигнализирует нам о возникновении ошибки и ее месте. В данном случае браузер также создаст сообщение об ошибке, которое можно посмотреть, открыв соответствующее окно (консоль). Однако новички нередко забывают открыть консоль ошибок или посмотреть, не появился ли желтый треугольник с восклицательным знаком в левом нижнем углу окна браузера Internet Explorer.

3.4. Ввод и вывод данных

При изучении любого языка программирования желательно в первую очередь ознакомиться со средствами ввода и вывода данных. В самом деле, мы хотим, чтобы наша, пусть даже очень простенькая программка, вывела на экран какое-нибудь нами же предусмотренное сообщение, продемонстрировав тем самым свою работоспособность.

Сценарии на JavaScript могут взаимодействовать с объектами браузера и загруженного в него документа. Для ввода и вывода данных можно воспользо-

ваться методами этих объектов. Каких-то специфических собственных методов вывода данных у JavaScript, интерпретируемого Web-браузерами, не предусмотрено.

Объект, представляющий свойства браузера, называется `window` (окно), а три его метода — `alert()`, `prompt()` и `confirm()` — предназначены для ввода и вывода данных посредством диалоговых окон (панелей). Поскольку это методы объекта window, то для их вызова, согласно каноническим правилам, следует писать `window.alert()`, `window.prompt()` и `window.confirm()` соответственно. Здесь используется так называемый точечный синтаксис, согласно которому перед обращением к свойству (переменной или методу) указывают соответствующий объект, за которым следует точка. Вместе с тем, поскольку объект `window` корневой в объектной модели, при обращении к его свойствам имя можно не указывать. В этом случае интерпретатор и так поймет, что вызываемый метод относится к объекту `window`. Иначе говоря, допускается сокращенная запись без префикса `window`. Например, если мы хотим вывести что-то, обозначенное через `x`, то можно написать в программе `alert(x)`. Однако указанные три метода ввода-вывода данных не единственно возможные.

Объект, представляющий свойства документа, называется `document`, а его методы `write()` и `writeln()` служат для вывода данных непосредственно в клиентскую область браузера, т. е. в ту область, где отображаются Web-страницы (HTML-документы). Объект `document` является очень важным, но не корневым в объектной модели, а потому при обращении к его свойствам требуется указывать имя объекта, например, `document.write("Привет!")`.

В следующих разделах мы рассмотрим перечисленные методы подробнее.

3.4.1. Метод *alert — окно предупреждения*

Данный метод позволяет выводить диалоговое окно с заданным сообщением и кнопкой **OK** (рис. 3.2). В такое окно обычно выводят предупреждения. Синтаксис соответствующего выражения имеет вид:

`alert(`*`сообщение`*`).`

Если ваше *`сообщение`* конкретно, т. е. представляет собой вполне определенный набор символов, то его необходимо заключить в двойные или одинарные кавычки: `alert("Привет!")`. Вообще говоря, *`сообщение`* представляет собой данные любого типа: последовательность символов, заключенную в кавычки, число (в кавычках или без них), переменную или выражение.

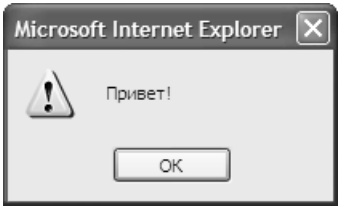

Рис. 3.2. Диалоговое окно, созданное методом `alert()`

Диалоговое окно, выведенное на экран методом `alert()`, можно закрыть, щелкнув на кнопке **OK**. До тех пор, пока вы не сделаете этого, переход к ранее открытым окнам невозможен. Окна, обладающие свойством останавливать все последующие действия пользователя и программ, называются *модальными*. Таким образом, метод `alert()` создает модальное окно.

Ранее уже отмечалось, что метод `alert()` пригоден для вывода промежуточных и окончательных результатов программ при их отладке. При этом можно отобразить результат вычисления какого-либо выражения и приостановить дальнейшее выполнение работы программы до тех пор, пока вы не нажмете кнопку **OK**.

3.4.2. Метод *confirm — окно подтверждения*

Метод `confirm` позволяет вывести диалоговое окно (рис. 3.3) с сообщением и двумя кнопками — **OK** и **Отмена** (Cancel). В отличие от `alert()`, этот метод возвращает логическую величину, значение которой зависит от того, какую из двух кнопок нажал пользователь. Если он щелкнул на кнопке **OK**, то возвращается значение `true` (истина, да); если же была нажата кнопка **Отмена** (Cancel), то возвращается значение `false` (ложь, нет). Возвращаемое значение можно обработать в программе и, следовательно, создать эффект интерактивности, т. е. диалогового взаимодействия программы с пользователем. Синтаксис метода `confirm()` следующий:

`confirm(`*сообщение*`).`

Если *сообщение* представляет собой вполне определенный набор символов, то его необходимо заключить в кавычки, двойные или одинарные:

```
confirm("Вы действительно хотите выйти из программы?").
```

Как и ранее *сообщение* может быть любым: последовательностью символов, заключенной в кавычки, числом (в кавычках или без них), переменной или выражением.

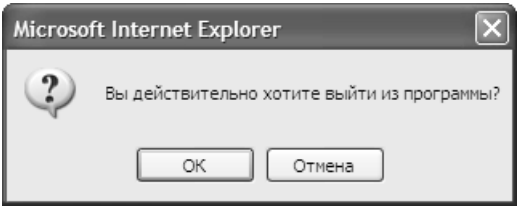

Рис. 3.3. Диалоговое окно, созданное методом `confirm()`

Диалоговое окно, выведенное на экран методом `confirm()`, можно убрать щелчком на любой из двух кнопок (**OK** или **Отмена**). До тех пор, пока вы не сделаете этого, переход к ранее открытым окнам невозможен. Следовательно, окно, создаваемое посредством `confirm()`, является модальным. Если пользователь щелкнет на кнопке **OK**, то метод вернет логическое значение `true` (истина, да), а если он щелкнет на кнопке **Отмена** (Cancel), то — `false`. Возвращаемое значение доступно для дальнейшей обработки в программе и реализации эффекта интерактивности, т. е. диалогового взаимодействия программы с пользователем.

3.4.3. Метод *prompt — окно запроса*

Метод `prompt()` позволяет вывести на экран диалоговое окно с сообщением, а также с текстовым полем, в которое пользователь может ввести данные. Кроме того, в окне предусмотрены две кнопки: **OK** и **Отмена** (Cancel). В отличие от `alert()` и `confirm()`, данный метод принимает два параметра: сообщение и значение, которое должно появиться в текстовом поле ввода данных по умолчанию. Если пользователь щелкнет на кнопке **OK,** метод вернет содержимое поля ввода данных, а если он щелкнет на кнопке **Отмена**, то возвращается логическое значение `false` (ложь, нет). По аналогии с предыдущими методами возвращаемое значение можно проанализировать и создать эффект интерактивности.

Синтаксис метода `prompt()` следующий:

`prompt(`*`сообщение, значение_поля_ввода_данных`*`).`

Параметры метода `prompt()` необязательные. Если вы их не укажете, то будет выведено окно без сообщения, а в поле ввода данных подставлено значение по умолчанию — `undefined` (не определено). Если вы не хотите, чтобы в поле ввода данных появлялось значение по умолчанию, то в качестве значения второго параметра укажите пустую строку `""` (рис. 3.4):

```
prompt("Введите Ваше имя, пожалуйста", "").
```

Рис. 3.4. Диалоговое окно, созданное методом `prompt()` без параметров

Диалоговое окно, выведенное на экран методом `prompt()`, можно закрыть щелчком на любой из двух кнопок, **OK** или **Отмена**. Как и для метода `confirm()`, переход к ранее открытым окнам невозможен, данное диалоговое окно тоже модальное.

3.4.4. Метод *document.write()*

Метод `document.write(список_строк)` позволяет вывести в окно браузера список текстовых строк, разделенных запятыми. Если выводимая строка представляет собой HTML-код, то браузер отобразит результат его интерпретации, а не последовательность тегов. Иначе говоря, теги HTML в выводимом сообщении будут восприниматься браузером как команды, а не просто как текстовые символы.

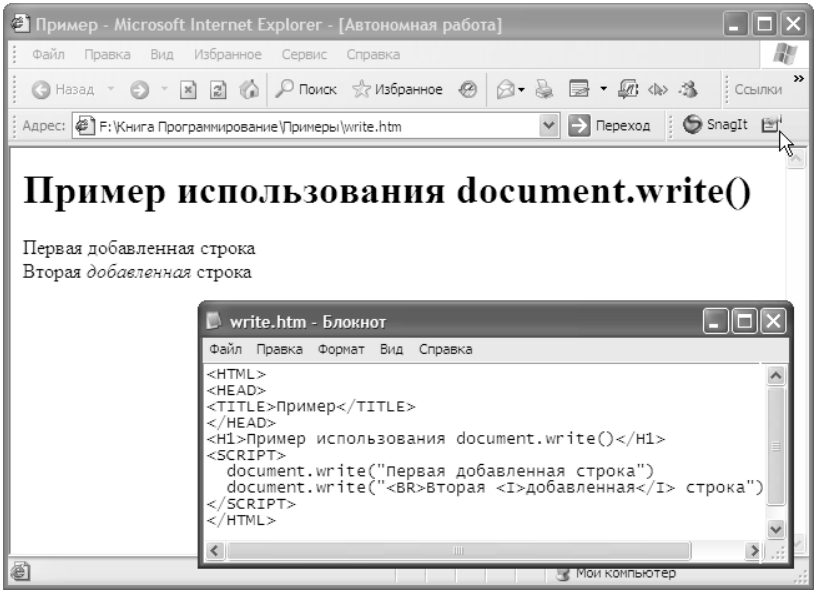

Рис. 3.5. Вывод текста в окно браузера методом `document.write()`

Метод `document.writeln(список_строк)` отличается от предыдущего только тем, что выводит указанную строку (или несколько строк), добавляя в конце невидимый управляющий символ перехода на новую строку. Однако этот и другие специальные символы браузер не воспринимает как команды, управляющие отображением, и заменяет их пробелом. Таким образом, символы, управляющие отображением, практически игнорируются браузером. Если требуется выполнить какое-либо форматирование выводимого текста, то следует указать соответствующие теги HTML, например, для перевода на новую строку пригодны теги `<BR>` и `<P>`. На рис. 3.5 показан результат работы метода `document.write()`.

3.5. Типы данных

Сведения в данном разделе очень важны для понимания языка программирования, хотя и могут показаться неподготовленному читателю скучными и далекими от практики. Тем не менее, новичкам следует мужественно перенести предлагаемое "испытание". Если по прочтении данного раздела у вас останутся белые пятна, то не отчаивайтесь и продвигайтесь далее. Вы всегда сможете вернуться к этим страницам, чтобы уяснить ранее недопонятое и все поставить на свои места.

Итак, данные в JavaScript, как и в любом другом языке программирования, относятся к тому или другому типу. Типы данных, в свою очередь, разделяют на примитивные и составные.

Примитивные типы содержат простые значения, с которыми мы встречаемся в повседневной жизни: числа, строки и логические значения вроде "Да" и "Нет". Так, с числами нам приходится оперировать не только при расчетах перед кассовым аппаратом в магазине, но и при решении более сложных математических задач. Строки содержат какие угодно символы. Семантика строки автоматически не определяется. Кто как не свободный отправитель должен вкладывать некий смысл в созданный им набор символов? Кто как не свободный получатель должен расшифровывать принятое сообщение? Разумеется, кое-что можно сделать автоматически, опираясь на нечто общезначимое, понятное всем и всеми однозначно интерпретируемое. Именно так и происходит в действительности. Одно жестко и однозначно интерпретируется по определенным правилам в соответствии с общечеловеческой культурой, а другое "отдается на откуп" пользователю почти без ограничений. Сакраментальный вопрос всей жизни состоит в том, как сочетать свободу с необходимостью. В программировании он не эпизодически появляется, а присутствует постоянно, искушая и мотивируя программистов всех мастей.

Произвольные данные, без структуры и достаточно четко оформленных идей их дальнейшего употребления, программисту дозволено и рекомендуется оформлять в виде строк. В строке может находиться все, что угодно. Восприятие строк — дело пользователя, а не интерпретатора языка. Однако применительно к скриптовым (интерпретируемым, а не компилируемым) языкам последнее замечание не вполне точно. Интерпретируемые языки, такие как JavaScript и PHP, не "оставляют в покое" строки. Символьные последовательности, участвующие в выражениях с данными других типов, подвергаются автоматическому преобразованию к соответствующему, с точки зрения интерпретатора, типу *(см. разд. 3.5.3)*. Интерпретаторы этих языков пытаются "извлечь" некий смысл, который, возможно, пользователь или программист в них вложил. Например, встречаясь со строкой "25", содержащей число 25, интерпретатор JavaScript попытается привести его к тому или иному типу в зависимости от контекста. Для новичков эта поддержка их интуиции со стороны интерпретатора языка более чем кстати. Но для искушенных программистов, привыкших все держать под своим контролем, это просто "беспредел".

> ***Замечание***
>
> Однако я хочу указать всем, в том числе и профессионалам, что слабый контроль типов с автоматическим приведением к нужному в зависимости от конъюнктуры является очень интересной возможностью, ждущей своих почитателей, которые смогут написать изящные и эффективные коды, решающие не только близлежащие задачи сайтостроительства, но и далеко идущие проблемы, связанные, например, с искусственным интеллектом.

Строка, пусть даже содержащая одни только цифры, не является числом. Поэтому арифметические операции над строками, содержащими числа, могут привести к результатам, существенно отличным от тех, которые мы ожидаем в случае действий над самими числами.

Чтобы как-то определиться, мы должны признать, что число — объект, достаточно ясно определенный в математике, — в языке программирования представляется символически так или иначе. Строка — это набор произвольных символов, которую нельзя интерпретировать, например, как число, дату или что-либо другое. Над строками мы производим одни операции, а над числами — другие. Логическое значение — одно из двух возможных значений, которые обычно обозначаются как true (ИСТИНА) и false (ЛОЖЬ). Для работы с логическими значениями существует своя алгебра, отличная от арифметики. Мы можем сформулировать некое утверждение, которое в данном контексте может быть истинным или ложным. Для обозначения оценки истинности логического выражения в языках программирования предусмотрены специальные ключевые слова: `true` (истина) и `false` (ложь).

Составные типы данных, в отличие от примитивных, могут содержать разнородные данные, а также данные других составных типов. Типичный пример составного типа данных — массив. Массив в JavaScript — это упорядоченный набор данных (называемых элементами), каждое из которых принадлежит к тому или иному типу. Место каждого элемента в массиве фиксировано, а потому доступ к ним возможен по порядковому номеру (индексу). Объект — еще один составной тип данных, причем настолько общий, что позволяет представить все другие составные типы данных, такие как массивы, даты, функции и даже объекты. В объекте данные не обязаны быть упорядоченными, как в массиве.

3.5.1. Примитивные типы данных

В JavaScript имеются пять примитивных типов данных (табл. 3.3).

Таблица 3.3. *Примитивные типы данных в JavaScript*

Тип данных	Примеры	Описание значений

Строковый или символьный (`String`)	`"Привет всем!"`
	`"т.123-4567"`
	`"Сегодня 14.11.2007г."`
		Последовательность символов, в том числе и пустая, заключенная в кавычки, двойные или одинарные

Числовой (`Number`)	`3.14`
	`-567`
	`+2.5`
	`5.7e16`
	`67e-28`
		Десятичное число, представленное в виде последовательности цифр, перед которой может быть указан знак числа (+ или –); перед положительными числами не обязательно ставить знак +; целая и дробная части чисел разделяются точкой. Возможно представление в экспоненциальной, шестнадцатеричной и восьмеричной формах. Числа записываются без кавычек.
		Диапазон чисел: от $\pm 2{,}2250 \times 10^{-308}$ до $\pm 1{,}7976 \times 10^{308}$.
		Диапазон целых чисел: от -2^{31} до $2^{31}-1$

Логический (булевский, `Boolean`)	`true` `false`	Этот тип данных имеет два значения: `true` (истина, да) и `false` (ложь, нет)

Таблица 3.3 *(окончание)*

Тип данных	Примеры	Описание значений
Пустой (Null)	null	Этот тип данных имеет одно значение — null, указывающее на отсутствие какого бы то ни было значения
Неопределенный (Undefined)	undefined	Этот тип данных имеет одно значение — undefined, указывающее, что переменной не присвоено никакое значение

Из перечисленных типов только String, Number и Boolean могут содержать настоящие данные, а Null и Undefined служат специальным целям.

Если вы не хотите присваивать переменной какое-либо конкретное значение, то присвойте ей для определенности значение null. Это действие и будет отвечать вашим чаяниям наилучшим образом. Переменная, имеющая значение null, определена. Она имеет пустое значение. Заметим, что пустая строка (не содержащая ни одного символа, даже пробела), а также число 0 — данные, отличающиеся от null.

Если какая-то переменная встречается в выражении впервые, и ей ранее не присвоено какое бы то ни было значение, то это может вызвать ошибку выполнения, как в листинге 3.10 слева, справа показано, как избежать появления ошибки.

Листинг 3.10

Код с ошибкой

```
<HTML>
<SCRIPT>
   alert(x); /* здесь возникнет
ошибка */
</SCRIPT>
</HTML>
```

Код без ошибки

```
<HTML>
<SCRIPT>
   x = null;
   alert(x); /* выведет сообщение
   "null", но ошибки не будет */
</SCRIPT>
</HTML>
```

Если переменная объявлена с помощью оператора var, например, var x, то она не имеет пока никакого значения и относится к типу undefined (листинг 3.11).

Листинг 3.11

```
<HTML>
<SCRIPT>
   var x;   // объявление переменной без присвоения ей значения
   alert(x); // выведет сообщение "undefined", но ошибки не будет
</SCRIPT>
</HTML>
```

3.5.2. Составные типы данных

В JavaScript поддерживаются три составных типа данных: объекты, массивы и функции. В действительности массивы и функции также являются объектами. Другими словами, такие понятия, как массив и функция, реализованы в JavaScript в виде объектов.

Объект — понятие, обладающее настолько большой общностью, что противопоставляется исторически более раннему понятию "тип данных". Действительно, объект есть контейнер, содержащий переменные и функции, или только одну из этих категорий: он может содержать только переменные или только определения функций. В свою очередь, переменные внутри объекта могут содержать данные как примитивных, так и составных типов (в том числе и данные типа объект). Последнее обстоятельство как раз и позволяет конструировать сколь угодно сложные типы данных. Если исходить из более традиционного понятия "тип данных", то объект — это сложный (составной) тип данных. С другой стороны, поддержка как примитивных, так и сложно устроенных данных может быть обеспечена соответствующими объектами.

Исторически сложилось, что сначала появилось понятие типа данных (в смысле множества допустимых значений). Более того, в первых языках программирования это было множество однородных значений, например, только чисел, строк или логических величин. Нельзя было сформировать тип данных, состоящих одновременно, например, из чисел и строк (числа и строки — разнородные данные). Далее появились структуры в виде упорядоченного множества (последовательности) данных различных типов. Структура — это составной тип данных, если угодно прибегнуть к такой интерпретации. Наконец, было введено еще более общее понятие объекта, который мог содержать не только разнородные значения (как и структура), но еще и функции (называемые методами). Другими словами, объект может содержать не только пассивные данные (переменные), но и активные (функции).

Обратите внимание, что при разъяснении того, что такое объект, мы использовали понятие "тип данных". Но если мы усвоили понятие "объект", то его, как будто, можно рассматривать в качестве первичного по отношению к понятию "составной тип данных". Однако это можно сделать лишь с некоторой натяжкой. Понятия "тип данных" и "объект" не вполне эквивалентны. Многие типы данных реализуются и представляются в конкретных языках, в том числе и в JavaScript, посредством соответствующих объектов — программных конструкций особого рода. Например, массив реализован и представлен в JavaScript как некий объект под названием Array. Этот объект имеет ряд свойств, содержащих не только элементы представляемого им массива, но и другие важные вещи: количество всех элементов массива и методы работы с ним (например, сортировка элементов, объединение двух массивов). Однако объект — неупорядоченное множество его свойств, а массив — упорядоченное множество его элементов. Кроме того, операции над массивом являются чем-то внешним по отношению к самому массиву — упорядоченному набору данных. По этим и некоторым другим причинам массивы выделяют в особый тип данных, отличающийся от объектов. Как бы то ни было, начинающие, и не только, могут относиться к массивам как к особому типу данных, т. е. как к специальной конструкции, идея и польза которой очевидны. Позже мы уточним это понятие настолько, чтобы его можно было легко применять в программировании.

Аналогично, все, что связано с понятием функции, можно реализовать и представить в виде некоторого объекта. В JavaScript каждая функция реализуется посредством объекта `Function`. Такое представление обладает многими достоинствами, которые оказываются востребованными при составлении более или менее сложных программ. Однако любой программист может ограничиться традиционным пониманием функции как блока программного кода, который можно вызвать для исполнения по имени этой функции. В этом смысле функцию можно отнести к особому типу данных. Интерпретаторы, прежде чем начать выполнение кода, анализируют его в целом на присутствие в нем определений функций, т. к. код функций необходимо "усвоить" прежде, чем он будет вызван где-то и когда-то (до или после определения). В отличие от функций, другие операторы выполняются интерпретатором друг за другом в порядке их упоминания в исходном коде.

Если все сказанное внесло некоторую сумятицу в скложившуюся у вас систему понятий, то не расстраивайтесь, со временем все станет на свои места. На первых порах вы можете пропустить как будто путаные общие рассуждения о типах и объектах, сосредоточившись пока на чем-то более понятном (например, на простом определении типа данных). Простейшие типы данных, как я уже говорил, — это примитивные данные, такие как числа, строки сим-

волов и двухэлементное множество, содержащее логические значения `true` и `false`. Необходимость в других типах данных обусловлена более сложными задачами их обработки.

В JavaScript имеется множество встроенных объектов, обеспечивающих работу с числами, строками, массивами, датами, функциями и др. Некоторые из этих объектов соответствуют, как говорят, типам данных. Например, для работы с массивами имеется объект `Array`, для манипулирования датами и временем — `Date`, а для выполнения более или менее сложных математических вычислений — `Math`. Перечень встроенных объектов приведен в табл. 3.4, а их подробное описание будет приведено в последующих разделах данной главы.

Таблица 3.4. *Встроенные объекты в JavaScript*

Объект	Описание
`Array`	Обеспечивает массив данных
`Boolean`	Объект, соответствующий логическому типу данных `Boolean`
`Date`	Обеспечивает работу с датами и временем
`Error`	Обеспечивает хранение информации об ошибках и возможность создания исключительных ситуаций
`Function`	Обеспечивает возможности, связанные с функциями, например, проверку количества переданных параметров
`Global`	Обеспечивает встроенные функции для преобразования и сравнения данных
`Math`	Обеспечивает множество математических функций и констант. Применяется для выполнения более сложных вычислений, чем те, которые обеспечиваются арифметическими операторами
`Number`	Объект, соответствующий числовому типу данных
`Object`	Базовый объект, из которого получаются все остальные объекты, в том числе и пользовательские
`RegExp`	Обеспечивает сложные способы обработки строк на основе так называемых регулярных выражений
`String`	Объект, соответствующий строковому типу данных

3.5.3. Автоматическое преобразование типов данных

При составлении выражений на языке программирования нередко бывает, что вовлекаемые в обработку данные принадлежат различным типам, в то время как операции над ними могут быть корректно выполнены, если данные

однотипны. Например, пусть переменная *x* имеет числовое значение 2, а переменная *y* — строковое значение "3". С обывательской точки зрения и то, и другое — числа, но для интерпретатора языка JavaScript это разные вещи: переменная *x* содержит само число, а переменная *y* — строку, содержащую число. Числа обычно составляются из цифр, знаков положительного и отрицательного числа, разделителя целой и дробной частей, а также других специальных символов. Строка является набором произвольных символов, который заключают в кавычки. Что будет означать операция сложения этих разнотипных данных, суммирование чисел или склейку соответствующих строк? Попытка сложить значения этих переменных в JavaScript даст в результате не число 5, а строку "23". В этом можно убедиться с помощью кода, приведенного в листинге 3.12.

Листинг 3.12

```
<HTML>
<SCRIPT>
    x = 2;
    y = "3";
    alert(x + y);   // выводится строка "23"
</SCRIPT>
</HTML>
```

В данном случае интерпретатор JavaScript автоматически преобразует число 2 в строку "2", а операцию "+" выполнит как склейку строк "2" и "3".

В других языках все может происходить прямо наоборот. Например, в выражениях языка PHP с разнотипными данными строки приводятся к числовому типу. Так что результат выполнения выражения 2 + "3" в PHP был бы совсем другим: 5. Будьте внимательны при переходе от одного интерпретируемого языка к другому, особенно при изучении особенностей автоматического преобразования типов данных.

JavaScript — язык со слабым или, как еще говорят, динамическим контролем типов данных. Это означает следующее:

- Одна и та же переменная может принимать значения различных типов. Переменная создается при первом ее упоминании в программе, а тип данных, которые ей можно присвоить, не является раз и навсегда заданным. В одном месте программы этой переменной можно присвоить значение

одного типа, а в другом месте — иного типа. Например, вполне допустима такая последовательность операторов присваивания: `x=2; x="Вася"`.

> ***Замечание***
>
> Во многих других языках (например, C, Pascal) при объявлении переменной устанавливается и ее тип, который остается постоянным. Попытка присвоить такой переменной значение недопустимого типа всегда приводит к сообщению об ошибке. Но в JavaScript это не так.

- При выполнении операций над данными различных типов интерпретатор автоматически приводит их к некоторому общему требуемому типу. Например, при сложении числа и строки число приводится к строковому типу, в результате вторая строка приписывается ("приклеивается") к первой. Так, результат выражения 125 + "Вася" — строка "125Вася". Рассмотрим еще один пример. Иногда в выражении по смыслу требуется логический тип. Так, в операторе условия `if(x){…}` (читается "если истинно `x`, то выполнить `{…}`") подразумевается, что переменная `x` должна иметь логическое значение (true или false). Однако в действительности она может принимать значение любого типа, но интерпретатор приведет его к логическому типу автоматически сообразно с некоторыми правилами.

Как происходит автоматическое преобразование типов, очень важно знать, чтобы избежать возможных недоразумений. Однако существуют методы принудительного приведения к заданному типу, которые позволяют контролировать типы данных вручную *(см. разд. 3.5.4)*.

Преобразование строк (*String*)

Автоматическое преобразование строк (данных типа `String`) в данные других типов происходит по следующим правилам:

- **Преобразование в Boolean** — в результате получается `false` (ложь), если строка пуста (""), т. е. не содержит ни одного символа, даже пробела (строка, содержащая один или более пробелов не пуста); в противном случае — `true` (истина).
- **Преобразование в Number** — происходит анализ строки как числового литерала с целью получения подходящего значения. Если этот процесс заканчивается неудачей, то возвращается константа `NaN` (от Not a Namber — не число); в противном случае возвращается число. Например, строка "5.2" преобразуется в число 5,2, а в результате преобразования строки "100px" будет получено `NaN`.
- **Преобразование в Object** — возвращается объект `String`, свойство `value` которого равно значению данной строки.

Преобразование чисел (*Number*)

Автоматическое преобразование чисел (данных типа `Number`) в данные других типов происходит по следующим правилам:

- **Преобразование в Boolean** — в результате получается `false` (ложь), если число равно нулю или `NaN` (Not a Number — не число); в противном случае — `true` (истина). `NaN` — константа, содержащаяся в объекте `Number`.
- **Преобразование в String** — в результате получается строка, содержащая (представляющая) данное число. Например, число 3,14 преобразуется в строку "3,14".
- **Преобразование в Object** — возвращается объект `Number`, свойство `value` которого равно значению данного числа.

Преобразование логических значений (*Boolean*)

Автоматическое преобразование логических значений (данных типа Boolean) в данные других типов происходит по следующим правилам:

- **Преобразование в Number** — в результате получается 1, если `true` (истина); в противном случае — 0.
- **Преобразование в String** — в результате получается строка "true", если `true`; в противном случае — строка "false".
- **Преобразование в Object** — возвращается объект `Boolean`, свойство `value` которого равно данному логическому значению.

Преобразование пустого значения (*null*)

Автоматическое преобразование пустого значения (данных типа `null`) в данные других типов происходит по следующим правилам:

- **Преобразование в Boolean** — в результате получается `false` (ложь).
- **Преобразование в Number** — в результате получается 0.
- **Преобразование в String** — в результате получится строка "null".
- **Преобразование в Object** — генерируется исключительная ситуация TypeError (ошибка преобразования типа).

Преобразование неопределенного значения (*undefined*)

Автоматическое преобразование неопределенного значения (данных типа undefined) в данные других типов происходит по следующим правилам:

- **Преобразование в Boolean** — в результате получается `false` (ложь).
- **Преобразование в Number** — в результате получается `NaN` (не число).

- **Преобразование в String** — в результате получится строка "undefined" (не определено).
- **Преобразование в Object** — генерируется исключительная ситуация TypeError (ошибка преобразования типа).

3.5.4. Принудительное преобразование типов данных

При попытке выполнить операции над разнотипными данными последние автоматически приводятся к определенному типу так, чтобы выражение могло быть выполнено. Об этом говорилось в *разд. 3.5.3*. Во многих случаях результат оказывается ожидаемым, но иногда возникают недоразумения, устранить которые часто удается принудительным приведением данных к требуемому типу. Например, пусть переменная `width="100px"` хранит значение ширины окна в пикселах, причем единица измерения (px — пикселы) указана в самом этом значении строкового типа. Если мы захотим увеличить размер окна на 10 пикселов, используя для этого выражение `width = width + 10`, то значение переменной `width` станет равным `"100px10"`, а не ожидаемой величине 110. В данном случае интерпретатор автоматически приведет число 10 к строковому значению "10" и припишет его справа к строковому значению "100px". Чтобы добиться своего, нам потребуется принудительно преобразовать строку "100px" в соответствующее число, а именно в 100.

Для преобразования строк в числа в JavaScript предусмотрены встроенные функции `parseInt()` и `parseFloat()`. В действительности это методы объекта `global`, предназначенного для хранения общедоступных данных и методов. В силу глобальности объекта `global` для вызова его методов не нужно указывать сам объект, т. е. для вызова `parseInt(x)` достаточно просто записать `parseInt(x)`. Возможно, по этой причине методы объекта `global` называют встроенными функциями JavaScript.

Функция

`parseInt(`*`строка`*`, `*`основание`*`)`

преобразует указанную в параметре строку в целое десятичное число. Второй параметр, *`основание`*, определяет систему счисления, в которой представлено содержащееся в строке число (8, 10 или 16); если основание не указано, то предполагается 10, т. е. десятичная система счисления.

При этом число округляется простым отбрасыванием дробной части.

Примеры

```
parseInt("3.14")        // результат = 3
parseInt("-7.875")      // результат = -7
parseInt("123")         // результат = 123
parseInt("Саша")        // результат = NaN, т. е. не является числом
parseInt("25 руб. 50 коп.")   // результат = 25
parseInt("25.5e3")      // результат = 25
parseInt("15",8)        // результат = 13
parseInt("0xFF",16)     // результат = 255
parseInt("ff",16)       // результат = 255
```

Функция

`parseFloat(`*строка*`)`

преобразует указанную строку в десятичное число с плавающей разделительной (десятичной) точкой.

Примеры

```
parseFloat("3.14")      // результат = 3.14
parseFloat("-7.875")    // результат = -7.875
parseFloat ("123")      // результат = 123
parseFloat ("Саша")     // результат = NaN, т. е. не является числом
parseFloat("25 руб. 50 коп.") // результат = 25
parseFloat ("25.5")           // результат = 25.5
parseFloat ("25.5 рублей")    // результат = 25.5
parseFloat("25.5e3")          // результат = 25500
```

Данные функции исследуют строку на содержание в ней числа. Если первый символ не цифра или знак числа, то сразу возвращается `NaN`. В противном случае анализируется следующий символ строки и если это не цифра, то анализ прекращается и возвращается число, выделенное на предыдущих этапах. Метод `parseInt()` закончит работу, как только встретит символ, не являющийся цифрой (например, точку, пробел или букву). Поэтому строка, содержащая число в экспоненциальной форме, будет преобразована в число, содержащее только целую часть мантиссы. Например, `parseInt("25.5e3")` вернет 25, а не 25500. Метод `parseFloat()` анализирует строку глубже: деся-

тичная точка и даже буква "e", строчная или прописная, а также знак и цифры за ней будут вовлечены в обработку для получения числа.

Числа в строки можно преобразовать еще проще. Для этого достаточно к пустой строке прибавить это число, т. е. воспользоваться оператором сложения `+`. Например, вычисление выражения `"" + 3.14` даст в результате строку `"3.14"`. Понятно, что в данном случае преобразование выполняет сам интерпретатор, используя свой "интеллект".

Для определения того, является ли значение выражения числом, служит метод (или встроенная функция)

`isNaN(`*значение*`)`,

который возвращает результат логического типа. Если указанное значение не является числом, функция возвращает `true`, иначе — `false`. Однако здесь термин "число" не совпадает с понятием "значение числового типа". Функция `isNaN()` считает числом и данные числового типа, и строку, содержащую только число. Логические значения также идентифицируются как числа. При этом значению `true` соответствует 1, а значению `false` — 0. Таким образом, если `isNaN()` возвращает `false`, то это указывает, что значение параметра имеет числовой тип, либо является числом, преобразованным в строковый тип, либо является логическим (`true` или `false`).

Примеры

```
isNaN(1234)        // результат false (т. е. это число)
isNaN("1234")      /* результат false (т. е. это число,
                        хотя и в виде строки) */
isNaN("25 рублей")  // результат true (т. е. это не число)
isNaN("Саша")      // результат true (т. е. это не число)
isNaN(true)        // результат false
isNaN(false)       // результат false
```

Как видно, значение `isNaN(x)` для любого логического значения `x` возвращает `false`, что указывает на то, что `x` – число. Дело в том, что функция `isNaN` пытается привести значение `x` к числовому типу по правилам автоматического преобразования типов *(см. разд. 3.5.3)*. Логические значения `true` и `false` преобразуются по этим правилам в числа 1 и 0 соответственно. Автоматическое приведение значений "Саша" и "25 рублей" к числовому типу заканчивается получением значения `NaN` и поэтому функция `isNaN()` возвращает для этих значений `true`.

Кроме рассмотренных, в JavaScript имеются и другие функции приведения данных к заданному типу:

- `Number(`*выражение*`)` — десятичное число или `NaN` (если не удалось преобразовать указанное *выражение* в число). Эта функция работает следующим образом:
  - если *выражение* есть число, то возвращается это же число;
  - если *выражение* есть логическая величина, то возвращается 1 или 0 в зависимости от ее значения (если `true`, то 1; если `false`, то 0);
  - если *выражение* есть строка, то функция пытается преобразовать ее в число как описанные выше функции `parseInt()` и `parseFloat()`;
  - если *выражение* есть `udefined` (не определено), то результатом является `NaN` (не число).
- `String(`*выражение*`)` — приводит данные, получающиеся в результате вычисления выражения, которое указано в качестве параметра, к строковому типу. Преобразование происходит по следующим правилам:
  - если значение выражения имеет строковый тип, то возвращается это же значение;
  - если значение выражения имеет числовой тип, то возвращается строка, содержащая это число;
  - если значение выражения имеет логический тип, то возвращается строка `"true"` или `"false"` в зависимости от значения выражения;
  - если значение выражения не определено (`undefined`), то возвращается пустая строка `""`.
- `Boolean(`*выражение*`)` — приводит данные, получающиеся в результате вычисления выражения, которое указано в качестве параметра, к логическому типу. Преобразование происходит по следующим правилам:
  - если значение выражения имеет логический тип, то возвращается это же значение;
  - если значение выражения имеет числовой тип, то возвращается `true`, если число не равно нулю, и `false` — в противном случае;
  - если значение выражения имеет строковый тип, то возвращается `true`, если строка не пуста, и `false` — в противном случае; напомню, что строка, содержащая только пробелы, не пуста.
- `Array(`*элемент0* `[`*элемент1*`, ..., `*элементN*`]])` — создает массив из элементов, указанных в качестве параметров.

В действительности перечисленные функции являются обращениями к соответствующим объектам: `Number`, `String`, `Boolean` и `Array`. Например, `String(`*`выражение`*`)` — обращение к так называемому статическому строковому объекту. Об этом и о других объектах будет рассказано далее.

3.6. Переменные и оператор присваивания

3.6.1. Имена переменных

Как уже упоминалось, переменная — это контейнер для хранения значений. Значения могут быть любых типов, предусмотренных в JavaScript, а присваивание происходит с помощью оператора, обозначаемого знаком равенства =. Например в выражении `x = 5` имя (идентификатор) переменной — `x`, а значение, которое ей присваивается (приписывается, назначается) — `5` (в данном случае это числовое значение), следовательно, переменная `x` является числовой. Если мы теперь запишем в своей программе выражение `x = "Вася"`, то переменная `x` приобретет новое значение строкового типа `"Вася"`.

Переменная имеет имя — последовательность букв, цифр, символа подчеркивания и даже символа $, которая не должна начинаться с цифры.

Примеры правильных имен переменных:

`myname`, `myName`, `_myname`, `my_Name` , `myName134`, `$myname`.

Примеры неправильных имен переменных:

`my name`, `310group`, `1000 $`.

Имя, соответствующее указанным правилам, может быть каким угодно. Однако желательно, чтобы оно отражало суть содержащихся в соответствующей переменной значений и/или цель ее использования. Если в имени переменной вы хотите указать несколько слов, то разделите их символом подчеркивания или напишите каждое из этих слов (начиная со второго) с прописной буквы, например, `my_first_name`, `myFirstName`. Нередко первый символ в имени переменной указывает на ее тип. Например, в имени `sMyName` первый символ указывает на то, что данная переменная содержит данные строкового типа (`string`). Вы можете придумать свой стиль образования имен переменных. Важно лишь то, чтобы он был понятен хотя бы вам самим. Тем не менее, профессиональные программисты JavaScript не рекомендуют выбирать в качестве первого символа имени переменной подчеркивание или знак $.

JavaScript регистрозависимый язык. Это означает, что изменение регистра символов (с прописных на строчные и наоборот) в имени переменной приво-

дит к другой переменной. Например, `myvar`, `Myvar` и `MYVAR` — различные переменные.

При выборе имен переменных (а также имен функций) недопустимы зарезервированные ключевые слова, которые используются или планируются к применению в последующих версиях в качестве элементов синтаксиса. Так, не следует выбирать в качестве имен слова из списка:

abstract	else	int	super
boolean	extends	interface	switch
break	false	long	synchronized
byte	final	native	this
case	finally	new	throw
catch	float	null	throws
char	for	package	transient
class	function	private	true
const	goto	protected	try
continue	if	public	typeof
default	Implements	reset	var
delete	import	return	void
do	in	short	while
double	instanceof	static	with

3.6.2. Создание переменных

В данном разделе мы рассмотрим чрезвычайно важную тему видимости переменных. Она актуальна не только применительно к JavaScript, но и вообще ко всем языкам программирования. Интересно, что писать вполне работоспособные сценарии для сайтов удается в большинстве случаев и тем, кто данной темой совсем не владеет. Однако это "пиррова победа", обусловленная благополучным стечением обстоятельств или простотой задачи. Стоит лишь немного усложнить сценарий, потребовать взаимодействия нескольких программных блоков, как сразу же возникнут недоразумения и ошибки, обусловленные, скорее всего, недопониманием того факта, что переменная может быть, а может и не быть доступной в том или ином блоке кода или контексте выполнения сценария.

Переменную в JavaScript можно создать с помощью оператора присваивания:

```
имя_переменной = значение.
```

Например, в выражении `myname = "Вадим"` переменной с именем (идентификатором) `myname` присваивается значение строкового типа `"Вадим"`. Пока данная переменная не получит какого-нибудь значения другого типа, она останется строковой.

Кроме того, для объявления переменной предусмотрен специальный оператор `var`, например,

```
var myname;
myname = "Вадим";
```

или еще короче:

```
var myname = "Вадим";
```

В данном примере одновременно с объявлением переменной ей присваивается значение. Присваивать значения не обязательно всем объявляемым переменным. Например в выражении

```
var x, y, z = "Привет!"
```

объявлены (созданы) переменные `x`, `y` и `z`, причем только переменной `z` присвоено конкретное значение. Переменные `x` и `y`, которым пока не присвоены значения в программе, относятся к типу `undefined` и имеют такое же значение. Чтобы убедиться в этом, достаточно выполнить код листинга 3.13.

Листинг 3.13

```
 <HTML>
<SCRIPT>
    var x, y, z = "Привет!";
    alert(x);
    alert( y);
   alert(z);
</SCRIPT>
</HTML>
```

Обратите внимание, что с помощью одного ключевого слова (оператора) `var` можно создать (инициализировать) сразу нескольких переменных. При этом имена переменных и/или операторы присваивания разделяются запятыми.

Хотя переменную в JavaScript можно создать как с использованием оператора `var`, так и без него, просто с помощью оператора присваивания, однако хороший стиль программирования состоит в применении `var`. Он хорош потому, что позволяет читателю программного кода достаточно четко различить, где переменная была объявлена впервые, а где она просто используется. Кроме того, оператор `var` позволяет задать так называемые локальные

переменные, которые доступны только из некоторой части программного кода или, как еще говорят, в рамках некоторого контекста. И это, пожалуй, самое главное назначение данного оператора. Дело в том, что переменные могут быть глобальными и локальными. Начинающие программисты обычно не чувствуют разницу, пока не встретятся с недоразумениями на практике.

Переменные, введенные в обиход с помощью оператора присваивания (без оператора `var`) являются глобальными, т. е. видимыми или, иначе говоря, доступными во всей программе, включая и программные блоки (например, тела функций), если в них не приняты специальные меры, о которых будет рассказано позже. Вместе с тем, глобальные переменные можно создать и с помощью оператора `var`.

Итак, мы приступаем к подробному рассмотрению очень важной темы — области видимости и контекста переменной. Область видимости — это часть программы, в которой переменная доступна или, как еще говорят, видна. Видимая переменная может быть использована в программе, а невидимая — нет. Контекст — набор определенных данных, образующих среду выполнения программы. При запуске браузера (интерпретатора JavaScript) создается глобальный контекст, который помимо специфических объектов браузера и загруженного в него документа содержит объекты собственно JavaScript (например, объекты массива `Array`, даты `Date`, объект для сложных математических вычислений `Math` и др.). При активизации обработчика события (например, после щелчка кнопкой мыши на элементе Web-страницы) создается локальный контекст этого обработчика. Аналогичное происходит и при вызове функции: интерпретатор создает локальный контекст, существующий до тех пор, пока не завершится выполнение этой функции. Все переменные, объявленные в теле функции с помощью оператора `var`, существуют только в рамках контекста этой функции во время ее выполнения. По завершении работы функции этот контекст уничтожается. Контексты могут быть иерархически вложены друг в друга. Интерпретатор, встречая переменную в некотором контексте, анализирует, определена ли она в нем в нем. Если да, то это — локальная переменная, существующая в данном контексте. В противном случае интерпретатор ищет определение переменной в объемлющем контексте. Эти действия выполняются рекурсивно, пока не будет найден контекст, содержащий определение переменной. Не исключено, что при выходе на глобальный контекст определение переменной так и не будет найдено. Тогда переменная оказывается неопределенной.

В ранее рассмотренных примерах все переменные были глобальными. В следующем примере (листинг 3.14) переменная `x` глобальная и имеет значение `"Привет!"`, несмотря на то, что после первичного ее объявления и присваивания значения эта же переменная вновь объявляется, но без присваивания.

Листинг 3.14

```
<HTML>
 <SCRIPT>
   var x = "Привет!";
   var x;
   alert(x);      // вывод строки "Привет!"
 </SCRIPT>
 </HTML>
```

Здесь интерпретатор, в контексте метода `alert()` не находит определения переменной `x`, а в объемлющем контексте он не останавливается на промежуточном определении `var x`, поскольку в данном пункте `x` остается `undefined`. Продолжение поиска в рассмотренном примере заканчивается успехом: выражение `var x = "Привет!"` полностью определяет переменную `x`, значение которой метод `alert()` выводит в диалоговое окно.

Понятия контекста и области видимости переменной особенно важны при задании функций, а также обработчиков событий. О функциях и обработчиках событий будет рассказано более подробно в следующих разделах, а также в *главе 4*. А сейчас мы сконцентрируем свое внимание лишь на теме видимости переменных в соответствующих программных блоках.

В теле функции можно использовать переменные, объявленные в программном коде, более высокого уровня, т. е. в коде, объемлющем определение данной функции. Например, в основной программе, из которой данная функция вызывается. Будут ли такие переменные видны в теле функции? А если в теле функции изменить значения таких переменных, то станут ли они доступными во внешней программе?

На рис. 3.6 представлен исходный код сценария, иллюстрирующий видимость переменной в контексте функции `myfunc()` и глобальном контексте, а также результат его выполнения в окне браузера. Переменная `x` определена в основной программе (глобальном контексте) вне тела функции и является глобальной. Она видна также и в теле функции (в контексте ее выполнения), а новое значение "До свидания!" этой переменной, присвоенное в теле функции, остается таким же и после завершения работы этой функции, т. е. в глобальном контексте.

Теперь немного изменим сценарий: переменную `x` в теле функции объявим с помощью оператора `var`. Код этого сценария в текстовом редакторе и результат его выполнения в браузере показаны на рис. 3.7. В данном случае при первом своем упоминании в теле функции `myfunc()` переменная `x` оказывается неопределенной (`undefined`).

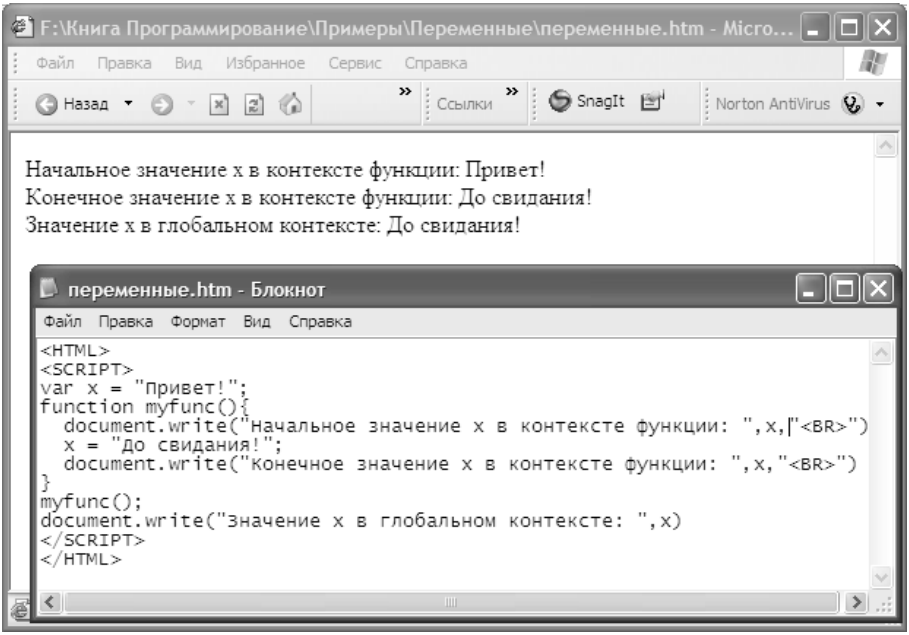

Рис. 3.6. Глобальная переменная x, видимая в теле функции

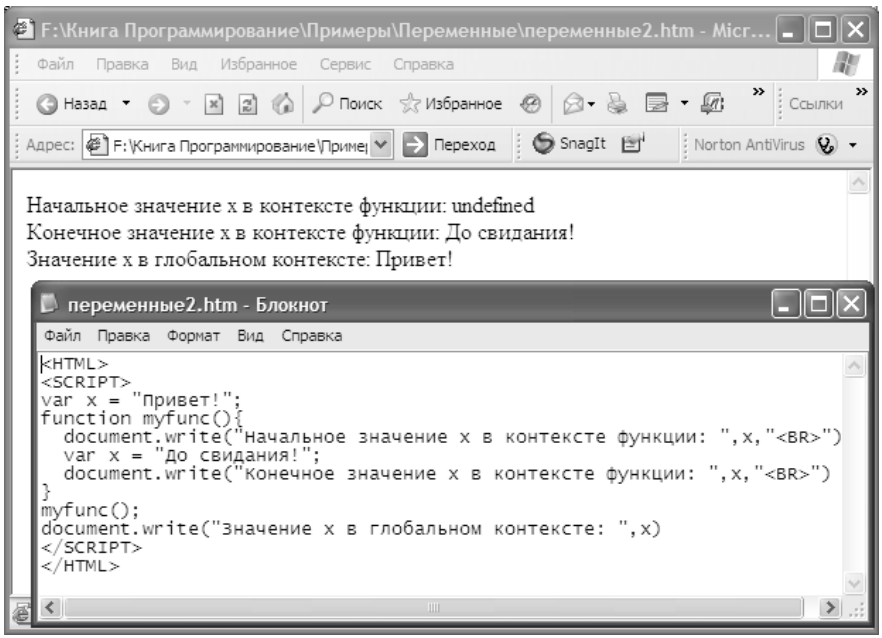

Рис. 3.7. Локальная переменная x в теле функции

Действительно, при вызове функции интерпретатор, встречая в первый раз переменную `x`, ищет ее определение в локальном контексте (т. е. в теле этой функции) и находит его: `var x = "До свидания!"`. Следовательно, это локальная переменная. Однако определение переменной `x` в теле функции следует после ее первого упоминания в пределах кода той же функции. Поэтому при выполнении выражения `document.write("Начальное значение x в контексте функции: ", x, "<BR>")` она еще не определена, и потому имеет значение `undefined`. Зато в выражении `document.write("Конечное значение x в контексте функции: ",x,"<BR>")`, следующем за определением `var x = "До свидания!"`, она имеет заданное в этом определении значение. Но значение `"До свидания!"` переменная `x` имеет только при выполнении функции `myfunc()`. Вне этого контекста (т. е. после завершения работы функции) переменная `x` видна как имеющая значение `"Привет!"`.

Таким образом, использование оператора `var` перед именем переменной в теле функции делает ее локальной, отличной от одноименной глобальной переменной. При этом говорят, что оператор `var` в программном блоке (например, в теле функции) маскирует эту переменную, т. е. делает невидимой в объемлющем контексте. На рис. 3.8 приведен еще один пример, показывающий маскирующий эффект оператора `var` в теле функции.

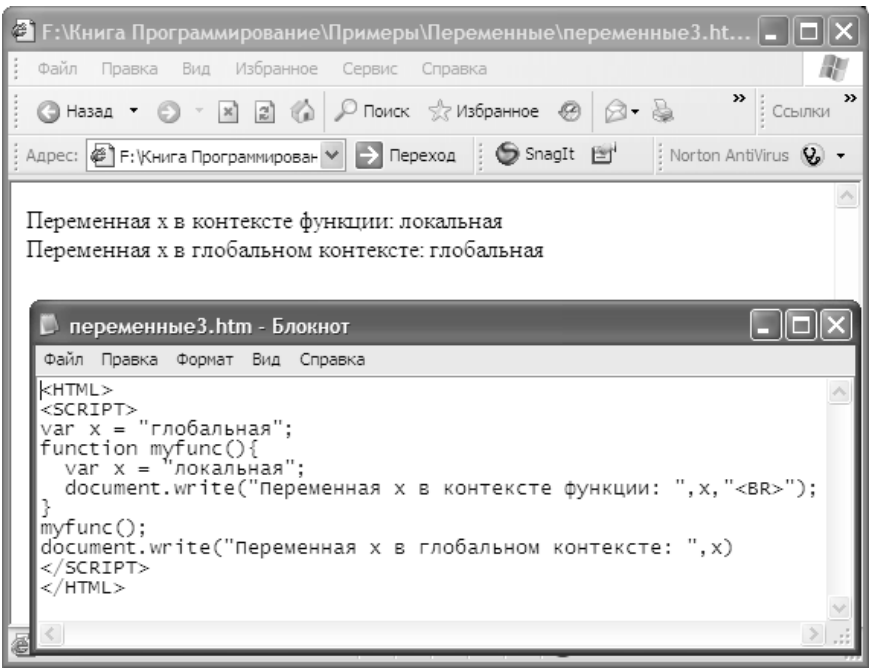

Рис. 3.8. Маскирующий эффект оператора `var`

Теперь рассмотрим очень важный для понимания областей видимости, но, возможно, не часто применяемый способ задания переменных. На рис. 3.9 показан пример, в котором переменная `x` объявлена в основной программе, а также в функции `myfunc1()`, которая содержит вызов другой функции — `myfunc2()`. Последняя функция просто использует, но не объявляет переменную `x`. Как переменная `x` видна в функции `myfunc2()`? Иначе говоря, какое значение, заданное в функции `myfunc1()`, или же в основной программе, будет видно в контексте выполнения функции `myfunc2()`? В данном случае объемлющим контекстом для `myfunc2()` будет не тело функции `myfunc1()`, а глобальный контекст. Оператор `var x` в функции `myfunc1()` делает переменную `x` локальной в полном смысле этого слова, т. е. не допускает ее видимости даже в функции `myfunc2()`, вызываемой из `myfunc1()`.

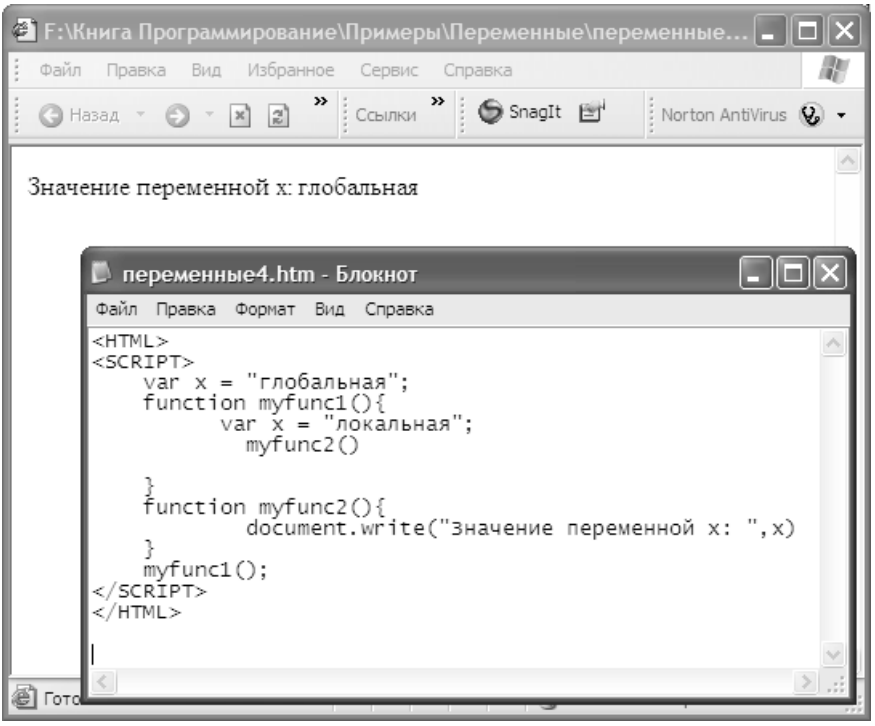

Рис. 3.9. Маскирующий эффект оператора `var` в случае вызова одной функции из другой

При задании определения функции в круглых скобках около ее имени указывают переменные, называемые формальными параметрами: `function (x, y) {…}`.

Эти переменные-параметры в теле функции являются локальными переменными. На рис. 3.10 показан пример, в котором параметр функции `myfunc(x)` имеет то же имя `x`, что и глобальная переменная. Однако при вызове этой функции создается локальная переменная `x`, которая принимает значение одноименной глобальной переменной. Любое изменение значения переменной `x` в теле функции не повлияет на значение глобальной переменной `x`.

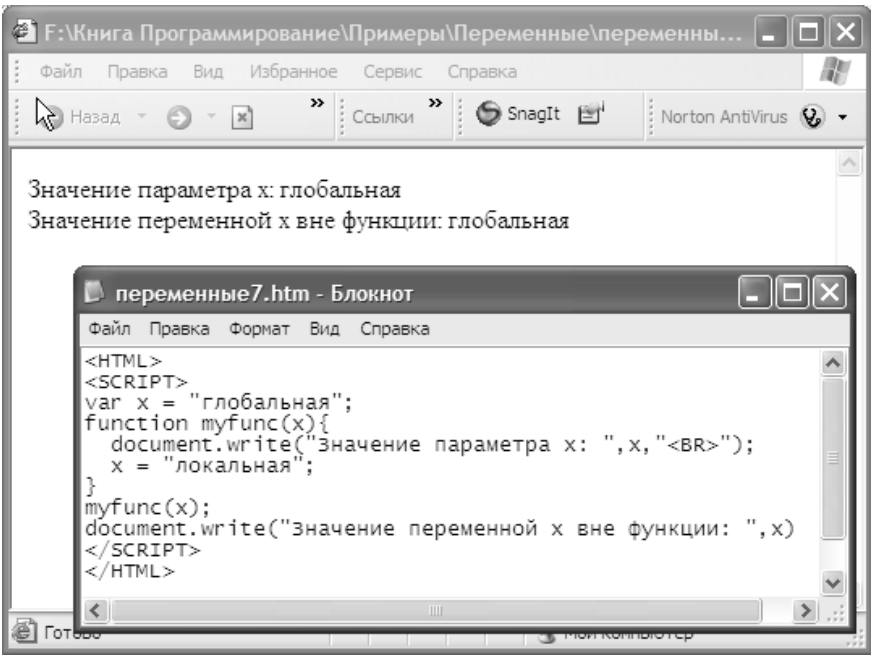

Рис. 3.10. Формальный параметр функции является локальной переменной в ее теле

В JavaScript можно задавать так называемые обработчики событий — программные коды, реагирующие на события, возникающие при работе с документом, загруженным в браузер, такие как щелчок кнопкой мыши, наведение ее указателя на элемент Web-страницы, нажатие на клавишу клавиатуры и т. п. В обработчиках событий нередко также требуется создавать переменные для сохранения данных. Программный код обработчика событий обладает собственным контекстом, как и тело функции. Об обработчиках событий более подробно будет рассказано в *разд. 4.3*. А сейчас мы рассмотрим лишь пример, иллюстрирующий особенности видимости переменных, определенных в обработчике событий (листинг 3.15).

Листинг 3.15

```
<HTML>
<SCRIPT>
var x = "глобальная";
</SCRIPT>
<INPUT TYPE="button" value="Кнопка 1"
onmouseover = "var x = 'локальная'; alert('Переменная x '+ x)">
<INPUT TYPE="button" value="Кнопка 2"
onmouseover = "alert('Переменная x '+ x)">
</HTML>
```

Данный код отображает в браузере две кнопки, наведение указателя мыши на которые выводит диалоговое окно с сообщением о значении переменной `x`. Вывод этого окна обеспечивает обработчик события `onmouseover`. В основной программе (секции, заданной тегом `<SCRIPT>`) определена глобальная переменная `x` со значением "глобальная". Наведем указатель мыши сначала на первую кнопку, а затем на вторую. При наведении указателя мыши на первую кнопку обработчик этого события объявляет локальную переменную `x` со значением "локальная" и выводит соответствующее сообщение в диалоговом окне. При наведении указателя на вторую кнопку будет выведено сообщение "Переменная x глобальная". Таким образом, значение "локальная" существует только в контексте обработчика события для первой кнопки. В обработчике события для второй кнопки нет определения переменной `x`, поэтому интерпретатор ищет его в объемлющем контексте — в данном случае в глобальном контексте всего документа.

Аналогичная ситуация происходит и в случае нескольких обработчиков различных событий, привязанных к одному и тому же элементу документа (листинг 3.16).

Листинг 3.16

```
<HTML>
<SCRIPT>
var x = "глобальная";
</SCRIPT>
<INPUT TYPE="button" value="Кнопка"
onmouseover="var x='локальная'; alert('Переменная x '+ x)"
 onclick="alert('Переменная x '+ x)">
</HTML>
```

Здесь кнопка — элемент документа, заданный тегом <INPUT>, имеет два обработчика событий: onmouseover и onclick, которые запускаются при наведении указателя мыши на кнопку и при щелчке на ней соответственно. Наведение указателя мыши на кнопку выводит сообщение "Переменная x локальная", а щелчок на кнопке — сообщение "Переменная x глобальная". Таким образом, переменная x является локальной в обработчике события onmouseover и не видна в обработчике события onclick. При щелчке (событие onclick) в соответствующем обработчике остается видимой лишь глобальная переменная x со значением "глобальная".

3.6.3. Операторы присваивания

Ранее мы уже неоднократно упоминали оператор присваивания значения переменной, обозначаемый знаком равенства. Этот оператор (выражение вида *имя_переменной* = *значение*) возвращает значение, совпадающее с тем, которое присваивается переменной. Чтобы убедиться в этом, достаточно выполнить код листинга 3.17.

Листинг 3.17

```
<HTML>
<SCRIPT>
alert(x="Привет!");
</SCRIPT>
</HTML>
```

В результате будет выведено диалоговое окно с сообщением "Привет!".

Оператор присваивания можно использовать каскадно для записи одного и того же значения сразу в несколько переменных, как это показано на рис. 3.11.

Переменной можно присвоить значение другой переменной:

```
x = 2;
y =3;
x = y // значение x равно 3
```

Переменной можно присвоить значение некоторого выражения:

```
y = 2;
x = y + 3 // значение x равно 5
```

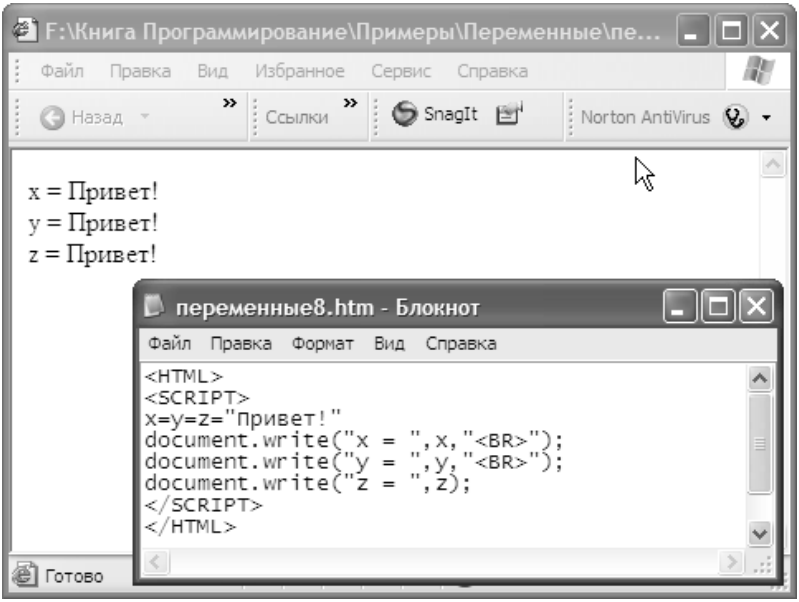

Рис. 3.11. Каскадное использование оператора присваивания значения сразу нескольким переменным

Кроме указанного и наиболее распространенного оператора присваивания в виде знака равенства в JavaScript имеются еще несколько его форм, представляющих собой комбинации оператора присваивания с арифметическими и поразрядными операторами, о которых будет рассказано позже. Это своего рода сокращения, обеспечивающие более экономную запись. Здесь мы рассмотрим лишь наиболее употребительные примеры. Так, в следующем примере значение переменной `x` формируется как результат арифметического сложения текущего значения `x` со значением переменной `y`:

```
x = 2;
y = 3;
x += y // значение x равно 5.
```

Символ `+` в JavaScript обозначает не только арифметическое сложение чисел, но и склейку (конкатенацию) строк. В этом случае его также можно комбинировать с обычным символом присваивания:

```
x = "Здравствуй,";
y = "Вася";
x += y // значение x равно "Здравствуй, Вася".
```

Разумеется, в обоих случаях вместо выражения `x += y` можно написать равносильное по смыслу выражение

```
x = x + y.
```

Для числовых переменных кроме `+=` возможны и такие конструкции: `-=`, `*=`, `/=`, `%=`. Эти операторы выполняют вычитание, умножение, деление и вычисление целого остатка от деления в комбинации с присвоением результата, например,

```
x = 10;
x*=2 // значение равно 20
x%=7 // значение равно 6.
```

3.6.4. Проверка типа переменной

Иногда в сценариях требуется проверить, к какому типу относится данная переменная (точнее, к какому типу относится ее текущее значение). Для этого служит оператор `typeof имя_переменной`, который также можно записать в виде вызова функции `typeof (имя_переменной)`.

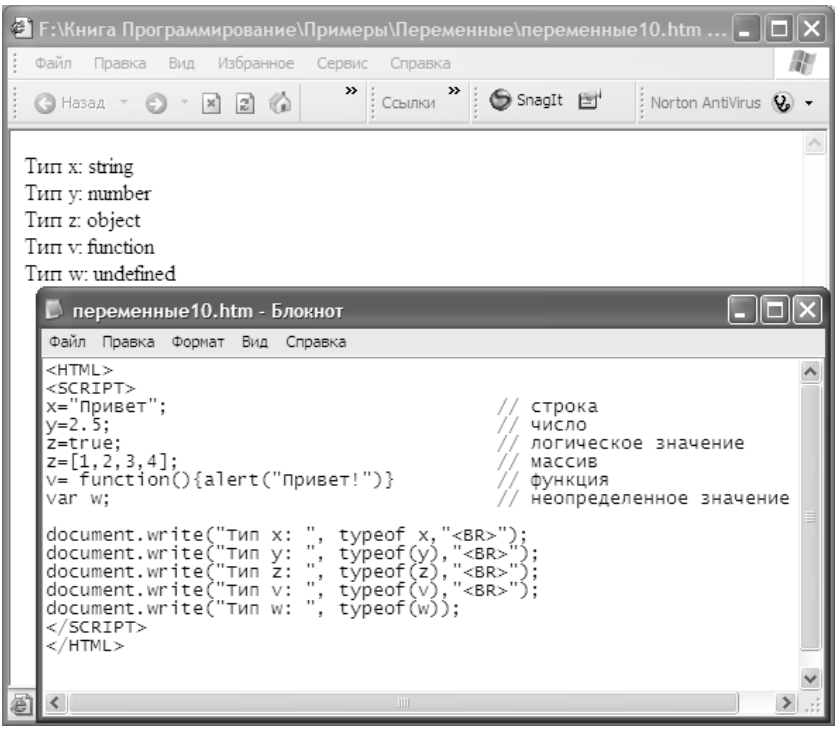

Рис. 3.12. Применение оператора `typeof` для выяснения типа переменной

Пример на рис. 3.12 иллюстрирует использование оператора `typeof`. Данный оператор возвращает строковое значение, указывающее на один из следующих типов значений: `string` (строка), `number` (число), `boolean` (логический тип), `object` (объект), `function` (функция) и `undefined` (неопределенный).

Обратите внимание на значения, возвращаемые оператором `typeof`. Так, в случае массива возвращается `"object"`. И действительно, в JavaScript массив реализован как объект. Однако для функции оператор `typeof` возвращает `"function"`, а не `"object"`, хотя функция также реализована в виде объекта.

3.7. Операторы

Из операторов составляют выражения. Оператор применяется к одному или двум данным, которые в этом случае называются *операндами*. Например, оператор сложения применяют к двум операндам, а оператор логического отрицания — к одному операнду. *Элементарное выражение*, состоящее из операндов и оператора, вычисляется интерпретатором и, следовательно, имеет некоторое значение. В этом случае говорят, что оператор возвращает значение. Например, оператор сложения, примененный к числам 2 и 3, возвращает значение 5. Типы оператора и возвращаемого им значения совпадают. Поскольку элементарное выражение с оператором и операндами возвращает значение, это выражение можно присвоить переменной.

3.7.1. Комментарии

Начнем с самых "безобидных", но важных в практическом программировании операторов комментария. Они позволяют выделить фрагменты кода, которые не выполняются (игнорируются) интерпретатором, а служат лишь для пояснений содержания программы.

В JavaScript, как и во многих других языках, допустимы два вида оператора комментария:

- `//` — одна строка символов, расположенная справа от этого оператора, считается комментарием;
- `/*...*/` — все, что заключено между `/*` и `*/`, считается комментарием; с помощью этого оператора можно выделить несколько строк в качестве комментария.

Примеры

```
x=y +10  // Это однострочный комментарий
// Это тоже однострочный комментарий
z="Привет!" /* Однострочный комментарий справа от выражения */
```

```
/* Это однострочный комментарий слева от выражения */ x=3 + 4
var xyz  /* Это двухстрочный
       комментарий */
```

Комментарии игнорируются браузером при выполнении программы, но передаются и загружаются в него вместе с основным кодом. Иногда они могут составлять даже более половины всего объема программы. Некоторые программисты избегают комментариев, чтобы уменьшить объем кода и сэкономить время передачи и загрузки. Однако даже при значительном объеме кода (несколько десятков килобайтов) польза от наличия комментариев все же больше, чем от их отсутствия, особенно в долгосрочной перспективе.

Не пренебрегайте комментариями в тексте программы. Они помогут при ее отладке и сопровождении. На этапе разработки лучше сначала превратить ненужный фрагмент программы в комментарий, чем просто удалить (а вдруг его придется восстанавливать). Даже опытные программисты, возвращаясь к работе над своей программой через месяц, с большим трудом вспоминают, что к чему. В примерах мы часто будем использовать комментарии.

3.7.2. Арифметические операторы

Арифметические операторы, такие как сложение, умножение и т. д., в JavaScript могут применяться к данным любых типов. Что из этого получается, мы рассмотрим немного позже, а сейчас просто перечислим их (табл. 3.5).

Таблица 3.5. *Арифметические операторы*

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

Действие арифметических операторов лучше всего проиллюстрировать на примерах с числами. Первые четыре оператора широко известны и применительно к числам не вызывают вопросов. Оператор деления по модулю возвращает целочисленный остаток от деления первого числа на второе. Например, `8%3` возвращает 2, поскольку 8 = 3*2 + 2.

Символ "-" служит не только бинарным (для двух операндов) оператором вычитания, но и указывает на то, что число является отрицательным, например, –2.5.

Операторы ++ и -- сочетают в себе действия операторов присваивания и соответственно сложения и вычитания. Выражения X++ и X-- почти эквивалентны выражениям X=X+1 и X=X-1 соответственно. Употребление слова "почти" вскоре прояснится.

Операторы ++ и -- называют соответственно *инкрементом* и *декрементом*. Они имеют префиксную и постфиксную формы записи. Так, x++ — постфиксная запись (++ после операнда), а ++x — префиксная (++ перед операндом). В обоих случаях переменная x увеличит свое значение на единицу, но само выражение x++ вернет предыдущее значение переменной x, а выражение ++x — новое значение. Аналогично, выполнение выражения x-- уменьшает текущее значение переменной x на единицу, но возвращает не новое, а прежнее значение переменной x. Выражение x-- уменьшает x на единицу и возвращает новое значение. Таким образом, ++x полностью эквивалентно x=x+1, а --x полностью эквивалентно x=x-1. Напомню, что выражения x=x+1 и x=x-1 изменяют текущее значение переменной и возвращают новое ее значение, т. е. их выполнение в точности совпадает с выполнением операторов соответственно ++x и --x. Однако операторы x++ и x-- лишь изменяют значение переменной x, но возвращают не вновь полученное значение, чем, собственно, и отличаются от x=x+1 и x=x-1 соответственно. Сказанное иллюстрирует пример на рис. 3.13.

Как уже отмечалось, по отношению к числам действие арифметических операторов обычно не вызывает недоразумений. Однако формально ничто не мешает нам применить эти операторы к данным других типов. В случае строковых данных оператор сложения дает в результате строку, полученную путем приписывания справа второй строки к первой. Например, выражение "Саша"+"Маша" возвращает в результате "СашаМаша". Поэтому для строк оператор сложения называется оператором склейки или конкатенации. Остальные арифметические операторы для строк дают результат NaN — значение, не являющееся числом. Когда оператор сложения применяется к строке и числу, интерпретатор переводит число в соответствующую строку и далее действует как оператор склейки двух строк.

Примеры

```
x="Саша"     // значение переменной x равно "Саша"
y="Маша"     // значение переменной y равно "Маша"
z=x+" "+y    // значение переменной z равно "Саша Маша"
z=x+5        // значение переменной z равно "Саша5"
n="20"+5     // значение переменной n равно "205", а не 30 или "30"
```

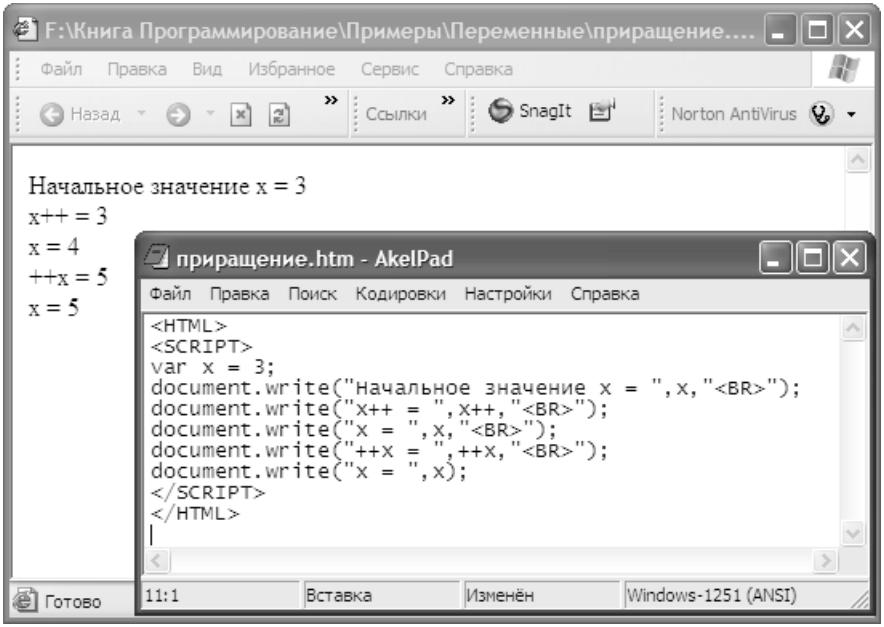

Рис. 3.13. Различия между префиксной и постфиксной записями оператора ++

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

Примеры

```
true + true   // возвращает 2
true + false  // возвращает 1
true * true   // возвращает 1
true/false    /* возвращает Infinity (неопределенность, т. к. на
              ноль делить нельзя) */
true + 5      // возвращает 6
false + 5     // возвращает 5
true * 5      // возвращает 5
true/5        // возвращает 0.2
```

Если один операнд строкового типа, а другой — логического, то при сложении интерпретатор переводит оба операнда в строковый тип и возвращает

строку символов, а в случае других арифметических операторов он переводит оба операнда в числовой тип.

Примеры

```
"Вася" + true   // возвращает "Васяtrue"
"5" + true      // возвращает "5true"

"Вася" * true   /* возвращает NaN (т. е. значение,
                не являющееся числом) */
"5" * true     // возвращает 5
"5" * false    // возвращает 0
"5"/true       // возвращает 5
```

3.7.3. Дополнительные операторы присваивания

В *разд. 3.6.3* мы уже говорили, что кроме обычного оператора присваивания = имеются еще пять дополнительных, сочетающие присваивание и арифметические действия. Дополнительные операторы присваивания мы не будем здесь подробно рассматривать, а лишь перечислим их (табл. 3.6) и приведем примеры.

Таблица 3.6. *Дополнительные операторы присваивания*

Оператор	Пример	Эквивалентное выражение
+=	X+=Y	X=X+Y
-=	X-=Y	X=X-Y
*=	X*=Y	X=X*Y
/=	X/=Y	X=X/Y
%=	X%=Y	X=X%Y

Примеры

```
x=3;
x+=2;    // значение x равно 5 (т. е. 3+2)
x*=4     // значение x равно 20 (т. е. 5*4)
x/=2     // значение x равно 10 (т. е. 20/2)
x%=3     // значение x равно 1 (т. е. 10%3)
```

3.7.4. Операторы сравнения

В программах часто приходится сравнивать значения и проверять выполнение каких-либо условий. Например, в сценариях Web-страниц нередко проверяется, какой браузер установлен у пользователя: Microsoft Internet Explorer или Netscape. В зависимости от результата процесс дальнейших вычислений может пойти по тому или другому пути. Проверяемые условия формируются на основе операторов сравнения, таких как "равно", "меньше", "больше" и т. п. (табл. 3.7). Результат вычисления элементарного выражения, содержащего оператор сравнения и операнды (сравниваемые данные), — логическое значение, т. е. `true` или `false`. Так, если выражение сравнения выполняется (верно, справедливо, истинно), то возвращается `true`, в противном случае — `false`.

Таблица 3.7. *Операторы сравнения*

Оператор	Название	Пример
`==`	Равно	`X==Y`
`===`	Равно и того же типа (строгое равенство)	`X===Y`
`!=`	Не равно	`X!=Y`
`!==`	Не равно или другого типа (строгое неравенство)	`X!==Y`
`>`	Больше, чем	`X*Y`
`>=`	Больше или равно (не меньше)	`X>=Y`
`<`	Меньше, чем	`X<Y`
`<=`	Меньше или равно (не больше)	`X<=Y`

Примечание

Различные варианты равенства записывают с помощью двух или трех символов = без пробелов между ними. Одиночный символ = обозначает, как известно, оператор присваивания. Новички часто совершают трудновыявляемую ошибку, указывая в выражении сравнения одиночный символ равенства. Если такое выражение фигурирует в качестве условия в операторе управления (например, `if` или `while`), то может оказаться, что оно всегда будет возвращать одно и то же значение истинности. В этих операторах должно быть выражение логического типа, а если это не так, то интерпретатор автоматически приведет его к логическому типу. Так например, выражение `Boolean(x=5)` всегда возвращает `true`, а `Boolean(x=null)` — `false`.

Числам присущ порядок: для любых двух различных вещественных чисел можно указать, какое из них больше другого. Операторы сравнения применительно к числам во всех языках программирования понимаются так же, как и в математике. Например, выражение `2<5` вернет `true`, а выражение `4==3.14` — `false`.

Сравнение строк несколько сложнее, оно происходит путем последовательного анализа символов слева направо с учетом их регистра, а также длин сравниваемых строк. Общие правила таковы:

- короткая строка меньше более длинной;
- символы верхнего регистра меньше символов нижнего;
- буквы, расположенные в алфавите раньше, меньше букв, расположенных позже;
- символы с меньшими ASCII- или Unicode-значениями меньше символов с бóльшими значениями этих кодов.

Мы не приводим здесь таблиц кодов ASCII и Unicode для всех символов. Отметим лишь, что некоторые из них упорядочены по возрастанию ASCII-кодов следующим образом: сначала идет пробел, затем в порядке возрастания цифры от 0 до 9, символ подчеркивания, латинские и кириллические буквы по алфавиту (сначала прописные, а затем строчные).

Примеры

```
"abcd" == "abc"      // возвращает false
"abc" == "abcd"      // возвращает false
"abcd" == "abcd"     // возвращает true
"abcd" !== "abcd"    // возвращает false
"abcd" == " abcd"    /* возвращает false
                     (1-й символ 2-го операнда — пробел) */
"" == " "            // возвращает false
"A" < "a"            // возвращает true
" " < "A"            // возвращает true
"" < " "             // возвращает true
"abcd" > " abcd"     // возвращает true
"abc" < "abcd"       // возвращает true
"235ab" < "abcdxyz" // возвращает true
"235xyz" < "abc"     // возвращает true
```

Логические значения сравниваются между собой так же, как и числа 1 и 0 (`true` соответствует 1, а `false` — 0). Например, выражение `true >false` вернет `true`, а выражение `true!==true` вернет `false`.

Операторы сравнения могут быть применены и к разнотипным данным. Например, можно сравнивать числа со строками, строки с логическими значениями и логические значения с числами. В таких случаях нечисловые данные сначала автоматически приводятся к числовому типу, а затем сравниваются полученные результаты. При приведении строки к числовому типу не всегда получается число, иногда возвращается константа `NaN` (например, строка "25рублей" преобразуется в `NaN`, а не в число 25). Все операторы сравнения, кроме `!=` и `!==`, при сравнении числа с константой NaN возвращают `false`, а операторы `!=` и `!==` возвращают `true`. Пустая строка (`""`) или содержащая только пробелы (например, `" "`) преобразуются в число 0.

Примеры

```
57 == "57"        // возвращает true
57 ==="57"        // возвращает false
57 == "57руб."    // возвращает false
57 <= "57руб."    // возвращает false
57 >= "57руб."    // возвращает false
57 != "57руб."    // возвращает true
57 !== "57руб."   // возвращает true
57 > " "          // возвращает true
57 > ""           // возвращает true

false < "57"      // возвращает true
false < "-2.5"    // возвращает false
true >= "0.5"     // возвращает true
false < "57ab"    // возвращает false
false > "57ab"    // возвращает false
true == "true"    // возвращает false
true != "true"    // возвращает true
true <= "true"    // возвращает false
true >= "true"    // возвращает false

true == 1         // возвращает true
true == 0         // возвращает false
false == 0        // возвращает true
false == 1        // возвращает false
true === 1        // возвращает false
false === 0       // возвращает false
true === true     // возвращает true
false === false   // возвращает false
```

Между специальными типами `null` и `undefined` выполняется равенство (`null==undefined` возвращает `true`), но не выполняется строгое равенство (`null===undefined` возвращает `false`).

3.7.5. Логические операторы

Логические данные, обычно получаемые с помощью элементарных выражений, содержащих операторы сравнения, можно объединять в более сложные конструкции, используя логические (булевские) операторы И, ИЛИ, НЕ (табл. 3.8). Например, нам может потребоваться сформировать сложное логическое условие:

```
"x<= 30 и y>10, или name==’Вася’".
```

В этом примере есть и операторы сравнения, и логические операторы. Выражения с логическими операторами возвращают значение `true` или `false` в зависимости от значений операндов.

Таблица 3.8. *Логические операторы*

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

Оператор отрицания `!` изменяет значение единственного операнда на противоположное: если `X` имеет значение `true`, то `!X` возвращает `false`, и наоборот.

В табл. 3.9 указаны результаты операторов И, ИЛИ при различных логических значениях двух операндов. По существу данная таблица является определением логических операций И (`&&`) и ИЛИ (`||`). Кратко суть этих операций можно сформулировать так: `X&&Y` истинно (возвращает `true`) только тогда, когда оба операнда, `X` и `Y`, истинны (имеют значение `true`); `X||Y` истинно только тогда, когда истинен хотя бы один из операндов, `X` или `Y`.

Таблица 3.9. *Значения, возвращаемые операторами И, ИЛИ*

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

Операторы `&&` и `||` еще называют *логическим умножением* и *логическим сложением* соответственно (в математике логические операции И и ИЛИ известны под названиями *конъюнкции* и *дизъюнкции*).

Если вспомнить, что значению `true` можно сопоставить единицу, а `false` — ноль, то нетрудно понять, как вычисляются элементарные выражения с логическими операторами. Нужно только учесть, что в алгебре логики 1+1=1 (а не 2). Аналогично, оператору отрицания соответствует вычитание из единицы числового эквивалента логического значения операнда.

Примеры

```
x=true                    // значение переменной x равно true
x=false || 2*2==4         // значение переменной x равно true
x=5<2 || "abcd" <="xy"    // значение переменной x равно true
y=!x                      // значение переменной y равно false
z=x && y                  // значение переменной z равно false
z=x || y                  // значение переменной z равно true
```

Сложные логические выражения, состоящие из нескольких более простых, соединенных операторами И и ИЛИ, выполняются по так называемому *принципу короткой обработки*. Дело в том, что значение всего выражения зачастую можно определить, вычислив лишь одно или несколько его составляющих, не затрагивая остальные. Например, выражение `x&&y` вычисляется слева направо; если значение `x` оказалось равным `false`, то значение `y` не вычисляется, поскольку и так известно, что все выражение равно `false` (см. табл. 3.9). Аналогично, если в выражении `x||y` значение `x` равно `true`, то `y` не вычисляется, поскольку уже ясно, что все выражение равно `true`. Данное обстоятельство требует особого внимания при тестировании сложных логических выражений. Так, если какая-нибудь логическая конструкция содержит ошибку, то она может остаться невыявленной, поскольку эта часть выражения просто не выполнялась при тестировании.

3.7.6. Операторы условия

Вычислительный процесс можно направить по тому или другому пути в зависимости от того, выполняется ли некоторое условие или нет. Для управления вычислениями служат операторы условия или, как еще говорят, операторы условного перехода, важнейшие из них — `if` (если), `?` и `switch` (переключатель).

Оператор *if*

Оператор `if` (если), присутствующий во всех языках программирования, позволяет реализовать структуру условного выражения:

если ..., то ..., иначе ...

Синтаксис оператора `if` следующий:

```
if (условие)
 {код, который выполняется, если условие истинно}
else
 {код, который выполняется, если условие ложно}
```

В фигурных скобках располагают блок кода: одно или несколько выражений. Если в блоке не более одного выражения, то фигурные скобки можно опустить. Часть этой конструкции, определяемая ключевым словом `else` (иначе), необязательна. В этом случае остается только часть, определенная ключевым словом `if`:

```
if (условие)
 {код, который работает, если условие истинно}
```

Конструкция оператора условного перехода допускает вложение других операторов условного перехода внутри блока кода, расположенного в фигурных скобках. Условие обычно представляет собой логическое выражение, значение которого есть `true` или `false`. Чаще всего это элементарные выражения с операторами сравнения.

Пример

```
/* Вывод диалогового окна с тем или иным сообщением в зависимости от значения переменной age (возраст) */
var age=15;
if (age<18)
 {alert("Вы слишком молоды для просмотра этого сайта")}
else
 {alert("Подтвердите свое решение заглянуть на этот сайт")}
/* Вывод диалогового окна с сообщением, если только значение переменной age меньше 18 */
if (age<18)
 {alert("Вы слишком молоды для просмотра этого сайта")}
```

Делать ли отступы при написании операторов, где располагать фигурные скобки — дело вкуса. Следует руководствоваться наглядностью и ясностью

структуры, при которой легко проверить правильность расстановки скобок. Кроме описанного ранее способа, часто встречается еще и такой:

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

Возможно, этот способ записи лучше отражает структуру оператора условного перехода.

Более сложная структура оператора условного перехода получается при вложении других операторов `if` (листинг 3.18).

Листинг 3.18

```
if (условие1) {
 код, который работает, если условие1 выполнено
} else { if (условие2){
  код, который работает, если условие2 выполнено
 }else{
  код, который работает, если условие2 не выполнено
 }
}
```

Ранее уже отмечалось, что условие в операторе `if` чаще всего является логическим выражением. Однако это может быть также строковое или числовое выражение, массив и даже объект. В общем случае интерпретатор приводит выражение условия к логическому типу автоматически *(см. разд. 3.5.3)*. Для строкового выражения условие считается истинным, если его значением будет непустая строка. Напомним, что пустая строка `""` не содержит ни одного символа, в том числе и пробела (строка, содержащая хотя бы один пробел, не пуста). Числовое выражение истинно, если его значением является число, отличное от нуля. Для массива это условие выполняется, если длина массива больше нуля. Хотя новичка подобная многозначность условий может обескуражить, если не просто ввергнуть в отчаяние, во многих случаях она оказывается очень удобной. Виртуозы легко и свободно используют эту возможность. Нужно просто привыкнуть к ней. Ортодоксам могу посоветовать записывать в качестве условия выражения явно логического типа либо принудительно приводить их к логическому типу *(см. разд. 3.5.4)*.

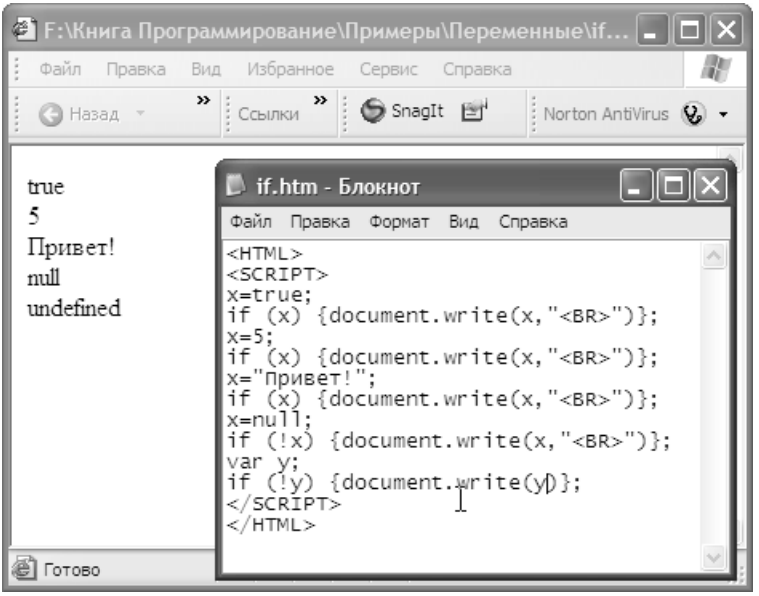

Рис. 3.14. Различные типы данных в качестве условия оператора `if`

На рис. 3.14 приведено несколько примеров использования значений различных типов в качестве условий оператора `if`.

В следующем примере (листинг 3.19) на Web-странице пользователю предоставляется ввести что-нибудь в поле ввода, а сценарий анализирует введенное значение.

Листинг 3.19

```
<HTML>
<INPUT type="text" value="" ID="myinput">
<SCRIPT>
var x=document.myinput // значение в поле ввода
if (!x){
    alert("Вы ничего не ввели") // вывод указанного сообщения
}
</SCRIPT>
</HTML>
```

Поскольку код выполняется сразу же после загрузки, не дожидаясь, когда пользователь успеет что-либо ввести, то сценарий сразу срабатывает: он об-

наруживает, что поле ввода пусто, а потому выводит с помощью метода `alert` сообщение "Вы ничего не ввели".

А каково действительное значение переменной `x` в данном сценарии, если пользователь ничего не ввел в поле, созданное HTML-кодом? Это значение не определено, т. е. равно `undefined`, в чем можно убедиться, выполнив код листинга 3.20.

Листинг 3.20

```
<HTML>
<INPUT type="text" value="" ID="myinput">
<SCRIPT>
var x=document.myinput // значение в поле ввода
if (!x){
       alert(x) // вывод сообщения "undefined">
}
</SCRIPT>
</HTML>
```

Итак, если пользователь ничего не ввел в поле, созданное HTML-кодом, то его значение остается для JavaScript неопределенным, т. е. `undefined`. Понимание этого обстоятельства очень важно при организации обработки данных форм, в элементы которых пользователь может ввести что-нибудь или просто проигнорировать.

Оператор условия *?:*

Этот оператор — сокращенная форма записи рассмотренного ранее оператора `if...else...`. Обычно он применяется вместе с оператором присваивания одного из двух возможных значений, в зависимости от истинности или ложности условного выражения. Если это выражение само по себе возвращает значения нелогического типа, то интерпретатор автоматически попытается выполнить приведение к логическому типу. Синтаксис оператора условия следующий:

условие ? *выражение1* : *выражение2*

Оператор условия возвращает значение выражения *выражение1*, если *условие* истинно, в противном случае — значение выражения *выражение2*.

Выражение с оператором условия можно использовать в выражении присваивания:

переменная=условие ? *выражение1* : *выражение2*

Выражение, содержащее условие, автоматически приводится к логическому типу.

Примеры

```
var x=2;
alert(x==2 ? "x равно 2": "x не равно 2") // вывод сообщения "x равно 2"

var x=2;            // x имеет значение 2
y = x==2 ? "x равно 2": "x не равно 2";   /* переменная y имеет значение
"x равно 2" */
alert(y)

var x="";             // x имеет значение "" (пустая строка)
y=x==2 ? "x равно 2": "x не равно 2"   // вывод сообщения "x не равно 2";
alert(y)              // вывод сообщения "x не равно 2"
```

Оператор *switch*

Для организации управления вычислениями с несколькими условиями вполне достаточно рассмотренного оператора `if`. Однако в случае однородных повторяющихся условий более удобным и наглядным оказывается оператор `switch` (переключатель). Он особенно целесообразен, если требуется проверить несколько условий, которые не являются взаимоисключающими.

Листинг 3.21 иллюстрирует синтаксис оператора `switch`.

Листинг 3.21

```
switch (выражение) {
 case вариант1:
  код
  [break]
 case вариант2:
  код
  [break]
 ...
 [default:
  код]
}
```

Параметр *выражение* оператора switch может принимать строковые, числовые и логические значения. Разумеется, в случае логического выражения возможны только два варианта. Ключевые слова (операторы) break и default необязательные (на это указывают прямоугольные скобки). Если оператор break указан, то остальные условия не проверяются. Если указан оператор default, то следующий за ним код выполняется тогда, когда значение выражения *выражение* не соответствует ни одному из вариантов. Если все варианты возможных значений предусмотрены в операторе switch, то оператор default можно опустить.

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

Пример 1

```
x=2
switch (x) {
 case 1:
  alert(1)
 case 2:
  alert(2)
 case 3:
  alert(3)
}
```

В первом примере сработают 2-й и 3-й варианты. Если мы хотим, чтобы сработал только один какой-нибудь вариант (только тот, который соответствует значению выражения), то нужно добавить оператор break.

Пример 2

```
x=2
switch (x) {
 case 1:
  alert(1);
```

```
  break
 case 2:
  alert(2);
  break
 case 3:
  alert(3);
  break
}
```

Во втором примере сработает только 2-й вариант.

3.7.7. Операторы цикла

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

Оператор *for*

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

```
for ([начальное_выражение]; [условие]; [выражение_обновления])
{
 код
}
```

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

```
for ([начальное_выражение]; [условие]; [выражение_обновления]) {
 код
}
```

Все, что находится в круглых скобках справа от ключевого слова `for`, называется *заголовком* оператора цикла, а содержимое фигурных скобок — его *телом*.

В заголовке оператора цикла начальное выражение выполняется только один раз в начале выполнения оператора. Второй параметр представляет собой

условие продолжения цикла. Он аналогичен условию в операторе `if`. Третий параметр содержит выражение, которое выполняется после выполнения всех выражений кода, заключенного в фигурные скобки.

Оператор цикла работает следующим образом. Сначала выполняется *начальное_выражение*. Затем проверяется *условие*. Если оно истинно, то оператор цикла прекращает работу (при этом *код* не выполняется). В противном случае выполняется *код*, расположенный в теле оператора `for`, т. е. между фигурными скобками. После этого выполняется *выражение_обновления* (третий параметр оператора `for`). Так заканчивается первый цикл или, как еще говорят, первая *итерация* цикла. Далее, снова проверяется *условие*, и все повторяется описанным выше способом.

Обычно начальным выражением служит оператор присваивания значения переменной, например, `i=0` или `var i=0`. Имя переменной и присваиваемое значение могут быть любыми. Эту переменную называют *счетчиком* цикла. В этом случае условие, как правило, представляет собой элементарное выражение сравнения переменной счетчика цикла с некоторым числом, например, `i<=nMax`. Выражение обновления в таком случае просто изменяет значение счетчика, например, `i=i+1` или более компактно `i++`.

В следующем примере оператор цикла просто изменяет значение своего счетчика, выполняя 15 итераций:

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

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

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

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

Рассмотрим еще пример. Допустим, требуется вычислить `x` в степени `y`, где `x`, `y` — целые положительные числа. Алгоритм решения этой задачи прост: нужно вычислить выражение `x*x*x*...*x`, в котором сомножитель `x` встречается `y` раз. Очевидно, что если `y` не превышает 10, то можно воспользоваться оператором умножения: выражение будет не очень длинным. А как быть, если `y` равно нескольким десяткам или сотням? Ясно, что в общем случае следует применить оператор цикла (листинг 3.22).

Листинг 3.22

```
/* Вычисляем x в степени y */
var z=x   // z хранит результат
for (i=2; i <= y; i++) {
 z=z*x
}
```

В этом примере результат сохраняется в переменной `z`. Ее начальное значение равно `x`. Если `y` равно 1, то оператор цикла не будет выполняться, поскольку не выполняется условие `2<=1` (обратите внимание на начальное значение счетчика цикла) и, следовательно, мы получим верный результат: `x` в степени 1 равно `x`. Если `y` равно 2, то оператор цикла выполнит одну итерацию, вычислив выражение `z=z*x` при `z`, равном `x` (т. е. `z=x*x`). При `y`, равном 3, оператор цикла сделает две итерации. На второй, последней, итерации выражение `z=z*x` вычисляется при текущем значении `z`, равном `x*x`, и, следовательно, становится равным `x*x*x` (т. е. `x` в степени 3).

Рассмотрим довольно традиционный при изучении операторов цикла пример вычисления факториала числа. Факториал числа `n` в математике обозначают как `n!`. Для `n`, равного 0 и 1, `n!` равен 1. В остальных случаях `n!` равен `2*3*4*...*n`. Поскольку возможны два варианта исходных данных, нам потребуется оператор условного перехода. Листинг 3.23 содержит код, решающий эту задачу.

Листинг 3.23

```
/* Вычисляем n! */
var z=1   // z хранит результат n!
if (n >1) {
 for (i=2; i <= n; i++) {
  z=z*i
 }
}
```

Для принудительного (т. е. не по условию) выхода из цикла используется оператор `break` (прерывание). Если вычислительный процесс встречает этот оператор в теле оператора цикла, то он сразу же завершается без выполнения последующих выражений кода в теле и даже выражения обновления. Обычно оператор `break` применяется при проверке некоторого дополнительного условия, выполнение которого требует завершения итераций, несмотря на то, что условие в заголовке цикла еще не выполнено. Типовая структура оператора цикла с `break` приведена в листинге 3.24.

Листинг 3.24

```
for ([начальное_выражение]; [условие1]; [выражение_обновления])
{
 код
 if (условие2){
  код
  break
 }
 код
}
```

Для управления вычислениями в операторе цикла допустим также оператор `continue` (продолжение). Как и `break`, этот оператор применяется в теле оператора цикла вместе с оператором условного перехода. Однако в отличие от `break`, оператор `continue` прекращает выполнение последующего кода, выполняет выражение обновления и возвращает вычислительный процесс в начало оператора цикла, где проверяется условие, указанное в заголовке. Листинг 3.25 иллюстрирует типовую структуру оператора цикла с `continue`.

Листинг 3.25

```
for ([начальное_выражение]; [условие1]; [выражение_обновления] )
{
 код
 if (условие2){
  код
  continue
 }
 код
}
```

Оператор *while*

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

```
while (условие)
{
 код
}
```

При выполнении этого оператора сначала проверяется условие, указанное в заголовке, т. е. в круглых скобках справа от ключевого слова `while`. Если оно истинно, то выполняется код в теле оператора цикла, заключенный в фигурные скобки. В противном случае код не выполняется. При выполнении кода (завершении первой итерации) вычислительный процесс возвращается к заголовку, где снова проверяется условие, и т. д.

Если сравнивать операторы `while` и `for`, то особенность первого заключается в том, что выражение обновления записывается в теле оператора, а не в заголовке. Часто забывают указать это выражение, и в результате цикл не завершается (программа "зависает").

Приведем два примера, которые мы уже рассматривали при описании оператора цикла `for`.

Примеры с оператором `while`

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

/* Ввычисляем n! */
var z=1   // z хранит результат n!
if (n >1) {
 i=2;
 while ( i <= n ) {
  z=z*I;
  i++;
 }
}
```

Во всех приведенных примерах есть счетчик цикла, который инициируется заранее, до оператора цикла и обновляется в теле цикла.

Для управления вычислительным процессом в операторе `while`, так же как и в операторе `for`, допустимы операторы прерывания `break` и продолжения `continue`.

Внимание

Если в `while` вы задействуете счетчики циклов, то будьте осторожны, применяя `break` и `continue`.

Оператор *do-while*

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

```
do {
 код
}
while (условие)
```

В отличие от `while` в операторе `do-while` код выполняется хотя бы один раз, независимо от условия, которое проверяется после выполнения кода. Если оно истинно, то снова выполняется код в теле оператора `do`. В противном случае работа оператора `do-while` завершается. Напомню, что в операторе `while` условие проверяется в первую очередь, до выполнения кода в теле цикла. Если при первом обращении к оператору `while` условие ложно, то код не будет выполнен никогда.

Рассмотрим те же две задачи, которые мы уже решали ранее с помощью операторов цикла `for` и `while`.

Примеры с оператором `do-while`

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

/* Вычисляем n! */
var z=1;   // z хранит результат n!
if (n >1) {
 i=2;
 do {
```

```
  z=z*I;
  i++;
 }
 while ( i <= n )
}
```

3.7.8. Об условиях в операторах условия и цикла

В условных операторах *(см. разд. 3.7.6)* и циклах *(см. разд. 3.7.7)* используются выражения условия, которые должны иметь логический тип, т. е. возвращать либо `true`, либо `false`. Если же эти выражения возвращают значения других типов (например, числа, строки, массивы и т. д.), то при наличии их в операторах условия или цикла интерпретатор JavaScript сделает попытку автоматически привести их значения к логическому типу. Так, число 0, пустую строку "" и массив, не содержащий элементов, он преобразует в логическое значение `false`.

Пример

```
var x=0; // число 0
var y=""; // пустая строка
var z=["Саша", "Маша", "Петя"]; // массив

if (x) alert("Переменная x >0")
else alert("Переменная x =0")

if (y) alert("Переменная y что-то содержит")
else alert("Переменная y пуста")

if (z) alert("Массив z не пуст")
else alert("Массив z пуст")
```

3.7.9. Побитовые операторы

Побитовые (поразрядные) операторы применяются к целочисленным значениям и возвращают результат такого же типа. При их выполнении операнды предварительно приводятся к двоичной форме, в которой число представляет собой 32-разрядную последовательность из нулей и единиц, называемых *двоичными разрядами* или *битами*. Далее производится некоторое действие над битами, в результате которого получается новая последовательность битов. В конце концов, эта последовательность битов преобразуется к обычному

целому числу — результату побитового оператора. В табл. 3.10 приведен список побитовых операторов.

Таблица 3.10. *Побитовые операторы*

Оператор	Название	Левый операнд	Правый операнд
`&`	Побитовое И	Целое число	Целое число
`|`	Побитовое ИЛИ	То же	То же
`^`	Побитовое исключающее ИЛИ	То же	То же
`~`	Побитовое НЕ	—	То же
`<<`	Смещение влево	Целое число	Количество битов, на которое производится смещение
`>>`	Смещение вправо	То же	То же
`>>>`	Заполнение нулями при смещении вправо	То же	То же

Операторы `&`, `|`, `^` и `~` напоминают логические операторы, но их область действия — биты, а не логические значения. Оператор `~` изменяет значение бита на противоположное: 0 на 1, а 1 — на 0. Действие операторов `&`, `|`, `^` ясно из табл. 3.11.

Таблица 3.11. *Действие операторов* `&`, `|`, `^`

`X`	`Y`	`X&Y`	`X|Y`	`X^Y`
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Например, `2&3` равно 2, а `2|3` равно 3. Действительно, в двоичном представлении 2 есть 10, а 3 — 11. Применение побитового оператора `&` даст двоичное число 10 (десятичное число 2), а оператора `|` — двоичное число 11 (десятичное 3).

У операторов смещения один операнд и один параметр, указывающий, на сколько битов выполняется смещение. Например, `3<<2` равно 12, потому что смещение влево на два бита двоичного числа 11 (десятичное 3) дает 1100, что

в десятичной форме есть 12. Результат вычисления выражения `6>>2` равен 1. Действительно, число 6 в двоичной форме — это 110; смещение его вправо на два бита дает 1 как в двоичной, так и в десятичной форме.

3.7.10. Другие операторы

Кроме перечисленных, имеется и ряд других операторов, которые мы рассмотрим далее в этой главе. Так, операторы `with`, `for…in`, `in`, `delete` и `instanceof` для работы с объектами будут рассмотрены в *разд. 3.13.3*, а операторы обработки ошибок (исключительных ситуаций) — в *разд. 3.14*.

3.7.11. Приоритет операторов

В одном и том же выражении могут участвовать несколько операторов, которые выполняются в соответствии с их приоритетами. Операторы с одинаковыми приоритетами выполняются слева направо. Для изменения данного порядка служат круглые скобки. Круглые скобки, позволяющие изменить порядок выполнения операций, можно тоже считать операторами, обладающими наивысшим приоритетом. Скобки могут образовывать вложенные структуры. Выражения, находящиеся во внутренних круглых скобках, выполняются раньше тех, которые заключены во внешние круглые скобки. Интерпретатор JavaScript начинает анализ выражения именно с выяснения структуры вложенности пар круглых скобок.

На втором месте по приоритету находится вычисление индексов и опредедение элементов массивов. Индексы массивов заключаются в квадратные скобки.

Третье место по приоритету занимает вызов функции.

На последнем месте находится запятая, как разделитель параметров или элементов массива, перечисленных в квадратных скобках.

Распределение операторов по приоритетам приведено в табл. 3.12. Некоторые из указанных в ней операторов мы уже рассмотрели, о других будет рассказано в следующих разделах данной главы.

Таблица 3.12. *Распределение операторов по приоритетам*

Приоритет	Оператор	Комментарий
1	`( )`	От внутренних к внешним
	`[ ]`	Значение индекса массива
	`function()`	Вызов функции

Таблица 3.12 *(продолжение)*

Приоритет	Оператор	Комментарий
2	!	Логическое НЕ
	~	Побитовое НЕ
	-	Отрицание
	++	Инкремент (приращение)
	--	Декремент
	new	
	typeof	
	void	
	delete	Удаление объектного элемента
3	*	Умножение
	/	Деление
	%	Деление по модулю (остаток от деления)
4	+	Сложение (конкатенация)
	-	Вычитание
5	<<	Побитовые сдвиги
	>>	
	>>>	
6	<	Меньше
	<=	Не больше (меньше или равно)
	>	Больше
	>=	Не меньше (больше или равно)
7	==	Равенство
	===	Строгое равенство (с учетом типов)
	!=	Неравенство
	!==	Строгое неравенство (с учетом типов)
8	&	Побитовое И
9	^	Побитовое исключающее ИЛИ
10	|	Побитовое ИЛИ (дизъюнкция)
11	&&	Логическое И (конъюнкция)

Таблица 3.12 *(окончание)*

Приоритет	Оператор	Комментарий
12	`||`	Логическое ИЛИ
13	`?`	Условное выражение (оператор условия)
14	`=`	Операторы присваивания
	`+=`	
	`-=`	
	`*=`	
	`/=`	
	`%=`	
	`<<=`	
	`>=`	
	`>>=`	
	`&=`	
	`^=`	
	`|=`	
15	`,`	Запятая

3.8. Функции

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

Программный код функции заключают в фигурные скобки, а перед ним пишут ключевое слово `function`, за которым следуют круглые скобки, обрамляющие список параметров. Иначе говоря, определение функции выглядит так:

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

Формат вызова функции в программе следующий:

`имя_функции``(`*`параметры`*`).`

Параметры функции могут присутствовать или отсутствовать. Если параметры требуются, то их указывают в круглых скобках через запятую. Если параметры не предусмотрены, то в круглых скобках ничего не записывают.

3.8.1. Встроенные функции

В JavaScript имеются встроенные функции, которые в действительности являются методами встроенного объекта `global`. Поскольку при вызове этих методов не нужно указывать объект `global` (т. е. пишут `метод()`, а не `global.метод()`), то их называют функциями. Некоторые из них уже рассматривались ранее, например, `parseInt()`, `parseFloat()`, `isNaN()` *(см. разд. 3.5.4)*. Далее в этом разделе рассмотрим другие встроенные функции.

- `eval(`*`строка`*`)` — вычисляет выражение в указанной строке. Выражение должно быть написано на языке JavaScript (не содержит тегов HTML).

Примеры

```
var y=3;           // значение y равно 3
var x="if(y<10) {y=y+2}";  // значение x равно строке символов
eval(x);           // значение y равно 5
```

Другой, более интересный, пример применения функции `eval()` — создание Web-страницы, содержащей текстовые области для ввода и отображения результатов выполнения выражений JavaScript. По существу это простой редактор скриптов, позволяющий их выполнять. Текст соответствующего HTML-кода со сценарием, содержащим функцию `eval()`, приведен в листинге 3.26.

Листинг 3.26. Редактор JavaScript

```
<HTML>
 <HEAD><TITLE>Редактор JavaScript</TITLE></HEAD>
 <BODY bgcolor="#a0a0a0">
 Введите скрипт:<BR>
 <TEXTAREA ID="mycode" ROWS=10 COLS=60></TEXTAREA>
 <BR>Результат:<BR>
 <TEXTAREA ID="myrezult" ROWS=3 COLS=60></TEXTAREA>
 <P>
 <BUTTON onclick =
 "document.getElementById('myrezult').value = eval(mycode.value)">
  Выполнить
 </BUTTON>
 <BUTTON onclick = "document.getElementById('mycode').value='';
```

```
        document.getElementById('myrezult').value=''">
  Очистить
 </BUTTON >
 </BODY>
</HTML>
```

Здесь сценарии записаны в виде символьных строк в качестве значений атрибутов `onclik`, определяющих событие "Щелчок кнопкой мыши" на кнопках, которые заданы тегами `<BUTTON>`. На рис. 3.15 показан вид приведенного HTML-документа в браузере.

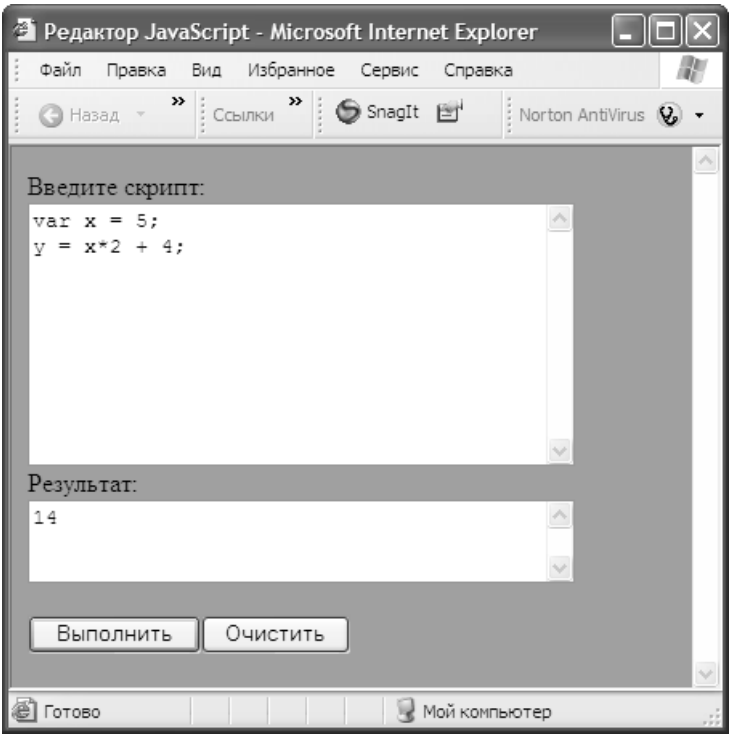

Рис. 3.15. Редактор кодов JavaScript, созданный с помощью функции `eval()`

- `escape(`*строка*`)` — возвращает строку в виде `%`*XX*, где *XX* — ASCII-код указанного символа (такую строку еще называют escape-последовательностью).

При взаимодействии браузеров и серверов протоколы обмена данными позволяют передавать не все символы в их естественном виде. Для передачи остальных символов используют их шестнадцатеричные ASCII-коды, перед

которыми указывают символ %. Например, пробел представляют в escape-последовательности как %20.

Примеры

```
escape("How do you do")  /* значение равно "How%20do%20you%20do" */
escape("Привет")       /* значение равно
              %u041F%u0440%u0438%u0432%u0435%u0442 */
```

- unescape(*строка*) — осуществляет обратное преобразование.
- typeof(*объект*) — возвращает тип указанного объекта в виде символьной строки; например, "boolean", "function" и т. п. В *разд. 3.6.4* рассматривался оператор typeof, который возвращает то же, что и встроенная функция typeof().

Пример

```
<HTML>
<SCRIPT>
var x = "Привет!", y = 5, z= function(){alert("Как дела?")};
document.write(typeof(x),"<BR>")
document.write(typeof(y),"<BR>")
document.write(typeof(z),"<BR>")
document.write(typeof x,"<BR>")
document.write(typeof y,"<BR>")
document.write(typeof z,"<BR>")
</SCRIPT>
</HTML>
```

3.8.2. Пользовательские функции

Пользовательские функции вы можете создать сами, по своему усмотрению, для решения собственных задач. Функция задается своим определением (описанием), которое начинается ключевым словом function. Описание функции имеет следующий синтаксис:

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

Имя функции выбирают так же, как и имя переменной. За именем функции обязательно следует пара круглых скобок. Программный код (тело) функции

заключается в фигурные скобки. Если функция принимает параметры, то список их имен (идентификаторов) указывают в круглых скобках около имени функции. Имена параметров выбирают согласно тем же требованиям, что и имена обычных переменных. Если параметров несколько, то в списке их разделяют запятыми. Если параметры для данной функции не предусмотрены, то в круглых скобках около имени функции ничего не пишут.

Когда создается определение функции, список ее параметров (если он необходим) содержит просто формальные идентификаторы (имена) этих параметров, понимаемые как переменные. В определении функции в списке параметров, заключенном в круглые скобки сразу же за именем функции после ключевого слова `function`, недопустимы конкретные значения и выражения. В этом смысле определение функции задает код, оперирующий формальными параметрами, которые конкретизируются лишь при вызове функции из внешней программы.

Если требуется, чтобы функция возвращала некоторое значение, то в ее теле размещают оператор возврата `return` с указанием справа от него того, что следует возвратить. В качестве возвращаемой величины может выступать любое выражение: простое значение, имя переменной или вычисляемое выражение. Оператор `return` может встречаться в коде функции несколько раз. Впрочем, возвращаемую величину, а также сам оператор `return` можно и не указывать. В этом случае функция ничего не будет возвращать.

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

имя_функции(*параметры*)

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

В JavaScript возможно несоответствие числа параметров в определении функции и в ее вызове. Если в функции определены, например, три параметра, а в вызове указаны только два, то последнему параметру будет автоматически присвоено значение `null`. Наоборот, лишние параметры в вызове функции будут просто проигнорированы.

Внутри тела функции можно создавать переменные либо оператором присваивания, либо с помощью ключевого слова `var`. При этом возникает вопрос об области действия переменных, который мы рассматривали в *разд. 3.6.2*.

Если вы используете просто оператор присваивания, то возможны две ситуации:

- Переменная в операторе присваивания встречается первый раз в вашей программе именно в теле функции. В этом случае она действует в пределах кода (тела) данной функции, но после выполнения функции эта переменная продолжает существовать и во внешней программе. Таким образом, эта переменная *глобальная*.
- Переменная в операторе присваивания уже определена во внешней программе. Тогда она действует также внутри функции и продолжает существовать после завершения ее выполнения, поскольку она была создана во внешней программе как глобальная переменная.

Если в теле функции вы инициализируете переменную с помощью ключевого слова `var`, то эта переменная будет действовать только в пределах кода функции, независимо от того, была ли одноименная переменная определена во внешней программе, или нет. Если переменная создана с ключевым словом `var` впервые в теле функции, то она будет *локальной*, т. е. недоступной из внешней программы. Таким образом, инициализация переменной с помощью выражения с ключевым словом `var` создает локальную переменную.

Если в теле функции вы используете переменную, определенную только во внешней программе, то изменение ее значения в теле функции сохранится даже после завершения выполнения этой функции, поскольку это глобальная переменная.

Если во внешней программе вы определили некоторые переменные, имена которых совпадают с формальными параметрами в определении функции, а затем указываете их в качестве параметров вызова этой функции, то произойдет следующее. Функция воспримет значения параметров, определенных во внешней программе, однако любые изменения их значений в теле функции останутся локальными, т. е. после завершения работы кода функции значения указанных переменных останутся прежними, как до ее вызова.

Программа может содержать и определение функции, и выражения ее вызова. При этом порядок их следования не важен: вы можете сначала написать определение функции, а затем где-нибудь в программе привести ее вызов или, наоборот, написать сначала вызов функции, а ее определение разместить где-нибудь в конце программы или даже в отдельном файле с расширением js. Вместе с тем, если ваш сценарий расположен в нескольких разделах (контейнерах `<SCRIPT>`), то определение функции должно находиться в ранее загружаемом разделе либо в том же разделе, что и вызов данной функции.

В *разд. 3.7.7*, посвященном операторам цикла, мы уже рассматривали программный код вычисления факториала числа `n`. Здесь программу вычисления `n!` мы оформим в виде функции.

Пример функции вычисления факториала `n!`

```
function factorial(n){
 if(n <= 1) return 1
 rezult=2   // result — переменная для результата
 for (i=3; i <=n; i++) {
  rezult=rezult*i
 }
 return result
}
```

Для вычисления факториала конкретного числа, например 12, следует записать выражение `factorial(12)`.

Если в нашей программе определена описанная функция `factorial()`, то вызов этой функции может выглядеть так:

```
var m=10
x=factorial(m)  // значение x равно 3628800
```

Определение функции может содержать вызов этой же функции — это так называемое *рекурсивное определение функции*. В качестве примера рекурсивного определения функции в учебной литературе обычно приводят функцию вычисления факториала.

Пример функции вычисления факториала `n!` с помощью рекурсии

```
function factorial(n){
 if(n <= 1){
  return 1
 }
 return n*factorial(n-1) // вызов функции factorial()
}
```

Чтобы избежать ошибок, старайтесь не использовать в коде функций имена переменных, которые были инициализированы во внешней программе. По возможности стремитесь к тому, чтобы все, что делается внутри вашей функции, было локальным, формально не зависимым от внешнего окружения.

Определение функции может содержать в себе определения других функций, однако такие вложенные функции доступны только из кода функции, содержащей их определения.

3.8.3. Объект *Function*

Определение функции, описанное в *разд. 3.8.2*, создает так называемый экземпляр объекта `Function`, обладающий полезными свойствами и методами. Однако функцию можно определить и непосредственно через объект `Function`:

имя_функции = new Function("*пар1*",… ,"*парN*", "*оператор1*;… ; *операторN*").

Здесь ключевое слово `new` — оператор вызова конструктора объекта `Function`, создающего экземпляр этого объекта. Названия всех параметров являются строковыми значениями и разделяются запятыми. Заключительная строка в списке параметров содержит операторы кода тела функции, разделенные точкой с запятой. Все элементы в круглых скобках необязательные. Имя функции совпадает с именем переменной в операторе присваивания.

Вызов функции, определенной как экземпляр объекта `Function`, можно выполнить обычным способом:

имя_функции(*параметры*)

Примеры

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

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

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

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

Перечислим свойства и метод объекта `Function`.

- `arguments` — массив значений параметров, переданных функции. Элементы массива индексируются начиная с нуля. Поскольку это массив, он имеет свойства и методы объекта `Array` (в частности, свойство `length` — длина массива). Массивы будут подробно рассмотрены в *разд. 3.9*.

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

Синтаксис выражения следующий:

имя_функции`.arguments`

Пример

```
/* Функция, приводимая в этом примере, возвращает строку, содержащую значения параметров и их общее количество, которые были указаны в вызове функции (а не в ее определении!) */
   function myfunc(a, b,c){
    var arglenth=myfunc.arguments.length  /* количество переданных
                         параметров */
    var x="";
    for (i=0; i< myfunc.arguments.length;i++) {
     x += myfunc.arguments[i] + ","
    }
    return x+"Всего: "+ myfunc.arguments.length
   }

   myfunc(5, "Вася")     // "5, Вася, Всего: 2"
   myfunc()         // "Всего: 0"
   myfunc(null,"",0,25)  // "null,,0,25,Всего: 4"
```

- `length` — количество параметров, указанных в определении функции. Синтаксис:

  имя_функции`.length`

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

Пример

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

- `caller` — содержит ссылку на функцию, из которой была вызвана данная функция. Если функция не вызывалась из другой функции, то значение этого свойства равно `null`. Синтаксис:

  имя_функции`.caller`

В свойстве *имя_функции*`.caller` содержится все определение функции, из которой была вызвана функция *имя_функции*.

Пример

```
function f1() {
 f2()
}
function f2(){
 alert(f2.caller)  // диалоговое окно со значением f2.caller
}
f1()
```

Вызов функции `f1()` приведет к вызову функции `f2()`, которая отобразит на экране окно с сообщением, содержащим код определения функции `f1()` (рис. 3.16).

Рис. 3.16. Результат выполнения функции `f1()` — окно со значением `f2.caller`

В листинге 3.27 приведен код функции, которая принимает произвольное количество параметров и возвращает их сумму.

Листинг 3.27. Функция вычисления суммы значений произвольного количества параметров

```
function sum() {
/* Параметры должны быть числовыми */
 var s = 0;
 for (var i = 0; i<sum.arguments.length; i++){
   s+= sum.arguments[i];
 }
 return s // сумма значений параметров
}
```

Обратите внимание, в определении функции `sum()` не предусмотрены никакие параметры, однако при вызове ей можно передать сколько угодно параметров.

Примеры вызовов функции `sum()`

```
sum(1,2,3)      // 6
sum(5,4)       // 9
sum()          // 0
```

- `toString()` — метод, возвращающий определение функции в виде строки. Синтаксис:

  имя_функции.toString()

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

Пример

```
function myfunc(a, b){
 return a*b/2
}
alert(myfunc.toString())
```

Результат выполнения выражения из последнего примера показан на рис. 3.17.

Рис. 3.17. Результат выполнения `alert(myfunc.toString())`

Кроме рассмотренных ранее, возможны так называемые анонимные определения функций:

имя_переменной = function(*параметры*){*код*}.

Здесь имя функции задается именем переменной. Функция вызывается через выражение `имя_переменной (параметры)`. Поскольку одной переменной можно присвоить другую, то к одной и той же функции можно обращаться под разными именами.

Пример

```
myfunc = function(width, height){var s= width*height; return s};
Srectangle = myfunc;
alert(Srectangle(2,3)) // диалоговое окно с результатом 6
```

В этом примере переменные `myfunc` и `Srectangle` имеют одно и то же значение — строку `"function(width, height){var s= width*height; return s}"`.

3.9. Строки

При разработке сценариев для Web-сайтов работа с текстовыми строками занимает ведущее место. Так, нередко возникает задача проанализировать, что именно ввел пользователь в поля формы, является ли введенный текст адресом электронной формы или URL и т. п. Возможности работы со строками предоставляют встроенные объекты `String` и `RegExp`. Наиболее мощные (но и более сложные для понимания) средства обработки так называемых регулярных выражений (шаблонов) дает объект `RegExp`, который мы не будем рассматривать в данной книге. В большинстве случаев достаточно свойств и методов объекта `String`.

3.9.1. Кавычки и специальные символы

Строковое значение — последовательность символов, заключенная в кавычки, двойные или одинарные:

```
x = "Это — текстовая строка";
y = 'Это — тоже текстовая строка';
```

Внутри строки может находиться подстрока, заключенная в кавычки другого вида:

```
str1 = "ОАО 'Рога и копыта' предлагает хвосты";
str2 = 'ОАО "Рога и копыта" предлагает хвосты';
```

Внешние кавычки — это элементы синтаксиса: они ограничивают набор символов, относящихся к одному и тому же значению строкового типа. При выводе строки на экран эти кавычки не отображаются.

Внутренние кавычки интерпретатор рассматривает как символы самой строки, если только они отличаются от внешних. Например, если внешние кавычки двойные, то внутренние должны быть одинарными, и наоборот. В противном случае возникнет ошибка. Вместе с тем, внутри строки допустимы кавычки того же вида, что и внешние, если перед ними написать обратную косую черту (\):

```
str3 = "ОАО \"Рога и копыта\" предлагает хвосты";
str4 = 'ОАО \'Рога и копыта\' предлагает хвосты';
```

На рис. 3.18 показано, как будут выглядеть строки с кавычками в окне браузера.

При написании простого, но длинного строкового значения в текстовом редакторе нельзя делать переход на новую строку с помощью клавиши <Enter>, как в следующем примере:

```
var x = "Дорогой друг!
Спешу сообщить тебе важную новость";
```

Если строка слишком велика, лучше написать так:

```
var x = "Дорогой друг!";
x+= "Спешу сообщить тебе важную новость";
```

В строках можно использовать специальные символы, называемые еще управляющими кодами. Два из них, \" и \', мы уже рассмотрели. Полный список специальных символов приведен в табл. 3.13.

На рис. 3.19 показаны несколько примеров использования управляющих кодов. Обратите внимание, что символ \n перехода на другую строку работает при выводе в диалоговое окно, а при выводе в окно браузера заменяется пробелом.

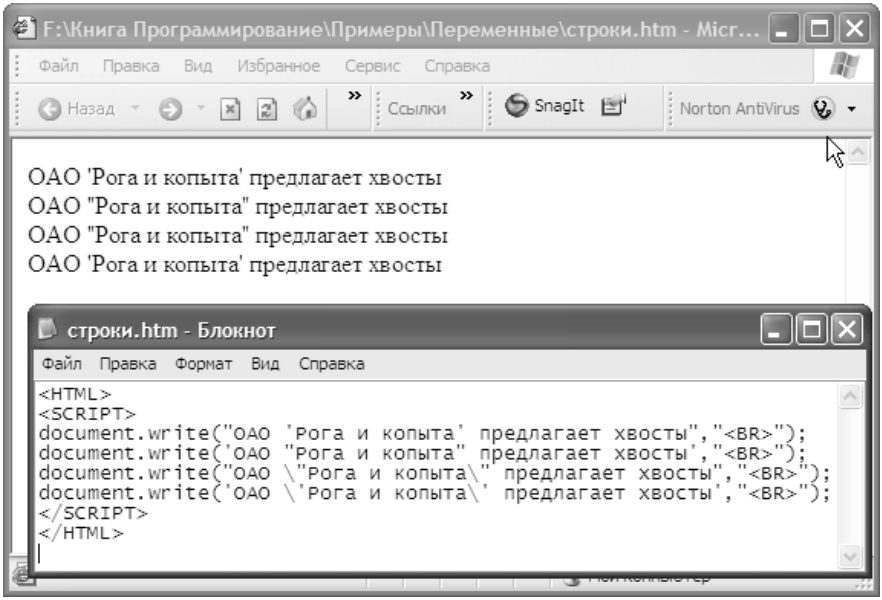

Рис. 3.18. Применение кавычек в строках

Таблица 3.13. *Управляющие коды*

Управляющий код	Значение
`\b`	Возврат на одну позицию
`\t`	Табуляция горизонтальная
`\n`	Переход на новую строку
`\v`	Табуляция вертикальная
`\f`	Подача страницы
`\r`	Возврат каретки
`\"`	Двойная кавычка
`\'`	Одинарная кавычка
`\\`	Обратная косая черта
`\`*ooo*	Символ в кодировке Latin-1, представленный 8-ричными цифрами `ooo`. Допустимы значения от 000 до 377
`\x`*hh*	Символ в кодировке Latin-1, представленный 16-ричными цифрами `hh`. Допустимы значения от 00 до FF
`\u`*hhhh*	Символ в кодировке Unicode, представленный 16-ричными цифрами `hhhh`

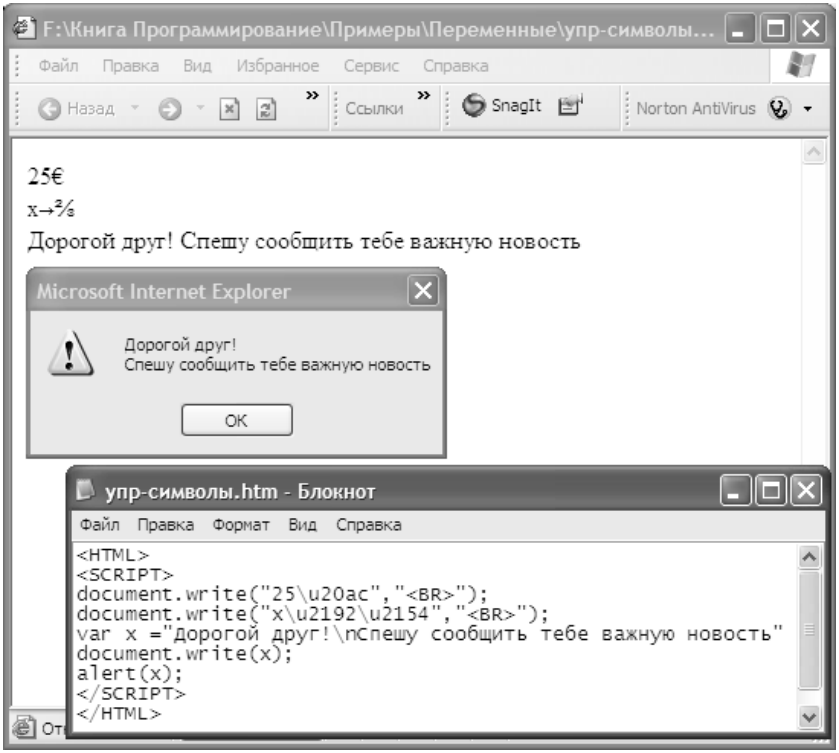

Рис. 3.19. Пример использования управляющих символов

3.9.2. Объект *String*

При использовании оператора присваивания переменной некоторого строкового значения автоматически создается экземпляр объекта `String`. Свойства и методы этого объекта применяются при обработке строк. Например, мы можем узнать длину строки, выделить из нее какую-нибудь часть (подстроку) и т. п.

Пусть `xstring` — переменная с некоторым строковым значением. Тогда синтаксис обращения к свойствам и методам объекта `String`, экземпляр которого ассоциирован с данной строкой, такой:

```
xstring.свойство;
xstring.метод(параметры);
```

Рассмотрим свойства и методы объекта `String` подробнее.

- `length` — длина или, иными словами, количество символов (включая пробелы) в строке; целое число.

Пример

```
"Иван".length      // значение равно 4
"Привет\nвсем".length  /* значение равно 11 (\n — один символ
              перевода строки) */
x=""         // пустая строка
x.length      // значение равно 0 (пустая строка имеет длину 0)
```

- `prototype` — свойство (прототип), позволяющее добавить новые свойства и методы ко всем создаваемым экземплярам строкового объекта, если уже существующих вам окажется недостаточно.

Пример.

```
/* Создадим новый метод для всех экземпляров строковых объектов.
Содержание этого метода определяется пользовательской функцией. */

   function myFunc() {   // функция для нового метода
    return "Вадим"
   }
   // Добавление нового метода myName к прототипу:
   String.prototype.myName()=myFunc
   mystring="Автор этой книги — " + "Дунаев ".myName()
   // значение mystring равно "Автор этой книги — Дунаев Вадим"
```

Подробнее о свойстве `prototype` см. *разд. 3.13.2*.

- `charAt(`*индекс*`)` — возвращает символ, занимающий в строке указанную позицию. Синтаксис:

  строка`.charAt(`*индекс*`)`

  Возвращает односимвольную или пустую строку.

  Параметр (*индекс*) является числом, индекс первого символа равен нулю.

Примеры

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

- `charCodeAt(`*индекс*`)` — преобразует символ в указанной позиции строки в его числовой эквивалент (код). Синтаксис:

  строка`.charCodeAt(`*индекс*`)`

Примеры

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

- `fromCharCode(`*номер1* `,` *номер2* `, ...` *номерN*`)` — возвращает строку символов, числовые коды которой указаны в качестве параметров. Синтаксис:

  `String.fromCharCode(`*номер1* `,` *номер2* `, ...` *номерN*`)`

  Возвращает строку.

Пример

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

- `concat(`*строка*`)` — конкатенация (склейка) строк. Синтаксис:

  строка1`.concat(`*строка2*`)`

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

Примеры

```
"Иван".concat("Иванов");  // значение равно "ИванИванов"
x="Федор" + " ";          // значение x равно "Федор "
x.concat("Иванов");       // значение равно "Федор Иванов"
```

- `indexOf(`*строка_поиска*`,` *индекс*`)` — ищет строку, указанную параметром, и возвращает индекс ее первого вхождения. Синтаксис:

  строка`.indexOf(`*строка_поиска*`,` *индекс*`)`

  Метод ищет позицию первого вхождения *строка_поиска* в строку *строка*. Возвращаемое число (индекс вхождения) отсчитывается от нуля. При неудаче

возвращается –1. Поиск в пустой строке всегда возвращает –1. Поиск пустой строки всегда возвращает нуль.

Второй (необязательный) параметр указывает индекс, с которого следует начать поиск.

Этот метод хорош в комбинации с методом выделения подстроки `substr()` (см. далее), когда сперва требуется определить позиции начала и конца выделяемой подстроки. Рассматриваемый здесь метод подходит для определения начальной позиции.

Примеры

```
x ="В первых строках своего письма";
x.indexOf("первых");   //значение равно 2
x.indexOf("первых строках");  //значение равно 2
x.indexOf("вторых строках");  //значение равно -1
x.indexOf("В");        //значение равно 0
x.indexOf("в");        //значение равно 5
x.indexOf("в", 7);     //значение равно 18
x.indexOf(" ");        //значение равно 1
x.indexOf(" ", 5);     //значение равно 8
x.indexOf("");         //значение равно 0
```

- `lastIndexOf(`*`строка_поиска`*`,`*`индекс`*`)` — ищет строку, указанную параметром, и возвращает индекс ее первого вхождения; при этом поиск начинается с конца исходной строки, но возвращаемый индекс отсчитывается от ее начала, т. е. от нуля. Синтаксис:

  `строка``.lastIndexOf(`*`строка_поиска`*`,`*`индекс`*`)`

  Возвращает число.

  Метод аналогичен рассмотренному ранее `indexOf()` и отличается лишь направлением поиска.

Примеры

```
x ="В первых строках своего письма"
x.lastIndexOf("первых"); // значение равно 2
x.lastIndexOf("a");      // значение равно 29
x.indexOf("a");          // значение равно 14
```

- `localeCompare(`*строка*`)` — позволяет сравнивать строки в кодировке Unicode, т. е. с учетом используемого браузером языка общения с пользователем. Синтаксис:

  строка1`.localeCompare(`*строка2*`)`

  Возвращает число.

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

- `slice(`*индекс1*`, `*индекс2*`)` — возвращает подстроку исходной строки, начальный и конечный индексы которой указываются параметрами, за исключением последнего символа. Синтаксис:

  строка`.slice(`*индекс1*`, `*индекс2*`)`

  Данный метод не изменяет исходную строку.

  Если второй параметр не указан, то возвращается подстрока с начальной позицией *индекс1* и до конца строки. Отсчет позиций начинается с начала строки. Первый символ строки имеет нулевой индекс. Если второй параметр указан, то возвращается подстрока исходной строки, начиная с позиции *индекс1* и до позиции *индекс2*, исключая последний символ. Если второй параметр отрицательный, то конечный индекс отсчитывается от конца строки. В этом основное различие методов `slice()` и `substr()`. Сравните также этот метод с методом `substring()`.

Примеры

```
x="В первых строках своего письма";
x.slice(3,8);  // значение равно "ервых"
x.slice(3,-2)  // значение равно "ервых строках своего пись"

/* Анализ адреса электронной почты */
x=mumu@gerasim.ru;
i=x.indexOf("@");           // значение равно 4
_name=x.slice(0, i);         // значение равно "mumu"
_domen=x.slice(i+1, x.indexOf(".")); /* значение равно "gerasim" */
_domen=x.slice(i+1, -3);       // значение равно "gerasim"
```

- `split(`*разделитель*`, `*ограничитель*`)` — возвращает массив элементов, полученных из исходной строки. Синтаксис:

  строка`.split(`*разделитель*`, `*ограничитель*`)`

  Первый параметр — строка символов в качестве разделителя строки на элементы. Второй необязательный параметр — число, указывающее, сколько элементов строки, полученной при разделении, следует включить в возвращаемый массив. Если разделитель — пустая строка, то возвращается массив символов строки.

Примеры

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

- `substr(`*индекс*`, `*длина*`)` — возвращает подстроку исходной строки, начальный индекс и длина которой указываются параметрами. Синтаксис:

  строка`.substr(`*индекс*`, `*длина*`)`

  Данный метод не изменяет исходную строку.

  Если второй параметр не указан, то возвращается подстрока с начальной позицией *индекс1* и до конца строки. Отсчет позиций выполняется с начала строки. Первый символ строки имеет нулевой индекс. Если второй параметр указан, то возвращается подстрока исходной строки, начиная с позиции *индекс1* и с общим количеством символов, равным значению *длина*. Сравните этот метод с методами `slice()` и `substring()`.

Примеры

```
x="Привет всем";
x.substr(7,4);      // значение равно "всем"

/* Анализ адреса электронной почты */
x = mumu@gerasim.ru;
i = x.indexOf("@");     // значение равно 4
name = x.substr(0, i);  // значение равно "mumu"
domen = x.substr(i+1)   // значение равно "gerasim.ru"
```

- `substring(`*индекс1*`, `*индекс2*`)` — возвращает подстроку исходной строки, начальный и конечный индексы которой указываются параметрами. Синтаксис:

  строка`.substring(`*индекс1*`, `*индекс2*`)`

  Данный метод не изменяет исходную строку.

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

Примеры

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

- `toLocaleLowerCase()`, `toLowerCase()` — переводят строку в нижний регистр. Синтаксис:

  строка`.toLocaleLowerCase()`

  строка`.toLowerCase()`

  Возвращают строку.

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

Примеры

```
x="ЗдраВствуйТе";
x.toLocaleLowerCase();    // значение равно "здравствуйте"
x.toLowerCase();          // значение равно "здравствуйте"
y="Здравствуйте";
x == y;                   // значение равно false
x.toLowerCase() == y.toLowerCase(); // значение равно true
```

- `toLocaleUpperCase()`, `toUpperCase()` — переводят строку в верхний регистр. Синтаксис:

  строка`.toLocaleUpperCase()`

  строка`.toUpperCase()`

  Возвращают строку.

  Назначение аналогично двум предыдущим методам.

Примеры

```
x="ЗдраВствуйТе";
x.toLocaleUpperCase();      // значение равно "ЗДРАВСТВУЙТЕ"
x.toUpperCase();            // значение равно "ЗДРАВСТВУЙТЕ"
y="Здравствуйте";
x == y;                     // значение равно false
x.toUpperCase() == y.toUpperCase(); // значение равно true
```

Следующая группа методов позволяет отформатировать строки при их выводе в окно браузера так же, как и с помощью тегов HTML:

- `anchor("`*anchor_имя*`")`
- `blink()`
- `bold()`
- `fixed()`
- `fontcolor(`*значение_цвета*`)`
- `fontsize(`*число от 1 до 7*`)`
- `italics()`
- `link(`*расположение или URL*`)`
- `big()`
- `small()`
- `strike()`
- `sub()`
- `sup()`

Примеры

```
document.write("Глава 2".anchor("volume2"))  /* эквивалентно HTML-коду:
                <A NAME="volume2"> Глава 2 </A> */


document.write("Страница автора".link("http:/dunaevv1.narod.ru"))
   /* эквивалентно HTML-коду:
    <A HREF=" http:/dunaevv1.narod.ru ")> Страница автора </A> */
```

3.9.3. Функции вставки и замены подстрок

При обработке строк часто требуется вставить или заменить подстроки. Удаление подстроки из данной строки можно рассматривать как частный случай замены (на пустую строку). С помощью методов объекта `String` можно написать программу для решения этой задачи. Поскольку она часто встречается, то целесообразно оформить программу в виде нескольких функций.

Рассмотрим сначала функцию вставки одной строки в другую. Назовем ее, например, `insstr`. Данная функция должна принимать три параметра: исходную строку `s1`, вставляемую строку `s2` и индекс позиции вставки `n`. Листинг 3.28 содержит ее определение и примеры вызова.

Листинг 3.28

```
function insstr(s1,s2,n) {
 return s1.slice(0,n) + s2 + s1.slice(n)
}

insstr("Привет, друзья", " мои", 7);   // "Привет, мои друзья"
insstr("Привет, друзья", " мои", 100);  // "Привет, друзья мои"
```

Теперь займемся функцией, которая заменяет в исходной строке все вхождения заданной подстроки на подстроку замены. Назовем эту функцию `replacestr`. Она должна принимать три параметра: исходную строку `s1`, заменяемую подстроку `s2` и подстроку `s3`, которой следует заменить все вхождения `s2` в `s1` (`s2` может встречаться в `s1` несколько раз).

Очевидно, прежде всего, необходимо найти все вхождения `s2` в `s1`. Если исходная строка не содержит в себе подстрок `s2`, то функция должна вернуть исходную подстроку без всяких изменений. В противном случае требуется изъять из `s1` все вхождения `s2`, а на их место вставить подстроку `s3`. Полученная таким образом строка должна возвращаться функцией в качестве результата. Листинг 3.29 содержит определение функции для замены подстрок и два примера ее вызова.

Листинг 3.29

```
function replacestr(s1, s2, s3) {
 var s="";       // обработанная часть строки
 while (true) {
  i=s1.indexOf(s2)  // индекс вхождения s2 в s1
  if (i >= 0) {
```

```
   s=s + s1.substr(0, i) + s3;  /* обработанная часть строки */
   s1=s1.substr(i + s2.length);  /* оставшаяся часть строки */
  }else break;
 }
 return s + sx
}

replacestr("bacdae","a","X");  // "bXcdXe"
x="Иван Иванов";
replacestr(x,"Иван","Федор");  // "Федор Федоров"
```

ЗАМЕЧАНИЕ

Если заменяемая подстрока `s2` пустая, то цикл будет продолжаться бесконечно, поскольку пустая строка входит в состав любой строки.

Чтобы избежать зацикливания, следует немного изменить код тела функции. Но что должна возвращать наша функция, если `s2` — пустая строка? Вы можете решить, что в этом случае следует просто вернуть исходную строку `s1`, или же, например, считать пустую строку строкой, содержащей пробел. Последний вариант представляется мне более интересным. Листинг 3.30 содержит код, реализующий эту идею.

Листинг 3.30

```
function replacestr(s1, s2, s3) {
 if (s2 == "") s2=" "; // заменяем пустую строку на строку с пробелом
 var s="";       // обработанная часть строки
 while (true) {
  i=s1.indexOf(s2);     // индекс вхождения s2 в s1
  if (i >= 0) {
   s = s + s1.substr(0, i) + s3;  /* обработанная часть строки */
   s1 = s1.substr(i + s2.length);  /* оставшаяся часть строки */
  }else break;            // выход из цикла
 }
 return s + sx
}
```

Рассмотренные функции вставки и замены подстрок (`insstr()` и `replacestr()`) без всяких доработок пригодны к практическому применению в программах, где требуется обработка строк.

3.9.4. Функции удаления ведущих и заключительных пробелов

При обработке строк (например, введенных пользователем в поля формы) нередко возникает задача удаления лишних ведущих и завершающих пробелов. Многие языки программирования имеют соответствующие встроенные функции, но в JavaScript их нет. Поэтому не помешает создать их самостоятельно с помощью имеющихся средств.

Решить поставленную задачу можно несколькими способами. Вот один из них. Преобразуем исходную строку в массив слов, используя в качестве разделителя один пробел `" "`. Затем необходимо проанализировать первые или последние элементы массива в зависимости от того, что нам требуется: удалить ведущие или завершающие пробелы в исходной строке. Допустим, необходимо избавиться от ведущих пробелов. Тогда если в исходной строке было `N` пробелов, то первые `N` элементов массива будут содержать пустые строки `""`. Мы проверяем в цикле значения первых элементов, пока не найдем непустой или пока не исчерпаем весь массив. Если первый непустой элемент массива имеет индекс `i`, то создадим новый массив, содержащий элементы исходного, начиная с этого индекса. Наконец, склеим элементы этого массива в одну строку, используя в качестве разделителя строку с единственным пробелом `" "`. Все, что нам понадобится, — методы объекта `String`, операторы цикла и условного перехода.

Сначала рассмотрим функцию `ltrim()`, удаляющую ведущие пробелы из исходной строки (листинг 3.31).

Листинг 3.31

```
function ltrim(xstr){
 if (!(xstr.indexOf(" ") == 0))
 return xstr;       /* вернуть исходную строку, если в ней нет
               ведущих пробелов */
 var astr = xstr.split(" "); // создаем массив из слов строки
 var i=0;
 while (i < astr.length){
  if (!(astr[i] == (""))) break; /* выходим из цикла,
                                    если элемент не пуст */
  i++;
 }
 astr = astr.slice(i);   // создаем массив
 return astr.join(" ");     // склеиваем элементы массива в строку
}
```

Функция `rtrim()` для удаления заключительных пробелов в строке устроена аналогично (листинг 3.32). Ее отличие в том, что поиск пробелов происходит с конца строки. Обратите внимание, что счетчик цикла в этом случае убывающий.

Листинг 3.32

```
function rtrim(xstr){
 if (!(xstr.lastIndexOf(" ") == xstr.length - 1)) return xstr;
 var astr=xstr.split(" ");
 var i=astr.length -1;
 while (i>0){
  if (!(astr[i] == (""))) break;
  i--;
 }
 astr = astr.slice(0, i+1);
 return astr.join(" ")
}
```

Чтобы удалить из строки и ведущие, и концевые пробелы, достаточно выполнить следующее выражение:

```
ystr=rtrim(ltrim(xstr));
```

Впрочем, можно создать специальную функцию `trim()`, которая тримингует строку:

```
function trim(xstr) {
 return rtrim(ltrim(xstr))
}
```

3.10. Массивы

Как уже упоминалось, *массив* представляет собой упорядоченный набор данных. В качестве элементов массива могут выступать как примитивные данные (например, строки, числа), так и другие массивы. В последнем случае можно говорить о многомерных массивах (т. е. о массивах массивов). Количество элементов в массиве называется *длиной* массива. К элементам массива можно обращаться в программе по их порядковому номеру *(индексу)*. Нумерация элементов массива начинается с нуля, так что первый элемент имеет индекс 0, а последний — на единицу меньший, чем длина массива.

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

3.10.1. Создание массива

Существует несколько способов создания массива. Самый простой, пригодный в случае небольших массивов, заключается в том, чтобы задать список элементов, заключенный в квадратные скобки:

```
имя_массива = [элемент1, элемент2, … ,элементN];
```

Здесь `имя_массива` — переменная, значением которой является массив. Доступ к элементам массива обеспечивается с помощью выражения `имя_массива[индекс]`, где `индекс` — целое число от 0 до N-1 (нумерация элементов массива начинается с нуля). Так например, `элемент1`— первый элемент массива, но его индекс равен нулю.

При таком задании массива автоматически создается экземпляр встроенного объекта `Array`. То же самое можно сделать и с помощью явного выражения создания экземпляра объекта `Array`:

`имя_массива` = new Array(*`длина_массива`*);

Здесь *`длина_массива`* – необязательный числовой параметр. Если длина массива не указана, то создается пустой массив, не содержащий ни одного элемента. В противном случае формируется массив с указанным количеством элементов, однако все они имеют значение `undefined` (т. е. не определены).

У объекта `Array` имеется свойство `length`, значением которого является длина массива. Получить значение этого свойства можно, указав выражение

`имя_массива`.length.

Создав массив, можно присвоить значения его элементам с помощью оператора присваивания. В его левой части указывают имя массива, а рядом с ним в квадратных скобках — индекс элемента. Мы уже говорили, что к элементам массива обращаются по индексу:

`имя_массива`[*`индекс`*].

Рассмотрим создание массива `земля`, содержащего в качестве элементов некоторые характеристики нашей планеты (листинг 3.33). Обратите внимание, что элементы в этом массиве различных типов (строковые и числовые).

Листинг 3.33

```
земля = new Array(4);   // массив из 4-х элементов
земля[0] = "Планета";
земля[1] = "24 часа";
земля[2] = 6378;
земля[3] = 365.25;

земля.length       // значение равно 4
```

Если нам потребуется значение, например, третьего элемента массива, то достаточно использовать выражение `земля[2]`.

При создании массива его элементы можно указать в круглых скобках за ключевым словом `Array`:

```
земля = new Array("Планета", "24 часа", 6378, 365.25);
```

Индексы для элементов такого массива JavaScript создает автоматически.

Элементам массива можно задать индивидуальные имена — присвоить имя каждому элементу, подобно имени свойства объекта (листинг 3.34).

Листинг 3.34

```
земля = new Array();   // пустой массив
земля.xtype = "Планета";
земля.xday = "24 часа";
земля.radius = 6378;
земля.period = 365.25;
```

В этом случае обращение к элементу происходит как к свойству объекта, например, `земля.radius`. Возможен и такой вариант: `земля["radius"]`. По числовому индексу к элементам в таком массиве обращаться нельзя. Так например, значение элемента `земля[2]` равно `undefined`, а не `"radius"`. Массивы с символьными индексами (с именами значений) называют еще ассоциативными.

Для удаления элемента массива служит оператор `delete`, при этом длина массива остается прежней, а удаленный элемент приобретает значение `undefined` (листинг 3.35).

Листинг 3.35

```
myarray = new Array(1,2,3,4);
delete myarray[2];
alert(myarray);     // массив 1,2,,4
alert(myarray.length);   // 4
alert(myarray[2]);     // undefined
```

Чтобы удалить массив целиком, необходимо присвоить ему значение `null`.

3.10.2. Многомерные массивы

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

имя_массива[*индекс_уровня1*] [*индекс_уровня2*]… [*индекс_уровняN*].

Типичный пример — массив опций меню. У такого меню есть горизонтальная панель с опциями, называемая главным меню. Некоторым опциям главного меню соответствуют раскрывающиеся вертикальные подменю со своими опциями. Мы создаем массив, длина которого равна количеству опций главного меню. Элементы этого массива определяем как массивы названий опций соответствующих подменю. Чтобы была ясна структура нашей конструкции, мы выбрали названия опций надлежащим образом. Например, "Меню 2.1" — название первой опции подменю, соответствующего второй опции главного меню (листинг 3.36).

Листинг 3.36

```
menu = new Array();
menu[0] = new array("Меню 1.1", "Меню 1.2", "Меню 1.3");
menu[1] = new array("Меню 2.1", "Меню 2.2");
menu[2] = new array("Меню 3.1", "Меню 3.2" , "Меню 3.3", "Меню 3.4");
```

Чтобы обратиться ко второй опции третьего подменю, следует написать:

```
menu[2][1]    // значение равно "Меню 3.2".
```

Усложним нашу конструкцию, чтобы она содержала не только названия опций подменю, но и названия опций главного меню (листинг 3.37).

Листинг 3.37

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

menu[0][1]   // значение равно "Меню 2"
menu[0][2]   // значение равно "Меню 3"
menu[1][1][0]  // значение равно "Меню 2.1"
menu[1][2][3]  // значение равно "Меню 3.2"
```

3.10.3. Копирование массива

Иногда требуется создать копию массива, чтобы сохранить исходные данные и предохранить их от последующих модификаций. Например, рассмотренный далее метод сортировки элементов изменяет исходный массив, и необходима предварительная операция копирования. Однако для копирования массива недостаточно просто присвоить его другой переменной. Используя новую переменную и оператор присваивания, мы создаем лишь новую ссылку на прежний массив, а не новый массив.

Пример

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

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

Чтобы скопировать массив, т. е. создать новый массив, элементы которого равны соответствующим элементам исходного, следует воспользоваться оператором цикла, в котором элементам нового массива присваиваются значения элементов исходного.

Пример

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

3.10.4. Объект *Array*

Для работы с массивами предназначены свойства и методы объекта `Array`.

Пусть `myarray` — переменная с некоторым значением типа массив (имя некоторого массива). Тогда синтаксис обращения к свойствам и методам объекта `Array` такой:

`myarray.`*`свойство`*`;`

`myarray.`*`метод`*`(`*`параметры`*`);`

Рассмотрим свойства и методы объекта `Array` подробнее.

- `length` — длина или, иными словами, количество элементов в массиве; целое число. Синтаксис:

  `имя_массива``.length`

  Поскольку индексация элементов массива начинается с нуля, индекс последнего элемента на единицу меньше длины массива. Это обстоятельство удобно использовать при добавлении к массиву нового элемента:

  `myarray[mayarray.length]=`*`значение;`*

  Чтобы удалить из массива `n` последних элементов, достаточно просто уменьшить его длину на `n`:

  `имя_массива``.length = `*`имя_массива`*`.length - n;`

  Аналогично можно увеличить длину массива. При этом добавленные элементы будут иметь значение `undefined`.

- `prototype` — свойство (прототип), позволяющее добавить новые свойства и методы ко всем существующим и создаваемым массивам.

  Например, следующее выражение добавляет свойство `author` ко всем уже созданным массивам:

  `Array.prototype.author="Иванов"`

  Если теперь вместо `Array.prototype` написать имя существующего массива, то можно изменить значение свойства `author` только для этого массива (листинг 3.38).

Листинг 3.38

```
myarray = new Array();         // создание массива myarray
xarray = new Array();         // создание массива xarray
Array.prototype.author = "Иванов";  /* добавление свойства author
                    ко всем массивам */
myarray.author = "Иванов младший";  /* изменение свойства author
                    для myarray */
xarray.author = "Сидоров";       /* изменение свойства author
                    для xarray */
```

Прототипу можно присвоить функции, т. е. методы для специфической обработки массивов, которые пополнят множество уже существующих методов объекта `Array`.

Рассмотрим вычисление суммы элементов массива (листинг 3.39). Мы определяем функцию `aSum()`, которая возвращает сумму элементов числового массива. В качестве параметра эта функция принимает массив. Затем создаем конкретный массив чисел. Наконец, присоединяем к прототипу массива новый метод `Sum` — определенную ранее функцию `aSum()`.

Листинг 3.39

```
function aSum(xarray){
 var s = 0;
 for(i = 0; i < xarray.length; i++){
  s = s + xarray[i]
 }
 return s
}
myarray = new Array(2, 3, 4); /* создаем массив myarray из 3-х чисел
*/
Array.prototype.Sum = aSum;  /* присоединяем метод (около имени
                                функции скобки указывать не нужно) */
s = myarray.Sum(myarray);     /* применяем метод Sum к массиву
                         myarray,передавая его в качестве параметра */
```

Этот пример призван просто проиллюстрировать действие свойства `prototype`. На самом деле вычислить сумму элементов массива можно гораздо проще:

```
s = aSum(myarray)
```

Подробнее о свойстве `prototype` см. *разд. 3.13.2*.

- `concat(`*массив*`)` — конкатенация массивов, объединяет два массива в третий. Синтаксис:

  имя_массива1`.concat(`*массив2*`)`

  Возвращает массив. Данный метод не изменяет исходные массивы.

Пример

```
a1 = new array(1, 2, "Звезда");
a2 = new array("а", "б", "в", "г");
a3 = a1.concat(a3);  /* результат – массив с элементами:
            1, 2, "Звезда", "а", "б", "в", "г" */
```

- `join(`*разделитель*`)` — создает строку из элементов массива с указанным разделителем между ними; является строкой символов (возможно, пустой). Синтаксис:

  имя_массива`.join(`*строка*`)`

Примеры

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

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

- `pop()` — удаляет последний элемент массива и возвращает его значение. Синтаксис:

  имя_массива`.pop()`

  Возвращает значение удаленного элемента массива. Данный метод изменяет исходный массив.

- `push(`*значение или объект*`)` — добавляет к массиву указанное значение в качестве последнего элемента и возвращает новую длину массива. Синтаксис:

  имя_массива1`.push(`*значение или объект*`)`

  Данный метод изменяет исходный массив.

- `shift()` — удаляет первый элемент массива и возвращает его значение. Синтаксис:

  имя_массива`.shift()`

  Данный метод изменяет исходный массив.

- `unshift(`*значение или объект*`)` — добавляет к массиву указанное значение в качестве первого элемента. Синтаксис:

  имя_массива`.unshift(`*значение или объект*`)`

  Метод изменяет исходный массив.

- `reverse()` — изменяет порядок следования элементов массива на противоположный. Синтаксис:

  имя_массива`.reverse()`

  Возвращает массив. Данный метод изменяет исходный массив.

Пример

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

- `slice(`*индекс1*`, `*индексN*`)` — создает массив из элементов исходного массива с индексами указанного диапазона. Синтаксис:

  имя_массива`. slise(`*индекс1*`, `*индексN*`)`

  Возвращает массив. Данный метод не изменяет исходный массив.

  Второй параметр (конечный индекс) не является обязательным. Если он не указан, то создаваемый массив содержит элементы исходного массива, начиная с индекса *индекс1* и до конца. В противном случае создаваемый массив содержит элементы исходного массива, начиная с индекса *индекс1* и до индекса *индекс2*, за исключением последнего элемента. При этом исходный массив остается без изменений.

Пример

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

- `sort(`*функция_сортировки*`)` — сортирует (упорядочивает) элементы массива с помощью функции сравнения. Синтаксис:

  имя_массива`.sort(`*функция_сравнения*`)`

  Возвращает массив. Данный метод изменяет исходный массив.

  Параметр можно опускать, если он не указан, то при сортировке сравниваются ASCII-коды символов значений. Это удобно для сравнения символьных строк, но не совсем подходит для чисел. Так, число 357 при сор-

тировке считается меньшим, чем 75, поскольку сначала сравниваются первые символы и только в случае их равенства сравниваются следующие, и т. д. Таким образом, метод `sort()` без параметра подходит для простой сортировки массива со строковыми элементами.

Можно создать собственную функцию для сравнения элементов массива, с помощью которой метод `sort()` упорядочит весь массив. Имя этой функции (без кавычек и круглых скобок) передается методу в качестве параметра. При работе метода функции передаются два элемента массива, а ее код возвращает методу значение, указывающее порядок следования элементов. Допустим, сравниваются два элемента, `x` и `y`. Тогда в зависимости от числового значения (отрицательного, 0 или положительного), возвращаемого функцией сравнения, методом `sort()` принимается одно из трех возможных решений (табл. 3.14).

Таблица 3.14. *Результаты сравнения двух элементов массива*

Значение, возвращаемое функцией сравнения	**Результат сравнения `x` и `y`**
Меньше 0	`y` следует за `x`
0	Порядок следования `x` и `y` не изменяется
Больше 0	`x` следует за `y`

Итак, критерий сортировки элементов массива определяется кодом функции сравнения. Если элемент массива имеет значение `null`, то в Internet Explorer он размещается в начале массива.

Пример

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

- `splice(`*индекс*`,` *количество*`,` *элем1*`,` *элем2* `,` `...`*элемN*`)` — удаляет из массива несколько элементов и возвращает массив из удаленных элементов или заменяет значения элементов. Синтаксис:

  имя_массива `splice(`*индекс*`,` *количество*`,` *элем1*`,` *элем2*`,` `...`*элемN*`)`

  Данный метод изменяет исходный массив.

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

Пример

```
a = new Array("Вася", "Иаван", "Марья", 12, 5);
x = a.splice(1,3);  /* x — массив элементов: "Иван", "Марья", 12
                       a — массив элементов: "Вася", 5 */
```

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

Пример

```
a = new Array("Вася", "Иван", "Марья", 12, 5);
x = a.splice(1,3,"Петр", "Кузьма", "Анна");
/* x — массив элементов: "Иван", "Марья", 12
  a — массив элементов: "Вася","Петр", Кузьма", "Анна",5 */

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

- `toLocaleString()`, `toString()` — преобразуют содержимое массива в символьную строку.

  Алгоритм преобразования по методу `toLocaleString()` зависит от браузера.

  Для преобразования содержимого массива в строку я рекомендую метод `join()`.

3.10.5. Функции обработки числовых массивов

Во многих приложениях требуется получить статистические характеристики числовых данных, хранящихся в виде массива: сумму всех чисел, среднее, максимальное и минимальное значения. В листинге 3.40 приведены коды функций, вычисляющих эти величины.

Листинг 3.40

```
// Функция, возвращающая сумму значений всех элементов непустого массива
/* Для вычисления среднего значения следует просто воспользоваться выра-
жением: S(aN)/aN.length */

   function S(aN){
    var S=aN[0];
    for(var i=1; i< aN.length; i++){
     S += aN[i]
    }
    return S
   }

// Функция, возвращающая минимальное значение среди элементов массива
   function Nmin(aN){
    var Nmin=aN[0];
    for(var i=1; i < aN.length; i++){
     if (aN[i] < Nmin)
      Nmin=aN[i]
     }
    return Nmin
   }

// Функция, возвращающая максимальное значение среди элементов массива
   function Nmax(aN){
    var Nmax=aN[0];
    for(var i=1; i < aN.length; i++){
     if (aN[i] > Nmax)
      Nmax=aN[i]
     }
    return Nmax
   }
```

Мы можем создать одну функцию, которая вычисляет все перечисленные статистические характеристики и возвращает их как значения массива (листинг 3.41).

Листинг 3.41

```
function statistic(aN){
 if (aN == 0 || aN == null || aN == "") return new Array(0,0,0,0)
 var S=aN[0];
 var Nmin=aN[0];
 var Nmax=aN[0];
 for(var i=1; i<=aN.length-1; i++){
  S += aN[i];
  if (aN[i] < Nmin) Nmin=aN[i];
  if (aN[i] > Nmax) Nmax=aN[i];
 }
 return new Array(S, S/aN.length, Nmin, Nmax)
}
```

В начале кода функции `statistic()` мы проверяем, не является ли параметр пустым. Если это так, то все статистические характеристики считаются равными нулю.

3.11. Числа

При разработке Web-страниц математические операции над числами обычно используются для изменения координат элементов страницы (свойства `top`, `left`, `width`, `height` каскадной таблицы стилей), а также для вычисления статистических характеристик данных, содержащихся, например, в некоторой таблице. Так или иначе, в этих задачах необходимо иметь дело с числами. О числах уже говорилось в *разд. 3.5*, посвященном типам данных. Теперь рассмотрим их более подробно.

3.11.1. Числа целые и с плавающей точкой

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

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

В JavaScript можно производить операции с числами различных типов. Это очень удобно. Однако при этом следует знать, какого числового типа будет результат. Если результат операции является числом с дробной частью, то он представляется как число с плавающей точкой. Если результат оказался без дробной части, то он приводится к целочисленному типу, а не представляется числом, у которого в дробной части одни нули.

Примеры

```
2 + 3     // 5 — целое число
2 + 3.6   // 5.6 — число с плавающей точкой
2.4 + 3.6 // 6 — целое число
6.00      // число с плавающей точкой
```

Числа можно представлять и в так называемой экспоненциальной форме, т. е. в формате: *число1*e*число2* или *число1*E*число2*. Такая запись числа означает $число1 \times 10^{число2}$.

Примеры

```
1e5     // 100000
2e6     // 2000000
1.5e3   // 1500
+1.5e3  // 1500
-1.5e3  // -1500
3e-4    // 0.0003
```

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

Запись числа в шестнадцатеричной форме начинается с префикса `0x` (или `0X`), где первый символ ноль, а не буква О, затем следуют символы шестнадцатеричных цифр: `0`, `1`, `2`, ..., `9`, `a`, `b`, `c`, `d`, `e`, `f` (буквы могут быть в любом регист-

ре). Например, шестнадцатеричное число `0x4af` в десятичном представлении есть `1199`.

Запись числа в восьмеричной форме начинается с нуля, за которым следуют цифры от `0` до `7`. Например, `027` (в десятичном представлении — `23`).

В арифметических выражениях числа могут быть представлены в любой из перечисленных систем счисления, однако результат всегда приводится к десятичной форме.

Рассмотрим пример функции преобразования из десятичной формы в шестнадцатеричную (листинг 3.42). Функция `to16()` здесь принимает в качестве параметра десятичное число и преобразует его в строку, содержащую это же число, но в шестнадцатеричной форме. При этом мы ограничиваемся числами, не превышающими 255 (тогда для представления числа потребуется не более двух шестнадцатеричных цифр).

Листинг 3.42

```
function to16(n10) {
 var hchars="0123456789abcdef";   /* строка, содержащая все
                                     шестнадцатеричные цифры */
 if (n10 > 255) return null;
 var i=n10%16;
 var j=(n10 – i)/16;
 rezult="0x";
 rezult += hchars.charAt(j);
 rezult += hchars.charAt(i);
 return rezult
}

to16(250);          // "0xfa"
to16(30);           // "0x1e"
to16(30)+10;        // "0x1e10" – склейка, а не сложение
parseInt(to16(30)) + 10;  // 40
```

Функцию `to16()` можно также создать и на основе массивов (листинг 3.43).

Листинг 3.43

```
function to16(n10) {
 var hchars=new Array("0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
"a", "b", "c", "d", "e", "f"};
```

```
 if (n10 > 255) return null;
 var i=n10%16;
 var j=(n10 – i)/16;
 rezult="0x";
 rezult += hchars[j];
 rezult += hchars[i];
 return rezult
}
```

Теперь рассмотрим пример функции преобразования из десятичной формы в двоичную (листинг 3.44). Здесь функция `to2()` принимает в качестве параметра десятичное число и преобразует его в строку, содержащую это же число, но в двоичной форме. Результирующая строка состоит из нулей и единиц. Здесь реализован рекурсивный вызов функции `to2()` самой себя. Обратите внимание, что использование ключевого слова `var` в данном случае принципиально важно из-за рекурсии, иначе функция будет работать неправильно.

Листинг 3.44

```
function to2(n10) {
 if (n10 < 2)return "" + n10; // чтобы результат был строковым
 var i=n10%2;
 var j=(n10-i)/2;
 return to2(j)+i
}
```

Для преобразования строк, содержащих числа, в данные числового типа служат встроенные функции `parseInt()` и `parseFloat()` соответственно для представления в целочисленном виде и в виде числа с плавающей точкой. Их мы уже рассматривали ранее (см. *разд. 3.5.4*). Здесь следует отметить, что параметрами этих функций могут быть строки, содержащие числа не только в десятичной, но и в шестнадцатеричной, и в восьмеричной формах. Для указания основания системы счисления служит второй параметр этих функций.

Примеры

```
parseInt("ff",16)   // значение равно 255
parseInt("ff")      // значение равно NaN
parseInt("010")     // значение равно 8
parseInt("010", 8)  // значение равно 8
```

```
parseInt("010", 10) // значение равно 10
parseInt("010", 2)  // значение равно 2
parseInt("010", 16) // значение равно 17
```

Для преобразования числа в строку, содержащую это число, достаточно использовать выражение сложения пустой строки с числом.

Примеры

```
"" + 25.78 // значение равно "25.78"
"" + 2.5e3 // значение равно "2500"
```

3.11.2. Объект *Number*

Обычно числа создают с помощью переменных и оператора присваивания. Однако с каждым числом ассоциируется экземпляр встроенного объекта `Number`, обладающий некоторыми полезными свойствами и методами, которые могут пригодиться.

Экземпляр объекта `Number` можно создать явным образом:

имя_еременной = new Number(*число*)

Доступ к свойствам и методам объекта Number обеспечивается такими выражениями:

число.*свойство*

Number.*свойство*

число.*метод*(*параметры*)

Number.*метод*(*параметры*)

Далее мы рассмотрим свойства и методы объекта `Number`.

- `MAX_VALUE` — константа, значение которой равно наибольшему допустимому в JavaScript значению числа (`1.7976931348623157e+308`);
- `MIN_VALUE` — константа, значение которой равно наименьшему допустимому в JavaScript значению числа (`5e-324`);
- `NEGATIVE_INFINITY` — число, меньшее, чем `Number.MIN_VALUE`;
- `POSITIVE_INFINITY` — число, большее, чем `Number.MAX_VALUE`;
- `NaN` — константа, имеющая значение NaN, посредством которой JavaScript сообщает, что данные (параметр, возвращаемое значение) не является числом (Not a Number);

- `prototype` — свойство (прототип), играющее такую же роль, что и в случае других объектов (`String`, `Array` и т. д.).

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

- `toExponential(`*количество*`)` — представляет число в экспоненциальной форме. Синтаксис:

  число`.toExponential(`*количество*`)`

  Возвращает строку.

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

Примеры

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

- `toFixed(`*количество*`)` — представляет число в форме с фиксированным количеством цифр после точки. Синтаксис:

  число`.toFixed(`*количество*`)`

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

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

Примеры

```
x = 25.65;
x.toFixed(3);  // 25.650
x.toFixed(2);  // 25.65
x.toFixed(1);  // 25.7
x.toFixed(0);  // 25.7
```

- `toPrecision(`*точность*`)` — представляет число с заданным общим количеством значащих цифр. Синтаксис:

  число`.toPrecision(`*точность*`)`

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

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

Примеры

```
x = 135.45;
x.toPrecision(6); // 135.450
x.toPrecision(5); // 135.45
x.toPrecision(4); // 135.5
x.toPrecision(3); // 135
x.toPrecision(2); // 1.4e2
x.toPrecision(1); // 1e2
x.toPrecision(0); // Сообщение об ошибке
```

- `toString([`*основание*`])` — возвращает строковое представление числа в системе счисления с указанным основанием. Синтаксис:

  число`.toString([`*основание*`])`

  Если параметр не указан, имеется в виду десятичная система счисления.

  Вы можете указать 2 для двоичной системы, или 16 — для шестнадцатеричной. Заметим, что этот метод имеют все объекты.

Примеры

```
x = 127.18;
x.toString();    // "127.18"
x.toString(10);  // "127.18"
x.toString(16);  // "7f.2e147ae147b"
x.toString(8);   // "177.134121727024366"

x = 5;
x.toString(2);   // "101"
```

3.11.3. Объект *Math*

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

`Math.`*свойство*`;`

`Math.`*метод*`(`*параметры*`);`

Значениями свойств объекта `Math` являются математические константы (табл. 3.15).

Таблица 3.15. *Константы объекта* `Math`

Константа	Значение
`E`	Постоянная Эйлера 2,71828…
`LN10`	Натуральный логарифм числа 10
`LN2`	Натуральный логарифм числа 2
`LOG10E`	Десятичный логарифм экспоненты (числа `e`)
`LOG2E`	Двоичный логарифм экспоненты
`PI`	Число π
`SQRT1_2`	Квадратный корень из 1/2
`SORT`	Квадратный корень из 2

Например, для вычисления длины окружности при известном радиусе требуется число π, которое можно взять как свойство объекта `Math`:

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

Перечислим методы объекта `Math`.

- `abs(`*число*`)` — модуль (абсолютное значение) числа;
- `acos(`*число*`)` — арккосинус числа;
- `asin(`*число*`)` — арксинус числа;
- `atan(`*число*`)` — арктангенс числа;
- `atan2(`*x*`, `*y*`)` — угол в полярных координатах точки;
- `ceil(`*число*`)` — округление числа вверх до ближайшего целого;
- `cos(`*число*`)` — косинус числа;
- `exp(`*число*`)` — число `e` в степени число;
- `floor(`*число*`)` — округление числа вниз до ближайшего целого;
- `log(`*число*`)` — натуральный логарифм числа;
- `max(`*число1*`, `*число2*`)` — большее из чисел *число1*, *число2*;
- `min(`*число1*`,`*число2*`)` — меньшее из чисел *число1*, *число2*;
- `pow(`*число1*`, `*число2*`)` — *число1* в степени *число2*;

- `random()` — случайное число между 0 и 1;
- `round(`*число*`)` — округление числа до ближайшего целого;
- `sin(`*число*`)` — синус числа;
- `sqrt(`*число*`)` — квадратный корень из числа;
- `tan(`*число*`)` — тангенс числа.

Приведем конкретные примеры.

- Метод `random()` возвращает случайное число, лежащее в интервале от 0 до 1:
  - получить случайное число в пределах от 0 до `Nmax` можно, написав выражение

    ```
    x = Nmax*Math.random()
    ```

  - получить случайное число в интервале от `Nmin` до `Nmax` можно из элементарного отношения пропорций:

    ```
    x = Nmin + (Nmax − Nmin)*Math.random()
    ```

  - создать функцию для вычисления случайного числа в заданном интервале можно так:

    ```
    function rand(a, b) {
     return a + (b - a)*Math.random()
    }
    ```

    Эта функция может потребоваться, например, для внесения некоторой непредсказуемости (нерегулярности) перемещения элементов на Web-странице, выбора цветов для мигающей надписи и т. п.
- Вычислить значение тригонометрической функции `sin(x)`, у которой аргумент `x` выражен в градусах, можно так:

  ```
  Math.sin(Math.PI*x/180)
  ```

3.11.4. Функции для решения некоторых математических задач

Язык JavaScript можно использовать не только для управления Web-страницами, но и для решения разного рода расчетных и математических задач. С другой стороны, составление или анализ соответствующих программ — хорошее упражнение в программировании. Поэтому предлагаю вам рассмотреть несколько несложных, но полезных программ расчетного характера.

Решение квадратного уравнения

Эта задача знакома нам еще со школы. Она, очевидно, относится к математике и довольно проста. Однако кажущаяся простота куда-то "улетучивается" при попытке рассмотреть ее с точки зрения программирования. Напомню формулировку задачи. Требуется найти два числа x_1 и x_2 такие, чтобы при подстановке любого из них в выражение $ax^2 + bx + c$ оно было бы равно нулю. Такие x_1 и x_2 называются корнями уравнения $ax^2 + bx + c = 0$. Все сказанное относится к математической постановке задачи.

На первый взгляд может показаться, что для решения задачи о квадратном уравнении следует просто вспомнить или найти в справочнике подходящую формулу. И действительно, это сделать нетрудно. Однако в программах для решения задач такого рода часто значительную долю кода занимают различные проверки и анализ исходных данных. Организация же вычислений по конечным формулам из справочника — не самое сложное. Так, в рассматриваемой задаче в зависимости от значений коэффициентов *a*, *b* и *c* возможны различные частные случаи. Корни могут быть компле́ксными и действительными. Допустим, нас интересуют только действительные корни. Тогда корней может и не быть совсем, или существовать только один, или два, одинаковые или различные. Таким образом, прежде чем мы дойдем до применения формулы, придется проанализировать, с каким именно случаем имеем дело.

Очевидно, функция (назовем ее `beq()`) для вычисления действительных корней квадратного уравнения должна принимать три параметра — коэффициенты `a`, `b` и `c`. Возвращаемым значением будет массив. Если корней нет, то этот массив сделаем пустым. В противном случае в нем будет два элемента. Если уравнение имеет два корня, то запишем их в качестве элементов возвращаемого массива. Если корень единственный, то он будет записан в первом элементе массива, а второй элемент будет пустым (`null`).

В листинге 3.45 приведен один из возможных вариантов определения функции.

Листинг 3.45

```
function beq(a, b, c){   // решение квадратного уравнения
 var aret = new Array();
 var D = b*b – 4*a*c;
 if (a == 0) {
  if (!(b == 0)){
   aret[0]=-c/b;
   aret[1]=null;
  }
```

```
  return aret   // корней нет или корень единственный
 }
 if (D == 0) {   // одинаковые корни
  aret[0]= -b/2/a;
  aret[1]= aret[0];
 }
 if (D>0){     // различные корни
  aret[0]=(-b - Math.sqrt(D))/2/a;
  aret[1]=(-b + Math.sqrt(D))/2/a;
 }
 return aret
}
```

Проверим работу функции на нескольких наборах исходных данных:

```
beq(0, 2, 6);   // массив [-3, null]
beq(1, -2, 1);  // массив [1, 1]
beq(3, 4, -2.5); // массив [-1.797054997187545, 0.4637216638542114]
beq(2, 0, 5);   // пустой массив
```

Вычисление интеграла

Интеграл от некоторого выражения $f(x)$ с одной переменной x на интервале ее значений от a до b можно понимать как площадь, ограниченную кривой $y = f(x)$, осью абсцисс x и двумя прямыми, параллельными оси ординат y и проходящими через точки a и b на оси x. Аналогия с площадью справедлива лишь отчасти. Если указать, что площадь под кривой, проходящей ниже оси абсцисс, отрицательная, тогда все в порядке.

Итак, для вычисления интеграла от выражения требуется определить площадь некоторой замысловатой фигуры. Алгоритм решения этой задачи основан на простой идее разбить сложную фигуру на множество простых, площадь которых легко рассчитать, а затем просуммировать эти площади. Здесь возникает вопрос о точности такого решения. Однако легко заметить, что чем меньше элементарные фигуры и чем больше их количество, тем ближе результат к действительной площади исходной фигуры. Это так называемая задача приближения или аппроксимации.

В качестве элементарной фигуры можно взять прямоугольник или трапецию, площади которых легко вычислить. Однако трапеция лучше, чем прямоугольник, вписывается в кривую, поэтому выберем именно ее. Напомню: чтобы найти площадь трапеции, необходимо умножить полусумму ее параллельных сторон на высоту. Для получения множества таких трапеций, требу-

ется разбить участок *ab* оси абсцисс на множество элементарных отрезков. В математике все идеально: отрезок *ab* разбивается на элементарные части, длина которых стремится к нулю, т. е. является бесконечно малой величиной. На практике мы имеем дело с конечными величинами. Более того, мы должны учесть, что с уменьшением длины элементарных отрезков и ростом их количества увеличивается время вычислений. Таким образом, нам потребуется найти компромисс между точностью решения задачи и временными затратами.

Теперь займемся технической стороной вопроса. Что следует передавать нашей функции в качестве параметров? Я вспомнил о великолепной встроенной функции `eval()`, которая может вычислять выражения JavaScript, переданные ей в виде строки. Поэтому функция `integral()` для вычисления интеграла будет принимать строку, содержащую выражение вида $f(x)$, в котором переменную обозначим строчной латинской буквой `x`, например, `"5*x*x+10"`. Два других параметра — числа `a` и `b`, соответствующие концам интервала интегрирования. Определение функции `integral()` приведено в листинге 3.46.

Листинг 3.46

```
function integral(expression, a, b){  // интеграл
 var x, y1, y2, n, length, dx, S=0;
 length = Math.abs(b - a);
 n = 100;          // количество элементарных отрезков
 dx = length/n;    // длина элементарного отрезка
 x = a;            // начальное значение переменной x в выражении
 y1 = eval(expression); // вычисление выражения
 x = a + dx;
 y2 = eval(expression); // вычисление выражения
 S = (y1 + y2)*dx/2;
 for(i = 2; i<=n;i++){
  y1 = y2;
  x = x + dx;
  y2 = eval(expression); // вычисление выражения
  S+= (y1 + y2)*dx/2;
 }
 return S
}
```

Проверим, как работает эта функция:

```
integral("x", 0, 1);   /* 0.5000000000000002 (точное значение=0.5) */
integral("x*x", 0, 1); /* 0.33335000000000037
         (точное значение=0.3333...) */
integral("x*x", 0, 10); /* 333.34999999999917
          (точное значение=333.3333...) */
```

Если точность вычислений нас не устраивает, то это обусловлено недостаточным количеством отрезков разбиения интервала *ab* (мала величина `n`). При большой длине этого интервала `n=100` будет явно мало. Конечно, можно передавать это значение функции в качестве четвертого параметра. Однако я поступил иначе: если длина интервала интегрирования больше 2, то `n` вычисляется по некоторой формуле, а иначе `n` равно 100. Кроме того, я добавил проверку предельных случаев, чтобы на границах не возникало ошибок. Итоговый код иллюстрирует листинг 3.47.

Листинг 3.47

```
function integral(expression, a, b){ // интеграл
 if (!expression || !b&&!a) return 0;
 if (a == b) return 0;
 var x, y1, y2, n, length, dx, S=0;
 length = Math.abs(b - a);
 y1 = Math.min(a,b); // если левый предел больше левого
 b = Math.max(a,b);  // если правый предел меньше левого
 a = y1;
 n = 100;
 if (length>2) n=Math.round(100*Math.log(length+1));
 dx = length/n;   // длина элементарного отрезка
 x = a;       // начальное значение переменной x в выражении
 y1 = eval(expression);
 x = a + dx;
 y2 = eval(expression);
 S = (y1 + y2)*dx/2;
 for(i=2; i<=n; i++){
  y1 = y2;
  x = x + dx;
  y2 = eval(expression);
  S+= (y1 + y2)*dx/2;
 }
return S
}
```

Выполним проверку:

```
integral("x*x", 0, 10)  /* 333.33749999999906
          (точное значение=333.3333...) */
```

Заметьте, что точность после нашей коррекции увеличилась на порядок.

Вычисление производной

Производная выражения (функции) $f(x)$ с одной переменной x в некоторой точке a интерпретируется как скорость изменения значения этого выражения при значении x, равном a. Например, если выражение $f(x)$ описывает зависимость пройденного пути от времени, то производная от $f(x)$ в момент времени t равна скорости движения в этот момент.

Алгоритм решения этой задачи таков: необходимо вычислить значения выражения в двух тестовых точках, расположенных рядом с заданной, а затем взять разность этих величин и разделить ее на расстояние между тестовыми точками. Чем ближе расположены тестовые точки к заданной, тем точнее получается результат. Мы выберем их так, чтобы заданная точка находилась посередине между тестовыми точками.

Как и интегрирование, вычисление производной будет базироваться на встроенной функции `eval()`, возвращающей значение выражения, переданное ей в виде строки как параметр. Функция `Dydxl()` для вычисления производной принимает строку, содержащую выражение, в котором переменная обозначена строчной латинской буквой `x`, например, `"25*x-2"`. Второй параметр указывает точку, в которой следует вычислить значение производной. Вариант кода приведен в листинге 3.48.

Листинг 3.48

```
function Dydx(expression,a){  // производная
 if (!expression) return 0;  // если нет выражения
 var x, y, dx;
 dx = 0.000025;
 x = a — dx;            // 1-я тестовая точка
 y = eval(expression);
 x = a + dx;            // 2-я тестовая точка
 return (eval(expression) — y)/2/dx)
}
```

Проверочные примеры:

```
Dydx("x",1);  // 0.9999999999998899 (точное значение равно 1)
Dydx("x*x",1); // 1.9999999999997797 (точное значение равно 2)
```

Поиск экстремума

Экстремум выражения (функции) $f(x)$ — это пара чисел (x_0, y_0), таких, что при подстановке x_0 в выражение $f(x)$ вместо переменной x значение этого выражения будет равно своему максимальному или минимальному значению y_0. Число экстремумов может быть различным (они могут вообще отсутствовать). Мы напишем код функции `extremum()`, которая вычисляет экстремум выражения, если он действительно у него есть (график этого выражения имеет "холм" или "впадину").

Алгоритм решения задачи основан на том, что значение производной выражения в точке экстремума или перегиба равно нулю.

Как и в предыдущих примерах, при поиске экстремума будет использоваться встроенная функции `eval()`, возвращающая значение выражения, переданного ей в виде строки как параметр. Функция `extremum()` для поиска экстремума принимает строку, содержащую выражение, в котором переменная обозначена строчной латинской буквой `x`, например, `"5*x*x-2*x"`. Второй и третий параметры указывают границы интервала, в котором следует искать x_0. Последний, четвертый параметр задает точность поиска. Вариант кода иллюстрирует листинг 3.49.

Листинг 3.49

```
function extremum(expression, a, b, d){  // экстремум
 if (!d) d=0.001;
 var x = a, y1, y2, i;
 y1 = Dydx(expression, a);
 i = 1;
 while ((x <= b)&&(i <= 1000)) {
  y2 = Dydx(expression, x+d);
  if ((y1<0)&&(y2>=0)||(y2<0)&&(y1>=0)) {
   x = x + d*Math.abs(y1)/(Math.abs(y1)+Math.abs(y2));
   return new Array(x, eval(expression))
  }
  x+= d;
  y1 = y2;
  i++
 }
 return new Array()
}
```

Проверочный пример:

```
extremum("x*x-1",-1, 1, 0.01)  /* (7.528699885739343e-16, -1)
                  точное значение равно (0, -1) */
```

Обратите внимание, что при наличии нескольких "холмов" или "впадин", функция определит лишь один из них, ближайший к точке *a*.

3.12. Дата и время

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

Со временем дела обстоят не так просто, как кажется на первый взгляд, поскольку существуют временны́е зоны (часовые пояса), а также сезонные поправки. Чтобы иметь возможность координировать деятельность во времени организаций и физических лиц в различных точках нашей планеты, была введена единая система отсчета, связанная с меридианом, проходящим через астрономическую обсерваторию в городе Гринвич в Великобритании. Эту временную зону называют средним временем по Гринвичу (Greenwich Mean Time, GMT). Кроме аббревиатуры GMT используется еще одна — UTC (Coordinated Universal Time, всеобщее скоординированное время).

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

В программе на JavaScript дата и время не являются литералами. Иначе говоря, нельзя написать, например, выражение `xdate = 10.03.2008`, чтобы получить значение даты, с которым в дальнейшем можно производить некие операции. Значения даты и времени создаются как экземпляры специального объекта `Date`. При этом объект сам будет "знать", что не бывает 31 июня и 30 февраля, а в високосных годах должно быть 366 дней.

3.12.1. Создание объекта *Date*

Итак, для работы с датами и временем необходимо, прежде всего, создать экземпляр встроенного объекта `Date`:

имяОбъектаДаты = new Date(*параметры*)

Здесь `имяОбъектаДаты` — переменная. Параметры могут отсутствовать.

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

имяОбъектаДаты.метод()

Если, например, объекту `Date` требуется присвоить новое значение, то для этого применяется соответствующий метод:

имяОбъектаДаты.метод(*новое_значение*)

Рассмотрим в качестве примера изменение значения года текущей системной даты (листинг 3.50).

Листинг 3.50

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

При создании объекта даты с помощью выражения `new Date()`, можно указать в качестве параметров, какую дату и время следует установить в этом объекте. Для этого существуют пять способов:

```
new Date("Месяц дд, гггг чч:мм:сс")
new Date("Месяц дд, гггг")
new Date(гг, мм, дд, чч, мм, сс)
new Date(гг, мм, дд)
new Date(милисекунды)
```

В первых двух вариантах параметры задаются в виде строки, в которой указаны компоненты даты и времени. Буквенные обозначения определяют шаблон параметров. Обратите внимание на разделители — запятые и двоеточия.

Время указывать не обязательно. Если компоненты времени опущены, то устанавливается значение 0 (полночь). Компоненты даты обязательно должны быть указаны. Месяц задается своим полным английским названием (аббревиатуры не допускаются). Остальные компоненты указывают в виде чисел. Если число меньше 10, то можно писать одну цифру, не записывая ведущий нуль (например, 3:05:32).

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

В последнем варианте задают целое число, которое представляет собой интервал в миллисекундах, начиная с 1 января 1970 года (с момента 00:00:00). Количество миллисекунд, отсчитанное от указанной стартовой даты, позволяет вычислить все компоненты и даты, и времени.

3.12.2. Методы объекта *Date*

Все многочисленные методы объекта `Date` (табл. 3.16) можно разделить на две категории: получения значений (их названия имеют префикс `get`) и установки новых значений (их названия имеют префикс `set`). В каждой категории выделяют две группы методов — для локального формата и формата UTC. Методы позволяют работать с отдельными компонентами даты и времени (годом, месяцем, числом, днем недели, часами, минутами, секундами и миллисекундами).

Таблица 3.16. *Методы объекта* `Date`

Метод	Диапазон значений	Описание
`getFullYear()`	1970 и далее	Год
`getYear()`	70 и далее	Год
`getMonth()`	0—11	Месяц (январь соответствует нулю)
`getDate()`	1—31	Число
`getDay()`	0—6	День недели (воскресенье соответствует нулю)
`getHours()`	0—23	Часы в 24-часовом формате
`getMinutes()`	0—59	Минуты
`getSeconds()`	0—59	Секунды
`getTime()`	0 и далее	Миллисекунды с 1.1.70 00:00:00 GMT

Таблица 3.16 *(продолжение)*

Метод	Диапазон значений	Описание
`getMilliseconds()`	0 и далее	Миллисекунды с 1.1.70 00:00:00 GMT
`getUTCFullYear()`	1970 и далее	Год UTC
`getUTCMonth()`	0—11	Месяц UTC (январь соответствует нулю)
`getUTCDate()`	1—31	Число UTC
`getUTCDay()`	0—6	День недели UTC (воскресенье соответствует нулю)
`getUTCHours()`	0—23	Часы UTC в 24-часовом формате
`getUTCMinutes()`	0—59	Минуты UTC
`getUTCSeconds()`	0—59	Секунды UTC
`getUTCMilliseconds()`	0 и далее	Миллисекунды UTC с 1.1.70 00:00:00 GMT
`setYear(`*знач*`)`	1970 и далее	Установка года (четырехзначного)
`SetFullYear(`*знач*`)`	1970 и далее	Установка года
`setMonth(`*знач*`)`	0—11	Установка месяца (январь соответствует нулю)
`setDate(`*знач*`)`	1—31	Установка числа
`setDay(`*знач*`)`	0—6	Установка дня недели (воскресенье соответствует нулю)
`setHours(`*знач*`)`	0—23	Установка часов в 24-часовом формате
`setMinutes(`*знач*`)`	0—59	Установка минут
`setSeconds(`*знач*`)`	0—59	Установка секунд
`setMilliseconds(`*знач*`)`	0 и далее	Установка миллисекунд с 1.1.70 00:00:00 GMT
`setTime(`*знач*`)`	0 и далее	Установка миллисекунд с 1.1.70 00:00:00 GMT
`setUTCFullYear(`*знач*`)`	1970 и далее	Установка года UTC
`setUTCMonth(`*знач*`)`	0–11	Установка месяца UTC (январь соответствует нулю)
`setUTCDate(`*знач*`)`	1—31	Установка числа UTC

Таблица 3.16 *(окончание)*

Метод	Диапазон значений	Описание
`setUTCDay(`*знач*`)`	0—6	Установка дня недели UTC (воскресенье соответствует нулю)
`setUTCHours(`*знач*`)`	0—23	Установка часов UTC в 24-часовом формате
`setUTCMinutes(`*знач*`)`	0—59	Установка минут UTC
`setUTCSeconds(`*знач*`)`	0—59	Установка секунд UTC
`setUTCMilliseconds(`*знач*`)`	0 и далее	Установка миллисекунд UTC с 1.1.70 00:00:00 GMT
`getTimezoneOffset()`	0 и далее	Разница в минутах по отношению к GMT/UTC
`toDateString()`	–	Строка с датой (без времени) в формате браузера (IE5.5)
`toGMTString()`	–	Строка с датой и временем в глобальном формате
`toLocaleDateString()`	–	Строка с датой без времени в локализованном формате системы (NN6, IE5.5)
`toLocaleString()`	–	Строка с датой и временем в локализованном формате системы
`toLocaleTimeString()`	–	Строка со временем без даты и в локализованном формате системы (NN6, IE5.5)
`toString()`	–	Строка с датой и временем в формате браузера
`toTimeString()`	–	Строка со временем без даты в формате браузера (IE5.5)
`toUTCString()`	–	Строка с датой и временем в глобальном формате
`Date.parse("дата")`	–	Преобразование строки с датой в число миллисекунд
`Date.UTC(`*знач*`)`	–	Преобразование строки с датой GMT в число

Вычислять разность двух дат или создавать счетчик времени, оставшегося до некоторого заданного срока, можно с помощью методов как для локального формата, так и UTC. Не следует применять выражения, сочетающие различ-

ные форматы времени, поскольку результаты могут оказаться неправильными. Формат UTC обычно применяется в расчетах, учитывающих часовой пояс.

> ***Замечание***
>
> Следует также принять во внимание, что нумерация месяцев, дней недели, часов, минут и секунд начинается с нуля. Для компонентов времени это естественно. Однако при такой нумерации декабрь оказывается одиннадцатым месяцем в году, а не двенадцатым, как это принято повсеместно. Воскресенье (Sunday) является нулевым днем недели, а не седьмым.

Значение года XX века представляется в двухзначном формате как разность между этим годом и 1900. Например, 1996 год представляется как 96. Годы до 1900 и после 1999 обозначаются в четырехзначном формате. Например, 2008 год нужно так и писать — 2008, поскольку 08 это 1908 год. Метод `getFullYear()` возвращает четырехзначное значение года.

Допустим, что сейчас 2008 год.

Примеры

```
today = new Date();      // текущие дата и время
Year = today.getYear();   // 2008
today = new Date(98,11,6); /* объект даты today содержит
              информацию о дате 6 ноября 1998 года,
              00:00:00 */
Year = today.getYear();   // 98
today.getFullYear();   // 1998
```

Любой компонент даты изменяется с помощью соответствующего метода, название которого начинается с приставки `set`. При этом значения других компонентов пересчитываются автоматически.

В следующем примере создается объект `Date`, содержащий некоторую конкретную дату. Затем мы устанавливаем новое значение года. При этом в объекте даты изменяется и число.

Пример

```
mydate = new Date(1952, 10,6);  /* "Thu Nov 6 00:00:00 UTC+0300 1952" */
myday = mydate.getDay();      // 6
mydate.setYear(2008);       // установка 2008 года
myday = mydate.getDay()      // 4
```

При попытке отобразить значение объекта `Date` оно автоматически преобразуется в строку методом `toString()`. Формат этой строки зависит от операционной системы и браузера. Например, для Windows XP и Internet Explorer 5.5+ строка, содержащая информацию объекта даты, имеет следующий вид:

```
Thu Feb 17 16:52:45 UTC+0300 2008
```

т. е. четверг, февраль, 17, 16 часов 52 минуты 45 секунд Всеобщего времени, скорректированного на 3 часа, 2008 год.

Если коррекция времени с учетом часового пояса не требуется, то для строкового представления даты и времени можно воспользоваться методом `toLocaleString()`:

```
mydate = new Date();
mydate.toLocaleString();    // "17 февраля 2008 г. 14:13:36"
```

Для представления в виде строк отдельно даты и времени имеются еще два метода:

```
mydate = new Date();
mydate.toLocaleDateString();  // "17 февраля 2008 г."
mydate.toLocaleTimeString();  // "14:13:36"
```

Заметим, что формат представления даты и времени, обеспечиваемый методами `toLocalDateString()` и `toLocalTimeString()`, зависит от настроек операционной системы и браузера. Если вы с помощью этих методов выведете дату и время на Web-страницу, то они будут выглядеть так, как пользователь привык их видеть на своем компьютере.

С датой иногда приходится выполнять различные вычисления, такие как определение даты через заданное количество дней от текущей или количество дней между двумя датами. Вот здесь как раз и требуется предварительный подсчет миллисекунд, содержащихся в минуте, часе, сутках и т. д.

Пример

```
// Определение даты, которая наступит через неделю относительно текущей
   week =1000*60*60*24*7       /* количество миллисекунд в неделе —
                               604800000 */
   mydate=new Date();          // объект Date с текущей датой
   mydate_ms=mydate.getTime();    /* текущая дата, представленная
                               количеством миллисекунд
                               от 1.01.1970 00:00:00 */
```

```
mydate_ms += week;          // maydate_ms=mydate_ms + week
mydate.setTime(mydate_ms);     /* установка новой даты в объекте
                    mydate */
newdate=mydate.toLocaleString() /* новая дата в виде строки */
```

Теперь определим количество дней между двумя датами, например, 10 февраля 2008 и 5 марта 2008. Для этого создадим сначала два соответствующих объекта даты:

```
date1=new Date(2008,01,10)
date2=new Date(2008,02,5)
```

Переменные `date1` и `date2` хранят длинные строки, содержащие даты (например, Thu `Feb 10 00:00:00 UTC+0300 2008`). Чтобы перевести их в миллисекунды, воспользуемся методом `parse()` объекта `Date`. Затем вычислим разность `Date.parse(date2)` и `Date.parse(date1)` и разделим ее на число миллисекунд в одних сутках:

```
days=(Date.parse(date2) - Date.parse(date1))/1000/60/60/24
 // результат: 24
```

Часто приходится иметь дело с датами, представленными в виде строк в формате "*дд.мм.гггг*", причем месяцы нумеруются с 1 до 12, а не с 0 до 11. В этом случае, чтобы воспользоваться методами объекта `Date`, необходимо предварительно выполнить соответствующие преобразования.

Допустим, исходная дата `strdate` представляется в виде строки в формате "*дд.мм.гггг*", а объект `Date` создан в формате "*гг, мм, дд*". Тогда необходимые преобразования исходных данных выглядят следующим образом:

```
astrdate=strdate.split(".");  /* массив подстрок, полученных из строки
                 strdate с использованием точки
                 как разделителя */
/* создаем объект Date */
mydate=new Date(astrdate[2], astrdate[1]-1, astrdate[0])
```

Развивая идею, рассмотренную в предыдущем примере, мы могли бы создать функцию, принимающую в качестве параметра строку, содержащую дату и время в привычном для нас формате "*дд.мм.ггг чч:мм:сс*" и возвращающую объект даты (листинг 3.51). Будем считать, что в строке параметра нашей функции дата отделена от времени одним пробелом, однако допускается, что информации о времени вообще может не быть. Далее, если время присутствует, секунды не обязательно должны указываться. Если параметр является пустой строкой или вообще отсутствует, то функция возвращает объект с текущей датой.

Листинг 3.51

```
function mydate(strdate){
 if (strdate == "" || strdate == null) {
  return new Date()
 }
 var astrdate=strdate.split(" ")
 if (astrdate.length == 1)
  astrdate[1] = "00:00:00"
 astrdate[0] = astrdate[0].split(".")
 astrdate[1] = astrdate[1].split(":")
 if (astrdate[1].length == 1)
  astrdate[1] = new Array(astrdate[1][0], "00","00")
 else {
  if (astrdate[1].length == 2)
   astrdate[1] = new Array(astrdate[1][0], astrdate[1][1] , "00")
 }
 return new Date(astrdate[0][2], astrdate[0][1]-1, astrdate[0][2],
astrdate[1][0], astrdate[1][1], astrdate[1][2])
}
```

Заметим, что в действующем коде выражение с последним оператором `return` необходимо написать в одну строку.

С использованием функции `mydate()` вычислить интервал (в днях) между двумя заданными датами можно следующим образом:

```
date1=mydate("10.2.2008");
date2=mydate("5.3.2008");
days=(Date.parse(date2) – Date.parse(date1))/1000/60/60/24;
```

Иногда в приложениях, в том числе и на Web-страницах, устанавливают часы (таймер), показания которых постоянно изменяются в соответствии с ходом системных часов. Чтобы сделать это, требуется в цикле создавать объект `Date` и формировать строку из составляющих времени, используя подходящие для этого методы. Однако при этом программа, в которой имеется код, выполняющий функцию часов, будет заниматься только этими часами (зациклится). Чтобы этого не произошло, задействуют метод `setInterval()` из объектной модели браузера.

Рассмотрим в качестве примера работы с объектом `Date` календарь, показанный на рис. 3.20. Календарь содержит текстовую область, в которой отображаются числа и названия дней недели, раскрывающийся список, из которого

можно выбрать месяц, текстовое поле для ввода года и несколько кнопок для переходов к другим месяцам и годам. С помощью элементов пользовательского интерфейса можно вывести календарь на любой месяц и год. Код программы представлен в листинге 3.52.

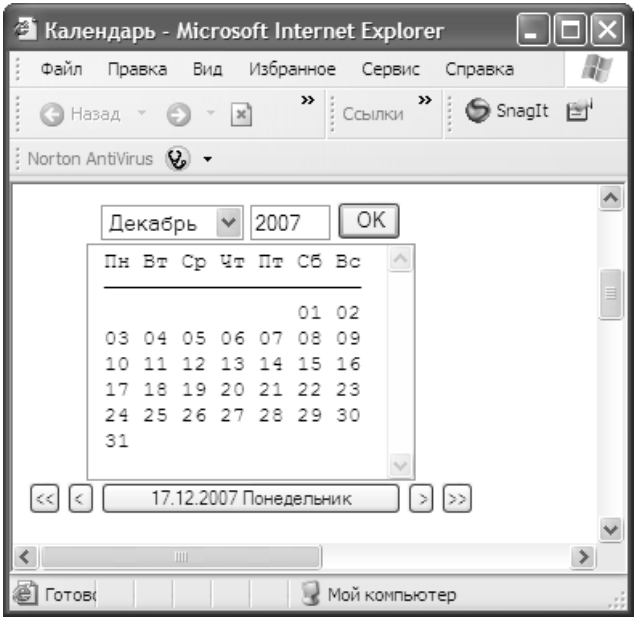

Рис. 3.20. Календарь

Листинг 3.52. Календарь

```
<HTML>
 <HEAD>
  <TITLE>Календарь</TITLE>
 </HEAD>
 <SCRIPT LANGUAGE="JavaScript">
  var nowdate;

  function setToday() {   // Установка текущей даты
   var now = new Date();
   var day = now.getDate();
   var month = now.getMonth();
   var year = now.getYear();
```

```
 var anameday = new Array("Воскресенье", "Понедельник",
   "Вторник", "Среда", "Четверг", "Пятница", "Суббота")
 var nameday=anameday[now.getDay()] // Название дня недели
 if (year < 2000) year = year + 1900;
 this.focusDay = day;
 document.calControl.month.selectedIndex=month;
 document.calControl.year.value=year;
 displayCalendar(month, year);
 nowdate = day+"." + parseInt(month+1) + "." + year;
 document.all["PrevYear2"].value=nowdate + " " + nameday;
}

function isFourDigitYear(year) {
 if (year.length != 4) {
  alert ("Год должен содержать 4 цифры");
  document.calControl.year.select();
  document.calControl.year.focus();
 } else { return true; }
}

function selectDate() {
 var year =document.calControl.year.value;
 if (isFourDigitYear(year)) {
  var day = 0;
  var month = document.calControl.month.selectedIndex;
  displayCalendar(month, year);
 }
}

function setPrevYear() {  // Предыдущий год
 var year = document.calControl.year.value;
 if (isFourDigitYear(year)) {
  var day = 0;
  var month = document.calControl.month.selectedIndex;
  year--;
  document.calControl.year.value = year;
  displayCalendar(month, year);
 }
```

```
}

function setPrevMonth() { // Предыдущий месяц
 var year =document.calControl.year.value;
 if (isFourDigitYear(year)) {
  var day =0;
  var month=document.calControl.month.selectedIndex;
  if (month == 0) {
   month=11;
   if (year > 1000) {
    year--;
    document.calControl.year.value=year;
   }
  } else { month-- }
  document.calControl.month.selectedIndex = month;
  displayCalendar(month, year);
 }
}

function setNextMonth() { // Следующий месяц
 var year = document.calControl.year.value;
 if (isFourDigitYear(year)) {
  var day = 0;
  var month = document.calControl.month.selectedIndex;
  if (month == 11) {
   month = 0;
   year++;
   document.calControl.year.value = year;
  } else { month++ }
  document.calControl.month.selectedIndex = month;
  displayCalendar(month, year);
 }
}

function setNextYear() {  // Следующий год
 var year=document.calControl.year.value;
 if (isFourDigitYear(year)) {
  var day = 0;
  var month = document.calControl.month.selectedIndex;
```

```
    year++;
    document.calControl.year.value = year;
    displayCalendar(month, year);
   }
  }

  function displayCalendar(month, year) {  // Показать календарь
   month=parseInt(month);
   year=parseInt(year);
   var i=0;
   var days = getDaysInMonth(month+1,year);
   var firstOfMonth =n ew Date (year, month, 1);
   var startingPos = firstOfMonth.getDay()-1;
   if (startingPos<0) startingPos = 6;
   days += startingPos;
   document.calButtons.calPage.value = " Пн Вт Ср Чт Пт Сб Вс";
   document.calButtons.calPage.value += "\n ———————————————————";
   for (i = 0; i < startingPos; i++) {
    if ( i%7 == 0 ) document.calButtons.calPage.value += "\n ";
    document.calButtons.calPage.value += "  ";
   }
   for (i=startingPos; i < days; i++) {
    if ( i%7 == 0 ) document.calButtons.calPage.value += "\n ";
    if (i-startingPos+1 < 10)
     document.calButtons.calPage.value += "0";
    document.calButtons.calPage.value += i-startingPos+1;
    document.calButtons.calPage.value += " ";
   }
   for (i=days; i < 42; i++) {
    if ( i%7 == 0 ) document.calButtons.calPage.value += "\n ";
    document.calButtons.calPage.value += "  ";
   }
   document.calControl.Go.focus();
  }

  function getDaysInMonth(month,year) { // Получить день в месяце
   var days;
   if (month==1 || month==3 || month==5 || month==7 || month==8 ||
month==10 || month==12) days=31;
```

```
   else if (month==4 || month==6 || month==9 || month==11) days=30;
   else if (month==2) {
    if (visYear(year)) { days=29; }  // Если год високосный
    else { days=28; }
   }
   return (days);
  }

  function visYear (Year) { // Високосные года
   if (((Year % 4)== 0) && ((Year % 100)!= 0) || ((Year % 400)== 0)) {
    return (true);
   } else { return (false); }
  }
</SCRIPT>
<FORM NAME="calControl" onSubmit="return false">
 <TABLE CELLPADDING=0 CELLSPACING=0 BORDER=0>
  <TR><TD COLSPAN=7>
   <CENTER>
   <SELECT NAME="month" onchange="selectDate()">
    <OPTION>Январь
    <OPTION>Февраль
    <OPTION>Март
    <OPTION>Апрель
    <OPTION>Май
    <OPTION>Июнь
    <OPTION>Июль
    <OPTION>Август
    <OPTION>Сентябрь
    <OPTION>Октябрь
    <OPTION>Ноябрь
    <OPTION>Декабрь
   </SELECT>
   <INPUT NAME="year" TYPE=TEXT SIZE=4 MAXLENGTH=4
      onchange="selectDate()">
  <INPUT TYPE="button" NAME="Go" value=" OK "
      onclick="selectDate()"
      TITLE="Если изменили год, щелкните здесь">
  </TD>
  </TR>
```

```
 </FORM>
 <FORM NAME="calButtons">
  <TR><TD align="center">
<TEXTAREA FONT="Courier"  NAME="calPage" WRAP=no ROWS=8 COLS=24 >
</TEXTAREA>
  </TD>
  <TR><TD>
   <CENTER>
   <INPUT TYPE=BUTTON NAME="PrevYear" VALUE="<<"
       style="font-size:9"
       onclick="setPrevYear()"
       TITLE="Год -">
   <INPUT TYPE=BUTTON NAME="PrevYear" VALUE="< "
       style="font-size:9"
       onclick="setPrevMonth()"
       TITLE="Месяц -">
   <INPUT TYPE=BUTTON NAME="PrevYear2" VALUE=""
       style="font-size:9"
       onclick="setToday()"
       TITLE="Календарь на текущую дату">
   <INPUT TYPE=BUTTON NAME="PrevYear"
       VALUE=" >"
       style="font-size:9"
       onclick="setNextMonth()"
       TITLE="Месяц +">
   <INPUT TYPE=BUTTON NAME="PrevYear" VALUE=">>"
       style="font-size:9"
       onclick="setNextYear()"
       TITLE="Год +">
   </CENTER>
  </TD></TR>
  </TABLE>
 </FORM>
 <! Запускаем функцию setToday >
 <SCRIPT>setToday()</SCRIPT>
</HTML>
```

В приведенном коде в разделе сценария реализован доступ к свойствам и методам элементов документа, а также обработка событий `onclick` (щелчок)

и onchange (изменение содержимого в поле ввода данных). Обработка событий, изменение свойств элементов и ряд других связанных с этим вопросов будут подробно рассмотрены в следующих главах.

3.13. Объекты

Хотя язык JavaScript объектно-ориентированный, данная технология программирования на этом языке редко применяется на практике. Во-первых, потому, что в JavaScript есть много обычных и более простых средств, вполне достаточных для решения типичных задач создания Web-приложений. Во-вторых, сценарии для Web-сайтов, как правило, настолько малы по объему, что трудно достичь явных преимуществ за счет применения объектно-ориентированной технологии. Вместе с тем, сценарии на JavaScript работают в основном с объектами — ассоциированными с типами данных (String, Number, Array и т. п.), а также объектами документа и браузера (document, window). Объектно-ориентированный стиль программирования оправдывается при разработке крупномасштабных Web-приложений. Как бы то ни было, знакомство с устройством объектов окажется полезным, даже если вы не будете создавать собственные объекты.

3.13.1. Создание объекта

Рассмотренные ранее встроенные объекты (String, Number, Array и т. п.) происходят от одного и того же корневого объекта Object. Вы можете создавать свои собственные (пользовательские) объекты, которые также будут базироваться на объекте Object.

Создать объект можно различными способами:

- ```
  имяОбъекта = new Object();
  имяОбъекта.свойство = значение;
  ```
- ```
  имяОбъекта = {свойство1: значение1, свойство2: значение2,...};
  ```
- ```
  function имя_конструктора(пар1,..., парN){
      код
  }
  имяОбъекта = new имя_конструктора(пар1,..., парN);
  ```

Для обращения к свойствам и методам объекта используется следующий синтаксис:

имяОбъекта.свойство

имяОбъекта.метод(*параметры*)
  ```

Рассмотрим перечисленные способы создания объектов подробнее. Допустим, например, что нам требуется создать объект `Сотрудник`, который содержал бы сведения о сотруднике некоторой фирмы, такие как `Имя`, `Отдел` и `Телефон`. Эту задачу можно решить следующим образом:

```
Сотрудник = new Object();
Сотрудник.имя = "Джеймс Бонд";
Сотрудник.отдел = "9";
Сотрудник.телефон = "123-456";
```

На рис. 3.21 показан пример обращения к свойствам объекта `Сотрудник`.

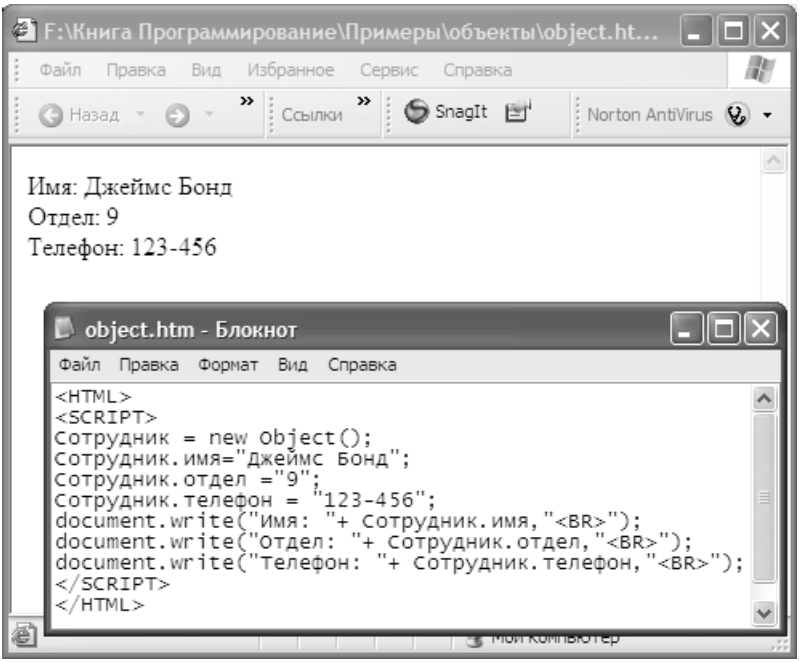

Рис. 3.21. Пример обращения к свойствам объекта

Тот же объект можно создать и таким способом:

```
Сотрудник = {
 имя:"Джеймс Бонд",
 отдел:"9",
 телефон:"123-456"
}
```

Рассмотренные способы задания объекта являются по существу определением экземпляра объекта `Object`. Если в фирме несколько сотрудников, то для каждого из них потребуется создать свой объект, что не всегда удобно. Хотелось бы создать один объект `Сотрудник`, задающий структуру данных о множестве сотрудников, а каждого из них представлять соответствующим экземпляром объекта `Сотрудник`. Такой подход обеспечивается посредством определения конструктора объекта.

Конструктор объекта — это функция, в теле которой задаются свойства и методы объекта — переменные и функции с префиксом `this`. Ключевое слово `this` (этот) представляет ссылку на текущий, т. е. определяемый конструктором объект. Конструктору, как и обычной функции, можно передать параметры. Листинг 3.53 содержит конструктор уже рассматривавшегося объекта `Сотрудник`.

Листинг 3.53

```
function Сотрудник(имя, отдел, телефон) {
 this.имя = имя;
 this.отдел = отдел;
 this.телефон = телефон;
 }
```

Здесь в коде конструктора задаются свойства объекта, которым присваиваются значения передаваемых ему параметров.

Экземпляры объекта создаются вызовом конструктора с ключевым словом `new`:

```
agent007 = new Сотрудник("Джеймс Бонд", 9, "123-456");
Shtirlitz = new Сотрудник("Максимов", 4, "765-4321");
```

Здесь переменные `agent007` и `Shtirlitz` — имена объектов, являющихся различными экземплярами одного и того же объекта `Сотрудник`.

Доступ к свойствам этого объекта производится обычным способом:

```
agent007.имя       // "Джеймс Бонд"
Shtirlitz.отдел    // 4
```

В рассмотренном примере конструктор принимает параметры. Если при вызове конструктора параметры не передавать, то свойства будут иметь значения `undefined`. Для свойств можно задать значения по умолчанию, прини-

маемые в том случае, если параметры в вызове конструктора не указаны или имеют следующие значения: пустая строка, 0, `false`, `null`, `undefined`. Значения по умолчанию указывают с помощью символа `||`, как в листинге 3.54.

Листинг 3.54

```
function Сотрудник(имя, отдел, телефон) {
this.имя = имя || "неизвестно";
this.отдел = отдел || 0;
this.телефон = телефон || "xxx-xxxx";
}
```

Теперь модифицируем код конструктора объекта `Сотрудник`, добавив к нему метод `resume()`, который возвращает строку со значениями всех свойств объекта (листинг 3.55).

Листинг 3.55

```
function Сотрудник(имя, отдел, телефон){
 this.имя = имя;
 this.отдел = отдел;
 this.телефон = телефон;
 this.resume = function() {
   return this.имя+", отдел "+this.отдел+", тел."+this.телефон
 }
}
```

Чтобы изменить значение свойства экземпляра объекта, достаточно воспользоваться выражением:

```
имяОбъекта.свойство = значение;
```

Например,

```
Shtirlitz.имя = "Штирлиц";
```

Чтобы добавить новое свойство к экземпляру объекта, достаточно воспользоваться аналогичным выражением:

```
имяОбъекта.новое_свойство = значение;
```

Например, выражение `agent007.зарплата=10000` добавляет свойство `зарплата` со значением 10000 только экземпляру `agent007` объекта `Сотрудник`, а экземпляр `Shtirlitz` этого свойства не имеет.

Кроме так называемого точечного синтаксиса обращения к свойствам объекта (*имяОбъекта.свойство*) существует синтаксис обращения к элементам ассоциативного массива: `имяОбъект["свойство"]`. Например, `Shtirlitz["имя"] = "Максимов"`.

Добавить новое свойство сразу всем существующим экземплярам объекта можно через так называемый прототип. Каждый объект имеет свойство `prototype`, являющееся ссылкой на базовый объект `Object`. Используя `prototype`, можно добавить в прототип конструктора данные и программный код, которые должны быть общими для всех экземпляров объекта.

Пусть, например, требуется добавить свойство `пол` со значением "муж." в объект `Сотрудник` (а значит, и во все его экземпляры). Это можно сделать так:

```
Сотрудник.prototype.пол="муж.";
```

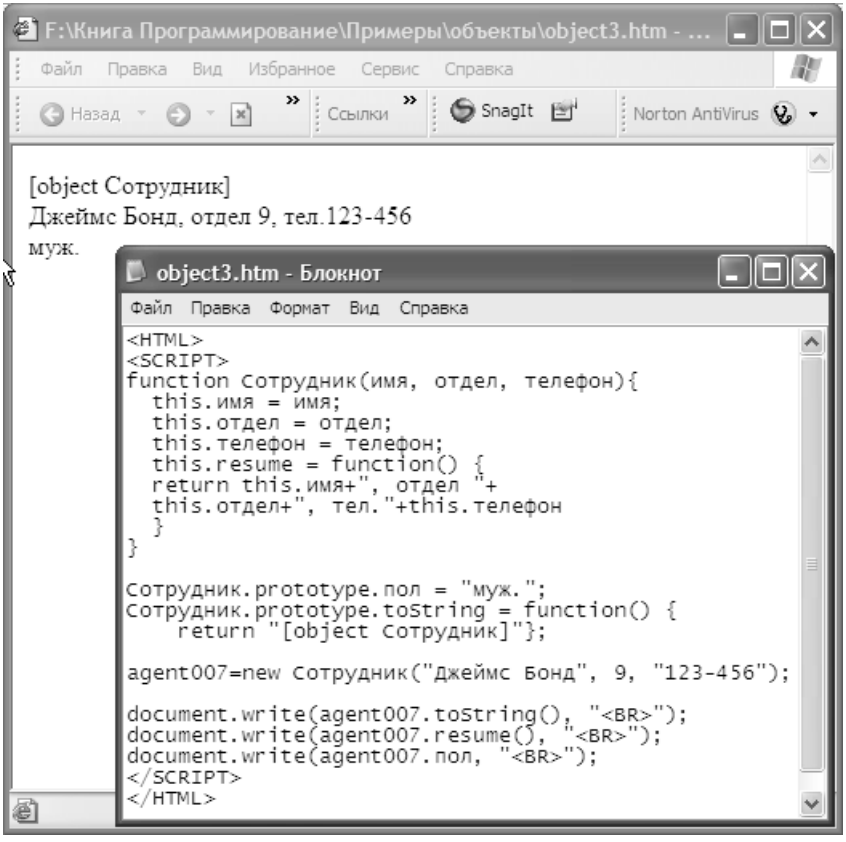

Рис. 3.22. Создание объекта и изменение его свойств через прототип

Аналогично добавляются и изменяются методы. Так, все объекты имеют метод `toString()`, который для пользовательского объекта возвращает малоинформативную строку "[object Object]". Если вы хотите, чтобы для объекта `Сотрудник` данный метод возвращал строку "[object Сотрудник]", то следует выполнить такой код:

```
Сотрудник.prototype.toString = function() {
                        return "[object Сотрудник]"};
```

На рис. 3.22 в качестве иллюстрации изложенного приведены код и результат его выполнения.

Два объекта можно связать таким образом, что один из них (дочерний) будет наследовать свойства другого (родительского). Наследование свойств объектов осуществляется через прототип. Допустим, мы хотим создать объект `Суперсотрудник`, отличающийся от рассмотренного ранее объекта `Сотрудник` только тем, что имеет дополнительное свойство `оружие`. Иначе говоря, `Суперсотрудник` должен унаследовать все свойства объекта `Сотрудник`, сохранив свои индивидуальные свойства. Для этого достаточно создать дочерний объект `Суперсотрудник`, содержащий только специфические (собственные) свойства, а затем через прототип связать его с родительским объектом `Сотрудник` (листинг 3.56).

Листинг 3.56

```
function Суперсотрудник(оружие){ /* содержит только специфические
                                    свойства */
this.оружие=оружие;
}
/* Присоединяем свойства родительского объекта */
Суперсотрудник.prototype = new Сотрудник("Пронин", 13, "777-7777");
```

Аналогично через цепочку прототипов можно связать несколько объектов.

На рис. 3.23 приведен пример, показывающий, что экземпляр объекта `Суперсотрудник` не только обладает пистолетом, но и свойствами родительского объекта `Сотрудник`.

Объекты являются наборами данных и программного кода и поэтому занимают некоторый объем памяти. Если объект или его экземпляр перестал быть нужным, то его можно удалить. Для этого достаточно просто присвоить ему значение `null`: `agent007 = null; Суперсотрудник = null`.

Рис. 3.23. Пример наследования свойств объекта через прототип

3.13.2. Свойства и методы объекта *Object*

Как уже отмечалось ранее, `Object` — базовый объект, на основе которого строятся все остальные. Свойствами и методами объекта `Object` обладают также и другие объекты, такие как `Array`, `String`, `Number`, `Function` и т. п., а также пользовательские объекты. С некоторыми из них (например, `prototype` и `toString()`) мы уже встречались. Так, единственное свойство `prototype` подробно рассматривалось в *разд. 3.13.1*. Здесь будут перечислены методы объекта `Object`.

- `hasOwnProperty("свойство")` — возвращает логическое значение, указывающее, имеет ли экземпляр объекта собственное *свойство* (не унаследованное через `prototype`).
- `isPrototypeOf(объект)` — возвращает логическое значение, указывающее, находится ли *объект* в цепочке прототипов данного объекта.

- `toString()` — возвращает строку; по умолчанию — "[object Object]". Иногда этот метод переопределяется, чтобы обеспечить больше информации (см. *разд. 3.3.1*).
- `valueOf()` — возвращает примитивное значение, ассоциированное с объектом; часто это строка "[object Object]".

На рис. 3.24 приведен пример использования указанных свойств. Так, объект `agent008` имеет собственное свойство `оружие`, но свойство `имя` он унаследовал от объекта `agent007`, а не приобрел как собственное при создании.

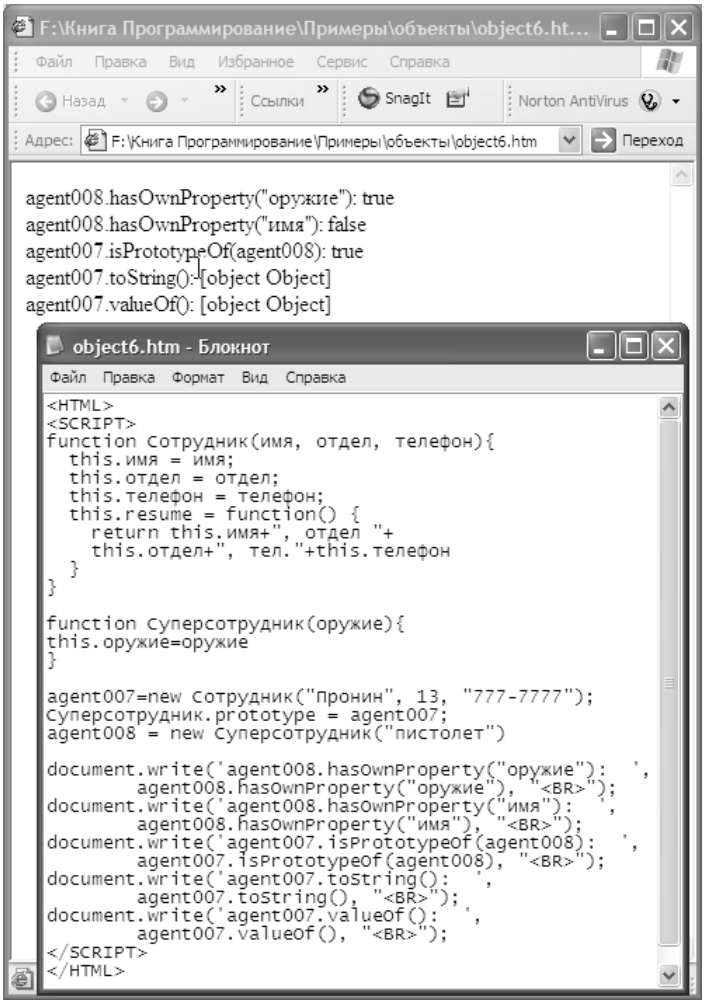

Рис. 3.24. Пример использования свойств объекта `Object`

3.13.3. Объектные операторы

При работе с объектами иногда удобны специальные операторы:

- `width (объект) {код}` — позволяет выполнить блок кода применительно к указанному объекту.

Пример

```
width(document){
write("<H1>Моя домашняя страница</H1>");
write("Здесь приводятся примеры скриптов на JavaScript");
write("<BR>");
write("Они могут пригодиться вам.")
}
```

Без данного оператора нам пришлось бы писать `document.write()`. При использовании оператора `width` важно, чтобы упоминаемые в блоке кода свойства и методы относились к объекту, указанному в качестве параметра.

- `for(переменная in объект) {код}` — оператор циклического перебора свойств объекта. Переменная может иметь любое имя.

На рис. 3.25 в качестве примера выводятся имена и значения объекта `agent007`, являющегося экземпляром объекта `Сотрудник`. Обратите внимание, что имя свойства читается в переменную `prop`, а значение данного свойства получается путем обращения к объекту как массиву: `agent007[prop]`.

Чтобы просмотреть свойства объекта `document`, представляющего загруженный в браузер документ, достаточно выполнить следующий код:

```
for (prop in document){
 document.write(prop,": " + document[prop],"<BR>")
}
```

- `свойство in объект` — возвращает логическое значение, указывающее, обладает ли объект данным свойством.

Пример

```
function Сотрудник(имя, отдел, телефон){
 this.имя = имя;
 this.отдел = отдел;
 this.телефон = телефон;
 this.resume = function() {
  return this.имя+", отдел "+
```

```
  this.отдел+", тел."+this.телефон
 }
}

function Суперсотрудник(оружие){
this.оружие=оружие
}

agent007=new Сотрудник("Штирлиц", 13, "123-456");
Суперсотрудник.prototype = agent007;
agent008 = new Суперсотрудник("пистолет");

alert("оружие" in agent008);      // true
alert("оружие" in agent007);      // false
```

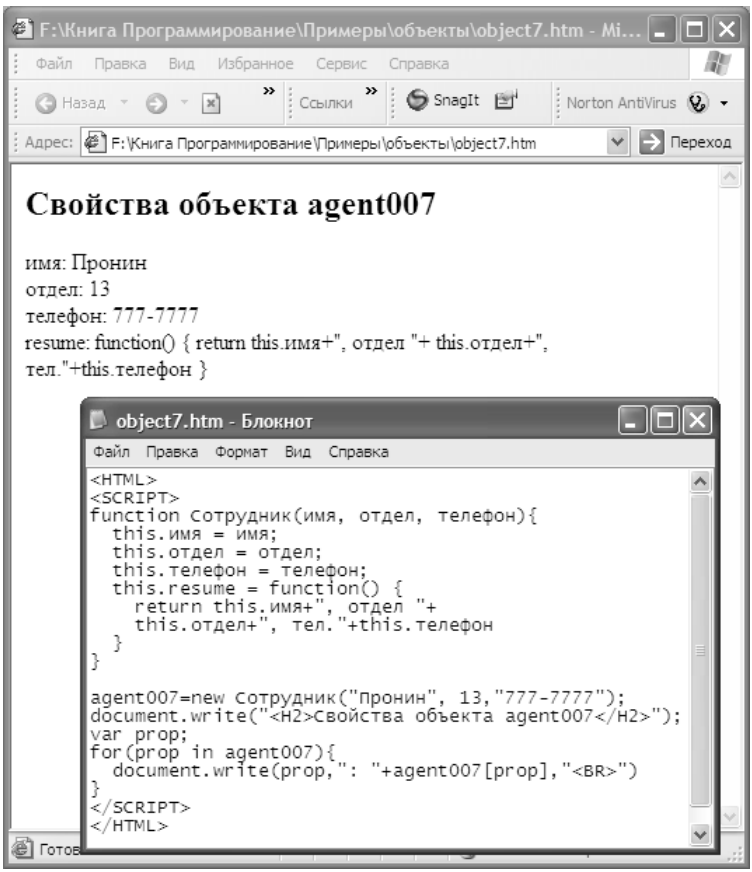

Рис. 3.25. Пример использования оператора `for(… in…)`

- `delete объект.свойство` — позволяет удалить свойство объекта. Обычно служит для удаления элементов массива (см. *разд. 3.10.1*).

Пример

```
function Сотрудник(имя, отдел, телефон){
 this.имя = имя;
 this.отдел = отдел;
 this.телефон = телефон;
 this.resume = function() {
  return this.имя+", отдел "+
  this.отдел+", тел."+this.телефон
 }
}

agent007 = new Сотрудник("Штирлиц", 13, "123-456");
alert("имя" in agent007);   // true
delete agent007.имя;
alert("имя" in agent007);   // false
```

- `объект1 instanceof объект2` — возвращает логическое значение, указывающее, принадлежит ли `объект1` объектной модели `объект2`.

Пример

```
function Сотрудник(имя, отдел, телефон){
 this.имя = имя;
 this.отдел = отдел;
 this.телефон = телефон;
 this.resume = function() {
  return this.имя+", отдел "+
  this.отдел+", тел."+this.телефон
 }
}

function Суперсотрудник(оружие){
this.оружие=оружие
}

agent007 = new Сотрудник("Штирлиц", 13, "123-456");
```

```
Суперсотрудник.prototype = agent007;
agent008 = new Суперсотрудник("пистолет");

alert(agent008 instanceof Object);      // true
alert(agent008 instanceof Сотрудник);     // true
alert(agent008 instanceof Суперсотрудник);   // true

alert(agent007 instanceof Object);      // true
alert(agent007 instanceof Сотрудник);     // true
alert(agent007 instanceof Суперсотрудник);   // false
```

3.14. Операторы обработки исключительных ситуаций

При выполнении некоторых операций могут возникнуть ошибки, даже если они не связаны с нарушением синтаксиса выражений языка. Это так называемые ошибки времени выполнения сценария. Например, сценарий может обратиться к объекту, который не поддерживается данным типом или версией браузера, пользователь мог ввести в поле вместо числа последовательность букв и т. п. В результате возникает исключительная ситуация, которую можно, как говорят, перехватить и обработать. Однако следует подчеркнуть, что ошибки, обусловленные нарушением синтаксиса, нельзя обрабатывать как исключительные ситуации. Это дело интерпретатора JavaScript, т. е. браузера, который в лучшем случае выведет диагностическое сообщение.

Перехват и обработку исключительных ситуаций можно осуществить с помощью такого оператора (листинг 3.57).

Листинг 3.57

```
try{
 код, ошибки выполнения которого перехватываются
} catch(e){
 код, выполняемый при перехвате ошибки
}
finally{
 код, выполняемый в любом случае
}
```

В блоке `try` (попытка) размещают ту часть программы, в которой могут возникнуть ошибки, которые вы желаете перехватить и каким-либо образом обработать. Собственно обработка ошибок выполняется в блоке `catch` (захват), которому в качестве параметра передаются сведения об ошибке (по умолчанию — объект ошибки). Здесь вы пишете программный код, который должен, по вашему мнению, осуществлять реакцию на возникшую исключительную ситуацию. В простейшем варианте это может быть вывод сообщения об ошибке. Блок `finally` не обязателен и содержит код, который будет выполнен после кода блоков `try` или `catch`.

Теперь рассмотрим пример. Известно, что попытка выполнить выражение `alert(x)`, где `x` — переменная, которая не объявлена с помощью оператора `var` или которой не присвоено значение, вызовет ошибку. Кстати, такого рода ошибки трудно выявить путем простого наблюдения за выполнением сценария в браузере. Следущий сценарий (листинг 3.58) перехватит эту ошибку и в качестве результата ее обработки выведет диалоговое окно с описанием ошибки.

Листинг 3.58

```
try{
 alert(x);
} catch(e){
 alert("Произошла ошибка:"+e.message);
}
```

Здесь `e.massage` — описание ошибки, возвращаемое объектом ошибки `Error`. Вместо `e` можно использовать любое имя переменной. В данной книге мы не будем рассматривать объект ошибки, а отметим только одно его важное свойство `message`, содержащее в качестве строкового значения описание ошибки. Это значение может быть тем или иным в зависимости от типа браузера. В приведенном примере значением `e.message` в Internet Explorer 7.0 является `"x" – определение отсутствует`, а в Mozilla Firefox 2.0 — `x is not defined`. Вы можете программно проанализировать строку сообщения об ошибке, чтобы принять решение о способе ее обработки.

Блок `try` может включать в себя другие блоки `try` и `catch`. Если таких конструкций несколько, то они могут быть только вложенными друг в друга. Вложенные блоки `try` и `catch` обычно применяют для обработки нескольких видов ошибок.

Исключительную ситуацию можно вызвать принудительно, чтобы передать управление блоку обработки `catch`. Для этого существует оператор

`throw` *значение*.

Указанное *значение* может быть любого типа, но рекомендуется использовать значение объекта ошибки или строковое сообщение об ошибке. Если это строковое значение, то оно будет передано в качестве параметра оператора `catch`. Листинг 3.59 иллюстрирует принцип применения операторов `try…catch` и `trow`. Более содержательный пример можно найти в *разд. 4.12.5*.

Листинг 3.59. Пример принудительного вызова исключительной ситуации

```
<HTML>
<HEAD><TITLE>try</TITLE></HEAD>
<SCRIPT>
try{
 var x;
 x=prompt("Введите число","");
 if (isNaN(x)){ // если не число
  throw "Вы ввели не число" // перейти к блоку catch
 }
} catch(e){
 alert(e); // сообщение, переданное throw
 x=0;
}
alert("x="+x);
</SCRIPT>
</HTML>
```

В рассматриваемом здесь примере, если пользователь введет в поле диалогового окна символы, не означающие в совокупности какое-нибудь число, то произойдет переход к блоку `catch` с передачей ему строки сообщения. Блок `catch` в этом случае выведет диалоговое окно с принятым сообщением и присвоит переменной `x` значение 0. Если же пользователь ввел число, то код в блоке `catch` не будет выполняться. Далее все идет своим чередом: выполняются выражения, расположенные за пределами оператора `try…catch`. В данном примере выводится диалоговое окно со значением переменной `x`. Разумеется, в своих проектах вы должны найти более разумное и, в конечном счете, эффективное применение конструкции `try…catch`.

В заключение замечу, что рассмотренная здесь в общих чертах языковая конструкция (т. е. оператор) `try…catch` довольно широко распространена в различных языках программирования, таких как PHP, C++, Pascal и др. Ее применяют в тех случаях, когда необходимо прежде проверить доступность какого-нибудь объекта, выбрать вариант доступа к нему (если это возможно в принципе), дождаться завершения соеденения с удаленным компьютером или выполнить какую-нибудь многоэтапную операцию, успех которой зависит больше от конъюнктуры и времени, чем от соблюдения элементарного синтаксиса языковых выражений. Так например, вы хотите получить данные от сервера. Но для их передачи в браузер требуется некоторое время, в течение которого может произойти что угодно. Вы, очевидно, должны дождаться завершения передачи данных прежде, чем начать заниматься манипуляциями с ними. Обычно программные средства обмена данными предоставляют программисту сообщения об этапах передачи информации (например, объект загружен частично, загрузка завершена с ошибками, загрузка завершена без ошибок и т. п.). Ваша программа может контролировать или нет все этапы выполнения многошаговой операции. Если контроль отсутствует, то вероятность "зависания" программы высока, что не свидетельствует о вашем профессионализме. Ведь "зависание" программы — это худшее, что может с ней произойти. Именно предусмотрительность во всем — один из важнейших признаков высокой квалификации программиста. Здесь и приходит на помощь конструкция `try…catch`.

Глава 4

Клиентские сценарии на JavaScript

4.1. Об объектной модели браузера и документа

4.1.1. Общие сведения

На языке JavaScript пишут в основном сценарии, выполняющиеся на стороне клиента, т. е. Web-браузером. Именно с такого рода сценариев начинают свое программистское творчество новички. Вот список типичных, но далеко не всех задач, которые решаются с помощью клиентских сценариев:

- проверка данных, введенных пользователем в форму, а также выполнение различного рода вычислений;
- обработка событий, вызванных манипуляциями мышью и клавиатурой;
- динамическое изменение элементов документа;
- запись и чтение информации cookie;
- управление окнами и фреймами;
- управление процессами во времени.

Чтобы создавать сценарии, необходимо не только знать синтаксис языка, но и объектные модели браузера и документа — иерархический набор объектов с их свойствами и методами, с которыми сценарии будут работать. Объекты являются программными единицами, представляющими в программной среде то, с чем имеет дело пользователь (элементы окна браузера и загруженного в него документа, например, тексты, картинки, формы и т. п.), и то, на что может влиять сценарий. Работа с объектами означает, по существу, чтение и/или изменение их свойств. Синтаксис доступа к объектам прост: `объект.свойство` или `объект.метод()`. До сих пор мы так и поступали *(см. главу 3)*. Однако объекты браузера и документа образуют иерархическую

структуру. Это означает, что одни объекты являются подобъектами или, иначе говоря, свойствами других. Например, объект, представляющий браузер, называется `window` (т. е. окно); в окно браузера загружается некий документ, который представляется объектом, называемым `document` и имеющим, среди прочих, свой собственный метод `write()`. Этот метод, как вы уже знаете, записывает в текущий документ то, что указано в качестве его параметра. Чтобы применить данный метод, мы должны добраться до него по иерархии объектной модели. Так, для записи в текущий документ слова "Привет!", в сценарий нужно поместить такое выражение: `window.document.write("Привет!")`. Здесь через точку мы указали, как добраться до метода `write("Привет!")`, принадлежащему объекту `document`, который, в свою очередь, является подобъектом или, точнее, свойством вышестоящего по иерархии объекта `window`. Поскольку объект `window` — самый верхний в иерархии, то его упоминание не обязательно. Другими словами, мы могли бы ограничиться более коротким выражением: `document.write("Привет!")`. Однако это лишь некая вольность, допускаемая интерпретатором языка (браузером) для удобства программирования, но не отменяющая общий принцип доступа к конечному свойству через упоминание последовательности всех объектов, лежащих на пути к нему. Так, в предыдущих главах мы нередко использовали метод `alert(`*сообщение*`)` для вывода сообщения в диалоговом окне. Поскольку это метод объекта `window`, без каких бы то ни было "поблажек" со стороны браузера мы должны были бы вызывать его следующим образом: `window.alert(`*сообщение*`)`.

Вот еще один пример. Элементы документа, соответствующие HTML-тегам, могут иметь атрибут `ID`, значения которого должны их однозначно идентифицировать. В рамках только HTML этот атрибут не несет специфической нагрузки, но очень важен для связи со сценариями, связанными с данным элементом. Если вы планируете работать с элементом посредством сценария, то введите для него атрибут `ID = "уникальное_значение"` и получите доступ к этому элементу с помощью выражения `document.getElementByID("уникальное_значение");`

или

```
document.all["уникальное_значение"];
```

Метод документа `getElementByID()` более универсален в том смысле, что он должен восприниматься всеми современными браузерами, уважающими стандарты и рекомендации W3C. Свойство (коллекция) `all` не относится к стандартным. Это фирменное средство обозревателя Internet Explorer, но поскольку для него написано очень много сценариев, оно воспринимается и другими браузерами, в частности, Mozilla Firefox и Opera.

Существуют и другие способы доступа к объектам. Далее мы подробно познакомимся, как правильно обращаться с объектами браузера и документа,

а сейчас давайте сделаем некоторое обобщение в словоупотреблении. Как уже говорилось, объект это нечто, имеющее свойства и методы. Свойства мы обычно ассоциировали с переменными, а методы — с функциями. Все это — внутренние переменные и функции, присущие некоторому объекту (контейнеру программного кода), а не всей программе. Впрочем, вы уже должны были привыкнуть к употреблению объектов в программном коде, если внимательно прочли *разд. 3.13*.

Мы можем называть свойства-переменные и свойства-методы просто свойствами, которые выражаются, в том или ином контексте, как пассивные (переменные) или как активные (методы). Тогда доступ к свойству или методу некоторого объекта можно осуществить по следующей общей формуле: *`объект.свойство…свойство`*. В этой цепочке только последнее свойство (переменная или функция) может что-то изменить в чувственном восприятии отображаемого браузером документа, а все промежуточные — только вехи на пути к видимой или невидимой реализации окончательных свойств. Для тех, кто чувствует себя комфортно в мире объектно-ориентированного программирования, на всякий случай замечу, что последовательность вида *`объект.свойство…свойство`* отнюдь не предполагает отношение наследования между свойствами-объектами. Это просто указание пути доступа к требуемому свойству (переменной или функции) и не более того.

Кроме набора объектов JavaScript, таких как `Array`, `String`, `Number`, `Date`, `Math` и др., имеются объектные модели браузера (BOM — Browser Object Model) и документа (DOM — Document Object Model).

Основные объекты браузера показаны на рис. 4.1. Объект `window` находится на самом верхнем уровне иерархии, он имеет свои свойства и методы. Кроме того, `window` содержит еще несколько объектов. Важнейший из них — `document`, состоящий из множества других объектов, представляющих свойства загруженного в браузер документа. Иерархически упорядоченная совокупность объекта `document` образует то, что называют объектной моделью документа (DOM). Объект `frames` представляет коллекцию (массив) фреймов. Каждый фрейм является частью (подокном) окна браузера и как объект устроен аналогично `window`. В частности, он может содержать объект `document`, соответствующий загруженному в него документу. Объект `history` содержит список последних URL-адресов, к которым обращался пользователь. Иначе говоря, `history` представляет собой то, что называется журналом браузера. Объект `location` содержит сведения о текущем расположении документа виде URL-адреса, а также о составных частях URL. В объекте `navigator` находится информация об основных характеристиках браузера, в том числе о его типе и версии. Именно из этого объекта извлекается информация о том, в каком браузере пользователь пытается открыть Web-страницу.

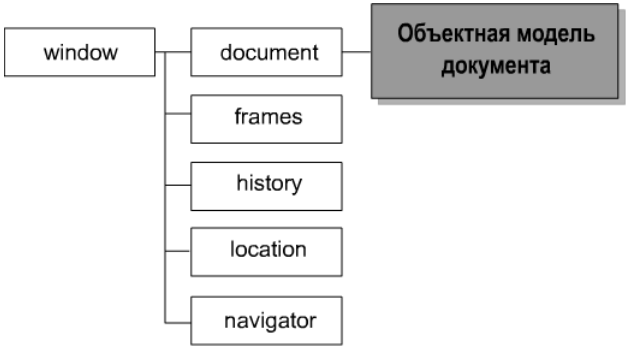

Рис. 4.1. Основные объекты браузера

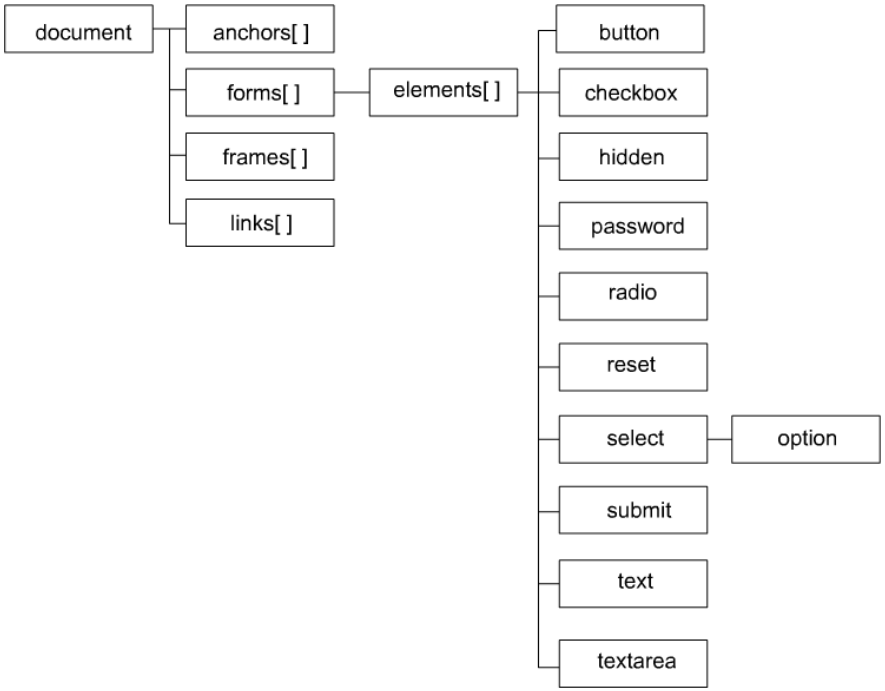

Рис. 4.2. Основные объекты документа

Объект `document` содержит информацию об элементах Web-страницы, таких как ссылки, графические изображения, фрагменты текста, кнопки, а также об их свойствах (цвете, размерах и др.). На рис. 4.2. показаны основные (но далеко не все) объекты, составляющие объектную модель документа. Объекты, около имен которых указана пара квадратных скобок, называют еще коллек-

циями — массивами, состоящими из других объектов. Так например, `forms[]` — это коллекция (массив) всех форм в текущем документе, а `links[]` — коллекция ссылок.

Разнообразие объектных моделей документа, поддерживаемых различными типами браузеров, а также версиями браузеров одного и того же типа, — главная причина несовместимости последних. Общая схема объекта `document` сложилась к 1996 г. и была воплощена в первых версиях браузеров Internet Explorer и Netscape. Эта схема в общих чертах и представлена на рис. 4.2. Далее разработчики Netscape и Microsoft усовершенствовали ее, добавив новые возможности и реализовав в браузерах Netscape 4*x* и Internet Explorer 4+. На рис. 4.3 показаны основные объекты документа, поддерживаемые в Internet Explorer 4+. Нетрудно заметить, что основные объекты исходной модели (рис. 4.2) сохранились, но добавлено много новых. Так, коллекция `all` содержит ссылки на все элементы документа, а `images` — на все графические изображения в документе.

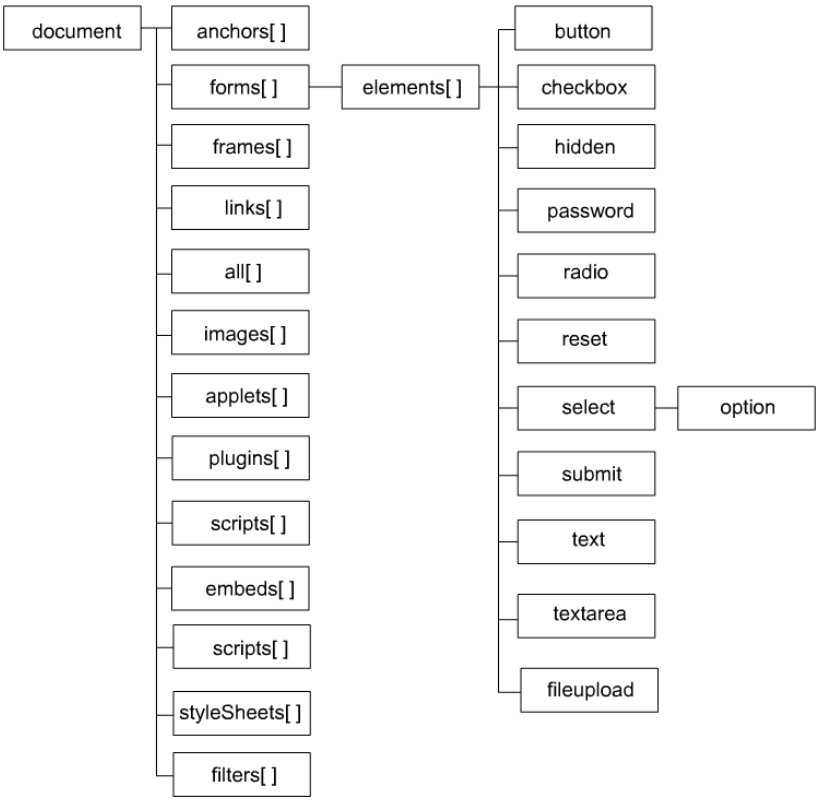

Рис. 4.3. Основные объекты документа в Internet Explorer 4+

Чтобы преодолеть несовместимость браузеров, консорциум W3C предложил стандарт, связывающий HTML- и XML-документы с иерархией объектов, предлагаемой программисту. Именно этот стандарт получил название "модель DOM". Сведения о ней можно получить по адресу **www.w3.org/DOM**.

Консорциум W3C, учитывая уже достигнутое Netscape и Microsoft, желая обеспечить межбраузерную совместимость, определил три уровня DOM:

1. DOM 0 — данный уровень практически полностью соответствует моделям, реализованным в Netscape 3.0 и Internet Explorer 3.0. Это классическая или традиционная объектная модель JavaScript.
2. DOM 1 — данный уровень обеспечивает работу со всеми элементами документа посредством стандартного набора функций. Здесь все части страницы доступны для чтения и изменения; обеспечиваются практически все возможности, предоставляемые коллекцией `all` в Internet Explorer. Данная модель стандартизирована и обеспечивает совместимость браузеров.
3. DOM 2 — обеспечивает более совершенные возможности доступа к элементам документа, объединяя модели нулевого и первого уровней и добавляя доступ к таблицам стилей. Данная форма DOM поддерживает улучшенную модель событий и некоторые другие расширения.

Все современные браузеры поддерживают стандарт DOM от W3C того или иного уровня, но пока не в полной мере. Однако основная тенденция развития браузеров, видимо, состоит в том, чтобы следовать этому стандарту в возможно большей степени. Вместе с тем, браузеры продолжают поддерживать и некоторые нестандартные фирменные расширения объектной модели. Последнее в наибольшей степени относится к Microsoft Internet Explorer. Поскольку в настоящее время существует огромное количество сценариев, корректно выполняющихся в Internet Explorer, другие браузеры также сохранили поддержку некоторых нестандартных элементов объектной модели.

В большинстве случаев при разработке сайтов, а также при начальном освоении программирования на JavaScript вполне можно ограничиться знанием и применением небольшого количества объектов браузера и документа, согласованных с объектными моделями одновременно всех наиболее популярных браузеров, таких как Internet Explorer, Mozilla и Opera. В данной книге мы поступим именно таким образом. Более подробную информацию об объектных моделях следует искать в справочниках, например, в широко известных книгах Д. Гудмана или Т. Пауэлла и Ф. Шнайдера, а также на сайте **www.w3.org**.

Далее мы рассмотрим свойства некоторых объектов, наиболее важных с точки зрения программирования. Причем будут приведены не все их свойства, а те, которые поддерживаются одновременно обозревателями Internet Explorer 4+,

Netscape 4*x*, Mozilla или соответствуют DOM от W3C. В некоторых случаях мы укажем некоторые очень полезные свойства, поддерживаемые только Internet Explorer.

4.1.2. Объект *window*

`Window` (окно) — объект высшего уровня, представляющий свойства окна браузера.

Свойства объекта *window*

Список основных свойств объекта `window` таков:

- `parent` — возвращает ссылку на окно (фрейм), являющееся родительским для текущего окна;
- `self` — возвращает ссылку на текущее окно;
- `top` — возвращает ссылку на главное окно;
- `name` — название окна (значение атрибута `name` окна или фрейма);
- `opener` — ссылка на окно, в котором было создано текущее окно;
- `closed` — логическое значение, указывающее, закрыто ли окно;
- `status` — текст, отображаемый в строке состояния браузера в текущий момент времени;
- `defaultStatus` — текст по умолчанию строки состояния браузера;
- `returnValue` — позволяет определить возвращенное значение для события или диалогового окна (поддерживается только Internet Explorer 4+);
- `document` — ссылка только для чтения на объект `document` документа, загруженного в данное окно;
- `history` — ссылка только для чтения на объект окна `history` *(см. разд. 4.1.5)*;
- `location` — ссылка только для чтения на объект окна `location` *(см. разд. 4.1.4)*;
- `navigator` — ссылка только для чтения на объект окна `navigator` *(см. разд. 4.1.6)*;
- `screen` — ссылка только для чтения на глобальный объект `screen`;
- `length` — количество фреймов в текущем окне.

Браузер Internet Explorer 4+ поддерживает дополнительные свойства, из которых мы отметим два наиболее важных:

- `returnValue` — позволяет определить значение, возвращаемое окном, созданным с помощью метода `showModalDialog()`;

- `event` — ссылка только для чтения на глобальный объект `event`, содержащий информацию о событии.

Свойство `parent` позволяет обратиться к объекту, находящемуся в иерархии на одну ступень выше, например, к окну, содержащему коллекцию фреймов, в которой находится наш фрейм. Для перемещения на две ступени вверх мы должны использовать `parent.parent` и т. д.:

```
parent.window.frames[0]
parent.parent.window.frames[0]
```

Чтобы обратиться к самому главному окну, т. е. к окну браузера, следует использовать свойство `top`. Однако `top` не может обращаться к главному фрейму вашей системы разбиения на фреймы.

Свойство `name` соответствует имени фрейма, которое мы задаем в теге `<FRAMESET>`.

Свойство `status` полезно для вывода сообщений во время работы сценария, например, при отладке:

```
window.status = "Сейчас работает сценарий"
```

Методы объекта *window*

Перечислим основные методы объекта `window`:

- `open()` — открывает новое окно (или закладку) браузера;
- `close()` — закрывает текущее окно браузера;
- `alert()` — показывает окно предупреждения с сообщением и кнопкой **OK**;
- `prompt()` — показывает окно приглашения с сообщением, текстовым полем и кнопками **OK** и **Отмена** (Cancel);
- `confirm()` — показывает окно подтверждения с сообщением и кнопками **OK** и **Отмена** (Cancel);
- `navigate()` — загружает в данное окно страницу с указанным адресом;
- `blur()` — убирает фокус с текущей страницы; соответствующее событие — `onblur`;
- `focus()` — устанавливает окно в фокус; соответствующее событие — `onfocus`;
- `scroll(x,y)` — сдвигает указанные в параметрах позиции `x` и `y` в верхний левый угол окна (не рекомендуется использовать);

- `scrollBy(x,y)` — смещает окно на указанные в параметрах количества пикселов по горизонтали и вертикали;
- `scrollTo(x,y)` — смещает окно в позицию, указанную в параметрах;
- `resizeBy(ширина,высота)` — изменяет размеры окна на указанные в параметрах количества пикселов (отрицательные значения соответствуют сжатию);
- `resizeTo(ширина,высота)` — устанавливает размеры окна в соответствии с указанными в параметрах значениями пикселов;
- `setInterval(`*процедура, период*`)` — запускает указанную процедуру (функцию) периодически через заданный интервал (в миллисекундах);
- `setTimeout()` — запускает программу через заданный интервал (в миллисекундах) после загрузки страницы;
- `clearInterval()` — обнуляет таймер, заданный методом `setInterval()`;
- `clearTimeout()` — обнуляет таймер, заданный методом `setTimeout()`;
- `print()` — открывает диалоговое окно печати.

Браузер Internet Explorer 4+ поддерживает дополнительные методы, из которых мы отметим четыре наиболее важных:

- `showHelp()` — показывает окно подсказки как диалоговое;
- `showModalDialog()` — показывает новое окно как диалоговое модальное (т. е. остающееся впереди других, пока не будет закрыто);
- `showModalessDialog()` — показывает новое окно как диалоговое немодальное;
- `execScript(`*выражение, язык*`)` — выполняет код сценария на указанном языке (по умолчанию JScript).

4.1.3. Объект *document*

Объект `document` (документ) — центральный в иерархической объектной модели, он представляет всю информацию об HTML-документе с помощью коллекций и свойств.

Свойства объекта *document*

Основные свойства объекта `document` приведены в табл. 4.1. Здесь указаны также атрибуты HTML-тегов, значения которых равны соответствующим свойствам документа.

Таблица 4.1. *Свойства объекта* `document`

Свойство	Атрибут	Назначение
`alinkColor`	`ALINK`	Цвет активных ссылок
`bgColor`	`BGCOLOR`	Цвет фона документа
`body`	–	Ссылка только для чтения на неявный основной объект документа, определенный в теге `<BODY>` или `<FRAMESET>`
`cookie`	–	Строка cookie-записи. Присвоение нового значение этому свойству приводит к записи cookie на диск после закрытия браузера
`doctype`	–	Ссылка на объект `DocumentType` документа
`documentElement`	–	Ссылка на объект корневого узла документа в иерархии объектов
`domain`	–	Строка, содержащая имя домена, откуда был получен документ
`fgColor`	`TEXT`	Устанавливает цвет текста
`lastModified`	–	Строка, содержащая дату последнего изменения документа
`linkColor`	`LINK`	Цвет еще не посещенных гиперссылок на странице
`readyState`	–	Определяет текущее состояние загружаемого объекта
`referer`	–	URL документа, который ссылается на текущий
`selection`	–	Ссылка только для чтения на дочерний для `document` объект `selection`, соответствующий выделенному в текущий момент тексту (поддерживается только Internet Explorer 4+)
`title`	–	Определяет название документа, отображаемое в строке заголовка окна браузера (содержимое тега `<TITLE>`)
`URL`	`URL`	URL-адрес документа клиента или в теге `<META>`; обычно это псевдоним для `location.href`
`vlinkColor`	`VLINK`	Цвет посещенных ссылок на странице

Коллекции объекта *document*

Перечислим основные коллекции объекта `document`:

- `all` — коллекция всех тегов и элементов в основной части документа (нестандартная коллекция, поддерживаемая Internet Explorer 4+, а также некоторыми другими браузерами, в частности, Mozilla Firefox 2.0 и Opera);
- `anchors` — "якорей" (закладок) в документе, т. е. элементов, задаваемых тегами вида `<A name=…>…</A>`;
- `applets` — объектов в документе, включая встроенные элементы управления, графические элементы, апплеты, внедренные и другие объекты;
- `embeds` — внедренных объектов в документе;
- `forms` — форм на странице;
- `frames` — фреймов, определенных в теге `<FRAMESET>`;
- `images` — графических элементов (изображений) на странице;
- `links` — ссылок и блоков `<AREA>` на странице;
- `styleSheets` — конкретных свойств стиля, определенных в документе.

Методы объекта *document*

Приведем список основных методов объекта `document`:

- `clear()` — очищает выделенный участок;
- `close()` — завершает выходной поток документа;
- `getElementByID()`— возвращает элемент со значением атрибута `id`, указанным в параметре, или `null`, если такого элемента не существует;
- `getElementByName()`— возвращает коллекцию элементов со значением атрибута `name`, указанным в параметре;
- `getElementByTagName()`— возвращает коллекцию элементов, определяемых именем тега, указанным в параметре; если значение параметра есть "*", то возвращаются имена всех тегов документа;
- `open()` — открывает документ как поток для обработки результатов применения методов `write()` и `writeln()`;
- `write()` — записывает текст и код HTML в документ;
- `writeln()` — записывает текст и код HTML, заканчивающийся возвратом каретки.

4.1.4. Объект *location*

Объект `location` (расположение) содержит информацию об URL-адресе (и его компонентах) текущей страницы, а также методы, позволяющие обновлять страницы.

Свойства объекта *location*

Далее приведен список свойств объекта `location`:

- `href` — полный URL-адрес в виде строки;
- `hash` — строка, следующая в URL за символом `#`, с помощью которого указывается, к какому якорю внутренней ссылки следует переместиться при загрузке документа;
- `host` — часть URL "*хост*:*порт*"; значение порта содержится лишь тогда, когда оно явно было указано в URL-адресе;
- `hostname` — имя хоста (домена);
- `pathname` — путь к объекту или файлу;
- `port` — номер порта URL;
- `protocol` — начальная часть, определяющая протокол, за которой следует двоеточие, например, `"http:"`;
- `search` — строка запроса или данные URL, следующие после имени файла (включая разделительный символ `?`).

Например, если вы загрузили страницу с адресом **http://www.cityline.spb.ru**, то значением `location.href` будет эта строка.

Присваивая свойству `href` новое значение, мы можем изменять показываемую в браузере страницу, например:

```
window.location.href="http://www.rambler.ru"
```

Методы объекта *location*

Перечислим методы объекта `location`:

- `assign()` — загружает другую страницу; этот метод эквивалентен изменению свойства `window.location.href`;
- `reload()` — обновляет текущую страницу;
- `replace()` — загружает страницу с указанным в параметре URL-адресом и заменяет URL-адрес текущей страницы (эквивалентен `location.href`).

Некоторые Web-сайты могут содержать страницы, которые не предназначены для посещения пользователем и занесения их в список посещенных страниц.

Например, перемещение по сайту может привести пользователя на некоторые промежуточные страницы, которые ему больше не понадобятся. При этом желательно, чтобы пользователь не мог вернуться к ним снова с помощью кнопки **Назад** (Back). В этом случае применяется метод `location.replace(`*`URL-адрес`*`)` для перехода к указанной странице без занесения его в список документов, переход к которым осуществляется щелчком на этой кнопке.

14.1.5. Объект *history*

Объект `history` (история) содержит информацию об адресах страниц, которые браузер отобразил во время текущего сеанса. Вы можете передвигаться по этому списку с помощью сценария и загружать соответствующие страницы. Объект `history` имеет только одно свойство и три метода.

Свойства объекта *history*

`length` — количество элементов в списке посещенных страниц.

Это свойство используется в сценарии для контроля выхода за пределы списка.

Методы объекта *history*

Методы таковы:

- `back()` — загружает предыдущую страницу из списка;
- `forward()` — загружает следующую страницу из списка;
- `go()` — загружает страницу с относительным номером `n` из списка (от 0 по `history.length` – 1) или с указанным URL-адресом.

В следующем примере загружается первая страница, а затем — пятая страница, причем предварительно проверяется, существует ли она в списке уже посещенных страниц:

```
window.history.go(1)
if (window.history.length>4)
   window.history.go(5)
```

Очевидно, метод `history.go(-1)` эквивалентен `history.back()`, а `window.history.go(1)` — методу `history.forward()`.

4.1.6. Объект *navigator*

Объект `navigator` (навигатор) содержит информацию о производителе браузера, его версии и возможностях.

Свойства объекта *navigator*

Далее перечислены свойства объекта `navigator`:

- `appCodeName` — название кода браузера; например, "Mozilla";
- `appName` — название браузера; например, "Microsoft Internet Explorer";
- `appVersion` — версия браузера; например, "4.0 (compatible; MSIE 6.0; Windows 98; Win 9*x* 4.90)";
- `cookieEnabled` — определяет возможность использования в браузере cookies со стороны клиента; является логическим значением;
- `userAgent` — название браузера, посылаемое им на сервер с помощью HTTP-протокола; например, "Mozilla/4.0 (compatible; MSIE 6.0; Windows 98; Win 9*x* 4.90)".

Коллекции объекта *navigator*

Перечислим коллекции объекта `navigator`:

- `mimeTypes` — коллекция всех типов документов и файлов, поддерживаемых браузером;
- `plugins` — всех внедряемых объектов на странице.

Методы объекта *navigator*

Методы объекта `navigator` таковы:

- `taintEnabled` — возвращает значение `false`; метод включен для совместимости с Netscape Navigator;
- `javaEnabled` — сообщает, возможен ли в данном браузере запуск кода сценария на языке JavaScript; является логическим значением.

Иногда требуется установить, каков браузер пользователя, чтобы выполнить переход к следующему HTML-документу. Например, мой сайт с учебными примерами (**http://dunaevv1.narod.ru**) изобилует элементами, которые правильно отображаются только в Internet Explorer. Чтобы у пользователя не возникало недоразумений, я проверяю, каким браузером он пользуется. В зависимости от результата этой проверки происходит переход на соответствующую страницу. Это можно сделать с помощью следующего сценария:

```
if (navigator.appName=="Microsoft Internet Explorer")
 location.replace("mypage1.htm")
else location.replace("mypage2.htm")
```

4.1.7. Объект *screen*

Объект `screen` (экран) содержит информацию о возможностях дисплея пользователя и может применяться, например, для определения размеров создаваемых окон и выбора разрешения передаваемой графики. Объект `screen` имеет следующие свойства:

- `availWidth` — возвращает ширину экрана пользователя в пикселах за вычетом элементов оформления окна (таких как панели инструментов Windows), создаваемых операционной системой;
- `availHeight` — возвращает высоту экрана пользователя в пикселах за вычетом элементов оформления окна, создаваемых операционной системой;
- `width` — возвращает ширину экрана пользователя (в пикселах);
- `height` — возвращает высоту экрана пользователя (в пикселах);
- `colorDepth` — возвращает информацию, позволяющую решить, как использовать цвета на экране; количество битов на пиксел устройства или видеобуфера пользователя;
- `updateInterval` — возвращает или задает интервал времени между обновлениями экрана пользователя.

4.2. Доступ к объектам браузера и документа

В *разд. 4.1.2 — 4.1.7* мы рассмотрели основные объекты браузера и документа. Синтаксис доступа к свойствам этих объектов таков:

```
Объект1.Объект2….ОбъектN.свойство.
```

Промежуточные объекты в этой последовательности упоминаются с учетом иерархии объектной модели (корневой объект `window` можно опускать). Например, объект `location` является объектом окна браузера, который представляется объектом `window`. Если мы хотим получить значение свойства `href` (URL-адрес загруженного документа) объекта `location`, то можем использовать выражение `window.location.href`, или (с учетом наивысшего расположения `window` в иерархии объектов) можно написать короче: `location.href`. Цвет текста документа задается свойством `fgColor`, которое является собственным свойством объекта `document`. Иначе говоря, для доступа к нему не нужно указывать промежуточные объекты, а достаточно написать так: `document.fgColor`.

Некоторые промежуточные объекты в последовательности `Объект1.Объект2….ОбъектN.свойство` являются коллекциями, т. е. массивами. Массивы, как нам уже известно, допускают обращения к их элементам по индексу — целочисленному, а во многих случаях и символьному.

Например, `document.images[0]` — объект 1-го графического изображения в документе; если соответствующий тег имеет атрибут, например, `mypict`, то обратиться к этому объекту можно и с помощью выражения `document.images["mypict"]`. В браузере Mozilla Firefox 2.0, например, `document.links["значение_id"]` возвращает значение URL-адреса ссылки, заданной тегом `<A>` с указанным значением атрибута `id`, а Internet Explorer возвращает `undefined`. Поэтому в Internet Explorer для обращения к элементам коллекции `links` требуется числовой индекс.

В листинге 4.1 в качестве примера приведен код, отображающий в окне интернет-обозревателя значения некоторых свойств браузера и документа. На рис. 4.4 показан результат работы данного сценария в Internet Explorer.

Листинг 4.1. Вывод свойств браузера и документа

```
<HTML>
<HEAD><TITLE>Свойства окна и документа</TITLE></HEAD>
<A name="Первая" href="http://dunaevv1.narod.ru">
Сам себе Web-дизайнер
</A><BR>
<A name="Вторая" href="http://www.yandex.ru">Яндекс</A><BR>
<A name="Третья" href="http://www.rambler.ru">Rambler</A>

<SCRIPT>
/* Объект window */
document.write("<h2 ID='zagl'>Некоторые свойства window</h2>");
document.write("<b>window.location: </b>", window.location, "<BR>");
document.write("<b>window.location.href: </b>", window.location.href,
"<BR>");
document.write("<b>window.navigator.appCodeName: </b>",
        navigator.appCodeName, "<BR>");
document.write("<b>window.navigator.appName: </b>",
           navigator.appName, "<BR>");
document.write("<b>window.navigator.userAgent: </b>",
           navigator.userAgent, "<BR>");
document.write("<b>window.screen.width: </b>",
           window.screen.width, "<BR>");
document.write("<b>window.frames.length: </b>",
           frames.length, "<BR>");
/* Объект document */
document.write("<h2>Некоторые свойства document</h2>");
document.write("<b>document.URL: </b>",document.URL, "<BR>" );
```

```
document.write("<b>document.lastModified: </b>",document.lastModified,
"<BR>");
document.write("<b>document.title: </b>",document.title, "<BR>");
document.write("<b>document.bgColor: </b>",document.bgColor, "<BR>");
document.write("<b>document.fgColor: </b>",document.fgColor, "<BR>");
document.write("<b>document.linkColor: </b>",document.linkColor,"<BR>");

if(document.links.length>0){    // ссылки
    document.write("<h3>Ссылки (" +
                    document.links.length + " шт.)</h3>");
    for(i=0;i<document.links.length;i++){
       document.write("<b>links[" + i + "]: </b>",
              document.links[i],"<BR>");
    }
}
</SCRIPT>
</HTML>
```

HTML-код документа в листинге 4.1 содержит три ссылки (теги `<A…>…</A>`). Этим элементам соответствует коллекция `links` объекта `document`. Как и всякий массив, данная коллекция имеет свойство `length` со значением, равным количеству элементов в коллекции, каждый из которых содержит в качестве значения адрес соответствующей ссылки (значение атрибута `href`). В данном примере мы обращаемся к элементам коллекции `links` по целочисленным индексам. Например, `document.links[0]` содержит адрес первой ссылки.

> ***Примечание***
>
> Коллекция `anchors` содержит адреса только тех ссылок, теги которых имеют непустые значения атрибутов `name` или (только для Internet Explorer) атрибутов `id`. Вообще говоря, коллекция `anchors` предназначена для хранения данных для внутренних ссылок (анкеров, якорей), задаваемых тегом `<A name=…>`, в котором атрибут `name` обязателен, а `href` — нет.

При загрузке HTML-документа в браузер почти для каждого его элемента инициируется объект, доступный из сценария. Эти объекты группируются в коллекции. Так, графические изображения попадают в коллекцию `images`, формы — `forms`, ссылки — `links`. Кроме того, все объекты принадлежат еще и коллекции `all`. Как уже упоминалось, коллекция `all` не входит в стандарт DOM W3C, но поддерживается всеми наиболее популярными браузерами. Доступ к элементу коллекции можно осуществить двумя способами:

- `document.коллекция[индекс]`
- `document.коллекция["значение_id"]`

Рис. 4.4. Результат работы сценария листинга 4.1 в браузере

Здесь `индекс` — номер элемента, соответствующий порядку его упоминания в HTML-коде (нумерация начинается с нуля); `значение_id` — значение атрибута `id` в теге, задающем элемент.

Обращение к объекту по числовому индексу достаточно удобно, если число элементов в коллекции невелико и постоянно. В большинстве же практических случаев надежнее доступ к объектам по их идентификаторам (значениям атрибута `id`). По идентификатору обращаться к объекту можно не только

посредством коллекций, но и с помощью метода документа `getElementById("значение_id")`.

Многие свойства объектов документа, доступные сценариям, совпадают по названию с именами атрибутов соответствующих HTML-тегов. Например, пусть в HTML-документе имеется тег для вставки графического изображения

`<IMG id="pict" src = "mypicture.jpg">`.

Тогда доступ к значению, указывающему на источник изображения, можно получить с помощью любого из следующих выражений:

- `document.images["pict"].src`
- `document.all["pict"].src`
- `document. getElementById ("pict")`

> ***Примечание***
>
> В Internet Explorer, в отличие от других браузеров, возможны и такие варианты синтаксиса обращения к объектам посредством коллекций:
>
> `document.коллекция("значение_id");`
>
> `document.коллекция.значение_id`.
>
> Ради межбраузерной совместимости не следует применять данный синтаксис.

К формам можно обращаться так же, как и к другим элементам документа. Вместе с тем, для них возможен и особый вариант:

```
document.имя_формы.elements.[индекс],
```

где `имя_формы` — значение атрибута `name` в теге `<FORM name=…>`.

> ***Примечание***
>
> Некоторые теги (например, `<FORM>` и `<INPUT>`) имеют атрибут `name`. Вместе с тем в подавляющем большинстве тегов можно использовать атрибут `id`. Эти атрибуты не всегда функционально эквивалентны. Так, для формы и ее элементов атрибут `name` играет особую роль. Чтобы избежать недоразумений, рекомендуется задавать одинаковые значения для `name` и `id` там, где требуется и то, и другое.

В листинге 4.2 в качестве примера приведен HTML-код, выводящий на Web-страницу ссылку, картинку и форму с двумя элементами, а также сценарий, который записывает в документ заголовки, список всех имен элементов документа (тегов) и некоторые свойства элементов формы и графического изображения. Вид соответствующей Web-страницы показан на рис. 4.5. В данном примере в учебных целях реализованы различные способы доступа к объектам. Так, для получения списка имен всех элементов документа

применена коллекция `all`. При этом собственно имя элемента извлекалось как значение свойства объекта `tagName`. К элементам формы мы обращались с помощью `document.myform.elements[i]`, где `myform` — значение атрибута `name` тега `<FORM>`, а `elements[i]` — объект *i*-го элемента формы. Например, `document.myform.elements[0].value` — содержимое поля ввода (первого элемента формы), а `document.myform.elements[1].value` — надпись на кнопке (первого элемента формы). Выражение `document.getElementById('mypict').src` возвращает URL-адрес источника графического изображения.

Листинг 4.2. Свойства HTML-элементов

```
<HTML>
<HEAD><TITLE>Свойства элементов документа</TITLE></HEAD>
<A id="myref" href="http://dunaevv1.narod.ru">
Сам себе Web-дизайнер</A>
<IMG id="mypict" src="picture1.jpg">
<FORM name="myform" id="myform">
 <INPUT id="mytext" type="text" value="Привет">
 <INPUT id="mybutton" type="button" value="Щелкни здесь">
</FORM>

<SCRIPT>
document.write("<h3 ID='zagl'>Элементы документа</h3>");
var l=document.all.length    // фиксируем длину коллекции all
/* Список HTML-элементов */
for(i=0;i<l;i++){
    document.write(i+". "+document.all[i].tagName,
   " id = "+document.all[i].id+"<BR>");
}
/* Свойства элементов формы */
document.write("<h3 ID='zagl'>Элементы формы
"+document.myform.name+"</h3>");
for(i=0; i<document.myform.elements.length;i++){
    document.write("id = "+document.myform.elements[i].id,
    ", type = "+document.myform.elements[i].type,
    ", value = "+document.myform.elements[i].value,"<BR>");
}
document.write("<h3>Графическое изображение</h3>");
/* Свойства графического изображения */
```

```
document.write("id = "+document.getElementById('mypict').id,
        ", src = "+document.getElementById('mypict').src);
</SCRIPT>
</HTML>
```

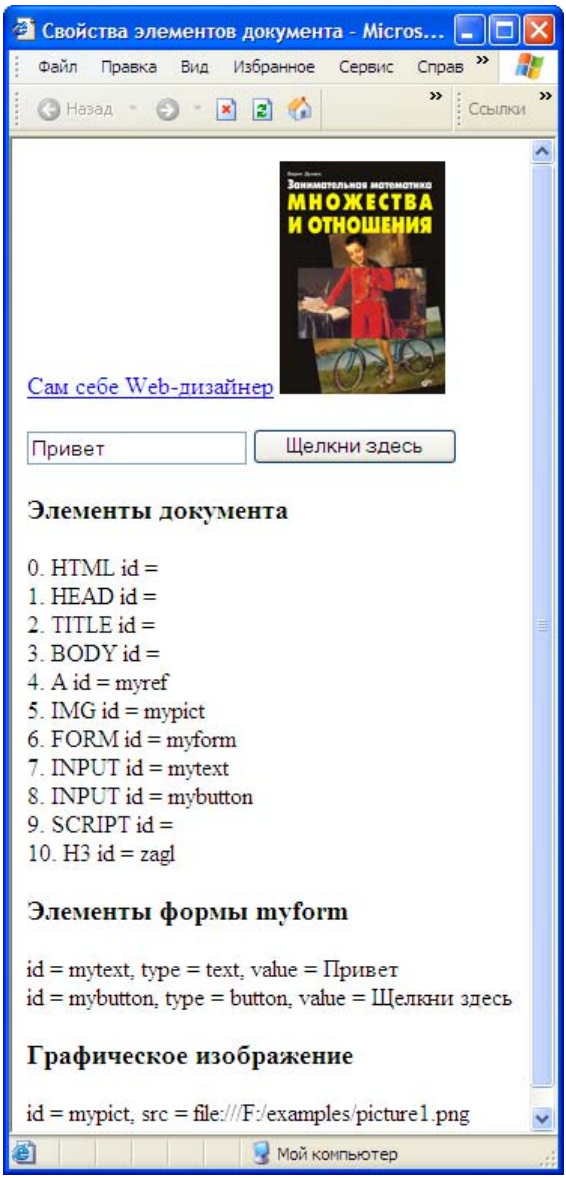

Рис. 4.5. Свойства HTML-элементов

Большинство свойств элементов документа можно изменять в сценарии с помощью оператора присваивания. Например, если требуется заменить одну картинку, вставленную в Web-страницу посредством тега `<IMG id="mypict" src="picture1.jpg">`, на другую, скажем, picture2.jpg, то достаточно указать такое выражение:

```
document.getElementById("mypict").src="picture2.jpg".
```

Некоторые свойства доступны только для чтения, т. е. их нельзя изменить в сценарии непосредственно. В листинге 4.3 приведен HTML-код, выводящий форму с текстовым полем и кнопку. Сценарий изменяет содержимое поля, надпись на кнопке и цвет фона документа. Однако мы не можем заменить поле ввода на кнопку, поскольку свойство `type`, определяющее тип элемента формы (задаваемого тегом `<INPUT>`), доступно только для чтения. Результат работы данного кода показан на рис. 4.6.

Листинг 4.3. Изменение свойств элементов документа

```
<HTML>
<HEAD><TITLE>Свойства элементов документа</TITLE></HEAD>
<FORM name="myform" id="myform">
 <INPUT id="mytext" type="text" value="Привет">
 <INPUT id="mybutton" type="button" value="Щелкни здесь">
</FORM>
<SCRIPT>
document.getElementById("mytext").value="До свидания"; // поле ввода
document.getElementById("mybutton").value="OK"; // надпись на кнопке
document.bgColor="#00ff00";     // устанавливаем зеленый фон
</SCRIPT>
</HTML>
```

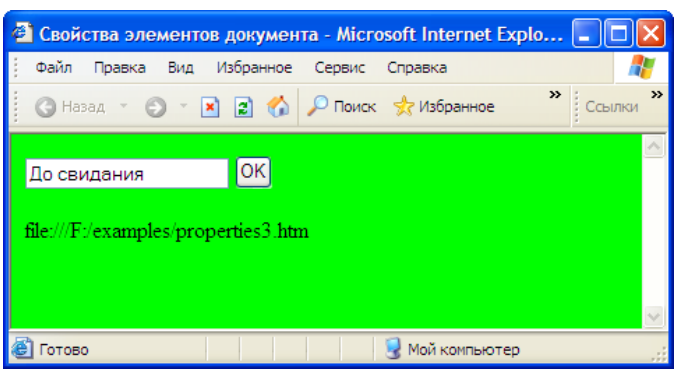

Рис. 4.6. Результат работы кода листинга 4.3

4.3. Обработка событий

Наиболее важным назначением сценариев для Web-сайтов является обработка событий — действий, совершенных пользователем с помощью мыши и/или клавиатуры, а также произошедших в браузере без непосредственного участия пользователя (например, окончание загрузки документа). Подробная информация о произошедшем событии сохраняется в глобальном объекте `event` (событие) до тех пор, пока не произошло следующее событие. Эта информация может быть использована, так или иначе, в сценарии, но она требуется далеко не во всех случаях практической разработки сайтов, а потому мы рассмотрим объект `event` после того, как разберемся с общей идеей и основными приемами обработки событий.

На начальном этапе изучения данной темы важно понять связь `событие – контекст – обработчик`. Событие возникает в результате каких-то действий пользователя посредством мыши и клавиатуры и/или по другим причинам. Любое событие происходит при каких-то обстоятельствах (т. е. в контексте) и может быть как-то обработано, чтобы браузер среагировал определенным образом.

4.3.1. Привязка обработчиков событий

Событие (event) связано с той или другой частью документа. Например, щелчок левой клавишей мыши может быть связан со ссылкой, кнопкой, графическим изображением и даже со всем документом. Другими словами, событие происходит в некотором контексте, в каком-то месте документа. Реакция браузера на событие определяется программой его обработки, которую обычно называют обработчиком (handler) события. Для некоторых событий предусмотрены встроенные обработчики (обработчики по умолчанию). Например, щелчок на ссылке вызывает обработчик по умолчанию, который загружает в браузер документ с указанным в ссылке адресом. Заметьте, что это происходит без какого-либо специально написанного на JavaScript сценария. В данном случае срабатывает встроенный обработчик, который обеспечивает реакцию браузера на щелчок мышью, когда ее указатель расположен на ссылке. Аналогично, щелчок на кнопке типа submit (т. е. на элементе `<INPUT type="submit">`) приводит к отправке данных, содержащихся в элементах формы. Однако для обработки, например, щелчка на обыкновенной кнопке, графическом изображении или обыкновенной текстовой строке требуется написать специальный (пользовательский) обработчик этого события. Специальными обработчиками можно также заменить обработчики по умолчанию. Например, вы можете создать свой обработчик щелчка на кнопке типа

submit, который перед отправкой проверяет данные, введенные пользователем в форму.

События, возникающие в результате действий пользователя, обычно имеют целевой объект — элемент HTML-документа, в пределах которого происходит событие. Например, щелчок (click) левой кнопкой мыши может происходить, когда указатель мыши находится на элементах документа, заданных тегами <BUTTON>, <IMG>, <P> и др. Мы можем создать различные обработчики одного и того же события для разных элементов. Для этого необходимо привязать обработчики к элементам.

Существуют несколько способов привязки обработчиков событий к элементам. Вот наиболее употребительные из них:

- Применение атрибутов-событий в HTML-тегах по следующей схеме:

  <Тег … атрибут-событие = "код_обработчика">

  Имена атрибутов-событий образуются из английского названия события добавлением префикса "on". Например, onclick (click — щелчок, onclick — при щелчке), onmouseover (при наведении указателя мыши), onsubmit (при отправке данных формы на сервер) и т. д.

  Программный код обработчика представляет собой одно или несколько выражений, разделенных точкой с запятой. В случае большого объема кода его обычно оформляют в виде функции. Например, обработчик щелчка на графическом изображении можно привязать к объекту изображения следующим образом:

  ```
  <IMG id = "myimg" src = "picture.jpg" onclick = "myfunc()">
  ```

- Установка обработчика в сценарии по схеме:

  document.getElementById(*"значение_id"*).*атрибут-событие* = *имя_функции*

  Например, `document.getElementById("myimg").onclick = myfunc`.

  Здесь атрибут-событие обязательно должен быть записан строчными буквами (в нижнем регистре), чего не требуется в случае его записи внутри HTML-тега. Имя функции в операторе присваивания в данном случае упоминается без пары круглых скобок.

К одному и тому же элементу можно привязать несколько обработчиков различных событий как в следующем примере:

```
<P onclick = "alert('Щелчок!')" onmouseover = alert('Наведение')">
Этот текст реагирует на щелчок и наведение указателя мыши
</P>
```

Здесь фрагмент текста, определенный в контейнере абзаца <P>, реагирует и на щелчок кнопкой мыши, и на наведение ее указателя.

В табл. 4.2 приведен список основных (далеко не всех) событий и HTML-элементов, к которым они могут быть привязаны.

***Таблица 4.2.** Основные события*

Атрибут-событие	Описание события	HTML-элементы
onblur	Потеря элементом фокуса ввода, когда пользователь активизирует другой элемент	<a>, <area>, <button>, <input>, <label>, <select>, <textarea>. В IE4+ также <applet>, , <embed>, <hr>, <img>, <marquee>, <object>, , <table>, <td>, <tr>; В IE4+ также <body>
onchange	Потеря фокуса ввода полем формы, когда его значение было изменено	<input>, <select>, <textarea>
onclick	Щелчок на элементе	Большинство видимых элементов; В IE4+ также <applet>, <font>
ondblclick	Двойной щелчок на элементе	То же
onfocus	Получение фокуса ввода элементом	<a>, <area>, <button>, <input>, <label>, <select>, <textarea>; В IE4+ также <applet>, , <embed>, <hr>, <img>, <marquee>, <object>, , <table>, <td>, <tr>; В IE4+ также <body>
onkeydown	Нажатие на клавишу, когда элемент имеет фокус ввода	Большинство видимых элементов; В IE4+ также <applet>, <font>
onkeypress	Нажатие на клавишу и освобождение ее, когда элемент имеет фокус ввода	То же
onkeyup	Освобождение ранее нажатой клавиши, когда элемент имеет фокус ввода	То же
onload	Окончание загрузки объекта (обычно окна или фрейма) в браузер	<body>, <frameset>; В IE4+ также <img>, <applet>, <embed>, <link>, <style>, <script>

Таблица 4.2 *(окончание)*

Атрибут-событие	Описание события	HTML-элементы
onmousedown	Нажатие на кнопку мыши, когда элемент имеет фокус ввода	Большинство видимых элементов; В IE4+ также <applet>, <font>
onmousemove	Перемещение указателя мыши над элементом	То же
onmouseout	Смещение указателя мыши за пределы элемента	То же
onmouseover	Перемещение указателя мыши на элемент	То же
onmouseup	Освобождение ранее нажатой кнопки мыши, когда элемент имеет фокус ввода	То же
onreset	Возвращение формы в исходное состояние (например, нажатием на кнопку типа reset)	<form>
onsubmit	Отправка данных формы на сервер (например, нажатием на кнопку типа submit)	<form>
onselect	Выбор пользователем текста (обычно выделением его с помощью мыши)	<input>, <textarea>
onunload	Выгрузка текущего документа из окна или фрейма	<body>, <frameset>

Кроме событий, перечисленных в табл. 4.1, браузеры поддерживают различные наборы нестандартных событий. Так например, Internet Explorer 5+ поддерживает, среди прочих, еще и такие очень полезные события:

- oncopy — попытка копирования выделенного содержимого объекта из документа в буфер обмена;
- oncut — попытка вырезания выделенного содержимого объекта из документа в буфер обмена;
- onpaste — попытка вставки содержимого буфера обмена в документ;
- ondrag — существует во время перетаскивания объекта мышью;

- ondragend — возникает при завершении перетаскивания объекта путем отпускания кнопки мыши;
- ondragstart — возникает при начале перетаскивания выделенной области.

4.3.2. Программный вызов обработчика события

Обработчики событий, создаваемые пользователем, можно вызвать программно, а не только манипулируя мышью и клавиатурой. Иначе говоря, можно выполнить программный код обработки события, которое не произошло из-за действий пользователя. Вызов обработчика того или иного события обеспечивает метод, название которого совпадает с названием атрибута-события. Например, для вызова обработчика щелчка используется метод `onclick()`.

В следующем примере (листинг 4.4) обработчик события onclick срабатывает сразу же при загрузке документа, даже если пользователь не щелкал мышью на кнопке.

Листинг 4.4. Программный вызов обработчика события

```
<HTML>
<HEAD><TITLE>Обработка событий</TITLE></HEAD>
<BUTTON id="mybtn" onclick="alert('Щелчок!')" >
Щелкни здесь
</BUTTON>
<SCRIPT>
document.getElementById("mybtn").onclick() // вызов обработчика
</SCRIPT>
</HTML>
```

При загрузке данного кода в браузер сразу же будет выведено диалоговое окно с сообщением "Щелчок!", хотя пользователь еще не успел щелкнуть на кнопке.

Для некоторых элементов обработчик можно вызвать программно не напрямую, как в предыдущем примере, а путем генерации соответствующих событий. События генерируют методы, названия которых подобны названиям атрибутов-событий, но не имеют префикса on. Например, генерацию щелчка обеспечивает метод `click()`. Программно сгенерированное событие влечет за собой вызов соответствующего обработчика, если, конечно, он определен.

В следующем примере (листинг 4.5) в сценарии генерируется событие, соответствующее щелчку на кнопке формы.

Листинг 4.5. Программная генерация события

```
<HTML>
<FORM name="myform" id="myform">
 <INPUT id="mytext" type = "text" value="Привет">
 <INPUT id="mybutton" type = "button" value = "Щелкни здесь"
     onclick = "alert('Щелчок')">
</FORM>
<SCRIPT>
document.getElementById("mybutton").click() // генерация события
/* Эквивалентное выражение:
document.myform.mybutton.click()
*/
</SCRIPT>
</HTML>
```

В некоторых случаях событие удобнее сгенерировать, а не просто непосредственно вызвать его обработчик. Так например, для ввода данных в текстовое поле (элемент `<INPUT>`) требуется сначала установить на нем фокус ввода или, другими словами, установить в нем курсор. После этого пользователь может вводить данные посредством клавиатуры. При загрузке документа с текстовыми полями курсор (фокус ввода) не устанавливается ни в одном из них, а потому пользователь должен щелчком мыши или клавишей табуляции активизировать нужное поле ввода прежде, чем вводить собственно данные. В большинстве случаев такое положение дел считается весьма неудобным: хорошо бы сразу активизировать какое-нибудь поле ввода. Это можно сделать программно, сгенерировав событие установки фокуса ввода на нужном элементе с помощью метода `focus()`. В следующем примере (листинг 4.6) страница содержит три текстовых поля и по окончании ее загрузки в браузер курсор сразу же устанавливается во втором поле методом `focus()`.

Листинг 4.6. Установка фокуса ввода

```
<HTML>
<INPUT type="text" id="inp1">
<INPUT type="text" id="inp2">
<INPUT type="text" id="inp3">
<SCRIPT>
 document.getElementById("inp2").focus(); // фокус на второе поле
</SCRIPT>
</HTML>
```

Здесь мы обращаемся к объекту `document.getElementById("inp2")`, соответствующему второму полю ввода, и применяем к нему метод `focus()` генерации события установки фокуса. Результат заключается в том, что во втором поле появляется курсор и, следовательно, оно готово к вводу данных с клавиатуры.

Как уже отмечалось, не все события можно сгенерировать программно. В табл. 4.3 приведены сведения о событиях и элементах документа, для которых это возможно.

Таблица 4.3. *События, которые можно сгенерировать программно*

Методы генерации событий	HTML-элементы
`click()`	`<input>` со значениями атрибута `type`: `"button"`, `"checkbox"`, `"radio"`, `"reset"`, `"submit"`; `<a>`
`blur()`	`<a>`, `<input>`, `<select>`, `<textarea>`
`focus()`	`<a>`, `<input>`, `<select>`, `<textarea>`
`select()`	`<input>` со значениями атрибута `type`: `"text"`, `"password"`, `"file"`; `<textarea>`
`reset()`	`<form>`
`submit()`	`<form>`

4.3.3. Изменение поведения элементов по умолчанию

При наступлении событий что-то может происходить, даже если для них программист явно не задал обработчик. В таких случаях говорят о поведении элементов по умолчанию. Например, при щелчке на ссылке по умолчанию осуществляется переход на указанную страницу. При щелчке на кнопке типа submit по умолчанию данные формы передаются на сервер. Щелчок на кнопке типа reset по умолчанию приводит к очистке всех данных формы.

Поведение по умолчанию можно изменить по своему усмотрению. Например, может потребоваться, чтобы при щелчке на ссылке выполнилась некоторая функция, а не попытка загрузки какого-то документа, а при щелчке на кнопке submit — предварительная проверка наличия в форме каких-либо данных или подтверждения намерения передать их. Для изменения поведения

элементов по умолчанию достаточно сделать так, чтобы обработчик события вернул значение `false`. Вот два типичных примера:

- Щелчок на ссылке должен приводить к вызову некоторой функции, а не загружать в браузер документ:

```
<A href="" onclick="myfunc(); return false">Щелкни здесь</A>
```

  Здесь в коде обработчика события onclick последнее выражение явным образом возвращает значение `false` и, следовательно, переход по URL-адресу ссылки (даже если он не пуст) не сработает, а будет вызвана функция `myfunc()`, которая что-то делает по усмотрению программиста. Впрочем, выражение `return false`, определяющее возвращаемое значение, можно вставить и в тело функции `myfunc()`, чтобы изъять его из общего кода обработчика.

- Щелчок на кнопке типа submit в форме требует подтверждения о необходимости отправить данные, введенные пользователем:

```
<INPUT type="submit"
onclick="return confirm('Вы действительно хотите передать данные на
сервер?')" value="Передать данные">
```

  Здесь метод `confirm()` объекта `window` возвращает `true` или `false` в зависимости от того, на какой кнопке выведенного диалогового окна щелкнул пользователь. Отправка данных произойдет, если метод `confirm()` вернет `true`, т. е. если пользователь щелкнет на кнопке **OK** (и если к тому же указан адрес и метод передачи данных в соответствующих атрибутах тега `<FORM>`, задающего форму); в противном случае браузер не будет отправлять данные.

В браузере Intenet Explorer 4+ для описанной цели можно использовать свойство `returnValue` объекта `event`, содержащего информацию о произошедшем событии. Так, для предотвращения поведения элементов по умолчанию следует в обработчике события написать `event.returnValue=false`.

4.3.4. Прохождение событий

Элементы HTML, как известно, образуют иерархическую структуру вложенности. Так например, текстовые поля и кнопки могут находиться в форме, а форма — в теле документа, т. е. в элементе `<BODY>`. Щелкая мышью на кнопке внутри формы, мы одновременно "щелкаем" и на самой форме, и на документе в целом. Иначе говоря, где-то возникшее событие распространяется далее по объектам. В различных браузерах модели распространения событий были реализованы по-разному. Например, в Netscape 4 анализ события начинается на вершине иерархии (т. е. на объекте самого высокого уровня)

и "спускается вниз" к целевому объекту. В Internet Explorer распространение события начинается от целевого объекта вверх по иерархии. Говорят, что событие "всплывает" подобно пузырьку с воздухом в воде. В модели событий DOM 2 предусмотрен как "спуск", так и "всплывание" событий. В большинстве практических задач представлений о всплывании событий оказывается вполне достаточно, тем более, что такая модель поддерживается всеми современными браузерами.

В листинге 4.7 в качестве примера приведен HTML-код документа, в котором кнопка (`<BUTTON>`) находится внутри контейнера, заданного тегом `<DIV>`. Контейнер `<DIV>` в свою очередь расположен в контейнере `<BODY>`. Внешний вид данного документа показан на рис. 4.7. Все перечисленные элементы документа реагируют на щелчок мышью, выводя диалоговое окно с сообщением о том, какой элемент принял данное событие. Если щелкнуть на кнопке (самом нижнем в иерархии элементе), то на щелчок последовательно отреагируют все элементы: сначала кнопка, затем `<DIV>` и, наконец, `<BODY>`.

Листинг 4.7. Пример для иллюстрации прохождения событий по иерархии объектов

```
<HTML>
<HEAD><TITLE>Обработка событий</TITLE></HEAD>
<BODY onclick="alert('Щелчок на body')">
Щелкните где-нибудь
   <DIV onclick="alert('Щелчок на div ')"
          style="background-color:#00ffff; width:300; height:200">
          Элемент DIV
          <BUTTON onclick="alert('Щелчок на button ')"
          style="position:absolute; top:100; left:50">Кнопка
          </BUTTON>
   </DIV>
</BODY>
</HTML>
```

Распространение событий по иерархии объектов можно отменить. Для этой цели используется свойство `cancelBubble` объекта `event`, содержащего информацию о произошедшем событии. Для прекращения дальнейшего всплывания события достаточно в его обработчике написать `event.cancelBubble=true`.

Код листинга 4.8 отличается от рассмотренного ранее примера только тем, что на щелчок на кнопке реагирует только сама кнопка: далее данное событие не распространяется.

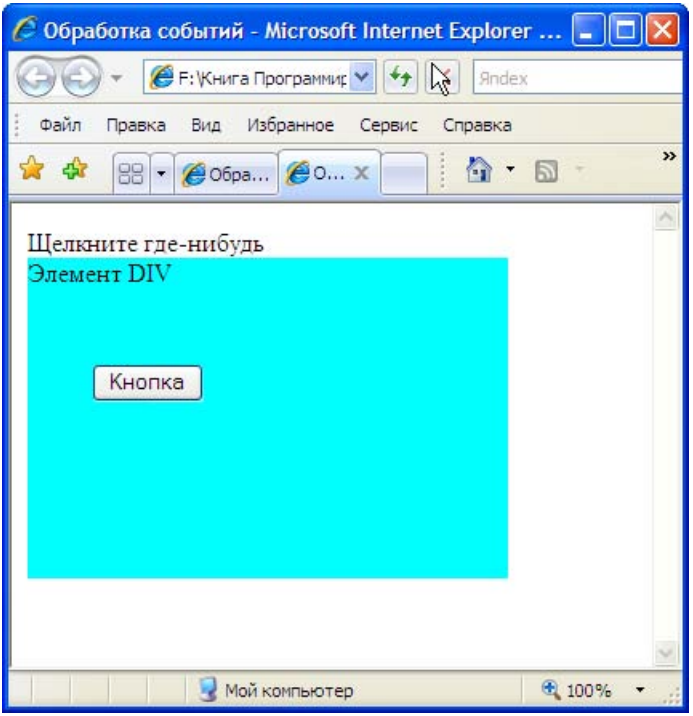

Рис. 4.7. Вид документа листинга 4.7

Листинг 4.8. Пример прекращения прохождения события по иерархии объектов

```
<HTML>
<HEAD><TITLE>Обработка событий</TITLE></HEAD>
<BODY onclick="alert('Щелчок на body')">
Щелкните где-нибудь
       <DIV onclick="alert('Щелчок на div ')"
            style="background-color:#00ffff; width:300; height:200">
            Элемент DIV
            <BUTTON
            onclick="alert('Щелчок на button '); event.cancelBubble=true"
            style="position:absolute; top:100; left:50">Кнопка
          </BUTTON>
       </DIV>
</BODY>
</HTML>
```

4.3.5. Информация о событии: объект *event*

Объект `event` (событие) предназначен для сохранения информации о произошедшем событии. Его экземпляр существует, пока событие не будет обработано. Все экземпляры инициированных событий заносятся в буфер и обрабатываются в том порядке, в каком они туда попали. Объект `event` глобальный, и к его свойствам обращаются так: `event.`*`свойство`*.

Например, при щелчке кнопкой мыши в экземпляре объекта события сохраняются координаты указателя мыши, сведения о том, была ли нажата клавиша <Alt> или <Ctrl>, в каком элементе произошло событие и т. д. Однако не для всех событий в экземпляре объекта `event` определяются все свойства. Так, экземпляры `event`, соответствующие событиям клавиатуры, не имеют свойств с информацией о положении указателя мыши.

В табл. 4.4 приведены основные свойства объекта `event`, которые доступны во всех наиболее популярных браузерах.

Таблица 4.4. *Основные свойства объекта event*

Свойства	Описание
`altKey`	Логическое значение, указывающее, была ли во время события нажата клавиша <Alt>
`ctrlKey`	Логическое значение, указывающее, была ли во время события нажата клавиша <Ctrl>
`shiftKey`	Логическое значение, указывающее, была ли во время события нажата клавиша <Shift>
`keyCode`	Целое число, представляющее значение Unicode соответствующей клавиши (В Netscape 6 это код ASCII)
`cancelBubble`	Логическое значение, указывающее, должно ли событие прекратить дальнейшее "всплывание" в иерархии объектов после его обработки текущим обработчиком
`clientX, clienY`	Координаты соответственно *x* и *y* указателя мыши в пикселах относительно клиентской области окна браузера
`screenX, screenY`	Координаты позиции в пикселах относительно всего экрана, в которой произошло событие
`srcElement`	Ссылка на объект элемента, являющегося целевым (IE4+). Для других браузеров следует использовать свойство `target`
`target`	Ссылка на объект элемента, в котором произошло событие (Netscape 4+, Mozilla, DOM 2). Для IE следует использовать свойство `srcElement`
`type`	Строка, указывающая тип события (например, "click")

Чтобы воспользоваться информацией о событии, необходимо передать экземпляр объекта `event` обработчику. Если обработчик определен в строке значения атрибута-события в HTML-теге, то в этой же строке можно обращаться к объекту события через его имя `event`:

```
<INPUT type="button" onclick="alert('Щелчок в точке '+event.clientX)">.
```

В других случаях обработчик следует оформлять в виде функции и передавать ей экземпляр объекта `event` в качестве параметра. В листинге 4.9 приведен пример, в котором функция-обработчик `myhandler(event)` щелчка привязана к элементу `<BODY>`. Обработчику передается в качестве параметра `event` экземпляр объекта события. Внешний вид документа такой же, как на рис. 4.7. Обработчик определяет, на каком из трех элементов документа (`<BODY>`, `<DIV>` или `<BUTTON>`) был произведен щелчок, и выводит диалоговое окно с сообщением о типе события, значении атрибута `ID` элемента, принявшего щелчок, и координатах щелчка относительно клиентской области браузера. Для определения элемента, в котором произошло событие, в Internet Explorer предусмотрено свойство `srcElement`, а в других браузерах — `target`. С целью обеспечения межбраузерной совместимости обработчик проверяет, какое из данных свойств определено и далее использует его для доступа к свойству `id` (значению атрибута `ID` элемента).

Листинг 4.9. Пример передачи объекта события обработчику в качестве параметра

```
<HTML>
<HEAD><TITLE>Обработка событий</TITLE></HEAD>
<BODY id="mybody" onclick="myhandler(event)">
Щелкните где-нибудь
<DIV id="mydiv" style="background-color:#00ffff;width:300;height:200">
Элемент DIV
<BUTTON id="mybutton" style="position:absolute; top:100; left:50"> Кнопка</BUTTON>
</DIV>
</BODY>
<SCRIPT>
function myhandler(evt){     // обработчик
// evt — экземпляр объекта event
var e = evt.srcElement;      // для IE
if (!e) e = evt.target;      // для других браузеров
alert(evt.type + " на id=" + e.id +
```

```
"\nclientX=" + evt.clientX +
"\nclientY=" + evt.clientY);
}
</SCRIPT>
</HTML>
```

> ***Примечание***
>
> В объектной модели Internet Explorer объект `window` имеет свойство `event`, являющееся ссылкой на объект `event`, а потому он доступен в обработчике событий посредством выражения window.`event.`*`свойство`*. В других браузерах это не так.

4.4. Окна и фреймы

Для работы с элементами управления браузера предназначен объект `window`, представляющий окно интернет-обозревателя. Фреймы, если они задаются, также являются окнами, но расположенными внутри клиентской области главного (исходного) окна браузера. С точки зрения программирования они почти ничем не отличаются от главного окна. Иногда требуется открыть отдельное окно, чтобы загрузить в него какой-либо документ, или изменить содержимое уже существующего фрейма.

4.4.1. Создание новых окон браузера

Средствами HTML открыть документ в новом окне можно с помощью атрибута `target` тега ссылки `<A HREF = . . .>`, например,

```
<A HREF = "http://www.yandex.ru" target = "newWindow"> Яндекс</A>.
```

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

```
window.open(параметры)
```

Этому методу передаются следующие необязательные параметры:

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

В строке свойств (features) записываются пары `свойство=значение`, которые отделяются друг от друга запятыми. В табл. 4.5 приведен список свойств окна, передаваемых в строке `features`. Значения `yes` и `no` можно заменить числовыми эквивалентами `1` и `0` соответственно.

Таблица 4.5. *Свойства окна, передаваемые в строке features*

Свойство	Значения	Описание
channelmode	yes, no, 1, 0	Показывает элементы управления Channel
directories	yes, no, 1, 0	Включает кнопки каталога
fullscreen	yes, no, 1, 0	Полностью разворачивает окно
height	Число	Высота окна в пикселах
left	Число	Положение по горизонтали относительно левого края экрана в пикселах
location	yes, no, 1, 0	Текстовое поле Address
menubar	yes, no, 1, 0	Стандартные меню браузера
resizeable	yes, no, 1, 0	Может ли пользователь изменять размер окна
scrollbars	yes, no, 1, 0	Горизонтальная и вертикальная полосы прокрутки
status	yes, no, 1, 0	Стандартная строка состояния
toolbar	yes, no, 1, 0	Включает панели инструментов браузера
top	Число	Положение по вертикали относительно верхнего края экрана в пикселах
width	Число	Ширина окна в пикселах

Вместо третьего параметра (`features`) можно подставить значение `true`. В этом случае указанный документ загружается в уже существующее окно, вытесняя предыдущий:

```
window.open("mypage.htm","NewWin", true).
```

Метод `window.open()` возвращает ссылку на объект открываемого окна, которую можно сохранить в переменной, чтобы потом использовать, например, при закрытии окна.

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

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

В листинге 4.10 в качестве примера приведен код, создающий Web-страницу со ссылкой на сайт Rambler и двумя кнопками. Щелчок на первой из них от-

крывает новое окно браузера и загружает в него главную страницу поисковой системы Яндекс, а нажатие второй кнопки закрывает это окно.

Листинг 4.10. Пример создания нового окна браузера

```
<HTML>
<HEAD><TITLE>Создание нового окна</TITLE></HEAD>
<A HREF = "http://www.rambler.ru" TARGET = "newWindow"> Rumbler</A><BR>
<BUTTON onclick = "openMyWindow()">Открыть новое окно</BUTTON>
<BUTTON onclick = "closeMyWindow()">Закрыть новое окно</BUTTON>
<SCRIPT>
var newWindow     // глобальная переменная для ссылки на окно
function openMyWindow(){    // открытие окна
window.status = "Первое окно"; // статусная строка главного окна
strfeatures = "top=100,left=50,width=500,height=350,toolbar=no";
strfeatures+=",scrollbars=yes";
newWindow=window.open("http://www.yandex.ru","win1",strfeatures)
}

function closeMyWindow(){     // закрытие окна
if (newWindow){           // если окно открыто
 newWindow.close()         // закрыть окно
 newWindow = null         // очистить ссылку на окно
 window.status = ""
}
}
</SCRIPT>
</HTML>
```

4.4.2. Работа с фреймами

Фрейм представляет собой прямоугольную область окна браузера, в которую можно загрузить HTML-документ. Разбиение окна на фреймы происходит с помощью HTML-кода, содержащегося в отдельном HTML-файле, который называется *установочным*. Установочный файл содержит только теги разметки фреймов `<FRAMESET>` и `<FRAME>`, с помощью которых задаются относительное расположение, размеры, ссылки на загружаемые в них документы и другие параметры фреймов. В установочном файле можно написать сценарий, создающий фреймовую структуру документа.

При разбиении окна на два фрейма в объектной модели браузера возникает иерархия объектов, в которой главное окно браузера представляется в виде родительского (главного) фрейма, а созданные два фрейма являются дочерними фреймами или, иначе говоря, фреймами-потомками. При разбиении любого из этих фреймов на следующие два фрейма, последние являются дочерними для исходного. Итак, между фреймами возникает отношение "родители — потомки" ("родители — дочки").

При запуске браузера объект главного окна `window` формируется автоматически. При загрузке документа с фреймовой структурой в главное окно оно получает статус родительского фрейма по отношению ко всем остальным. С другой стороны, каждый тег `<FRAME>` внутри контейнера `<FRAMESET>` создает свой собственный объект `window`, в который загружается соответствующий документ. Каждому из фреймов сопоставлен свой объект `document`. С точки зрения объекта `document` ему соответствует единственный контейнер — фрейм. Хотя родительский объект и не виден пользователю, он все равно присутствует в объектной модели.

Итак, в вершине иерархии находится окно браузера `window`. Оно поделено на два фрейма с именами, например, `frame1` и `frame2`. Объект `window` родительский по отношению к дочерним `frame1` и `frame2`. Для ссылки на родительское окно служит ключевое слово `parent`.

Допустим, пользователь активизирует некоторую ссылку в первом фрейме, но соответствующий документ должен загрузиться не в этот же фрейм, а в другой. При решении данной задачи необходимо рассмотреть три случая:

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

Между объектом `window` и объектами `frame1` и `frame2` существует прямая связь "родитель — потомок". Поэтому если вы пишете сценарий в установочном файле, создающем эти фреймы, то к фреймам можно обращаться по имени (значению атрибута `name` тега `<FRAME>`).

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

в `location.href` родительского окна новый URL-адрес. Таким образом, чтобы загрузить новый документ в родительское окно, нужно записать в сценарии следующее:

```
parent.location.href="URL-адрес_документа".
```

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

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

Рассмотрим пример создания документа с навигационной панелью. Окно браузера разделено на два фрейма: первый — это навигационная панель, а второй — окно для отображения документов (рис. 4.8). Фреймовая структура такого документа создается с помощью установочного файла, представленного в листинге 4.11.

Листинг 4.11. Установочный файл frames.htm

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

Здесь `start.htm` — документ, который будет первоначально показан во фрейме `main`, а `menu.htm` — навигационная панель (листинг 4.12), содержащая набор ссылок на другие документы.

Листинг 4.12. Документ menu.htm (навигационная панель)

```
<HTML>
 <SCRIPT >
  function load(url){ // загрузка документа во фрейм main
   parent.main.location.href=url;
```

```
  }
 </SCRIPT>
 <BODY>
  <A HREF="javascript:load('первый.htm')">Первый</A><BR>
  <A HREF="второй.htm" TARGET="main">Второй</A><BR>
  <A HREF="третий.htm" TARGET="top">Третий </A>
 </BODY>
</HTML>
```

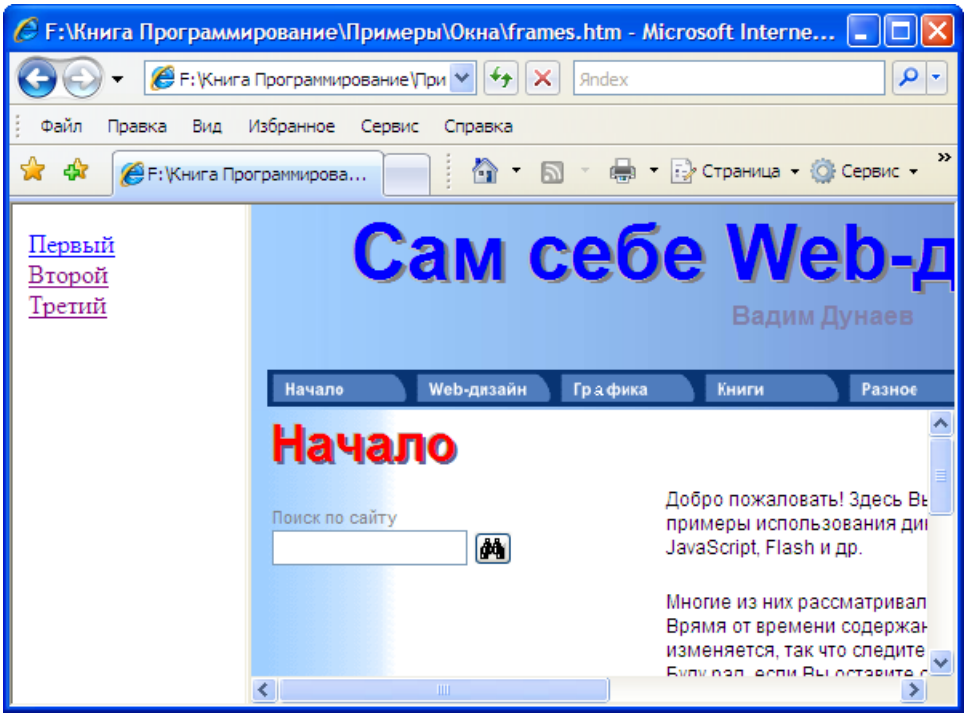

Рис. 4.8. Пример двухфреймовой структуры документа

В данном примере показано несколько способов загрузки новой страницы во фрейм `main`. В первой ссылке используется функция `load()`, которая загружает файл `первый.htm` во второй фрейм. Во второй ссылке используется атрибут `TARGET`. Третья ссылка показывает, как можно избавиться от фреймов. Чтобы удалить фреймы с помощью функции `load()`, достаточно написать в ней следующую строку:

```
parent.location.href=url.
```

Атрибут TARGET в теге ссылки <A HREF=...> обычно применяется в тех случаях, когда требуется загрузить одну страницу в один фрейм. Язык сценариев предпочтительнее, если при активизации ссылки нужно выполнить несколько действий, например, загрузить несколько страниц в разные фреймы.

Для ссылок из родительского окна к объектам его дочерних фреймов можно воспользоваться коллекцией frames всех фреймов, которая представляет собой массив объектов фреймов. Обратиться к конкретному фрейму из этой коллекции можно по индексу или по имени фрейма, указанному в качестве значения атрибута NAME в теге <FRAME>:

```
window.frames[индекс]
window.имя_фрейма
```

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

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

```
window.frames[0].document.getElementbyId("Myinput").value
window.LEFT.document. getElementbyId("Myinput").value
```

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

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

С помощью ссылок top и self можно предотвратить отображение вашего сайта внутри фреймов другого сайта. Для этого необходимо, чтобы документ верхнего уровня проверял, в какое окно он загружен. Это должно быть самое верхнее (top) или родительское (parent) окно. Если это действительно так, ссылка на свойство top совпадает со ссылкой self на текущее окно. При несовпадении этих значений документ следует перезагрузить заново, но

уже в окно верхнего уровня. Сценарий, выполняющий эту работу, необходимо разместить в начале документа. Вот его код:

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

При посещении вашего многофреймового сайта, пользователь может сделать закладку (поместить в папку Избранное) только на один фрейм, в то время как все средства навигации по сайту находятся в другом фрейме. При повторном посещении в браузере окажется усеченный вариант вашего сайта, что обычно нежелательно. Следующий простой сценарий проверяет, загружается ли Web-страница в своем наборе фреймов, сравнивая URL-адреса окна `top` и текущего окна. Если они совпадают, значит, загружен лишь один фрейм, и необходимо загрузить весь набор фреймов, определенный в установочном файле, например, frameset.htm:

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

Однако при выполнении этого сценария в окне браузера отобразятся документы, заданные в установочном файле, и, возможно, среди них не окажется запрошенного пользователем. Поэтому для каждого HTML-документа, отображаемого во фрейме, следует предусмотреть отдельный установочный файл, в котором задана загрузка этого документа.

Допустим, документы, предназначенные для отображения во фрейме, хранятся в файлах file1.htm, file2.htm, ..., file*N*.htm. Тогда следует создать установочные файлы frameset1.htm, frameset2.htm, ..., frameset*N*.htm, в которых задана загрузка во фрейм соответствующего документа. Например, установочный файл frameset1.htm для двухфремовой структуры может иметь следующий вид:

```
<FRAMESET rows = "128,*" border = 0>
 <FRAME SRC = "navpane.htm" NAME = "FRAME1" scrolling = no>
 <FRAME SRC = "file1.htm"  NAME = "FRAME2" scrolling = auto>
</FRAMESET>
```

Здесь HTML-документ из файла file1.htm предполагается загружать во второй фрейм. При этом файл file1.htm где-то в самом начале должен содержать такой сценарий:

```
<SCRIPT>
 if (top.location.href == window.location.href)
  top.location.href = "frameset1.htm"
</SCRIPT>
```

Аналогично формируются остальные установочные и информационные файлы сайта.

4.4.2. Работа с "плавающими" фреймами

"Плавающие" или внутренние фреймы задаются тегом <IFRAME> и для помещения их в окно браузера не нужен установочный файл с тегами <FRAMESET>. Плавающие фреймы, как и статические, при загрузке документа попадают в коллекцию (массив) frames текущего окна. Обращение к ним возможно по числовому индексу или значению атрибута name. Кроме того, можно использовать метод документа getElementById(). Загрузить документ в плавающий фрейм можно посредством свойства location или src. Однако загружать можно только те документы, которые находятся на том же сайте, что и исходный документ.

В листинге 4.13 в качестве примера приведен HTML-код документа с плавающим фреймом и двумя кнопками, с помощью которых изменяется содержимое фрейма, причем различными способами.

Листинг 4.13. Пример управления плавающими фреймами

```
<HTML>
<HEAD><TITLE>Плавающий фрейм</TITLE></HEAD>
<IFRAME src="http://dunaevv1.narod.ru"
 name="myframe" id="myframe"
 height=300 width=400></IFRAME>
<INPUT type="button" value="Примеры"
onclick="frames['myframe'].location =
'http://dunaevv1.narod.ru/examples.htm'">
 <INPUT type="button" value="Книги"
onclick="document.getElementById('myframe').src =
'http://dunaevv1.narod.ru/mybook.htm'">
</HTML>
```

4.5. Работа с каскадными таблицами стилей

Каскадные таблицы стилей (Cascading Style Sheets, CSS) позволяют задать внешний вид и позиционирование элементов документа. В сценариях доступ к каскадным таблицам стилей осуществляется через специальные объекты:

- объект элемента, соответствующего тегу <STYLE> или <LINK> (объект элемента стиля);

- объект `styleSheet` (элемент коллекции `styleSheets`);
- объект `cssRule` (элемент коллекций `rules` и `cssRules`);
- объект `style`.

Несмотря на то, что тег `<STYLE>` содержит строки правил таблицы стилей, объект элемента стиля не предназначен для получения доступа к отдельным правилам. Для этого служит объект `styleSheet`, являющийся элементом коллекции документа `styleSheets`. С помощью этого объекта можно подключить или отключить всю таблицу, получить доступ к отдельным правилам, добавить и удалить их. Доступ к отдельным правилам осуществляется посредством подобъектов `rules` или `cssRules`, являющихся коллекциями правил и представляющих собой массивы, к элементам которых можно обращаться с помощью индексов.

Для получения доступа к конкретному правилу можно воспользоваться выражением:

```
document.styleSheets[i].cssRules[j].
```

Напомним, что индексация элементов массива в JavaScript начинается с нуля.

Каждому тегу `<STYLE>` можно задать атрибут `ID`, чтобы вместо `styleSheets` использовать обращение по значению идентификатора. Объект элемента стиля обладает логическим свойством `disabled`. Значение `true` этого свойства деактивизирует (отключает) таблицу стилей, а значение `false` — активизирует (подключает). Пример приведен в листинге 4.14.

Листинг 4.14. Подключение таблицы стилей в сценарии

```
<HTML>
 <STYLE ID="mystyle">
  H1 {font-size:96pt; color:red}
  P {color:blue}
 </STYLE>
 <H1>Привет!</H1>
 <P>Привет, друзья!</P>
 <BUTTON onclick="document.getElementById("mystyle").disabled=true">
 Отключить стиль
 </BUTTON>
 <BUTTON onclick="document. getElementById("mystyle").disabled=false">
 Подключить стиль
 </BUTTON>
</HTML>
```

Объект `style` (не путайте с объектом элемента стиля) является основным. Он определен для любого HTML-элемента и обычно применяется для изменения стилевых параметров элементов документа. Для этого можно использовать следующий синтаксис:

```
document. getElementById(id_элемента).style.параметр=значение
```

Например:

```
document. getElementById("myelement").style.color="blue"
```

Объект `style` — это свойство не только объектов, представляющих HTML-элементы документа, но и объекта `cssRules`. Поэтому к конкретному стилевому параметру элемента документа, находящемуся под действием определенного правила таблицы стилей, можно обратиться, например, и так:

```
document.styleSheets[i].cssRules[j].style.color="blue"
```

Пример приведен в листинге 4.15.

Листинг 4.15. Обращение к стилевому параметру в сценарии

```
<HTML>
 <STYLE>
  H1 {font-size:96pt; color:red}
  P {color:blue}
 </STYLE>
 <H1>Привет!</H1>
 <P>Пишет тебе Вася Пупкин</P>
 <SCRIPT>
  x=document.styleSheets[0].cssRules[0].style.color // red
  y=document.styleSheets[0].cssRules[1].style.color // blue
  alert("Цвет в 1-м правиле: " + x+"\nЦвет во 2-м правиле: "+ y)
 </SCRIPT>
</HTML>
```

Таблицы стилей, вставленные в HTML-документ посредством тега `<STYLE>`, могут быть импортированы из внешних css-файлов с помощью оператора `@import`. В этих случаях для доступа к конкретному правилу импортированной таблицы возможна такая запись:

```
document.styleSheets[i].imports[k].rules[j]
```

Как уже отмечалось ранее, чтобы установить значение стилевого параметра, необходимо задействовать объект `style`. Это можно сделать, даже если таблица стилей для элемента документа не определена (листинг 4.16).

Листинг 4.16. Использование стилевого параметра без таблицы стилей

```
<HTML>
 <H1 ID="myheader">Добро пожаловать!</H1>
 <SCRIPT>
  document.getElementById("myheader").style.color="red"
 </SCRIPT>
</HTML>
```

Обратите внимание, что в этом HTML-коде таблица стилей нигде не определена, хотя сценарий устанавливает красный цвет для стилевого параметра `color` заголовка первого уровня.

Если таблица стилей для элемента документа не определена, то в объекте `style` для этого элемента все свойства имеют пустые значения, а не значения стилевых параметров, принятые по умолчанию. Поэтому при выполнении кода листинга 4.17 размер шрифта для заголовка будет не увеличен на 10, а установлен равным 10.

Листинг 4.17. Установка размера шрифта

```
<HTML>
 <H1 ID=myheader>Добро пожаловать!</H1>
 <SCRIPT>
  x=document.getElementById("myheader").style.fontSize; // пустая строка
  document.all.myheader.style.fontSize=x + 10;
  alert(document.getElementById("myheader").style.fontSize); // 10px
 </SCRIPT>
</HTML>
```

Значения стилевых параметров (свойств объекта `style`) строковые, поэтому их следует записывать в кавычках. Например, выражение

```
document.getElementById("myheader").style.color=red
```

приведет к ошибке.

Для параметров, связанных с размерами и координатами (например, `width`, `height`, `top`, `left`, `fontSize`), значения можно задавать в виде чисел без кавычек. Однако в действительности этим параметрам будут присвоены соответствующие строковые значения с указанием в качестве единицы измерения пикселов (`px`). Это обстоятельство необходимо иметь в виду особенно при составлении арифметических выражений для вычисления значений. Пример правильного применения арифметических операторов представлен в листинге 4.18.

Листинг 4.18. Использование арифметических операторов

```
<HTML>
 <H1 ID=myheader>Добро пожаловать!</H1>
 <SCRIPT>
  document.getElementById("myheader").fontSize=10;          // 10px
  document.getElementById("myheader").style.fontSize =
   parseInt(document.getElementById("myheader").fontSize) + 10; // 20px
 </SCRIPT>
</HTML>
```

Здесь функция `parseInt()` преобразует строку `"10px"` в целое число 10.

При необходимости задать размер шрифта не в пикселах, а в пунктах (`pt`), следует воспользоваться кодом из листинга 4.19.

Листинг 4.19. Установка размера шрифта в пунктах

```
<HTML>
 <H1 ID=myheader>Добро пожаловать!</H1>
 <SCRIPT>
  document.getElementById("myheader").style.fontSize="10pt"; // 10pt
  document.getElementById("myheader").style.fontSize =
  String(parseInt(document.getElementById("myheader").style.fontSize)+
    10) + "pt";                  // 20pt
 </SCRIPT>
</HTML>
```

Если таблица стилей вставлена в документ с помощью тега `<STYLE>` или `<LINK>` и задает параметры для некоторого HTML-элемента, то даже в этом случае, используя объект `style`, не удастся прочитать значения стилевых параметров (листинг 4.20).

Листинг 4.20. Попытка чтения стилевых параметров *(вариант 1)*

```
<HTML>
 <STYLE>
  H1 {color:red}
 </STYLE>
 <H1 ID="myheader">Добро пожаловать!</H1>
 <SCRIPT>
```

```
  alert(document.getElementById("myheader").style.color) /* выводится
                                                        пустая строка */
 </SCRIPT>
</HTML>
```

Если же таблица стилей задана как значение атрибута `style`, то с помощью объекта `style` прочитать значения определенных стилевых параметров можно (листинг 4.21).

Листинг 4.21. Попытка чтения стилевых параметров *(вариант 2)*

```
<HTML>
 <H1 ID="myheader" STYLE="color:red">Добро пожаловать!</H1>
 <SCRIPT>
  alert(document.getElementById("myheader").style.color) /* выводится
                                                          строка "red" */
 </SCRIPT>
</HTML>
```

Следует заметить, что свойства, соответствующие стилевым параметрам, в сценариях записываются не так, как в правилах каскадных таблиц стилей. Названия свойств, как и имена переменных, в сценариях не должны содержать дефисов, широко распространенных в таблицах стилей. Поэтому названия стилевых параметров в сценариях образуются из имен соответствующих параметров таблиц стилей путем перезаписи последних в стиле `interCap`. Это означает, что в составных именах опускают дефисы, а второе и последующие слова записывают с прописной буквы. Например, параметру `font-size` в таблице стилей соответствует свойство `fontSize` в объекте `style`. В табл. 4.6 приведено соответствие между способами представления некоторых параметров в правилах таблиц стилей и сценариях (как свойств стиля).

***Таблица 4.6.** Соответствие между именами параметров в CSS и в сценариях*

Имя параметра в таблице стилей	Свойство стиля
`width`	`width`
`height`	`height`
`z-index`	`zIndex`
`font-size`	`fontSize`
`border-color`	`borderColor`
`border-top-color`	`borderTopColor`
`list-style-image`	`listStyleImage`

4.6. Управление во времени

Управление вычислительными процессами во времени обеспечивают методы объекта `window`: `setInterval()`, `clearInterval()` и `setTimeout()`. Вы можете периодически, через заданный интервал, вызывать одну или несколько функций, а также задерживать их вызов.

Для организации периодического выполнения некоторого выражения или функции служит метод `setInterval()` объекта `window`. Этот метод имеет следующий синтаксис:

`setInterval(`*выражение*`,` *период* `[,` *язык*`]).`

Первый параметр представляет собой строку, содержащую выражение (в частности, вызов функции). Второй параметр — целое число, указывающее временну́ю задержку в миллисекундах перед последующими выполнениями выражения, заданного в первом параметре. Третий, необязательный параметр определяет язык, на котором написано выражение; по умолчанию — JavaScript.

Метод `setInterval()` возвращает некоторое целое число — идентификатор временного интервала, который может быть использован в дальнейшем для прекращения выполнения процесса, запущенного с помощью данного метода (см. далее метод `clearInterval()`).

Пусть, например, требуется, чтобы некоторая функция `myfunc()` выполнялась периодически через 0,5 с. Тогда в сценарии следует записать следующее выражение:

```
setInterval("myfunc()", 500).
```

Тот факт, что первый параметр метода `setInterval()` является строкой, обусловливает некоторые особенности передачи параметров периодически вызываемой функции. Если она принимает параметры, то мы должны сначала сформировать строку, содержащую имя этой функции, круглые скобки, значения параметров и запятые между ними, а затем передать ее в качестве первого параметра методу `setInterval()`. В следующем примере показано, как передать методу `setInterval()` функцию с двумя параметрами `param1` и `param2`, значения которых определены в другом месте сценария:

```
var xstr="myfunc(" + param1 + "," + param2 +")";
setInterval(xstr, 500);
```

Выражение, переданное методу `setInterval()`, будет периодически выполняться сколь угодно долго. Если оно осуществляет, например, приращение координат какого-нибудь видимого элемента документа, то этот элемент станет перемещаться в окне браузера.

Для остановки запущенного временного процесса служит метод

`clearInterval(`*идентификатор*`)`,

который принимает в качестве параметра целочисленный идентификатор, возвращаемый соответствующим методом `setInterval()`, например:

```
var myproc=setInterval("myfunc(), 100");
if (confirm("Прервать процесс ?")) clearInterval(myproc);
```

В листинге 4.22 приведен пример использования методов `setInterval()` и `clearInterval()` для задания движения заголовка (элемента `<H1>`) по горизонтали.

Листинг 4.22. Движение заголовка

```
<HTML>
<HEAD><TITLE>Движение</TITLE></HEAD>
<H1 id="myzag" style="position:absolute; left:10;top:50">
Движение — это жизнь
</H1>
<INPUT type="button" onclick="clearInterval(myinterval)"
value="Остановить">
<SCRIPT>
var flag=false;
var myinterval=setInterval("move()",10) // запуск перемещения
function move(){
 var x=parseInt(document.getElementById("myzag").style.left);
 if (x==500) {flag=true};
 if (x==10) flag=false;
 if (flag) d=-1; else d=1;
 document.getElementById("myzag").style.left=x+d;
}
</SCRIPT>
</HTML>
```

Выполнить выражение с некоторой временной задержкой позволяет метод `setTimeout()`, имеющий следующий синтаксис:

`setTimeout(`*выражение*`, `*задержка*` [, `*язык*`])`.

Первый параметр представляет собой строку, содержащую выражение (в частности, вызов функции). Второй параметр — целое число, указывающее временную задержку в миллисекундах выполнения выражения, заданного

в первом параметре. Третий, необязательный параметр указывает язык, на котором написано выражение; по умолчанию — JavaScript.

Метод `setTimeout()` возвращает некоторое целое число — идентификатор временного интервала, который может быть использован в дальнейшем для отмены задержки выполнения процесса, запущенного с помощью данного метода.

Пусть, например, требуется, чтобы некоторая функция `myfunc()` выполнялась спустя 1 с. Тогда в сценарии нужно записать следующее выражение:

```
setTimeout("myfunc()", 1000).
```

> ***Примечание***
>
> Помните, что это выражение не задерживает выполнение всех последующих выражений сценария. Оно задерживает лишь выполнение функции `myfunc()`.

Для отмены задержки процесса, запущенного с помощью метода `setTimeout()`, предусмотрен метод

```
clearTimeout(идентификатор),
```

который принимает в качестве параметра целочисленный идентификатор, возвращаемый соответствующим методом `setTimeout`.

В следующем HTML-документе имеются две кнопки (листинг 4.23). Щелчок на первой кнопке через 5 с открывает новое окно и загружает в него документ. Однако это действие можно отменить с помощью второй кнопки, если щелкнуть на ней, пока новое окно еще не открыто.

Листинг 4.23. Использование временной задержки

```
<HTML>
<BUTTON ID="start" onclick="myproc=start()">
Загрузить с задержкой 5с
</BUTTON>
<BUTTON ID="stop" onclick="stop(myproc)">
Отменить загрузку
</BUTTON>
 <SCRIPT>
 function start(){
  return=setTimeout("window.open('http://dunaevv1.narod.ru')", 5000)
 }
 function stop(proc){
```

```
  clearTimeout(proc)
 }
</SCRIPT>
</HTML>
```

Примечание

При работе с методами `setInterval()` и `setTimeout()` следует иметь в виду, что их вторые числовые параметры задают лишь приблизительные значения временных задержек.

4.7. Работа с cookie

Для хранения небольших объемов информации на диске компьютера пользователя в браузере предусмотрен так называемый механизм cookie. Обычно он применяется для хранения имени пользователя и пароля, который вводится в поле формы защищенного Web-сайта, а также информации о предыдущем посещении сайта. Например, можно сохранить на диске дату последнего посещения сайта данным пользователем. При загрузке сайта это значение сравнивается с некоторой датой, установленной автором сайта в качестве даты обновления. Если вторая (авторская) дата более поздняя, чем первая, то на Web-странице появляется соответствующая отметка, например, текст или изображение с надписью "New" или "Обновлено!". Разумеется, сравнение дат и вывод на страницу отметки реализует сценарий. А вот еще один пример: сценарий проверяет, посетил ли сайт, скажем, Вася Пупкин; если это произошло, то на страницу выводится персональное сообщение.

По существу, cookie — это пока единственный более или менее безопасный способ сохранения данных на диске компьютера пользователя. Как известно, Web-браузеры препятствуют свободному обращению к папкам и файлам на клиентском компьютере. Вместе с тем, следует помнить, что многие пользователи не любят cookie, блокируют их запись или удаляют по завершении сеанса.

Записи cookie браузер Internet Explorer сохраняет в отдельных текстовых файлах, расположенных в папке Cookies. Расположение этой папки в системе Windows зависит от версии. Имя файла для записи cookie обычно образуется на основе имени пользователя и домена того сервера, на котором создавался cookie-файл. Вообще говоря, структура данных в cookie-файлах для различных браузеров не столь существенна, поскольку, во-первых, не рекомендуется открывать и изменять эти файлы в текстовых редакторах, а во-вторых, браузеры используют одинаковый синтаксис чтения и записи cookie-данных, основанный на свойстве `document.cookie`.

Итак, данные в cookie-файлах организованы в виде записей. Каждую такую запись можно представить себе в виде строки, содержащей следующие элементы (поля):

- имя записи;
- содержание записи;
- срок хранения (годности) записи;
- домен сервера, который создал запись;
- сведения о необходимости установки безопасного HTTP-соединения для доступа к записям;
- расположение документов, которым разрешен доступ к записям.

Зависимость записей от домена обеспечивает безопасность хранения так называемых невосстанавливаемых паролей (пар вида *имя_пользователя-пароль*), поскольку запись, созданную сервером одного домена, не может прочитать сервер с другим доменом.

Учитывая срок хранения, браузер автоматически удаляет просроченные записи, чтобы предотвратить чрезмерное разрастание объема cookie-файлов. Запись данных в cookie-файл с помощью JavaScript осуществляется выражением присваивания строки, содержащей cookie-данные, свойству `document.cookie`. При этом важно соблюдать формат строки:

```
document.cookie="cookieName=данные
[; expires=строка_времени_GMT]
[; path=путь]
[; domain=домен]
[; secure]"
```

Здесь квадратные скобки указывают, что заключенное в них содержимое не является обязательным (может быть опущено).

Рассмотрим элементы cookie-строки.

- `cookieName` — каждая cookie-запись должна иметь имя и строковое значение, которое может быть и пустой строкой. Например, если требуется сохранить слово "Вася" в cookie-записи с именем `User_Name`, то соответствующее выражение JavaScript будет иметь вид:

  ```
  document.cookie="User_Name=Вася".
  ```

  При выполнении этого выражения браузер попытается найти cookie-запись с таким именем. Если он не найдет ее в текущем домене, то создаст автоматически. Если запись с таким именем уже существует, то браузер

заменит ее данные новыми. Элемент cookie-записи не должен содержать точек с запятыми, запятых и пробелов. Чтобы заменить их соответствующими символами, строка с данными предварительно обрабатывается функцией `escape()`.

- `expires` — дата и время хранения (годности) cookie-записи должны быть представлены строкой и содержать данные по Гринвичу (GMT). Например, вычислить дату истечения срока хранения записи месяц спустя от текущей даты можно следующим образом:

```
var expdate=new Date()      // создаем объект даты
var monthFromNow=expdate.getTime() + (30*24*60*60*1000)
expdate.setTime(monthFromNow) // устанавливаем значение даты
```

После этого полученную дату следует привести к строковому формату GMT:

```
document.cookie="User_Name=Вася; expires=" + expdate.toGMTString().
```

Cookie-запись можно удалить и до истечения заданного срока хранения, установив новый срок, заведомо уже прошедший:

```
expdate=Thu, 01-Jan-70 00:00:01 GMT.
```

Отсутствие срока хранения означает для браузера, что данная cookie-запись временная и не подлежит записи в файл.

Об использовании методов объекта даты *см. разд. 3.12.2*.

- `path` — cookie-записи, производимые компьютером пользователя, имеют путь, принятый по умолчанию (в текущей папке). Однако можно создать копию cookie в другой папке, указав путь к ней в качестве значения этого параметра.
- `domain` — для синхронизации cookie-данных с определенным документом или и группой документов браузер выясняет домен текущего документа и помещает в cookie-файл записи, соответствующие этому домену. Если пользователю требуется получить список всех cookie-записей, содержащихся в свойстве `document.cookie`, то он должен просмотреть все пары *имя-значение*, находящиеся в cookie-файле с именем домена текущего документа. Формат представления домена должен охватывать, по крайней мере, два уровня, например, `rambler.ru`.
- `secure` — принимает логические значения (`true` или `false`). При создании cookie-записей на стороне клиента (компьютера пользователя) этот параметр опускается.

Теперь рассмотрим чтение и запись данных cookie. Данные cookie, которые можно получить с помощью сценария на JavaScript, представляют собой единственную строку — значение свойства `document.cookie`. Значения отдельных элементов (параметров) cookie выбирают на основе анализа содержимого этой строки методами объекта `String`. Кроме того, если две и более cookie-записи (до 20) соответствуют одному и тому же домену, то в JavaScript их все равно записывают в одну строку и разграничивают точкой с запятой и пробелом. Далее мы приведем коды функций, которые обеспечивают работу с cookie.

Рассмотрим сначала функцию `readCookie(name)`, читающую cookie-данные, соответствующие имени записи `name`, которое передается этой функции в качестве параметра (листинг 4.24).

Листинг 4.24

```
function readCookie(name) {  // чтение cookie-данных записи
 var xname=name + "=";
 var xlen=xname.length;
 var clen=document.cookie.length;
 var i=0;
 while(i < clen){
  var j=i + xlen;
  if (document.cookie.substring(i, j) == xname)
   return getCookieVal(j);
  i=document.cookie.indexOf(" ",1) + 1;
  if (i == 0) break;
 }
 return null
}
```

Функция `readCookie(name)` возвращает значение cookie-записи с именем `name` или `null`, если такая запись не найдена. В теле этой функции содержится вызов еще одной, вспомогательной, функции `getCookieVal(j)`, возвращающей декодированное значение cookie-данных. Декодирование производится с помощью встроенной функции `unescape()`. Дело в том, что cookie-запись должна представлять собой кодированную строку, полученную путем обработки встроенной функцией `escape()`, чтобы, в частности, заменить пробелы специальными символами (%20). Код этой функции приведен в листинге 4.25.

Листинг 4.25

```
function getCookieVal(n){  /* вспомогательная функция, вызываемая
                из readCookie() */
 var endstr=document.cookie.indexOf(";", n);
 if (endstr == -1) endstr=document.cookie.length;
 return unescape(document.cookie.substring(n, endstr))
}
```

Теперь рассмотрим функцию `writeCookie()`, позволяющую создать или обновить cookie-запись (листинг 4.26).

Листинг 4.26

```
function writeCookie(name, value, expires, path, domain, secure) {
  /* запись cookie */
 document.cookie =
  name + "=" + escape(value) +
  ((expires) ? "; expires=" + expires.toGMTString(): "") +
  ((path) ? "; path=" + path : "") +
  ((domain) ? "; domain=" + domain : "") +
  ((secure) ? "; secure" : "")
}
```

Эта функция принимает следующие параметры:

- `name` — строка, содержащая имя cookie-записи (обязательный параметр);
- `value` — строка, содержащая значение cookie (обязательный параметр);
- `expires` — объект даты (`Date`), содержащий срок хранения cookie-записи; если отсутствует, то после завершения работы браузера cookie-запись удаляется;
- `path` — строка, содержащая путь cookie-записи; если параметр не указан, то используется путь вызванного документа;
- `domain` — строка, содержащая домен нужной cookie-записи; если параметр не указан, то используется домен вызванного документа;
- `secure` — логическое значение (`true` или `false`), определяющее необходимость использования безопасного HTTP-соединения.

Обратите внимание, что в теле функции `writeCookie()` происходит кодирование значения параметра `value` с помощью встроенной функции `escape()`.

Для удаления cookie-записи можно вызывать следующую функцию (листинг 4.27).

Листинг 4.27

```
function deleteCookie(name, path, domain) {
 /* удаление cookie-записи */
 if (readCookie(name)) {
  document.cookie =
   name + "=" +
   ((path) ? "; path=" + path : "") +
   ((domain) ? "; domain="+domain : "") +
   "; expires=Thu, 01-Jan-70 00:00:01 GMT"
 }
}
```

Эта функция устанавливает дату срока хранения cookie-записи так, что запись будет удалена. Значение параметра `path` должно быть таким же, как при создании cookie-записи, или пустым (`null`), если при создании записи он не был определен. Так же задается значение параметра `domain`.

Советую вам поэкспериментировать с чтением, созданием и удалением cookie-записей (см. листинг 4.28). При этом следует иметь в виду, что созданные или измененные cookie-записи будут записаны на диск только после закрытия браузера. До этого они существуют лишь в кэше (оперативной памяти). С другой стороны, cookie-записи загружаются в оперативную память и становятся доступными как значение свойства `document.cookie` только при запуске браузера.

Листинг 4.28. HTML-код для эксперимента со сценарием, записывающим и читающим cookie-запись

```
<HTML>
 <SCRIPT>
  function readCookie(name) { // чтение cookie-данных записи
   ... // код
  }

  function getCookieVal(n){  // вспомогательная функция, вызываемая
                 // из readCookie()
   ... // код
```

```
  }

  function writeCookie(name, value, expires, path, domain, secure) {
  // запись cookie
   ... // код
  }

  / * Срок хранения — 1 год от текущей даты: */
  var d1=new Date();
  var d2=d1.getTime() + (365*24*60*60*1000);
  d1.setTime(d2);
  /* Запись и чтение cookie: */
  writeCookie("myrecord","Привет!",d1);
  alert("Содержимое записи cookie:\n"+readCookie("myrecord"))
 </SCRIPT>
</HTML>
```

4.8. Работа с таблицами

Таблицы играют значительную роль в компоновке содержимого документа. Те, кто не любит использовать фреймы, применяют вместо них таблицы. Как известно, таблицы задаются контейнерными тегами <TABLE>, <THEAD>, <TFOOT>, <TBODY>, <TR>, <TD> и <TH>. Как объект документа, таблица имеет две коллекции, посредством которых осуществляется доступ к ее содержимому:

- rows — коллекция строк; содержит все строки таблицы, включая разделы <THEAD> и <TFOOT>;
- cells — коллекция ячеек одной строки; содержит элементы таблицы, созданные посредством тегов <TH> и <TD>.

К элементам коллекций можно обращаться по числовому индексу, а также по значению атрибута id. Ячейки в коллекции элементов одной строки (cells) нумеруются слева направо, начиная с нуля.

> ***Примечание***
>
> В Internet Explorer объект таблицы имеет свою коллекцию всех ячеек cells, а коллекция строк rows — свою. Ячейки в коллекции cells всей таблицы нумеруются слева направо и сверху вниз, т. е. имеют сквозную нумерацию.

Например, для доступа к *j*-й ячейке *i*-й строки таблицы можно использовать такое выражение:

```
document.getElementById("mytab").rows[i].cells[j].
```

В листинге 4.29 приведен пример таблицы и обращения к ее ячейкам из сценария. При этом обращение к ячейкам выполнено различными способами. Собственно содержимое ячеек получается как значение свойства `innerHTML` — строки HTML-кода, находящегося внутри контейнерного тега элемента. Значение свойства `innerHTML` можно изменять и, следовательно, модифицировать содержимое ячеек. Подробнее о свойстве `innerHTML` будет рассказано в *разд. 4.12.4*. В данном примере читается значение нулевой ячейки первой строки (второй сверху), а затем заменяется строкой "Пупкин". Для браузера Internet Explorer должно сработать обращение к ячейке через общую коллекцию `cells`, а в Mozilla Firefox, например, соответствующий код будет проигнорирован. Вид рассматриваемого документа в окне Internet Explorer показан на рис. 4.9.

Листинг 4.29. Пример работы с таблицей

```
<HTML>
<HEAD><TITLE>Работа с таблицей</TITLE></HEAD>
<TABLE id="mytab" border=1>
<TR><TD>Иванов</TD><TD>1000</TD></TR>
<TR><TD id="x">Петров</TD><TD>2000</TD></TR>
<TR><TD>Сидоров</TD><TD>3000</TD></TR>
</TABLE>
<BR>
<SCRIPT>
if(document.getElementById("mytab").cells){// для IE
    document.write("В ячейке 2 находится "+
    document.getElementById("mytab").cells[2].innerHTML,"<BR>")};
document.write("Был "+
document.getElementById("mytab").rows[1].cells[0].innerHTML,",<BR>");
document.write("а стал "+
(document.getElementById("mytab").rows[1].cells['x'].innerHTML=
"<B>Пупкин</B>"));
</SCRIPT>
</HTML>
```

Метод `insertRow()` добавляет новую строку в таблицу. Он возвращает ссылку на вновь созданную строку, которая затем используется для вставки ячеек. Ячейки вставляются в строку с помощью метода `insertCell(`*`индекс_ячейки`*`)`. Данный метод возвращает ссылку на созданную ячейку, которая в дальнейшем служит для задания содержимого ячейки. Ячейки вставляются в строку по порядку без пропусков, начиная с нулевой. Так, нельзя создать ячейку с индексом 3, если ячейки с индексами 0, 1 и 2 еще не созданы.

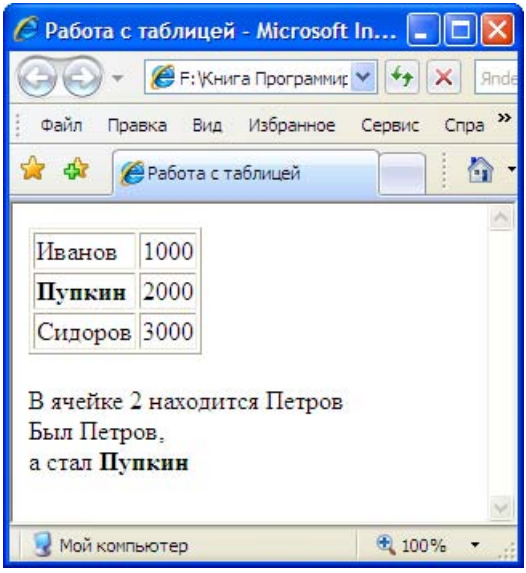

Рис. 4.9. Вид документа из листинга 4.29

Примечание

В Internet Explorer при добавлении новой строки вы можете последовательно создать любое количество ячеек, даже превышающее их число в уже имеющихся строках. В других браузерах это не так.

Таким образом, новая строка создается в три этапа: сначала к таблице добавляется новая строка, затем в нее вставляются ячейки, после чего они заполняются значениями. Листинг 4.30 иллюстрирует пример добавления строки к существующей таблице и заполнения ее ячеек значениями.

Листинг 4.30

```
newrow=document.getElementById("mytab").insertRow(); /* добавляем новую
                                                        строку */
newcell=newrow.insertCell(0);       // вставляем ячейку с индексом 0
newcell.innerHTML="Ипполит";        // заполняем ячейку значением
newcell=newrow.insertCell(1);
newcell.innerHTML="<IMG SRC='pict.jpg'>"; // вставляем картинку
```

Для удаления строки таблицы служит метод `deleteRow(`*`индекс_строки`*`)`. Параметр указывает номер удаляемой строки:

```
document.getElementById("mytab").deleteRow(2).
```

4.9. Работа с формами

В HTML форма создается с помощью контейнерного тега <FORM>, внутри которого располагаются теги элементов пользовательского интерфейса — поля ввода данных, текстовые области, переключатели и флажки, раскрывающиеся списки и кнопки. В объектной модели документа каждой форме соответствует свой объект, входящий в коллекцию forms. Форма предназначена главным образом для организации отправки на сервер всех данных, имеющихся в элементах этой формы, — введенных пользователем и/или сгенерированных сценарием. Для отправки данных на сервер сценарий не обязателен. Чтобы отправить данные, достаточно в теге <FORM> атрибуту ACTION присвоить надлежащее значение — URL-адрес получателя, а в самой форме разместить кнопку типа **Отправить** (submit). Щелчок на этой кнопке инициализирует отправку данных. Если атрибут ACTION не указан или его значение пусто, данные формы не будут отправлены, даже если вы щелкнете на кнопке типа **Отправить** (submit).

> ***Внимание***
>
> Обычную кнопку типа submit (<input type=submit>) можно заменить графическим изображением (<input type=image src="адрес_файла">).

Кроме action в теге <FORM> следует указать еще два атрибута: metod и enctype. Атрибут method имеет значение POST или GET. В методе GET передаваемые данные видны в адресной строке браузера и следуют за адресом. Обычно этот метод применяют для передачи небольшого объема данных. Если у вас нет особых причин задумываться об этом, выбирайте значение POST. Атрибуту enctype присвойте значение "text/plain". В этом случае отправляемое сообщение будет представлять собой последовательность пар вида *имя_элемента=значение*. Здесь *имя_элемента* — значение атрибута name в теге элемента, содержащегося в форме, а *значение* — значение атрибута value в этом же теге. Если не указывать атрибут ENCTYPE, то сообщение будет представлено в неудобочитаемом (закодированном) виде.

4.9.1. Проверка данных перед отправкой

Если перед отправкой данных формы требуется предварительно их проверить или еще что-нибудь сделать, то для этого необходим сценарий. В примере из листинга 4.31 проверяется, имеется ли символ @ в поле ввода адреса электронной почты отправителя и не пусты ли другие поля ввода. Если символа @ в адресе нет или хотя бы одно поле пусто, то отправка не производится. Функция validator() обрабатывает событие onsubmit, возникающее при

щелчке на кнопке типа **Отправить** (submit), и возвращает значение `true`, если все условия выполнены и `false` — в противном случае. Чтобы возвращаемое значение могло разрешить или запретить действие обработчика `onsubmit` по умолчанию, в теге `<FORM>` указан атрибут `onsubmit="return validator()"`. Форма содержит еще одно невидимое поле `<INPUT NAME="browser" TYPE="hidden" VALUE="">`, значение которого (название браузера пользователя) формируется программно.

Листинг 4.31. Проверка данных формы перед отправкой

```
<HTML>
<HEAD><TITLE>Отправка данных по E-mail</TITLE></HEAD>
<META http-equiv="conten-type" content="text/html; charset=windows-1251">
<FORM ID="myform" METHOD=POST
  ACTION="http://www.anyserver.ru/program.php"
  onsubmit="return validator()"
>
  Ваше имя:
  <INPUT NAME="username" id="username" TYPE="text" VALUE="">
  <P>
  Ваш e-mail:
  <INPUT NAME="email" TYPE="text" VALUE="">
  <P>
  Ваше сообщение:<BR>
  <TEXTAREA NAME="message" TYPE="text" VALUE=""></TEXTAREA>
  <BR>
  <INPUT NAME="browser" TYPE="hidden" VALUE="">
  <P>
  <INPUT NAME="send" TYPE="submit" VALUE="Отправить">

</FORM>
<SCRIPT>
document.getElementById('username').focus();
function validator(){ // проверка данных формы
 var nousername=myform.username.value=="";
 var noemail=(myform.email.value.indexOf('@') == - 1);
 var nomsg=myform.message.value=="";

 if (nousername || noemail || nomsg){
  errortext="Одно из полей не заполнено,";
```

```
  errortext+="\nили неверно указан адрес e-mail.";
  errortext+="\nСообщение не отправлено";
  alert(errortext);
  return false // отменить отправку
 } else {
  myform.browser.value=document.navigator.appName; // название браузера
  return true
 }
}
</SCRIPT>
</HTML>
```

4.9.2. Создание баннера

Баннер — небольшое графическое изображение (часто анимированное), которое что-то рекламирует. Нередко щелчок на баннере просто загружает в браузер некоторую страницу. В данном случае это просто графическая ссылка, создаваемая HTML-элементом `<A HREF=…><IMG src=…></A>`. Поскольку функциональность баннера реализуется тривиальным образом, дизайнеры сосредотачиваются на его внешнем виде. Но не в этом главная сущность этого элемента для программиста. Баннер — конструкция, содержащая что-то видимое и обеспечивающая некоторые функции. С точки зрения функциональности баннер должен обеспечить передачу каких-то данных на сервер, который как-то их обработает. Для создания баннера очень подходит HTML-элемент `<FORM>`, т. е. контейнер для элементов пользовательского интерфейса, таких как поля ввода данных, а также кнопки и различного рода переключатели. В баннер обычно ничего не нужно вводить, а потому он, как форма, должен содержать только невидимые (скрытые) элементы, получаемые из обычных с помощью атрибута `type="hidden"`. Однако они могут содержать данные (значения атрибута `value`), например, URL-адрес страницы, сведения о браузере пользователя и др., необходимые серверному приложению для учета щелчков и, в частности, для начисления бонусов владельцу страницы с данным баннером. Элементы формы могут содержать предустановленные данные (значения атрибутов `value`) или сформированные программно путем присвоения свойству `value` требуемого значения.

Форма для баннера, кроме скрытых полей, должна содержать элемент `<INPUT type="image" src=url-адрес_изображения>`. Это графическое изображение, которое играет ту же роль, что и кнопка типа submit. Иначе говоря, щелчок на ней приводит к передаче данных, содержащихся во всех элементах формы, в том числе и скрытых. Разумеется, файл графического изображения, указан-

ный в атрибуте src, должен иметь надлежащее содержание, соблазняя посетителя страницы щелкнуть на его картинке. Но сейчас не об этом. Пример кода баннера приведен в листинге 4.32.

Листинг 4.32. Пример кода баннера

```
<HTML>
<FORM NAME="myform"
 ACTION="http://www.anyserver.ru/anyprogram.php" METHOD=POST
 onsubmit="send()">
<INPUT TYPE="hidden" NAME="mydata">
<INPUT TYPE="image" SRC="grbutton.png">
</FORM>
<SCRIPT>
function send(){
       myform.mydata.value=window.location.href;
}
</SCRIPT>
</HTML>
```

Таким образом, если форма содержит только скрытые элементы, то посетитель Web-страницы может и не догадаться, что, щелкнув на картинке или кнопке, он отправляет какие-то данные, сформированные программно.

4.9.3. Переходы между полями по клавише <Enter>

Если форма на Web-странице содержит несколько полей ввода, то переход между ними можно осуществить с помощью мыши или клавиши <Tab>. Однако в обычных приложениях то же самое можно сделать с помощью клавиши <Enter>. Казалось бы, переход к следующему полю формы при нажатии на клавишу <Enter> можно реализовать с помощью метода focus() по событию onkeypress. Но на Web-странице нажатие на клавишу <Enter> при активном поле формы приводит по умолчанию к отправке данных на сервер, как если бы вы щелкнули на кнопке типа submit. Задача состоит в том, чтобы по клавише <Enter> осуществить переход к следующему полю, не передавая данные на сервер.

В следующем примере (листинг 4.33) форма содержит три поля ввода данных и обычную кнопку (<INPUT type="button">). Если поле активно (курсор находится в поле), то при нажатии на клавишу <Enter> происходит активиза-

ция следующего поля. При этом если активизируемый элемент является кнопкой типа submit, то происходит передача данных формы на сервер. Разумеется, щелчок на кнопке типа submit также вызывает отправку данных. В рассматриваемом примере нажатие на кнопку <Enter> в последнем поле ввода приводит к передаче данных. Обратите внимание, что форма содержит обычную, а не специальную кнопку типа submit, а потому передача данных инициируется принудительно путем программной генерации события `onsubmit` с помощью метода `submit()`. О программном вызове обработчиков событий и программной генерации событий уже говорилось в *разд. 4.3.2*.

Листинг 4.33. Пример переходов между полями по клавише <Enter>

```
<HTML>
<FORM id="myform"
 action="http://www.anyserver.ru/program.php" method=POS >
<input type="text" id="inp1" onkeypress=keypr(event,"inp2")><br>
<input type="text" id="inp2" onkeypress=keypr(event,"inp3")><br>
<input type="text" id="inp3" onkeypress=keypr(event,"subm")><br>
<input type="button" id="subm" value="Отправить"
onclick="document.forms['myform'].submit()"
onkeypress=keypr(event,"subm")>
</FORM>

<SCRIPT>
document.getElementById("inp1").focus() // фокус на 1-м поле
function keypr(event,xid){ // обработка нажатия на клавишу
/* event — объект события;
  xid - id элемента, на который следует установить фокус
*/
if (event.keyCode==13) { // если нажали на клавишу <Enter>
  document.getElementById(xid).focus(); // установить фокус на xid
  if (xid=="subm") document.forms['myform'].submit(); // отправить
}
}
</SCRIPT>
</HTML>
```

Примечание

Если для передачи данных в форму вставить не обычную кнопку, а кнопку типа submit, то возникнут проблемы, связанные с тем, что при нажатии на клавишу <Enter> автоматически вызывается обработчик события `onsubmit` по умолчанию, т. е. делается преждевременная попытка передать данные.

4.10. Создание меню

Меню часто встречаются на Web-страницах. Они могут иметь различный внешний вид и внутреннюю конструкцию. Обычно меню создают на основе раскрывающихся списков (<SELECT> и <OPTION>) или таблиц (<TABLE>, <TR>, <TD>,…). В любом случае необходимо обеспечить программную обработку выбора опции (пункта меню), произведенного пользователем, или, другими словами, придать перечням опций некоторую функциональность. В данном разделе мы рассмотрим оба способа построения меню.

4.10.1. Меню на основе раскрывающегося списка

Простейшее одноуровневое меню можно создать на основе тегов <SELECT> и <OPTION>, задающих так называемый раскрывающийся список. Впрочем, список может быть частично или полностью раскрытым в зависимости от числового значения атрибута size тега <SELECT>. Этот атрибут указывает количество одновременно видимых опций меню.

В листинге 4.34 приведен пример кода меню, в котором можно выбрать сайт для последующего перехода на него. Сценарий определяет номер выбранной пользователем опции (свойство selectedIndex), а затем загружает в браузер документ по адресу, указанному в соответствующем атрибуте value. Внешний вид меню показан на рис. 4.10. Заметим, что его можно модифицировать, задав надлежащие стилевые параметры, но сейчас не об этом.

Листинг 4.34. Меню на основе раскрывающегося списка

```
<HTML>
<HEAD><TITLE>Раскрывающийся список</TITLE></HEAD>
Перейти на:
<SELECT ID="mymenu" onchange ="myselection()">
 <OPTION value="" selected>Выберите сайт
 <OPTION value="http://www.yandex.ru">Яндекс
 <OPTION value="http://www.rambler.ru">Rambler
 <OPTION value="http://www.google.com">Google
 <OPTION value="http://dunaevv1.narod.ru">Сам себе Web-дизайнер
</SELECT>
<SCRIPT>
function myselection() {
  var i=document.getElementById("mymenu").selectedIndex;
```

```
  if (i>0)
       window.location.href=
     document.getElementById("mymenu").options[i].value;
}
</SCRIPT>
</HTML>
```

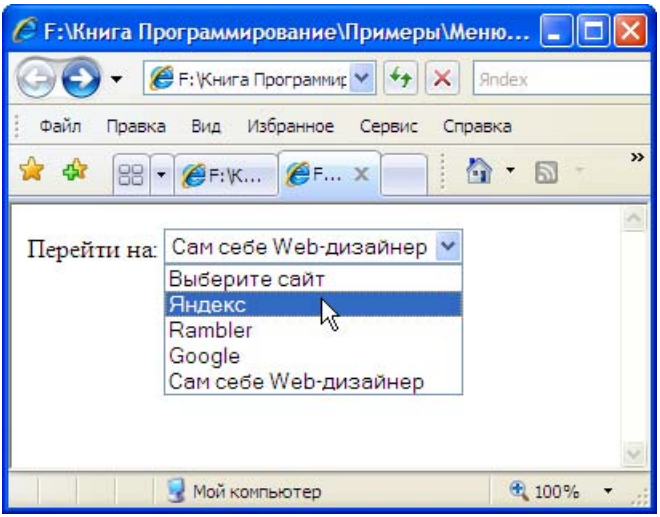

Рис. 4.10. Меню на основе раскрывающегося списка

В данном примере элемент `<SELECT>`, задающий внутреннее содержимое (атрибут `value` тега `<OPTION>`) и отображаемые на экране тексты опций списка, реагирует на изменение выбранной опции (событие `onchange`), которое производит пользователь щелчком на той или иной опции. Реакция состоит в вызове обработчика данного события — функции `myselection()`, которая определяет номер (индекс) выбранной опции посредством выражения

```
document.getElementById("mymenu").selectedIndex,
```

а затем по этому индексу определяет значение свойства `value` соответствующего элемента списка, представленного в объектной модели как элемент коллекции (массива) `options`. В рассматриваемом примере свойство `value` хранит URL-адрес документа, который требуется загрузить в браузер. Этот адрес присваивается свойству `window.location.href`, чего вполне достаточно для действительного осуществления загрузки (если, разумеется, такой документ существует и доступен). Очевидно, можно написать и другой сценарий обработки выбранной опции меню.

4.10.2. Двухуровневое меню на основе таблиц

Меню можно создать не только на основе раскрывающегося списка, но и путем использования табличных тегов. Эта идея практически очевидна. В самом деле, меню легко представить как набор ячеек, внутри которых находятся тексты опций меню. Цвет фона ячеек и текстов в них можно задать и далее в сценарии управлять ими в зависимости от действий пользователя. В данном разделе мы рассмотрим двухуровневое меню, состоящее из горизонтального главного меню и вертикальных раскрывающихся подменю. Горизонтальное меню отображается постоянно, а раскрывающиеся подменю появляются при наведении указателя мыши на элементы горизонтального меню и скрываются при уходе указателя мыши с подменю. Такова, в общих чертах, идея двухуровневого меню, которая осуществлена в обычных приложениях, но далеко не всегда реализуется на Web-страницах.

Двухуровневое меню можно составить из нескольких одностолбцовых таблиц, расположенных в одном ряду. В исходном состоянии видны только самые верхние ячейки всех таблиц, которые и образуют горизонтальное главное меню. При наведении указателя мыши на отображаемую ячейку становятся видимыми и другие ячейки той же таблицы (раскрывается подменю), а при уходе указателя мыши подменю вновь становится невидимым. Остается еще выделить цветом пункт подменю, на котором находится указатель, а также обработать щелчок на нем. Видимость, а также цвет фона и шрифта пунктов подменю устанавливаются значениями соответствующих стилевых параметров.

В листинге 4.35 приведен пример двухуровневого меню (рис. 4.11), в котором главное горизонтальное меню содержит три опции и только первым двум из них соответствуют раскрывающиеся подменю.

Листинг 4.35. Двухуровневое меню на основе таблиц

```
<html>
<head><title>Простое двухуровневое меню</title></head>
<h1>Пример двухуровневого меню</h1>
<table id="menu1" border=1
  style="position:absolute;top:100;left:50;width:80;z-index:5;"
  rules=none cellpadding=2 cellspacing=0
  onmouseout="document.getElementById('tb1').style.display='none'"
  onmouseover="document.getElementById('tb1').style.display='';
  document.getElementById('tb1').style.background='#e0e0e0'"
  onclick="menuclick(event)"
```

```
>
<tr><td id="menuzag1" style="background:#00ff00"
  onmouseover="onmousemenu(event)" onmouseout="outmousemenu(event)">Меню
1
</td></tr>
<tbody id="tb1" style="display:none;" >
<tr><td id="e11" onmouseover="onelement(event)"
onmouseout="outelement(event)">
Один
</td></tr>
<tr><td id="e12" onmouseover="onelement(event)"
onmouseout="outelement(event)">
Два
</td></tr>
<tr><td id="e13" onmouseover="onelement(event)"
onmouseout="outelement(event)">
Три
</td></tr>
</tbody>
</table>

<table id="menu2" border=1
   style="position:absolute;top:100;left:130;width:80; z-index:5;"
   width=40 rules=none cellpadding=2 cellspacing=0
   onmouseout="document.getElementById('tb2').style.display='none'"
   onmouseover="document.getElementById('tb2').style.display='';
     document.getElementById('tb2').style.background='#e0e0e0'"
   onclick="menuclick(event)"
>
<tr><td id="menuzag2" style="background:#00ff00"
onmouseover="onmousemenu(event)" onmouseout="outmousemenu(event)">Меню 2
</td></tr>
<tbody id="tb2" style="display:none;" >
<tr><td id="e21" onmouseover="onelement(event)"
onmouseout="outelement(event)">
Первый
</td></tr>
<tr><td id="e22" onmouseover="onelement(event)"
onmouseout="outelement(event)">
Второй
</td></tr>
```

```
</tbody>
</table>

<table id="menu3" border=1
  style="position:absolute;top:100;left:210;width:80; z-index:5;"
  width=40 rules=none cellpadding=2 cellspacing=0
  onclick="menuclick(event)"
>
<tr><td id="menuzag3" style="background:#00ff00"
onmouseover="onmousemenu(event)" onmouseout="outmousemenu(event)">Меню 3
</td></tr>
</table>

<script>
function onelement(event){ // наезд на элемент
var e=event.srcElement;  // для IE
if (!e) e=event.target; // для других браузеров
xid=e.id
document.getElementById(xid).style.color='white'
document.getElementById(xid).style.background='#0000ff'
}
/*********************************************************************/
function outelement(event){ // уход с элемента
var e=event.srcElement;    // для IE
if (!e) e=event.target; // для других браузеров
xid=e.id
document.getElementById(xid).style.color='black'
document.getElementById(xid).style.background='#e0e0e0'
}
/*********************************************************************/
function onmousemenu(event){ // наезд на заголовок меню
var e=event.srcElement;    // для IE
if (!e) e=event.target; // для других браузеров
xid=e.id
document.getElementById(xid).style.background='#00dd00'
}
/*********************************************************************/
function outmousemenu(event){ // уход с заголовка меню
var e=event.srcElement;    // для IE
```

```
if (!e) e=event.target; // для других браузеров
xid=e.id
document.getElementById(xid).style.background='#00ff00'
}
/*************************************************************************/
function menuclick(event){ // щелчок на элементе меню
var e=event.srcElement;    // для IE
if (!e) e=event.target; // для других браузеров

alert("Элемент меню: "+e.id) // вы можете написать здесь свой код
/* Пример выбора первого элемента
if (e.id=="e11"){window.open("http://dunaevv1.narod.ru")}
*/
}
</script>
</html>
```

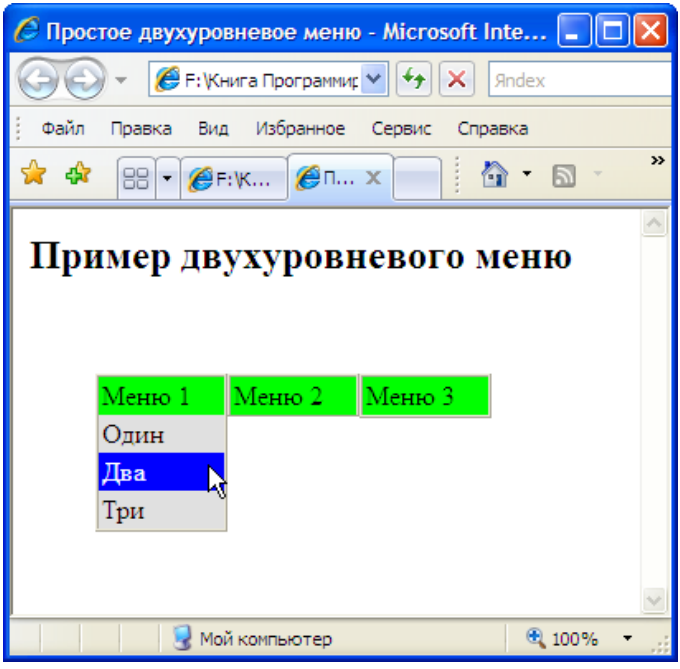

Рис. 4.11. Пример двухуровневого меню на основе таблиц

Аналогичным образом вы можете создать меню с бóльшим количеством опций, как в главном меню, так и в различных подменю. Обратите внимание, что единство организации двухуровневого меню обеспечивается лишь тем, что таблицы располагаются рядом друг с другом. Ничто не мешает вам позиционировать их иначе, чтобы на экране у вас появилось несколько одноуровневых вертикальных меню.

4.11. Перемещение элементов мышью

В Web-приложениях задачу перемещения видимых элементов документа нельзя назвать очень актуальной. В самом деле, разработчик Web-страницы не хотел бы, чтобы пользователь вверг его презентацию в хаос простыми манипуляциями мыши. Тем не менее, изучение этой задачи дает читателю нечто очень важное для понимания обработки событий, в частности, и Web-программирования вообще. Кроме того, при надлежащем дизайне, можно придумать проект, в котором технологии перемещения элементов окажутся не просто экзотикой, но и получат достаточное оправдание. Наглядный пример — компьютерные игры. Менее очевидное применение: на странице представлено много элементов, которые, "борясь" за пространство дисплея, в исходном состоянии частично перекрывают друг друга. Чтобы выбрать что-то конкретное для детального просмотра, пользователю нужно лишь перетащить элемент на передний план. Каков бы ни был замысел проекта такого рода, в основе его лежит техническая задача реализации перемещения элементов документа посредством мыши, которую мы и рассмотрим в данном разделе.

Очевидно, для перемещения элементов потребуется создать обработчики таких событий, как `onmousedown` (нажатие на кнопку мыши), `onmousemove` (перемещение указателя мыши) и `onmouseup` (освобождение ранее нажатой кнопки мыши). В самом деле, при перетаскивании элемента пользователь наводит на него указатель мыши, нажимает на левую кнопку и, удерживая ее, перемещает указатель, а в точке назначения отпускает кнопку. Однако не все элементы нужно и можно перемещать. Так, элементы `<INPUT>` и `<TEXTAREA>` (поле и область ввода данных) реагируют на события мыши по-своему, например, щелчок устанавливает курсор, и мы не можем воспользоваться событиями мыши для других целей, в частности, управления перемещением. Хотя задача перемещения и этих элементов вполне разрешима, мы здесь не будем ее рассматривать. С другой стороны, не все элементы на странице нужно перемещать. Другими словами, разработчик должен как-то указать, что позволено клиенту перемещать, а что нет. В противном случае хаос может разрушить проект.

В предлагаемом здесь решении задачи элементы документа, для которых возможно перемещение, должны иметь атрибут `class="dragelement"`. Атрибут `class` задумывался для связи элемента с каскадными таблицами стилей. В данной задаче мы не используем его по своему прямому назначению. Сейчас он играет роль метки для тех элементов, которые мы разрешаем перемещать мышью. Разумеется, значение этого атрибута выбрано произвольно и может быть изменено разработчиком по своему усмотрению.

Далее, перемещение инициируется нажатием на кнопку мыши при наведении ее указателя на элемент, т. е. событием `onmousedown`. Обработчик этого события — функция `dragstart()`, привязанная не к конкретному перемещаемому элементу, а к контейнерному элементу `<BODY>`. Следовательно, событие `onmousedown` может восприниматься любым элементом документа. Однако обработчик `dragstart()` анализирует, в каком именно элементе произошло это событие (в частности, имеет ли он атрибут `class="dragelement"`) и, если все необходимые условия выполнены, начинает работу обработчик перемещения мыши — функция `drag()`. Эта функция присваивает координатам элемента значения координат указателя мыши до тех пор, пока не будет освобождена кнопка мыши (событие `onmouseup`).

В листинге 4.36 в качестве примера приведен HTML-код и сценарий, обеспечивающий перемещение элементов, которые имеют атрибут `class="dragelement"`. В рассматриваемом примере перемещаемыми элементами являются два графических изображения и текст, заключенный в контейнер `<B>` (выделение полужирным). Внешний вид данного документа показан на рис. 4.12.

Листинг 4.36. Перемещение элементов мышью

```
<HTML>
<HEAD><TITLE>Перемещение мышью</TITLE></HEAD>
<BODY id="mybody"
 onmousedown="return dragstart(event)"
>
<IMG src="picture1.jpg" class="dragelement"
style="position:absolute;top:150;left:10"
>
<IMG src="picture2.jpg" class="dragelement"
style="position:absolute;top:100;left:300"
>
<B id="txt" class="dragelement">Перетащи меня</B>
<BR>
```

```
<INPUT type="text" class="dragelement" value="Привет">
</BODY>

<SCRIPT>
/* Глобальные переменные */
operation=false;
id_el=null;

window.onload=initdrag; // загрузив окно, выполняем initdrag()
/*********************************************************************/
function initdrag(){ // устанавливаем обработчики
document.body.onmousemove=drag;
document.body.onmouseup=dragend;
}
/*********************************************************************/
function dragstart(evt){    // начало перемещения элемента
operation=true; // операция перемещения началась
/* Получаем объект элемента, который нужно перетащить */
var e=evt.srcElement;    // для IE
if (!e) e=evt.target; // для других браузеров

id_el=e.id;    // ID перетаскиваемого элемента
if (!id_el) {    // если нет ID, то создаем его
 id_el="xid_"+parseInt((1+99999*Math.random()));
 e.id=id_el
/* Элементы input и textarea перетаскивать нельзя */
if(e.tagName.toLowerCase()=="input" ||
                  e.tagName.toLowerCase()=="textarea")
    return false;
}

if (document.getElementById(id_el)&&
       (document.getElementById(id_el).className=="dragelement")){
/* если нет позиционирования, то добавляем его: */
  if(!document.getElementById(id_el).style.position){
document.getElementById(id_el).style.position="relative"
  }
```

```
/* вид указателя мыши: */
  document.getElementById(id_el).style.cursor = 'move';
/* увеличиваем z-Index: */
  document.getElementById(id_el).style.zIndex++;
}
return false
}
/*****************************************************************/
function drag(event){    // перемещение элемента
if (flag&&document.getElementById(id_el)&&
       (document.getElementById(id_el).className=="dragelement")){
  var evt;
/* для межбраузерной совместимости: */
  if (event) {evt=event} else {evt=window.event};
/* изменяем координаты элемента */
  document.getElementById(id_el).style.top=evt.clientY-25;
  document.getElementById(id_el).style.left=evt.clientX-25;
}
return false
}
/*****************************************************************/
function dragend(){       // конец перемещения элемента
document.getElementById(id_el).style.cursor ="auto";
operation=false; // операция перемещения завершилась
id_el=null
}
</SCRIPT>
</HTML>
```

В функции `dragstart(evt)`, которой передается экземпляр объекта события, сначала определяется, в каком именно элементе документа произошло событие. В дальнейшем обращение к объекту данного элемента происходит по значению атрибута `id`. Однако это значение может быть не установлено в HTML-коде, т. е. в теге элемента отсутствует атрибут `id`. В таком случае свойство `id` устанавливается программно с помощью датчика случайных чисел (`random()`). Если имя тега элемента есть `input` или `textarea` (поле или область ввода), то работа функции `dragstart` завершается. В противном случае проверяется доступность объекта элемента и то, что у него есть атрибут `class="dragelement"` (соответствующее свойство объекта — `className`).

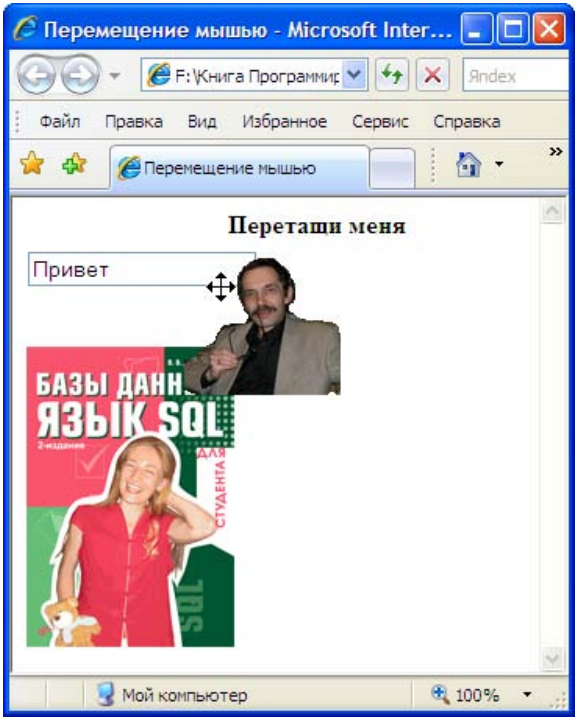

Рис. 4.12. Страница, сформированная кодом из листинга 4.36

Перемещать можно элементы, среди параметров стиля (`style`) которых есть `position`. Иначе говоря, если этот параметр не определен, то вы не можете обращаться к параметрам `top` и `left`, которые задают соответственно вертикальную и горизонтальную координаты позиционирования элемента. Если у элемента не оказалось предварительно заданного стилевого параметра `position`, то он создается программно. Далее устанавливается вид указателя мыши `"move"` (перекрестие), соответствующий операции перетаскивания, и увеличивается текущее значение стилевого параметра `zIndex`, чтобы перемещаемый элемент продвинулся ближе к переднему плану и мог расположиться поверх остальных.

Функция `drag(event)` обрабатывает событие `onmousemove` путем присваивания координатам элемента (стилевым параметрам `top` и `left`) значений координат указателя мыши относительно клиентской области браузера за вычетом 25 пикселов (эмпирически подобранное значение), чтобы указатель во время перемещения находился над элементом.

Функция `dragend()` обрабатывает событие `onmouseup`, означающее завершение операции перетаскивания. Она возвращает обычный вид указателя мыши

и присваивает переменной `operation` значение `false`. Переменная `operation`, принимающая логические значения, служит для увязки между собой трех обработчиков событий так, чтобы вместе они обеспечили выполнение операции перетаскивания элемента. С этой же целью применяется оператор `return`, который во всех местах кода возвращает `false`.

4.12. Динамическое изменение содержимого документа

Внешний вид элементов и содержимое частей документа в окне браузера можно модифицировать. Например, вы можете заменить одно графическое изображение на другое, поменять цвет и тип шрифта текста, а также изменить всю информацию, отображаемую в главном окне браузера или только в его части. Что при этом происходит? Ответить однозначно нельзя. В одних случаях осуществляется полная перезагрузка документа, а в других — частичная, т. е. подзагружается только то, что необходимо обновить в документе, а не весь документ с новыми вставками. Так например, вы хотите на странице своего сайта обновить только биржевую сводку или сведения о погоде, все остальное оставив по-старому. Это можно сделать путем формирования и загрузки в браузер клиента нового HTML-документа с информационными обновлениями, а можно лишь передать, а затем отобразить в рамках того же документа только новые сведения. Второй вариант предпочтительней с точки зрения экономии временных ресурсов.

Изменение содержимого называют динамическим, если оно происходит без перезагрузки всего документа в браузер. Для этого существуют несколько способов:

- изменение свойств элементов, заданных атрибутами тегов;
- изменение свойств объектов, ассоциированных с атрибутами соответствующих элементов, а также смена стилевых параметров;
- модификация HTML-кода, содержащегося в контейнерном теге данного элемента, с помощью свойства `innerHTML`;
- загрузка нового документа во фреймы, статические или плавающие;
- применение специальных объектов и технологии AJAX.

О загрузке документов во фреймы достаточно подробно рассказывалось в *разд. 4.4*. Поэтому здесь мы не будем рассматривать данный способ динамического изменения содержимого документа, сосредоточив свое внимание на остальных.

4.12.1. Изменение свойств, ассоциированных с атрибутами элементов, и свойств стиля

Изменение свойств объектов, ассоциированных с соответствующими HTML-элементами документа, уже неоднократно рассматривалось на примерах в предыдущих разделах. Как правило, свойствам HTML-элементов, заданным с помощью атрибутов, соответствуют аналогичные свойства объектов данных элементов. Например, атрибуту `ID` соответствует свойство `id`, атрибуту `NAME` — `name`, а `SRC` — `src`. Следует помнить, что свойства, соответствующие атрибутам элементов, в сценариях должны записываться строчными буквами, в то время как имена атрибутов в HTML-тегах могут быть в любом регистре. Стандартный способ доступа к свойствам, ассоциированным с атрибутами, обеспечивается выражениями вида:

```
document.getElementById('id_элемента').свойство;
document.all['id_элемента'].свойство;
```

Второе выражение не соответствует стандарту, но применимо в Internet Explorer, а также во многих других современных браузерах.

В следующем примере (листинг 4.37) сценарий изменяет первоначально заданное графическое изображение на другое, если пользователь навел на него указатель мыши и устанавливает прежнее изображение при уходе указателя.

Листинг 4.37

```
<HTML>
<IMG id=”myimg” src= "jpg pict1.jpg"
onmouseover="overimg()" onmouseout="outimg()">
<SCRIPT>
function overimg(){ // наведение указателя мыши
document.getElementById("myimg").src="pict2.jpg";
}
function outimg(){ // уход указателя мыши
document.getElementById("myimg").src="pict1.jpg";
}
</SCRIPT>
</HTML>
```

Если вы пишете сценарий, изменяющий значения атрибутов, в строке атрибута-события, то удобно использовать ключевое слово `this`, указывающее, что рассматриваемое в сценарии свойство относится к данному объекту. Так, приведенный в листинге 4.37 код можно переписать короче (листинг 4.38).

Листинг 4.38

```
<HTML>
<IMG src="pict1.gif"
 onmouseover="this.src='pict2.jpg'" onmouseout="this.src='pict1.jpg'">
</HTML>
```

При изменении адреса источника графического изображения документ, в котором оно расположено, в целом не перезагружается. Данное свойство элемента графического изображения может быть использовано не только для загрузки в браузер файлов с картинками. Подробнее об этом будет рассказано в *разд. 4.12.3*.

Элементы документа могут иметь стилевые параметры, т. е. параметры, заданные каскадными таблицами стилей *(см. разд. 4.5)*. Нередко стилевые параметры HTML-элементов задают в строке, которая является значением атрибута `style`. Доступ к таким параметрам обеспечивает выражение вида:

```
document.getElementById('id_элемента').style.свойство_стиля;
```

Здесь `свойство_стиля` — стилевой параметр, который в сценарии записывается не в синтаксисе CSS с использованием дефисов (например, `font-size`), а без дефисов и с заменой первой буквы перед дефисом на прописную (например, `fontSize`). Это общее правило, но для некоторых свойств возможны и исключения. Поэтому имя свойства, соответствующего параметру CSS, лучше уточнить в справочнике.

Многие параметры внешнего вида HTML-элементов, а также координаты их позиционирования, можно поменять программно, создавая эффекты динамического изменения параметров элемента. Примеры этого были рассмотрены в предыдущих разделах данной главы.

4.12.2. Предварительная загрузка изображений

С помощью сценария можно организовать предварительную загрузку изображений в кэш-память браузера, не отображая их на экране. Это особенно эффективно при начальной загрузке страницы. Пока изображения загружаются в память, оставаясь невидимыми, пользователь может рассматривать текстовую информацию, его не раздражает медленное появление графических элементов.

Для предварительной загрузки изображения следует создать его объект в памяти браузера. Такой объект несколько отличается от объекта изображения, создаваемого с помощью тега `<IMG>`. Как и все объекты, создаваемые сцена-

риями, объект изображения не выводится в окне браузера. Однако его наличие в коде загружаемого документа уже обеспечивает загрузку самого изображения. Чтобы создать в памяти объект изображения, необходимо в сценарии предусмотреть следующее выражение:

```
Myimg = new Image(ширина, высота).
```

Параметры функции-конструктора объекта определяют размеры изображения и должны соответствовать значениям атрибутов `WIDTH` и `HEIGHT` тега `<IMG>`, который служит для отображения предварительно загруженной картинки.

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

```
myimg.src="URL-адрес_изображения".
```

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

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

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

Теперь рассмотрим в качестве примера HTML-документ, в котором отображается список названий графических элементов и одно исходное изображение. Щелчок на элементе списка приводит к выводу требуемой картинки. Все графические элементы из этого списка предварительно загружаются в кэш-память и поэтому быстро отображаются при выборе. Список, как известно, создается с помощью контейнерного тега `<SELECT>`, содержащего теги `<OPTION>`. Соответствующий фрагмент HTML-кода мы сгенерируем с помощью сценария и запишем в текущий документ (листинг 4.39).

Листинг 4.39. Пример использования объекта `Image` для предварительной загрузки изображений

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

```
  var imgFile=new Array()   // массив имен графических файлов
  imgFile[0]="pict1.jpg"
  imgFile[1]="pict2.jpg"
  imgFile[2]="pict3.jpg"
  imgFile[3]="pict4.jpg"

  var imgName=new Array()   // массив названий картинок
  imgName[0]="Картинка1"
  imgName[1]="Картинка2"
  imgName[2]="Картинка3"
  imgName[3]="Картинка4"

  /* Создание объектов изображений и загрузка изображений в кэш */
  var imgObj=new Array()      // массив объектов изображений
  for(i=0; i < imgFile.length; i++){
   imgObj[i]=new Image(100, 200)  // создаем объект изображения
   imgObj[i].src=imgFile[i]    /* загрузка картинки в память
                     без отображения */
  }

  function imgshow(list){ // отображение картинки при выборе из списка
   var x=list.options[list.selectedIndex].value  // выбранный номер
                                                 // картинки
   document.getElementById['img0'].src=eval("imgObj["+ x + "].src")
  }

  /* Создание списка картинок */
  var clist="<SELECT onchange='imgshow(this)'>"
  for(i=0; i<imgFile.length; i++){
   clist+= "<OPTION VALUE=" + i + ">" + imgName[i]
  }
  clist+= "</SELECT>"
  document.write(clist)    // запись списка картинок в документ
 </SCRIPT>
 </HEAD>
 <! Исходная картинка>
 <IMG ID="img0" SRC="pict0.jpg" WIDTH=150 HEIGHT=100>
</HTML>
```

Обратите внимание, что для преобразования строки в настоящую ссылку на объект используется функция `eval()`. HTML-код, определяющий список картинок, сгенерированный сценарием, выглядит следующим образом:

```
<SELECT onchange='imgshow(this)'>
 <OPTION VALUE=0>Картинка1
 <OPTION VALUE=1>Картинка2
 <OPTION VALUE=2>Картинка3
 <OPTION VALUE=3>Картинка4
</SELECT>
```

4.12.3. Использование изображения для парольной защиты страницы

В данном разделе мы рассмотрим нетрадиционное использование элемента графического изображения — защиту доступа к Web-странице вашего сайта посредством пароля. Защита доступа к Web-странице обычно обеспечивается с посредством серверных сценариев. Однако эту же задачу можно решить и с помощью клиентского сценария, применив элемент графического изображения.

Решение задачи основано на том факте, что пароль задается как имя графического файла. При этом не важно, каково его содержимое, но желательно, чтобы изображение было возможно меньших размеров, например, 1×1 пиксел. Его атрибуту `src` следует присвоить пустое значение (`<IMG src="">`). В этом случае браузер отобразит на экране маленькую пиктограмму, указывающую на отсутствие указанного в `src` файла. Эта пиктограмма имеет различные размеры в зависимости от типа браузера, но большие чем 10×10 пикселов. Элемент этого графического изображения следует сделать невидимым на экране с помощью атрибута стиля `style="visibility:hidden"`.

При загрузке страницы с предложением ввести пароль в текстовое поле (элемент `<INPUT type="text">` или `<INPUT type="password">`) пользователь вводит набор символов, который присваивается атрибуту `src` элемента графического изображения. Если указанный таким образом файл существует, то ширина и высота изображения в документе станут равными одному пикселу. В противном случае они останутся прежними или же примут другие значения. Чтобы узнать пароль, необходимо угадать имя графического файла, а также довольно редко используемые размеры изображения (1×1 пиксел). Далее размеры графического изображения сравниваются с единицей. Если ширина и высота равны единице, то пароль верен, а в противном случае — нет.

Если введенный пользователем пароль корректен, то можно, например, загрузить некоторую страницу или еще что-то сделать.

Итак, чтобы реализовать парольную защиту, необходимо разместить в корневом каталоге вашего сайта графический файл с произвольным изображением размером 1×1 пиксел. При таких размерах лучше выбрать формат BMP. Имя этого файла является паролем. При вводе пароля расширение файла указывать не нужно, поскольку его можно добавить в сценарии.

Реализация рассмотренного способа представлена в листинге 4.40. В данном примере функция `validator()` выводит диалоговое окно с результатами проверки. Разумеется, вы можете вместо этого написать код загрузки той страницы, которая защищена данным паролем, например, `window.location="myspecialpage.htm"`. Обратите внимание, что пароль в явном виде нигде не упоминается в коде сценария.

Листинг 4.40. Парольная защита страницы

```
<HTML>
Введите пароль:<br>
<INPUT TYPE="text" ID="pw" VALUE="">
<BUTTON onclick="enterpw()">OK</BUTTON>
<BR>
<IMG SRC="" ID="ipw" STYLE="visibility:hidden">
<SCRIPT>
function enterpw(){ // обработка введенного слова и проверка пароля
document.getElementById("ipw").src =
document.getElementById("pw").value+".bmp";
setTimeout("validator()",1000); // задержка для загрузки изображения
}


function validator(){ // проверка пароля
if ((document.getElementById("ipw").width==1)&&
(document.getElementById("ipw").height==1)){
 alert("Пароль верен");
} else {
 alert("Пароль НЕ верен");
}
}
</SCRIPT>
</HTML>
```

4.12.4. Применение свойства *innerHTML*

Объекты, соответствующие элементам документа, заданным HTML-тегами, имеют очень полезное свойство `innerHTML`, значением которого является строка, содержащая HTML-код, заключенный в данном элементе.

Пусть, например, задан следующий элемент формы:

```
<FORM id="myform">
<INPUT id="myinp" type="text">
<INPUT id="mysubm" type="submit" value="Отправить">
</FORM>
```

Тогда `document.getElementById("myform").innerHTML` содержит в качестве значения строку:

```
<INPUT type="text"><INPUT type="submit" value="OK">
```

Тег `<FORM>` является контейнером (есть закрывающий тег `</FORM>`) и для него свойство `innerHTML` возвращает весь внутренний HTML-код. Для неконтейнерных тегов, таких как `<IMG>`, `innerHTML` возвращает пустую строку. Исключением является тег ссылки, для которого `innerHTML` содержит URL-адрес (значение атрибута `href`).

В листинге 4.41 приведен некоторый HTML-код и сценарий, который выводит пронумерованный список значений `innerHTML` для всех элементов данного документа, за исключением элемента `<HTML>`. Результат работы данного сценария в браузере InternetExplorer показан на рис. 4.13. В других браузерах он может быть несколько иным (рис. 4.14).

Листинг 4.41. Значения свойства `innerHTML` для всех элементов документа

```
<HTML>
<HEAD><TITLE>Свойство innerHTML</TITLE></HEAD>
<A HREF = "http://www.rambler.ru">Rumbler</A><BR>
<IMG src="picture1.png">
<FORM id="myform">
<INPUT type="text">
<INPUT type="submit" value="Отправить">
</FORM>
<SCRIPT>
var xstr="";
l=document.all.length;
for (i=1; i<l;i++){
```

```
elem=document.all[i];
innerelem=elem.innerHTML;
xstr+=i+". "+elem+" "+elem.tagName+":\n"+innerelem+"\n"
}
alert(xstr)
</SCRIPT>
</HTML>
```

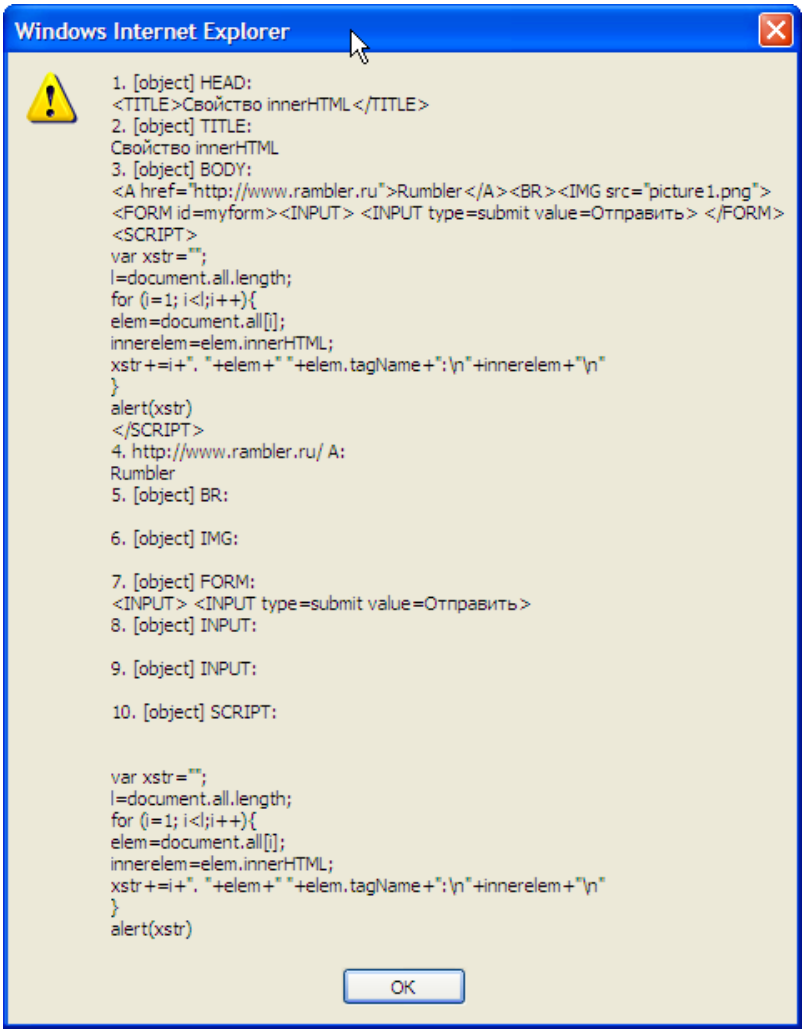

Рис. 4.13. Результат работы сценария из листинга 4.41 в Internet Explorer 7.0

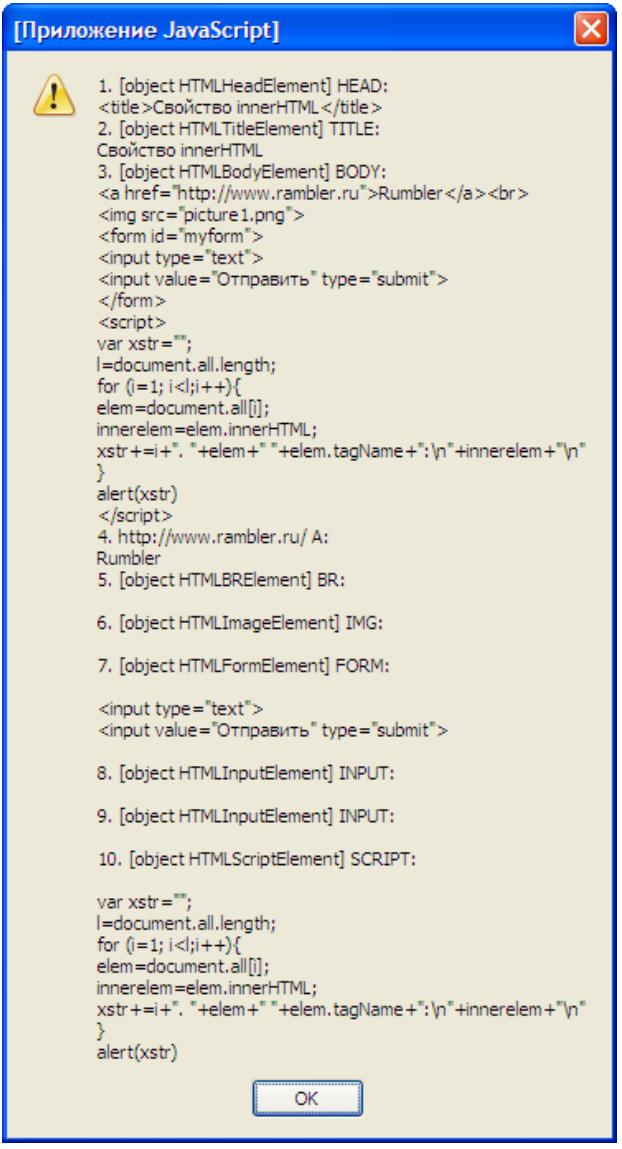

Рис. 4.14. Результат работы сценария из листинга 4.41 в Mozilla Firefox 2.0

В *разд. 4.9* уже рассматривался пример, в котором свойство `innerHTML` обеспечивало доступ к содержимому ячеек таблицы. Но наиболее ценно то, что свойство `innerHTML` можно изменять. Как только вы присвоите ему новое значение, браузер сразу же соответствующим образом модифицирует документ. Другими словами, с помощью `innerHTML` можно заменять одни элементы

документа на другие. Например, вы можете заменить абзац (элемент <P>) на картинку (листинг 4.42).

Листинг 4.42. Замена элемента с помощью свойства innerHTML

```
<HTML>
<HEAD><TITLE>Свойство innerHTML</TITLE></HEAD>
<H1>Изменение элементов</H1>
<P id="mytext" onclick="toimage()">
Если щелкнуть на этом тексте,
то он будет заменен графическим изображением
</P>
<SCRIPT>
function toimage(){
document.getElementById("mytext").innerHTML=
"<IMG src='picture1.png'>"
}
</SCRIPT>
</HTML>
```

Другой пример использования свойства innerHTML приведен в *разд. 4.12.5* (см. листинг 4.43).

Браузер Internet Explorer кроме innerHTML поддерживает еще три аналогичные свойства: outerHTML, innerText и outerText.

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

Значение свойства innerText — все текстовое содержимое, заключенное между открывающим и закрывающим тегами элемента. Если в эту строку входят HTML-теги, то они игнорируются. Таким образом, innerText следует понимать как весь внутренний текст, содержащийся в контейнере.

В отличие от innerText, свойство outerText включает в себя данные, содержащиеся и в открывающем, и закрывающем тегах элемента. Таким образом, outerText следует понимать как весь текст, содержащийся в контейнере, включая и его внешние теги.

4.12.5. Применение технологии AJAX

Обычная технология взаимодействия с Web-сервером состоит в том, что данные клиента (например, введенные в форму) передаются на сервер и обрабатываются указанным при передаче серверным сценарием. Для написания

серверных сценариев существует несколько языков, например, Perl, VBSript, JScript, PHP. Язык PHP сейчас наиболее популярен. Серверный сценарий может сформировать ответ браузеру, представляющий собой документ, который загружается либо в текущее окно браузера, либо в отдельное новое окно, либо во фрейм (статический или плавающий). Иначе говоря, окна и фреймы — типичные приемники информации, получаемой от сервера, не требующие перезагрузки всего исходного документа, в котором инициировался запрос к серверу. В соответствии с обычной технологией запрос к серверу какого-либо документа в случае успеха завершается его загрузкой в заранее предусмотренное для этого окно или фрейм. При этом вы не можете сначала получить ответ сервера, а затем с помощью клиентского сценария распорядиться, где и как его разместить для обозрения. Например, нельзя разместить полученные от сервера данные в каком-то элементе документа, скажем, в абзаце (`<P>`) или разделе (`<DIV>`). Вместе с тем, во многих современных браузерах имеется специальный объект `XMLHttpRequest` (а в Internet Explorer 5+ — `XMLHTTP`), который позволяет передать на сервер данные и получить от него ответную информацию без автоматического отображения в браузере, что и является главной особенностью, стимулирующую его применение. Используя этот объект, вы можете в клиентском сценарии обработать полученную с сервера информацию, а также задать место и способ ее отображения в текущем документе. При этом информационный обмен с сервером может происходить асинхронно, т. е. независимо по времени от других вычислительных процессов. Например, при приеме данных с сервера остальные процедуры клиентского сценария не дожидаются его завершения, а начинают и продолжают свою работу.

Систематическое применение объекта `XMLHttpRequest` в клиентских сценариях для организации информационного обмена с сервером получило название AJAX (Асинхронный JavaScript и XML). Само название возникло в 2005 г., хотя технология AJAX сформировалась по существу в 1998 г., когда в Internet Explorer 5.0 появились новые элементы управления ActiveX для работы с табличными данными, а затем и собственно объект `XMLHttpRequest`. Применение этого объекта сулило значительное ускорение загрузки данных любого типа в текущий документ, а потому производители других браузеров (например Mozilla Firefox 1+, Opera 8+, Safari…) включили его поддержку.

В браузере Internet Explorer версий, более ранних, чем 7.0, объект `XMLHTTP` (аналог `HttpRequest`) активизируется как элемент управления ActiveX одним из следующих двух способов:

- `req = new ActiveXObject('Microsoft.XMLHTTP')` — для самых ранних версий;
- `req = new ActiveXObject('Msxml2.XMLHTTP')` — для более поздних версий, но меньших, чем 7.0.

Здесь с помощью оператора `new` создается экземпляр объекта `XMLHttpRequest`, ссылка на который присваивается переменной `req`. Эту ссылку затем можно использовать для доступа к свойствам и методам данного объекта.

В других браузерах, а также Internet Explorer 7.*x* экземпляр объекта `XMLHttpRequest` создается более простым выражением:

```
req = new XMLHttpRequest().
```

Для обеспечения межбраузерной совместимости в сценарии следует предусмотреть все возможные варианты активизации объекта `XMLHttpRequest`. Общий порядок активизации `XMLHttpRequest` выглядит следующим образом.

1. Создание экземпляра объекта `XMLHttpRequest`.
2. Установка обработчиков событий.
3. Открытие соединения с сервером.
4. Отправка запроса на сервер.

В листинге 4.43 приведен пример использования объекта `XMLHttpRequest`. При загрузке в браузер данного кода первоначально отображается текст и кнопка с надписью "Получить данные с сервера". Щелчок на этой кнопке приводит к взаимодействию с сервером посредством указанного объекта. Говоря точнее, выполняется следующая последовательность действий.

1. Вызывается пользовательская функция `Load()`, которая создает экземпляр объекта `XMLHttpRequest`. Для обеспечения межбраузерной совместимости с помощью оператора `try…catch` (см. *разд. 3.14*) обработки исключительных ситуаций выбирается способ создания экземпляра `XMLHttpRequest`, подходящий для браузера того или иного типа или версии.
2. Затем функция `Load()` устанавливает обработчик `receive()` события `onreadystatechange` (изменение состояния), открывает методом `open()` соединение с серверным сценарием, расположенным в файле myprogram.php, и отправляет запрос методом `send()`.
3. В случае успеха данной операции сервер должен ответить браузеру. Методу `open()` в данном примере передаются три параметра: метод передачи (в данном примере `GET`), URL-адрес файла с серверным сценарием и значение `true`, задающее асинхронный режим взаимодействия. Впрочем, `true` является значением по умолчанию. Методу `send()` можно передать в качестве параметра какие-то данные для сервера, если задан метод передачи `POST`. Для метода `GET` следует указать пустую строку или `null`.
4. При получении от сервера ответа возникает событие `onreadystatechange` и вызывается ранее назначенный обработчик — функция `receive()`, которая анализирует коды состояния объекта `XMLHttpRequest`, т. е. проверяет, успешно ли завершен запрос к серверу.

5. Если загрузка данных с сервера завершена (код 4) и не было ошибок (код 200), то функция `receive()` заменяет содержимое элемента `<DIV>` полученными от сервера данными, хранящимися в свойстве `responseText`. Это делается посредством свойства `innerHTML` *(см. разд. 4.12.4)*.

Листинг 4.43. Пример применения технологии AJAX

```
<HTML>
<HEAD><TITLE>Применение AJAX</TITLE></HEAD>
<SCRIPT>
var req=false;
function Load() {
  try { // определить метод поддержки
    req=new ActiveXObject('Msxml2.XMLHTTP');// для IE
  } catch (e) {
    try {
      req=new ActiveXObject('Microsoft.XMLHTTP'); // для IE
    } catch (e) {
      if(window.XMLHttpRequest){ // для других браузеров
        req=new XMLHttpRequest();// экземпляр объекта
      }
    }
  }
  if (req){ // если объект XMLHTTP поддерживается
    /* назначаем обработчик событию готовности объекта: */
    req.onreadystatechange = receive;
    /* открываем объект для взаимодействия:
     передача методом GET;
     серверный сценарий в файле myprogram.php;
     режим — асинхронный:
    */
    req.open("GET","..//AJAX/myprogram.php",true);
   /* посылаем запрос серверу: */
    req.send(null);
  }else {
    alert("Объект XMLHTTP не поддерживается данным браузером")
  }
}
```

```
function receive() { // получение данных от сервера
  if (req.readyState == 4){// если запрос завершен
    if (req.status == 200) { // если запрос завершен без ошибок
     /* Отображаем ответ сервера в HTML-элементе документа */
    document.getElementById('content').innerHTML =(req.responseText);
    } else {
      alert("Ошибка "+ req.status+":\n" + req.statustext);
    }
  }
}
</SCRIPT>
Щелчок на кнопке загрузит в данный документ ответ сервера.
<INPUT type="button" value="Получить данные с сервера" onclick="Load()">
<BR>
Данные, принятые от сервера:
<BR>
<DIV id="content" style="color:#ff0000"></DIV>
</HTML>
```

Опробовать данный пример можно на локальном компьютере с установленными Web-сервером (например, Microsoft IIS или Appache) и модулем PHP. Как это сделать, рассказывается в *приложении 2*. Для этого в домашнем каталоге Web-сервера создайте папку, например, AJAX и разместите в ней файл с кодом из листинга 4.43 (пусть это будет ajax.htm), а также файл myprogram.php с серверным сценарием на языке PHP. Для первого эксперимента серверный сценарий можно сделать очень простым:

```
<?php
echo 'В первых строках <i>своего письма</i> передаю всем <b>привет</b>';
?>
```

Данный сценарий возвращает браузеру строку текста, которая должна быть отображена в документе ajax.htm. Чтобы избежать проблем с кириллицей, файл myprogram.php с серверным сценарием сохраните в кодировке UTF-8. Язык PHP и программирование на нем серверных сценариев мы рассмотрим в следующих главах.

Если в адресную строку браузера ввести localhost/AJAX/ajax.htm, то в окне браузера должны появиться текст и кнопка. Щелчок на этой кнопке отобразит в нижней части документа строку текста, переданную сервером (рис. 4.15).

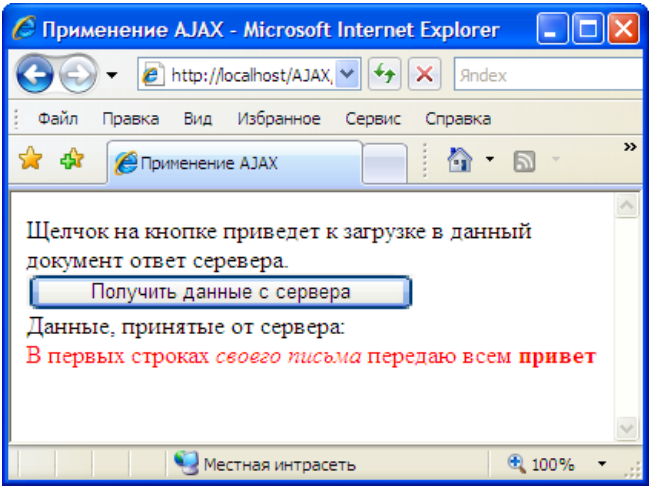

Рис. 4.15. Результат взаимодействия с сервером сценария из листинга 4.43

Аналогично можно передавать клиенту данные других типов, например, графические изображения, звук, видео и даже сценарии на языке JavaScript. Передача сценария для выполнения браузеру позволяет разгрузить сервер. В следующем примере (листинг 4.44) при щелчке на кнопке в браузер загружается код сценария, который затем выполняется с помощью функции `eval()`. Для эксперимента вы можете написать следующий серверный сценарий на языке PHP:

```
<?php
echo 'alert("Привет!"); y=2+3; alert(y)';
?>
```

Данный сценарий возвращает браузеру текстовую строку, содержащую код на языке JavaScript. Код JavaScript в данном примере выводит диалоговое окно с сообщением "Привет!", присваивает переменной значение 5 и выводит диалоговое окно с этим значением.

Листинг 4.44. Прием и выполнение клиентского сценария, сформированного на сервере

```
<HTML>
<HEAD><TITLE>Применение AJAX</TITLE></HEAD>
<SCRIPT>
var req=false;
function Load() {
  try { // определить метод поддержки
```

```
    req=new ActiveXObject('Msxml2.XMLHTTP'); // для IE
  } catch (e) {
    try {
      req=new ActiveXObject('Microsoft.XMLHTTP'); // для IE
    } catch (e) {
      if(window.XMLHttpRequest){ // для других браузеров
        req=new XMLHttpRequest();// экземпляр объекта
      }
    }
  }
  if (req){ // если объект XMLHTTP поддерживается
    /* назначаем обработчик событию готовности объекта: */
    req.onreadystatechange = receive;
    /* открываем объект для взаимодействия:
     передача методом GET;
     серверный сценарий в файле myprogram.php;
     режим — асинхронный:
    */
    req.open("GET", '..//AJAX/myprogram.php',true);
   /* посылаем запрос серверу: */
    req.send(null);
  }else {
    alert("Объект XMLHTTP не поддерживается данным браузером")
  }
}
function receive() { // получение данных от сервера
  if (req.readyState == 4){// если запрос завершен
    if (req.status == 200) { // если запрос завершен без ошибок
     /* Отображаем ответ сервера в HTML-элементе документа */
    eval(req.responseText); // выполнение принятого сценария
    } else {
      alert("Ошибка "+ req.status+":\n" + req.statustext);
    }
  }
}
</SCRIPT>
Щелчок на кнопке приведет к загрузке и выполнению сценария
ответ сервера.
<INPUT type="button" value="Получить данные с сервера" onclick="Load()">
</HTML>
```

Для создания настоящих Web-приложений на основе технологии AJAX следует подробнее ознакомиться со свойствами и методами `XMLHttpRequest` (табл. 4.7).

Таблица 4.7. *Свойства и методы объекта XMLHttpRequest*

Свойство или метод	Описание
`readyState`	Состояние запроса — целое число: 0 (не инициализирован), 1 (загрузка), 2 (получены заголовки ответа), 3 (получена часть содержимого ответа), 4 (запрос завершен)
`onreadystatechange`	Свойство, значением которого является имя функции, вызываемой при изменении значения свойства `readyState` (состояния запроса)
`status`	http-код состояния, возвращаемый сервером (например, "200")
`statusText`	Полная строка http-состояния, возвращенная сервером (например, "200OK")
`responseText`	Полный ответ сервера в виде строки
`responseXML`	Объект `document`, представляющий собой ответ сервера, рассмотренный как XML-документ
`abort()`	Отменяет асинхронный запрос
`getAllResponseHeaders()`	Возвращает строку, содержащую все http-заголовки, имеющиеся в ответе сервера. Каждый http-заголовок представляется парой *имя/значение,* разделителем служит двоеточие, а строки заголовка разделяются парой символов *возврат каретки/переход на новую строку*
`getResponseHeader` (*имя_заголовка*)	Возвращает строку, соответствующую указанному параметру `имя_заголовка` возвращенного сервером ответа (например, req.getResponseHeader("Set-cookie"))
`open`(*метод, url-адрес, асинхронный, login, password*)	Инициализирует (параметризует) запрос серверу. Параметр *метод* указывает метод передачи (например, GET или POST). Параметр `url-адрес` задает адрес, по которому будет направлен запрос. Последние три параметра не обязательны. Параметр `асинхронный` указывает режим обмена и принимает логические значения (`true` соответствует асинхронному режиму и установлено по умолчанию). Параметры `login` и `password` используются тогда, когда для указанного URL-адреса требуется http-аутентификация. Если для такого URL-адреса данные параметры не указаны, то пользователю будет предложено ввести их

Таблица 4.7 *(окончание)*

Свойство или метод	Описание
`setRequestHeader(`*имя*`,` *значение*`)`	Добавляет http-заголовок, заданный указанными параметрами `имя` и `значение`
`send(`*содержимое*`)`	Передает серверу запрос, параметры которого заданы методом `open()`. Параметр `содержимое` представляет собой строку вида `"имя_поля1=значение&имя_поля2=значение…"` для передачи запроса методом `POST`; пустую строку или `null` при передаче методом `GET`

Как уже отмечалось, передачу каких-до данных серверу можно осуществить методом `POST`. Однако применение метода `POST` имеет некоторое отличие по сравнению с методом `GET`. Оно состоит в том, что необходимо установить тип содержимого запроса с помощью метода `setRequestHeader()`.

В листинге 4.45 приведен код сценария, который передает на сервер данные методом `POST` и отображает полученный ответ. В приведенном примере передаются две переменные (`name` и `pw`), представленные в виде строки `"name=Вася&pw=1234567"`. Обратите внимание на использование метода `setRequestHeader("Content-Type","application/x-www-form-urlencoded")`.

Листинг 4.45. Передача данных на сервер методом `POST`

```
<HTML>
<HEAD><TITLE>Передача методом POST</TITLE></HEAD>
<SCRIPT>
var req=false;
try { // определить метод поддержки
   req=new ActiveXObject('Msxml2.XMLHTTP');// для для IE
  } catch (e) {
    try {
      req=new ActiveXObject('Microsoft.XMLHTTP'); // для IE
    } catch (e) {
      if(window.XMLHttpRequest){ // для других браузеров
        req=new XMLHttpRequest();// экземпляр объекта
     }
   }
 }
 if (req){ // если объект XMLHTTP поддерживается
```

```
 /* назначаем обработчик событию готовности объекта: */
   req.onreadystatechange = receive;
  /* открываем объект для взаимодействия:
   передача методом POST;
   серверный сценарий в файле myprogram.php;
   режим — асинхронный:
  */
   req.open("POST", '..//AJAX/myprogram.php',true);
  /* устанавливаем тип данных */
  req.setRequestHeader("Content-Type","application/x-www-form-
urlencoded");
  /* данные, передаваемые на сервер: */
  var data="name=Вася&pw=1234567";
  req.send(data); // посылаем запрос серверу
 }else {
    alert("Объект XMLHTTP не поддерживается данным браузером")
 }
}
function receive() { // получение данных от сервера
  if (req.readyState == 4){// если запрос завершен
    if (req.status == 200) { // если запрос завершен без ошибок
     /* Отображаем ответ сервера в HTML-элементе документа */
    document.getElementById("content").innerHTML=req.responseText;
    } else {
      alert("Ошибка "+ req.status+":\n" + req.statustext);
    }
  }
}
</SCRIPT>
Данные, принятые от сервера:
<BR>
<DIV id="content"></DIV>
</HTML>
```

Чтобы проверить работу этого примера, на стороне сервера (в файле myprogram.php) можно использовать следующий тестовый сценарий:

```
<?php
$x=$_POST['name']."<br>".$_POST['pw'];
echo $x;
?>
```

Данный сценарий просто возвращает браузеру значения переменных `name` и `pw`, переданные серверу методом `POST` и хранящиеся на сервере в массиве `$_POST`.

Термин "AJAX" в последнее время часто употребляется для обозначения современной технологии Web-программирования и в связи с новой идеологией, называемой Web 2.0. Как бы то ни было, новая технология есть не что иное, как применение уже известных и, быть может, вновь созданных технических средств для достижения каких-то целей. Технология AJAX в настоящее время — это набор приемов, обеспечивающих динамическое обновление содержимого Web-страниц без полной их перезагрузки, а также организацию взаимодействия с сервером, при которой можно рационально распределить вычислительные ресурсы между клиентом и сервером. Для реализации этой технологии имеются как традиционные (фреймы), так и специализированные (объект `XMLHttpRequest`) технические средства. Именно широкое применение в настоящее время уже давно поддерживаемого браузерами объекта `XMLHttpRequest` обусловило появление модного ныне термина "AJAX". Как видим, все очень просто: средства для реализации AJAX существовали раньше того, что теперь называется "AJAX". Нередко бывает, что необходимое обеспечение возникает раньше, чем оно будет действительно востребовано и практически освоено. Вместе с тем, не следует забывать, что в истоке почти всех великих технологических подвижек всегда стояло "слово". Похоже, настало время для AJAX. Однако это еще не означает, что данную технологию следует применять всегда и везде. Ведь появление автомобилей не уничтожило велосипеды, а развитие авиации не снизило интерес к автомобилям.

Применение AJAX, кроме указанных ранее преимуществ, имеет и недостатки. Во-первых, документы, сформированные динамически посредством сценария, оказываются недоступными для поисковых систем. Во-вторых, такие страницы не регистрируются браузером в журнале посещенных, а потому для возврата вы не можете воспользоваться кнопкой **Назад**. Имеются и другие эффекты, вызывающие нежелательные осложнения.

4.13. Распознавание типа браузера

Вообще говоря, сценарий на языке JavaScript не инвариантен относительно браузера, т. е. некоторые языковые конструкции функционируют в одном интернет-обозревателе так, в другом — иначе, а то и совсем не работают. В настоящее время все сценарии, корректно выполняющиеся браузерами на базе Mozilla и Opera, почти наверняка будут без ошибок выполняться и в Internet Explorer. Однако обратное утверждение неверно. Разработчик сценариев может, конечно, ограничиться лишь теми средствами, которые

инвариантны относительно наиболее популярных браузеров. Иначе говоря, он может придерживаться принципа аскетизма в своих разработках. Другой подход состоит в том, чтобы для каждого из наиболее популярных браузеров создать свою версию сценария. В этой книге учебные примеры разрабатывались так, чтобы они выполнялись в Internet Explorer, Mozilla Firefox и Opera, т. е. в соответствии с первым подходом. Для реализации второго варианта необходимо прежде всего выяснить, с каким браузером имеет дело данный сценарий.

Для выявления свойств браузера клиента служит объект `navigator`, рассмотренный в *разд. 4.1.6*. Чтобы узнать название интернет-обозревателя, можно воспользоваться его свойствами `appName` и `userAgent`, а номер его версии содержится в свойстве `appVersion`. Значения этих свойств формируются браузером и не имеют стандартной структуры. Некоторые обозреватели (в частности, Opera) могут подменять значение `appName` на соответствующее другому браузеру. Например, в Opera можно установить значение "Microsoft Internet Explorer". Это делается для того, чтобы преодолеть отказ в доступе к сайту, который разработан исключительно для Internet Explorer. В обозревателе Mozilla Firefox свойство `appName` имеет значение "Netscape". Так что не стоит полностью доверять значению `appName` при распознавании типа браузера. Более надежная информация содержится в `userAgent`, но ее необходимо проанализировать. Так например, значения `userAgent` в различных браузерах для Windows XP выглядят следующим образом:

- в Internet Explorer 7.0:

  Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; InfoPath.2; .NET CLR 2.0.50727);

- в Mozilla Firefox 2.0:

  Mozilla/5.0 (Windows; U; Windows NT 5.1; ru; rv:1.8.1.12) Gecko/20080201 Firefox/2.0.0.12;

- в Опера 9.25 возможны несколько вариантов:
  - Opera/9.25 (Windows NT 5.1; U; ru);
  - Mozilla/5.0 (Windows NT 5.1; U; ru; rv:1.8.0) Gecko/20060728 Firefox/1.5.0 Opera 9.25;
  - Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; ru) Opera 9.25.

Нетрудно заметить что слово "Mozilla" присутствует в большинстве вариантов строкового значения `userAgent` и, следовательно, не может идентифицировать браузер. Слово "Opera" однозначно указывает на тип браузера. В листинге 4.46 приведен код, с помощью которого можно с достаточно большой вероятностью распознать основные браузеры.

Листинг 4.46. Распознавание типа браузера

```
<HTML><HEAD><TITLE>Тип браузера</TITLE></HEAD>
<SCRIPT>
var userAgent = navigator.userAgent;
var msg = "Браузер на базе ";
if (userAgent.indexOf("Opera")!=-1)
  document.write(msg+"Opera");
else if(userAgent.indexOf("Gecko")!=-1)
     document.write(msg+"Mozilla");
   else if (userAgent.indexOf("MSIE")!=-1)
        document.write(msg+"Internet Explorer");
     else if (userAgent.indexOf("Mozilla")!=-1)
           document.write(msg+"Netscape (старый)");
        else
           document.write("Неизвестный браузер");

</SCRIPT>
</HTML>
```

Если для каждого из распознанных браузеров у вас имеется свой вариант документа, то вместо вывода сообщения можно переадресовать пользователя на соответствующую страницу: `window.location=`*`URL-адрес`*.

Глава 5

Основы языка PHP

PHP — это язык программирования, на котором пишутся сценарии (скрипты) различного назначения. Однако этот язык наиболее известен как средство создания серверных сценариев, выполняющихся не браузером клиента, а Web-сервером — специальным программным обеспечением, установленным, как правило, на удаленном компьютере. Из предыдущих глав вы уже знаете, что JavaScript — основной язык создания клиентских сценариев, т. е. программ, которые выполняет интерпретатор на локальном компьютере, например, Web-браузер. Надеюсь понятно, что жизнь в Интернете протекает во взаимодействии сервера и клиента. На обеих сторонах этой связи могут существовать сценарии, которые организуют ее правила на высшем уровне, т. е. уровне пользователя страниц сайта.

PHP был задуман как язык для программирования серверных сценариев, т. е. выполняемых средствами Web-сервера. Первая версия этого языка была разработана Расмусом Лердофом для поддержки Web-узлов (сайтов). Как обычно это бывает, исходный вариант, нацеленный на весьма скромные задачи, назывался просто — PHP (Personal Home Page — персональная домашняя страница). Но очень скоро PHP превратился в мощный объектно-ориентированный язык программирования, пригодный не только для Web-программирования. Теперь его называют Hypertext Processing (обработка гипертекста), хотя и новое название не отражает суть данного языка.

PHP, как и JavaScript, интерпретируемый язык. Немаловажную роль в завоевании его популярности сыграло то обстоятельство, что исходный код интерпретатора открытый. В разработке PHP принимает участие международное сообщество программистов, а лучше сказать — программисты всего мира, которые хотят внести свою лепту в улучшение и расширение исходного кода. Данное обстоятельство обеспечивает удивительно быстрое развитие PHP. Интерпретатор и ресурсы PHP предоставляются бесплатно. Их можно получить на сайте **www.php.net**.

Между языками PHP и JavaScript много общего, но имеются и существенные различия. Читателям, знакомым с JavaScript, следует в первую очередь обратить внимание на особенности определения переменных, задания строковых значений, а также на приведение типов данных в операторах языка PHP, поскольку именно здесь разница между PHP и JavaScript наиболее велика.

Поскольку PHP служит для создания серверных сценариев, т. е. программ, которые выполняются Web-сервером, то при изучении этого языка эксперименты разумнее провести сначала на локальном компьютере или в пределах локальной сети, а не в Интернете. Поэтому для выполнения PHP-сценариев прежде всего необходимо иметь Web-сервер, настроенный на работу с PHP, а также сам модуль PHP. Об установке Web-сервера Internet Information Services (IIS) и модуля PHP под Windows рассказывается в *приложении 2*.

Если вы установили и настроили Web-сервер (например, на своем локальном компьютере), то сможете изучать язык PHP, выполняя предлагаемые примеры, а также собственные их модификации или свои авторские экспериментальные скрипты. Сценарии решения более полезных для практики задач мы рассмотрим в *главе 6*.

5.1. Предварительные сведения

5.1.1. Где писать сценарии на PHP

Серверные сценарии на языке PHP сохраняются в обычных текстовых файлах с расширением php. Содержимое php-файла может быть каким угодно: обычный текст, HTML-теги, а также код на языке PHP, заключенный между дескрипторами `<?php` и `?>`, который может занимать одну и более строк. Каждое выражение на языке PHP должно обязательно завершаться точкой с запятой. В одном и том же php-файле может располагаться несколько фрагментов (секций) PHP-кода, перемежающихся блоками обычного текста и/или HTML-кода. Тогда каждый фрагмент PHP-кода должен окаймляться дескрипторами `<?php` и `?>`. Содержимое этих дескрипторов будет выполнено PHP-интерпретатором на стороне сервера, а все, находящееся за их пределами, будет без изменений передано сервером браузеру клиента.

Рассмотрим порядок создания, запуска и редактирования сценариев на языке PHP. Для этого выполните следующие действия:

1. Откройте текстовый редактор, например, блокнот Windows и введите следующий текст:

```
Привет от <b>сервера</b>.
<?php
```

```
    echo "Это тест PHP-сценария. <br>";
    $x=2*5;
    echo "Пример вычисления: 2*5 = $x";
?>
```

2. Сохраните введенный текст в обычном текстовом файле с расширением php (например, proba.php) в домашнем каталоге Web-сервера (для сервера IIS таким каталогом является C:\Inetpub\wwwroot).
3. Откройте Web-браузер, например, Internet Explorer, и в адресной строке введите http://localhost/proba.php (префикс http:// можно не указывать). Вместо localhost можно ввести сетевое имя вашего компьютера либо его десятичный IP-адрес. Если Web-сервер установлен на другом компьютере в локальной сети, к которой вы подключены, то следует использовать сетевое имя или IP-адрес того компьютера, а не вашего. В результате в клиентской области браузера должен отобразиться результат интерпретации файла proba.php.

В приведенном коде первая строка файла proba.php представляет собой текст, содержащий HTML-тег `<b>` выделения полужирным шрифтом. Далее следует вставка PHP-сценария (скрипта), который посредством оператора `echo` выводит сообщения и выполняет арифметическое действие. Сейчас не важно понимать в точности выражения данного фрагмента кода. Скоро вы все узнаете. Обратите внимание на то, как следует корректировать сценарий.

Внимание!

Если вам потребуется изменить содержимое текущего php-файла (например, proba.php), то не следует выполнять команду меню браузера **Вид | Просмотр HTML-кода**, поскольку в этом случае вы увидите не содержимое исходного файла, а результат его интерпретации. Сценария на языке PHP там уже нет. В рассматриваемом примере в браузер будет загружен такой документ:

```
Привет от <b>сервера</b>.
Это тест PHP-сценария.<br>
Пример вычисления: 2*5 = 10
```

Иначе говоря, помните, что содержимое исходного php-файла, хранящегося на сервере, может существенно отличаться от того, что загружено в браузер. Поэтому для коррекции открывайте в текстовом редакторе именно исходный файл. Чтобы изменения вступили в силу, сохраните файл под тем же именем и в том же месте, а затем обновите содержимое окна браузера (перезагрузите страницу).

Примечание

Ранее в данном разделе отмечалось, что все, находящееся вне дескрипторов `<?php` и `?>`, а также текст, указанный в операторе `echo` (например, обычный текст или HTML-код), отправляется без изменений браузеру клиента. По умолчанию

информация, отправляемая браузеру, отображается в главном окне или фрейме, в зависимости от настроек отображения текущего документа. Однако текущий документ клиента может инициировать специальный объект `XMLHttpRequest`, который особым образом обеспечивает связь с сервером. Если в сценарии текущего документа на стороне клиента используется объект `XMLHttpRequest`, то весь вывод серверной страницы будет направлен именно в данный объект без отображения в браузере. Клиентский сценарий может воспользоваться полученными данными так или иначе по замыслу программиста. Иначе говоря, в последнем случае поток вывода не будет автоматически отображаться интернет-обозревателем, а подчинится соответствующему клиентскому сценарию. Данная специфика лежит в русле так называемой технологии AJAX, о которой более подробно рассказывалось в *разд. 4.12.5*.

Некоторые программы на PHP могут иметь универсальный характер, позволяющий использовать их неоднократно и в различных проектах. В подобных случаях можно сохранять фрагменты кода (определения функций, классов и др.) в отдельных файлах, а в конкретный сценарий включать их с помощью встроенной функции `include(имя_файла)`. Ее параметр — имя (возможно, с указанием пути) или URL-адрес файла, содержимое которого необходимо вставить в место вызова данной функции `include()`. Включаемый файл содержит PHP-код, который должен быть заключен в дескрипторы `<?php` и `?>`, если мы хотим, чтобы он выполнился, а не просто отобразился в браузере клиента. Расширение этого файла может быть произвольным, но обычно указывают inc, чтобы напомнить разработчику, что это включаемый файл. Например, `include("mylib.inc")`. Если включаемый файл содержит определения разработанных вами функций, то, возможно, вы захотите назначить ему расширение lib (от англ. library — библиотека).

5.1.2. Сообщения об ошибках

В сценарии могут встречаться ошибки, сообщения о которых по умолчанию выводятся непосредственно в окно браузера:

```
Warning: Division by zero in c:\Inetpub\wwwroot\proba.php on line 4
```

(Предупреждение: Деление на ноль в c:\Inetpub\wwwroot\proba.php в строке 4).

Вы можете исправить ошибку в исходном файле и перезагрузить его в браузер. При желании вывод сообщений об ошибках можно подавить с помощью специального оператора `@`, который нередко пишется перед вызовами (именами) функций. Однако на стадии разработки и отладки программ сообщения об ошибках очень полезны.

5.1.3. Принудительный выход из сценария

Нередко требуется принудительно прекратить выполнение сценария (выйти из программы). Для этого можно использовать функцию `exit()` или `die()`,

которая является псевдонимом для первой. При вызове функции `exit()` (или `die()`) допустимо задать параметр: строку сообщения, выводимую перед завершением сценария, или целое число от 0 до 254 (так называемый статус завершения), например,

```
exit("Принудительный выход");
```

5.1.4. Справка по PHP

Сила PHP заключается в простоте синтаксиса языка, поддержке объектно-ориентированного программирования и огромной библиотеке функций, чуть ли не на все случаи жизни.

Подробные сведения о PHP и, в частности о его встроенных функциях, можно найти в Интернете по адресу: **http://ru.php.net/download-docs.php**. Для пользователей Windows она представлена в CHM-формате (откомпилированном HTML-коде) и, следовательно, отображается в диалоговом окне с инструментами поиска и печати. Там вы найдете не только недостающие сведения по функциям и другим конструкциям языка PHP, но и особенности различных его версий. PHP, будучи открытым продуктом и кодом, постоянно и довольно быстро обновляется. Поэтому важно следить за изменениями. С этой целью посещайте сайт **www.php.net** по крайней мере два раза в год.

> ***Внимание!***
>
> При работе со справочным руководством по функциям PHP обращайте особое внимание на состав и тип их параметров, а также на тип возвращаемого значения. От версии к версии PHP они могут изменяться.

Итак, коль скоро модуль PHP и Web-сервер установлены и слаженно работают, можно вплотную приступить к изучению языка PHP и программированию на нем.

5.2. Вывод и типы данных

Вывод данных различных типов (обычных текстовых строк, строк HTML-кода, а также чисел, массивов и др.) в окно Web-браузера с помощью PHP можно выполнить посредством оператора `echo`. Оператор особенно удобен для вывода сообщений (как предусмотренных вашим приложением, так и отладочных).

Синтаксис оператора вывода `echo`:

```
echo элемент1, элемент2, … , элементN.
```

Данные `элемент1, элемент2, … , элементN` выводятся в окне браузера друг за другом без пробелов и запятых (рис. 5.1). Заметим, что отображаемый результат не изменится, если мы перепишем оператор так

```
echo "AAABBB123";
```

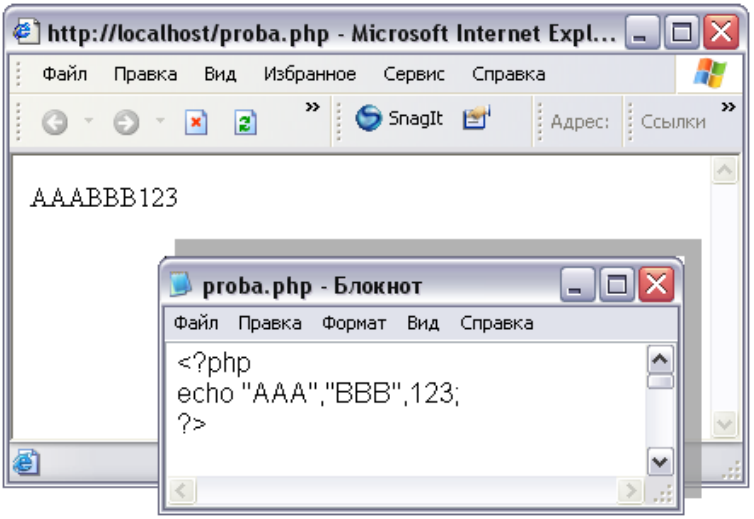

Рис. 5.1. Вывод нескольких элементов данных с помощью оператора `echo`

Особо следует отметить вывод строковых данных, которые заключаются в двойные или одинарные кавычки. Если вы хотите вывести произвольную последовательность символов с помощью оператора `echo`, то ее следует заключить в кавычки. Внутри этих кавычек могут быть любые символы (буквы, цифры и т. п.), в том числе имена переменных (см. *разд. 5.3*). В случае двойных кавычек строка символов будет интерпретироваться PHP. Если, например, в ней находятся имена переменных, то в отображаемой строке вместо них будут подставлены значения. Более того, если в этой строке найдутся теги HTML (дескрипторы, заключенные в угловые скобки), то браузер отобразит этот HTML-код так, как он и должен это делать при восприятии HTML-документов. Если же вы хотите увидеть строку символов так, как ее написали, без какой бы то ни было интерпретации, то ее следует заключить в одинарные кавычки. В листинге 5.1 приведен тестовый PHP-код, результат которого отображается в браузере, как показано на рис. 5.2.

Листинг 5.1. Пример использования оператора `echo`

```
<?php
$x="Привет"; // присвоение значения переменной
echo $x, " всем!", "<br>";
echo "$x всем!<br>";
echo '$x всем!<br>'
?>
```

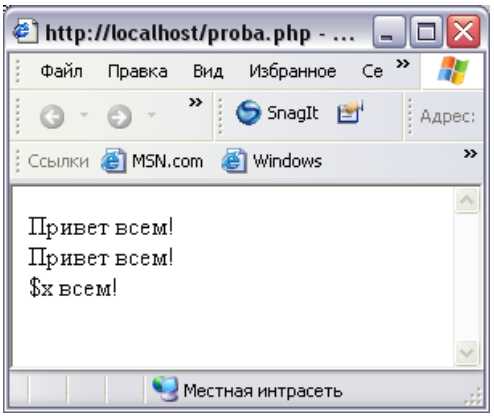

Рис. 5.2. Пример вывода с помощью оператора `echo`

Здесь использован оператор присваивания значения переменной `$x="Привет"`. Подробнее о переменных и присваивании им значений рассказано в *разд. 5.3*. Далее в сценарии следуют три оператора `echo`. Первые два из них дают одинаковые результаты, а третий выводит строку символов, заключенную в одинарные кавычки. Элемент `$x` кода PHP в этой строке не интерпретируется (т. е. не заменяется его значением), а выводится просто в виде последовательности символов.

> ***Внимание!***
>
> В PHP способ отображения символьных строк существенно зависит от того, в двойных или одинарных кавычках заключены символы строки. В JavaScript кавычки различных типов эквивалентны.

В PHP, как и в JavaScript, к данным применяется слабый (динамический) контроль типов. В операторах с разнотипными данными последние автоматически приводятся к требуемому типу. Этой возможностью следует пользоваться осторожно во избежание недоразумений.

В табл. 5.1 приведены основные (примитивные) типы данных, которые определены в PHP.

Таблица 5.1. *Типы данных в PHP*

Тип данных		Примеры	Описание значений

Строковый или символьный (`String`)		`"Привет всем!"`
		`"д.т. 123-4567"`
		`'Сегодня 14.03.08г.'`
			Последовательность символов, заключенная в кавычки, двойные или одинарные

Числовой	Целочисленный (`integer`)	`-789`
		`45`
		`0`
			Число, последовательность цифр, перед которой может быть указан знак числа (+ или –); перед положительными числами не обязательно ставить знак +. Число записывается без кавычек

	С плавающей точкой (`float`)	`3.14`
		`12.5E2`
		`35E-3`
			Число с дробной частью. Целая и дробная части чисел разделяются точкой.
			В экспоненциальной форме символ E используется для обозначения 10, за которым следует число, указывающее степень. Например, запись 5E2 означает $5 \cdot 10^2$, т. е. 500

Логический (булевский, `Boolean`)		`true` `false`	Этот тип имеет два значения: `true` (истина, да) или `false` (ложь, нет)
`Null`		`null`	Этот тип данных имеет одно значение — `null`, указывающее на отсутствие какого бы то ни было значения
`Массив (Array)`			Этот тип данных имеет одно множество значений, которые могут быть различных типов
Объект (`Object`)			Программный объект, определяемый своими свойствами
Ресурс (Resourse)			Указатель на открытый файл, соединение с базой данных, область изображения и т. п.; создается и используется специальными функциями

Внимание!

Ключевые слова `true`, `false` и `null` могут присутствовать в выражениях PHP в любом регистре. В JavaScript они могут записываться только в нижнем регистре.

В выражениях с операторами допустимы данные различных типов. Например, в арифметическом выражении могут оказаться данные не только числового, но и строкового или логического типов. В таких случаях интерпретатор PHP автоматически приводит данные к нужному типу.

В выражении с арифметическим оператором сложения + операнды могут быть как строковыми, так и числовыми.

Пример

```
5 + 3                  // результат: 8
"5" + "3"              // результат: 8
"5" + 3                // результат: 8
"5рублей" + 3          // результат: 8
"Доход 5" + 3          // результат: 8
"Привет" + 3           // результат: 3
"Привет" + "всем"      // результат: 0
5 + true               // результат: 6
5 + false              // результат: 5
"Привет" + true        // результат: 1
"Привет" + false       // результат: 0
```

Если строка не начинается с числа (перед которым может быть указан знак), то в арифметических выражениях она преобразуется в число 0.

Примечание

В PHP оператор + работает совсем не так, как в JavaScript. В JavaScript оператор + применительно к числам выполняет сложение, а к строкам — конкатенацию (склейку). В выражениях JavaScript с этим оператором и данными строкового и числового типов последние приводятся к строковому, а не числовому типу. Так например, выражение `"5" + 3` даст в результате "53", а не 8, как в PHP.

Данные различных типов могут участвовать в логических операциях и операциях сравнения, которые возвращают логическое значение `true` или `false`. В таких случаях интерпретатор PHP автоматически приводит данные нелогического типа к логическому. При этом следующие значения преобразуются в `false`:

- строка "FALSE", независимо от регистра;
- пустая строка "" и строка "0";

- число 0, целое или с плавающей точкой;
- Null;
- пустой массив (количество элементов равно 0);
- пустой объект (количество элементов равно 0).

Другие значения преобразуются в `true`.

Значения логического типа могут участвовать в арифметических операциях. В таких случаях значение `false` преобразуется в 0, а `true` — в 1. Например, выражение `true + true` возвращает 2, а `true + false` — 1.

Иногда требуется некоторые данные привести к определенному типу, чтобы гарантированно обеспечить корректность последующих операций над ними, не полагаясь на возможности автоматического преобразования типов. Для этого служат специальные операторы — имя типа в круглых скобках, записываемое перед данными, которые необходимо преобразовать:

- `(int)` или `(integer)` — приведение к целочисленному типу;
- `(float)`, `(double)` или `(real)` — приведение к числовому типу с плавающей точкой;
- `(string)` — приведение к строковому типу;
- `(bool)` или `(boolean)` — приведение к логическому (булевскому) типу;
- `(array)` — приведение к типу массив;
- `(object)` — приведение к типу объект.

Примеры

```
(int) 12.65           // результат: 12
(float) "12.65 руб."  // результат: 12.65
(bool) "12.65 руб."   // результат: true
```

Перечисленные операторы преобразования типов применимы и к переменным, и к вызовам функций, о чем будет рассказано в следующих разделах.

5.3. Переменные и оператор присваивания

5.3.1. Имена переменных

Имя переменной (идентификатор) — последовательность букв, цифр, символов подчеркивания без пробелов и знаков препинания, начинающаяся с буквы или символа подчеркивания. Перед именем переменной обязательно пишется символ доллара (`$`).

По существу префикс $ в имени переменной указывает интерпретатору, что он имеет дело с переменной, а не с чем-либо иным. Поэтому в имени переменной допустимы и ключевые слова языка, такие как if, else, while и т. п.

Примеры правильных имен переменных: $myName, $my_adress, $_x, $tel123_4567, $var, $Сумма, $if.

Примеры неправильных имен переменных: $512group, $my adress, tel:412 3456, $Сумма$.

ВНИМАНИЕ

PHP является регистрозависимым языком только относительно имен переменных и констант. Это означает, что изменение регистра символов (с прописных на строчные, и наоборот) в имени переменной приводит к другой переменной. Например, $variable, $Variable и $vaRiabLe — различные переменные. Однако ключевые слова языка (if, while, function и т. п.), а также имена функций можно записывать в любом регистре. Напомню, что в отличие от PHP, язык JavaScript полностью регистрозависимый.

В JavaScript символ $ может встречаться в имени переменной несколько раз и в любом месте или отсутствовать совсем. В именах переменных PHP символ $ обязательно находится на первом месте. Повторение его возможно, но только сразу же после первого. Однако в этом случае он выполняет особую роль *(см. разд. 5.3.4)*.

5.3.2. Создание переменных

Переменная создается при первом ее упоминании в программном коде с присваиванием ей некоторого значения. Оператор присваивания обозначается знаком равенства (=). Слева от него записывают имя переменной, а справа — выражение языка, возвращающее в результате вычисления некоторое значение. В простейшем случае справа от символа оператора присваивания указывают простое значение — число, строку символов в кавычках, ключевое слово, обозначающее специальное значение некоторых типов (true, false, null).

Создать переменную можно с помощью оператора присваивания (=).

Пример

```
$var = "Привет всем";       /* переменной $var присвоено строковое
                               значение "Привет всем" */
$x = 12.5E2;    // переменной $x присвоено числовое значение 1250
$x = true;      // переменной $x присвоено логическое значение true
```

Одной и той же переменной можно присвоить то одно, то другое значение. Тип переменной определяется типом значения, которое она в данный момент имеет (содержит). Так, в рассмотренном примере переменная `$x` сначала имела значение числового типа, а затем — логического.

Не следует путать оператор присваивания с отношением равенства и соответствующей операцией сравнения. Выражение с оператором `=` интерпретатор вычисляет следующим образом: переменной слева от него присваивается значение, расположенное справа.

Значение одной переменной можно присвоить другой переменной:

```
$x = 5;
$y = $x;           // переменная $y имеет значение 5
```

Переменные могут участвовать в выражениях с операторами:

```
$x = 5;
$y = $x + 2;      // переменная $y имеет значение 7
```

Когда мы пишем `$y = $x`, то тем самым выражаем следующее: значение переменной `$x` присвоить переменной `$y`. Это так называемое присвоение по значению, при котором значение одной переменной копируется в другую. Однако кроме этого способа в PHP можно организовать присвоение по ссылке, при котором копирование значения не происходит, а новая переменная становится псевдонимом (еще одним именем) для той переменной, на которую она ссылается. Для присвоения значения по ссылке используется символ `&` (амперсанд). Так, в выражении `$x = &$y;` переменная `$x` ссылается на `$y`. При этом обращение к первой переменной возвращает значение второй, а изменение значения `$x` отражается на значении `$y`, как и, наоборот, смена значения `$y` изменяет значение `$x`.

Пример

```
<?php
$x = "Саша";
$y = "Маша";
$x = &$y;            // значение $x равно "Маша"
$x = "Вася";         // значения $x и &y равны "Вася"
/* Вывод значений */
echo '$x=', $x;      // вывод строки: $x=Вася
echo "<br>";         // перевод строки
echo '$y=', $y;      // вывод строки: $y=Вася
?>
```

Примечание

Присвоение значений переменным по ссылке происходит быстрее, чем по значению, и это становится заметным, например, при установке значений элементам больших массивов с помощью оператора цикла.

У существующей переменной можно удалить текущее значение, т. е. очистить переменную и занимаемую ею память. При этом переменная продолжает существовать. Для этой цели предусмотрена функция `unset(имя_переменной)`:

```
$x = 5;
unset($x);        // удаление значения переменной $x
```

Замечание

В действительности в результате выполнения функции `unset($x)` переменная `$x` приобретает значение `null`.

Как уже отмечалось, тип переменной определяется типом значения, которое она имеет в текущий момент времени. Однако в PHP имеются операторы приведения типов (см. *разд. 5.2*). Обычно они обеспечивают надлежащее преобразование данных, в результате которого можно рассчитывать на корректное выполнение тех или иных операций. Напомню, что и без этих операторов интерпретатор PHP, анализируя выражение, преобразует данные к некоторому подобающему в данной ситуации типу. Однако его "интеллектуальные потуги" иногда могут не соответствовать ожиданиям автора программы, и тогда разработчик может форсировать приведение типов по своему усмотрению.

Пример

```
$ x= ”5.3 рублей”;     // строка символов ”5.3 рублей”
$y = $x + 7;           // число с плавающей точкой 12.3
$y = (int)$x + 7;      // целое число 12
$y = (float)$x + 7;    // число с плавающей точкой 12.3
$z = (bool)$y;         // логическое значение true
```

5.3.3. Отображение значений переменных

Значения переменных можно вывести на экран оператором `echo`. Этот оператор направляет вывод значений в выходной документ, отображаемый браузером. При выполнении примеров и других экспериментальных кодов на языке PHP, а также при отладке ваших реальных сценариев очень полезно организовать вывод промежуточных и окончательных результатов на экран. Для этого и удобен оператор `echo` (см. *разд. 5.1*). Вывод текущих значений пере-

менных можно реализовать несколькими способами. Вот некоторые наиболее употребительные из них:

- `echo $var1, $var2, …, $varN;` — вывод значений переменных, перечисленных в операторе `echo` через запятую; при отображении значения переменных следуют друг за другом без пробелов (запятые между именами переменных не отображаются);
- `echo "Переменные: $var1, $var2, …, $varN";` — вывод символьной строки (заключенной в двойные кавычки), в которой имена переменных заменяются их значениями;
- `echo 'Переменные: $var1, $var2, …, $varN';` — вывод текста строки, т. е. всего, что указано внутри одинарных кавычек без подстановок значений переменных.

Замечу попутно, что в строку символов, подлежащую выводу в окно браузера, можно вставлять теги HTML, которые позволяют отформатировать выводимый текст. Листинг 5.2 содержит PHP-код, который иллюстрирует приведенные замечания, а результат его выполнения, отображенный в Internet Explorer, показан на рис. 5.3.

Листинг 5.2. Пример вывода переменных в окно браузера

```
<?php
$var1="Привет";
$var2=" всем!";
echo $var1, $var2; // вывод значений нескольких переменных
echo "<br>Переменные: $var1, $var2<br>";
echo 'Переменные: $var1, $var2';
?>
```

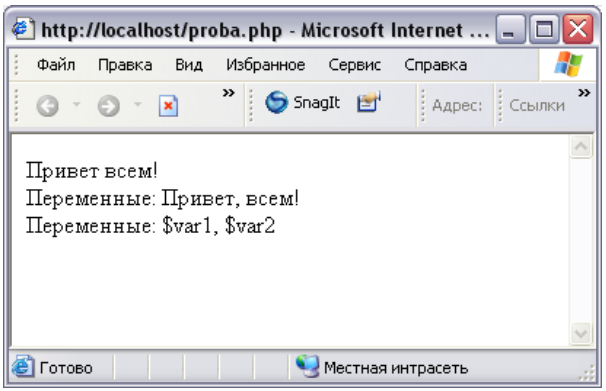

Рис. 5.3. Пример отображения значений переменных

Внимание!

Результат вывода с помощью оператора `echo` строки, в которой указаны переменные, существенно зависит от того, в какие кавычки заключена эта строка символов — двойные или одинарные. В случае двойных кавычек вместо имен переменных в строку подставляются значения. Если кавычки одинарные, то содержимое строки интерпретатор PHP не анализирует и не вычисляет, а выводит как есть.

Значения некоторых типов оператор `echo` выводит не всегда так, как они были заданы в коде программы. Например, логическое значение `true` отображает как 1, а `false` — не отображает вовсе; числовые значения могут быть представлены в экспоненциальной или неэкспоненциальной формах в зависимости от их величины (рис. 5.4). Тем не менее, вы можете отформатировать выводимые сообщения по своему усмотрению (см. *разд. 5.6*).

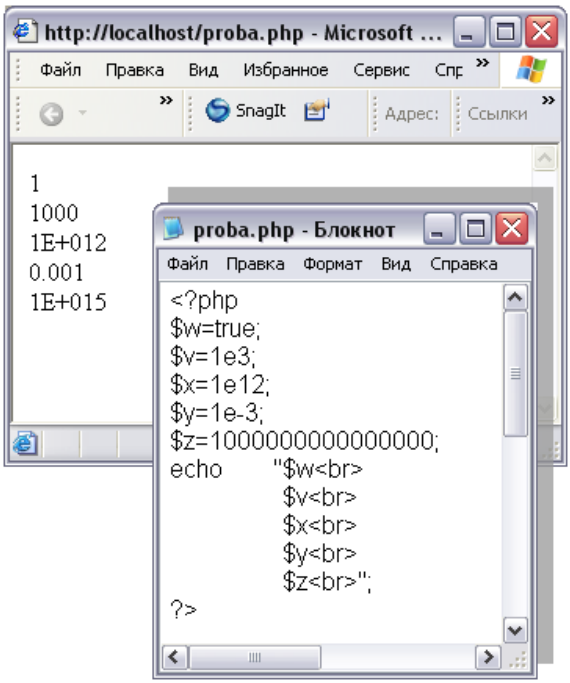

Рис. 5.4. Отображение логического и числовых значений с помощью оператора `echo`

Вывод значений элементов массива в составе текстовых строк имеет некоторую особенность: элементы массива необходимо заключать в фигурные скобки (см. *разд. 5.9.3*).

Для вывода более подробной информации о переменных, которая может понадобиться при отладке программы, служит функция `var_dump(список_переменных)`. В списке переменных указывают одно или несколько имен переменных, разделенных запятыми. Эта функция ничего не возвращает. На рис. 5.5 показан пример использования `var_dump()` для переменных различных типов. Так, для строковой переменной выводится ее тип, длина значения и само значение, для числовых и логических переменных — тип и значение. Данная функция особенно полезна при выводе информации о массивах. В этом случае выводится тип переменной `(array)`, число элементов массива и список сведений об элементах в виде `индекс => элемент`. Индекс может быть числовым или символьным в зависимости от того, как он был задан в определении массива. Подробнее о массивах см. *разд. 5.9*.

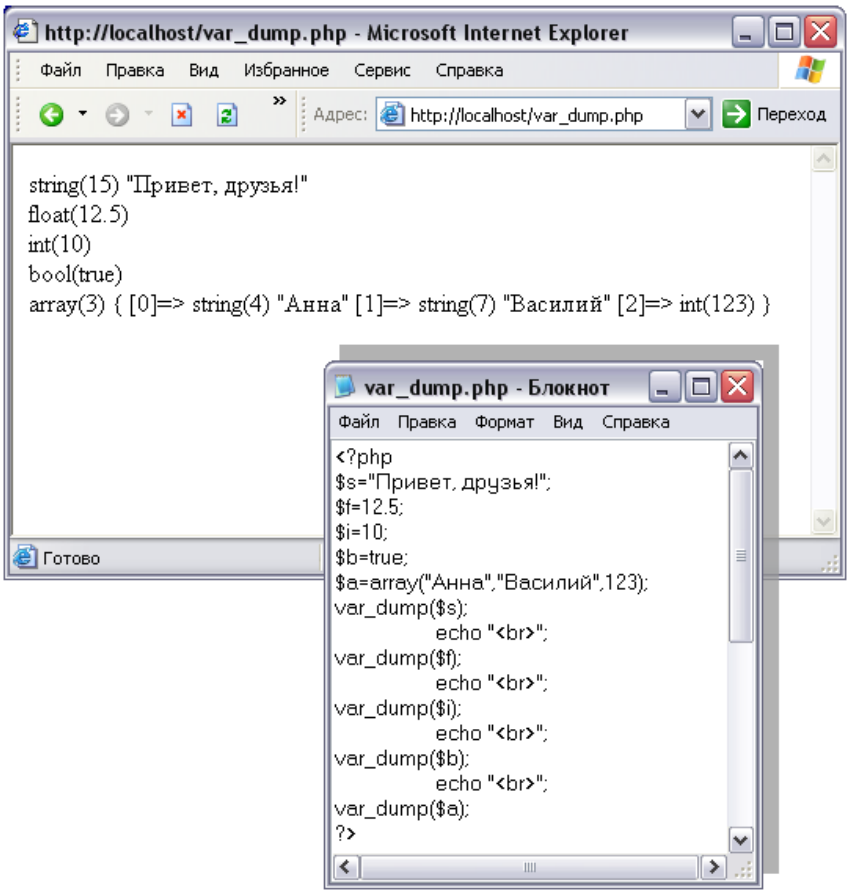

Рис. 5.5. Вывод информации о переменных с помощью функции `var_dump()`

Менее информативной, чем `var_dump()`, функцией вывода информации о переменных является `print_r(список_переменных)`. Для массивов эта функция выводит список вида `индекс => элемент` без указания типов и других характеристик элементов. Для переменных других типов выводятся только их значения. Пример использования этой функции можно найти в *разд. 5.7.3*.

5.3.4. Переменные переменные

В программах иногда может понадобиться интерпретировать строковое значение как имя переменной. Иначе говоря, мы хотим присвоить некоторой переменной в качестве значения имя другой переменной. Так можно создавать переменные динамически, т. е. программно задавать переменные, которые необходимы в данный момент вычислительного процесса. Другими словами, переменная может содержать другую переменную, называемую "переменной переменной". Для создания подобной переменной перед ее именем пишут двойной символ `$`. Так, запись `$$x` означает переменную, имя которой указано в качестве значения переменной `$x`.

На рис. 5.6 показаны код программы в текстовом редакторе и результат его выполнения в окне браузера. Здесь переменная `$x` содержит строковое значение `"Девочка"`. Выражение `$$x="Маша"` означает, что переменной, имя которой содержится в переменной `$x`, присваивается значение `"Маша"`. Таким образом, создается переменная `$Девочка` и ей присваивается значение `"Маша"`. С помощью оператора `echo` выводится строка, содержащая значение переменной `$x` и динамически созданной переменной `$Девочка`.

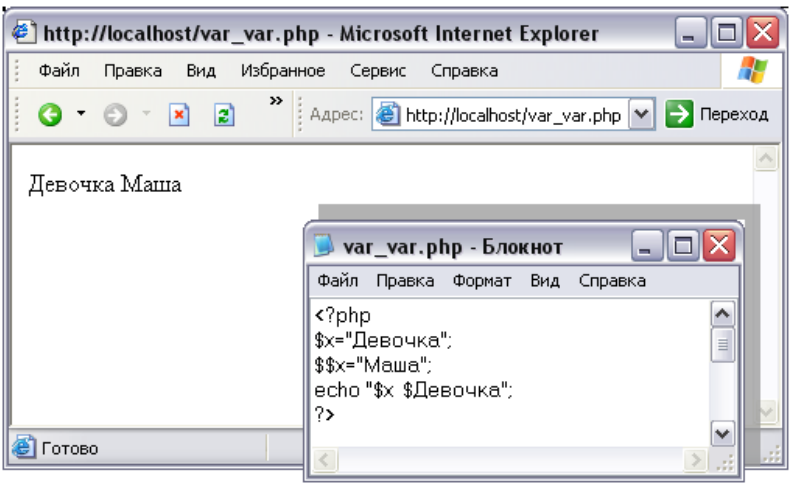

Рис. 5.6. Пример использования переменных переменных

В рассматриваемом примере оператор `echo "$$x"` выведет строку `"$Девочка"`, оператор `echo "${$x}"` — строку `"Маша"`. Здесь использовались фигурные скобки, чтобы указать интерпретатору, что требуется вывести, а именно значение переменной `$Девочка: значение переменной {значение переменной x}`.

На рис. 5.7 показан еще один пример, в котором над переменными переменными производятся арифметические действия.

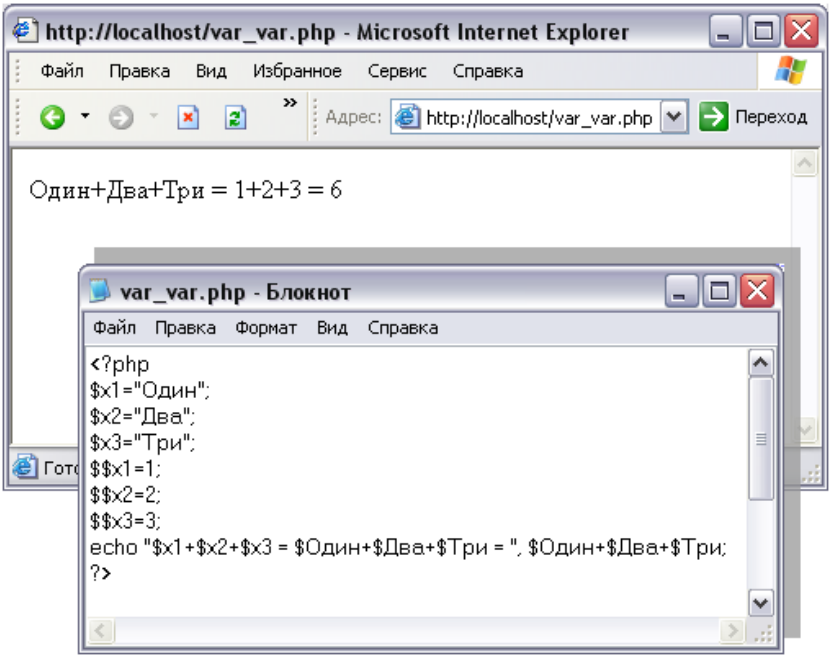

Рис. 5.7. Пример вычислений над переменными переменными

5.3.5. Область действия переменных

Область действия (видимости, доступности) переменной — программный код из того же файла, в котором она была определена. При этом она также видна и в PHP-кодах, включаемых в данный файл из внешнего файла с помощью функции `include()`. Однако внутри кода функций такие переменные не видны, если не предпринять специальных мер.

В следующем примере переменная `$x` видна во всем программном коде, включая и код из файла mylib.php. Если код из этого файла не изменяет значение переменной `$x`, то выведенное значение будет равно 9.

Пример

```
<?php
$x=5;
?>
Далее выводим переменную $x:
<?php
include("mylib.php"); // включение файла
$x=$x+4;
echo "$x"; // вывод значения переменной $x
?>
```

Таким образом, переменные в PHP были бы глобальными в полном смысле этого слова, если не одно обстоятельство. Дело в том, что программный код может содержать определения и вызовы функций. Если созданная во внешнем коде переменная присутствует еще и в коде функции, то там она будет не глобальной, как ожидается, а локальной. Это означает, что значение переменной, заданное во внешнем коде, не видно внутри кода функции. С другой стороны, изменения значения этой переменной внутри функции не отразятся на этой же переменной во внешнем коде. Чтобы сделать переменные, созданные во внешнем коде, доступными и внутри кода функций (т. е. обеспечить им настоящую глобальность), они должны быть объявлены глобальными в коде функций с помощью ключевого слова `global`. Кроме того, переменные внутри функций могут быть объявлены как статические `(static)`. Статическая переменная является локальной, но сохраняет свое значение и при следующем вызове функции, где она была определена, данное значение может быть использовано. Пример такой функции — `strtok()` (см. *разд. 5.6.3*).

Примечание 1

В языке JavaScript глобальные переменные автоматически доступны и внутри функций, если только они не были определены там как локальные, т. е. глобальные переменные видны везде. Так, в JavaScript локализация переменной в теле функции задается ключевым словом `var`. Если, например, переменная `myvar` глобальная, то для использования в коде функции локальной переменной с таким же именем, достаточно там написать выражение `var myvar`.

Примечание 2

В PHP имеется множество предопределенных переменных типа массив. Эти переменные глобальны в полном смысле этого слова и содержат сведения о Web-сервере, окружении и данных, которое сервер получил от клиента (браузера).

Примечание 3

Базовый оператор присвоения значений обозначается знаком равенства. Однако допускаются и специальные операторы присвоения значений, которые в ряде случаев оказываются удобнее. О них рассказывается в *разд. 5.5.4*.

5.3.6. Проверка существования переменных и их типов

Иногда требуется проверить, определена ли та или иная переменная. Для этого служит функция

```
isset(имя_переменной1, имя_переменной2, …, имя_переменнойN)
```

Если все переменные в списке существуют, то функция возвращает `true`, в противном случае — `false`.

Выяснить, является ли значение переменной пустым, позволяет функция

```
empty(имя_переменной)
```

которая возвращает `true`, если переменная не объявлена, является пустым массивом или ее значение равно:

- нулю;
- пустой строке;
- строке с нулем;
- `false`;
- `null`.

В остальных случаях функция `empty()` возвращает `false`.

Примечание

Начиная с версии 5 PHP, объект без свойств не считается пустым с точки зрения функции `empty()`.

Проверить тип переменной можно с помощью следующих функций:

- `is_string(имя_переменной)` — возвращает `true`, если переменная является строковой, и `false` — в противном случае;
- `is_int(имя_переменной)` — возвращает `true`, если переменная является целочисленной, и `false` — в противном случае;
- `is_float(имя_переменной)` — возвращает `true`, если переменная является числовой с плавающей точкой, и `false` — в противном случае;
- `is_numeric(имя_переменной)` — возвращает `true`, если переменная является числовой (типа `integer` или `float`), а также строкой, содержащей только числа. В противном случае возвращает `false`;

- `is_null(имя_переменной)` — возвращает `true`, если переменная имеет значение `null`, и `false` — в противном случае;
- `is_array(имя_переменной)` — возвращает `true`, если переменная является массивом (имеет тип `array`), и `false` — в противном случае.

5.4. Константы

Константы похожи на переменные, поскольку имеют имена (идентификаторы) и значения. Константа получает значение однажды в коде программы, и далее оно остается неизменным. Константы имеют следующие особенности:

- имя константы не содержит префикса `$`;
- в имени константы недопустимы ключевые слова PHP, такие как `if`, `else`, `while`, `echo`, `default`, `function` и т. п.;
- значения констант могут принадлежать только типам `string`, `integer`, `float` и `boolean`;
- создание констант (объявление имени и присвоение значений) обеспечивает специальная функция `define("имя_константы", значение)`. Например, `define("MYNAME","Дунаев Вадим Вячеславович")`. Обычный оператор присваивания значений (=) здесь не годится.

> ***Примечание***
>
> В JavaScript имеются встроенные (предопределенные) константы, но создавать свои собственные константы программист не может.

Константы полезны, если вы хотите обезопасить себя от случайного изменения в программе данных, которые должны быть постоянными. При необходимости их изменить это придется делать в одном месте, а именно там, где вы их однажды определили с помощью функции `define()`. Кроме того, вы можете создавать один и тот же программный код, который должен работать с различными наборами исходных данных, адаптированными, например, для конкретного пользователя (среды выполнения, варианта реализации и т. п.). В этом случае использование констант поможет улучшить технологичность вашего проекта. Пример определения и вывода констант приведен на рис. 5.8.

При указании констант в строке, заключенной в двойные кавычки, подстановка их значений вместо имени (как в случае переменных) не происходит.

> ***Примечание***
>
> В PHP имеется множество предопределенных констант. Например, константа `__FILE__` (окаймляется двойными символами подчеркивания) содержит полное имя текущего PHP-файла, а константа `__LINE__` содержит номер текущей строки в PHP-файле. Математические константы приведены в *разд. 5.7.2*.

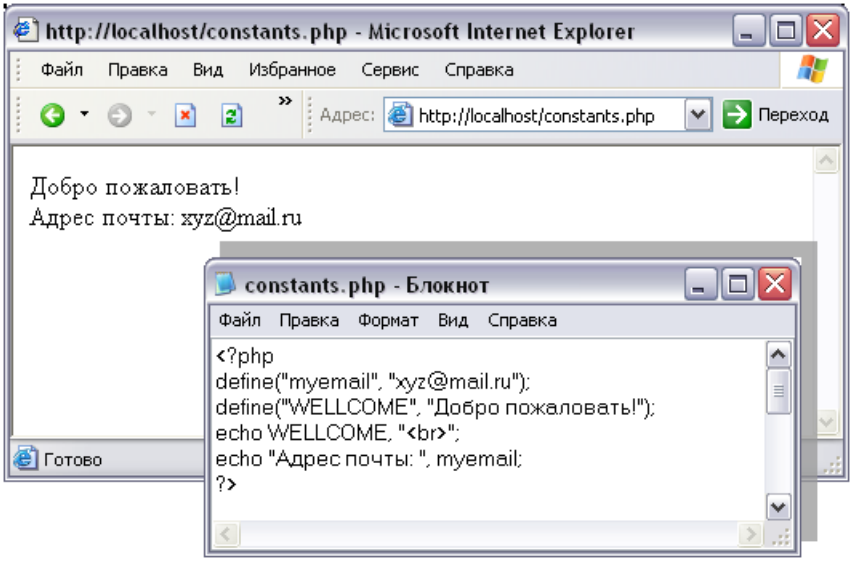

Рис. 5.8. Пример определения и вывода констант

Иногда требуется проверить, определена ли та или иная константа. Для этого существует функция

`defined(имя_константы1, имя_ константы2, …, имя_ константыN)`

Если все константы в списке определены, то функция возвращает `true`, в противном случае — `false`.

Примечание

Аналогичная функция для проверки существования переменных — `isset()`.

5.5. Операторы

В предыдущем разделе мы рассмотрели оператор присваивания. Теперь обратимся к другим операторам, предназначенным для работы с данными различных типов.

5.5.1. Комментарии

Операторы комментария позволяют выделить фрагмент кода, который не выполняется интерпретатором, а служит лишь для пояснения программы. Не пренебрегайте комментариями при написании своих программ.

В PHP допустимы три вида оператора комментария:

- `//` — одна строка символов, расположенная справа от этого оператора, считается комментарием;
- # — действует так же, как и оператор //;
- `/*...*/` — все, что заключено между `/*` и `*/`, считается комментарием; оператор позволяет выделить несколько строк в качестве комментария.

5.5.2. Арифметические операторы

Арифметические операторы (табл. 5.2), такие как сложение, умножение и т. д., в PHP предназначены в основном для манипулирования числовыми данными, хотя могут применяться и к данным других типов.

Таблица 5.2. *Арифметические операторы*

Оператор	Название	Пример
`+`	Сложение	`$X+$Y`
`-`	Вычитание	`$X-$Y`
`*`	Умножение	`$X*$Y`
`/`	Деление	`$X/$Y`
`%`	Деление по модулю (остаток от деления)	`$X%$Y`
`++`	Увеличение на единицу	`$X++` `++$X`
`--`	Уменьшение на единицу	`$Y--` `--$X`

Первые четыре оператора не требуют особых пояснений. Оператор деления (`/`) всегда возвращает число типа `float`, даже если в результате получается целое.

Оператор деления по модулю возвращает остаток от деления первого числа на второе, если первое число не меньше второго. В противном случае возвращается первое число. Например, `10%4` возвращает 2, 3%5 — 3, 12%12 — 0. При этом результат будет отрицательным числом, если первый из операндов отрицателен. Например, –5%2 возвращает –3, 5%–2 — 3.

Операторы `++` и `--` сочетают в себе действие операторов соответственно сложения и вычитания, а также присваивания. Выражение `$X++` эквивалентно выражению `$X=$X+1`, однако оно возвращает не новое, а прежнее значение

переменной $x. Таким образом, оператор $x++ сначала возвращает текущее значение $x, а затем увеличивает его на единицу. В противоположность этому, оператор ++$x сначала увеличивает значение переменной $x на единицу, а затем возвращает новое значение. Аналогично, оператор $x-- сначала возвращает текущее значение $x, а затем уменьшает его на единицу. Оператор --$x делает это в противоположном порядке.

Операторы ++ и -- называют соответственно *инкрементом* и *декрементом*.

Символ - используется не только как бинарный (для двух операндов) оператор вычитания, но и как унарный (для одного операнда) оператор для указания того, что число отрицательное.

Однако формально ничто не мешает нам применить эти операторы к данным нечисловых типов. При этом все нечисловые операнды автоматически преобразуются в числа. Например, строковые данные интерпретатор попытается привести к числовому типу. Если префиксом строки является число, то эта строка будет преобразована в соответствующее число, в противном случае она будет преобразована в нуль. Логические значения преобразуются в единицу и нуль для true и false соответственно. Подробнее об этом см. *разд. 5.3*.

Примеры

```
5+3                // результат: 8
"5"+"3"            // результат: 8
"5"+3              // результат: 8
"5рублей"+3        // результат: 8
"Доход 5"+3        // результат: 3
"Привет"+3         // результат: 3
"Привет"+"всем"    // результат: 0
$x=8;
$y=3;
$x/$y;            // результат: 2.66666666667
$x%$y;            // результат: 2
$x++;      /* результат: значение переменной $x становится равным 9,
                  но выражение возвращает 8 */
++$x;     /* результат: значение переменной $x становится равным 9,
             и выражение возвращает 9 */
```

Примечание

В JavaScript в выражениях с оператором сложения + данных строкового и числового типа последние приводятся к строковому, а не числовому типу. Поэтому, выражение "Вася"+25 даст в результате "Вася25", а не 25, как в PHP. С другой стороны, в PHP выражение "25"+4 вернет 29, а в JavaScript — "254".

5.5.3. Строковый оператор

Две строки можно объединить (склеить) с помощью операции конкатенации, обозначаемой точкой. Применение этой операции к двум строкам дает в результате строку, полученную путем приписывания справа второй строки к первой. Например, выражение `"Саша"."Маша"` возвращает в результате `"СашаМаша"`.

Пример

```
$x="Привет";
$y="всем";
echo $x. " ".$y;
```

Здесь склеиваются три строки: значение переменной `$x`, строка с пробелом и значение переменной `$y`.

> ***Примечание***
>
> В JavaScript оператор конкатенации строк обозначается так же, как и оператор сложения чисел (+).

5.5.4. Дополнительные операторы присваивания

Кроме обычного оператора присваивания (=) имеются еще шесть дополнительных, сочетающих в себе присваивание и арифметические операции (табл. 5.3).

Таблица 5.3. *Дополнительные операторы присваивания*

Оператор	Пример	Эквивалентное выражение
+=	X+=Y	X=X+Y
-=	X-=Y	X=X-Y
*=	X*=Y	X=X*Y
/=	X/=Y	X=X/Y
%=	X%=Y	X=X%Y
.=	X.=Y	X=X+Y

Пример

```
$x="Привет"
$x.=" всем"       // "Привет всем"
$x=5; $x*=7       // 35
```

5.5.5. Операторы сравнения

Результат вычисления элементарного выражения, содержащего оператор сравнения и операнды (сравниваемые данные), — логическое значение (`true` или `false`). Операторы сравнения приведены в табл. 5.4.

Таблица 5.4. *Операторы сравнения*

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

> ***Внимание***
>
> Оператор "равно" записывается с помощью двух символов = без пробелов между ними, а оператор "тождественно равно" — с помощью трех подряд идущих символов =.

Сравнивать можно данные различных типов. Что касается оператора равенства (неравенства), то допустимы две их разновидности:

- *гибкий оператор равенства* (==) — выражение возвращает `true`, если операнды равны, иначе — `false`. При этом типы сравниваемых данных могут различаться;
- *жесткий оператор равенства* (===) — выражение возвращает `true`, если операнды не только равны, но имеют еще и одинаковые типы. В противном случае возвращается `false`.

Примеры

```
"7.2" == 7.2        // возвращает true
"7.2" === 7.2       // возвращает false
"1" == true         // возвращает true
1 == true           // возвращает true
```

```
"1" == "true"       // возвращает false
1 == "true"         // возвращает false
"0" == "false"      // возвращает false
"0" == false        // возвращает true
0 == "false"        // возвращает true
0 == false          // возвращает true
0 === "false"       // возвращает false
null==null          // возвращает true
null===null         // возвращает true
"abcd" > " abcd"    // возвращает true
"abc" < "abcd"      // возвращает true
"235ab" < "abcdxyz" // возвращает true
"235xyz" < "abc"    // возвращает true
```

Внимание

При практическом программировании следует обратить особое внимание на сравнение логических значений `true` и `false` с данными других типов, такими как пустые строки, `null`, "0", 0 и т. д. Многие функции возвращают `false` тогда, когда что-то не так (например, возникла ошибка преобразования данных, произошел выход за границы массива). Подобные ситуации обычно каким-либо образом обрабатываются в программе, и при этом чаще всего проверяется условие, не равен ли результат `false`. Здесь важно корректно сформулировать проверяемое условие.

5.5.6. Логические операторы

Логические данные, обычно получаемые с помощью элементарных выражений, содержащих операторы сравнения, можно объединять в более сложные конструкции. Для этого предусмотрены логические (булевские) операторы — логические союзы И, ИЛИ, исключающее ИЛИ, а также оператор отрицания НЕ (табл. 5.5). Выражения с логическими операторами возвращают значение `true` или `false`.

Таблица 5.5. *Логические операторы*

Оператор	Название	Пример
`!`	Отрицание (НЕ)	`!$X`
`&&` `and`	И	`$X&&$Y`
`||` `or`	ИЛИ	`$X\|\|$Y` `$X or $Y`
`xor`	Исключающее ИЛИ	`$X xor $Y`

Оператор отрицания ! применяется к одному операнду, изменяя его значение на противоположное: если $X имеет значение true, то !$X возвращает значение false, и наоборот. Логическим союзам И и ИЛИ соответствуют по два оператора. Они действуют одинаково, но имеют различные приоритеты, определяющие порядок их вычисления в сложных выражениях. Приоритеты логических операторов уменьшаются в порядке перечисления: !, &&, ||, and, xor, or.

В табл. 5.6 указано, какие значения возвращают операторы И, ИЛИ и xor при различных логических значениях двух операндов.

Таблица 5.6. *Примеры использования операторов* И, ИЛИ

$X	$Y	$X && $Y	$X || $Y	$X xor $Y
true	true	true	true	false
true	false	false	true	true
false	true	false	true	true
false	false	false	false	false

Операторы && и || еще называют *логическим умножением* и *логическим сложением* соответственно (конъюнкцией и дизъюнкцией).

Примеры

```
$x=false || 2*2==4;       // значение переменной x равно true
$x=5<2 || "abcd" <="xy"; // значение переменной x равно true
$y=!$x;                   // значение переменной y равно false
$z=$x && $y;              // значение переменной z равно false
$z=$x || $y;              // значение переменной z равно true
$x=true; $y=false;
$z=$x xor !$y;             // значение переменной z равно false
```

Сложные логические выражения, состоящие из нескольких более простых, соединенных операторами И, ИЛИ и исключающее ИЛИ, выполняются по так называемому *принципу короткой обработки*. Дело в том, что значение всего выражения зачастую можно определить, вычислив лишь одно или несколько более простых его составляющих, не затрагивая остальные. Данное обстоятельство требует особого внимания при тестировании сложных логических конструкций. Так, если какой-нибудь компонент выражения некорректен, то ошибка может остаться невыявленной, поскольку оно не протестируется.

5.5.7. Побитовые операторы

Побитовые (поразрядные) операторы применяются к целочисленным значениям и возвращают целочисленные значения. При их выполнении операнды предварительно приводятся к двоичной форме, в которой число представляется 32-разрядной последовательностью нулей и единиц (*двоичные разряды* или *биты*). Далее над битами выполняется некоторое действие, в результате которого получается новая последовательность битов, которая в итоге преобразуется к обычному целому числу — результату побитового оператора. В табл. 5.7 приведен список побитовых операторов.

Таблица 5.7. *Побитовые операторы*

Оператор	Название	Левый операнд	Правый операнд
`&`	Побитовое И	Целое число	Целое число
`|`	Побитовое ИЛИ	То же	То же
`^`	Побитовое исключающее ИЛИ	То же	То же
`~`	Побитовое НЕ	—	То же
`<<`	Смещение влево	Целое число	Число битов, на которое производится смещение
`>>`	Смещение вправо	То же	То же

Операторы `&`, `|`, `^` и `~` напоминают логические операторы, но их область действия — биты, а не логические значения. Оператор `~` изменяет значение бита на противоположное: 0 на 1, а 1 — на 0.

Примечание

Действие побитовых операторов в PHP такое же, как и в языке JavaScript.

5.5.8. Операторы условного перехода

Вычислительным процессом управляют операторы условного перехода `if`, `?:` и `switch`. Они обеспечивают выполнение того или иного фрагмента кода в зависимости от выполнения некоторого условия.

Оператор *if*

Синтаксис оператора `if` перехода следующий:

```
if (условие)
 {код, который выполняется, если условие истинно}
else
 {код, который выполняется, если условие ложно}
```

В фигурных скобках располагают блок кода — несколько выражений, разделенных точкой с запятой, как это обязательно в PHP. Если блок содержит не более одного выражения, то фигурные скобки можно опустить. Часть этой конструкции, определяемая ключевым словом `else` (иначе), необязательна. В этом случае остается только часть, определенная ключевым словом `if` (если):

```
if (условие){
 код, который выполняется, если условие истинно
}
```

Ключевое слово `else` можно заменить на `elseif (условие)`, чтобы установить проверку еще одного условия.

Пример

```
if (условие1){
        код, который выполняется, если условие1 истинно
} elseif (условие2){
        код, который выполняется, если условие1 ложно, а условие2 истинно
} else {
        код, который выполняется, если условие1 и условие2 ложны
};
```

Конструкций `elseif` в операторе `if` может быть несколько.

ПРИМЕЧАНИЕ

В JavaScript конструкция `elseif` не допускается.

Оператор *switch*

Листинг 5.3 иллюстрирует синтаксис оператора `switch`.

Листинг 5.3. Синтаксис оператора `switch`

```
switch (выражение) {
 case вариант1:
 код
 [break]
 case вариант2:
 код
 [break]
```

```
 ...
 [default:
 код]
}
```

Пример

```
$x=2;
switch ($x) {
 case 1:
 echo "x равно 1";
 case 2:
 echo "x равно 2";
 case 3:
 echo "x равно 3";
}
```

В приведенном примере сработают второй и третий варианты. Если мы хотим, чтобы сработал только один какой-нибудь вариант (только тот, который соответствует значению выражения), то нужно добавить оператор `break`.

Пример

```
$x=2;
switch ($x) {
 case 1:
 echo "x равно 1";
 break;
 case 2:
 echo "x равно 2";
 break;
 case 3:
 echo "x равно 3";
 break;
}
```

В этом примере сработает только второй вариант.

Оператор условия *?:*

Оператор условия является сокращенной формой оператора условного перехода `if...else...`. Обычно он применяется вместе с оператором присваивания

одного из двух возможных значений, в зависимости от значения условного выражения. Синтаксис оператора условия следующий:

```
условие ? выражение1 : выражение2
```

С оператором присваивания оператор условия имеет такой вид:

```
Переменная = условие ? выражение1 : выражение2
```

Оператор условия возвращает значение выражения *выражение1*, если *условие* истинно, в противном случае — значение выражения *выражение2*. Поэтому он может входить в составные выражения.

Пример

```
$x=5;
$x<10 ? "x меньше 10": "x больше или равно 10";
echo $x;
```

5.5.9. Операторы цикла

В PHP предусмотрены четыре оператора цикла: `for`, `while`, `do-while` и `foreach`. Операторы `for`, `while` и `do-while` мы рассмотрим в данном разделе. Оператор `foreach` (для каждого) специально предназначен для работы с массивами в PHP. Он очень удобен и будет подробно рассмотрен в *разд. 5.9.4*.

Оператор *for*

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

```
for ([начальное_выражение]; [условие]; [выражение_обновления])
{
 код
}
```

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

```
for ([начальное_выражение]; [условие]; [выражение_обновления]) {
 код
}
```

В следующем примере оператор цикла просто изменяет значение своего счетчика, выполняя 15 итераций:

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

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

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

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

Допустим, требуется вычислить `x` в степени `y`, где `x`, `y` — целые положительные числа. Алгоритм решения этой задачи прост: нужно вычислить выражение `x*x*x*...*x`, в котором сомножитель `x` встречается `y` раз. Очевидно, что если `y` не превышает 10, то можно воспользоваться оператором умножения: выражение будет не очень длинным. А как быть, когда `y` равно нескольким десяткам или сотням? Ясно, что в общем случае следует применить оператор цикла.

Пример: `x` в степени `y`

```
/* Вычисляем x в степени y */
$z=$x;  // $z хранит результат
for ($i=2; $i <= $y; $i++) {
 z=z*x;
}
```

В этом примере результат сохраняется в переменной `$z`. Ее начальное значение равно `$x`. Если `$y` равно 1, то оператор цикла не будет выполняться, поскольку не выполняется условие `2<=1` (обратите внимание на начальное значение счетчика цикла), и, следовательно, мы получим верный результат: `$x` в степени 1 равно `$x`. Если `$y` равно 2, то оператор цикла выполнит одну итерацию, вычислив выражение `$z=$z*$x` при `$z`, равном `$x` (т. е. `$z=$x*$x`). При `$y`, равном 3, оператор цикла сделает две итерации. На второй, последней, итерации выражение `$z=$z*$x` вычисляется при текущем значении `$z`, равном `$x*$x`, и, следовательно, становится равным `$x*$x*$x` (т. е. `x` в степени 3).

Рассмотрим довольно традиционный при изучении операторов цикла пример вычисления факториала числа.

Пример: n!

```
/* Вычисляем n! */
$z=1;  // z хранит результат n!
if ($n >1) {
 for ($i=2; $i <= $n; $i++) {
 $z=$z*$i;
 }
}
```

Для принудительного (т. е. не по условию) выхода из цикла используется оператор `break` (прерывание). Если этот оператор встречается в теле оператора цикла, то он сразу же завершается без выполнения последующих выражений кода в теле и даже выражения обновления. Обычно оператор `break` применяется при проверке некоторого дополнительного условия, выполнение которого требует завершения цикла, несмотря на то, что условие в заголовке цикла еще не выполнено. Листинг 5.4. иллюстрирует типовую структуру оператора цикла с использованием `break`.

Листинг 5.4

```
for ([начальное_выражение]; [условие1]; [выражение_обновления])
{
 код
 if (условие2){
 код
 break
 }
 код
}
```

Для управления вычислениями в операторе цикла указать оператор `continue` (продолжение). Так же, как и `break`, этот оператор применяется в теле цикла вместе с условным переходом. Однако в отличие от `break`, оператор `continue` прекращает выполнение последующего кода, выполняет выражение обновления и возвращает вычислительный процесс в начало оператора цикла, где проверяется условие, указанное в заголовке. Типовая структура оператора цикла с использованием `continue` приведена в листинге 5.5.

Листинг 5.5

```
for ([начальное_выражение]; [условие1]; [выражение_обновления] )
{
 код
 if (условие2){
 код
 continue
 }
 код
}
```

Оператор *while*

Синтаксис этого оператора следующий:

```
while (условие)
{
 код
}
```

При выполнении этого оператора сначала проверяется условие, указанное в заголовке, т. е. в круглых скобках справа от ключевого слова `while`. Если оно истинно, то выполняется код в теле оператора цикла, заключенный в фигурные скобки. В противном случае код не выполняется. При выполнении кода (завершении первой итерации) вычислительный процесс возвращается к заголовку, где снова проверяется условие, и т. д.

Рассмотрим два примера задач, которые мы уже решали ранее с использованием оператора цикла `for`.

Пример: `x` в степени `y`

```
/* Вычисляем x в степени y */
$z=$x;  // $z хранит результат
$i=2;
while ($i==2) {
 $z=$z*$x;
 $i++;
}
```

Пример: n!

```
/* Вычисляем n! */
$z=1;  // $z хранит результат $n!
if ($n >1) {
 $i=2;
 while ($i <= $n ) {
 $z=$z*$i;
 $i++;
 }
}
```

Во всех приведенных примерах есть счетчик цикла, который инициировался заранее, до оператора цикла и обновлялся в теле оператора цикла.

Управлять вычислительным процессом в операторе `while`, так же как и в операторе `for`, можно с помощью операторов `break` и `continue`.

> ***Внимание***
>
> Если в `while` вы используете счетчики циклов, то будьте осторожны, применяя `break` и `continue`.

Оператор *do-while*

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

```
do {
 код
}
while (условие);
```

В отличие от `while` в операторе `do-while` код выполняется хотя бы один раз, независимо от условия. Условие проверяется после выполнения кода. Если оно истинно, то снова выполняется код в теле оператора `do`. В противном случае работа оператора `do-while` завершается. В операторе `while` условие проверяется, в первую очередь, до выполнения кода в теле. Если при первом обращении к оператору `while` условие ложно, то код не будет выполнен никогда.

Рассмотрим те же два примера, что и ранее.

Пример: x в степени y

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

Пример: n!

```
/* Вычисляем n! */
$z=1;  // $z хранит результат $n!
if ($n >1) {
 $i=2;
 do {
 $z=$z*$i;
 $i++;
 }
 while ( $i <= $n );
}
```

5.6. Строки

Работа со строками в PHP очень важна, особенно при решении задач, связанных с Web-сайтами. Строковые значения (последовательности символов) должны быть заключены в двойные или одинарные кавычки, а от вида кавычек зависит, как строка будет отображена.

5.6.1. Двойные и одинарные кавычки

В двойных кавычках помимо обычных видимых и служебных символов могут находиться имена переменных (идентификаторы, начинающиеся с символа `$`), а также теги HTML. При выводе таких строк на экран с помощью оператора `echo` интерпретатор PHP заменяет идентификаторы переменных их значениями, а теги HTML интерпретирует браузер.

Переменные в строках, заключенных в одинарные кавычки, при выводе с помощью оператора `echo` не заменяются значениями, а теги HTML интерпретируются так же, как и в случае двойных кавычек.

> ***Внимание***
>
> При выводе строк оператором `echo` служебные символы, такие как `\n` (перевод строки), `\r` (возврат каретки) и `\t` (табуляция), в браузере не выполняют своей функции, но отображаются в виде пробела. Этого не происходит, если строковое значение заключено в одинарные кавычки. Служебные символы в строках, заключенных в двойные кавычки, выполняют форматирующую роль при работе с некоторыми специальными функциями (см. *разд. 5.6.4*), а также модулем PHP CLI.

На рис. 5.9 показано, как отображаются в браузере строки со служебными символами.

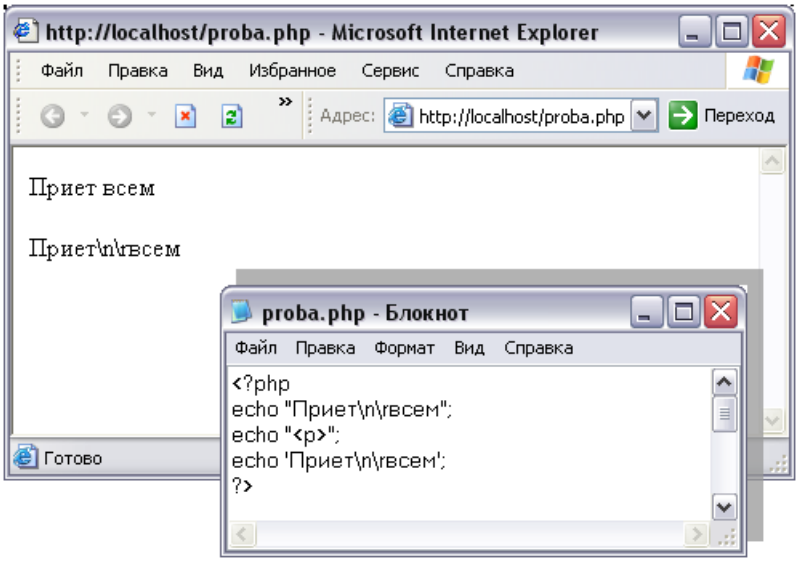

Рис. 5.9. Отображение служебных символов в браузере в зависимости от вида кавычек

Длинные последовательности символов, которые в текстовом редакторе занимают несколько строк, можно задавать с помощью так называемого heredoc-синтаксиса, не используя кавычек:

Переменная=<<<*МЕТКА*

текст

МЕТКА;

Здесь *МЕТКА* — произвольная последовательность букв, цифр и символа подчеркивания, начинающаяся не с цифры.

Пример 1

```
<?php
$x=<<<MYLABEL
        В первых
        строках
        своего письма
MYLABEL;
echo $x;
?>
```

При работе с модулем PHP CLI (с интерфейсом командной строки) текст на экране будет расположен так же, как и между метками в PHP-коде. Однако в окне браузера он отобразится в виде одной или нескольких строк в зависимости от ширины клиентской области браузера. При этом части текста, находящиеся в PHP-коде в различных строках, могут отображаться в одной строке через пробел.

Следующий пример в окне браузера отображается точно так же, как и предыдущий.

Пример 2

```
<?php
$x="
        В первых
        строках
        своего письма
";
echo $x;
?>
```

ВНИМАНИЕ!

При использовании heredoc-синтаксиса необходимо соблюдать следующие два правила: 1) за первым идентификатором метки не должно быть никаких символов, в том числе и пробелов; 2) за заключительным идентификатором метки должен находиться единственный символ точки с запятой.

Чтобы в строке, заключенной в двойные кавычки, отключить интерпретацию таких служебных символов, как двойные кавычки и `$`, перед ними следует указать обратный слэш (`\`). Фигурные скобки не отображаются, если перед открывающей скобкой не поставить обратный слэш.

Строки в одинарных кавычках хранятся так, как они написаны, за исключением пары `\'`, которая имеет вид символа апострофа (одинарная кавычка).

На рис. 5.10 показаны примеры действия обратного слэша, а также сочетаний кавычек различного вида.

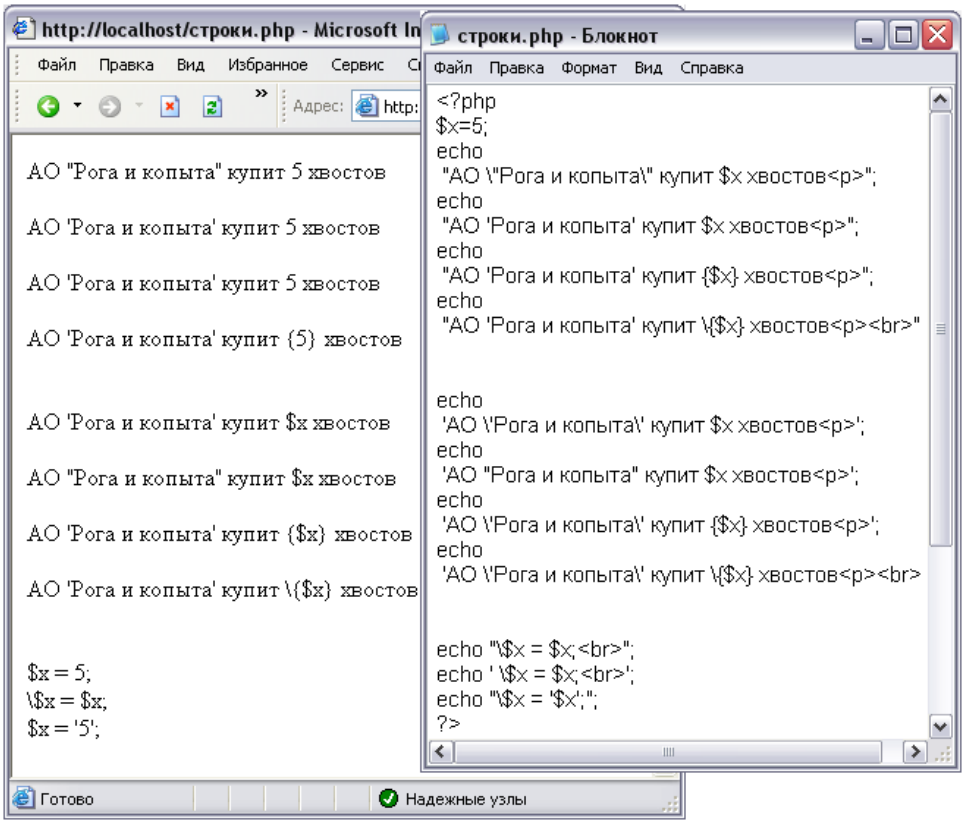

Рис. 5.10. Действие обратного слэша в строках

Как уже отмечалось, в строках могут встречаться теги HTML, которые позволяют отформатировать выводимый текст, а также выполнить другие действия. Интерпретатор PHP сохраняет их без изменений. С помощью оператора `echo` они направляются в выходной документ, который отображается

браузером. Поскольку внутри HTML-кода могут использоваться кавычки различных видов, при формировании строк в PHP следует быть внимательным, чтобы правильно их расставлять и сочетать друг с другом.

5.6.2. Склейка строк

Операция склейки (конкатенации) соединяет две строки и обозначается точкой (.). В результате строка справа от этого символа приписывается без пробелов к концу строки, расположенной слева.

Пример

```
$x="Привет";
$y=$x. " всем";                  // Привет всем
$y="$x всем"." моим друзьям";    // Привет всем моим друзьям
$y.= " и знакомым";              // Привет всем моим друзьям и знакомым
```

В операции склейки могут участвовать не только строки, но и данные других типов. При этом все операнды приводятся к строковому типу автоматически.

Пример

```
$x=5;
$y=$x. " рублей";      // 5 рублей
$y=($x/2). " рубля";   // 2.5 рубля
$z=3>2;                // true
$y=$x.$z;              // 51
```

Обратите внимание на круглые скобки в третьем выражении, задающие порядок вычислений. Без них было бы выведено сообщение об ошибке.

Примечание

В JavaScript склейку строк осуществляет оператор +.

5.6.3. Преобразование строк

Для преобразования символьных строк в PHP имеется много встроенных функций. Здесь мы рассмотрим лишь наиболее важные из них.

- `strlen(строка)` — возвращает длину строки (количество символов), указанной в качестве параметра. При этом служебные символы, такие как пе-

ревод строки, табуляция и другие, также будут подсчитаны. Например, строка "Привет, друзья!" может быть набрана в PHP-коде по-разному, что повлияет на ее длину, хотя все варианты при этом выглядят одинаково.

Пример

```
$x="Привет, друзья! ";
strlen($x);  // возвращает 15

$x="Привет,
друзья";     // возвращает 16
```

Здесь во втором варианте присваивания значения переменной `$x` после запятой присутствует невидимый служебный символ перевода строки, который в действительности состоит из двух символов, а перед словом "друзья" нет ни одного пробела (в первом варианте есть один пробел). Нетрудно подсчитать, что 15 – 1 + 2 = 16. Данное обстоятельство следует учесть, если вам потребуется написать какой-нибудь код анализа текстовых строк.

- `trim(строка)`, `ltrim(строка)`, `rtrim(строка)` — удаляют ведущие и заключительные пробелы, только ведущие и только заключительные пробелы соответственно.
- `strchr(строка, символ)` — возвращает часть строки, начиная с указанного символа и до конца строки. Если строка не найдена, то возвращает `false`.

Пример

```
strchr("BcADaf ag", "a"); // af ag
```

- `stristr(строка, символ)` — возвращает часть строки, начиная с указанного символа и до конца строки. Если строка не найдена, то возвращается `false`. В отличие от `strchr()` регистр строки и символа не учитывается.

Пример

```
stristr("BcADaf ag", "a"); // ADaf ag
```

- `strichr(строка, символ)` — возвращает часть строки, начиная с указанного символа и до конца строки. Если строка не найдена, то возвращается `false`. В отличие от `strchr()` поиск символа ведется от конца строки.

Пример

```
strichr("BcADaf ag", "a"); // ag
```

- `strpos(строка1, строка2)` — возвращает позицию первого вхождения второй строки в первую. Если строка не найдена, то возвращается `false`. Позиции нумеруются слева направо начиная с нуля.

Пример

```
strpos("Привет, мои друзья и мои знакомые", "мои");      // 8
```

- `strrpos(строка1, строка2)` — возвращает позицию последнего вхождения второй строки в первую. Если строка не найдена, то возвращается `false`. Позиции нумеруются слева направо, начиная с нуля.

Пример

```
strrpos("Привет, мои друзья и мои знакомые", "мои");      // 21
```

- `substr(строка, число1, число2)` — возвращает подстроку, начинающуюся с позиции `число1` и длиной `число2`. Позиции нумеруются слева направо, начиная с нуля.

Пример

```
substr("Привет, мои друзья и мои знакомые", 8, 10); //мои друзья
```

- `str_replace(строка1, строка2, строка)` — заменяет в `строка` все вхождения `строка1` на `строка2`. В качестве заменяемой и замещающих строк могут быть не только видимые символы, но и символы табуляции, конца строки и файла.

Пример

```
str_replace("мои", "наши", "Привет, мои друзья и мои знакомые");
 /* Привет, наши друзья и наши знакомые */
```

- `strtr(строка, строка1, строка2)` — заменяет в `строка` каждое вхождение любого символа из `строка1` на соответствующий ему символ в `строка2`. Если длина `строка1` и `строка2` различна, то "лишние" символы в более длинной строке игнорируются.

Пример

```
strtr("Привет, мои друзья", "иуяПрвтдзь","iuaPrvtdzj");
// Privet, moi druzja
```

Здесь второй и третий параметры функции содержат информацию о соответствии символов для "перекодировки" строки, указанной в качестве первого параметра. Так, в примере каждое вхождение буквы "и" заменяется буквой "i", "у" — "u" и т. д.

- `str_repeat(строка, число)` — повторяет указанную строку заданное число раз и возвращает ее.

Пример

```
str_repeat("#",10);      // ##########
```

- `strrev(строка)` — возвращает указанную строку, заменив порядок следования символов на противоположный.

Пример

```
strrev("абвгд");      // дгвба
```

- `strtolower(строка)` — возвращает строку, преобразовав все ее символы в нижний регистр;
- `strtoupper(строка)` — возвращает строку, преобразовав все ее символы в верхний регистр;
- `ucfirst(строка)` – преобразует первый символ указанной строки в верхний регистр и возвращает ее;
- `ucwords(строка)` — преобразует первый символ каждого слова указанной строки в верхний регистр и возвращает ее;
- `str_word_count(строка [, формат])` — возвращает массив слов, входящих в строку, или число слов в строке, если второй параметр не указан. Параметр `формат` может принимать два целочисленных значения, которые определяют вид возвращаемого массива:
  - 1 — числовые индексы элементов массива соответствуют порядковому номеру слова в строке; нумерация начинается с нуля;
  - 2 — числовые индексы элементов массива соответствуют позиции слова в строке; нумерация начинается с нуля.

Пример

```
$x="How do you do";
str_word_count ($x);            // 4
print_r(str_word_count ($x, 1));
```

```
/* Выводит информацию о массиве слов: */
Array (
[0] => How
[1] => do
[2] => you
[3] => do
 )
print_r(str_word_count ($x, 2);
/* Выводит информацию о массиве слов: */
Array (
[0] => How
[4] => do
[7] => you
[11] => do
 )
```

ПРИМЕЧАНИЕ

Массивы подробно рассматриваются в *разд. 5.9*.

- `strtok(строка1, строка2)` — возвращает первую подстроку строки `строка1`, отделенную от остальной части любым из символов-разделителей, указанным в строке `строка1`. Если исходная строка не содержит указанных разделителей, то она возвращается целиком.

Пример

```
<?php
$string = "Это пример\nстроки";
$word=strtok($string, " \n");      // значение $word равно "Это"
/?>
```

Здесь во втором параметре функции `strtok()` указано два разделителя — пробел и перевод строки (`\n`). Разумеется, вы можете указать там и другие символы, например, символ табуляции (`\t`), вертикальную черту (|), запятую (,) и др.

ЗАМЕЧАНИЕ

Функция `strtok()` имеет особенности, которые следует учесть при ее применении: она сохраняет позицию выделенной подстроки и при втором и последующих вызовах ей достаточно передать в качестве единственного параметра только строку разделителей, чтобы разбить всю исходную строку на подстроки.

В следующем примере выводятся все слова (подстроки), которые принципиально можно выделить посредством таких разделителей, как пробел, перевод строки (\n) и табуляция (\t). Обратите внимание, что в исходной строке символ табуляции отсутствует, а потому этот символ во втором параметре функции не производит никакого действия.

Пример

```
<?php
$string = "Это пример\nстроки";
$word = strtok($string, " \n\t");
while ($word) {
 echo "$word<br>";
 $word = strtok(" \n\t");
}
?>
```

В результате выполнения этого кода на экран будет выведен список из трех слов, которые выделил из исходной строки оператор echo. Заметьте, что тег <br>, добавляемый к выводимой строке, обеспечивает лишь вывод обнаруженных слов в отдельных строках. Иначе говоря, он здесь нужен лишь для форматирования вывода.

Примечание

Начиная с версии PHP 2.1.0, пустые подстроки пропускаются функцией strtok().

Итак, функция strtok() берет в качестве параметра некоторую строку и возвращает подстроку в соответствии с заданными разделителями. Повторными вызовами, обычно в цикле (операторы for, while, do-while), с помощью этой функции можно "выудить" все подобные подстроки.

На практике обычно требуется преобразовать строку в массив или наоборот. В PHP для этого используются функции explode() и implode(), которые мы рассмотрим в *разд. 5.9*.

5.6.4. Форматирование строк

Текстовые строки, выводимые в окно браузера оператором echo, часто форматируют с помощью HTML и CSS. Служебные символы \n (перевод строки) и \t (табуляция) также можно использовать в строках. Если они находятся в строке, заключенной в одинарные кавычки, то при выводе этой строки оператором echo эти символы не будут работать по своему назначению и ото-

бразятся на экране. В случае двойных кавычек они, вообще говоря, также не сработают, но будут выглядеть в виде пробелов. Однако если вы хотите все же заставить их функционировать, то строку в двойных кавычках следует начать тегом `<pre>`. Например, следующий оператор выведет сообщение в виде двух строк, причем название фирмы отобразится полужирным шрифтом:

```
echo "<pre><b>АО 'Рога и копыта'<b>\nприглашает на работу главного бухгалтера"
```

Для форматирования можно применять специальные функции PHP `printf()` и `sprintf()`. Если вы используете PHP CLI, работающий в режиме командной строки, то эти функции окажутся единственным средством форматирования строк.

На практике часто приходится форматировать строки, содержащие не только обычные алфавитные символы и знаки препинания, но также числа и значения переменных. При этом числа могут быть представлены в различных системах счисления, а в десятичной системе — в различных видах. В функциях `printf()` и `sprintf()` можно указать шаблон, в соответствии с которым будет отформатирована строка. Эти функции имеют одинаковый синтаксис, но отличаются действием и возвращаемым значением:

- `printf(формат_строка, список_переменных)` — выводит отформатированную строку и возвращает ее длину;
- `sprintf(формат_строка, список_переменных)` — возвращает отформатированную строку (но не выводит ее).

Список переменных — имена переменных, разделенные запятой, значения которых следует подставить в форматируемую строку. Этот параметр не обязателен.

Параметр `формат_строка` представляет собой текстовую строку, которую требуется отформатировать в соответствии с описателями преобразований. Описатели преобразований находятся в форматируемой строке, и каждый из них описывает представление значения соответствующей переменной, указанной в списке: первый описатель для первой переменной, второй — для второй переменной и т. д. Разумеется, строка может и не содержать описателей, тогда она просто не будет отформатирована.

Каждый описатель преобразований состоит из знака процента (`%`), за которым следует один или более дополнительных элементов в следующем порядке: *%ЗаполнительВыравниваниеДлинаТочностьТип*. Рассмотрим эти необязательные элементы.

- *Заполнитель* — символ, который будет использоваться для дополнения результата преобразования до заданной длины. Это может быть пробел

(по умолчанию), 0 или любой другой символ, перед которым необходимо указать одинарную кавычку (').

- *Выравнивание* — символ минус (-); если он указан, то выравнивание происходит по левому краю поля, выделенного для отображения значения, а в противном случае (т. е. по умолчанию) — по правому краю.
- *Длина* — число, определяющее ширину поля, отводимого для вывода результата преобразования. Если результат содержит меньше символов, то оставшееся пространство будет заполнено пробелом или символом заполнения (если он был указан).
- *Точность* — число, перед которым указывается точка. Определяет, сколько десятичных разрядов в дробной части числа следует отображать. Если число имеет больше таких разрядов, то результат получается округлением, а не отбрасыванием лишних знаков. Применяется для чисел с плавающей точкой (`float`).
- *Тип* — символ, определяющий, как трактовать тип значения переменной из списка, которое подставляется в форматируемую строку. Допустимые следующие указатели типа:
  - % — значение переменной не используется;
  - *b* — значение трактуется как целое и выводится в виде двоичного числа;
  - c — значение трактуется как целое и выводится в виде символа с соответствующим кодом ASCII;
  - d — значение трактуется как целое и выводится в виде десятичного числа со знаком;
  - e — значение трактуется как число с плавающей точкой и представляется в экспоненциальной форме;
  - *u* — значение трактуется как целое и выводится в виде десятичного числа без знака;
  - *f* — значение трактуется как число с плавающей точкой и представляется в неэкспоненциальной форме (т. е. только с разделительной точкой);
  - *o* — значение трактуется как целое и выводится в виде восьмеричного числа;
  - *s* — значение трактуется как строка;
  - *x* — значение трактуется как целое и выводится в виде шестнадцатеричного числа в нижнем регистре;
  - *X* — значение трактуется как целое и выводится в виде шестнадцатеричного числа в верхнем регистре.

В листинге 5.6 приведен код, в котором функция `printf()` форматирует строки так, чтобы получился список. Результат выполнения этого кода показан на рис. 5.11.

Листинг 5.6. Пример форматирования строк (вариант 1)

```
<?php
$x1=2700;
$x2=14.378;
$x3=145.2;
$n1="Бендер";
$n2="Балаганов";
$n3="Паниковский";
$format="%'.-12s%'.10.2f руб.\n";
printf("<pre><b>Платежная ведомость</b>\n");
printf($format, $n1, $x1);
printf($format, $n2, $x2);
printf($format, $n3, $x3);
?>
```

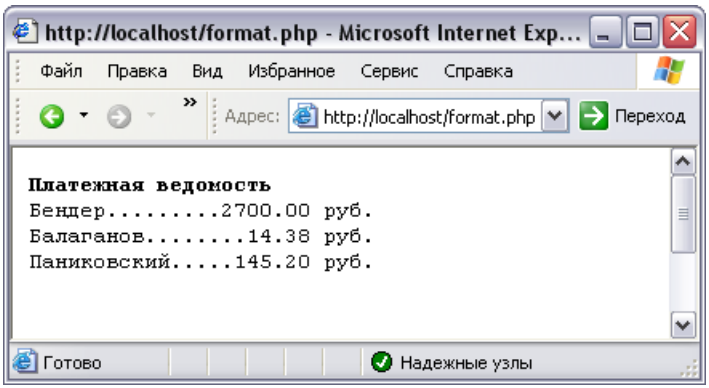

Рис. 5.11. Пример форматирования строк (листинг 5.6)

Тег `<pre>` в данном примере обеспечивает отображение браузером строк в соответствии с их предварительным форматированием. Здесь это понадобилось, чтобы сработали служебные символы `\n` для перехода на другую строку. В качестве заполнителя выбрана точка (`'.`). Фамилии выравнивались по левому краю (-) поля шириной 12 символов с заполнителем в виде точки (`'.`). Таким образом, описатель преобразования для строковых значений имеет вид: `%'.-12s`.

Числа представлялись с десятичной точкой и двумя разрядами после нее. Они выравнивались по правому краю поля из 10 символов, а заполнителем служила точка. Таким образом, описатель преобразования для чисел имеет вид: `%'.10.2f`.

Такой же результат можно получить, используя функцию `sprintf()` для создания отформатированной строки и оператор `echo` — для ее вывода (листинг 5.7).

Листинг 5.7. Пример форматирования строк (вариант 2)

```
<?php
$x1=2700;
$x2=14.378;
$x3=145.2;
$n1="Бендер";
$n2="Балаганов";
$n3="Паниковский";
$format="%'.-12s%'.10.2f руб.\n";
echo
"<pre><b>Платежная ведомость</b>\n".
sprintf($format,$n1, $x1).
sprintf($format,$n2, $x2).
sprintf($format,$n3, $x3);
?>
```

Заметим, что порядок описателей в форматируемой строке соответствует порядку других параметров функций `sprintf()` и `printf()`. Для PHP версии 2.0.6 и старше можно изменить этот порядок и даже использовать один и тот же описатель несколько раз. Для этого в каждом описателе сразу же за символом % следует указать номер, обратный слэш и символ `$` (далее могут располагаться другие элементы). Этот номер соответствует порядковому номеру последующих аргументов функций `sprintf()` и `printf()`. В листинге 5.8 приведен пример кода, выводящего строку "Человек, обезьяна и корова есть млекопитающие".

Листинг 5.8. Пример форматирования с использованием нумерации параметров

```
<?php
$x="человек";
$y="обезьяна";
```

```
$z="корова";
$type="млекопитающие";
echo
 ucfirst(sprintf("%2\$s, %3\$s и %4\$s есть %1\$s", $type, $x, $y, $z));
?>
```

Функция `ucfirst()` необходима здесь для того, чтобы отформатированная строка начиналась с прописной (заглавной) буквы.

5.7. Числа

В PHP числа могут быть только двух типов: целые (`integer`) и с плавающей точкой (`float`). Целые числа не имеют дробной части и не содержат разделительной точки. Числа с плавающей точкой имеют целую и дробную части, разделенные точкой.

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

В PHP возможны операции с числами различных типов, но при этом следует знать, какого типа будет результат. Если результат операции — число с дробной частью, то он представляется числом с плавающей точкой. Если результат оказался без дробной части, то он приводится к целочисленному типу, а не представляется числом, у которого в дробной части одни нули.

Отметим еще одно важное обстоятельство: арифметические операции можно непосредственно выполнять над строками, содержащими числа. Другие, более сложные, вычисления реализуют встроенные функции, которых много PHP. Кроме того, имеются функции, позволяющие отображать числа в нужном виде.

5.7.1. Математические функции

Из большого набора имеющихся в PHP математических функций приведем лишь наиболее употребительные:

- `abs(`*число*`)` — модуль (абсолютное значение) числа;
- `acos(`*число*`)` — арккосинус числа;

- `asin(`*число*`)` — арксинус числа;
- `atan(`*число*`)` — арктангенс числа;
- `atan2(`*x*`, `*y*`)` — величина угла в полярных координатах;
- `ceil(`*число*`)` — округление числа вверх до ближайшего целого;
- `cos(`*число*`)` — косинус числа;
- `exp(`*число*`)` — экспоненциальная функция;
- `floor(`*число*`)` — округление числа вниз до ближайшего целого;
- `fmod(`*число1*`,`*число2*`)` — возвращает дробный остаток от деления первого числа на второе. Этот остаток определяется как число, удовлетворяющее уравнению `число1 = i*число2 + r`, где `i` — некоторое целое. Остаток `r` имеет такой же знак, что и `число1`;
- `log(`*число*`)` — натуральный логарифм числа;
- `log10(`*число*`)` — десятичный логарифм числа;
- `max(`*число1*`, `*число2*`,…,`*числоN*`)` — большее из чисел в списке;
- `max(массив_чисел)` — большее из чисел в массиве;
- `min(`*число1*`, `*число2*`,…,`*числоN*`)` — меньшее из чисел в списке;
- `min(массив_чисел)` — меньшее из чисел в массиве;
- `pow(`*число1*`, `*число2*`)` — *число1* в степени *число2*;
- `rand(`*число1*`, `*число2*`)` — случайное число между заданными числами;
- `round(`*число*`)` — округление числа до ближайшего целого;
- `sin(`*число*`)` — синус числа;
- `sqrt(`*число*`)` — квадратный корень из числа;
- `srand(`*число*`)` — устанавливает начальное значение генератора случайных чисел в указанное число; если параметр не указан, то начальное число выбирается случайно;
- `tan(`*число*`)` — тангенс числа.

5.7.2. Математические константы

В PHP имеется множество предопределенных числовых констант, которые необходимы в инженерных и научных расчетах (табл. 5.8).

Таблица 5.8. *Математические константы*

Имя константы	Значение	Описание
M_PI	3,14159265358979323846	Число π
M_E	2,7182818284590452354	Число e (основание натуральных логарифмов)
M_LOG2E	1,4426950408889634074	$\log_2 e$
M_LOG10E	0,43429448190325182765	lg e
M_LN2	0,69314718055994530942	ln e
M_LN10	2,30258509299404568402	ln 10
M_PI_2	1,57079632679489661923	$\pi/2$
M_PI_4	0,78539816339744830962	$\pi/4$
M_1_PI	0,31830988618379067154	$1/\pi$
M_2_PI	0,63661977236758134308	$2/\pi$
M_SQRTPI	1,77245385090551602729	$\sqrt{\pi}$
M_2_SQRTPI	1,12837916709551257390	$2/\sqrt{\pi}$
M_SQRT2	1,41421356237309504880	$\sqrt{2}$
M_SQRT3	1,73205080756887729352	$\sqrt{3}$
M_SQRT1_2	0,70710678118654752440	$1/\sqrt{2}$
M_LNPI	1,14472988584940017414	ln π
M_EULER	0,57721566490153286061	Постоянная Эйлера

5.7.3. Представление чисел в различных системах счисления

Числа в PHP можно представлять в различных системах счисления, т. е. в системах с различными основаниями: 10 (десятичной), 16 (шестнадцатеричной), 8 (восьмеричной) и 2 (двоичной). Десятичная форма представления чисел наиболее привычна в обиходе. Однако в компьютерных технологиях часто предпочтительнее другие системы: восьмеричная (цифры от 0 до 7), шестнадцатеричная (цифры 0, …, 9, a, b, …, f) и двоичная (цифры 0 и 1). Например, десятичное число 17 в восьмеричной системе представляется как 15, в шестнадцатеричной — b0, а в двоичной — 10001.

Для преобразований чисел из одной системы счисления в другую служат специальные функции:

- `base_convert (строка1, основание1, основание2)` — преобразует строку `строка1`, содержащую число в системе счисления по основанию `основание1`, в строку с числом по основанию `основание2` и возвращает строку, содержащую результат этого преобразования.

Примеры

```
base_convert("17", 10, 16);    // "11"
base_convert("17", 10, 8);     // "21"
base_convert("17", 10, 2);     // "111"
base_convert("ff", 16, 10);    // "255"
base_convert("101", 2, 10);    // "5"
```

Учтите, что данная функция возвращает строку, содержащую число в том или ином представлении, а не число (т. е. данные числового типа). При рассмотрении следующих функций обратите внимание на то, какого типа данные они возвращают и принимают в качестве параметров.

- `dechex(целое_число)` — возвращает строку, содержащую указанное целое десятичное число, в шестнадцатеричном виде.

Примеры

```
dechex(3);        // "3"
dechex(250);      // "fa"
dechex(255);      // "ff"
```

- `hexdec(строка)` — возвращает десятичное число, соответствующее указанному в строковом параметре шестнадцатеричному числу.

Пример

```
hexdec("a");          // 10
```

- `decbin(целое_число)` — возвращает строку, содержащую указанное целое десятичное число, в двоичном виде.

Примеры

```
decbin(3);            // "11"
decbin(250);          // "11111010"
decbin(255);          // "11111111"
```

- `bindec(строка)` — возвращает десятичное число, соответствующее указанному в строковом параметре двоичному числу.

Пример

```
hexdec("101");      // 5
```

- `decoct(целое_число)` — возвращает строку, содержащую указанное целое десятичное число, в восьмеричном виде.

Примеры

```
decoct(3);              // "3"
decoct(250);            // "372"
decoct(255);            // "377"
```

- `octdec(строка)` — возвращает десятичное число, соответствующее указанному в строковом параметре восьмеричному числу.

Пример

```
octdec("10");           // "7"
```

ВНИМАНИЕ!

Во всех перечисленных функциях строковые параметры можно передавать и без кавычек (т. е. в виде литералов).

Представление чисел в различных системах счисления, отличных от десятичной, являются строковыми данными. Арифметические операторы для этих данных действуют так, будто это десятичные числа, независимо от того, принадлежат они к числовому или строковому типу. Иначе говоря, арифметика остается десятичной, независимо от системы счисления операндов.

Примеры

```
decbin(2)+ decbin(3);           // 21 ("10"+"11")
dechex(3)+dechex(16);          // 13 ("3"+"10")
bindec("10")+bindec("11");     // 5 (2+3)
```

5.7.4. Форматирование чисел

Числа в PHP хранятся и выводятся в наиболее эффективном формате. Например, если переменная имеет значение 25,00, оно будет отображено на экране как 25. В приложениях может потребоваться отобразить число в некото-

ром определенном формате, например, разделить группы разрядов пробелом, показать два знака после разделительной точки для представления денежных сумм и т. п. Некоторые возможности форматирования чисел мы уже рассматривали (см. функции `printf()` и `sprintf()` в *разд. 5.6.4*). Здесь мы остановимся на специальной функции форматирования чисел:

```
number_format(число, точность, разделитель1, разделитель2)
```

Эта функция возвращает строку, содержащую отформатированное число, и принимает следующие параметры:

- `число` — число (`float`), которое требуется отформатировать; если следующие параметры отсутствуют, то число будет представлено без дробной части (с округлением до ближайшего большего целого) и запятыми в качестве разделителя групп по три разряда;
- `точность` — целое число, указывающее, сколько знаков в дробной части числа следует показать (по умолчанию нуль); этот параметр обязателен, если вы собираетесь использовать параметры `разделитель1` и `разделитель2`; если вы указываете меньше знаков, чем есть в числе, то происходит округление в бóльшую сторону;
- `разделитель1` — строка с символом, который разделяет целую и дробную части числа (по умолчанию — точка); этот параметр обязателен, если вы собираетесь указывать параметр *разделитель2*;
- `разделитель2` — строка с символом, разделяет группы по три разряда в целой части числа (по умолчанию запятая); этот параметр обязателен, если вы использовали *разделитель1*.

Таким образом, функция `number_format()` может принимать один, два или четыре параметра, но не три. Третий и четвертый параметры могут быть строками из нескольких символов, но разделителями будут лишь первые их символы.

На рис. 5.12 приведены примеры форматирования чисел.

Примечание

В западной традиции в качестве разделителя целой и дробной частей числа принято использовать точку, а для разделения групп разрядов — запятую. В отечественных документах применяют запятую и пробел соответственно.

Внимание!

Математические вычисления следует выполнять до форматирования чисел. Не забывайте, что функция форматирования возвращает текстовую строку и применяется только для вывода чисел в надлежащем виде.

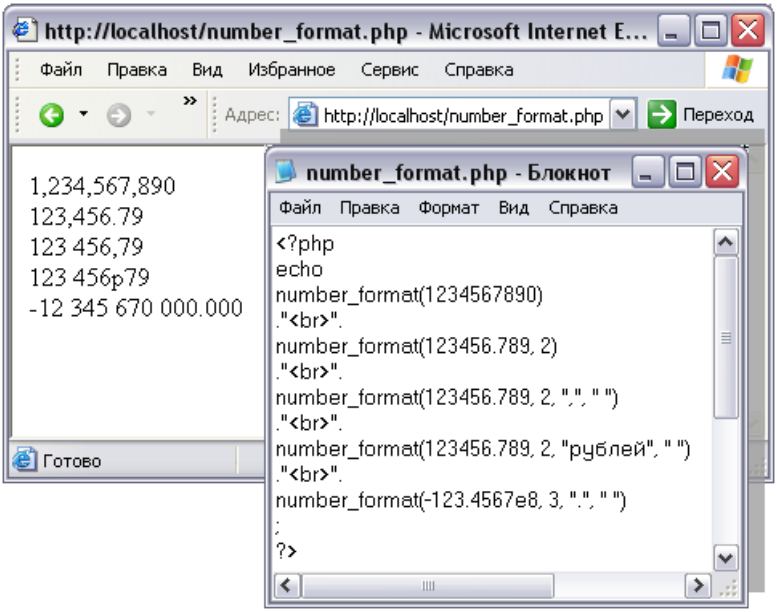

Рис. 5.12. Примеры форматирования чисел с помощью функции `number_format()`

5.8. Дата и время

Дата и время хранятся в компьютере в специальном формате `timestamp` как интервал в секундах с момента 1.01.1970 00:00:00 GMT (или UCT) (см. *разд. 3.12*). В приложениях обычно требуется другое представление даты и времени, для чего в PHP предусмотрены специальные функции.

Текущее системное время в формате `timestamp` (т. е. как целое количество секунд) получается с помощью функции `time()`. Для выдачи в этом же формате любого другого времени служит функция

```
mktime(час, мин, сек, мес, день, год)
```

с целочисленными параметрами, смысл которых ясен.

Например, вызов функции `mktime(12, 49, 0, 2, 9, 2006)` вернет число 1139478540 — число секунд, прошедшее от 1.01.1970 00:00:00 до 9.02. 2006 12:49:00 GMT.

Для представления даты и времени в виде строки в требуемом формате можно использовать функцию

```
date(формат, timestamp).
```

Здесь параметр `timestamp` — целое число, соответствующее интервалу в секундах с момента 1.01.1970 00:00:00 GMT, а `формат` — строка символов форматирования (табл. 5.9) даты и времени.

Таблица 5.9. *Символы форматирования даты и времени*

Символ	Значение
d	День месяца (две цифры, используется ведущий нуль)
j	День месяца (одна или две цифры)
m	Месяц (две цифры, используется ведущий нуль)
n	Месяц (одна или две цифры)
Y	Год (четыре цифры)
y	Год (две цифры)
w	Номер дня недели (0 — воскресенье, 6 — суббота)
g	Часы (12-часовой формат без ведущего нуля, от 1 до 12)
G	Часы (24-часовой формат без ведущего нуля, от 0 до 23)
h	Часы (12-часовой формат с ведущим нулем, от 01 до 12)
H	Часы (24-часовой формат с ведущим нулем, от 00 до 23)
i	Минуты (от 00 до 59)
s	Секунды (от 00 до 59)
a	До или после полудня: am или pm (в нижнем регистре)
A	До или после полудня: AM или PM (в верхнем регистре)
U	Целое число секунд, прошедшее от 1.01.1970 00:00:00 GMT
M	Трехбуквенное сокращение английского названия месяца
F	Английское название месяца
D	Трехбуквенное сокращение английского названия дня недели
l	Английское название дня недели

Между символами форматирования возможны символы-разделители, например, дефис (`-`), точка (`.`), косая черта (`/`), двоеточие (`:`), пробел и др.

Если второй параметр функции `date()` не указан, то предполагается текущее время.

Примеры

```
date("d.m.Y");              // текущая дата, например, 10.02.2006
date("d.m.Y H:i:s", 10000000000);     // 07.09.2014 08:50:08
date("j F.Yг. g:i:s A", 10000000000);/*
                                  7 September.2014г. 8:50:08 AM */
```

Чтобы выполнить какие-либо вычисления над датой или временем, следует сначала перевести их в формат `timestamp`. В следующем примере выводится отформатированная дата и время, соответствующие 12 часам 49 минутам 0 секундам пяти суткам спустя после 9 февраля 2006 г.:

```
<?php
$t=mktime(12, 49, 0, 2, 9, 2006) + 5*24*60*60;
echo date("d.m.Y H:i:s", $t);      // 14.02.2006 12:49:00
?>
```

Для перевода текстовых строк, содержащих дату и время, в формат `timestamp` служит функция

```
strtotime(дата-время, timestamp).
```

Эта функция принимает строку, которая содержит ключевые слова, связанные с датой и временем и необязательный целочисленный параметр. Если он указан, то функция возвращает число секунд относительно его значения, а в противном случае — относительно текущего времени. При возникновении ошибки преобразования функция возвращает –1.

В строковом параметре `дата-время` функция `strtotime()` распознает следующие ключевые слова на английском языке:

- названия месяцев и их трехбуквенные сокращения;
- названия дней недели и их трехбуквенные сокращения;
- названия элементов даты и времени: year (год), month (месяц), week (неделя), day (день), hour (час), minute (минута), second (секунда), am (до полудня), pm (после полудня). Если требуются слова во множественном числе (что не обязательно), то следует добавить окончание s, например, days, weeks и т. п.;
- слова: ago (тому назад), now (сейчас), last (последний), next (следующий), this (этот), tomorrow (завтра), yesterday (вчера);
- числа и знаки "плюс" (+) и "минус" (–) ;
- временные зоны: например gmt (Greenwich Mean Time — среднее время по Гринвичу), pdt (Pacific Daylight Time — дневное тихоокеанское время).

Далее приведены примеры ключевых слов для формирования значения параметра дата-время функции `strtotime()`:

- now — текущее время;
- now + 48 hours — через 48 часов от текущего времени; здесь слово now опустить;
- + 48 hours — через 48 часов от текущего времени;
- 10am + 5 days — 10 часов до полудня через 5 дней;
- 3 months ago — 3 месяца назад;
- tomorrow 5pm — 5 часов по полудни завтра;
- last Friday — предыдущая пятница;
- next year gmt — один год спустя;
- 2006/4/12 — 12 апреля 2006г. 00:00:00;
- 2006-4-12 — то же самое;
- 10 Apr 2006 — то же самое.

Предположим, необходимо вывести дату и время, соответствующие 6 часам пополудни 3 месяца и 2 дня спустя. Это можно сделать с помощью следующего кода.

Пример

```
<?php
$timestamp="d.m.Y H:i:s";
$t=strtotime("+ 3 months + 2 days 6pm");
echo date($timestamp, $t);
?>
```

Допустим, требуется узнать, сколько часов между двумя моментами времени, заданными как строковые значения, например, `"9 Feb 2006 5am"` и `"2 Mar 2006 3pm"`. Для этого следует перевести строковые значения даты и времени в формат `timestamp` с помощью функции `strtotime()`, найти разницу между ними и разделить ее на 3600.

Пример

```
<?php
$t1=strtotime("9 Feb 2006 5am");
$t2=strtotime("8 Mar 2006 3pm");
echo ($t2-$t1)/3600;     // 658
?>
```

5.9. Массивы

Как уже неоднократно упоминалось, массив представляет собой упорядоченный набор данных (элементов), объединенных под общим именем. Обращение к элементам массива происходит по индексу, который может быть как числовым, так и символьным.

5.9.1. Создание массива

Массив можно создать несколькими способами. Самый простой из них состоит в том, чтобы рядом с именем переменной поставить квадратные скобки и с помощью оператора присваивания задать значение элемента:

```
$myarray[]="Иван";
```

В результате будет создан массив с именем `$myarray` и единственным элементом `"Иван"`. К этому массиву можно добавить другие элементы различных типов, используя тот же синтаксис:

```
$myarray[]="Федоров";
$myarray[]=125.7;
```

В рассматриваемом случае массив `$myarray` будет содержать уже три элемента: два строковых и один числовой.

При создании массива таким способом PHP автоматически индексирует элементы, используя для этого целые числа, начиная с нуля. Индексы (ключи) позволяют получить доступ к значениям элементов массива:

```
$myarray[0];     // "Иван"
$myarray[1];     // "Федор"
$myarray[2];     // 125.7
```

При создании массива можно сразу индексировать элементы по своему усмотрению, причем не обязательно с нуля и по порядку:

```
$myarray[1]= "Иван";
$myarray[2]= "Федоров";
$myarray[3]=125.7;
```

Кроме числовых индексов массивы могут иметь символьные (строковые) индексы:

```
$myarray['Имя']= "Иван";
$myarray['Фамилия']= "Федоров";
$myarray['Зарплата']=125.7;
```

> ***Примечание***
>
> Числовые индексы могут быть представлены и соответствующими строками. Например, индексы 3 и "3" соответствуют одному и тому же элементу массива, а "03" — другому.

Массив также можно создать с помощью следующей конструкции:

```
$имя_массива=array([индекс1=>]значение1[,[индекс2=>]значение2,…])
```

Здесь квадратные скобки указывают лишь на то, что заключенный в них элемент не является обязательным.

Примеры

```
$myarray=array("Иван", "Петров", 12.8);
$myarray=array('Имя'=>"Иван",'Фамилия'=>"Петров",'Зарплата'=>12.8);
$myarray=array('Имя'=>"Иван", 2=> "Петров", 3=>12.8);
```

В первом случае элементы массива автоматически получат числовые индексы, во втором — указанные символьные индексы, а в третьем — символьные и числовые. При создании массива возможны индексы различных типов. Однако каждый элемент имеет только один индекс — числовой или символьный. Если какой-то элемент имеет символьный индекс, то обращаться к этому элементу с помощью его порядкового номера в массиве нельзя.

Создать регулярные массивы можно с помощью функции диапазона:

```
range (начало, конец, шаг),
```

которая принимает в качестве параметров:

- `начало` — число или символ начала диапазона;
- `конец` — число или символ конца диапазона;
- `шаг` — число, соответствующее приращению значений элементов массива (необязательный параметр, по умолчанию равный единице).

Примеры

```
range(20, 25);     // массив из элементов: 20, 21, 22, 23, 24, 25
range(20, 25, 2);     // массив из элементов: 20, , 22, 24
range(25, 22, 2);     // массив из элементов: 25, 23, 21
range(20, 25, 1.5);   // массив из элементов: 20, 21.5, 23, 24.5
range("a", "d");       // массив из элементов: "a", "b", "c" , "d"
range("a", "d", 2);    // массив из элементов: "a", "c"
range("d", "a", 2);         // массив из элементов: "d", "b"
range("Борис", "Дмитрий"); // массив из элементов: "Б", "В", "Г", "Д"
```

Количество элементов (длину) массива можно определить с помощью функций `count(массив)` или `sizeof(массив)`.

Пример

```
$myarray=array("Иван", "Федоров", 125.7);
$n=count($myarray);      // 3
```

Независимо от способа создания массива значения его элементов всегда можно изменить с помощью оператора присваивания.

Добавить новый элемент к существующему массиву можно, написав следующие выражения:

- `$имя_массива[]=значение` — добавляется новый элемент с числовым индексом;
- `$имя_массива[индекс]=значение` — если в массиве нет элемента с указанным индексом, то элемент добавляется, в противном случае — изменяется (перезаписывается).

Пример

```
$myarray=array("Имя"=>"Иван",125.7); /* Структура массива:
                                     ["Имя"]=>"Иван"
                                     [0]=>125.7     */

$myarray[]="Профессор";              /* Структура массива:
                                     ["Имя"]=>"Иван"
                                     [0]=>125.7
                                     [1]=> "Профессор" */

$myarray["тел"]="123-4567";          /* Структура массива:
                                     ["Имя"]=>"Иван"
                                     [0]=>125.7
                                     [1]=> "Профессор"
                                     ["тел"]=>”123-4567” */

$myarray[0]="Федоров";               /* Структура массива:
                                     ["Имя"]=>"Иван"
                                     [0]=> "Федоров"
                                     [1]=> "Профессор"
                                     ["тел"]=>"123-4567" */
```

Удалить элемент из массива или массив целиком можно функцией `unset()` с параметром, указывающим на элемент массива или сам массив соответственно.

Пример

```
$myarray=array("Один", "Два", "Три");   // массив из трех элементов
unset($myarray[1]);                     /* удаление 2-го элемента; $myarray теперь
                                        содержит два элемента: "Один" и "Три" с
                                        индексами 0 и 2 соответственно */
unset($myarray);   // переменная $myarray теперь имеет значение null
```

Внимание!

При удалении элементов из массива оставшиеся в нем элементы сохраняют свои прежние индексы. Иначе говоря, переиндексация массива (даже только с числовыми индексами) не происходит.

Как отмечалось ранее, обращение к элементам массива происходит с помощью индексов. Однако при обращении к несуществующему элементу выводится сообщение об ошибке. В ряде случаев это нежелательно. Чтобы подавить возможное сообщение об ошибке, используют символ `@`, например,

```
@$anyelement=$myarray[5];
```

5.9.2. Многомерные массивы

Рассмотренные ранее массивы одномерные. Однако элементы массива могут содержать данные различных типов, в том числе и массивы. Если в качестве элементов некоторого одномерного массива создать массивы, то получится двумерный массив. Обращение к элементам такого массива происходит в соответствии со следующим синтаксисом:

имя_массива[*индекс_уровня1*] [*индекс_уровня2*]

Если размерность массива больше двух, то обращение к массивам имеет аналогичный синтаксис: следует добавить необходимое число квадратных скобок, заключающих нужные индексы.

Пример

```
/* Двумерный массив */
$сотрудники=array(
     array("Иван", "Иванов", 100),
     array("Василий", "Васильев", 200),
     array("Федор", "Федоров", 300)
);
echo $сотрудники[2][1];          // Федоров
```

Массив, приведенный в данном примере, можно представить в виде таблицы из трех столбцов и трех строк. Каждой строке соответствует массив значений, который сам является элементом массива `$сотрудники`.

В листинге 5.9 приводится сценарий, выводящий элементы двумерного массива в виде таблицы (рис. 5.13).

Листинг 5.9. Вывод таблицы с элементами двумерного массива (вариант 1)

```
<?php
$сотрудники=array(
array("Иван", "Иванов", 100),
array("Василий", "Васильев", 200),
array("Федор", "Федоров", 300)
);

$str="<table border=1>";
for ($i=0; $i<count($сотрудники); $i++) {
     $str.="<tr>";
     for ($j=0; $j<count($сотрудники[$j]); $j++){
         $str.="<td>".$сотрудники[$i][$j]."</td>";
     }
     $str.="</tr>";
}
$str.="</table>";
echo $str;
?>
```

Другой вариант сценария мы рассмотрим в следующем разделе.

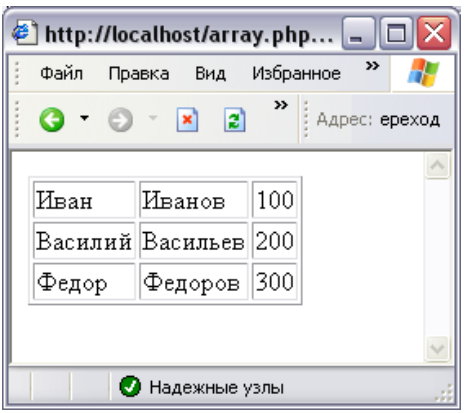

Рис. 5.13. Пример представления двумерного массива в виде таблицы (листинг 5.9)

5.9.3. Отображение массивов

Для просмотра структуры и значений элементов массива (это может понадобиться при отладке программ) используются функции `print_r()` и `var_dump()`, достаточно подробно описанные в *разд. 5.3.4*.

Разумеется, для вывода значений элементов массива вы можете воспользоваться оператором цикла.

Пример

```
<?php
$myarray=array("Иван","Федоров", 125.7);
$i=0;
while ($i<count($myarray)) {
     echo $myarray[$i],"<br>";
     $i++;
}
?>
```

Вывод элемента массива с помощью оператора `echo` не отличается от обычной переменной, например,

```
echo $myarray[2];
```

Однако если требуется вывести элемент массива в составе текстовой строки, то его следует заключить в фигурные скобки.

Пример

```
$myarray=array("Иван", "Федоров", 125.7);
echo "Сотрудник {$myarray[1]} получает {$myarray[2]} руб."; /* вывод
строки:
 "Сотрудник Федоров получает 125.7 руб. " */
```

Используя фигурные скобки, код листинга 5.9 можно переписать так, как показано в листинге 5.10.

Листинг 5.10. Вывод таблицы с элементами двумерного массива (вариант 2)

```
<?php
$сотрудники=array(
array("Иван", "Иванов", 100),
array("Василий", "Васильев", 200),
```

```
array("Федор", "Федоров", 300)
);

$str="<table border=1>";
for ($i=0; $i<count($сотрудники); $i++) {
     $str.="<tr>";
     for ($j=0; $j<count($сотрудники[$j]); $j++){
         $str.="<td>{$сотрудники[$i][$j]}</td>";
     }
     $str.="</tr>";
}
$str.="</table>";
echo $str;
?>
```

5.9.4. Операции над массивами

Копирование массивов

Чтобы скопировать массив, достаточно содержащую его переменную присвоить другой переменной. Изменение значения элемента в одном из массивов никак не отразится на соответствующем элементе в другом.

Пример

```
$a=array(5, 2, 4, 3);
$x=$a;
$a[2]=25;
echo $x[2];      // 4
```

Примечание

В JavaScript присвоение переменной типа "Массив" другой переменной не приводит к копированию массива, а создает лишь новую ссылку (псевдоним) на тот же самый массив.

Сортировка массивов

Упорядочивать (сортировать) элементы массива можно по индексу и по значению. В PHP существует много функций, позволяющих осуществить сортировку массивов различными способами. Здесь мы рассмотрим только наиболее употребительные из них.

Для сортировки элементов массива с числовыми индексами по возрастанию значений служит функция

```
sort(массив [, режим]),
```

Здесь квадратные скобки указывают, что параметр не обязателен.

Функция `sort()` возвращает `true` или `false` соответственно при удачном и ошибочном завершении операции. Данная функция изменяет указанный массив (меняет индексы элементов в соответствии с новым порядком их расположения). Если массив имеет символьные индексы, то они заменяются числовыми. Второй необязательный параметр задает режим сортировки (способ сравнения значений элементов) и может принимать следующие значения:

- `SORT_REGULAR` — сравнивать элементы, не изменяя их типы (по умолчанию);
- `SORT_NUMERIC` — сравнивать элементы как числа;
- `SORT_STRING` — сравнивать элементы как текстовые строки;
- `SORT_LOCALE_STRING` — сравнивать элементы как строки с учетом установленной локали (начиная с версии PHP 5.0.2).

Перечисленные константы имеют целочисленные значения 0, 1, 2 и 3 соответственно.

Пример

```
$myarray=array("Вася", "Анна", 5);      /* Структура массива:
                                        [0]=>"Вася"
                                        [1]=>"Анна"
                                        [2]=> 5 */
sort($myarray);                         /* Структура массива:
                                        [0]=> "Анна"
                                        [1]=> "Вася"
                                        [2]=>5*/
$myarray=array("Вася", "Анна", 5);      /* Исходный массив */
sort($myarray, SORT_STRING );           /* Структура массива:
                                        [0]=>5
                                        [1]=>"Анна"
                                        [2]=> "Вася"*/
```

При использовании функции `sort()` исходная связь "индекс — значение" теряется: индексы элементов в отсортированном массиве изменяются в соот-

ветствии с их новым положением. Это бывает нежелательно, особенно в случае массивов с символьными индексами. Сортировка, при которой эта связь сохраняется, может быть выполнена с помощью функции

```
asort(массив [, режим]),
```

которая по возвращаемой величине и параметрам аналогична функции `sort()`, но отличается своим действием: элементы массива упорядочиваются по возрастанию значений, но индексы сохраняются прежними.

Пример

```
$myarray=array("Вася", "Анна", 5);      /* Структура массива:
                                        [0]=>"Вася"
                                        [1]=>"Анна"
                                        [2]=> 5 */
asort($myarray);                        /* Структура массива:
                                        [1]=> "Анна"
                                        [0]=> "Вася"
                                        [2]=>5 */
$myarray=array("Вася", "Анна", 5);      /* Исходный массив */
sort($myarray, SORT_STRING );           /* Структура массива:
                                        [2]=>5
                                        [1]=>"Анна"
                                        [0]=> "Вася"*/
```

Сортировка массива по значениям элементов с сохранением прежних индексов может понадобиться при отображении элементов с прежним способом обращения к ним.

Рассмотренные функции `sort()` и `asort()` упорядочивают элементы массива по возрастанию их значений. Для сортировки по убыванию служат соответственно функции `rsort()` и `arsort()`.

Сортировать элементы массива можно не только по их значениям, но и по индексам. Как вы уже заметили, порядок расположения элементов в массиве, задаваемый при его создании или последующей сортировке, не обязательно соответствует порядку индексов элементов. Для сортировки массива по индексам служат функции:

- `ksort()` — сортирует элементы массива по возрастанию значений индекса;
- `krsort()` — сортирует элементы массива по убыванию значений индекса.

Пример

```
$myarray=array(
        "name"=>"Иван",
        "family"=>"Федоров",
        "bonus"=>125.7
);
ksort($myarray);       /* Структура массива:
            ["bonus"]=>125.7
            ["family"]=>"Федоров"
            ["name"]=>"Иван" */
```

Функции `ksort()` и `krsort()` имеют такие же параметры и возвращают такие же значения, что и рассмотренные ранее `sort()` и `assort()`. Обратите внимание, что применение функций `ksort()` и `krsort()` изменяет порядок расположения элементов в массиве, но сохраняет исходную связь "индекс — значение".

> ***ВНИМАНИЕ!***
>
> Функции сортировки изменяют исходный массив, а не возвращают его отсортированный вариант. Если вам необходимо сохранить исходный массив, то для этого следует предварительно сделать его копию.

Перемещение по массиву

Нередко требуется "пробежаться" по всем элементам массива, чтобы выполнить какие-нибудь операции над ними. Это можно сделать с помощью обычных операторов цикла `while`, `do-while` и `for` (см. *разд. 5.5.9*). Вместе с тем для этой цели в PHP существует специальная очень удобная конструкция `foreach (для каждого)`, которая имеет две синтаксические формы:

- `foreach (массив as $значение) {выражения};`
- `foreach (массив as $индекс=>$значение) {выражения};`

Здесь `$значение` — переменная, выбираемая по своему усмотрению для представления значения текущего элемента массива; `$индекс` — переменная для представления значения текущего индекса элемента массива. Слово `as` в выражении, указанном в круглых скобках, является ключевым.

Конструкция `foreach` последовательно перебирает все элементы указанного ей массива, присваивая их значения переменной `$значение`, а индексы (во втором варианте синтаксиса) — переменной `$индекс`. Эти переменные дос-

тупны в коде, указанном в фигурных скобках. Код, работая с данными переменными, имеет дело с текущим на данной итерации элементом массива.

В листинге 5.11 показано применение `foreach` в двух своих формах для вывода элементов массива (рис. 5.14).

Листинг 5.11. Применение оператора `foreach`

```
<?php
$сотрудник=array(
      "Имя"=>"Иван",
      "Фамилия"=>"Иванов",
      "Зарплата"=>10000
);

foreach($сотрудник as $value){
      echo "$value<br>";
}
echo "<p>";
foreach($сотрудник as $key=>$value){
      echo "$key - $value<br>";
}
?>
```

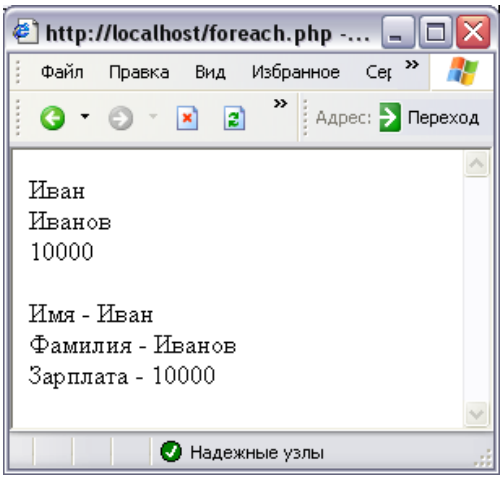

Рис. 5.14. Пример использования оператора `foreach`

Перебор элементов массива можно выполнить и с помощью специальных функций, которые реализуют перемещение указателя на текущий элемент массива:

- `current(массив)` — возвращает текущее значение элемента массива, не перемещая указатель;
- `next(массив)` — перемещает указатель на следующий элемент массива;
- `prev(массив)` — перемещает указатель на предыдущий элемент массива;
- `end(массив)` — перемещает указатель на последний элемент массива;
- `reset(массив)` — перемещает указатель на начальный элемент массива.

Если к массиву не применялась функция перемещения указателя, то считается, что он установлен на первый (начальный) элемент.

Пример

```
$myarray=array("name"=>"Иван","family"=>"Федоров", "bonus"=>125.7);
$x=current($myarray);        // Иван
$x=next($myarray);           // Федоров
$x=reset($myarray);          // Иван
```

Все перечисленные функции перемещения указателя возвращают значение элемента массива. Однако применение функций `prev()` и `next()` может вывести указатель за пределы массива, и тогда они вернут логическое значение `false`, а не значение элемента массива. Поэтому эти функции обычно используются совместно с проверкой выхода за границы массива.

Пример

```
$x=next($myarray);                // переход к следующему элементу массива
if ($x===false) {                 // если выход за конец массива
echo "Выход за конец массива";    // вывод предупреждения
} else {
echo $x;                          // вывод значения элемента массива
}
```

Обратите внимание, что здесь и в других подобных случаях необходим оператор жесткого равенства, поскольку элемент массива, возвращаемый функцией `next()`, может, вообще говоря, иметь значения null, "", 0, "0" и "false", но это еще не повод остановить обработку. Оператор жесткого равенства учитывает типы сравниваемых значений (см. *разд. 5.6.5*).

Запись значений элементов массива в переменные

Чтобы значения элементов массива с числовыми индексами, начинающимися с нуля, присвоить переменным из заданного списка, можно применить следующую конструкцию:

```
list(список_переменных) = массив;
```

При этом создаются переменные, указанные в списке через запятую, и им присваиваются элементы массива с числовыми индексами: первой переменной присваивается значение элемента с индексом 0, второй переменной — значение элемента с индексом 1 и т. д.

Пример

```
$myarray=array("a", "b", 5, 10);
list($x,$y)=$myarray;
var_dump($x, $y);     // вывод значений и типов переменных $x и $y
```

> ***Примечание***
>
> Конструкция `list()` по своему синтаксису похожа на вызов функции, но отличается от него тем, что может быть записана слева от оператора присваивания. Однако выражению с вызовом настоящей функции нельзя присвоить какое бы то ни было значение. Иначе говоря, вызов функции может встречаться только справа от оператора присваивания.

В списке переменных допустимы элементы массива, например,

```
list($b[0], $b[1], $b[2]) = $a;
```

В этом случае происходит копирование значений элементов массива из `$a` в `$b`. Однако данная операция осуществляется, начиная с крайнего правого элемента массива `$a`, в результате чего порядок расположения элементов в массиве `$b` будет противоположен порядку следования элементов в массиве `$a`, хотя индексы элементов сохраняются.

Пример

```
$a=array("a", "b", 5, 10);
list($b[0], $b[1], $b[2])=$a;
var_dump($b);     /* Структура массива $b:
                  [2]=> 5
                  [1]=> "b"
                  [0]=> "a" */
```

Преобразование массива в текстовую строку

Чтобы значения элементов массива объединить в их строковом представлении в одну текстовую строку, можно использовать функцию

`implode(разделитель, массив)`

Параметр `разделитель` является строкой символов, которая вставляется между элементами массива при их объединении в текстовую строку. Если этот параметр не указан, то предполагается пустая строка "". Данная функция возвращает строку, полученную в результате преобразования.

Пример

```
$myarray=array("Один", "Два", "Три");
$str=implode($myarray);          // строка "ОдинДваТри"
$str=implode("-", $myarray);     // строка "Один-Два-Три"
```

Преобразование текстовой строки в массив

Для записи фрагментов текстовой строки в элементы массива служит функция `explode(разделитель, строка [, количество])`.

Параметр `разделитель` — это строка символов, которая рассматривается как разделитель фрагментов строки. Необязательный параметр (о чем говорят квадратные скобки) `количество` указывает максимальное количество элементов в возвращаемом массиве (при этом последний элемент будет содержать весь остаток строки).

Если `разделитель` не присутствует в строке, то функция возвращает массив, содержащий единственный элемент, значением которого является исходная строка. Если `разделитель` пустая строка, то функция возвращает логическое значение `false`.

Пример

```
$str="Один Два Три";
$myarray=explode(" ",$str);      /* Структура массива:
                                 [0]=>"Один"
                                 [1]=> "Два"
                                 [2]=> "Три" */

$myarray=explode("Два",$str);      /* Структура массива:
                                 [0]=>"Один"
                                 [1]=> " Три" */
```

Другие операции над массивами

Здесь перечислены еще несколько полезных функций для работы с массивами и приведены примеры. Подробное их описание можно найти в справочном руководстве по PHP.

- `array_unique(массив)` — возвращает массив, полученный из указанного в качестве параметра путем удаления из него повторяющихся элементов;

Пример

```
$myarray=array("a", "b", "a", "c", 5, 5);
$x=array_unique($myarray);      /* Структура массива:
                                [0]=> "a"
                                [1]=> "b"
                                [3]=> "c"
                                [4]=>5 */
```

- `array_sum(массив)` — возвращает сумму значений элементов массива, указанного в качестве параметра;

Пример

```
$myarray=array(1, 2, "3");
$x=array_sum($myarray);     // 6
```

- `array_slice(массив, n1, n2)` — возвращает массив, являющийся частью исходного массива, указанного в качестве параметра. В выходной массив попадет `n2` элементов исходного, начиная с элемента с номером `n1`;

Пример

```
$myarray=array(1, 2, 3, 4, 5, 6);
$x=array_slice($myarray, 2, 3);      /* Структура массива:
                                    [0]=>3
                                    [1]=> 4
                                    [2]=> 5 */
$x=array_slice($myarray, -2, 3);    /* Структура массива:
                                    [0]=>5
                                    [1]=> 6 */
```

- `array_merge(массив1, массив2,…, массивN)` — возвращает массив, получающийся в результате объединения двух или более массивов, указанных в качестве параметров;

Пример

```
$a=array("a", "b", "c");
$b=array("a", "x", "y");
$x=array_merge($a, $b);      /* Структура массива:
                             [0]=> "a"
                             [1]=> "b"
                             [2]=> "c"
                             [3]=> "a"
                             [4]=> "x"
                             [5]=> "y" */
```

- `array_diff(массив1, массив2,…, массивN)` — возвращает массив, состоящий из тех элементов массива, указанного в качестве первого параметра, которых нет в остальных массивах;

Пример

```
$a=array("a", "b", "c");
$b=array("a", "x", "y");
$x=array_diff($a, $b);          /* Структура массива:
                                [1]=>"b"
                                [2]=> "c" */
```

- `array_flip(массив)` — возвращает массив, получающийся из указанного в качестве параметра путем перестановки местами индексов и значений;

Пример

```
$a=array("a", "b", "c");
$x=array_flip($a);          /* Структура массива:
                            ["a"]=>0
                            ["b"]=>1
                            ["c"]=>2 */
$a=array("Планета"=>"Земля", "Радиус"=>6378);
$x=array_flip($a);          /* Структура массива:
                            ["Земля"]=>"Планета"
                            [6378]=>"Радиус" */
```

- `extract(массив)` — создает переменные из символьных индексов массива и присваивает им значения соответствующих элементов массива; возвращает число созданных переменных;

```
$a=array("Планета"=>"Земля", "Радиус"=>6378);
$x=extract($a);
echo "Наша планета называется $Планета. Ее радиус равен $Радиус км";
```

- `compact(список_имен_переменных)` — возвращает массив, созданный из указанных переменных; действие этой функции противоположно действию `extract()`.

Пример

```
$name="Иван";
$lastname="Федоров";
$bonus=1234;
$x=compact("name", "lastname","bonus");        /* Структура массива:
                                                ["name"]=>"Иван"
                                                ["lastname"]=>"Федоров"
                                                ["bonus"]=> 1234 */
```

5.10. Глобальные предопределенные переменные

В PHP имеются предопределенные переменные типа `array` (массив) с глобальной областью видимости. Их еще называют автоглобальными или суперглобальными массивами, поскольку они доступны из любого сценария на языке PHP. Далее приведен их список:

- `$GLOBALS` — содержит все глобальные переменные (как предопределенные, так и созданные программистом). Имена глобальных переменных в массиве `$GLOBALS` являются его символьными индексами (ключами). Поэтому, чтобы получить доступ к глобальной переменной, например, `$myvar`, достаточно использовать выражение `$GLOBALS["myvar"]`. Обратите внимание, что имя переменной в качестве индекса пишется без символа `$`;
- `$_GET` — содержит данные, переданные в сценарий на PHP в виде части URL-адреса. Данные HTML-форм, переданные методом `GET`, также сохраняются в массиве `$_GET`;
- `$_POST` — содержит данные, переданные в сценарий на PHP из HTML-форм методом `POST`;

- $_COOKIE — содержит данные, переданные в текущий сценарий через механизм cookie;
- $_ENV — содержит переменные окружения, такие как название операционной системы, системный диск и др.; содержимое этого массива зависит от операционной системы;
- $_FILES — содержит имена файлов, загруженных методом POST с помощью браузера;
- $_SERVER — содержит переменные, установленные Web-сервером, либо непосредственно связанные с окружением выполнения текущего сценария. Эта информация зависит от того, какой Web-сервер используется. Более того, если PHP-сценарий запущен из командной строки, то некоторые переменные заведомо будут недоступными. Таким образом, состав переменных в данном массиве может быть различным. Например, переменная DOCUMENT_ROOT содержит полное имя каталога, в котором Web-сервер ищет запрашиваемые Web-страницы; PHP_SELF — имя файла текущего сценария; REQUEST_METHOD — название метода, который задан для доступа к странице: "GET", "POST", "HEAD", "PUT";
- $_SESSION — содержит все переменные сеанса, доступные в текущем сценарии;
- $_REQUEST — содержит все переменные, находящиеся в массивах $_GET, $_POST и $_COOKIE.

Для просмотра содержимого глобальных массивов можно воспользоваться функцией var_dump(), например, var_dump($_SERVER) или конструкцией foreach. Например, просмотреть массив $_SERVER можно с помощью такого сценария:

```
foreach($_SERVER as $var=>$value){
echo "$var = $value <br>";
}
```

Если, например, серверному сценарию из HTML-формы было передано методом POST содержимое элемента с атрибутом name="userdata", то его можно получить в сценарии как $_POST['userdata'].

Подробное описание содержимого глобальных массивов можно найти в справочном руководстве по PHP. Применение некоторых из них будет рассмотрено в *главе 6*.

> ***Внимание!***
>
> По умолчанию глобальные массивы доступны, когда в конфигурационном файле php.ini имеется установка register_globals = off. Если заменить ее на register_globals = on, то глобальные массивы будут недоступными.

Примечание

В более ранних версиях PHP существовали глобальные массивы с длинными именами: `$HTTP_GET_VARS`, `$HTTP_POST_VARS`, `$HTTP_COOKIE_VARS`, `$HTTP_ENV_VARS`, `$HTTP_FILES_VARS`, `$HTTP_SERVER_VARS`, `$HTTP_SESSION_VARS`. В PHP 5 они также доступны, хотя и не рекомендуются к использованию. Вместе с тем установка `register_long_arrays = off` в конфигурационном файле php.ini позволяет их отключить. Если же вы задействуете старые сценарии, то оставьте установку `register_long_arrays = off`.

5.11. Функции

Функция — это конструкция, которая позволяет оформить блок программного кода для многократного использования. Такой блок кода (тело функции) можно озаглавить, назначив ему имя, и указать при необходимости параметры (значения переменных и выражений), с которыми он будет работать. Тело функции можно построить так, чтобы она возвращала некоторое значение в ту программу, из которой была вызвана. Впрочем, функция в PHP (как и в JavaScript) может и не требовать параметров, а также ничего не возвращать. Вызов функции (имя функции с возможным указанием передаваемых ей параметров) возможен в простых и сложных выражениях языка PHP, если эта функция что-нибудь возвращает.

5.11.1. Пользовательские функции

Пользовательские функции — это функции, которые определяются программистом. Определение (описание) функции начинается ключевым словом `function`. Точнее, описание функции имеет следующий синтаксис:

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

Имя функции выбирают так же, как и имя переменной, за исключением того, что первым символом в имени функции не может быть `$`. Ключевые слова языка PHP в имени функции тоже недопустимы. За именем функции обязательно следует пара круглых скобок. Программный код (тело) функции заключают в фигурные скобки. Они определяют группу выражений, которые относятся к коду именно этой функции. Если функция принимает параметры, то список их имен (идентификаторов) указывают в круглых скобках около имени функции. Имена параметров выбирают аналогично именам обычных переменных. Если параметров несколько, то в списке их разделяют запятыми. Если параметры для данной функции не предусмотрены, то в круглых скобках около имени функции ничего не пишут.

Если требуется, чтобы функция возвращала некоторое значение, то в ее теле (в программном коде, заключенном в фигурные скобки) предусматривают оператор возврата `return` с указанием справа от него того, что следует возвратить. Возвращаемой величиной может быть любое выражение: простое значение, имя переменной или вычисляемое выражение. Оператор `return` может встречаться в коде функции несколько раз. Если оператор `return` не использовать или не указывать справа от него возвращаемую величину, то функция ничего не будет возвращать.

Пример

```
function S_rectangle($width, $height) {
       $S=$width* $height;
       return $S;
}
```

Здесь функция с именем `S_rectangle` принимает два параметра — `$width` и `$height`, вычисляет и возвращает произведение их значений.

Определение функции, будучи помещенным в программу, усваивается интерпретатором, но сама функция (ее код или тело) не выполняется. Чтобы выполнить функцию, определение которой задано, ее необходимо вызвать:

```
имя_функции(параметры);
```

Пример

```
S_rectangle(3, 5);       // возвращаемое значение равно 15
```

Имя функции в ее вызове должно совпадать с именем ранее определенной функции. Однако совпадение с точностью до регистра не обязательно. Например, выражения `S_rectange(3, 5)` и `s_Rectangle(3, 5)` эквивалентны.

> ***Примечание***
>
> В JavaScript имя функции в ее вызове и в определении должны совпадать с учетом регистра.

Параметры, если они заданы в определении функции, в вызове этой функции представляются конкретными значениями, переменными или выражениями:

```
$x=5;
$S=S_rectangle(3, $x+2);
```

Параметры функции могут быть переданы по значению и по ссылке. По умолчанию они передаются по значению. Это означает, что изменение в теле функции значения переменной, переданной в качестве параметра, никак не отра-

зится на значении этой же переменной во внешнем коде. Кроме того, любая переменная, созданная в теле функции, является локальной, т. е. видимой только в пределах ее кода.

Пример 1

```
<?php
$x=5;
function myfunc($arg){
   $arg=$arg+2;
   $y=10;
   return      $arg;
}
   myfunc($x);     // возвращает 7
   echo $x;        // выводит 5 (прежнее значение)
   echo $y;        /* ничего не выводится, т. к.
                   значение $y вне функции myfunc() равно null
?>
```

При передаче параметров по значению в функции создаются локальные переменные с такими же именами, что и имена формальных параметров, указанные в определении функции. Это локальные переменные, отличные от внешних, даже если их имена совпадают. Следующий пример эквивалентен предыдущему.

Пример 2

```
<?php
$x=5;
function myfunc($x){
     $x=$x+2;
     $y=10;
     return      $x;
}
myfunc($x);         // возвращает 7
echo $x;            // выводит 5 (прежнее значение)
echo $y;            /* ничего не выводится, т.к.
                    значение $y вне функции myfunc() равно null */
?>
```

Чтобы изменения параметров, произведенные в теле функции, были видны и за ее пределами, их необходимо передавать по ссылке. Для этого перед именем параметра в списке параметров в определении функции следует написать символ & (амперсанд):

```
function имя_функции(&$arg1, &$arg2, …){
 код
}
```

Пример

```
<?php
$x=5;
function myfunc(&$arg){
     $arg=$arg+2;
     return      $arg;
}
myfunc($x);
echo $x;          // 7 (новое значение)
?>
```

При передаче параметров по ссылке формальному параметру, указанному в определении функции, передается не значение, а адрес внешней переменной.

> ***Примечание***
>
> В JavaScript параметры примитивных типов передаются функции только по значению.

При передаче функции большего числа параметров, чем предусмотрено в определении, лишние будут просто проигнорированы. Если передается меньше, то пропущенным параметрам присваивается значение `null` и при этом может быть выведено соответствующее предупреждающее сообщение, что, однако, не приведет к остановке выполнения кода. Вместе с тем, в определении функции можно указать значения параметров, принимаемые по умолчанию. Это полезно, если вы укажете в вызове функции меньше параметров, чем предусмотрено в определении. Для задания значений по умолчанию служит оператор присваивания непосредственно в списке формальных параметров определения функции.

Пример

```
<?php
function S_rectangle($width=10, $height=5) {
     $S=$width* $height;
     return $S;
}
echo S_rectangle();      // 50
echo "<br>";
echo Srectangle(3);      // 15
?>
```

ВНИМАНИЕ!

1. При задании для параметров в определении функций значений по умолчанию, последние должны быть простыми значениями, а не переменными, вычисляемыми выражениями или вызовами функций и методов объектов.
2. Если в списке одновременно присутствуют параметры без значений по умолчанию, а также параметры с заданными значениями по умолчанию, то вторые должны следовать за первыми.

При создании функций, способных работать с переменным по длине списком параметров, можно применять следующие встроенные функции:

- `func_num_args()` — возвращает количество параметров, переданных функции, из которой она была вызвана;

Пример

```
<?php
function S_rectangle($width, $height) {
if (func_num_args()==0){      // если нет параметров
          $width=10;
          $height=10;
}
if (func_num_args()==1){      // если один параметр
          $height=10;
}
return $width* $height;
  }
  echo S_rectangle();      // 100
  echo "<br>";
  echo S_rectangle(2);      // 20

  ?>
```

- `func_get_args()` — возвращает массив значений параметров, переданных функции, из которой она была вызвана;

Пример

```
<?php
function S_rectangle($width, $height) {
/* значения параметров по умолчанию: */
   $width=10;
   $height=10;
   $args=func_get_args();          // массив значений параметров
   $narg=count(func_get_args());   // длина массива значений параметров
if ($narg==1) $width=$args[0];     // если один параметр
if ($narg==2) {$width=$args[0]; $height=$args[1];} //если два параметра
return $width* $height;
  }
  echo S_rectangle();      // 100
  echo "<br>";
  echo S_rectangle(2);     // 20
  echo "<br>";
  echo S_rectangle(2, 3);  // 6
  ?>
```

- `func_get_arg(номер_параметра)` — возвращает значение параметра, указанного с помощью его номера. Если `номер_параметра` превышает длину списка переданных параметров, то данная функция возвращает `false`. Обычно эта функция используется совместно с `func_get_args()` и `func_num_args()`.

Пример

```
<?php
function S_rectangle($width, $height) {
  echo "width=".func_get_arg(0)." height=".func_get_arg(1);
}
S_rectangle();          // width= height=
echo "<br>";
S_rectangle(2);          // width=2 height=
echo "<br>";
S_rectangle(2, 3);     // width=2 height=3
?>
```

Внимание!

При вызове функции с некорректным количеством параметров (даже если вы обрабатываете эти ситуации в теле функции) может выводиться предупреждающее сообщение "Warning: Missing argument…". Однако программа при этом продолжает выполняться. Если вы желаете предотвратить вывод подобных сообщений, то перед вызовом функции следует указать символ `@`, например, `@S_rectangle()`.

5.11.2. Переменные функции

Если рядом с именем переменной написать круглые скобки, то интерпретатор PHP возьмет значение этой переменной и попытается выполнить одноименную функцию. Иначе говоря, если строковое значение переменной совпадает с именем определенной функции и рядом с переменной указаны круглые скобки, то будет совершена попытка выполнить эту функцию. В круглых скобках при необходимости можно указать список параметров.

Пример

```
<?php
function S_rectangle($width, $height) { // определение функции
  return $width*$height;
}
$x="S_rectangle";    // значение переменной $x совпадает с именем функции
echo $x(2, 3);       // 6
?>
```

Имена функций можно хранить и в массивах.

Пример

```
<?php
$afuncname=array("sqrt","sin","cos");     // массив имен функций
$x=0.5;
$i=0;
while ($i<count($afuncname)){
     echo $afuncname[$i]($x);
     echo "<br>";
     $i++;
}
?>
```

Данный код выводит на экран значения математических функций `sqrt(0.5)`, `sin(0.5)` и `cos(0.5)`, имена которых заданы в массиве `$afuncname`.

5.11.3. Встроенные функции

В PHP есть много *встроенных функций*. Некоторые из них были рассмотрены в предыдущих разделах, подробное описание других можно найти в справочном руководстве по PHP. Следует заметить, что некоторые из встроенных функций доступны при обычной установке PHP, другие требуют особых, хотя и несложных, настроек модуля PHP. Все рассмотренные ранее встроенные функции PHP доступны всегда без специфических настроек модуля PHP. Если обратиться к справочной информации по функциям PHP, то там можно найти сведения об их доступности. Для доступа к некоторым функциям, например, работы с базами данных, графической информацией и др. требуется подключение специальных библиотек.

5.11.4. Как узнать, есть ли такая функция

Вызов функции, встроенной или пользовательской, возможен, если имеется ее определение. Иногда в сценарии требуется узнать, имеется ли определение той или иной функции. Это можно сделать с помощью встроенной функции `function_exists(имя_функции)`, которая возвращает `true`, если определение указанной в параметре функции существует, и `false` — в противном случае. Однако следует иметь в виду, что данная функция возвращает `true` и для тех функций, которые существуют, но недоступны из-за настроек конфигурации.

5.12. Классы и объекты

Поскольку PHP 5 — объекто-ориентированный язык программирования, то он предполагает особый синтаксис и подходы к разработке и анализу программ. Основные элементы программ в ООП — объекты. Обычно они так или иначе соответствуют предметам действительности, моделируемой посредством программ. Например, автомобиль, банковский счет, форму для ввода данных и персонажи компьютерной игры можно рассматривать как объекты. С формальной же точки зрения объект является неким контейнером, в котором сгруппированы свойства и методы объекта. Свойства и методы можно понимать как переменные и функции, связанные с объектом или, если хотите, присущие ему.

Наряду с понятием объекта в ООП существует понятие класса. Класс — это описание, своего рода шаблон, по которому создаются объекты. Иногда вместо

термина "класс" применяют термин "объект", а вместо "объект" — "экземпляр объекта" (такую терминологию мы использовали при рассмотрении JavaScript). В этой главе мы будем говорить о классах и объектах.

Между классами может быть установлено отношение наследования. Мы можем создать класс A как контейнер, содержащий описание свойств некоторого моделируемого предмета. Эти свойства могут быть различного уровня. Так, они могут восприниматься нашими органами чувств более или менее непосредственно, обнаруживаться с помощью датчиков и измерительных приборов, а также "вычисляться" в уме посредством усвоенных концепций и теорий. Класс A может быть более или менее абстрактным (общим). Возможно, мы захотим уточнить его, сохранив некоторые исходные свойства и добавив новые. Речь идет о создании нового класса B, связанного каким-то образом с исходным. Если класс B является наследником (потомком) класса A, то он имеет (наследует) все его свойства и методы, а также, возможно, обладает и собственными свойствами и методами. При этом класс A называют родительским или базовым для класса B. Например, можно создать два класса автомобилей: Грузовик и Легковой. Ни один из них не является наследником другого. Поскольку между ними много общего, можно сначала создать общий класс Автомобиль, а затем два класса-наследника — Грузовик и Легковой.

> ***Примечание***
>
> PHP 5 не в полной мере реализует принципы ООП в отличие от, например, классического ООП-языка C++. Так, PHP не поддерживает полиморфизм (в одном классе не может быть двух и более одноименных методов с различными наборами параметров), а также множественное наследование (каждый класс не может иметь более одного родительского класса). Тем не менее, ООП-возможности PHP 5 существенно шире, чем в JavaScript.

5.12.1. Определение класса

Определение класса, который не является наследником другого класса, имеет следующий синтаксис:

```
class имя_класса {
        определение свойств
        определение методов
}
```

Чтобы определить класс-наследник, используется такой синтаксис:

```
class имя_класса-наследника extends имя_класса-родителя {
        определение свойств
        определение методов
}
```

В объекте, созданном на основе класса имя_класса-наследника, доступны все свойства и методы класса имя_класса-родителя. Однако обратное утверждение неверно: в объекте родительского класса свойства и методы его наследников недоступны.

Свойства и методы

Свойства в определении класса задаются, как обычные переменные. При этом установить для свойств значения по умолчанию можно с помощью оператора присваивания. Значения должны быть простыми (различных типов), а не задаваться посредством вычисляемых выражений. Вместе с тем, можно (и рекомендуется) объявлять переменные ключевым словом var.

Пример определения класса с несколькими свойствами-переменными

```
class myobject{
     var $x;
     var $y=5;
     var $color="красный"
     var $address("Санкт-Птербург", "Литейный пр. ", 22, 3);
}
```

> ***Внимание!***
>
> Спецификатор (ключевое слово) var перед именем свойства означает доступность этого свойства из-за пределов кода класса. var можно заменить на public (публичный, общедоступный). Допустимы также и другие спецификаторы: private и protected, но они задают ограничение доступа к свойствам (см. *разд. 5.13.3*). При объявлении свойств указание спецификатора обязательно.

Методы объекта также являются его свойствами, но задаются не как переменные, а как функции. Методы обычно изменяют значения свойств объекта и/или выполняют какие-нибудь другие действия. Если в методе (функции) требуется обратиться к какому-нибудь свойству (переменной) этого же объекта, то перед именем этого свойства следует указать префикс $this->. Например, выражение $this->color означает обращение к свойству $color этого же класса (англ. this — этот). Метод задается в определении класса, как обычная функция ключевым словом function.

Пример

```
/* описание простого класса для создания счета */
class myaccount {
     var $summa=0;                // остаток на счете
```

```
     function addsum($sum) {     // добавление к сумме на счете
     $this->summa = $this->summa + $sum;
     echo "Добавлено $sum рублей";
}
}
```

Данный класс имеет одно свойство `$summa` и один метод `addsum()`. Он предназначен для хранения некоторого числа, понимаемого как остаток денежной суммы на некотором счете, т. е. как состояние счета. С помощью метода `addsum()` можно изменить текущее состояние счета, указав в качестве параметра число, которое следует прибавить к остатку на счете (если это число положительное) или вычесть из остатка (если число отрицательное).

Конструктор

В определении класса при необходимости можно задать один особый метод, называемый конструктором. Конструктор выполняется при создании объекта на основе данного класса, хотя объект можно создать, не определяя конструктор явно. Обычно конструктор определяют явным образом, если при создании объекта требуется сразу же выполнить некоторый код, установить начальные значения свойств объекта и/или передать значения его свойств как параметры. Формально конструктор — это функция, имя которой не произвольно, а является ключевым словом `__constuct` (с двумя ведущими символами подчеркивания) или же должно совпадать с именем класса.

Пример

```
class account {
     var $summa=0;               // свойство
     function __construct(){     // конструктор
          $x=1000;
          $this->summa=$x;
          echo "На вашем счете $x руб.";
     }
     function addsum($sum) {     // метод: добавление к сумме на счете
          $this->summa = $this->summa + $sum;
          echo "Добавлено $sum рублей";
     }
}
```

Здесь конструктор устанавливает текущее значение счета `$summa` (хотя значение по умолчанию для этого свойства уже задано) и выводит соответст-

вующее сообщение. Заметим, что в данном примере в качестве имени конструктора можно было бы выбрать `account`, т. е. имя, совпадающее с именем класса.

Как уже отмечалось, конструктор в явном виде применяется еще и для того, чтобы передать создаваемому объекту значения его свойств подобно тому, как это делается в случае обычных функций.

Пример

```
class account {
     var $summa;                          // свойство
     function __construct($summa){        // конструктор
          $this->summa=$summa;
          echo "На вашем счете $this->summa руб.";
     }
     function addsum($sum) {      // метод: добавление к сумме на счете
          $this->summa = $this->summa + $sum;
          echo "Добавлено $sum рублей";
     }
}
```

Здесь в определении класса объявлено свойство `$summa`. Конструктор выполняет две операции:

- обеспечивает передачу создаваемому объекту значения свойства `$summa` как параметра;
- выводит сообщения о состоянии счета на момент создания объекта.

Теперь рассмотрим создание двух классов — родительского с именем `account` и его наследника с именем `myaccount` (листинг 5.12).

Листинг 5.12. Пример родительского класса и его наследника

```
<?php
class account {
     var $summa;                 // объявление свойства
     function addsum($sum) {     // метод: добавление к сумме на счете
          $this->summa = $this->summa + $sum;
     }
}

class myaccount extends account{
```

```
    var $name;      // объявление свойства
    function __construct($nam, $sum){      // конструктор
        $this->name=$nam;
        $this->summa=$sum;
    }
    function set_name($name) {      /* метод: установка имени владельца
                                    счета */
        $this->name = $name;
    }
}
?>
```

Класс `account` содержит только свойство `$summa` для хранения суммы счета и метод `addsum()` для его пополнения. Класс `myaccount` уточняет (дополняет) родительский класс `account` свойством `$name` для хранения имени владельца счета и методом `set_name()` для его установки. Конструктор класса `myaccount` обеспечивает установку имени и суммы на счете при создании объекта `myaccount`. Создание и применение объектов будет рассмотрено в следующем разделе.

Разумеется, рассмотренную задачу можно решить и на основе одного, а не двух классов. Скорее всего, так и следовало бы поступить, но приведенный код призван лишь проиллюстрировать наследование свойств классов.

5.12.2. Применение объектов

Если в сценарии находится определение класса, то его можно использовать для создания объектов. Определение класса может располагаться в сценарии непосредственно либо находиться в другом файле, содержимое которого включается в текущий сценарий с помощью функции `include()`.

Объект создается на основе определенного класса с помощью оператора `new`:

```
$имя_переменной = new имя_класса(параметры);
```

Оператор = здесь присваивает переменной ссылку на созданный объект, которую затем удобно применять для доступа к его свойствам и методам. Эта переменная имеет тип `object`. Для доступа к свойствам и методам объекта применяется так называемый стрелочный (а не точечный, как в JavaScript) синтаксис:

```
объект->свойство;
объект->метод();
```

Пример

```
<?php
class account {
… // код из листинга 5.12
}
class myaccount extends account{
… // код из листинга 5.12
}
$w=new myaccount("Вася",1000);      // Положили Васе на счет 1000
$w->addsum(500);                    // Добавили 500
$w->set_name("Петя");               // Заменили имя на Петя
echo "Имя: $w->name".". Сумма: $w->summa рублей";     /* Вывод строки:
"Имя: Петя. Сумма: 1500 рублей" */
?>
```

В описании класса `myaccount` (см. листинг 5.12) конструктор организует прием значений двух параметров: имени и начального значения счета. Например, в результате выполнения выражения

```
$w=new myaccount("Вася",1000);
```

свойствам `$name` и `$summa` будут присвоены значения "Вася" и "1000" сразу же при создании объекта. Кроме того, переменной `$w` присвоена ссылка на созданный объект, которая была использована в выражениях вызова методов:

```
$w->addSum(500);
$w->set_name("Петя");
```

Обратите внимание на то, что первый метод определен в родительском классе, а второй — в его классе-наследнике. Тем не менее, мы можем вызывать их как методы объекта класса-наследника `myaccount`.

5.12.3. Ограничение доступа к свойствам и методам

Рассмотренные ранее свойства и методы класса доступны из сценария как свойства и методы объекта, созданного на основе этого класса. Иначе говоря, при синтаксически правильном обращении свойства и методы объекта доступны из программного кода, являющегося внешним по отношению к коду описания класса. Однако в ряде случаев требуется, чтобы некоторые свойства и/или методы объекта были недоступны из внешнего кода. Так, например, считается, что прямой доступ к свойствам-переменным класса — плохой

стиль программирования. Лучше обеспечивать доступ к свойствам извне с помощью специальных методов класса.

В PHP можно установить ограничение доступа к свойствам (переменным и функциям) класса с помощью следующих спецификаторов (ключевых слов):

- `private` (закрытый, частный) — запрещает доступ к свойствам и методам класса за его пределами (из сценариев и других классов);
- `protected` (защищенный) — запрещает доступ к свойствам и методам класса за его пределами, за исключением его классов-наследников.

Примеры

```
private $x;
protected $myvar;
protected function myfunc($arg){…}
```

Перепишем определение класса `account` (см. листинг 5.12) так, чтобы нельзя было воспользоваться его методом непосредственно (листинг 5.13).

Листинг 5.13

```
class account {
     var $summa;            // объявление свойства
     protected function addsum($sum){ /* защищенный метод:
                                         добавление к сумме на счете */
          $this->summa = $this->summa + $sum;
     }
}
```

Теперь попытка выполнить, например, код

```
$x=new account();
$x->addsum(500);
```

приведет к выводу сообщения об ошибке. Однако изменить значение свойства `$summa` можно из кода класса-наследника `myaccount`, если добавить к нему соответствующий метод (листинг 5.14).

Листинг 5.14

```
class myaccount extends account{
     var $name;      // объявление свойства
     function __construct($nam, $sum){          // конструктор
          $this->name=$nam;
```

```
            $this->summa=$sum;
        }
        function set_name($name) {      // метод: добавление к сумме на счете
            $this->name = $name;
        }
        function add_sum($sum) {       // метод: добавление к сумме на счете
            $this->addsum($sum);       // вызов защищенного метода
        }
}
```

Здесь метод `add_sum($sum)` просто вызывает защищенный метод `addsum($sum)` родительского класса `account`.

> ***Внимание!***
>
> Спецификатор `var` означает, что следующее за ним свойство не имеет ограничений доступа. Его можно заменить эквивалентным спецификатором `public`.

5.12.4. Клонирование и удаление объектов

Для клонирования (копирования) объекта предусмотрен оператор

```
$объект_копия = clone $исходный_объект;
```

вызывающий специальный метод `__clone()` (два символа подчеркивания), который по умолчанию создает новый объект со свойствами исходного.

Пример

```
$w->new mycount("Саша", 500); // см. листинг 5.14.
$q-> clone $w;      // $q — копия объекта $w
$q->add_sum(500);
$q->set_name("Маша");
echo "Имя: $q->name".". Сумма: $q->summa";
```

Однако метод `__clone()` можно переопределить подобно тому, как это делается с конструктором (методом `__construct()`).

Удалить ранее созданный объект можно с помощью функции `unset($объект)`.

Пример

```
$w->new mycount("Саша", 500);// создание объекта (см. листинг 5.14)
unset($w);                   // удаление объекта $w
```

Нередко при удалении объекта требуется выполнить какие-то действия, например, записать данные в файл или базу данных, закрыть файлы и т. п. В подобных случаях можно в классе определить код специального метода `__destruct()` (два символа подчеркивания), который выполнится перед тем, как объект будет удален. Этот метод называют *деструктором*.

Пример

```
class account {
     var $summa;            // объявление свойства
     function _destructor(){
     echo "Объект уничтожен";
}
     protected function addsum($sum){ /* защищенный метод:
                                    добавление к сумме на счете */
          $this->summa = $this->summa + $sum;
     }
}
```

5.12.5. Использование методов несозданных объектов

Чтобы обратиться к методу или свойству класса, объект для которого еще не создан, можно использовать следующий синтаксис:

```
имя_класса::свойство();
имя_класса::метод();
```

В следующем примере определен класс с единственным методом, выводящим сообщение, и далее вызов этого метода без предварительного создания объекта данного класса.

Пример

```
<?php
class myclass {
     function mymethod(){
          echo "Привет из класса myclass";
     }
}
myclass::mymethod();
?>
```

5.12.6. Обработка исключений

Обработка ошибок в программах на языке PHP происходит на основе так называемого механизма исключительных ситуаций, который использует встроенный класс `Exception` (исключение). Вы можете определить некоторые нежелательные (исключительные) ситуации и обработать их по своему усмотрению. Если этого не сделать, то при возникновении ошибок интерпретатор PHP обработает их некоторым стандартным образом с выводом соответствующих сообщений.

Обратимся в качестве примера к определению класса `account`, приведенному в листинге 5.12. Метод `addsum()` в этом классе позволяет изменять состояние счета (свойство `$summa`). Предположим, что состояние счета не должно быть отрицательным. Иначе говоря, отрицательное значение свойства `$summa` определяется нами как исключительная ситуация, которую следует обработать особо. В листинге 5.15 обработка данной ситуации состоит в выводе сообщения и прекращении выполнения сценария.

Листинг 5.15. Пример обработки исключений

```
<?php
class account {
     var $summa;                // объявление свойства
     function addsum($sum) {     // метод: добавление к сумме на счете
          $this->summa = $this->summa + $sum;
          try {
               if ($this->summa < 0) { // если исключение
                    throw new Exception ("Вы должник!");
                     }
          } catch (Exception $e) {
               exit($e->getMessage());
          }
     }
}
$w=new account();      // создание объекта класса account
$w->addsum(-100);     // снять со счета 100
?>
```

Здесь код метода `addsum()` содержит блоки:

- `try` (попытка) — содержит оператор условия `if`; если результат проверки условия равен `true`, то генерируется исключение и выполняется выражение, следующее за ключевым словом `throw`, в данном случае — создание нового объекта `Exception` с передачей ему текста сообщения;

- `catch` (перехват) — указывается объект класса `Exception`, созданный для данного исключения, а также блок кода, который следует выполнить, если исключительная ситуация возникла. В данном примере это — вызов метода `getMessage()`, который возвращает сообщение о ситуации, хранящееся во встроенном классе `Exception`, вывод этого сообщения и прекращение выполнения сценария с помощью функции `exit()`. Если исключение не сгенерировано, то блок кода `coatch` не выполняется.

5.12.7. Пример класса формы

Рассмотрим в качестве примера программную реализацию HTML-формы на основе класса, содержащего следующие свойства и методы:

- `$afld` — массив полей ввода данных;
- `$action` — значение атрибута `action` тега `<FORM>`, которое используется для указания серверной программы, обрабатывающей данные формы;
- `$submitlabel` — надпись на кнопке типа `submit` для отправки данных на сервер;
- `$nfld` — число полей в форме;
- `__construct()` — конструктор для присвоения значений свойствам `$action` и `$submitlabel` при создании объекта формы;
- `addfld()` — метод, добавляющий поле в форму;
- `showform()` — метод, отображающий форму.

Таким образом, создаваемая форма будет содержать кнопку типа `submit` и поля ввода данных типа `input`, которые при необходимости можно добавлять к форме. В листинге 5.16 приведено определение класса `Form` и сценарий, добавляющий в форму два поля ввода данных, а затем отображающий ее в окне браузера (рис. 5.15).

Листинг 5.16. Пример создания HTML-формы на основе класса

```
<?php
class Form {
/* Класс формы */
     var $afld=array();
     var $action;
     var $submitlabel="Отправить";
     var $nfld=0;
/* Конструктор */
     function __construct($action, $submitlabel){
```

```
            $this->action=$action;
            $this->submitlabel=$submitlabel;
      }
/* Добавление полей */
      function addfld($name, $label){
            $this->afld[$this->nfld]["name"]=$name;
            $this->afld[$this->nfld]["label"]=$label;
            $this->nfld=$nfld + 1;
      }
/* Отображение формы */
      function showform(){
            echo "<FORM action={$this->action} method=POST>";
            for($i=1; $i<=count($this->afld); $i++){
                  echo "{$this->afld[$i-1]['label']}: ";
                  echo "<input type=text
                        name={$this->afld[$i-1]['name']}><br>";
            }
            echo "<input type=submit value='$this->submitlabel'>";
            echo "</FORM>";
      }
}
/* Создание объекта класса Form */
$f=new Form("http://localhost/myform.php", 'Отправить данные');
/* Добавление полей */
$f->addfld("username","Имя");
$f->addfld("address","Адрес");
/* Отображение формы */
$f->showform();
?>
```

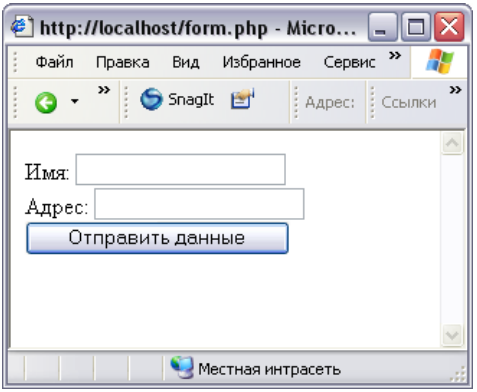

Рис. 5.15. Форма, сгенерированная сценарием листинга 5.16

5.13. Выполнение PHP-кода в текстовых строках

С помощью сценария можно написать или сгенерировать текстовую строку, содержащую PHP-код, а затем, когда это потребуется, выполнить его. Такие строки могут храниться, например, в базе данных. В этом случае их можно занести в переменную для последующего выполнения.

Для выполнения PHP-кода в текстовой строке служит функция `eval(строка)`.

Пример

```
<?php
$x=2; $y=3;
$str = '$z = $x + $y;';
eval($str);     // вычисляет выражение $z = $x + $y;
echo $z;       // выводит число 5
 ?>
```

Обратите внимание, что значение переменной `$str` записано в одинарных кавычках, чтобы заключенные в них переменные не интерпретировались. Функция `eval($str)` вычисляет выражение, являющееся значением переменной `$str`.

Функция `eval(строка)` может возвращать значение, указанное в строке справа от оператора `return`, если, разумеется, он есть. В противном случае возвращается `null` (если не было ошибок) или `false` (при возникновении ошибки). Если возникает фатальная ошибка, происходит прекращение работы всего сценария.

Пример

```
<?php
$x=2; $y=3;
$str = '$z = $x + $y; return $z;';
echo eval($str);// выводится возвращаемое функцией eval() значение $z
?>
```

Обратите внимание, что значение переменной `$str` заключено в одинарные кавычки. При указании двойных кавычек значение `$str` было бы равно

```
" =2 + 3; return ; ",
```

а при выполнении функции `eval($str)` появилось бы сообщение об ошибке:

Parse error: parse error, unexpected '=' in **c:\Inetpub\wwwroot\eval.php(4) : eval()'d code** on line **1** (Ошибка преобразования: недопустимый символ '=' в c:\Inetpub\wwwroot\eval.php(4): код строки 1 функции `eval()`).

Из приведенных примеров видно, что с помощью функции `eval()` можно создавать новые переменные и присваивать им значения.

Пример

```
<?php
$x="newvar";
eval('$'.$x.'=12345;');
echo $newvar;     // выводится число 12345
?>
```

Примечание

Аналогичная функция `eval()` имеется и в JavaScript.

Глава 6

Основы создания серверных сценариев на PHP

В данной главе мы рассмотрим основные приемы решения типичных задач на стороне сервера:

- взаимодействие с серверным сценарием посредством HTML-форм, которые принимают от клиента данные и передают их серверу для последующей обработки;
- организация сеансов, в пределах которых обеспечивается доступ ко всем данным, сгенерированным различными сценариями, подключенными к данному сеансу;
- работа с файловой системой;
- работа с базами данных.

6.1. Получение данных из форм клиента

В *главе 4* мы рассматривали задачи передачи данных из Web-страницы серверу. В частности, мы говорили о том, что данные, содержащиеся в элементах формы, можно передать на сервер. Это можно выполнить как с помощью клиентских сценариев, так и без них. Если на Web-странице имеется форма (например с текстовыми полями, переключателями и кнопкой типа submit), то щелчок пользователя мышью на кнопке типа submit (Отправить) инициирует отправку данных на сервер. А как он их получает и что можно сделать на стороне сервера?

В данном разделе мы рассмотрим получение сервером данных из клиентских HTML-форм (элементов документа, созданных с помощью контейнерного HTML-тега `<FORM>`). При этом сначала опишем передачу содержимого обычных полей формы, а затем — загрузку на сервер файлов, расположенных на компьютере клиента.

6.1.1. Получение данных из элементов форм

HTML-форма, создаваемая контейнерным тегом <FORM>, является стандартным средством для отправки на Web-сервер данных, введенных пользователем с клавиатуры или сгенерированных сценарием (клиентским или серверным). В атрибуте action тега <FORM> указывается URL-адрес файла со сценарием, который должен принять эти данные и, быть может, как-то их обработать. В контейнере <FORM action=URL_адрес> могут содержаться различные элементы пользовательского интерфейса (поля ввода данных, флажки, селекторные переключатели, кнопки) и должна быть, по крайней мере, одна специальная кнопка, создаваемая тегом <INPUT type=submit>. Щелчок на кнопке типа submit (Отправить), вставленной в форму, приводит к отправке на сервер данных, содержащихся в элементах формы. В этом разделе мы сосредоточимся на задаче приема данных сервером (точнее, серверным сценарием).

Получив данные от клиента, серверный сценарий может распорядиться ими так или иначе, например, просто сохранить в файле или базе данных, обработать каким-либо образом и, в зависимости от результата, послать клиенту некоторое сообщение, сгенерировать HTML-документ или сделать что-нибудь иное. Сейчас для нас важнее узнать, как данные клиентской формы воспринимаются серверным сценарием.

Ранее уже упоминалось, что данные HTML-форм могут быть переданы на сервер методами GET и POST. Метод GET добавляет передаваемые данные к URL-адресу сценария-обработчика и, следовательно, их можно увидеть в адресной строке браузера. Иногда это нежелательно, что вынуждает прибегнуть к методу POST. Кроме того, методом POST можно передавать сколь угодно большие по объему данные. Поэтому он более распространен.

Метод передачи указывают в атрибуте method тега <FORM>, например, <FORM action=URL_адрес method=POST>. Данные, полученные сервером при их передаче тем или иным методом, сохраняются в суперглобальных встроенных массивах $_GET и $_POST соответственно (см. *разд. 5.10*). Если в тегах элементов формы (<INPUT>, <TEXTAREA> и т. п.) вы указали значение атрибута name (имя элемента), то сервер получит соответствующие пары вида *имя_элемента=значение_элемента*. При этом имя_элемента (значение атрибута name) станет символьным индексом (ключом) массива $_GET или $_POST в зависимости от того, каким методом передавались данные. По индексу массива, как известно, можно получить значение его элемента, т. е. значение соответствующего элемента формы. Например, чтобы получить содержимое элемента формы с именем (значением атрибута name тега этого элемента) myelement, достаточно обратиться к элементу массива $_POST['myelement'] или $_GET['myelement'] в зависимости от использованного метода передачи.

Примечание

Чтобы получить данные формы независимо от метода передачи (`GET`, `POST` или `cookie`), можно воспользоваться суперглобальным массивом `$_REQUEST`. Указав в качестве символьного индекса имя (значение атрибута `name`) элемента формы, вы получите значение этого элемента.

В листинге 6.1 в качестве примера приводится содержимое файла Форма.php, который предоставляет браузеру клиента некоторую форму и отображает введенные пользователем данные. Точнее, вызов пользователем данного файла приводит к загрузке в браузер формы с двумя полями для ввода данных и кнопкой типа `submit`. Щелчок на этой кнопке отправляет на сервер данные, введенные пользователем, и отображает полученные данные как список имен элементов формы и их значений (рис. 6.1).

Листинг 6.1. Получение и отображение данных HTM-формы (Форма.php)

```
<FORM action="http://localhost/forma.php" method=POST>
<INPUT type="text" name="name" value="Вася">
<INPUT type="text" name="message" value="Привет всем">
<INPUT type="submit" name="submit" value="Отправить">
</FORM>
<?php
/* Вывод содержимого массива $_POST */
foreach($_POST as $key=>$value){
     echo "$key=$value <br>";
}
?>
```

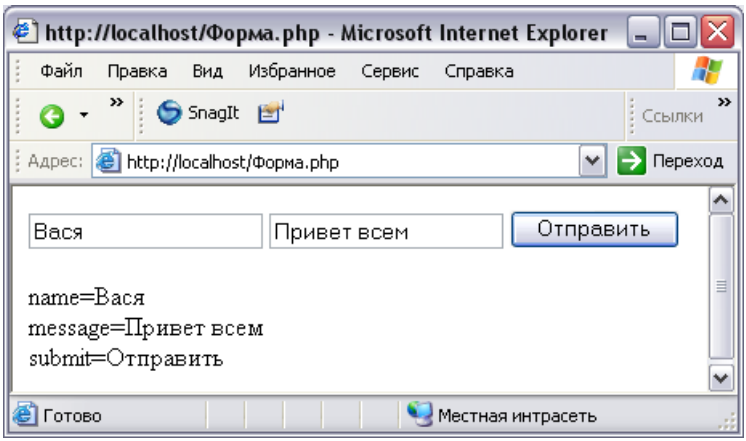

Рис. 6.1. Форма для ввода данных и их отображение с помощью сценария

Если в атрибуте `method` тега `<FORM>` указать значение `GET`, то для получения данных необходимо воспользоваться суперглобальным массивом `$_GET` или `$_REQUEST`. Разумеется, в PHP-сценарии вместо вывода элементов массива `$_POST` можно сделать что-либо более интересное или полезное. Здесь мы лишь демонстрируем возможность доступа к полученным от клиента данным.

Теперь рассмотрим подробнее, что именно происходит при реализации данного примера:

1. При загрузке в браузер файла Форма.php (например, http://localhost/Форма.php) в окне браузера отображаются элементы формы (два поля ввода и кнопка типа `submit`): HTML-код этого файла отображается, а PHP-код ничего не выводит, поскольку в данный момент массив `$_POST` пуст.
2. После ввода в элементы формы каких-либо данных и щелчка на кнопке типа `submit`, эта информация передается серверу — исходному файлу Форма.php. Таким образом, браузер клиента вновь запрашивает файл Форма.php. Теперь массив `$_POST`, всегда доступный любому серверному PHP-сценарию, уже не пуст. Сервер возвращает браузеру клиента форму с пустыми полями, а затем то, что посылает оператор `echo` в PHP-сценарии, — индексы и значения массива `$_POST`.

Обратите внимание, что в рассмотренном примере форма была создана с помощью HTML-кода непосредственно. Однако то же самое можно сделать исключительно с помощью PHP-сценария, используя оператор `echo` (листинг 6.2). В этом сценарии с помощью оператора `echo` в окно интернет-обозревателя выводится строка, содержащая теги формы, которые интерпретируются браузером: в его окне отображается форма в соответствии с заданной разметкой.

Листинг 6.2. Форма, созданная PHP-сценарием

```
<?php
echo '<FORM action="http://localhost/Форма.php" method=POST>
<INPUT type="text" name="name" value="">
<INPUT type="text" name="message" value="">
<INPUT type="submit" name="submit" value="Отправить">
</FORM>';
foreach($_POST as $key=>$value){
     echo "$key=$value <br>";
}
?>
```

В форме может содержаться набор элементов, допускающих множественный выбор (т. е. указание сразу нескольких значений). Типичный пример — набор флажков, каждый из которых задается тегом `<INPUT type=checkbox>`, а пользователь может установить произвольную их комбинацию. Принадлежность флажков одному и тому же набору обеспечивают одинаковые имена — значения атрибута `name`. В этом случае удобно задавать значение атрибута `name` в виде массива, поскольку в PHP-сценарии легко организовать разбор того, что выбрал пользователь в наборе одноименных элементов. В примере, приведенном в листинге 6.3, показано, как организовать получение массива значений, содержащихся в элементах типа "флажок". Вид формы в окне браузера и отклик сценария иллюстрирует рис. 6.2.

Листинг 6.3. Передача значений флажков в виде массива

```
<FORM action="http://localhost/check.php" method=POST>
<h2>Выберите товар:</h2>
<INPUT type="checkbox" name="check[]" value="Автомобиль">Автомобиль<br>
<INPUT type="checkbox" name="check[]" value="Телевизор">Телевизор<br>
<INPUT type="checkbox" name="check[]" value="Пирожок">Пирожок<br>
<INPUT type="submit" name="submit" value="Отправить">
</FORM>
<?php
if (isset($_POST['check'])){     // если $_POST['check'] существует
     echo "<b>Вы выбрали: </b><br>";
     /* Вывод принятых значений флажков */
     foreach($_POST['check'] as $key=>$value){
     echo "$value <br>";
  }
}
?>
```

Обратите внимание, что все флажки имеют одинаковые имена `"check[]"`, в совокупности образующие массив. `$_POST['check']` — тоже массив, содержащий значения атрибутов `value` только установленных флажков, а доступ к ним можно получить с помощью числовых индексов. В данном примере `$_POST['check'][0]` содержит `"Автомобиль"`, а `$_POST['check'][1]` — `"Пирожок"`. Таким образом, длина массива `$_POST['check']` заранее не фиксирована и зависит от того, сколько флажков было установлено пользователем.

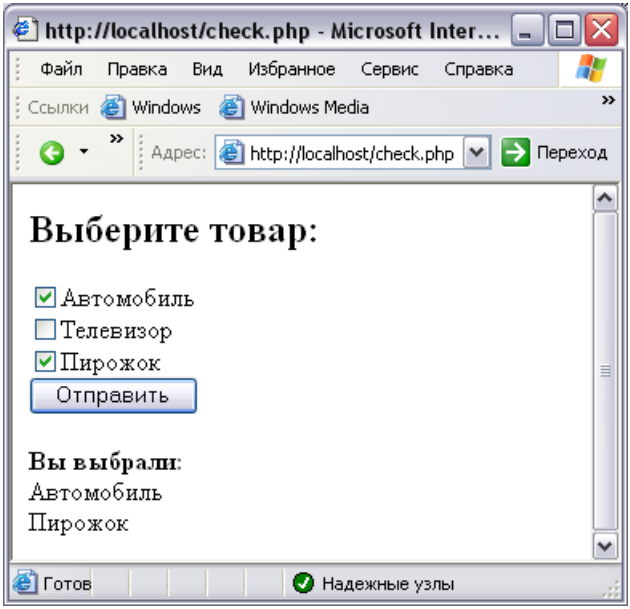

Рис. 6.2. Результат работы кода из листинга 6.3

В сценарии функция `isset()` проверяет существование массива `$_POST['check']`. При первом вызове данного файла массив `$_POST['check']` не существует, поэтому сценарий ничего не выводит. Если наличие массива `$_POST['check'])` не проверить, то при первом вызове файла появится сообщение об ошибке.

> ***Внимание!***
>
> Аналогично можно организовать отправку на сервер групп значений радиокнопок и раскрывающихся списков.

Особенностью всех ранее рассмотренных в данном разделе примеров является то, что после отправки значений элементов формы сервер очищает поля ввода. В некоторых случаях это нежелательно. Чтобы сохранить в элементах формы те значения, которые были отправлены последний раз, сценарий придется несколько усложнить (листинг 6.4).

Листинг 6.4. Передача значений флажков с сохранением их в форме

```
<?php
/* Какие флажки были установлены?
 * Массив $is_checked для каждого флажка содержит
```

```
 * либо " CHECKED" либо ""
*/
if (isset($_POST['check'])){
     for ($i=0; $i<3;$i++){
          if (isset($_POST['check'][$i])){
               $ischecked[$i]=" CHECKED";
          }else{
                    $ischecked[$i]="";
              }
     }
}
/* Вывод формы */
echo '<FORM action="http://localhost/check.php" method=POST>
<h2>Выберите товар:</h2>
<INPUT type="checkbox" name="check[0]" value="Автомобиль"'.
$ischecked[0].'>Автомобиль<br>
<INPUT type="checkbox" name="check[1]" value="Телевизор"'.
$ischecked[1].'>Телевизор<br>
<INPUT type="checkbox" name="check[2]" value="Пирожок"'.
$ischecked[2].'>Пирожок<br>
<INPUT type="submit" name="submit" value="Отправить">
</FORM>';
if (isset($_POST['check'])){     // если $_POST['check'] существует
     echo "<b>Вы выбрали: </b><br>";
     /* Вывод принятых значений флажков */
     foreach($_POST['check'] as $key=>$value){
     echo "$value <br>";
  }
}
?>
```

Данный сценарий устанавливает флажки в генерируемой форме, если это было сделано в предыдущий раз. Установить флажок программным способом можно, если в теге этого элемента указать атрибут `CHECKED`, не имеющий значений. Таким образом, нам необходимо выводить форму, теги элементов которой имеют или не имеют атрибуты `CHECKED` в зависимости от того, были они установлены или нет при последней передаче на сервер. Чтобы было удобно работать с массивом `$_POST['check']`, имена флажков определены как массив с числовыми индексами (`check[0]`, `check[1]` и `check[2]`). В результате каждому переданному значению флажка в массиве `$_POST['check']`

будет соответствовать элемент с фиксированным индексом. Например, первому флажку ("Автомобиль") всегда будет соответствовать элемент $_POST['check'][0]. Разумеется, этот элемент будет находиться в массиве $_POST['check'], только если соответствующее ему значение формы было передано на сервер.

Собственно форма создается строкой тегов и выводится с помощью оператора $echo. В эту строку в соответствующих местах вставляются элементы массива $ischecked, имеющие значения "CHECKED" либо "" в зависимости от того, установлен или нет соответствующий флажок при последней передаче.

Теперь рассмотрим другую типичную задачу передачи данных из формы. Пусть требуется сначала предоставить пользователю форму для выбора элементов (например, флажков) из заданного списка, а после отправки данных вывести только подтверждение об этом. Код для решения этой задачи представлен в листинге 6.5.

Листинг 6.5. Форма не выводится после передачи данных

```
<?php
if (!isset($_POST['submit'])){      /* если данные не передавались,
                                    выводим форму */
      echo '<FORM action="http://localhost/check.php" method=POST>
      <h2>Выберите товар:</h2>
      <INPUT type="checkbox" name="check[]"
      value="Автомобиль">Автомобиль<br>
      <INPUT type="checkbox" name="check[]"
      value="Телевизор">Телевизор<br>
      <INPUT type="checkbox" name="check[]"
      value="Пирожок">Пирожок<br>
      <INPUT type="submit" name="submit" value="Отправить">
</FORM>';
}else{          /* если была передача, выводим сообщение */
      if (isset($_POST['check'])){          // если что-то выбрано
           echo "<b>Благодарю за Ваш прекрасный выбор: </b><br>";
           foreach($_POST['check'] as $key=>$value){
               echo "$value <br>";
               }
          } else {
        echo "Вы ничего не выбрали";
       }
}
?>
```

Немного изменим только что рассмотренную задачу: пусть в форме будут две кнопки — **Отправить** и **Отмена**. Форма должна отображаться в браузере только в следующих случаях:

- при первоначальном вызове файла check.php со сценарием;
- при отправке (был щелчок на кнопке **Отправить**) пустых данных (не установлен ни один из флажков).

Если были отправлены непустые данные (при отправке был установлен хотя бы один из флажков) или же был щелчок на кнопке **Отмена**, то форма не выводится. Обе кнопки — **Отправить** и **Отмена** — элементы типа `submit`.

Для решения этой задачи нам потребуется дважды в различных местах использовать фрагмент кода, который выводит форму. Поэтому целесообразно сохранить его в отдельном файле, например myform.inc (листинг 6.6). Затем в основной сценарий содержимое этого файла можно вставить с помощью функции `include()`.

Листинг 6.6. Файл myform.inc, включаемый в сценарий из файла check.php

```
 <?php
/* Форма, включаемая в сценарий check.php */
echo '<FORM action="http://localhost/check.php" method=POST>
<h2>Выберите товар:</h2>
<INPUT type="checkbox" name="check[]" value="Автомобиль">Автомобиль<br>
<INPUT type="checkbox" name="check[]" value="Телевизор">Телевизор<br>
<INPUT type="checkbox" name="check[]" value="Пирожок">Пирожок<br>
<INPUT type="submit" name="send" value="Отправить">
<INPUT type="submit" name="cancel" value="Отмена">
</FORM><br>';
?>
```

Основной сценарий представлен в листинге 6.7.

Листинг 6.7. Основной сценарий, содержащий включения кода из файла myform.inc

```
<?php
if (isset($_POST['cancel'])) exit("<h1>До свидания!</h1>"); /* выход,
                                если был щелчок на кнопке "Отмена" */
if (!isset($_POST['send'])){     // если был щелчок на кнопке "Отправить"
        include("myform.inc");     // вывод формы
```

```
}else{
     if (isset($_POST['check'])){          // если что-то выбрано
          echo "<b>Благодарю за Ваш прекрасный выбор: </b><br>";
          foreach($_POST['check'] as $key=>$value){
               echo "$value <br>";
          }
     } else {
          include("myform.inc");     // вывод формы
          echo "К сожалению, Вы ничего не выбрали";
     }
}
?>
```

ВНИМАНИЕ!

При наличии в форме нескольких кнопок типа `submit` на сервер передается значение только одной из них, а именно той, на которой был щелчок.

СОВЕТ

При модификации сценариев, приведенных в этой книге, а также при создании и отладке своих собственных программ, рекомендую использовать функцию `var_dump(переменная)`. Например, с ее помощью полезно выводить на экран значения предопределенных переменных, таких как `$_POST`, `$_FILE` и др.

6.1.2. Передача файлов на сервер

Форма позволяет передать (загрузить) на сервер файлы, находящиеся на компьютере клиента.

Для этого необходимо выполнить следующее:

1. В теге `<FORM>` указать атрибуты:
   - `action=файл_со_сценарием` — php-файл, содержащий сценарий, обрабатывающий загруженный файл;
   - `enctype=multipart/form-data`
   - `method=POST`
2. Добавить в форму элементы:
   - `<INPUT type=file>` — поле ввода имени передаваемого файла с кнопкой **Обзор**, щелчок на которой открывает диалоговое окно для выбора файла из числа имеющихся;
   - `<INPUT type=hidden name=MAX_FILE_SIZE value=число>` — скрытое поле, содержащее `число`, которое указывает максимальный размер

файла в байтах. Файл большего размера передать нельзя. Однако это число не может превышать величины, указанной в конфигурационном файле php.ini;

- `upload_max_filesize` — максимальный размер загружаемого файла (по умолчанию 2 Мбайта, значение при необходимости можно изменить);
- `post_max_size` — максимальный объем данных, который можно передать методом POST (по умолчанию 8 Мбайт, значение при необходимости можно изменить). Этот скрытый элемент не обязателен, но если он используется, то должен предшествовать элементу `<INPUT type=file>`;
- `<INPUT type=submit>` — кнопка отправки данных.

Полученный сервером файл сначала автоматически сохраняется в папке для временных файлов. Эту папку с указанием пути к ней можно установить в конфигурационном файле php.ini, указав соответствующее значение переменной `upload_tmp_dir`. Разумеется, папку предварительно нужно создать. Если же в переменной `upload_tmp_dir` ничего не указано, то используется папка по умолчанию, например, C:\WINDOWS\TEMP. Принятый файл далее необходимо с помощью сценария (или вручную) скопировать или переместить в другую папку, поскольку содержимое временной папки не сохраняется после завершения работы сценария.

Кроме самого файла сервер получает и всю информацию о нем, которая сохраняется в суперглобальном массиве `$_FILES` с символьными индексами. Если файл на компьютере клиента был выбран с помощью элемента `<INPUT type=file name="userfile">`, то получить в серверном сценарии сведения об этом файле позволяют следующие элементы массива `$_FILES['userfile']`:

- ❒ `$_FILES['userfile']['name']` — оригинальное имя файла (без пути) на компьютере клиента;
- ❒ `$_FILES['userfile']['type']` — mime-тип файла, если браузер предоставил такую информацию (например, `"text/plain"`, `"image/gif"` и др.);
- ❒ `$_FILES['userfile']['size']` — размер файла в байтах;
- ❒ `$_FILES['userfile']['tmp_name']` – полное имя, под которым принятый файл был сохранен в папке для временных файлов;
- ❒ `$_FILES['userfile']['error']` — код ошибки, которая может возникнуть при загрузке файла:
  - 0 — ошибок нет (загрузка завершилась успешно);
  - 1 — размер принятого файла превысил максимально допустимое значение, заданное директивой `upload_max_filesize` конфигурационного файла `php.ini`;

- 2 — размер загружаемого файла превышает значение MAX_FILE_SIZE, указанное в HTML-форме;
- 3 — загружаемый файл был принят только частично;
- 4 — файл не был загружен.

Еще раз отмечу, что здесь 'userfile' — имя элемента формы, т. е. значение его атрибута name, которое разработчик выбирает по своему усмотрению.

В листинге 6.8 приведен код, предоставляющий пользователю форму для выбора передаваемого файла (рис. 6.3) и содержащий сценарий, который перемещает временный файл в папку DOWNLOADED, предварительно созданную в том же каталоге, в котором находится файл со сценарием.

Листинг 6.8. Загрузка файла на сервер

```
<H2>Передача файла</H2>
<FORM        action="http://localhost/fileup.php"        method=POST
enctype=multipart/form-data>
<INPUT type=hidden name="MAX_FILE_SIZE" value="10000">
<INPUT type="file" name="userfile" value="" SIZE=40>
<INPUT type="submit" name="submit" value="Отправить">
</FORM>
<?php
$tempfile=$_FILES['userfile']['tmp_name']; // имя временного файла
if (is_uploaded_file($tempfile)){
      if (!copy($tempfile,"./DOWNLOADED/".$_FILES['userfile']['name'])){
          echo "Не удалось скопировать файл ";
     } else {
          echo "Файл {$_FILES['userfile']['name']}
               ({$_FILES['userfile']['size']}байт) загружен успешно";
     }
} elseif (!empty($_FILES['userfile']['name'])){
     echo "Не удалось загрузить файл {$_FILES['userfile']['name']}";
}
?>
```

Успешность загрузки файла во временную папку проверяет встроенная функция is_uploaded_file(имя_файла). Для его перемещения в требуемую папку применяется функция move_uploaded_file(исходный_файл, целевой_файл). Просто скопировать файл можно, вызвав функцию copy(исходный_файл, целевой_файл). Перечисленные функции принимают строковые параметры и возвращают true или false в зависимости от успешности операции.

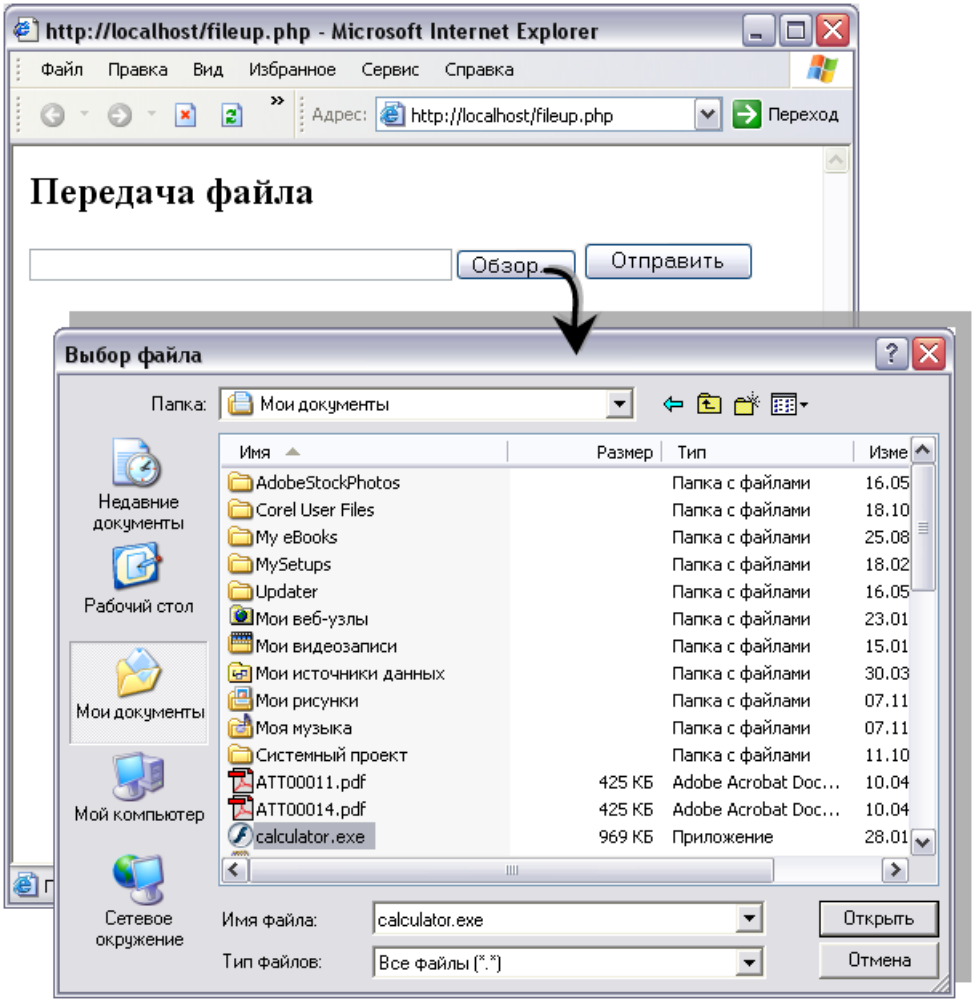

Рис. 6.3. Форма для передачи файла на сервер

6.2. Переходы между Web-страницами

Переходы между Web-страницами обычно реализуют посредством тех HTML-элементов, в которых можно указать URL-адрес, — ссылок и форм. Вместе с тем иногда требуется перейти на другую страницу сайта без вмешательства пользователя, т. е. программным способом. Кроме того, при переходах нередко возникает задача передачи данных от одной страницы к другой. Причем передаваемая информация может быть не только введена пользователем в элементы формы, но и сгенерирована программно.

В данном разделе мы рассмотрим способы перехода между страницами, а также средства обеспечения доступности данных, сгенерированных где-то на одной или нескольких страницах сайта, другим страницам (точнее, PHP-сценариям).

Итак, для переходов между страницами в PHP имеются следующие средства:

- ссылки;
- формы;
- встроенная функция PHP `header()`.

Передать данные между страницами можно с помощью таких приемов:

- добавить информацию к URL-адресу;
- применить формы (см. *главу 3*);
- использовать cookie;
- применить сеансы PHP;

6.2.1. Вывод ссылок

Чтобы отобразить на Web-странице ссылку, в PHP-файле достаточно где-нибудь вне дескрипторов `<?` и `?>` записать соответствующий HTML-код или внутри них — оператор `echo`:

```
<a href = 'page1.php'>Страница 1</a>
<?php echo "<a href = 'page2.php'>Страница 2</a>" ;?>
```

Напомню, что значением атрибута `href` является, вообще говоря, URL-адрес и, следовательно, к нему можно дописать информацию, которая будет передана указанному сценарию (см. *разд. 6.2.4*).

6.2.2. Применение форм

Возможность применения HTML-форм для перехода на другую страницу обусловлена наличием в теге `<form>` атрибутов `action` с URL-адресом, `method` с указанием метода передачи и кнопки типа `submit`. Передача данных из элементов формы рассматривалась в *разд. 6.1*. Однако форма может и не содержать никаких элементов для ввода и/или хранения данных. В этом случае щелчок на кнопке типа `submit` приведет к загрузке в браузер страницы с указанным адресом. Так реализуются, например, кнопки для навигации с надписями "Далее", "Назад" и т. п. Если к URL-адресу добавить информацию, то она будет передана на сервер (см. *разд. 6.2.4*).

6.2.3. Переадресация с помощью функции *header()*

Чтобы перейти на заданную страницу без каких-либо действий пользователя, можно применить встроенную функцию `header("Location: URL-адрес")`. Предположим, на странице page0.php выводится форма для ввода данных о пользователе (login, password и т. п.). Эти данные отправляются серверному сценарию, например, redirect.php, который как-то их обрабатывает и в зависимости от результата автоматически перенаправляет пользователя к странице page2.php или page3.php.

> ***Примечание***
>
> Все файлы страниц, за исключением файла с серверным сценарием (в нашем случае это redirect.php), не обязательно должны быть php-файлами: они могут содержать только обычные HTML-коды и иметь расширения htm или html.

Допустим, что пользователь со страницы page0.php передал на сервер значение поля с именем `login`. Тогда содержимое файла redirect.php может иметь вид, например, как в листинге 6.9.

Листинг 6.9. Переадресация в зависимости от полученных данных

```
<?php
if (strtolower($_REQUEST['login']) == "admin"){
       header("Location: http://www.myserver.ru/page1.php");
}else{
       header("Location: http://www.myserver.ru/page2.php");
}
?>
```

Суперглобальный массив `$_REQUEST` позволяет не привязываться к конкретному методу передачи данных из формы. Функция `strtolower()` применена для того, чтобы результат проверки не зависел от регистра, в котором были введены данные.

> ***Внимание!***
>
> Особенность функции `header()` состоит в том, что ее применение должно предшествовать всему, что отправляется браузеру клиента. Кроме того, если функция `header("Location: URL-адрес")` была вызвана, то последующий PHP-код, отправляющий что-либо браузеру, не будет выполнен, а текст и/или HTML-код не будет отправлен.

Функцию `header()` нельзя использовать после текста, HTML-кода и операторов `echo`. Другими словами, если что-то уже было выведено в выходной

документ, отображаемый браузером, то вызов функции `header()` приведет к предупреждающему сообщению о невозможности изменить заголовок, поскольку он уже отправлен.

Пример неправильного использования функции `header()`

```
<?php
echo "Ваше регистрационное имя - {$_REQUEST['login']}";
if (strtolower($_REQUEST['login']) == "admin"){
       header("Location: http://www.myserver.ru/page1.php");
}else{
       header("Location: http://www.myserver.ru/page2.php");
}
?>
```

Примечание

Вообще говоря, функция `header()` предназначена для дополнения или изменения HTTP-заголовков, передаваемых браузеру вместе с данными.

В листинге 6.10. приведен в качестве примера сценарий, который перенаправляет пользователя на одну страницу, если браузер клиента — Microsoft Internet Explorer, и на другую — в противном случае. Сведения о браузере клиента, который обратился к данному файлу со сценарием, можно взять из `$_SERVER['HTTP_USER_AGENT']`, в виде строки, например, такой

```
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; MRA 4.3
(build 01218))
```

Листинг 6.10. Переадресация в зависимости от браузера клиента

```
<?php
if (strpos($_SERVER['HTTP_USER_AGENT'],"MSIE")>0){
        header("Location: http://www.myserver.ru/page1.php");
}else{
        header("Location: http://www.myserver.ru/page2.php");
}
?>
```

6.2.4. Добавление информации к URL-адресу

Информация, которую можно добавить к URL-адресу, состоит из элементов вида `имя_переменной=значение`. Перед именем переменной символ `$` не ука-

зывают. Если таких элементов несколько, то их разделяют символом `&`. Всю последовательность элементов отделяют от URL-адреса знаком вопроса (`?`):

```
<a href=page1.php?name=Вадим&email=vadim@xyz.ru>Страница 1</a>.
```

Информация, добавляемая к URL-адресу, передается методом `GET` и в серверном сценарии может быть получена посредством суперглобальных массивов `$_GET` и `$_REQUEST`.

Пусть, например, текущая страница сгенерирована следующим кодом:

```
<?php
echo "<a href = page1.php?ip={$_SERVER['REMOTE_ADDR']}
&name=Вадим&email=vadim@xyz.ru>
Страница 1</a>";
?>
```

Если в файле page1.php записать код `<?php var_dump($_GET) ?>`, то в окне браузера отобразится примерно следующее:

```
array(3) {
           ["ip"]=> string(9) "127.0.0.1"
           ["name"]=> string(5) "Вадим"
           ["email"]=> string(12) vadim@xyz.ru
 }
```

Здесь `$_SERVER['REMOTE_ADDR']` — IP-адрес компьютера, на котором просматривают страницу, сгенерированную данным сценарием.

Не следует забывать, что добавляемая к URL-адресу информация отображается в адресной строке браузера и может быть в нем сохранена (в папке Избранное, например). Поэтому таким способом не рекомендуется передавать пароли и другие конфиденциальные сведения. Кроме того, объем передаваемых методом `GET` данных ограничен.

6.2.5. Применение cookie

О назначении и применении cookie в клиентских сценариях на языке JavaScript рассказывалось в *разд. 4.7*. В PHP значения cookie-записей устанавливают встроенной функцией `setcookie()`, а читают — с помощью суперглобального массива `$_COOKIE`.

Функция `setcookie(name, value, expires, path, domain, secur)` принимает следующие параметры:

- `name` — строка, содержащая имя cookie-записи (обязательный параметр);
- `value` — строка, содержащая значение cookie;

- `expires` — срок хранения cookie-записи в секундах; если отсутствует, то после завершения работы браузера cookie-запись удаляется; время жизни следует задавать с помощью функци `time()` или `mktime()` (см. *разд. 4.7*). Например, если необходимо установить время жизни cookie-записи 2 часа, начиная от момента ее создания, то значение данного параметра следует задать как `time()+2*60*60`;
- `path` — строка, содержащая путь cookie-записи;
- `domain` — строка, содержащая домен нужной cookie-записи;
- `secure` — логическое значение (`true` или `false`), определяющее необходимость использования безопасного HTTP-соединения.

В выражении функции `setcookie()` обязательный только первый параметр — имя cookie-записи. Для создания cookie-записи достаточно указать непустое значение второго параметра.

Чтобы удалить cookie-запись, достаточно установить для нее пустое значение:

```
setcookie("mycookie");
setcookie("mycookie", "");
```

> ***Внимание!***
>
> Функция `setcookie()` имеет такую же особенность, что и `header()`: ее применение должно предшествовать всему, что отправляется браузеру. Так, ее нельзя вызывать после текста, HTML-кода и операторов `echo`.
>
> Данные cookie недоступны из PHP-сценария на той же странице, где они были созданы. Чтобы получить к ним доступ с помощью суперглобального массива `$_COOKIE`, следует перейти на другую страницу либо перезагрузить текущую. Это не обременительно, поскольку cookie для того и существуют, чтобы сохранять данные, доступные для разных страниц. Тем не менее, доступ к cookie все-таки возможен и на той же странице, на которой они были созданы, но из клиентского сценария.

Для проверки возможности применения механизма cookie убедитесь, что в вашем браузере он не отключен, и создайте два файла — setcookie.php и readcookie.php, как в листингах 6.11 и 6.12 соответственно. Затем загрузите в браузер файл setcookie.php, который установит cookie-запись и выполнит переход к файлу readcookie.php. Последний прочитает cookie-запись и отобразит ее значение.

Листинг 6.11. Установка cookie-записи (файл setcookie.php)

```
<?php
setcookie("test","Привет!"); // установка cookie-записи с именем 'test'
header("Location: readcookie.php");     // переход
?>
```

Листинг 6.12. Чтение cookie-записи (файл readcookie.php)

```
<?php
echo "Значение cookie-записи с именем test: {$_COOKIE['test']}";
?>
```

6.2.6. Сеансы

Сеанс (сессия, session) в PHP — механизм, который позволяет обеспечить доступность данных, созданных на различных серверных страницах в рамках одного сайта, для других страниц того же сайта. Технология сеансов перекрывает возможности и преодолевает ограничения cookie по обмену данными между Web-страницами. Сеанс создается и закрывается, когда это необходимо. В интервале времени между этими событиями все PHP-сценарии, расположенные в различных файлах и подключенные к данному сеансу, имеют доступ к переменным, созданным в других сценариях этого же сайта.

Создание сеанса

Чтобы воспользоваться возможностями, предоставляемыми механизмом сеанса, его необходимо создать или подключиться к нему. Сеанс создается с помощью встроенной функции `session_start()` на каждой серверной странице, которую требуется подключить к сеансу. Если необходимо, чтобы все страницы имели доступ ко всем данным, сгенерированным на любой их них, то функцию `session_start()` следует вызвать на каждой странице.

Функция `session_start()` не имеет параметров и всегда возвращает `true`.

> ***Внимание***
>
> Вызов функции `session_start()` должен предшествовать выводу какой-либо информации в браузер.

Рассмотрим механизм создания сеанса.

При вызове в сценарии функции `session_start()` интерпретатор PHP проверяет, был ли создан сеанс ранее. Если сеанс уже создан на какой-либо странице сайта, то переменные сеанса (созданные на других страницах) устанавливаются и для данного сценария. Если сеанс еще не существует, то создается новый сеанс, для которого автоматически генерируется уникальный идентификатор (длинная строка символов), который присваивается глобальной предопределенной переменной `$PHPSESSID`. Таким образом, вызов функции `session_star()` либо создает сеанс и подключает к нему сценарий, либо только подключает сценарий, если сеанс уже имеется.

В пределах существующего сеанса (после вызова функции `session_star()`) можно установить сеансовые переменные. Для этого служит суперглобальный массив `$_SESSION`. Присвоить значение переменной можно так:

```
$_SESSION['имя_переменной'] = значение;
```

а получить значение уже созданной переменной так:

```
$_SESSION['имя_переменной'].
```

Чтобы удалить сеансовую переменную, можно применить функцию

```
unset($_SESSION['имя_переменной']);
```

Все переменные сеанса сохраняются на сервере в файле с именем, совпадающим с идентификатором сеанса. Можно самому определить папку для хранения файлов сеансов. Такая предварительно созданная папка задается директивой `session.save_path` в конфигурационном файле php.ini. Файл сеанса доступен для всех сценариев в пределах данного сеанса.

Интерпретатор PHP передает идентификатор сеанса каждой странице. Передача данных между сервером и клиентом происходит с использованием cookie или без него. Механизм cookie применяется, если пользователь браузера не отключил его. В противном случае работа определяется директивой `session.use_trans_sid` в конфигурационном файле php.ini, о чем будет рассказано далее.

> ***Внимание!***
>
> Можно заставить интерпретатор автоматически добавлять функцию `session_start()` в начале каждой серверной страницы. Для этого необходимо в конфигурационном файле php.ini установить директиву `session.auto_start = 1`.

Получить идентификатор текущего сеанса можно не только как значение переменной `$PHPSESSID`, но и с помощью функции `session_id()`. Эта функция возвращает строку, содержащую идентификатор сеанса, или пустую строку, если сеанс не был создан. Кроме того, передав этой функции строку символов в качестве параметра, можно установить соответствующий идентификатор для текущего сеанса.

В ряде случаев требуется принудительно закрыть сеанс. Например, при организации посещения сайта через регистрационное имя (login) и пароль (password) для каждого посетителя создается отдельный сеанс, который следует закрыть после проверки этих данных. Для этого служит встроенная функция `session_destroy()`, удаляющая все переменные сеанса и прекращающая передачу его идентификатора. Однако в текущем сценарии эти переменные остаются доступными, для их удаления следует воспользоваться функцией `unset()`.

Особенности сеансов

В предыдущем разделе уже отмечалось, что работа механизма сеансов определяется тем, включена ли поддержка cookie на стороне клиента, а также значением директивы `session.use_trans_sid` в файле php.ini. Рассмотрим эти особенности:

- **Поддержка cookie включена** — в этом случае глобальной переменной `$PHPSESSID` присваиваивается значение идентификатора сеанса, а cookie используется для передачи значения `$PHPSESSID` между сценариями (страницами);
- **Поддержка cookie отключена** — интерпретатор PHP создает константу `SID`, значением которой является `"PHPSESSID=идентификатор_сеанса"`. Использование идентификатора сеанса определяется значением директивы `session.use_trans_sid`. По умолчанию оно равно нулю (директива отключена). Чтобы включить данную директиву, необходимо в файле php.ini изменить соответствующую строку: `session.use_trans_sid = 1`. При включенной директиве можно создавать сценарии, не зависящие от настройки браузера по применению cookie. Рассмотрим различные варианты директивы `session.use_trans_sid`:
  - **session.use_trans_sid = 1** — идентификатор сеанса при отключенной поддержке cookie пересылается посредством элементов, в которых указан URL-адрес. При переходе на другую страницу с помощью ссылки, функции `header()` или формы с методом `GET` идентификатор сеанса добавляется к URL-адресу и виден в адресной строке браузера, что иногда нежелательно.

    ЗАМЕЧАНИЕ

    Следует иметь в виду, что идентификатор сеанса добавляется к URL-адресам, соответствующим одному и тому же сайту. К URL-адресу, содержащему имя сервера, идентификатор сеанса не будет добавлен. Например, для ссылки `<a href="nextpage.php">` идентификатор будет добавлен к адресу, а для `<a href="http://myserver.ru/nextpage.php">` — нет.

    Однако можно передать идентификатор сеанса в скрытых полях формы методом `POST`, при котором идентификатор сеанса не добавляется к URL-адресу: `<input type=hidden name='PHPSESSID' value='$PHPSESSID'>`.

  - **session.use_trans_sid = 0** — идентификатор сеанса при отключенной поддержке cookie не пересылается. При необходимости разработчик приложения делает это самостоятельно, используя константу `SID`.

    ЗАМЕЧАНИЕ

    Поскольку эта константа имеет значение вида `"PHPSESSID=идентификатор_сеанса"`, то можно просто добавить ее к URL-адресу, например, `<a href="mainpage.php?<?php echo SID?>">Главная страница</a>`. Однако при этом идентификатор сеанса, видимый в адресной строке браузера, может быть использован злоумышленниками в неблаговидных целях.

Пример организации сеанса

Рассмотрим тестовый пример организации сеанса. Для его воспроизведения убедитесь, что в вашем браузере не отключена поддержка cookie, или установите директиву `session.use_trans_sid = 1` в файле php.ini. Чтобы изменения в php.ini вступили в силу, необходимо перезапустить Web-сервер.

В приведенном далее примере имеются две серверных страницы, каждая из которых вносит собственный вклад в формирование множества сеансовых переменных. Сначала первая страница создает две переменные, затем вторая страница добавляет еще одну переменную и, наконец, все три переменные становятся доступными для обеих страниц.

В листинге 6.13 приведен код, формирующий первую страницу (файл s1.php). Здесь создается сеанс и две сеансовые переменные — `myname` и `myaddr`. Однако сценарий пытается вывести значение переменной `mylastname`, которая еще не была создана. Поэтому на странице в браузере (рис. 6.4, слева вверху) это значение не будет отображаться.

Листинг 6.13. Применение сеанса (первая страница)

```
<?php
     session_start();// открытие сеанса
     /* Создание переменных сеанса */
     $_SESSION['myname']="Вадим";
     $_SESSION['myaddr']="Санкт-Петербург";
?>
<html>
<h2>Тестирование сеансов</h2>
<b>Страница 1</b><br>
<a href=s2.php>Перейти на страницу 2</a>
</html>
<?php
     /* Вывод переменной сеанса */
     echo "<br>";
     echo "Фамилия: {$_SESSION['mylastname']}";
?>
```

В результате щелчка на ссылке будет загружена вторая страница (файл s2.php), сформированная кодом листинга 6.14. Здесь создается только одна переменная `mylastname`, которой еще не было при загрузке первой страницы, и выводятся значения всех трех переменных текущего сеанса (рис. 6.4, справа вверху). Таким образом, одна переменная была создана на второй странице, а две других — переданы ей из первой. Если на данной странице щелкнуть

на ссылке, произойдет возвращение к первой странице, на которой теперь будут доступными все переменные сеанса. Поэтому значение переменной `mylastname` отобразится в браузере (рис. 6.4, внизу).

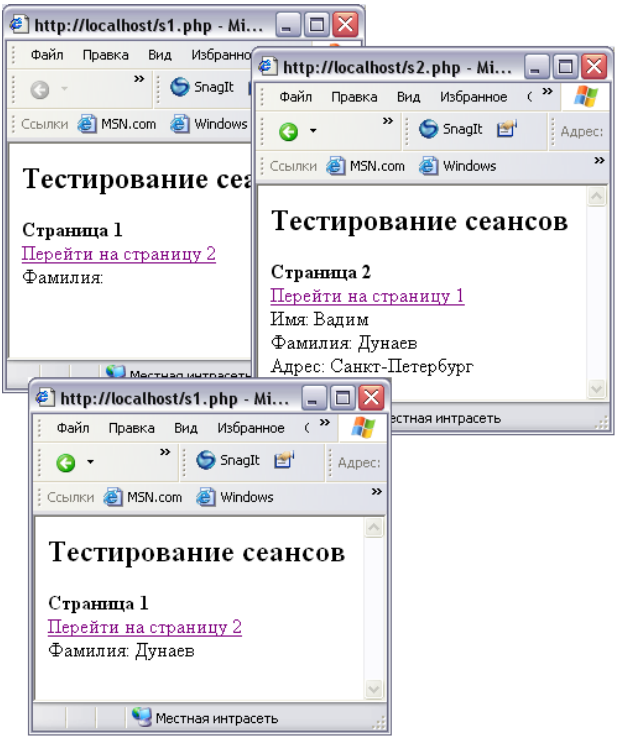

Рис. 6.4. Пример использования сеанса

Листинг 6.14. Применение сеанса (вторая страница)

```
<?php
     session_start();// подключение к сеансу
     /* Создание переменной сеанса */
     $_SESSION['mylastname']="Дунаев";
?>
<html>
<h2>Тестирование сеансов</h2>
<b>Страница 2</b><br>
<a href=s1.php>Перейти на страницу 1</a>
</html>
```

```
<?php
     /* Вывод переменных сеанса */
     echo "<br>";
     echo "Имя: {$_SESSION['myname']}<br>";
     echo "Фамилия: {$_SESSION['mylastname']}<br>";
     echo "Адрес: {$_SESSION['myaddr']}";
?>
```

Защита страниц паролем

Теперь рассмотрим в качестве примера задачу защиты страниц сайта паролем и/или требованием регистрации. С помощью сеанса можно потребовать от пользователя зарегистрироваться или ввести пароль только один раз, а не делать это при посещении каждой страницы сайта. В общих чертах алгоритм действий может быть такой:

1. Отобразить в браузере страницу, содержащую форму для ввода регистрационного имени и/или пароля.
2. Если регистрация по вашему критерию прошла успешно, создать сеанс с соответствующими переменными.
3. При переходе пользователя на другую страницу проверить успешность регистрации на основе соответствующих сеансовых переменных. Если пользователь зарегистрировался правильно, то загрузить ему требуемую страницу, а в противном случае — предложить ему заново зарегистрироваться, т. е. загрузить страницу с элементами ввода регистрационных данных (см. п. 1).

Рассмотрим два варианта реализации данного алгоритма.

В первом варианте серверная страница pwinput.php, на которой пользователь вводит пароль, может иметь код, представленный в листинге 6.15.

Листинг 6.15. Код для ввода пароля (pwinput.php). Вариант 1

```
<?php
/* Страница для ввода пароля */
session_start();// открытие сеанса
echo "<form action='nextpage.php' method=POST>Введите пароль:
<input type=password name='userpw'>
<input type=submit value='OK'>
</form>";
?>
```

Здесь пароль, введенный пользователем в поле, маскируемое символами "звездочка", передается на страницу nextpage.php методом `POST`. Код этой страницы может иметь вид, представленный в листинге 6.16.

Листинг 6.16. Код с проверкой правильности пароля (nextpage.php). Вариант 1

```
<?php
/* Проверка правильности пароля */
session_start(); // подключение к текущему сеансу
if (!isset($_SESSION['userpw'])) {
     $_SESSION['userpw']=$_POST['userpw'];
}
$truepw="abracadabra";                  // верный пароль
if ($_SESSION['userpw'] !=$truepw) {  // если пароль не верен
     session_destroy();                  // закрываем сеанс
     header("Location: pwinput.php");  // вернуться к вводу пароля
} else {                            // если пароль верен
     $_SESSION['userpw']=$truepw;
}
?>
<! Здесь может быть какой-нибудь HTML-документ или PHP-код >
```

Сценарий в листинге 6.16 проверяет правильность пароля и, если он не верен, производит переход к странице для ввода пароля. Рассмотрим этот сценарий подробнее.

Поскольку пароль вводится пользователем в поле формы и передается методом `POST`, при первой его проверке сеансовая переменная `$_SESSION['userpw']` еще не установлена. В этом случае мы создаем ее, присваивая полученное от пользователя значение `$_POST['userpw']`.

Далее происходит сравнение полученного значения пароля с правильным. Если они не совпадают, то мы закрываем сеанс и возвращаемся на страницу ввода пароля. Заметьте, что если не закрыть сеанс, то при следующем обращении к странице проверки пароля новое, полученное от пользователя, значение пароля не присвоится сеансовой переменной `$_SESSION['userpw']`, поскольку она уже установлена. В результате даже при правильно введенном пароле последующий код сценария всегда переадресовывал бы пользователя на страницу ввода пароля.

Если полученный от пользователя пароль верен, то его значение присваивается переменной `$_SESSION['userpw']` и, далее, отображается содержимое данной страницы (в листинге 6.16 оно не указано).

Код, приведенный в листинге 6.16, следует добавить в каждую серверную страницу, которую вы хотите защитить паролем. Можно сохранить сценарий в отдельном файле и вставлять его содержимое в защищаемые страницы с помощью функции `include()`. В любом случае для перемещения по страницам сайта пользователю достаточно ввести верный пароль только один раз.

Во втором варианте (листинг 6.17) на серверной странице pwinput.php форма для ввода пароля отображается, только если пользователь еще не ввел верный пароль. Кроме того, на этой же странице выводится ссылка на страницу nextpage.php. Щелчок на кнопке **OK** типа `submit` отправляет данные формы не на nextpage.php, как в первом варианте, а на ту же самую страницу pwinput.php. Если отправленный пароль верен, то форма на этой странице не отображается. Вместо нее выводится сообщение об успешной регистрации. Обратите внимание, что на странице pwinput.php сценарий сначала проверяет значение сеансовой переменной `$_SESSION['userpw']`. Поскольку она может еще не существовать или содержать неверный пароль, то проверяется значение поля `userpw`, полученное при передаче из формы. Если значение `$_POST['userpw']` совпадает с верным паролем, то оно присваивается сеансовой переменной, которая далее используется в сценарии проверки правильности пароля на всех защищаемых страницах и, в частности, на странице nextpage.php (листинг 6.18).

Листинг 6.17. Код для ввода пароля (pwinput.php). Вариант 2

```
<?php
/* Страница для ввода пароля */
     session_start();// открытие сеанса
$truepw="abracadabra";     // верный пароль
if ($_SESSION['userpw']!=$truepw) {
     if ($_POST['userpw']!=$truepw){
          echo "<form action= pwinput.php' method=POST>
               Введите пароль:
          <input type=password name='userpw'>
          <input type=submit value='OK'>
          </form>";
     } else {
          $_SESSION['userpw']=$truepw;
          echo "Регистрация прошла успешно";
     }
}
echo "<br><a href=s8.php>Следующая страница</a>";
?>
```

Листинг 6.18. Код с проверкой правильности пароля (nextpage.php). Вариант 2

```
<?php
/* Проверка правильности пароля */
session_start();           // подключение к текущему сеансу
$truepw="abracadabra";     // верный пароль
if ($_SESSION['userpw'] !=$truepw) {      // если пароль не верен
     header("Location: pwinput.php");      // вернуться к вводу пароля
}
?>
<! Здесь может быть какой-нибудь HTML-документ или PHP-код >
```

Сценарий, приведенный в листинге 6.18, можно сохранить в отдельном файле и с помощью функции `include()` вставить в каждую серверную страницу, которую вы хотите защитить паролем.

6.3. Работа с графикой

В PHP имеется библиотека функций для работы с графическими изображениями. С помощью этих функций можно осуществлять различные преобразования изображений из графических файлов: масштабирование, повороты, наложение одного изображения на другое, вставку текста в изображение и др. Создание и обработка изображений на стороне сервера очень важны, например, при создании баннеров и счетчиков посещений Web-страниц. В данном разделе мы рассмотрим лишь несколько приемов работы с графикой.

Для работы с графикой необходимо подключить библиотеку графических функций (для Windows — файл php_gd2.dll). Для различных графических форматов (GIF, PNG, JPEG и др.) в PHP существуют свои функции. Здесь лишь покажем на примерах их использование, отсылая за подробностями к справочному руководству по PHP.

Для отображения в браузере графического изображения из файла формата JPEG можно воспользоваться сценарием, представленным в листинге 6.19.

Листинг 6.19. Вывод изображения

```
<?php
header("Content-type: image/jpeg"); // отправка HTTP-заголовка
/* Сздание ресурса изображения из файла: */
$im=imageCreateFromjpeg("picture.jpg");
imagejpeg($im);     // вывод изображения
?>
```

Здесь с помощью функции `header("Content-type: image/jpeg")` браузеру передается MIME-тип посылаемой информации. Функция `imageCreateFromjpeg()` создает изображение на основе указанного файла формата JPEG и возвращает идентификатор этого ресурса. Функция `imagejpeg()` направляет изображение с указанным идентификатором для отображения. Для GIF-файла предусмотрены аналогичные функции `imagecreatefromgif()` и `imagegif()`, а для PNG-файла — `imagecreatefrompng()` и `imagepng()`.

В сценарий, выводящий графическое изображение, нельзя вставлять операторы `echo`, а в файле с этим сценарием недопустимы HTML-теги. Это ограничение связано с применением функции `header()`, которая должна предшествовать любому выводу данных.

В листинге 6.20 приведен сценарий, который изменяет ширину и высоту изображения в заданное число раз.

Листинг 6.20. Масштабирование изображения

```
<?php
header("Content-type: image/jpeg"); // отправка HTTP-заголовка
$k=2.5; // коэффициент масштабирования
/* Массив размеров исходного изображения (ширина, высота): */
$size=getImageSize("picture.jpg");
/* Новые размеры изображения: */
$w=$k*$size[0];
$h=$k*$size[1];
/* Создаем новое пустое изображение (подложку): */
$new=imageCreateTrueColor($w,$h);
/* Создаем изображение из файла: */
$im=imageCreateFromjpeg("picture.jpg ");
/* Копируем изображение из файла на подложку с масштабированием: */
imageCopyResampled($new,$im,0,0,0,0,$w,$h,$size[0], $size[1]);
imagejpeg($new);      // вывод изображения
?>
```

Вы можете повернуть изображение на произвольный угол и задать цвет фонового прямоугольника, не занятого изображением после поворота. Пример соответствующего сценария приведен в листинге 6.21.

Листинг 6.21. Поворот изображения

```
<?php
header("Content-type: image/jpeg"); // отправка HTTP-заголовка
$angle=25;     // угол поворота, град.
```

```
/* Создаем изображение из файла: */
$im=imageCreateFromjpeg("picture.jpg");
/* Цвет фона для изображения $im (сотавляющие RGB): */
$bgcolor=imageColorAllocate($im,180,180,180);
/* Поворот изображения: */
$im2=imageRotate($im,$angle,$bgcolor);
imagejpeg($im2);     // вывод изображения
?>
```

Положительные значения угла соответствуют повороту против часовой стрелки, а отрицательные — по часовой стрелке.

В листинге 6.22 приведен сценарий, который вставляет одно изображение в другое.

Листинг 6.22. Вставка одного изображения в другое

```
<?php
header("Content-type: image/gif");     // отправка HTTP-заголовка
/* Создание фонового изображения: */
$im1=imageCreateFromgif("picture1.gif");
/* Создание вставляемого изображения: */
$im2=imageCreateFromgif("picture2.gif");
/* Копирование изображения $im2 в $im1: */
imageCopy($im1,$im2,250,100,0,0,150,97);
imagegif($im1); // вывод изображения
?>
```

Здесь функция `imageCopy()` принимает следующие параметры:

- изображение, в которое требуется скопировать другое изображение;
- копируемое изображение;
- горизонтальную и вертикальную координаты точки копирования;
- горизонтальную и вертикальную координаты верхнего левого угла участка копируемого изображения;
- ширину и высоту участка копируемого изображения.

Координаты, ширину и высоту задают в пикселах.

В графическое изображение с помощью функции PHP `imagettftext(изображение, размер_шрифта, угол, x, y, цвет, имя_шрифта, текст)` можно вставить текст, задав имя, размер и цвет шрифта, а также координаты и угол наклона

строки. Однако у этой функции есть следующая особенность. Если десятичный код символа текстовой строки не превосходит 127, то его можно указать непосредственно. Другие символы следует задавать в виде `&#код;`. Так например, кириллическая буква А имеет код 1040, и ее в строке следует записывать в виде `А`. В листинге 6.23 представлена функция `str_rus(текст)`, которая возвращает текстовую строку, полученную из указанной в качестве параметра путем замены кириллических символов на специальные. Это не встроенная функция PHP, а пользовательская.

Листинг 6.23. Функция представления текста специальными символами

```
function str_rus($str){
$nstr="";     // выходная строка
for ($i=0; $i<strlen($str);$i++){
     $symbol=substr($str,$i,1); // текущий символ строки
     $asci=ord($symbol);     // ASCII-код
     if ($asci<128) {
          $nstr.=$symbol;
     } elseif ($asci>191 and $asci<256){
          $nstr.="&#".(string)(848+ord($symbol)).';';
     } else {
          $nstr.=$symbol;
     }
}
return $nstr;
}
```

Пример сценария вставки текста в графическое изображение приведен в листинге 6.24.

Листинг 6.24. Вставка текста в изображение

```
<?php
header("Content-type: image/gif");          // отправка HTTP-заголовка
/* Создаем изображение из файла: */
$im = imageCreateFromgif("picture.gif");
/* Цвет текста для изображения $im (составляющие RGB): */
$color = imagecolorallocate($im, 240, 0, 0);
$font="times.ttf";       // имя шрифта
```

```
$fontsize=80;              // размер шрифта
$angle=-6;                 // угол наклона текстовой строки, град.
/* Координаты начала строки: */
$x=40;
$y=80;
/* Текст, который вставляется в изображение: */
$text=str_rus("Поздравляю\nс днем рождения");
/* Вставка текста в изображение: */
imagettftext($im, $fontsize, $angle, $x, $y, $color, $font, $text);
imagegif($im);       // вывод изображения с текстом

/* Определение функции представления теста спец. символами: */
function str_rus($str){
… // код листинга 6.24
}
?>
```

Линию можно нарисовать путем многократного копирования изображения точки в фоновое изображение, изменяя координаты положения копии. Несмотря на то, что в качестве фона можно выбрать любой графический файл, мы его создадим автономно с помощью функции `imageCreateTrueColor()`.

В листинге 6.25 приведен сценарий, рисующий несколько линий на сером фоне (рис. 6.5). Точкой служит изображение размером 1×1 пиксел из файла point.jpg, который можно создать в любом графическом редакторе.

Листинг 6.25. Рисование линий

```
<?php
header("Content-type: image/gif");      // отправка HTTP-заголовка
$width=300;     // ширина фона
$height=200;    // высота фона
$n=2;           // толщина линии
/* Создание фонового изображения: */
$im1=imageCreateTrueColor($width, $height);
/* Цвет фона (составляющие RGB): */
$bgcolor=imageColorAllocate($im1, 200, 200, 200);
/* Заливка фона цветом $bgcolor: *.
imagefill($im1, 0, 0, $bgcolor); //
/* Создание изображения точки: */
$im2=imageCreateFromjpeg("point.jpg");
```

```
/* Рисование линий: */
for ($x=0; $x<250; $x++){
        /* Горизонтальная прямая: */
        imageCopy($im1, $im2, $x, 50, 0, 0, $n, $n);
        /* Наклонная прямая: */
        imageCopy($im1, $im2, $x, 2*$x, 0, 0, $n, $n);
        /* Синусоида: */
        imageCopy($im1, $im2, $x, 50 + 20*sin(0.1*$x), 0, 0, $n, $n);
        /* Окружность: */
        imageCopy($im1, $im2, 250 + 30*cos(0.1*$x),
                  150 + 30*sin(0.1*$x), 0, 0, $n+1, $n+1);
        /* Эллипс: */
        imageCopy($im1, $im2, 150 + 50*cos(0.1*$x),
                  150 + 0*sin(0.1*$x + 3.5), 0, 0, $n, $n);
}
imagegif($im1); // вывод изображения
?>
```

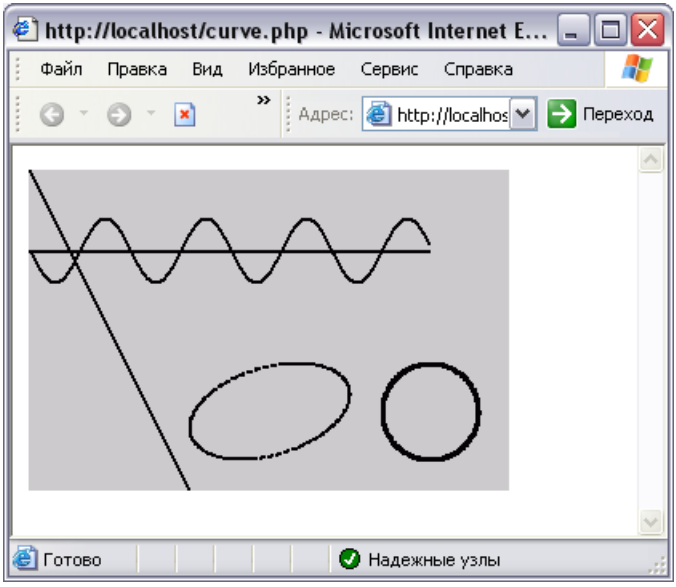

Рис. 6.5. Рисование линий

Заменив точку более сложными и крупными изображениями, можно получить интересные визуальные эффекты.

При вызове файла с рассмотренными сценариями в окне браузера появится изображение и ничего более. Не забывайте, что дополнять файлы со сценариями, выводящими изображения, операторами `echo` и/или HTML-тегами нельзя из-за особенностей функции `header()`.

Встроить графическое изображение в Web-страницу на стороне сервера позволяет HTML-тег `<img src=файл_сценария>`, атрибут `src` которого — адрес файла со сценарием, формирующим изображение. С помощью других атрибутов можно задать размеры изображения и способ обтекания его текстом.

В листинге 6.26 приведен пример Web-страницы (HTML-файла), в которую вставляется графическое изображение, сформированное серверным сценарием из файла mypicture.php. Данный код может быть сохранен в HTML- или PHP-файле.

Листинг 6.26. Вставка в Web-страницу изображения, сформированного сценарием

```
<html>
<H1>Моя страница</H1>
Это графическое изображение сформировано
<IMG SRC=mypicture.php WIDTH=200 HEGHT=100 ALIGN=RIGHT>
сценарием из файла mypicture.php и вставлено в страницу посредством
тега < IMG SRC = mypicture.php>
</html>
```

6.4. Работа с файлами

Длительное хранение данных, как хорошо известно, организуется с помощью файловой системы. Довольно часто для этого используют обычные текстовые файлы. Вот типичная последовательность действий при работе с файлом:

1. Открыть файл.
2. Прочитать данные из файла или записать их туда.
3. Закрыть файл.

С открытым файлом связано понятие указателя. При выполнении различных файловых операций указатель может перемещаться, отмечая место, с которого можно получить доступ к содержимому файла. Например, если указатель находится в начале файла, то операция записи новых данных в этот файл будет уничтожать содержащиеся в нем предыдущие. Если указатель находится в конце файла, то записываемые в него данные будут добавляться к уже имеющимся.

6.4.1. Открытие файла

Файл открывается с помощью встроенной функции `fopen(имя_файла, режим)`, которая возвращает дескриптор (идентификатор) открытого файла или `false` — при неудаче. Дескриптор файла используется в сценарии для обращения к открытому файлу как к ресурсу. Параметры имеют строковый тип.

Первый параметр — имя или URL-адрес файла. Если путь к файлу не задан, то предполагается текущая папка, т. е. та, в которой расположен файл со сценарием. Все папки, указанные в пути, должны существовать, а вы должны иметь к ним и открываемому файлу право доступа.

> ***Внимание!***
>
> В Windows путь указывают либо одинарные прямые, либо двойные обратные слэши, например, C:/myfolder/myfile.txt, C:\\myfolder\\myfile.txt.

Второй параметр определяет режим доступа к открываемому файлу и может принимать следующие строковые значения:

- `'r'` — только чтение; указатель в начале файла; если файл не существует, то функция `fopen()` вернет `false` и выведет предупреждение. Таким образом, это режим чтения файла с начала;
- `'r+'` — чтение и запись; указатель в начале файла; если файл не существует, то функция `fopen()` вернет `false` и выведет предупреждение. Таким образом, это режим чтения файла с начала и перезаписи данных;
- `'w'` — только для записи; указатель в начале файла; если файл существует, то он обрезается до нулевой длины, в противном случае — делается попытка его создать. Таким образом, это режим только перезаписи данных;
- `'w+'` — чтение и запись; указатель в начале файла; если файл существует, то он обрезается до нулевой длины, в противном случае — делается попытка его создать. Таким образом, это режим чтения и перезаписи данных;
- `'a'` — только запись; указатель в конце файла; если файл не существует, то делается попытка его создать. Таким образом, это режим только добавления данных;
- `'a+'` — чтение и запись; указатель в конце файла; если файл не существует, то делается попытка его создать. Таким образом, это режим чтения и добавления данных.
- `'x'` — файл создается и открывается только для записи с указателем в начале файла. Если файл уже существует, то функция `fopen()` вернет `false` и выведет предупреждение;

- `'x+'` — файл создается и открывается для чтения и записи с указателем в начале файла. Если файл уже существует, то функция `fopen()` вернет `false` и выведет предупреждение.

Например, открыть уже существующий или создать (с последующим открытием) файл mytest.txt в режиме чтения и перезаписи можно так:

```
$fh=fopen("mytest.txt", "w+");
```

В данном случае предполагается, что файл расположен или будет создан в текущей папке. Для той же операции над файлом в папке C:\myfolder следует применить такое выражение:

```
$fh=fopen("C:/myfolder/mytest.txt", "w+");
```

Однако в этом случае предполагается, что указанная папка существует, и вы имеете право доступа к ней.

При открытии существующего файла в режиме `w+` его содержимое уничтожается.

> ***Внимание!***
>
> Вывод предупреждающих сообщений в случаях возвращения функцией `fopen()` значения `false` при неудачах открытия файла не прерывает выполнения последующих выражений сценария, что, однако, может привести к его некорректной работе.

Подавить вывод предупреждающего сообщения при неуспешном применении функции `fopen()` можно с помощью выражения `@fopen()`. Если же требуется предотвратить последующее выполнение сценария, то воспользуйтесь такой конструкцией:

```
$fh=fopen("mytest.txt", "w+r") or exit("Не удалось открыть файл");
```

При работе с файлами часто применяются функции проверки существования папки и файла:

- `file_exists(имя_файла)` — проверяет существование указанного файла и возвращает `true` или `false`;
- `is_dir(имя_файла)` — возвращает `true`, если параметр соответствует имени существующей папки; в противном случае возвращает `false`;
- `is_file(имя_файла)` — возвращает `true`, если параметр является именем существующего файла, а не папки, и `false` — в противном случае.

6.4.2. Закрытие и удаление файлов

После использования открытый файл следует закрыть. Для этого применяется функция `fclose(дескриптор)` с указанием в качестве параметра дескриптора ранее открытого файла. Данная функция возвращает `true` или `false`

в зависимости от успешности операции закрытия файла. При неудаче выводится предупреждающее сообщение.

Пример

```
$fh=fopen("mytest.txt", "w+r");
// здесь могут быть какие-то операции над открытым файлом
fclose($fh);
```

Для удаления файла с диска на сервере применяется функция `unlink(имя_файла)`, которая возвращает `true` или `false` в зависимости от успешности операции. При неудаче выводится предупреждающее сообщение. Удалять можно как открытый, так и закрытый файл.

6.4.3. Чтение файла

Содержимое файла можно записать в переменную или массив. Для этого предусмотрены специальные встроенные функции чтения файла:

- чтение файла в переменную: `fgets()`, `file_get_contents()`, `fread()`, `fgetss()`, `fgetcsv()`;
- чтение файла в массив: `file()`.

Чтение файла в переменную

Функция `fgets(дескриптор, длина)` позволяет читать содержимое файла побайтно до тех пор, пока не будет прочитано столько байтов, сколько указано в параметре `длина` минус 1 либо не будет достигнут символ конца строки или файла. Если параметр `длина` не указан, то чтение будет происходить до конца строки или файла. Данная функция возвращает прочитанную строку с добавлением символов конца строки и файла или `false`, если при чтении произошла ошибка. Напомню, что в браузере символы конца строки и файла отображаются в виде пробела.

Чтобы прочитать весь файл в переменную, достаточно выполнить код листинга 6.27.

Листинг 6.27. Построчное чтение файла в переменную

```
<?php
$filename="test.txt";
$fh=fopen($filename,"r");  // открытие файла в режиме только чтения
while (!feof($fh)) {       // пока не достигнут конец файла
```

```
    $buff .= fgets($fh);  // запись в переменную
}
fclose($fh);              // закрытие файла
echo $buff;               // вывод значения переменной
?>
```

Здесь в операторе цикла присутствует встроенная функция `feof(дескриптор)`, которая возвращает `true` при достижении указателем конца файла и `false` — в противном случае.

Запись содержимого файла в переменную можно выполнить, вызвав встроенную функцию `file_get_contents(имя_файла)`. При этом не требуется предварительно открывать файл. Данная функция имеет еще несколько необязательных параметров *(см. справочное руководство по PHP)*.

Записать в переменную заданное число байтов из открытого файла можно с помощью функции `fread(дескриптор, длина)`.

Чтение файла в массив

Нередко строки, прочитанные из файла, для последующей обработки удобно разместить в массиве. С этой целью можно воспользоваться кодом, приведенным в листинге 6.28.

Листинг 6.28. Построчное чтение файла в массив

```
<?php
$filename="test.txt";
$fh=fopen($filename,"r");  // открытие файла в режиме только чтения
while (!feof($fh)) {       // пока не достигнут конец файла
    $abuff[]=fgets($fh);   // запись в массив
}
fclose($fh);               // закрытие файла
?>
```

Вместо данного кода для построчного чтения файла в массив можно воспользоваться встроенной функцией `file(имя_файла)`.

Чтение файла с удалением тегов HTML

Иногда возникает задача прочитать файл, удалив из полученной строки все или некоторые теги HTML, а также дескрипторы `<?php` и `?>`. Например, вам может потребоваться выделить из HTML-документа только текстовое содер-

жимое, удалив графику, Flash-ролики и другие объекты. Для этого можно применить функцию `fgetss(дескриптор, длина, допустимые_теги)`, аналогичную рассмотренной ранее функции `fgets()`. Второй и третий параметры необязательные. Строковый третий параметр может содержать список тегов, которые требуется оставить в возвращаемой строке. Если он не указан, то удаляются все теги. Пример с функцией `fgetss()` приведен в листинге 6.29.

Листинг 6.29. Чтение файла в переменную с удалением тегов HTML (`fgetss()`)

```
<?php
$filename="test.txt";
$fh=fopen($filename,"r");       // открытие файла в режиме только чтения
while (!feof($fh)) {            // пока не достигнут конец файла
     $buff=$buff+fgetss($fh,1024, "<p>,<br>,<a>");
}
fclose($fh);                    // закрытие файла
echo $buff;                     // вывод значения переменной
?>
```

Здесь в переменной `$buff` сохраняется содержимое файла. При этом из значения этой переменной удаляются все теги за исключением тегов абзаца, перевода строки и ссылок. Разумеется, исходный файл остается без изменений.

Кроме функции `fgetss()` с этой же целью можно использовать функцию `strip_tags(строка, допустимые_теги)`, которая удаляет из указанной строки теги, не перечисленные во втором необязательном параметре (листинг 6.30).

Листинг 6.30. Чтение файла в переменную с удалением тегов HTML (`strip_tags()`)

```
<?php
$filename="test.txt";
$buff=file_get_contents($filename); // чтение файла в переменную
$buff=strip_tags($buff, "<p>,<br>,<a>");
echo $buff;                 // вывод значения переменной
?>
```

Здесь функция `strip_tags()` применяется к строке, которая получена из файла с помощью функции `file_get_contents()`.

6.4.4. Запись в файл

В файл, открытый в режиме, допускающем изменение его содержимого (см. *разд. 6.4.1*), можно записать данные. Для этого служит функция

```
fwrite(дескриптор, строка, длина),
```

которая возвращает количество записанных байтов, или `false` — в случае ошибки.

Записываемые данные представляют в виде строки символов. Если указан необязательный параметр `длина`, меньший чем длина строки, то будет записано заданное количество байтов, в противном случае — все данные. В листинге 6.31 приведен пример сценария записи в файл текущей даты и пояснения. При этом данные записываются в две строки. Обратите внимание, что в Windows для перевода строки используются два служебных символа — `\r\n`, а в системах семейства Unix достаточно одного — `\n`.

Листинг 6.31. Запись данных в файл

```
<?php
$fh = fopen("myfile.txt", "w+");   // открытие файла
$data=date('d.m.Y')."\r\nПробная запись в файл";     // данные
fwrite($fh,$data);                 // запись данных в файл
fclose($fh);                       // закрытие файла
?>
```

6.4.5. Работа с папками

Создать папку (каталог, директорию) можно с помощью встроенной функции `mkdir(имя_папки)`. В параметре указывают имя создаваемой папки, которое может содержать путь к ней. Функция возвращает `true` или `false` в зависимости от успешности операции.

Если имя папки содержит путь к ней, то все указанные там папки, кроме создаваемой, должны существовать. Чтобы проверить существование папки, достаточно воспользоваться функцией `is_dir(имя_папки)` (см. *разд.6.4.1*).

Листинг 6.32 содержит определение пользовательской функции `mkfolder(имя_папки)`, которая создает папку, даже если папки верхнего уровня не существует. Разумеется, все недостающие папки эта функция создает автоматически. Функция возвращает полное имя папки (включающее путь), завершающееся прямым слэшем, если она была успешно создана или уже существовала; в противном случае — `false`.

Листинг 6.32. Функция создания папки

```
<? php
function mkfolder($path){
if (empty($path)) {          // если параметр пуст
return false;
}
if (is_dir($path)) {     // если папка существует
     return $path.'/';
}
$apath=explode("/",$path); // преобразуем строку $path в массив
$x=$apath[0];
for ($i=1; $i<sizeof($apath); $i++){
     $x.="/".$apath[$i];
     if (!is_dir($x)) mkdir($x);/* если папка не существует,
 создаем ее */
}
if (is_dir($path)) {     // если папка существует
     return $path.'/';
} else {
     return false;
}
}
?>
```

Допустим, требуется создать файл myfile.txt в папке C:/folder1/folder2. При этом вы не уверены, что папки folder1 и folder2, указанные в пути, существуют. Тогда можно воспользоваться следующим выражением:

```
fopen(mkfolder("C:/folder1/folder2")."myfile.txt", "w+");
```

Удалить папку можно встроенной функцией `rmdir(имя_папки)`, которая возвращает `true` или `false` в зависимости от успешности операции. При уничтожении папки последняя должна быть пустой, а удаляющий папку должен иметь право на это действие. В качестве упражнения напишите код функции, которая удаляет папку, даже если она не пустая.

Во многих приложениях нужно знать содержимое папки. Для открытия и чтения папок в PHP имеется множество разнообразных функций. Чтобы прочитать содержимое папки, необходимо сначала открыть ее и тем самым получить к ней доступ. Функция `opendir(имя_папки)` принимает в качестве параметра имя открываемой папки (возможно с указанием пути), а возвращает ее дескриптор.

Для чтения открытой папки предназначена функция `readdir(дескриптор_папки)`, которая возвращает имя файла без указания пути или `false`, если папка пуста или достигнут ее конец. Дело в том, что вызов функции `readdir()` перемещает указатель на текущий файл как при просмотре массива. Чтобы прочитать все содержимое папки, функцию `readdir()` вызывают в цикле.

Пример

```
$dh = opendir(("C:/folder1/folder2");
while (false !==($filename=readdir($dh))) {
       echo "$filename <br>";
}
```

По окончании работы с папкой ее следует закрыть с помощью функции `closedir(дескриптор_папки)`.

6.4.6. Простой счетчик посещений страницы

Рассмотрим совместное применение функций чтения и записи в файл на примере простого счетчика посещений Web-страницы. Текущее количество посещений Web-страницы можно хранить на сервере в текстовом файле, например, mycounter.txt. При вызове страницы следует прочитать содержимое этого файла, увеличить значение счетчика на единицу и перезаписать файл mycounter.txt с новым значением. Все это можно сделать с помощью сценария, приведенного в листинге 6.33. Простота счетчика состоит в том, что мы ведем учет только общего количества посещений страницы, не различая тех, кто это сделал.

Листинг 6.33. Простой счетчик посещений страницы

```
<?php
$fh = fopen("mycounter.txt", "r"); // открыли файл со значением счетчика
$counter=fread($fh,20);           // прочитали текущее значение счетчика
fclose($fh);                      // закрыли файл
$counter++;                      // увеличили значение счетчика на 1
$fh = fopen("mycounter.txt", "w");      // открыли файл
fwrite($fh, "$counter");           // записали новое значение счетчика
fclose($fh);                      // закрыли файл
?>
```

Данный код можно сохранить в отдельном файле, например counter.inc, а затем включить его в каждую серверную страницу сайта. Тогда вызов любой из них будет увеличивать значение счетчика. Листинг 6.34 содержит код страницы, отображающей счетчик. Значение счетчика выводится в таблице с помощью функции форматирования `sprintf()` в виде числа, дополненного ведущими нулями (см. *разд. 5.6.4*).

Листинг 6.34. Страница со счетчиком посещений

```
<?php
include("counter.inc");      // см. листинг 6.33
/* Значение счетчика находится в переменной $counter */
?>
<h1>Моя Web-страница</h1>
<table border=7>
<tr><td align=center><b>Счетчик</b></td></tr>
<tr><td>
<?php echo sprintf("%'09s",$counter); ?>
</td></tr>
</table>
```

Примечание

Описанным способом можно создавать счетчики посещений страниц своего собственного сайта, если у вас есть доступ к вычислительным ресурсам PHP вашего Web-сервера. При этом вы формируете страницу со счетчиком, размещая его там, где хотите, используя HTML-теги, каскадные таблицы стилей, оператор `echo` или серверные средства работы с графикой. Вы также можете создать распространяемый счетчик, который посторонний пользователь вставит в свою страницу, не располагая PHP-поддержкой своего сайта. Такой счетчик будет использовать PHP-поддержку вашего сайта. Распространяемые счетчики обычно вставляют в страницу с помощью тега `<img>`. Подробнее об этом будет рассказано в *разд. 6.4.7*.

6.4.7. Работа с таблицами в текстовых файлах

Данные в текстовом файле могут иметь табличную структуру: строки и столбцы. В каждой строке данные, принадлежащие соседним столбцам, отделяются друг от друга некоторым символом-разделителем. Разделенные данные называются полями. Текстовые файлы с такой организацией данных называются CSV-файлами (Comma Separated Values — значения, разделенные запятой), хотя символом-разделителем может быть не только запятая. Многие прикладные системы, работающие с таблицами и базами данных (например,

MS Excel, MS Access и др.), могут экспортировать свои данные в CSV-файлы и, наоборот, импортировать CSV-файлы в свои структуры. Это обеспечивает обмен данными между различными приложениями.

В сценариях табличные данные удобно представлять двумерными массивами, поскольку в PHP имеется много функций для работы с ними. В основу таких приложений, как счетчик посещений Web-страниц, баннер, гостевая книга и др., можно положить несколько функций, обеспечивающих работу с таблицами. В данном разделе мы сначала рассмотрим эти функции, а затем — их применение в перечисленных ранее приложениях. Несмотря на различное назначение и некоторую специфику этих приложений, они имеют много общего — единый архетип. Вы это заметите.

Чтение CSV-файла

Текстовый файл (CSV-файл) с табличной организацией данных для последующей обработки нужно сначала прочитать. Для чтения из текстового файла данных, разделенных некоторым символом (не только запятой) служит встроенная функция

```
fgetcsv(дескриптор, длина, разделитель, ограничитель).
```

Данная функция, подобно рассмотренной в *разд. 6.4.3* функции `fgets()`, читает строку открытого файла, но дополнительно анализирует ее и возвращает массив данных, отделенных друг от друга в этой строке указанным разделителем. При возникновении ошибки, а также при достижении конца файла, функция возвращает `false`. Таким образом, функция выделяет поля в строке CSV-файла и записывает их содержимое в массив.

Первый параметр функции `fgetcsv(дескриптор, длина, разделитель, ограничитель)` является дескриптором открытого файла. Второй и последующие параметры не обязательны. Если параметр `длина` не указан или равен нулю, то предполагается, что строка может быть сколь угодно длинной, однако при этом скорость анализа уменьшается по сравнению со случаем, когда этот параметр больше нуля. Если вы указали его, то он должен превышать длину самой большой строки в файле. Значением по умолчанию параметра `разделитель` является запятая.

Если в поле могут находиться запятые, то при создании CSV-файла разделять поля запятой нежелательно. Обычно в качестве ограничителя выбирают символ "|".

Параметр `ограничитель` — символ, который влияет на объединение строк CSV-файла и разграничение полей в строке. Значение по умолчанию — двойная кавычка. Так, например, если в CSV-файле имеется единственная

двойная кавычка, расположенная в его начале, то функция `fgetcsv()` вернет массив с единственным элементом, содержащим весь файл.

В листинге 6.35 приведен сценарий, читающий содержимое файла test.csv, данные в котором разделены вертикальной чертой, и выводящий их в окно браузера (рис. 6.6). Попутно выводится номер строки и количество найденных в ней полей.

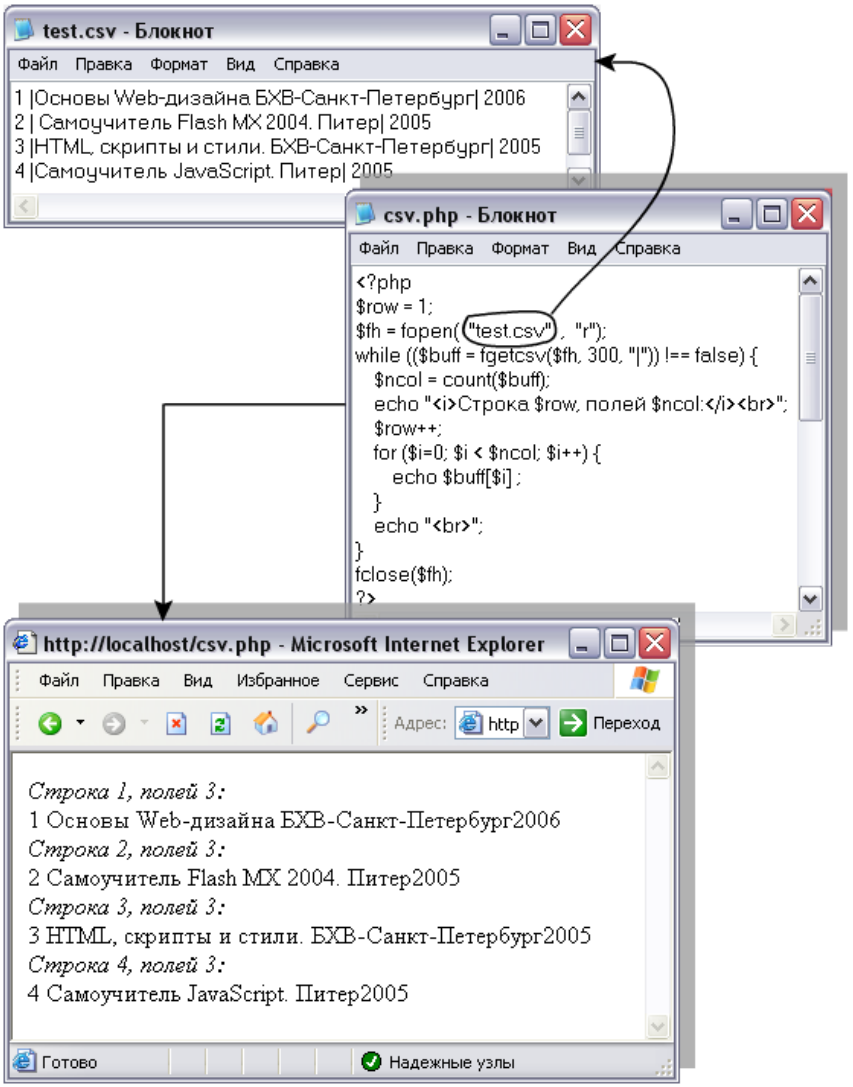

Рис. 6.6. Содержимое CSV-файла, сценарий и результат его работы в браузере

Листинг 6.35. Чтение CSV-файла и вывод его содержимого

```
<?php
$row = 1;
$fh = fopen("test.csv", "r");      // открытие файла
while (($buff = fgetcsv($fh, 300, "|")) !== false) {
 $ncol = count($buff);      // количество полей
 echo "<i>Строка $row, полей $ncol:</i><br>";
 $row++;
 for ($i=0; $i < $ncol; $i++) {      // вывод полей строки
  echo $buff[$i] ;
 }
 echo "<br>";
}
fclose($fh);      // закрытие файла
?>
```

Функции для работы с табличными данными

Данные с табличной организацией естественно сохранять в оперативной памяти с помощью двумерного массива. Такой массив содержит в качестве элементов другие массивы. Иначе говоря, таблица представляется массивом ее строк, а строки — массивами значений полей таблицы. Вот пример создания таблицы из четырех строк и трех столбцов.

Пример

```
$rows[]=array("a1","b1","c1");
$rows[]=array("a2","b2","c2");
$rows[]=array("a3","b3","c3");
$rows[]=array("a4","b4","c4");
```

Так например, в таблице `$rows` в первой строке нулевой столбец (поле) имеет значение `"a2"`, т. е. значение `$rows[1][0]` равно `"a2"`. Напомню, что по умолчанию нумерация элементов массива начинается с нуля.

В качестве имен столбцов допустимы символьные индексы.

Пример

```
$row[]=array("f1"=>"a1","f2"=>"b1","f3"=>"c1");
$row[]=array("f1"=>"a2","f2"=>"b2","f3"=>"c2");
$row[]=array("f1"=>"a3","f2"=>"b3","f3"=>"c3");
$row[]=array("f1"=>"a4","f2"=>"b4","f3"=>"c4");
```

Здесь во второй строке поле `f2` имеет значение `"b3"`, т. е. значение `$rows[2]['f2']` равно `"b3"`.

Далее мы рассмотрим несколько задач, встречающихся при работе с табличными данными, и ряд функций, полезных при создании конкретных приложений. Коды определений этих функций вы можете собрать вместе в одном файле, например, tablib.inc. Впоследствии этот файл можно включать в сценарии приложений с помощью функции `include()`.

❒ Запись двумерного массива в CSV-файл.

Двумерный массив будем записывать в текстовый CSV-файл построчно, разделяя поля специальным символом. Иначе говоря, требуется записать двумерный массив в CSV-файл с указанным разделителем. Определение пользовательской функции, решающей эту задачу, приведено в листинге 6.36.

Листинг 6.36. Функция записи массива в файл (`atofile()`)

```
function atofile($arr,$filename, $separator="|"){
/* Параметры:
* $arr — двумерный массив
* $filename — имя файла
* $separator — символ-разделитель (по умолчанию "|")
* Возвращает число записанных байтов или false — в случае ошибки */
foreach($arr as $ind=>$r){
     $i=0; // начало строки
     foreach($r as $fld=>$value){
          if ($i>0){     // если не начало строки
               $str.=$separator.$value;

          } else {
               $str.=$value;
               $i=1;
          }
     }
     $str.="\r\n";     // добавляем символ конца строки
     $i=0;
}
$fh=fopen($filename,"w");
$ret=fwrite($fh,$str);
fclose($fh);
return $ret; // возвращаем то же, что и fwrite()
}
```

❒ Чтение двумерного массива из CSV-файла.

В листинге 6.36 приводился код сценария чтения CSV-файла и вывода значений его полей в окно браузера. Немного модифицировав этот сценарий, можно написать функцию чтения CSV-файла в двумерный массив (листинг 6.37).

Листинг 6.37/ Функция чтения CSV-файла в двумерный массив (`filetoa()`)

```
function filetoa($filename,$separator="|"){
/* Параметры:
* $filename — имя файла
* $separator — символ-разделитель (по умолчанию "|")
* Возвращает: двумерный массив или null — при неудаче
*/
$fh = fopen($filename, "a+");
while (($buff = fgetcsv($fh, 1024, $separator)) !== false) {
       $ret[]=$buff;     // добавляем массив в массив строк
}
return $ret;
}
```

Здесь файл открывается в режиме чтения и добавления, чтобы при его отсутствии была предпринята попытка создания нового файла.

❒ Поиск строки в массиве.

В таблице, представленной в виде двумерного массива, может потребоваться найти строку, которая удовлетворяет некоторому условию. В листинге 6.38 приведена функция, которая ищет элемент одномерного массива, индекс которого (числовой или символьный) удовлетворяет некоторому условию вида:

`поле отношение значение.`

Здесь `поле` — индекс (числовой или символьный) элемента в массиве-строке; `отношение` — строка, содержащая оператор сравнения, например, "==", ">" и т. п.; `значение` — образец значения.

Функция `find()`, приведенная в листинге 6.38, возвращает номер первой найденной строки, удовлетворяющей условию, или –1 — при неудачном поиске.

Листинг 6.38. Функция поиска строки в массиве

```
function find($arr,$fld,$val,$rel="=="){
/* Параметры:
* $arr — двумерный массив, в котором осуществляется поиск
```

```
* $fld — поле, т. е. индекс элемента в строке
* $val — значение
* $rel — отношение (по умолчанию — "==")
* Возвращает: номер (индекс) строки, удовлетворяющей условию
* $fld $rel $val, или -1 — при неудаче
*/
$criterion='return $r[$fld]'. "$rel \"$val\";"; /* строка с условием
                                          поиска */
foreach($arr as $ind=>$r){
     if (eval($criterion)===true){
          return $ind;           // возвращаем индекс
     }
}
return -1;
}
```

Обратите внимание, что здесь сначала формируется строка `$criterion`, содержащая условие поиска, которая затем вычисляется с помощью функции `eval()`.

Листинг 6.39 содержит тестовый сценарий с применением функции `find()`.

Листинг 6.39. Тестовый сценарий с применением функции поиска

```
<?php
$rows[]=array("a1","b1","c1");
$rows[]=array("a2","b2","c2");
$rows[]=array("a1","b3","c3");
$rows[]=array("a4","b4","c4");
function find($arr,$fld,$val,$rel="=="){
// см. листинг 6.35
}
echo find($rows,"1","b2","==");     // 1
echo "<br>";
echo find($rows,"0","a1","==");     // 0
echo "<br>";
echo find($rows,"2","a4","==");     // -1
?>
```

❒ Выборка строк в массиве.

Немного модифицировав код листинга 6.39, можно получить функцию, возвращающую номера (индексы) всех строк в двумерном массиве, которые удовлетворяют заданному условию. Если таких строк не оказалось, то возвращается `null`. Код функции `filter()`, которая производит выборку или фильтрацию строк в двумерном массиве, приведен в листинге 6.40.

Листинг 6.40. Функция выборки строк в массиве

```
function filter($arr,$fld,$val,$rel="=="){
/* Параметры:
* $arr — двумерный массив, в котором осуществляется поиск
* $fld — поле, т. е. индекс элемента в строке
* $val — значение
* $rel — отношение (по умолчанию — "==")
* Возвращает: массив номеров (индексов) всех строк,
* удовлетворяющих условию $fld.$rel.$val,
* или null — при неудаче
*/
$criterion= $criterion='return $r[$fld]'. "$rel \"$val\";"; /* строка
                                    с условием выборки */
foreach($arr as $ind=>$r){
     if (eval($criterion)===true){
          $ret[]=$ind;     // добавляем индекс в массив
     }
}
return $ret;     // возвращаем массив индексов или null
}
```

Листинг 6.41 содержит сценарий с применением функции `filter()`.

Листинг 6.41. Тестовый сценарий с применением функции выборки

```
<?php
$rows[]=array("a1","b1","c1");
$rows[]=array("a2","b2","c2");
$rows[]=array("a1","b3","c3");
$rows[]=array("a4","b4","c4");
function filter($arr,$fld,$val,$rel="=="){
// см. листинг 6.40
```

```
}
$x=find($rows,"0","a1","==");
print_r($x);            // выводит массив {0, 2}
?>
```

Сложный счетчик посещений страницы

В данном разделе мы создадим счетчик, более сложный, чем рассмотренный в *разд. 6.4.6*. В этом приложении используются приемы, пригодные и для других проектов, таких как баннер и гостевая книга.

Простота счетчика, рассмотренного в *разд. 6.4.6*, заключалась в том, что файл, хранивший состояние счетчика, содержал лишь одно число — количество посещений страницы, а сценарий считывал это число, увеличивал его на единицу и сохранял новое значение в том же файле. Однако можно создать более сложный счетчик, который будет хранить не только количество посещений, но и сведения о пользователе (например, IP-адрес компьютера, название браузера, URL-адрес страницы и т. п.).

Очевидно, что сложный счетчик можно создать на основе текстового CSV-файла с табличными данными. В листинге 6.42 приведен код, реализующий счетчик, который сохраняет свое состояние в таблице с двумя столбцами. Первый столбец содержит IP-адрес компьютера, который запросил страницу со сценарием счетчика, а второй — количество посещений страницы данным пользователем. Обратите внимание, что CSV-файл предварительно создавать не нужно: если он отсутствует, то будет создан автоматически функцией `filetoa()`.

Основная задача сценария состоит в том, чтобы добавить в таблицу новую запись (строку) или изменить существующую. Что именно следует сделать, зависит от результата поиска в таблице IP-адреса посетителя. Если пользователь впервые посетил страницу, то добавляется новая запись, в противном случае соответствующая ему запись изменяется. Переменная `$counter` содержит общее количество посещений страницы всеми пользователями. Определения пользовательских функций `atofile()`, `filetoa()` и `find()` (листинги 6.36, 6.37 и 6.38), необходимых для работы сценария листинга 6.42, сосредоточим в файле tablib.inc.

Листинг 6.42. Сложный счетчик посещений страницы

```
<?php
include('tablib.inc');       /* определения функций
                             filetoa(), find() и atofile() */
```

```
$filename="mycounter.txt";          // файл с таблицей состояния счетчика
$rows=filetoa($filename,"|");     // читаем файл с таблицей в массив
$ip=$_SERVER['REMOTE_ADDR'];          // IP посетителя
if ($rows!=NULL){               // если что-то есть
     $i=find($rows,"0",$ip,"==");     // поиск строки с данным IP
     if ($i>=0) {     /* если уже есть такой IP,
                      то увеличиваем количество посещений на 1 */
          $rows[$i][1]=$rows[$i][1]+1;
     } else {     /* иначе добавляем строку с данным IP и количеством
                      посещений 1 */
          $rows[]=array($ip,1);
     }

} else{          // в файле счетчика ничего нет
$rows[]=array($ip, 1);      /* добавляем строку с данным IP и
                            количеством посещений 1 */
}
atofile($rows,$filename, "|");     // записываем массив в файл
foreach ($rows as $r){     // вычисляем общее количество посещений
     $counter=$counter+$r[1];
}
?>
```

На страницу обычно выводят общее количество ее посещений. Если сохранить приведенный код в отдельном файле, например, counter.inc, то для отображения суммарного значения счетчика можно воспользоваться сценарием листинга 6.34.

Можно также создать отдельную страницу, которая будет отображать детальное состояние счетчика, например, в виде таблицы. В листинге 6.43 приведен код сценария, который отображает таблицу состояния счетчика.

Листинг 6.43. Страница с детальным состоянием счетчика посещений

```
<?php
include('tablib.inc');     // здесь понадобится только функция filetoa()
$filename="mycounter.txt";
$rows=filetoa($filename,"|");
echo "<h2>Счетчик посещений</h2>
```

```
<table border=1><tr><th>№п/п</th><th>IP-адрес</th>
<th>Количество</th></tr>";
foreach ($rows as $r){
     $i++;
     echo "<tr><td>$i</td><td>$r[0]</td><td>$r[1]</td></tr>";
     $counter=$counter+$r[1];     // общее количество посещений
}
echo "</table><br>Общее количество: <b>$counter</b>";
?>
```

Таблица, с которой работает счетчик, может содержать и больше столбцов для хранения дополнительных параметров, которые являются элементами суперглобального массива `$_SERVER` *(см. справочное руководство по PHP)*.

Итак, мы рассмотрели создание счетчика посещений страниц собственного сайта. При этом страница со счетчиком должна содержать серверный сценарий, например, вставку PHP-кода вида:

```
<?php include("counter.inc"); ?>
```

Разработчик сайта, не имеющий доступа к PHP, не сможет воспользоваться описанной технологией.

Теперь рассмотрим создание счетчика, который можно предоставить всем желающим, в том числе и не использующим PHP. Иначе говоря, займемся разработкой свободно распространяемого счетчика.

Счетчик, о котором пойдет речь, вставляется в HTML-код страницы с помощью тега

```
<img src=URL_адрес_счетчика>
```

Здесь `URL_адрес_счетчика` — URL-адрес файла со сценарием, реализующим функции счетчика и создающим графическое изображение со значением счетчика. Это изображение отображается на странице пользователя с помощью тега `<img>` в том месте, где он позиционирован. В листинге 6.44 приведен пример HTML-кода страницы с таким счетчиком, а на рис. 6.7 показан ее вид в окне браузера.

В теге `<img>` в атрибуте `src` указан Web-сервер (`localhost`), установленный на вашем компьютере, и имя файла (counter.php) со сценарием, реализующим счетчик. Однако html-файл из листинга 6.44 может находиться в любом месте на вашем компьютере. Это справедливо для экспериментов на локальном компьютере. В общем же случае HTML-страница со счетчиком может располагаться на одном Web-сервере, а сценарий счетчика (counter.php) — на другом.

Листинг 6.44. HTML-код с распространяемым счетчиком посещений страницы

```
<html>
<head><title>Моя страница</title></head>
<body bgcolor=#f0f0f0>
<center><h1>Моя страница</h1></center>
<! Вставка счетчика >
<img src='http://localhost/counter.php'>
</body>
</html>
```

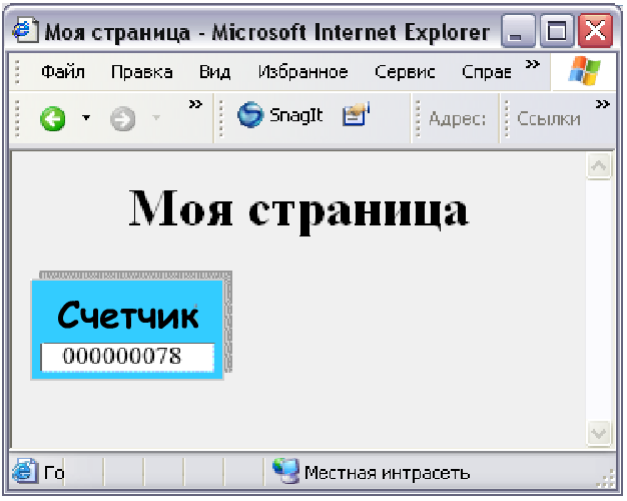

Рис. 6.7. Страница со счетчиком посещений в виде графического изображения

Листинг 6.45 содержит сценарий, реализующий функции счетчика и создающий его графическое изображение.

Листинг 6.45. Сценарий распространяемого графического счетчика

```
<?php
$filename="mycounter.txt";      // файл с таблицей состояния счетчика
$rows=filetoa($filename,"|"); // читаем из файла в массив
/* Адрес страницы, обратившейся к счетчику: */
$page= $_SERVER['HTTP_REFERER'];
if ($rows!=NULL){      // если что-то есть
     $i=find($rows,"0",$page,"==");
```

```
     if ($i>=0) {
          $rows[$i][1]=$rows[$i][1]+1;
          $counter=$rows[$i][1];
     } else {
          $rows[]=array($page,1);
          $counter=1;
     }
} else{          // если массив пуст
          $rows[]=array($page, 1);
          $counter=1;
}
// Значение счетчика находится в переменной $counter
atofile($rows,$filename, "|");     // запись массива в файл
/*************************************************************/
$counter=sprintf("%'09s",$counter); /* форматируем значение счетчика как
                                    строку с добавлением ведущих нулей */
/**** Создание изображения счетчика ***************************/
header("Content-type: image/gif");     // HTTP-заголовок о типе данных
$img = imagecreatefromgif("counter.gif");     /* создаем ресурс
                         изображения из графического файла counter.gif*/
/**** Параметры для внедряемого текста — значения счетчика: ****/
$color = imagecolorallocate($img, 255, 0, 0);      // красный цвет
$font="times.ttf"; $fontsize=12;               // шрифт
$angle=0; $x=20; $y=57;                 // угол наклона и координаты
/* Внедряем в изображение отформатированное значение счетчика: */
imagettftext($img, $fontsize,$angle,$x,$y,$color,$font,$counter);
imagegif($img);          // выводим изображение
/**** Определения функций **************************/
function find($arr,$fld,$val,$rel="=="){
// см. листинг 6.38
}
function atofile($arr,$filename, $separator="|"){
// см. листинг 6.36
}
function filetoa($filename,$separator="|"){
// см. листинг 6.37
}
?>
```

В данном примере идентифицируется адрес страницы, с которой произошло обращение к сценарию счетчика (`$_SERVER['HTTP_REFERER']`).

> ***Внимание!***
>
> Для формирования изображения счетчика применяются специальные функции PHP из библиотеки работы с графикой. Эта библиотека, являющаяся расширением PHP (для Windows — файл php_gd2.dll), не устанавливается по умолчанию, но входит в дистрибутив модуля PHP. Примеры работы с графикой в PHP рассмотрены в *разд. 6.3*.

В данном сценарии предполагается, что на сервере находится небольшой графический файл, представляющий собой подложку для значения счетчика (в примере — counter.gif). Его можно создать с помощью графического редактора. Из этого файла с помощью функции `imagecreatefromgif()` создается ресурс соответствующего изображения, над которым можно выполнять различные манипуляции, в том числе внедрить другое изображение или текст. Функция `imagettftext()` внедряет в изображение подложки счетчика его значение, отформатированное функцией `sprintf()`, а функция `imagegif()` возвращает браузеру результирующее изображение, отображаемое им. Если изображение подложки счетчика достаточно простое (как, например, на рис. 6.7), то его можно сгенерировать непосредственно с помощью сценария (см. *разд. 6.3*), а не загружать из готового графического файла.

> ***Примечание***
>
> В листинге 6.45 определения функций `find()`, `atofile()` и `filetoa()` вставлены непосредственно, а не путем включения отдельного файла. Попытка включить файл с помощью `include()` приводит к тому, что графическое изображение счетчика не отображается в браузере. Это связано с особенностями применения функции `header()`.

Создание баннера

Баннер (banner — знамя, символ, газетный заголовок) представляет собой небольшое графическое изображение, размещаемое на Web-странице для рекламы сайтов, товаров и услуг. Картинка может быть статической (файлы формата GIF, JPEG, PNG) или анимированной (файлы формата GIF89a или SWF (Flash-ролики)).

Щелчок на баннере может приводить к переходу на определенную Web-страницу (страницу спонсора, хозяина баннера). Кроме того, показ страницы с баннером и/или щелчок на баннере может запускать серверный сценарий, который ведет учет этих действий с целью последующего анализа эффективности баннерной рекламы, назначения бонусов владельцам страниц, на которых размещен баннер, и т. п.

Таким образом, баннер можно создать как графическое изображение, заключенное в контейнерный тег (например, `<a>` или `<form>`), в котором указать адрес перехода. В первом случае баннер оказывается графической ссылкой, а во втором — выполняет роль кнопки типа submit **Отправить**. Однако баннер — это не просто графическая ссылка или форма с графической кнопкой. Дело в том, что в его HTML-коде нежелательно указывать адрес графического файла, поскольку в противном случае владелец сайта мог бы задать его по своему усмотрению. В результате спонсор (хозяин баннера) начислял бы бонусы за показ не своей рекламы, а информации владельца сайта. Далее мы рассмотрим, как этого избежать.

Со структурно-функциональной точки зрения баннер похож на сложный счетчик, рассмотренный в предыдущем разделе. Действительно, баннер, как и счетчик, получает некоторые данные от посетителя страницы и сохраняет их в таблице, расположенной на сервере. В следующем разделе мы увидим, что подобным образом может быть устроена и гостевая книга.

Рассмотрим создание баннера в виде графической ссылки:

```
<a href=banner.php?click=yes><img src= banner.php></a>
```

Здесь адреса ссылки (значение атрибута `href`) и источника графического изображения (значение атрибута `src`) совпадают. Это в данном случае файл banner.php (вообще говоря, можно использовать URL-адрес), который содержит сценарий учета показов страницы с баннером и количества щелчков на нем, а также выводит в браузер графическое изображение. Такая конструкция позволяет отдельно учитывать щелчки на баннере и просто показы страницы с баннером.

Обратите внимание, что в качестве адреса изображения указан не графический файл, а файл со сценарием. Если недобросовестный владелец страницы изменит значение атрибута `src`, то показы баннера не будут учитываться. Сценарий будет обрабатывать только щелчки на изображении. И это правильно, поскольку скрипт banner.php не только учтет щелчок, но и выполнит переход на страницу хозяина баннера (спонсора). Если владелец страницы изменит только значение атрибута `href`, то щелчки на баннере не будут учитываться, а показы обработаются.

При щелчке на баннере сценарию banner.php будет передано значение `yes` переменной `click`. Эта информация понадобится, чтобы узнать, был ли щелчок на баннере.

В листинге 6.46 приведен пример простого HTML-кода страницы, содержащей баннер, а в листинге 6.47 — сценарий работы с ним (файл banner.php).

Листинг 6.46. Страница с баннером

```
<html>
<h1>Моя страница</h1>
<a href="http://localhost/banner.php?click=yes">
<img src="http://localhost/banner.php"></a>
</html>
```

Листинг 6.47. Сценарий работы с баннером (banner.php)

```
<?php
if ($_REQUEST['click']=="yes"){     // был щелчок на баннере
     header("Location:anypage.htm"); // переход на стр. хозяина баннера
} else {
     header("Content-type:image/gif"); // посылаем заголовок
     $img=imagecreatefromgif("banner.gif"); /* создаем изображение как
                                        ресурс из графического файла*/
     imagegif($img);     // выводим изображение
}
include("bcounter.php");     // счетчик
?>
```

> ***Внимание!***
>
> Данный сценарий будет работать, если установлена библиотека функций PHP для работы с графикой. Для Windows это — php_gd2.dll. Примеры работы с графикой в PHP рассмотрены в *разд. 6.3*.

В сценарии листинга 6.47 включается файл bcounter.php (разумеется, имя может быть произвольным), в котором обрабатываются щелчки на баннере и его показы. Для проверки работоспособности приведенного кода на первых порах в качестве счетчика подойдет код листинга 6.42. Напомню, что этот сценарий записывает в таблицу IP-адрес компьютера пользователя и количество его посещений страницы. Однако для нашего баннера такая информация не вполне подходит. Во-первых, важен не IP-адрес посетителя, а адрес страницы с баннером. Во-вторых, наш баннер потенциально может фиксировать как его показы, так и щелчки. Поэтому таблица, в которой сохраняется информация о работе баннера, должна содержать, по крайней мере, три столбца:

- адрес Web-страницы с баннером;
- количество показов (т. е. посещений или загрузок страницы);
- количество щелчков на баннере.

Пример сценария, реализующего счетчик для баннера (включаемый файл bcounter.php), приведен в листинге 6.48.

Листинг 6.48. Счетчик показов и щелчков на баннере (`bcounter.php`)

```
<?php
$isclick=($_REQUEST['click']=="yes"); // был ли щелчок на баннере
include('tablib.inc');      /* определения функций
                            filetoa(), find() и atofile() */
$filename="bcounter.txt";          // файл с таблицей состояния счетчика
$ref= $_SERVER['HTTP_REFERER'];     // адрес страницы
$rows=filetoa($filename,"|");     // читаем файл с таблицей в массив
if ($rows!=NULL){               // если что-то есть
     $i=find($rows,"0",$ref,"==");     /* поиск строки с данным
                                        адресом страницы */
     if ($i>=0) {     // если уже есть такая страница
          if ($isclick) { // был щелчок
               $rows[$i][2]=$rows[$i][2]+1;
          } else {     // не было щелчка
               $rows[$i][1]=$rows[$i][1]+1;
          }
     } else {     // иначе добавляем строку
          if ($isclick) {
               $rows[]=array($ref,1,1);
          } else {
               $rows[]=array($ref,1,0);
              }
     }

} else{          // в файле счетчика ничего нет
     if ($isclick) {
              $rows[]=array($ref, 1, 1);
       } else {
              $rows[]=array($ref, 1, 0);
       }
}
atofile($rows,$filename, "|");     // записываем массив в файл
?>
```

Таким образом, чтобы проверить работу рассмотренного баннера, требуется создать:

- страницу с баннером (листинг 6.46);
- сценарий работы с баннером (banner.php, листинг 6.47);
- сценарий счетчика показов и щелчков на баннере (bcounter.php, листинг 6.49). Файл bcounter.txt, содержащий информацию о состоянии счетчика, создается автоматически;
- файл tablib.inc с определениями функций `atofile()`, `filetoa()` и `find()` (листинги 6.36—6.38).

Загрузите в браузер страницу с баннером, перезагрузите (обновите) ее несколько раз, щелкните на баннере и посмотрите в текстовом редакторе содержимое файла bcounter.txt.

Создание гостевой книги

Гостевая книга предназначена для хранения на сервере сообщений от посетителей вашего сайта. Работа с гостевой книгой предполагает добавление записей посетителями и их коррекцию владельцем книги, который также может отвечать на сообщения посетителей. Так называемые форумы устроены аналогично, хотя и имеют свои особенности.

На практике гостевые книги и форумы создают на основе баз данных, предполагающих наличие на сервере СУБД. Тем не менее, хранить относительно небольшие объемы данных можно и в текстовом CSV-файле, а сценарий работы с ним аналогичен скриптам счетчика посещений и баннера.

В проекте гостевой книги есть ряд особенностей. Запись (строка) в таблице гостевой книги обычно содержит следующие поля:

- имя посетителя;
- город (место);
- URL-адрес сайта посетителя;
- адрес электронной почты посетителя;
- текст сообщения посетителя;
- ответ хозяина гостевой книги;
- дата и время отправки сообщения посетителем.

Возможны различные алгоритмы работы с гостевой книгой. Вот один из простейших.

1. При обращении к гостевой книге в браузер загружается страница, содержащая уже имеющиеся в книге записи, а также кнопку **Добавить запись**.

2. Щелчок на этой кнопке загружает в браузер другую страницу с формой, содержащей поля ввода данных и кнопку **Добавить запись**.
3. Щелчок на этой кнопке приводит к выполнению сценария, который анализирует принятые данные, введенные в форму. Если результат анализа положителен (по некоторому критерию), то данные записываются в файл хранения гостевой книги, в противном случае — нет.
4. Завершение работы сценария на предыдущем этапе приводит к переходу на начальную страницу, отображающую все записи гостевой книги. Если новая запись была добавлена, то теперь она отображается.

Для реализации данного алгоритма потребуются три файла (имена, разумеется, могут быть произвольными):

- gbook.php — вывод всех записей гостевой книги из файла mygbook.txt;
- gbadd.php — вывод формы ввода данных посетителя, а также их анализ и запись в файл mygbook.txt;
- mygbook.txt — файл хранения данных гостевой книги (его не нужно создавать предварительно).

В листинге 6.49 приведен сценарий, выводящий существующие записи гостевой книги и форму, содержащую единственную кнопку типа submit **Добавить запись**, щелчок на которой приведет к переходу на страницу gbadd.php. Обратите внимание, что записи гостевой книги выводятся в порядке, противоположном тому, в котором они расположены в файле хранения. Это обеспечивает функция сортировки `ksort()`. Таким образом, более поздние записи оказываются наверху списка. Пример выполнения данного сценария показан на рис. 6.8.

Листинг 6.49. Вывод содержимого гостевой книги (gbook.php)

```
<?php
include('tablib.inc'); // содержит определение filetoa() (листинг 6.37)
$filename="mygbook.txt";          // файл с данными гостевой книги
$rows=filetoa($filename,"|");     // читаем файл в массив
/* Форма с кнопкой для перехода к странице добавления записи */
echo "<center><h2>Моя гостевая книга</h2>";
echo "<form action=gbadd.php method=POST>
<input type=submit name=addrecord value='Добавить запись'>
</form><br>";
/* Вывод содержимого гостевой книги */
if ($rows!=NULL){     // если что-то есть
       krsort($rows); // сортируем массив по убыванию индексов
```

```
     $i=0;
          foreach ($rows as $ind=>$r){
          $i++;
          echo "<b>$i. {$r[0]}</b> {$r[1]} {$r[6]}<br>
          <b>Сайт:</b> <a href={$r[2]}>{$r[2]}</a><b> E-mail:</b>
               <a href={$r[3]}>{$r[3]}</a><br>
          <br><textarea rows=4 cols=60>{$r[4]}</textarea><br>";
          if ($r[5]){
               echo "<b>Ответ:</b><br>
                    <textarea rows=4 cols=60>{$r[5]}</textarea>";
          }
          echo "<hr>";     // горизонтальная линия
       }
}
if ($i>0){
     echo "Всего записей: $i";
}else{
     echo "Нет записей";
}
?>
```

Щелчок на кнопке **Добавить запись** (`name='addrecord'`) на странице, показанной на рис. 6.8, приведет к переходу на страницу gbadd.php, код которой содержит листинг 6.50. Эта страница выводит форму с полями и кнопкой типа submit (`name='adddok'`) **Добавить запись** (рис. 6.9), щелчок на которой передает данные формы той же странице — gbadd.php. Интерпретатор PHP начинает работать с ней заново. Теперь значение `$_POST['addrecord']` не определено, поскольку среди элементов формы, из которой передавались данные, нет элемента с именем `'addrecord'`. Зато определено и не пусто значение `$_POST['addok']`. Следовательно, форма ввода данных не отображается. Происходит переход к странице gbook.php, содержащей все записи гостевой книги. При этом сначала проверяется, не пусто ли поле ввода сообщения. Если оно пусто, то выполнение сценария завершается, и данные формы не записываются в файл хранения гостевой книги. В противном случае данные полей формы, а также "пустой" ответ хозяина книги и текущая дата-время добавляются в массив, который затем записывается файл mygbook.txt. Обратите внимание, что сообщение вводится пользователем в HTML-элемент `<textarea>` и оно может содержать невидимые символы перевода строки. При записи сообщения в файл мы их заменяем символом пробела с помощью функции `str_replace()`. Если этого не делать, нарушится структура данных в файле хранения гостевой книги.

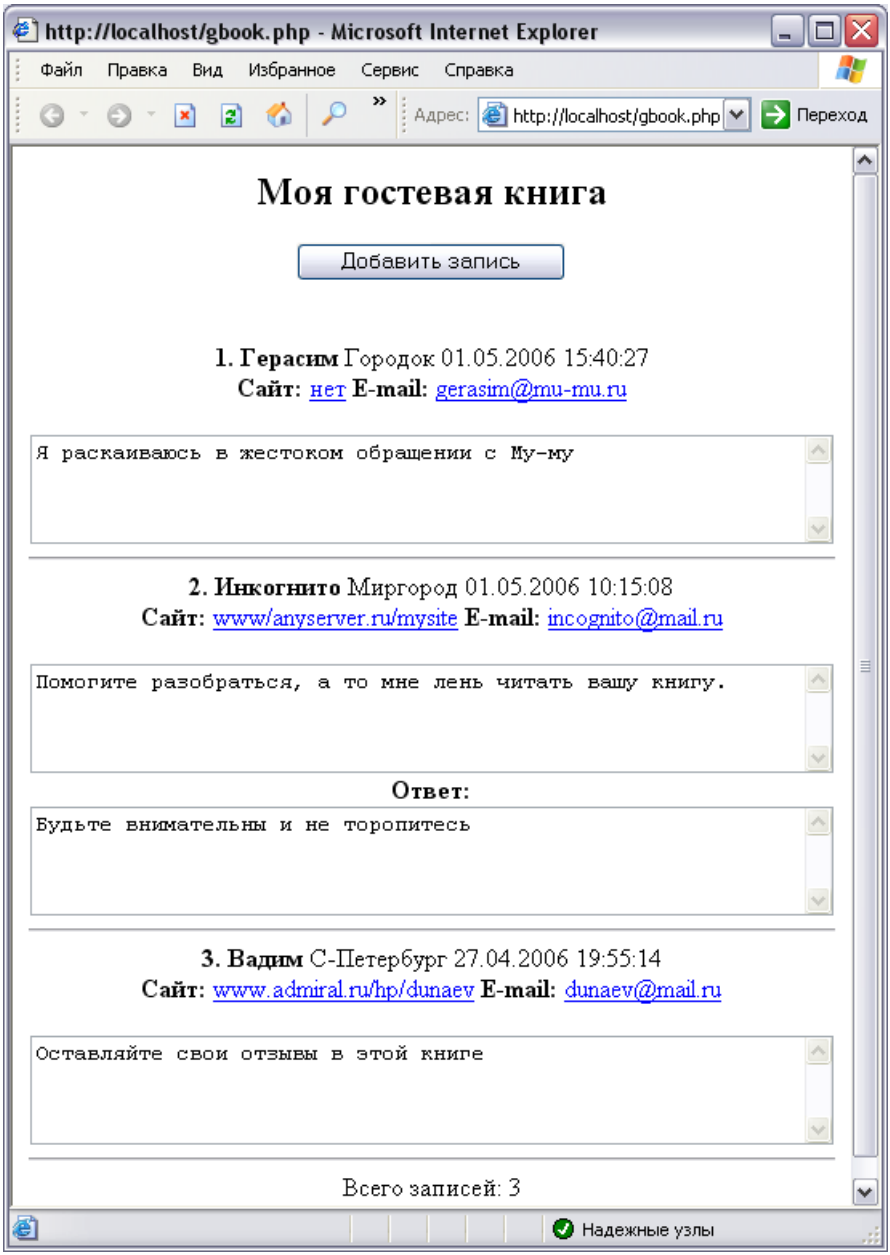

Рис. 6.8. Страница с записями гостевой книги, отображаемая сценарием листинга 6.49

Листинг 6.50. Отображение формы ввода и добавление данных в гостевую книгу (gbadd.php)

```
<?php
if ($_POST['addrecord']){     // если требуется добавить запись
     /* Форма для ввода данных */
     echo "<html>
      <center><h2>Моя гостевая книга</h2></center>
      <form action=gbadd.php method=POST>
     <table>
     <tr><td>Ваше имя:</td>
     <td><input type=text name=username size=20></td></tr>
     <tr><td>Город:</td>
     <td><input type=text name=city size=20></td></tr>
     <tr><td>URL Вашего сайта:</td>
     <td> <input type=text name=site size=40></td></tr>
     <tr><td>E-mail:</td>
     <td><input type=text name=email size=40></td></tr>
     </table>
     Сообщение:<br>
     <textarea name=msg rows=4 cols=60></textarea><br>
     <input type=submit name=addok value='Добавить запись'>
     <input type=reset name=reset value='Отмена'>
     </form><br>
     </html>";
} elseif ($_POST['addok']) {      // если подтвердили ввод в форму
     header("Location:gbook.php"); /* Переход к странице
                                      показа записей гостевой книги */
     if (!$_POST['msg']){exit;};     // если сообщение пусто, то выход
       include('tablib.inc');     /* содержит определения функций
            filetoa() (листинг 6.38) и atofile() (листинг 6.36) */
     $filename="mygbook.txt";          // файл хранения гостевой книги
     $rows=filetoa($filename,"|");     // читаем файл в массив
     /* Добавляем запись в массив */
     $rows[]=array(trim($_POST['username']),
                trim($_POST['city']),
                trim($_POST['site']),
                trim($_POST['email']),
                str_replace("\r\n"," ",trim($_POST['msg'])),
```

```
                '',
                date("d.m.Y H:i:s"));
    atofile($rows,$filename,"|");      // запись массива в файл
}
?>
```

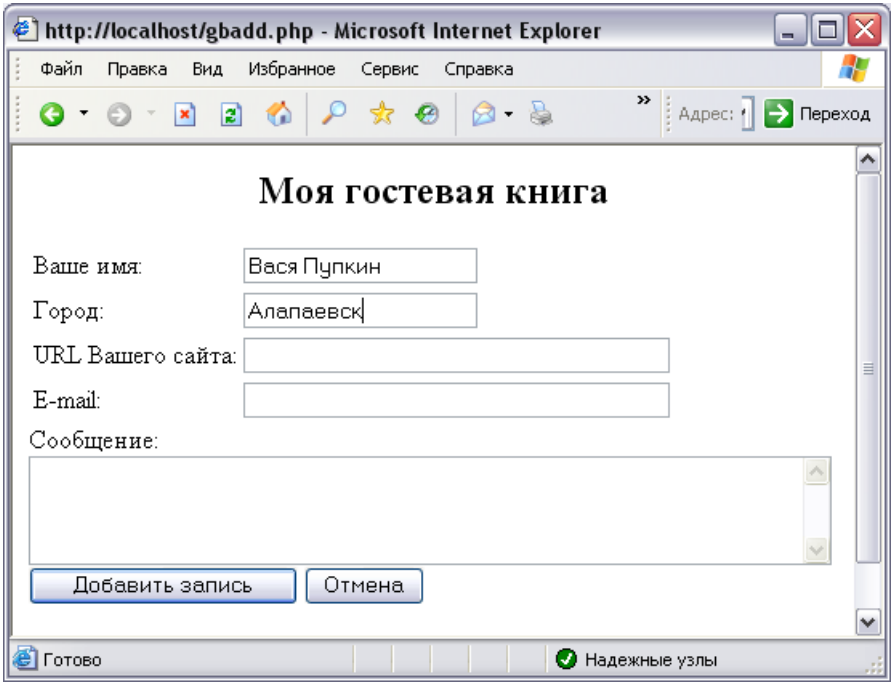

Рис. 6.9. Форма ввода данных посетителем гостевой книги

Итак, посетитель может просматривать и добавлять записи в гостевую книгу. Однако на этом проект гостевой книги не завершается. Необходимо еще создать защищенную паролем страницу для редактирования и ввода ответов, которой может пользоваться хозяин гостевой книги. Подробно эту задачу мы здесь не будем рассматривать. Ограничимся лишь общими замечаниями.

Страница, на которой пользователь может корректировать записи и вводить ответы, устроена аналогично странице показа всех записей. Ее основная особенность в том, что кроме полей, выводимых для обычных посетителей, для каждой записи следует предусмотреть поле для ввода ответов, а также кнопки типа submit **Принять изменения** и **Удалить**. В другом, лучшем, варианте вместо этих кнопок для каждой записи можно вывести пару флажков с такими

же надписями. Тогда понадобится только одна кнопка типа submit **Принять изменения**. При щелчке на ней следует перейти к сценарию, который перезаписывает (а не добавляет) или удаляет отмеченные записи гостевой книги.

Гостевая книга может содержать сотни и тысячи записей, и тогда выводить их одновременно оказывается неэкономичным. Лучше показывать по 3—5 записей. Для этого на странице следует разместить панель навигации, например, с кнопками **Вперед**, **Назад** и т. п. Анализ данных, введенных посетителем (в частности, проверку наличия сообщения), лучше выполнять не на стороне сервера, как в рассмотренном примере, а на стороне клиента посредством сценария, написанного на JavaScript. Этот сценарий должен вызываться по событию `onsumit`, но его код может быть сформирован на стороне сервера (`echo "<SCRIPT> … </SCRIPT>"`).

Так называемый форум устроен аналогично гостевой книге. Главная его особенность состоит в том, что посетители могут выбирать тему обсуждения из уже существующих, а также создавать новые темы. Для каждой темы можно создать на сервере отдельный текстовый CSV-файл, содержащий сведения о посетителях. Кроме того, там должен быть еще один файл, в котором хранятся соответствия между названиями тем и именами файлов с информацией по данным темам. Назовем этот файл каталогом.

Главная страница форума содержит, кроме прочего, список всех тем, имеющихся в каталоге. При выборе темы из этого списка серверный сценарий определяет по каталогу соответствующий ей файл и формирует страницу с данными, которая отображается браузером. Посетитель форума может добавить запись, как и в случае гостевой книги.

Как на главной странице форума, так и на странице, отображающей информацию по выбранной теме, посетитель может щелкнуть на кнопке **Создать тему**. В результате должна появиться форма для ввода названия темы. Эти сведения отправляются серверному сценарию, который проверяет, имеется ли в каталоге полученное название темы. Если нет, то название темы и соответствующее ей сгенерированное каким-либо образом имя файла добавляются в каталог. Затем создается новый пустой файл с этим именем и формируется страница без данных, но с кнопкой **Добавить запись**. Что делать далее, вам уже известно. Если полученное от посетителя название темы не является новым, то в браузере отображается страница с данными по уже существующей теме.

Таков, в общих чертах, алгоритм работы форума. Разумеется, его можно развить в различных направлениях. Я советую попробовать реализовать этот проект, даже если вам не нужен собственный форум. Работая над ним, вы многому научитесь в плане программирования.

6.5. Работа с базами данных

Общим для таких приложений, как счетчик посещений Web-страниц, баннер, гостевая книга, форум, прайс-лист, учет продаж и т. п., является работа с хранилищем данных. При относительно небольшом объеме информации, простых алгоритмах ее обработки и малой интенсивности многопользовательского режима, данные можно хранить в обычных текстовых файлах. Однако если в вашем проекте несколько таблиц, содержащих тысячи и более записей, интенсивный многопользовательский режим, защита данных, выбор нужных сведений по различным критериям, а также более или менее сложный их анализ, то следует подумать о применении настоящих баз данных.

6.5.1. Что такое база данных

База данных представляет собой хранилище информации, представленное одним или несколькими файлами. В настоящее время наиболее широко распространены так называемые реляционные базы данных, в которых информация организована в виде таблиц, состоящих из строк и столбцов. Таблицы могут быть связаны между собой, образуя нередко сложную систему. Создание серьезных баз данных и доступ ним обеспечивает особая система управления базами данных (СУБД). Вместе с тем доступ к базам данных через СУБД возможен и из приложений, написанных на том или ином языке программирования.

PHP имеет наборы специальных функций для работы с многими СУБД, такими как Microsoft SQL Server (MS SQL), MySQL, Oracle, PostgreeSQL, mSQL, Informix, Sybase, Ingres и IBM DB2. Кроме того, PHP поддерживает стандарт ODBC (Open Database Connectivity — открытый интерфейс доступа к базам данных), которому соответствуют почти все реляционные СУБД. Используя ODBC, можно обращаться к таким СУБД, как MS Access. Очень популярной СУБД, с которой часто работают посредством PHP, является MySQL.

СУБД устанавливается независимо от PHP. Чтобы работать с ней посредством PHP-сценария, следует установить соответствующую данной СУБД библиотеку функций. Например, при работе в Windows для СУБД MySQL версии не старше 4.0 требуется библиотека php_mysql.dll, для MySQL версии 4.1 и старше — php_mysqli.dll; для MS SQL Server — php_mssql.dll; для PostgreeSQL — php_pgsql.dll. При установке PHP библиотеки для работы с СУБД автоматически не устанавливаются. Как это сделать вручную, описано в *приложении 2*.

Библиотеки для разных СУБД содержат различные наборы функций. Если вы собираетесь работать с базой данных в своем приложении, то должны, преж-

де всего, узнать, какие СУБД поддерживает ваша хостинговая компания, поскольку от этого зависит перечень необходимых вам функций.

Большинство современных СУБД поддерживают язык SQL (Structured Query Language — язык структурированных запросов), посредством которого формируются выражения, называемые запросами к базе данных. Запросы, выполняемые СУБД, позволяют создавать и удалять базы данных и таблицы, содержащиеся в базах данных, а также добавлять, редактировать, удалять и выбирать данные в таблицах. Кроме того, они позволяют обрабатывать данные. Более того, с помощью SQL-запросов можно осуществлять администрирование баз данных: определять пользователей (имя и пароль), устанавливая их права (привилегии) на доступ к базе данных.

Запросы на языке SQL могут возвращать, а могут и не возвращать данные. Например, запрос на выборку данных по некоторому критерию возвращает данные, а запрос на создание новой таблицы или, например, удаление записей — нет. Результатом выполнения SQL-запроса на выборку данных является набор записей (временная таблица), удовлетворяющих условию запроса. Если таких записей в базе данных не оказалось, то возвращаемая таблица будет пустой. Если же запрос вернул какие-то записи, то их набор доступен PHP-сценарию и вы можете работать с ним как с массивом.

Чтобы работать с базой данных, необходимо знать язык SQL. Наиболее часто используемые выражения этого языка имеют довольно простой синтаксис. Для отправки SQL-запросов к базе данных из PHP-сценария применяются специальные функции, принимающие в качестве параметра строку, содержащую SQL-выражение. Язык SQL развивается, что отражается в его версиях. Возможно, не все конструкции этого языка последней версии поддерживает данная СУБД. Поэтому, прежде чем использовать SQL, вы должны обратиться к справочной документации по СУБД, чтобы узнать особенности SQL-интерпретатора, встроенного в нее. Так, например, если объявлено, что ваша СУБД поддерживает SQL:92, то это еще не означает, что ее SQL-интерпретатор соответствует требованиям SQL:92 в полной мере. Возможны частные отклонения от официального стандарта. Работая с конкретной СУБД, вам следует выяснить все нюансы с помощью документации и/или экспериментов.

Для эффективного применения SQL-запросов знания языка SQL еще не достаточно. Кроме этого, необходимо понимать, как устроены и взаимосвязаны таблицы базы данных. Однако изложение основ языка SQL и теории реляционных баз данных выходит за рамки данной книги. В несложных проектах база данных содержит одну или несколько таблиц. Например, для счетчика посещений Web-страницы, баннера, гостевой книги и даже форума вполне можно обойтись одной таблицей.

В случае одной таблицы задача проектирования базы данных сводится к правильному определению столбцов (полей) — их имен, типов и размеров. База данных может состоять из нескольких не связанных между собой таблиц. Эта ситуация принципиально не отличается от однотабличной базы данных. Если вам необходимо спроектировать систему из нескольких связанных таблиц, а вы новичок в этом деле, то обратитесь к специальной литературе по проектированию баз данных. Далее мы рассмотрим создание гостевой книги на основе простой однотабличной базы данных. Главная наша цель — понять, как работать с базой данных в PHP-сценариях.

6.5.2. Основные средства PHP для взаимодействия с базой данных

При взаимодействии сценария с базой данных обычно выполняются следующие действия:

1. Подключение к серверу базы данных (предполагается, что сервер базы данных запущен).
2. Передача SQL-запроса, который должен быть выполнен СУБД. SQL-запрос представляется на языке SQL, оформляется в виде строки и передается специальной PHP-функции в качестве параметра. Эта функция свяжется с СУБД и передаст ей SQL-запрос. СУБД его выполнит (если нет ошибок и других проблем). Если SQL-запрос предполагает возврат данных, то они будут получены с помощью специальной PHP-функции.
3. Получение от СУБД данных, соответствующих переданному ранее SQL-запросу. С помощью специальных функций можно получить данные и далее их обработать (например, просто вывести на экран).
4. Обработка полученных данных. Вы можете отформатировать по своему усмотрению или подвергнуть более сложному анализу и преобразованию полученные данные.
5. Отключение от базы данных, если не предполагается повторение этапов 2 — 4. Отключаться от базы данных рекомендуется всякий раз, как только вы прекращаете работать с ней на более или менее длительный срок.

Эти и другие действия в PHP-сценарии выполняют соответствующие функции.

Подключение к базе данных

Будем считать, что база данных (возможно, пустая) уже создана. Чтобы выполнить все остальное, необходимо к ней подключиться.

Подключение к базе данных (БД) осуществляется с помощью специальных функций PHP, которым передаются в качестве параметров следующие данные:

- *местоположение* (`host`) — для многих серверов БД это доменное имя или IP-адрес компьютера, на котором расположена БД; если она находится на том же компьютере, что и модуль PHP, то достаточно указать `localhost`. Для MS SQL Server указывают его имя. При необходимости можно указать через двоеточие порт, который прослушивается сервером базы данных;
- *регистрационное имя* (`username`) — строка символов, соответствующая имени пользователя, под которым он может получить доступ к БД (его сообщает администратор БД);
- *пароль* (`password`) — строка символов, которая вместе с регистрационным именем служит для обеспечения доступа к БД (его сообщает администратор БД);
- *имя базы данных* — СУБД может поддерживать несколько БД, каждая из которых должна иметь уникальное имя. Вы должны указать имя конкретной БД, с которой хотите работать.

При подключении к базам данных под управлением разных СУБД в PHP используются различные функции. Далее приведены примеры подключения к базе данных для двух популярных СУБД:

- MS SQL Server

```
$connect=mssql_connect(имя_сервера_БД, username, password);
$db=mssql_select_db(имя_БД, $connect);
```

- MySQL 4.0:

```
$connect=mysql_connect(host, username, password);
$db=mysql_select_db($connect, имя_БД);
```

Для подключения к БД MySQL и MS SQL Server требуются две функции:

- `…_connect()` — выполняет соединение с сервером БД и возвращает идентификатор соединения при успехе и `false` — при неудаче. При этом для MySQL в первом параметре этой функции указывают `host` (доменное имя или IP-адрес компьютера), а для MS SQL Server — имя сервера БД, устанавливаемое при регистрации сервера. Идентификатор соединения, возвращаемый функцией, используется в сценарии, в частности, при выборе базы данных, создании запросов и т. д.;
- `…_select()` — выбирает конкретную базу данных на сервере, на который указывает идентификатор `$connect`, возвращенный первой функцией. Эта функция возвращает `true` при успешном завершении операции и `false` — в противном случае. Если требуемая база данных еще не создана, то это

необходимо сделать. Так, в MS SQL Server базу данных (хотя бы пустую) можно создать с помощью утилиты Enterprise Manager или с помощью соответствующих выражений SQL, которые можно ввести в окно интерфейса или сформировать и выполнить посредством PHP-сценария. В MySQL вы можете ввести необходимые SQL-выражения в командной строке либо написать соответствующий PHP-сценарий, а затем выполнить его.

Отключение от базы данных осуществляет функция, которой передается идентификатор соответствующего подключения. Например, `mysql_close($connect)` — для MySQL, `mssql_close($connect)` — для MS SQL Server.

Подробные сведения о функциях работы с базами данных как рассмотренных в этой книге, так и других, можно найти в справочном руководстве по PHP.

Передача запросов к базе данных

Итак, вы подключились к серверу баз данных и выбрали требуемую конкретную базу данных. Теперь с ней можно выполнять различные манипуляции, которые задаются выражениями на языке SQL.

SQL-запрос к базе данных в качестве строкового параметра передается специальной функции PHP, которая, в свою очередь, передает его СУБД. СУБД выполняет этот запрос, а PHP-функция возвращает указатель на временную таблицу данных, являющуюся результатом выполнения запроса.

Для различных СУБД функции передачи запроса похожи, но все же имеют некоторые отличия:

- MS SQL Server:
  ```
  mssql_query(SQL_запрос, идентификатор_соединения);
  ```
- MySQL 4.0:
  ```
  mysql_query(SQL_запрос, идентификатор_соединения);
  ```

Пример подключения к серверу MS SQL Server

```
/* Подключение к серверу БД: */
$connect=mssql_connect("VADIM", "guest")or
              exit("Не удалось соединиться с сервером");
/* Выбор БД: */
$db=mssql_select_db("gbook", $connect)
              exit("Не удалось выбрать БД");
$strsql="SELECT * FROM gb";     // строка с SQL-запросом
$selectall=mssql_query($strsql, $connect);     // выполнение SQL-запроса
$mssql_close($connect);     // отключение от БД
```

Здесь производится подключение к MS SQL Server с именем `VADIM` (устанавливается при регистрации сервера) для пользователя с регистрационным именем `guest` без пароля. Затем выбирается база данных `gbook` и выполняется SQL-запрос на выборку из таблицы `gb` всех полей (столбцов) и записей (строк). Наконец, соединение с БД разрывается.

Подключение к серверу и выбор БД может оказаться неудачным по тем или иным причинам. Однако выполнение сценария продолжится, что может оказаться бессмысленным и вызвать лавинообразный поток сообщений об ошибках. Предотвратить это помогает функция `exit()` (или `die()`), выводящая указанное сообщение и прекращающая дальнейшее выполнение сценария.

Пример подключения к серверу MySQL

```
/* Подключение к серверу БД: */
$connect=mysql_connect("localhost", "guest") or
            exit("Не удалось соединиться с сервером".mysql_error());
/* Выбор БД: */
$db= mysql_select_db("gbook", $connect) or
            exit("Не удалось выбрать БД"".mysql_error());
$strsql="SELECT * FROM gb";     // строка с SQL-запросом
$selectall= mysql_query($strsql, $connect);     // выполнение SQL-запроса
$ mysql_close($connect);     // отключение от БД
```

Здесь функция `mysql_error()` возвращает описание возникшей ошибки.

Примечание

Подключение к базе данных для различных СУБД осуществляют в большинстве случаев похожие функции. Однако могут быть и существенные отличия. Так, например, подключение к серверу PostgreeSQL и выбор базы данных выполняет одна функция:

```
$connect=pg_connect("host=host user=username password=password
        dbname=имя_БД");
```

Обработка данных в сценарии

Выбор нужных данных из одной или нескольких таблиц выполняет SQL-запрос, начинающийся с ключевого слова (оператора) `SELECT`. Критерий (условие) выбора данных формулируется в выражении SQL-запроса вслед за ключевым словом `WHERE`. Результатом выполнения такого запроса является временная таблица (возможно, пустая, если нет данных, удовлетворяющих условию запроса). Функция PHP, выполняющая запрос, возвращает указатель

на полученную в результате временную таблицу. Этот указатель используется в PHP-сценарии для доступа к данным с помощью специальных функций.

Довольно часто данные из временной таблицы читают в массив, а затем его обрабатывают в сценарии (например, выводят на экран, изменяют значения, производят вычисления и т. п.). Если требуется изменить значения полей таблицы, добавить новую или удалить имеющуюся запись, то формируют строку с соответствующим SQL-запросом, а затем передают ее в качестве параметра функции, выполняющей запрос (например, `mssql_query()`).

Чтение в массив записей временной таблицы, полученной в результате выполнения SQL-запроса, осуществляют функции, в имени которых присутствует слово `fetch` (достать). Например, функция `mssql_fetch_assoc()` читает текущую запись таблицы БД MS SQL Server в массив с символьными индексами (ключами). При этом значения индексов совпадают с именами соответствующих полей таблицы. Функция `mssql_fetch_row()` создает аналогичный массив, но с числовыми индексами, значения которых совпадают с номерами полей таблицы. Для MySQL существуют аналогичные функции: `mysql_fetch_assoc()`, `mysql_fetch_row()`.

6.5.3. Создание гостевой книги на основе базы данных

В качестве иллюстрации применения функций работы с базами данных рассмотрим создание гостевой книги. Проект гостевой книги в общих чертах обсуждался в *разд. 6.4.7*. Его особенность состояла лишь в том, что данные сохранялись не в базе данных под управлением СУБД, а в текстовом файле с символьным разделителем полей (т. е. в CSV-файле). Теперь мы задействуем СУБД, которую необходимо предварительно установить.

Создание гостевой книги на основе базы данных состоит из двух основных этапов:

1. Создание БД для хранения сообщений посетителей. Для гостевой книги достаточно одной таблицы в этой БД.
2. Разработка серверных сценариев, которые должны осуществлять подключение к БД, выводить ее содержимое для просмотра, предоставлять форму для ввода новых записей, а также добавлять эти записи в БД.

В данном разделе описывается работа с MS SQL Server 2000. Применение других серверов БД (например, MySQL) более или менее аналогично. Администрирование MS SQL Server, создание базы данных, ее таблиц, а также заполнение таблиц данными реализует специальная утилита Enterprise Manager, которая имеет удобный пользовательский интерфейс.

Создание базы данных

Для создания базы данных под управлением MS SQL Server запустите утилиту Enterprise Manager (рис. 6.10). В древовидном списке в левой части окна этой утилиты раскройте группу узлов с именем сервера БД: **Microsoft SQL Servers/SQL Server Group/*Имя_сервера* (Windows NT)**. В рассматриваемом примере сервер БД называется **VADIM**. Щелкните правой кнопкой мыши на узле **Databases** и в раскрывшемся контекстном меню выберите пункт **New Database** (Новая база данных). В открывшемся диалоговом окне введите имя БД, например **Гостевая книга**, и щелкните на кнопке **OK**. В результате это имя появится в древовидном списке в группе **Databases**.

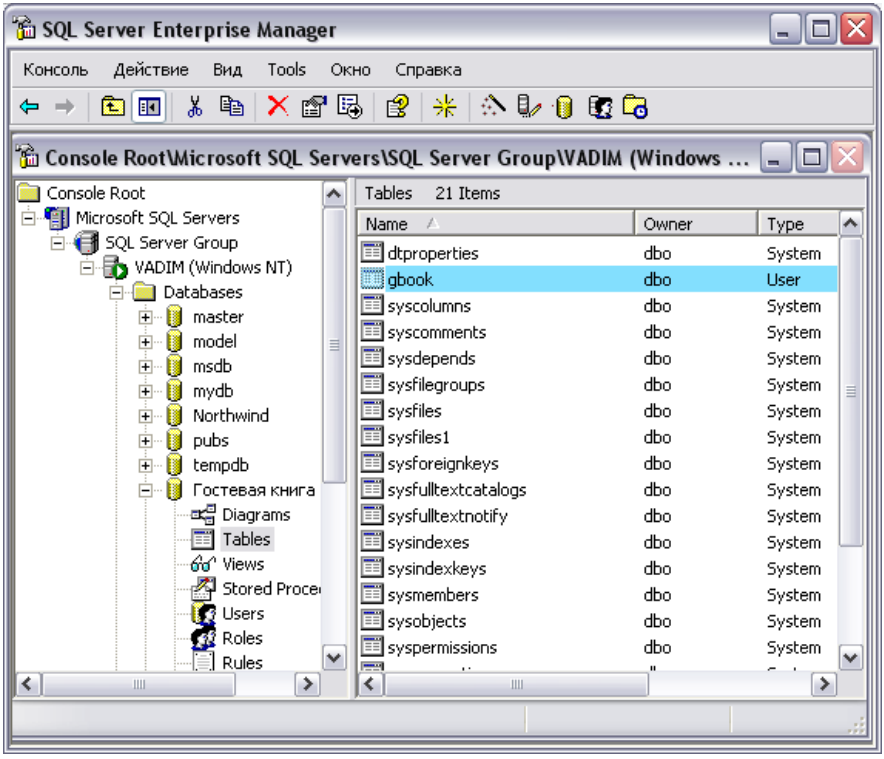

Рис. 6.10. Окно **SQL Server Enterprise Manager**

Создание таблицы для хранения данных

Итак, база данных создана, но она пока не содержит ни одной пользовательской таблицы. Теперь необходимо создать таблицу, в которой будут храниться данные гостевой книги. Для этого следует щелкнуть правой кнопкой мыши

на имени БД в древовидном списке и в контекстном меню выбрать опцию **Создать**, а затем — **Table**. В результате откроется диалоговое окно **New Table in '*имя БД*' on '*Имя сервера БД*'** (рис. 6.11), в котором можно определить поля (столбцы) таблицы, т. е. задать их имена, типы, размеры и другие характеристики. Помимо полей, которые предназначены для хранения данных, зададим поле **recid**, значения которого присваиваются автоматически и являются уникальными идентификаторами записей. Характеристики поля **recid** показаны на рис. 6.11.

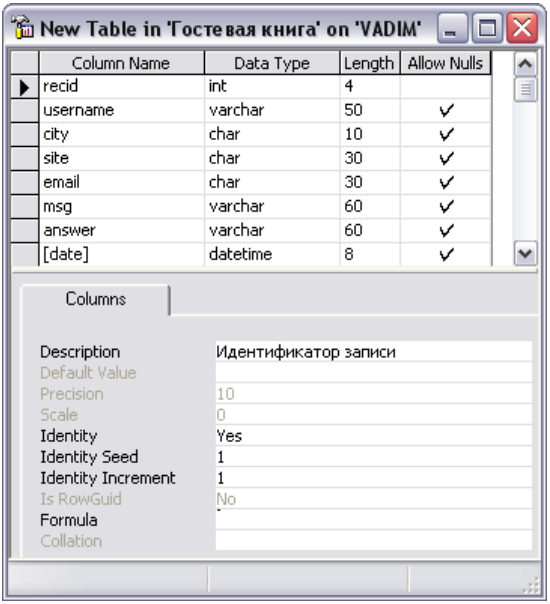

Рис. 6.11. Окно для определения полей таблицы

После закрытия окна **New Table** предлагается сохранить таблицу под требуемым именем (например, **gbook**). Вручную вводить данные в таблицу не нужно, поскольку это будут делать посетители гостевой книги.

Определение регистрационного имени и пароля пользователя

Теперь следует определить пользователя (регистрационное имя и пароль), а также его права по работе с таблицей. Сначала определите пользователя. Для этого в древовидном списке окна **SQL Server Enterprise Manager** щелкните правой кнопкой мыши на узле **Users** (Пользователи) и в контекстном меню выберите опцию **New Database User** (Новый пользователь БД). В результате

откроется диалоговое окно **Database User Properties — New User** (Свойства пользователя БД — Новый пользователь). В раскрывающемся списке **Login name** (Регистрационное имя) выберите пункт **<new>**. Откроется окно **SQL Server Login Properties** (Свойства регистрации), в котором в поле **Name** введите любое значение, например **Гость**; установите переключатель **SQL Server Athentication** (Идентификация сервером), а в поле **Password** введите значение пароля или оставьте его пустым, если пароль не нужен**.** Для посетителей гостевой книги пароль можно не вводить. Нажмите кнопку **ОК**, чтобы сохранить настойки и вернуться к окну **Database User Properties — New User**. В этом окне в раскрывающемся списке **Login name** выберите имя, которое было создано ранее (например **Гость**) и щелкните на кнопке **ОК**. В результате имя пользователя появится в списке пользователей базы данных **Гостевая книга**.

Аналогичным образом определяются регистрационное имя и пароль владельца гостевой книги. В данном случае пароль необходим.

Определение прав пользователя

Далее необходимо определить права созданного пользователя БД. Посетителю гостевой книги, в отличие от ее владельца, не стоит предоставлять все возможные права по манипуляции базой данных.

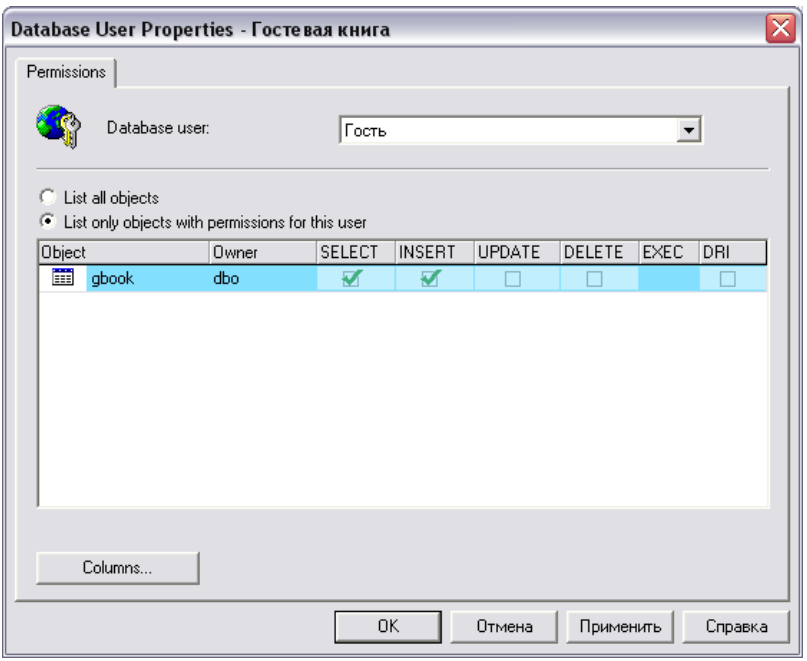

Рис. 6.12. Окно установки прав пользователя по работе с данными

В окне **SQL Server Enterprise Manager** найдите в списке пользователей имя **Гость**, щелкните на нем правой кнопкой мыши и выберите опцию **Все задачи | Manage Permissions** (Управление разрешениями). В раскрывшемся окне **Database User Properties — Гостевая книга** (рис. 6.12) в строке, значение поля **Object** которой равно **gbook** (имя таблицы), установите флажки в полях **SELECT** и **INSERT**. Тем самым пользователю предоставляется право выполнять указанные SQL-операторы, обеспечивающие выборку и добавление данных соответственно. Это типичные права посетителя гостевой книги.

Аналогично определяются права владельца гостевой книги. Кроме прав гостя (выборка и добавление данных), он должен иметь возможность изменять (**UPDATE**) и удалять (**DELETE**) записи.

Сценарии для взаимодействия с посетителем

Для взаимодействия гостевой книги с посетителем потребуются два сценария. Первый формирует страницу, содержащую все записи из базы данных гостевой книги. Кроме того, на этой странице находится кнопка **Добавить запись**, щелчок на которой приводит к переходу ко второму сценарию, предоставляющему форму для ввода данных и выполняющему запись их в БД. Страницы, созданные этими сценариями, выглядят в окне браузера так же, как на рис. 6.8 и 6.9. Пусть эти сценарии сохранены в файлах gbookdb.php и gbadd_db.php.

В листинге 6.51 приведен код сценария из файла gbookdb.php.

Листинг 6.51. Вывод содержимого гостевой книги (gbookdb.php)

```
<?php
/* Форма с кнопкой для перехода к странице добавления записи */
echo "<center><h2>Моя гостевая книга</h2>";
echo "<form action=gbadd_db.php method=POST>
<input type=submit name=addrecord value='Добавить запись'>
</form><br>";
/**** Подключение к БД ********************************************/
$servername="VADIM";     // имя сервера БД
$dbname="[Гостевая книга]";     // имя БД
$user='Гость';          // login пользователя
/* Соединение с сервером: */
$connect=mssql_connect($servername,$user) or
                    exit("Не удалось соединиться с сервером");
/* Выбор БД: */
```

```
$db=mssql_select_db($dbname, $connect) or
                    exit("Не удалось выбрать БД");
/**** Выбор всех записей ****************************************/
$strsql="SELECT * FROM gbook ORDER BY date DESC";
$rez=mssql_query($strsql,$connect);     // выполнение запроса
/**** Вывод содержимого БД ***************************************/
while ($row=mssql_fetch_assoc($rez)){ // цикл по всем записям таблицы
     $i++;
       $t=$row['date'];
     echo "<b>$i. {$row['username']}</b> {$row['city']} $t<br>
     <b>Сайт:</b> <a href={$row['site']}>{$row['site']}</a>
       <b> E-mail:</b> <a href={$row['email']}>{$row['email']}</a><br>
     <br><textarea rows=4 cols=60>{$row['msg']}</textarea><br>";
     if ($row['answer']) {
          echo "<b>Ответ:</b><br>
              <textarea rows=4 cols=60>{$row['answer']}</textarea>";
     }
     echo "<hr><br>";
}
mssql_close($connect); // отключение от БД
?>
```

ВНИМАНИЕ!

Если имя базы данных содержит пробелы, дефисы и некоторые другие специальные символы, то его следует заключить в квадратные скобки. Например, `$dbname="[Гостевая книга]";`.

SQL-запрос в данном сценарии возвращает все записи и все поля таблицы `gbook`, упорядоченные по убыванию значений поля `date` (дата и время). Таким образом, новые записи располагаются в таблице выше старых.

Листинг 6.52 содержит код сценария из файла gbadd_db.php.

Листинг 6.52. Отображение формы ввода и добавление данных в гостевую книгу (gbadd_db.php)

```
<?php
if ($_POST['addrecord']){     // если требуется добавить запись
     /* Вывод формы для ввода сообщения */
     echo "<html>
      <center><h2>Моя гостевая книга</h2></center>
```

```
      <form action=gbadd_db.php method=POST>
     <table>
     <tr><td>Ваше имя:</td>
     <td><input type=text name=username size=20></td></tr>
     <tr><td>Город:</td>
     <td><input type=text name=city size=20></td></tr>
     <tr><td>URL Вашего сайта:</td>
     <td> <input type=text name=site size=40></td></tr>
     <tr><td>E-mail:</td>
     <td><input type=text name=email size=40></td></tr>
     </table>
     Сообщение:<br>
     <textarea name=msg rows=4 cols=60></textarea><br>
     <input type=submit name=addok value='Добавить запись'>
     <input type=reset name=reset value='Отмена'>
     </form><br>
     </html>";
} elseif ($_POST['addok']) { // если подтвердили ввод в форму
     header("Location:gbookdb.php"); /* Переход к странице показа
                                        гостевой книги */
     if (!$_POST['msg']){exit;};    // если сообщение пусто, то выход
   $t=date("Y-m-d H:i:s");          // текущие дата и время
/**** Подключение к БД ************************************/
   $servername="VADIM";             // имя сервера БД
   $dbname="[Гостевая книга]";      // имя БД
   $user='Гость';                   // login пользователя
   $connect=mssql_connect($servername,$user);  // соединение с сервером
   $db=mssql_select_db($dbname, $connect);        // выбор БД
   $strsql="INSERT INTO gbook
   (username, city, site, email, msg, date)
   VALUES (\"{$_POST['username']}\", \"{$_POST['city']}\",
   \"{$_POST['site']}\",\"{$_POST['email']}\",
   \"{$_POST['msg']}\",
   CAST('".$t."' AS datetime))";
$rez=mssql_query($strsql,$connect);    // выполнение запроса
mssql_close($connect);                 // отключение от БД
}
?>
```

SQL-запрос в данном сценарии содержит SQL-функцию `CAST()` для преобразования типов данных. В нашем случае необходимо привести строковое значение даты-времени `$t` к типу `datetime` поля `date` таблицы БД.

Определение параметров подключения к БД можно разместить в отдельном файле и включить его в оба рассмотренных сценария с помощью функции `include()`.

В данном примере мы не предусмотрели обработку возможных ошибок подключения и выполнения запроса к БД. В реальном проекте это следует сделать.

Сценарии для владельца гостевой книги

Для владельца гостевой книги необходимо создать отдельную, защищенную паролем страницу, а в сценариях ее обслуживания установить другие параметры соединения с БД (регистрационное имя и пароль). Владелец книги, в отличие от обычного посетителя, обладает правами на изменение и удаление записей.

Страницы для владельца книги и для обычного посетителя внешне могут быть аналогичными, но есть и отличия. Видимо, проще всего каждую выводимую запись гостевой книги заключить в HTML-теги формы с кнопками типа submit **Принять изменения** и **Удалить**. Кроме того, в скрытое поле формы следует поместить значение поля БД **recid** (идентификатор записи). Тогда в сценарии обработки щелчков на этих кнопках полученное значение скрытого поля можно использовать в условии SQL-запроса к БД. Например, пусть кнопка **Удалить** имеет имя (значение атрибута `name`) `delete`, а скрытое поле со значением поля БД **recid** — `recid` (т. е. такое же, что и поле БД). Тогда для удаления записи может быть применен следующий фрагмент PHP-кода:

```
if ($_POST['delete']){
     $strsql="DELETE FROM gbookdb WHERE recid={$_POST['recid']}";
     mssql_query($strsql,$connect);
}
```

Предположим, кнопка **Принять изменения** имеет имя (значение атрибута `name`) `update`, а поле формы для ввода ответа — `answer` (т. е. такое же, что и поле БД). Допустим, вы хотите сохранить только введенный вами ответ. Тогда можно воспользоваться следующим фрагментом PHP-кода:

```
if ($_POST['update']){
      $strsql="UPDATE gbookdb SET answer=\"{$_POST['recid']}\"
       WHERE recid={$_POST['recid']}";
       mssql_query($strsql,$connect);
}
```

В качестве упражнения попробуйте завершить проект гостевой книги самостоятельно.

6.5.4. Применение SQLite

В PHP имеется расширение SQLite, которое позволяет сохранять данные в текстовых файлах и манипулировать ими посредством языка SQL. Если объем ваших данных относительно невелик, структура базы данных проста, а требования по защите данных невысоки, то применение SQLite — хорошее решение. При этом вам не нужны ни СУБД, ни знание команд для работы с файлами. Достаточно знать только SQL. Так например, гостевую книгу или форум для личного сайта вполне можно сделать на основе SQLite.

В *разд. 6.4.7* мы уже рассматривали проекты, основанные на использовании текстовых файлов с табличной структурой данных (CSV-файлы). Однако там мы не могли манипулировать данными с помощью SQL. SQLite сочетает хранение табличных данных в текстовых файлах с возможностью работы с ними посредством SQL. В Windows расширению SQLite соответствует библиотека php_sqlite.dll. Несмотря на то, что база данных SQLite хранится в виде текстового файла, ее создание, редактирование и другие манипуляции следует выполнять не с помощью текстового редактора, а посредством специальных PHP-функций.

Чтобы открыть базу данных (установить соединение с базой данных), необходимо вызвать функцию:

```
sqlite_open(имя_БД);
```

Данная функция возвращает указатель на открытую базу данных или `false`, если операция не удалась. Если указанная в качестве параметра база данных не существует, то предпринимается попытка ее создать. Имя базы данных — абсолютный или относительный путь к файлу. Например, `sqlite_open("/db/mydb")` попытается открыть или создать файл базы данных `mydb`, расположенный в папке `db`, которая находится в домашнем каталоге Web-сервера.

После открытия файла базы данных можно использовать SQL-выражения, которые выполняются с помощью функций:

- `sqlite_query(SQL_запрос)` — для выполнения SQL-запросов, возвращающих данные (например, `SELECT …`); при успешном выполнении запроса возвращает указатель на временную таблицу данных, в противном случае — `false`;
- `sqlite_exec(SQL_запрос)` — для выполнения SQL-запросов, не возвращающих данные (например, `DELETE…`); при успешном выполнении запроса возвращает `true`, а в противном случае — `false`.

В частности, после создания новой базы данных следует выполнить SQL-запрос на создание таблицы, в которой будут храниться собственно данные.

По завершении работы с базой данных ее следует закрыть с помощью функции `sqlite_close(идентификатор_БД)`.

В листинге 6.53 в качестве примера приведен сценарий, который выполняет следующие действия.

1. Открывает базу данных `mydatabase`. Если она не существует, то предпринимается попытка создать ее. Идентификатор базы данных присваивается переменной `$db`.
2. Создает таблицу `mytable` с тремя столбцами: `first_name`, `last_name` и `zarplata`. Перед вызовом функции выполнения запроса `sqlite_query()` указан оператор `@` подавления вывода сообщений об ошибке. Дело в том, что если таблица `mytable` уже существует, то запрос на ее создание не выполнится, но будет выведено соответствующее сообщение, которое в данном случае нежелательно. При первом выполнении сценария таблица должна быть создана, а при последующих — нет.
3. Добавляет в таблицу `mytable` две записи.
4. Выбирает из таблицы `mytable` все имеющиеся в ней записи.
5. В цикле с помощью функции `sqlite_fetch_array()` и оператора `echo` данные выборки отображает в окне браузера. Функция `sqlite_fetch_array()` возвращает значения столбцов текущей записи таблицы как массив и переводит указатель на следующую запись.
6. Закрывает базу данных с идентификатором `$db`.

При первом выполнении данный сценарий выведет в окно браузера две строки данных.

Листинг 6.53. Пример использования SQLite

```
<?php
/* Открытие БД: */
$db=sqlite_open("mydatabase") or exit("Не удалось открыть БД");
/* Создание таблицы: */
$strsql="CREATE TABLE mytable1 (
      first_name varchar(20),
      last_name varchar(20),
```

```
      zarplata numeric(8,2))";
$result = @sqlite_query($db,$strsql);     /* @ — для подавления вывода
                                             сообщений об ошибке */
/* Вставка записи: */
$strsql="INSERT INTO mytable1 VALUES ('Петр', 'Петров', 1500.50)";
$result=sqlite_query($db,$strsql);
/* Вставка еще одной записи: */
$strsql="INSERT INTO mytable1 VALUES ('Иван','Иванов', 3500)";
$result=sqlite_query($db,$strsql);
/* Выборка всех записей из таблицы: */
$strsql="SELECT *FROM mytable1";
$result=sqlite_query($db,$strsql);
/* Отображение записей: */
while ($row = sqlite_fetch_array($result)){
     echo      $row['first_name']." ".
               $row['last_name']." - ".
              $row['zarplata']."<br>";
}
sqlite_close($db);     // закрытие БД
?>
```

Получив с помощью SQL-запроса набор записей, можно воспользоваться функциями для перемещения по записям:

- `sqlite_next()` — переход к следующей записи; возвращает `true` в случае успеха и `false`, если предпринята попытка перейти к несуществующей записи;
- `sqlite_prev()` — переход к предыдущей записи; возвращает `true` в случае успеха и `false`, если предпринята попытка перейти к несуществующей записи;
- `sqlite_rewind()` — переход к первой записи; возвращает `false`, если набор не содержит записей, и `true` — в противном случае;
- `sqlite_current(указатель)` — возвращает текущую запись в указанном наборе как массив; если указатель текущей записи находится за последней записью набора, то функция возвращает `false`;
- `sqllite_seek(указатель, номер_записи)` — переход к записи с указанным номером (нумерация начинается с нуля) в указанном наборе; если указанная запись отсутствует в наборе, то возвращается `false`, иначе — `true`.

6.6. Другие возможности PHP

В данной книге мы рассмотрели элементы PHP, наиболее важные с практической и, главным образом, с учебной точек зрения, а также технологии программирования на их основе. Это всего лишь небольшая часть PHP, однако, изучив ее, вы сможете более или менее уверенно проникнуть дальше в мир его возможностей.

В заключение я хотел бы обратить ваше внимание еще на несколько интересных и практически важных вопросов, оставшихся за рамками настоящей книги:

- отправка данных по электронной почте с помощью почтового SMTP-сервера и PHP-функции `mail()`;
- работа с файлами через протокол FTP;
- разработка сетевых (клиент-серверных) приложений и, в частности, создание Web-служб с использованием протокола SOAP (простого протокола доступа к объектам);
- работа с PDF- и XML-документами.

Необходимую информацию по данным темам можно почерпнуть в электронном справочном руководстве, а также в других книгах по PHP.

ПРИЛОЖЕНИЯ

Приложение 1

Краткий справочник по HTML и CSS

П1.1. Теги HTML

В табл. П1.1—П1.10 приведены наиболее часто употребляемые теги HTML и их атрибуты. Поскольку многие различные теги имеют одинаковые атрибуты, последние разделены на следующие группы:

- базовые (применяемые к большинству тегов) — `id`, `class`, `style`, `title`;
- языковые (указывающие на особенности языка) — `dir`, `lang`;
- атрибуты-события (используемые в сценариях) — `onclick`, `ondblclick`, `onkeydown`, `onkeypress`, `onkeyup`, `onmousedown`, `onmousemove`, `onmouseout`, `onmouseover`, `onmouseup`.

  Для некоторых элементов возможны и другие события (например, `onload`, `onselect` и т. п.), но они перечисляются отдельно;
- фокус (ссылки на состояние элемента, когда он выделен и готов принять данные от пользователя) — `accesskey`, `onblur`, `onfocus`, `tabindex`.

Обязательные атрибуты помечены восклицательным символом (!). Теги, не рекомендованные W3C к использованию, помечены символом (*).

Таблица П1.1. *Структура документа*

Тег	Атрибуты	Описание
<!>	–	Определяет комментарий, который игнорирует анализатор HTML

Таблица П1.1 *(продолжение)*

Тег	Атрибуты	Описание
<!DOCTYPE> Варианты для документов HTML 4.01: <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/HTML4.01/loose.dtd"> <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN" "http://www.w3.org/TR/HTML4.01/frameset.dtd"> <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/HTML4.01/strict.dtd">	–	Объявляет тип документа (DTD — Document Type Definition). Размещается в начале документа перед тегом <HTML>. Возможны три версии DTD: 1) Strict DTD — исключает все нерекомендованные теги и атрибуты, такие как <FONT>, <CENTER>, align и другие, которые можно заменить соответствующими параметрами CSS; 2) Transitional DTD — включает то же, что и Strict DTD, а также все нерекомендованные элементы для обеспечения обратной совместимости со всеми существующими браузерами. 3) Frameset DTD — включает то же, что и Transitional DTD, а также элементы, необходимые для Web-страниц с фреймовой структурой
<BASE>	href (!), id, target	Устанавливает базовый URL-адрес документа, используемый во всех случаях применения относительных ссылок
<BODY> </BODY>	Базовые, языковые, атрибуты-события; onload, onunload, alink, background, bgcolor, link, text, vlink, leftmargin, rightmargin, topmargin, bottommargin, bgroperties, scroll	Определяет основную часть документа

***Таблица П1.1** (продолжение)*

Тег	Атрибуты	Описание
<DIV> </DIV>	Базовые, языковые, атрибуты-события; align	Осуществляет логическое разделение документа, создавая фрагмент внутри него
<HEAD> </HEAD>	id, profile	Заключает в себе теги, содержащие невидимую информацию о документе, которая должна быть загружена в браузер первой. Обычно содержит контейнеры <TITLE>, <STYLE>, <SCRIPT> и <META>
<HTML> </HTML>	–	Идентифицирует документ как содержащий элементы HTML
<LINK>	Базовые, языковые, атрибуты-события; charset, href, hreflang, media, rel, rev, target, type	Служит для обозначения связей между текущим и другим документами. Обычно используется для указания связи с внешней таблицей стилей
<META>	Языковые; content, http-equiv, id, name, sheme	Предоставляет различные типы невидимой информации и инструкций для браузера. Должен размещаться внутри контейнера <HEAD>
<SPAN> </SPAN>	Базовые, языковые, атрибуты-события	Используется с таблицей стилей для определения нестандартных атрибутов текста на странице
<STYLE> </STYLE>	Языковые; id, media, title, type (!)	Заключает в себе таблицу стилей

***Таблица П1.2.** Заголовки и названия*

Тег	Атрибуты	Описание
<H_n> </H_n>,n=1,2,…,n	Базовые, языковые, атрибуты-события; align	Заголовок *n*-го уровня
<TITLE> </TITLE>	Языковые; id	Название документа, отображаемое в заголовке окна браузера. Данный тег обязателен; размещается в контейнере <HEAD>

Таблица П1.3. *Абзацы и строки*

Тег	Атрибуты	Описание
 	Базовые; clear	Вставляет конец строки (переход на другую строку)
<CENTER> </CENTER> (*)	Базовые, языковые, атрибуты-события	Центрирует заключенные в него компоненты по ширине страницы или содержащего их контейнера
<HR>	Базовые, языковые, атрибуты-события; align, noshade, size, width	Помещает на страницу горизонтальную линию
<NOBR> </NOBR>	–	Запрещает обтекание текстом
<P> </P>	Базовые, языковые, атрибуты-события; align	Обозначает абзац (переход на другую строку с пропуском одной строки)
<WBR>	–	Вставляет мягкий перенос строки в блок текста, заключенный в теге <NOBR>

Таблица П1.4. *Стили*

Тег	Атрибуты	Описание
<ADDRESS> </ADDRESS>	Базовые, языковые, атрибуты-события	Определяет информацию об адресе, контрольной сумме и авторстве
<B> </B>	То же	Делает начертание текста полужирным
<BASEFONT> (*)	id, color, face, size	Устанавливает шрифт, используемый по умолчанию
<BIG> </BIG>	Базовые, языковые, атрибуты-события	Задает размер шрифта на 1 больше текущего (размер основного шрифта по умолчанию равен 3)
<BLOCKQUOTE> </BLOCKQUOTE>	Базовые, языковые, атрибуты-события; cite	Обозначает отступ в тексте. Обычно в контейнер <BLOCKQUOTE> включают несколько абзацев, заключенных в контейнеры <P>
<CITE> </CITE>	Базовые, языковые, атрибуты-события	Выделяет текст курсивом; часто используется как знак авторского права или для выделения цитат

Таблица П1.4 *(продолжение)*

Тег	Атрибуты	Описание
<CODE> </CODE>	То же	Представляет текст моноширинным шрифтом. Обычно используется для отображения примеров программных кодов
<DFM> </DFM>	То же	Применяется для пометки терминов, используемых в первый раз
<EM> </EM>	То же	Выделяет текст, обычно курсивом
<FONT> </FONT> (*)	Базовые, языковые, атрибуты-события; color, face, size	Определяет вид, размер и цвет шрифта для текста
<I> </I>	Базовые, языковые, атрибуты-события	Делает начертание текста курсивным
<KBD> </KBD>	То же	Используется для отображения на экране текста, который должен ввести пользователь
<PLAINTEXT> </PLAINTEXT>	То же	Представляет текст моноширинным шрифтом без обработки тегов
<PRE> </PRE>	Базовые, языковые, атрибуты-события; width	Представляет текст так, как он был введен в текстовом редакторе
<S> </S> (*)	Базовые, языковые, атрибуты-события	Выводит текст зачеркнутым
<SAMP> </SAMP>	То же	Показывает текст как пример программы
<SMALL> </SMALL>	То же	Определяет, что текст должен быть показан уменьшенным шрифтом относительно основного
<STRIKE> </STRIKE> (*)	То же	Выводит текст зачеркнутым
<STRONG> </STRONG>	То же	Выделяет текст жирным шрифтом
<STYLE> </STYLE>	Языковые; id, media, title, type (!)	Заключает в себе таблицу стилей
	Базовые, языковые, атрибуты-события	Выделяет текст, делая его нижним индексом

Таблица П1.4 *(окончание)*

Тег	Атрибуты	Описание
	То же	Выделяет текст, делая его верхним индексом
<TT> </TT>	То же	Задает моноширинный шрифт
<U> </U> (*)	То же	Делает текст подчеркнутым
<VAR> </VAR>	То же	Задает мелкий шрифт, иногда курсив, фиксированной ширины, используемый обычно для обозначения переменных

Таблица П1.5. *Списки*

Тег	Атрибуты	Описание
<DD> </DD>	Базовые, языковые, атрибуты-события	Используется внутри списка определений для задания текста в теге <DT>
<DIR> </DIR> (*)	То же	Служит для создания списка каталога (например, файлов). Элементы списка помещаются в контейнер <DIR> с помощью тегов <LI>
<DL> </DL>	Базовые, языковые, атрибуты-события; compact	Используется для создания списка элементов со связанными описаниями, например, списка определений. В контейнер <DL> вставляются теги <DT> и <DD>
<DT> </DT>	Базовые, языковые, атрибуты-события	Обозначает определение, описание которого находится в следующем за ним теге <DD>
<LI> </LI>	Базовые, языковые, атрибуты-события; type, start, value	Обозначает один элемент в нумерованном или маркированном списке
<OL> </OL>	Базовые, языковые, атрибуты-события; compact, start, type	Преобразует строки текста, заключенные в теги <LI>, в нумерованный список
<UL> </UL>	Базовые, языковые, атрибуты-события; compact, type	Преобразует строки текста, заключенные в теги <LI>, в маркированный список

Таблица П1.6. *Таблицы*

Теги	Атрибуты	Описание
<CAPTION> </CAPTION>	Базовые, языковые, атрибуты-события; align	Определяет заголовок таблицы или текст, предшествующий остальным ее элементам
<COL>	Базовые, языковые, атрибуты-события; align, char, charoff, span, valign, width	Задает параметры столбцов таблицы по умолчанию
<COLGROUP> </COLGROUP>	То же	Определяет контейнер для группы столбцов
<TABLE> </TABLE>	Базовые, языковые, атрибуты-события; align, bgcolor, border, cellpadding, cellspacing, frame, height, rules, summary, width	Определяет таблицу
<TBODY> </TBODY>	Базовые, языковые, атрибуты-события; align, char, charoff, valign	Указывает раздел, содержащий основную часть таблицы
<TD> </TD>	Базовые, языковые, атрибуты-события; abbr, align, axis, background, bgcolor, char, charoff, colspan, headers, height, nowrap, rowspan, scope, valign, width	Определяет ячейки таблицы
<TFOOT> </TFOOT>	То же, что и для <TBODY>	Определяет набор строк для использования в нижней части таблицы
<TH> </TH>	То же, что и для <TD>	Задает заголовок столбца таблицы; содержимое выровнено по центру и выводится жирным шрифтом
<THEAD> </THEAD>	То же, что и для <TBODY>	Определяет набор строк, используемых как заголовок таблицы
<TR> </TR>	Базовые, языковые, атрибуты-события; align, bgcolor, char, charoff, valign	Задает строку таблицы

Таблица П1.7. *Ссылки*

Тег	Атрибуты	Описание
<A> </A>	Базовые, языковые, атрибуты-события; charset, cords, href (!), hreflang, id, name (!), rel, rev, shape, target, type	Определяет гиперссылку. Обязательным является либо атрибут href (для ссылок на внешний ресурс), либо name (для ссылок на элемент внутри документа)

Таблица П1.8. *Графика, объекты, мультимедиа и сценарий*

Тег	Атрибуты	Описание
<APPLET> </APPLET> (*)	Базовые; align, alt, archive, code, codebase, height, hspace, name, object, vspace, width	Помещает апплет Java или другой выполняемый элемент на страницу
<AREA>	Базовые, языковые, атрибуты-события, фокус; alt, cords, href, nohref, shape, target	Определяет форму активного участка изображения-карты клиента. Используется внутри контейнера <MAP>
<BGSOUND> (*)	src, loop, balance, volume	Определяет фоновое звуковое сопровождение страницы. Нестандартный тег
<EMBED> </EMBED> (или <EMBED>) (*)	align, height, hidden, name, palette, plaginspage, src, width. Для Internet Explorer можно использовать дополнительные атрибуты: alt, code, codebase, units	Внедряет документы любого типа на страницу для просмотра соответствующим приложением. Нестандартный тег, который рекомендуется заменять на <OBJECT>
<IMG>	Базовые, языковые, атрибуты-события; align, alt, border, height, hspace, ismap, longdesc, lowsrc, name, src (!), usemap, vspace, width	Используется для вставки графического элемента или видеоклипа на страницу

Таблица П1.8 *(окончание)*

Тег	Атрибуты	Описание
<MAP> </MAP>	Базовые, языковые, атрибуты-события; id, name	Задает коллекцию активных участков изображения-карты клиента
<MARQUEE> </MARQUEE> (*)	Базовые, атрибуты-события; align, behavior, bgcolor, direction, height, hspace, loop, scrollamount, crolldelay, vspace, width	Создает на странице бегущую строку с элементами различного типа (текст, графические изображения, видеоклипы). Нестандартный тег, интерпретируемый Internet Explorer
<NOSCRIPT> </NOSCRIPT>	Базовые, языковые, атрибуты-события	Определяет, что будет показывать браузер, не поддерживающий сценарии
<OBJECT> </OBJECT>	Базовые, языковые, атрибуты-события; align, archive, border, classid, codebase, codetype, data, declare, height, hspace, name, standby, tabindex, type, usemap, vspace, width	Встраивает на страницу объект или другой элемент управления, не встроенный в HTML
<PARAM>	id, name, type, value, valuetype	Используется в тегах <OBJECT> и <APPLET> для установки дополнительных свойств объекта
<SCRIPT> </SCRIPT>	charset, defer, id, language, src, type	Определяет раздел страницы, который должен обрабатываться интерпретатором кода сценария

Таблица П1.9. *Формы*

Тег	Атрибуты	Описание
<BUTTON> </BUTTON>	Базовые, языковые, атрибуты-события, фокус; disabled, name, value, type	Создает на странице HTML-кнопку
<FIELDSET> </FIELDSET>	Базовые, языковые, атрибуты-события	Рисует прямоугольник вокруг содержащихся в нем элементов для обозначения связанных объектов

Таблица П1.9 *(окончание)*

Тег	Атрибуты	Описание
<FORM> </FORM>	Базовые, языковые, атрибуты-события; onsubmit, onblur, onreset, accept, accept-charset, action, enctype, method, name, target	Описывает форму на странице, которая может содержать другие компоненты и элементы управления
<INPUT>	Базовые, языковые, атрибуты-события, фокус; onselect, onchange, alt, accept, accesskey, align, checked, disabled, ismap, maxlength, name, readonly, size, src, tabindex, type, usemap, value	Задает параметры элемента управления формы
<LABEL> </LABEL>	Базовые, языковые, атрибуты-события, фокус; for	Определяет текст метки или заголовка для блока элементов управления
<LEGEND> </LEGEND>	Базовые, языковые, атрибуты-события; accesskey, align	Определяет текст, помещаемый в прямоугольник, созданный тегом <FIELDSET>
<OPTION> </OPTION>	Базовые, языковые, атрибуты-события; disabled, label, selected, value	Обозначает одну из альтернатив в раскрывающемся списке (меню), созданном тегом <SELECT>
<SELECT> </SELECT>	Базовые, языковые, атрибуты-события, фокус; onfocus, onblur, onchange, disabled, multiple,name, size, tabindex	Определяет раскрывающийся список (меню)
<TEXTAREA> </TEXTAREA>	Базовые, языковые, атрибуты-события, фокус; onselect, onchange, cols, disabled, name, readonly, rows	Определяет многострочное поле ввода, содержащее при необходимости полосы прокрутки

Таблица П1.10. *Фреймы*

Тег	Атрибуты	Описание
`<FRAME>`	Базовые; `bordercolor`, `frameborder`, `longdesc`, `marginheight`, `marginwidth name`, `noresize`, `scrolling`, `src`	Определяет конкретный фрейм внутри набора фреймов, задаваемого тегом `<FRAMESET>`
`<FRAMESET>` `</FRAMESET>`	Базовые; `onload`, `onunload`, `border`, `bordercolor`, `cols`, `frameborder`, `rows`	Определяет набор фреймов, который содержит фреймы или подчиненные наборы фреймов
`<IFRAME>` `</IFRAME>`	Базовые; `align`, `frameborder`, `height`, `hspace`, `longdesc`, `marginheight`, `marginwidth`, `name`, `scrolling`, `src`, `vspace`, `width`	Используется для создания встроенных "плавающих" фреймов на странице
`<NOFRAMES>` `</NOFRAMES>`	Базовые, языковые, атрибуты-события	Определяет, что именно будет показывать браузер, не поддерживающий фреймы

П1.2. Таблицы стилей

П1.2.1. Единицы измерения

Есть две основные категории единиц: относительные и абсолютные (кроме того, есть еще процентные). Относительные единицы предпочтительнее, поскольку определение абсолютных требует знания конкретной системы отображения: на какой принтер, монитор или другое устройство будет выводиться информация.

- **Относительные единицы.**

  Значения `em`, `en` и `ex` соответствуют типографским терминам и определяют соотношение с размерами других символов; `px` обозначает пикселы — элементы экрана, размер которых зависит от установок монитора и видеокарты пользователя.

  В Internet Explorer, начиная с версии 4, `em` и `ex` заменены на `pt`, `en` — на `px`.

- **Абсолютные единицы.**

  Это `in`, `cm`, `mm`, `pt`, `pc`: `in` дает значение в дюймах, `cm` — в сантиметрах, `mm` — в миллиметрах, `pt` — в пунктах (72 пункта на дюйм) и `pc` — в пиках (1 пика равна 12 пт). Эти единицы рекомендуется выбирать только в том случае, если вы знаете, каким будет размер рабочей области устройства вывода, т. к. браузеры будут пытаться показать все в натуральную величину.

- **Проценты.**

  Эти цифровые значения задаются как числа (с десятичной точкой или без нее), показывающие отношение к единице длины (обычно размер шрифта текущего элемента).

П1.2.2. Параметры и свойства CSS

Далее указаны параметры каскадных таблиц стилей (CSS) и соответствующие им свойства. Параметры определяются в правилах CSS, а свойства доступны из сценариев. Если многие параметры записываются через дефис, то соответствующие им свойства дефисов не допускают, а при их записи используется так называемый стиль interCap. Например, параметр `z-index` как свойство в выражениях сценария записывается в виде `zIndex`.

Свойства шрифта

Свойство *font*

Значения	`<font-size>, [/<line-height>], <font-family>`
Значение по умолчанию	Не определено
Поддерживается	Всеми элементами
Работает ли механизм наследования	Да
Возможна запись в процентах	Только для `<font-size>`, `<line-height>`

Это свойство позволяет устанавливать сразу несколько свойств шрифта в одном месте с начальными значениями, определенными для используемых свойств (т. е. значение, определенное по умолчанию для `<font-size>`, отличается от значения по умолчанию `<font-family>`). Соответствующие значе-

ния этого свойства можно разделять пробелами или запятыми, если устанавливается несколько шрифтов.

Свойство *font-family*

Свойство	`fontFamily`
Значение	Название семейства шрифта (например, Arial)
Значение по умолчанию	Устанавливается браузером
Поддерживается	Всеми элементами
Работает ли механизм наследования	Да
Возможна ли запись в процентах	Нет

Вы можете установить несколько возможных значений в порядке предпочтения (на тот случай, когда в браузере нет нужного шрифта). Для этого просто разделите их запятыми. Список следует закончить родовым названием шрифта (допустимые значения: `serif`, `sans-serif`, `cursive`, `fantasy` или `monospace`). Если имя шрифта состоит из нескольких слов, необходимо заключить эти слова в кавычки.

Свойство *font-size*

Свойство	`fontSize`
Значения	`<absolute>`, `<relative>`, `<length>`, `<percentage>`
Значение по умолчанию	`medium`
Поддерживается	Всеми элементами
Работает ли механизм наследования	Да
Возможна ли запись в процентах	Да (относительно родительского размера шрифта)

Значения для этого свойства можно задать различными способами:

- абсолютный размер — допустимые значения — `xx-small`, `x-small`, `small`, `medium`, `large`, `x-large`, `xx-large`;
- относительный размер — допустимые значения — `larger`, `smaller`;
- длина в любых единицах измерения — см. ранее;
- процентное отношение — значения представлены в процентах от родительского размера шрифта.

Свойство *font-style*

Свойство	`fontStyle`
Значения	`normal`, `italic`, `oblique`
Значение по умолчанию	`normal`
Поддерживается	Всеми элементами
Работает ли механизм наследования	Да
Возможна ли запись в процентах	Нет

Это свойство определяет стиль вашего шрифта. Если существует уже готовый вариант шрифта (например, New York Oblique), то он будет применен, в противном случае шрифт будет изменен программно.

Свойство *font-variant*

Свойство	`fontVariant`
Значения	`normal`, `small-caps`
Значение по умолчанию	`normal`
Поддерживается	Всеми элементами
Работает ли механизм наследования	Да
Возможна ли запись в процентах	Нет

`normal` — это стандартный вид, установленный по умолчанию, `small-caps` использует прописные буквы того же размера, что и обычные строчные.

Свойство *font-weight*

Свойство	`fontWeight`
Значения	`normal`, `bold`, `bolder`, `lighter` или числовые значения от 100 до 900
Значение по умолчанию	`normal`
Поддерживается	Всеми элементами
Работает ли механизм наследования	Да
Возможна ли запись в процентах	Нет

Определяет жирность шрифта текста, которая зависит от толщины линии. Если задано числовое выражение, то число обязательно должно быть кратным 100: 400 — то же, что и `normal`, 700 — то же, что и `bold`.

Свойства цвета и фона

Свойство *color*

Свойство	`color`
Значения	Название цвета или его номер
Значение по умолчанию	Зависит от браузера
Поддерживается	Всеми элементами
Работает ли механизм наследования	Да
Возможна ли запись в процентах	Нет

Устанавливает цвет текста элемента страницы. Цвет может быть задан названием (например, `green`) или номером в соответствии со шкалой RGB. Это значение может быть задано несколькими способами: в шестнадцатеричной системе, например `"#FFFFFF"`, процентами — `"80%,20%,0%"`, или в десятичной системе — `"245,0,20"`.

Свойство *background*

Свойство	`background`
Значения	`transparent`, <*цвет*>, <*URL-адрес*>, <repeat>, <*прокрутка*>, <*положение*>
Значение по умолчанию	`transparent`
Поддерживается	Всеми элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Да (относительно размера самого элемента)

Определяет параметры фона документа. Значение `transparent` соответствует отсутствию фона. Вы можете использовать для фона либо цвет, либо изображение с заданным URL-адресом. Адрес может быть абсолютным или относительным, но его следует обязательно заключить в скобки и расположить непосредственно за словом `url`:

```
BODY { background: url (http://mumu.com/image/picture.gif) }.
```

Можно комбинировать фон и изображение. В таком случае рисунок будет расположен поверх цветного фона. Цвет может быть либо чистым, либо смешанным из двух. Изображение имеет несколько настроек:

- <repeat> — может иметь значения `repeat`, `repeat-x`, `repeat-y` и `no-repeat`; если эти значения не указаны, по умолчанию задано `repeat`;

- <*прокрутка*> — определяет, будет ли изображение оставаться на месте при прокручивании страницы или прокручиваться вместе с ней; возможные значения — `fixed`, `scroll`;
- <*положение*> — определяет расположение изображения на странице; значения могут быть в процентах (горизонтальное, вертикальное), абсолютным расстоянием (горизонтальное, вертикальное) или определенными с помощью ключевых слов (допустимые значения: `top`, `middle`, `bottom`, `left`, `center`, `right`).

В следующих разделах описаны свойства, позволяющие задать свойства фона по отдельности.

Свойство *background-attachment*

Свойство	`backgroundAttachment`
Значения	`fixed`, `scroll`
Значение по умолчанию	`scroll`
Поддерживается	Всеми элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Нет

Определяет, будет ли прокручиваться фоновое изображение при пролистывании страницы или нет.

Свойство *background-color*

Свойство	`backgroundColor`
Значения	`transparent`, <*цвет*>
Значение по умолчанию	`transparent`
Поддерживается	Всеми элементами
Работает ли механизм наследования	Нет
Возможна ли процентная запись	Нет

Определяет цвет фона. Может быть один или два смешанных цвета. Цвет задается названием (например, `green`) или номером по шкале RGB. Это значение может быть задано несколькими способами: в шестнадцатеричной системе (например, `"#FFFFFF"`), процентами — `"80%,20%,0%"` или в десятичной системе — `"255,0,0"`. Синтаксис для использования двух смешанных цветов такой:

```
BODY { background-color: red/blue }.
```

Свойство *background-image*

Свойство	`backgroundImage`
Значения	`<URL-адрес>`, `none`
Значение по умолчанию	`none`
Поддерживается	Всеми элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Нет

С помощью этого свойства можно определить URL-адрес фонового изображения. Адрес может быть абсолютным или относительным, но его следует обязательно заключить в скобки и расположить непосредственно за словом `url`.

Свойства *background-position*

Свойства	`backgroundPosition`, `backgroundPositionX`, `backgroundPositionY`
Значения	`<положение>`, `<длина>`, `top`, `center`, `bottom`, `left`, `right`
Значение по умолчанию	`top`, `left`
Поддерживается	Всеми элементами
Работает ли механизм наследования	Нет
Возможна ли процентная запись	Нет

Указывают начальное положение фонового изображения с помощью двух значений, определяющих занимаемую часть страницы (по горизонтали, по вертикали) и абсолютное расстояние (в соответствующих единицах измерения, сначала по горизонтали, потом по вертикали), или используя два ключевых слова из возможных.

Свойство *background-repeat*

Свойство	`backgroundRepeat`
Значения	`repeat`, `repeat-x`, `repeat-y`, `no-repeat`
Значение по умолчанию	`repeat`
Поддерживается	Всеми элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Нет

Определяет, повторяется ли фоновое изображение для заполнения страницы или элемента. Если указаны значения `repeat-x`, `repeat-y`, то изображение повторяется лишь вдоль одного направления. По умолчанию задано повторение по обоим направлениям.

Свойства списков

Свойство *list-style*

Свойство	`listStyle`
Значения	*<ключевое_слово>*, *<положение>*, *<URL-адрес>*
Значение по умолчанию	В зависимости от браузера
Поддерживается	Всеми элементами
Работает ли механизм наследования	Да
Возможна ли запись в процентах	Нет

Определяет, как должны изображаться элементы списка. Может быть использовано для установки всех свойств. Их можно также установить по частям, прибегнув к помощи описанных далее свойств.

Свойство *list-style-image*

Свойство	`listStyleImage`
Значения	`none`, *<URL-адрес>*
Значение по умолчанию	`none`
Поддерживается	Всеми элементами
Работает ли механизм наследования	Да
Возможна ли запись в процентах	Нет

Определяет адрес изображения, служащего маркером или значком для каждого элемента списка.

Свойство *list-style-position*

Свойство	`listStylePosition`
Значения	`inside`, `outside`
Значение по умолчанию	`outside`

Поддерживается	Всеми элементами
Работает ли механизм наследования	Да
Возможна ли запись в процентах	Нет

Указывает, внутри или вне тела списка должен находиться маркер.

Свойство *list-style-type*

Свойство	`listStyleType`
Значения	`none`, `circle`, `disk`, `square`, `decimal`, `lower-alpha`, `upper-alpha`, `lower-roman`, `upper-roman`
Значение по умолчанию	`disk`
Поддерживается	Всеми элементами
Работает ли механизм наследования	Да
Возможна ли запись в процентах	Нет

Определяет вид маркера списка, используемого перед каждым элементом.

Свойства текста

Далее представлено описание свойств, определяющих различные параметры отображения текста.

Интервал между буквами: свойство *letter-spacing*

Свойство	`letterSpacing`
Значения	`normal`, `<`*длина*`>`
Значение по умолчанию	`normal`
Поддерживается	Всеми элементами
Работает ли механизм наследования	Да
Возможна ли процентная запись	Нет

Определяет расстояние между буквами. Длина обозначает добавочное расстояние между буквами, которое должно быть выражено с указанием единиц измерения.

Высота строки: свойство *line-height*

Свойство	`lineHeight`
Значения	<*число*>, <*длина*>, <*проценты*>, `normal`
Значение по умолчанию	В зависимости от браузера
Поддерживается	Всеми элементами
Работает ли механизм наследования	Да
Возможна ли запись в процентах	Да, по отношению к размеру шрифта текущего элемента

Определяет высоту текущей строки. Числовые значения интерпретируются как размер шрифта текущего элемента, умноженный на соответствующий коэффициент (например, 1,2). Если используется длина, должны указываться единицы измерения. Процентное соотношение применяется в сравнении с текущим размером шрифта и должно быть больше 100%.

Выравнивание текста: свойство *text-align*

Свойство	`textAlign`
Значения	`left`, `right`, `center`, `justify`
Значение по умолчанию	В зависимости от браузера
Поддерживается	Всеми элементами
Работает ли механизм наследования	Да
Возможна ли запись в процентах	Нет

Определяет, как текст будет выровнен относительно содержащего его элемента.

По существу, делает то же, что и тег `<DIV ALIGN= ...>`.

Украшение текста: свойства *text-decoration*

Свойства	`textDecoration`, `textDecorationLineThrough`, `textDecorationUnderline`, `textDecorationOverline`
Значения	`none`, `overline`, `underline`, `line-through`
Значение по умолчанию	`none`

Поддерживается	Всеми элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Нет

Позволяет добиться специального вида текста. Открыт для расширений, неопознанные расширения интерпретируются как подчеркивание. Это свойство не наследуется, но обычно распространяется на дочерние элементы, лежащие внутри родительских.

Красная строка: свойство *text-indent*

Свойство	`textIndent`
Значения	`<длина>`, `<`*проценты*`>`
Значение по умолчанию	`zero`
Поддерживается	Всеми элементами
Работает ли механизм наследования	Да
Возможна ли запись в процентах	Да, относительно ширины родительского элемента

Устанавливает величину отступа, в единицах измерения или в процентах от ширины родительского элемента.

Преобразование текста: свойство *text-transform*

Свойство	`textTransform`
Значения	`capitalize`, `uppercase`, `lowercase`, `none`
Значение по умолчанию	`none`
Поддерживается	Всеми элементами
Работает ли механизм наследования	Да
Возможна ли запись в процентах	Нет

Возможные варианты значений:

- `capitalize` — делает заглавной первую букву каждого слова в текстовом элементе;
- `uppercase` — преобразует все буквы в словах текстового элемента в заглавные;

- `lowercase` — преобразует все буквы в словах текстового элемента в строчные;
- `none` — снимает все установки, приобретенные в результате наследования.

Расположение по вертикали: свойство *vertical-align*

Свойство	`verticalAlign`
Значения	`baseline`, `sub`, `super`, `top`, `text-top`, `middle`, `bottom`, `text-bottom`, <*проценты*>
Значение по умолчанию	`baseline`
Поддерживается	Встроенными элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Да, по отношению к высоте строки

Определяет расположение текущего текстового элемента по вертикали:

- `baseline` — устанавливает выравнивание по образцу родительского элемента;
- `middle` — устанавливает вертикальную среднюю линию текстового элемента на основе родительского элемента плюс половина строки последнего;
- `super` — переводит текстовый элемент в верхний регистр;
- `sub` — переводит текстовый элемент в нижний регистр;
- `text-top` — выравнивает текстовый элемент по верху текста, набранного шрифтом родительского элемента;
- `text-bottom` — выравнивает текстовый элемент по низу текста, набранного шрифтом родительского элемента;
- `top` — выравнивает текстовый элемент по верху самого высокого элемента текущей строки;
- `bottom` — выравнивает текстовый элемент по низу самого низкого элемента текущей строки.

Блочные свойства

Значения этих свойств позволяют установить характеристики области вокруг элемента (символа, изображения и т. п.).

Свойства *border-top-color*, *border-right-color*, *border-left-color*, *border-bottom-color*, *border-color*

Свойства	`borderTopColor`, `borderRightColor`, `borderLeftColor`, `borderBottomColor`, `borderColor`
Значения	*<цвет>*
Значение по умолчанию	*<нет цвета>*
Поддерживается	Блоками элементов и замещаемыми элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Нет

Определяют цвет четырех сторон рамок. Если параметр `цвет` заменить на URL-адрес изображения, оно будет повторяться, образуя рамку.

Свойства *border-top-style*, *border-right-style*, *border-left-style*, *border-bottom-style*, *border-style*

Свойства	`borderTopStyle`, `borderRightStyle`, `borderLeftStyle`, `borderBottomStyle`, `borderStyle`
Значения	`none`, `solid`, `double`, `groove`, `ridge`, `inset`, `outset`
Значение по умолчанию	none
Поддерживается	Блоками элементов и замещаемыми элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Нет

Определяют стиль четырех сторон рамок.

Свойства *border-top*, *border-right*, *border-left*, *border-bottom*, *border*

Свойства	`borderTop`, `borderRignt`, `borderLeft`, `borderBottom`, `border`
Значения	*<ширина_рамки>*, *<стиль_рамки>*, *<цвет>*
Значение по умолчанию	`medium`, `none`, *<нет цвета>*

Поддерживается	Блоками элементов и замещаемыми элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Нет

Определяют свойства четырех сторон рамок. Работают так же, как свойства отступов (см. далее), за исключением того, что рамка видна:

- `<ширина_рамки>` — может иметь значения `medium`, `thin` и `thick` или быть задана в единицах измерения;
- `<стиль_рамки>` — может принимать значения `none` или `solid`.

Аргумент `color` определяет цвет заполнения фона элемента, пока он загружается, а также позади прозрачных частей элемента. Если передать вместо него адрес изображения, то рисунок будет повторяться, заполняя контур рамки.

Свойства *border-top-width, border-right-width, border-left-width, border-bottom-width, border-width*

Свойства	`borderTopWidth`, `borderRightWidth`, `borderLeftWidth`, `borderBottomWidth`, `borderWidth`
Значения	`thin`, `medium`, `thick`, `<длина>`
Значение по умолчанию	`medium`
Поддерживается	Блоками элементов и замещаемыми элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Нет

Определяют ширину рамки вокруг элемента. Каждая сторона может быть задана соответствующим свойством. Можно заменить установку четырех индивидуальных свойств одним свойством `border-width` так же, как и для свойств отступа `margin`.

Свойство *clear*

Свойство	`clear`
Значения	`none`, `both`, `left`, `right`
Значение по умолчанию	`none`
Поддерживается	Всеми элементами

Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Нет

Указывает, что следующие элементы должны быть показаны ниже элемента, выровненного по левому или правому краю. По умолчанию текст обтекает такие элементы.

Свойство *clip*

Свойство	`clip`
Значения	`rect(<top><right><bottom><left>)`, `auto`
Значение по умолчанию	`auto`
Поддерживается	Всеми элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Нет

Определяет, какая часть элемента видна. Все, что находится за пределами области, указанной этим свойством, увидеть нельзя.

Свойство *display*

Свойство	`display`
Значения	`""`, `none`
Значение по умолчанию	`""`
Поддерживается	Всеми элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Нет

Это свойство указывает, будет ли отображен элемент.

Свойство *float*

Свойство	`styleFloat`
Значения	`none`, `left`, `right`
Значение по умолчанию	`none`
Поддерживается	Тегами `<DIV>`, `<SPAN>` и замененными элементами

Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Нет

Определяет обтекание элемента другими элементами слева или справа вместо помещения их под ним.

Свойства *height*

Свойства	`height`, `pixelHeight`, `posHeight`
Значения	`auto`, <*длина*>
Значение по умолчанию	`auto`
Поддерживается	Тегами `<DIV>`, `<SPAN>` и замененными элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Нет

Устанавливают высоту элемента и измеряют ее, если это необходимо. Значение возвращается в виде строки, включающей единицы измерения (`px`, `%` и т. д.). Для получения числового значения используется свойство `posHeight`.

Свойства *left*

Свойства	`left`, `pixelLeft`, `posLeft`
Значения	`auto`, <*длина*>, <*проценты*>
Значение по умолчанию	`auto`
Поддерживается	Всеми элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Да, относительно ширины родительского элемента

Задают горизонтальную координату элемента, позволяя корректно устанавливать и передвигать элементы. Значение возвращается как строка, включающая единицы измерения (`px`, `%` и т. д.). Для получения значения в виде числа используется свойство `posLeft`.

Свойства *margin-top*, *margin-right*, *margin-left*, *margin-bottom*, *margin*

Свойства	`marginTop`, `marginRight`, `marginLeft`, `marginBottom`, `margin`
Значения	`auto`, <*длина*>, <*проценты*>

Значение по умолчанию	`zero`
Поддерживается	Блоками элементов и замещаемыми элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Да, относительно ширины родительского элемента

Определяют размеры отступов вокруг данного элемента. Свойство `margin` служит для задания четырех отступов. Если задано несколько значений, но меньше четырех, оставшиеся противоположные стороны будут равными. Эти значения устанавливают минимальное расстояние между текущим элементом и остальными элементами.

Свойство *overflow*

Свойство	`overflow`
Значения	`none`, `clip`, `scroll`
Значение по умолчанию	`none`
Поддерживается	Всеми элементами
Работает ли механизм наследования	Нет
Возможна ли процентная запись	Нет

Свойство управляет тем, что произойдет, если содержимое элемента выйдет за свои границы:

- `visible` — если содержимое и выйдет за пределы отведенного ему места, элемент все равно будет показан полностью (значение по умолчанию);
- `hidden` — выступающие за границы элемента-контейнера части содержимого будут обрезаны;
- `auto` — добавляет полосы прокрутки, если содержимое выходит за границы элемента-контейнера;
- `scroll` — добавляет полосы прокрутки в любом случае.

Свойства *padding-top, padding-right, padding-left, padding-bottom, padding*

Свойства	`paddingTop`, `paddingRight`, `paddingLeft`, `paddingBottom`, `padding`
Значения	`auto`, <*длина*>, <*проценты*>

Значение по умолчанию	`zero`
Поддерживается	Блоками элементов и замещаемыми элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Да, относительно ширины родительского элемента

Определяют расстояние между содержимым и рамкой элемента. Свойство `padding` используется для задания четырех таких расстояний. Если имеется несколько значений, но меньше четырех, оставшиеся противоположные расстояния будут равными.

Свойство *position*

Свойство	`position`
Значения	`absolute`, `relative`, `static`
Значение по умолчанию	`relative`
Поддерживается	Всеми элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Нет

Устанавливает, каким образом вычисляется положение элемента в плоскости экрана:

- `absolute` — элемент будет поставлен в некоторое положение относительно фона и станет двигаться вместе с ним;
- `static` — элемент будет поставлен в некоторое положение относительно фона, но не будет двигаться при прокручивании страницы;
- `relative` — элемент будет помещен в обычное положение относительно других в соответствии с положением в исходном коде (значение по умолчанию).

Свойства *top*

Свойства	`top`, `pixelTop`, `posTop`
Значения	`auto`, <*длина*>, <*проценты*>
Значение по умолчанию	`auto`
Поддерживается	Всеми элементами

Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Да, относительно ширины родительского элемента

Задают или возвращают вертикальную координату элемента, позволяя корректно устанавливать и передвигать элементы. Значение возвращается как строка, включающая единицы измерения (`px`, `%` и т. д.). Для получения значения в виде числа используется свойство `posTop`.

Свойство *visibility*

Свойство	`visibility`
Значения	`visible`, `hidden`, `inherit`
Значение по умолчанию	`inherit`
Поддерживается	Всеми элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Нет

Позволяет элементу быть видимым или невидимым на странице. Невидимые (`hidden`) элементы занимают то же место и так же влияют на расположение других элементов, как и видимые (`visible`), но становятся прозрачными. Это свойство может быть использовано для отображения лишь одного из элементов, занимающих одно и то же место:

- `visible` — элемент виден на экране;
- `hidden` — элемент не виден на экране;
- `inherit` — элемент виден на экране, если его родительский элемент является видимым.

Свойства *width*

Свойства	`width`, `pixelWidth`, `posWidth`
Значения	`auto`, <*длина*>, <*проценты*>
Значение по умолчанию	`auto` за исключением элементов с внутренними установками размера
Поддерживается	Тегами `<DIV>`, `<SPAN>` и замещаемыми элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Да, относительно ширины родительского элемента

Устанавливают ширину элемента и измеряют ее, если это необходимо. Значение возвращается как строка, включающая тип единиц измерения (`px`, `%` и т. д.). Для получения значения в виде числа используется свойство `posWidth`.

Свойство *z-index*

Свойство	`zIndex`
Значения	<*число*>
Значение по умолчанию	В зависимости от контекста в исходном тексте HTML
Поддерживается	Всеми элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Нет

Свойство `z-index` указывает, в каком порядке элементы будут перекрывать друг друга. Элементы с более высоким *z*-индексом появятся поверх элементов с более низким *z*-индексом. При положительных значениях элемент ставится перед обычным текстом, при отрицательных — позади него. Таким образом, на страницах создается набор плоских слоев, для которых задается порядок перекрывания.

Основные свойства печати

Свойство *page-break-after*

Свойство	`pageBreakAfter`
Значения	`<auto>`, `<always>`, `<left>`, `<right>`
Значение по умолчанию	`<auto>`
Поддерживается	Всеми элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Нет

Управляет началом и концом страницы, разрывом страницы и тем, где на странице содержание продолжится, т. е. слева или справа.

Свойство *page-break-before*

Свойство	`pageBreakBefore`
Значения	`<auto>`, `<always>`, `<left>`, `<right>`

Значение по умолчанию	`<auto>`
Поддерживается	Всеми элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Нет

Управляет началом и концом страницы, разрывом страницы и тем, где на странице содержание продолжится, т. е. слева или справа.

Свойства фильтров

Графические фильтры могут быть установлены только в Internet Explorer. Все фильтры вызываются с помощью ключевого слова `filter`. Есть два вида фильтров: *статические* и *динамические*. Динамические фильтры разделены на две группы: управляющие переходом (переходные) и управляющие появлением объекта на экране. Фильтры первой группы предназначены для того, чтобы объект накладывался на другие объекты или уходил с соседних объектов. Фильтры второй группы заставляют объекты появляться или исчезать одним из 23 возможных способов. Все фильтры вызываются одинаково из каскадной таблицы стилей:

```
filter: имя_фильтра(параметр1, параметр2, ...)
```

Существует 14 типов статических фильтров и 2 типа динамических.

Visual filters

Свойство	`объект.style.<имя_фильтра> (параметр1, ...)`
Значения	`<имя_фильтра>(параметр1, параметр2, ...)`
Значение по умолчанию	Нет примененного фильтра
Поддерживается	Всеми элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Нет

Управляет внешним видом объектов при помощи набора встроенных фильтров. В табл. П1.11 приведен список доступных фильтров с указанием их функций.

Таблица П1.11. *Визуальные фильтры*

Фильтр	Функция
alpha	Управляет степенью видимости объекта
blur	Создает эффект движения объекта
chroma	Переводит определенный цвет изображения в прозрачный (transparent)
dropShadow	Отображает силуэт определенного цвета для выбранного объекта, создавая иллюзию, что объект находится над страницей, отбрасывая тень
fliph	Отражает объект симметрично центральной горизонтальной оси
flipv	Отражает объект симметрично центральной вертикальной оси
glow	Создает для объекта эффект сияния
gray	Отображает объект в градациях серого
invert	Обращает цвета объекта
light	Моделирует освещение объекта различными источниками цвета
mask	Отображает прозрачные (transparent) символы определенным цветом, а непрозрачные делает прозрачными
shadow	Отрисовывает "боковые" грани объекта, создавая впечатление объемности
wave	Выполняет синусоидальное преобразование объекта вдоль вертикальной оси
xray	Изменяет глубину цвета объекта и отрисовывает черно-белым, делая его похожим на рентгеновский снимок

revealTrans

Свойство

```
объект.style.revealtrans
 (duration = <длительность>,
  transition = <тип_перехода>)
```

Значения

```
revealtrans
 (duration = <длительность>,
  transition = <тип_перехода>)
```

Значение по умолчанию — Нет примененного фильтра

Поддерживается — Всеми элементами

Работает ли механизм наследования — Нет

Возможна ли запись в процентах — Нет

Позволяет открывать или закрывать одни элементы другими, используя 23 встроенных шаблона (табл. П1.12).

Параметры

- ❒ *<длительность>* — время, за которое превращение с помощью фильтра произойдет полностью (в миллисекундах);
- ❒ *<тип_перехода>* — целое число, соответствующее номеру применяемого фильтра.

Таблица П1.12. *Встроенные шаблоны*

Тип превращения	**Номер фильтра**
Box In (стягивающийся прямоугольник)	0
Box Out (расширяющийся прямоугольник)	1
Circle In (стягивающийся круг)	2
Circle Out (расширяющийся круг)	3
Wipe Up (стирание вверх)	4
Wipe Down (стирание вниз)	5
Wipe Right (стирание вправо)	6
Wipe Left (стирание влево)	7
Vertical Blinds (вертикальные жалюзи)	8
Horisontal Blinds (горизонтальные жалюзи)	9
Checkerboard Across (сужающиеся клетки шахматной доски)	10
Checkerboard Down (закрывающаяся шахматная доска)	11
Random Dissolve (случайный наплыв)	12
Split Vertical In (вертикальное деление внутрь)	13
Split Vertical Out (вертикальное деление наружу)	14
Split Horisontal In (горизонтальное деление внутрь)	15
Split Horisontal Out (горизонтальное деление наружу)	16
Strips Left Down (стирание влево вниз)	17
Strips Left Up (стирание влево вверх)	18
Strips Right Down (стирание вправо вниз)	19
Strips Right Up (стирание вправо вверх)	20
Random Bars Horisontal (случайные горизонтальные полосы)	21
Random Bars Vertical (случайные вертикальные полосы)	22
Random selection of (0—22) (случайный выбор из предыдущих вариантов)	23

blendTrans

Свойство	*объект*.style.blendtrans
Значения	blendTrans (duration= <*длительность*>), (duration = <*длительность*>)
Значение по умолчанию	Нет примененного фильтра
Поддерживается	Всеми элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Нет

Позволяет управлять появлением и исчезновением выбранных видимых объектов. Здесь <*длительность*> — время, за которое действие фильтра полностью завершится (в миллисекундах).

Свойство *cursor*

Свойство	cursor
Значения	auto, crosshair, default, hand, move, e-resize, ne-resize, nw-resize, n-resize, se-resize, sw-resize, s-resize, w-resize, text, wait, help
Значение по умолчанию	auto
Поддерживается	Всеми элементами
Работает ли механизм наследования	Нет
Возможна ли запись в процентах	Нет

Определяет вид указателя мыши.

Приложение 2

Установка Web-сервера и PHP

П2.1. Установка Web-сервера

В качестве Web-сервера можно использовать, например, Apache или Microsoft Internet Information Services (IIS, Информационные службы Интернета). IIS для Windows NT/2000/XP Professional поставляется вместе с дистрибутивом операционной системы, но не устанавливается автоматически. В Windows 98 SE программным обеспечением Web-сервера, поставляемым вместе с операционной системой, является Personal Web Server (PWS, персональный Web-сервер). Однако PWS не поддерживается Windows 2000/XP.

Чтобы установить Web-сервер IIS под Windows XP, необходимо выполнить следующие действия:

1. В Панели управления выберите утилиту **Установка и удаление программ**.
2. Щелкните на кнопке **Добавление компонентов Windows**. Откроется окно **Мастер компонентов Windows**.
3. Следуя появляющимся на экране инструкциям, установите IIS с компакт-диска дистрибутива Windows. При необходимости удалите или добавьте компоненты. Дополнительно вы можете установить почтовый и FTP-сервер.

Чтобы проверить IIS, установленный на вашем локальном компьютере, запустите браузер и в поле адреса введите `localhost`. В Windows XP Professional откроются два окна браузера со справочной информацией. Здесь можно найти достаточно много сведений как по IIS, так и по ASP.

Для настройки сервера с целью, например, создания дополнительных виртуальных каталогов, установки разрешений или для перезапуска сервера в случае неполадок выполните команду **Пуск | Панель управления | Администрирование | Internet Information Services**. В результате откроется диалоговое окно **Internet Information Services** (консоль управления IIS), с помощью которого можно произвести необходимые настройки.

Папка, используемая для размещения Web-публикаций, называется также *домашним каталогом*. При установке IIS создается домашний каталог \Inetpub\wwwroot, который определен по умолчанию. В этой папке сохраняются все файлы вашего приложения. Однако в качестве домашнего каталога можно выбрать и другую папку. Все ваши файлы, размещенные в домашнем каталоге, можно вызвать из браузера по адресу: **http://*имя_сервера/имя_файла***. Здесь ***имя_сервера*** — сетевое имя или цифровой IP-адрес компьютера, выполняющего роль сервера. Если сервер установлен на вашем локальном компьютере, то вызов файла можно выполнить с помощью следующего URL: **http://localhost/*имя_файла***. Если сервер расположен на другом компьютере локальной сети и имеет имя, например, myserver, а вас интересует файл mytestfile.php, находящийся в домашнем каталоге сервера, то введите **http://myserver/mytestfile.php**. Аналогично, если Web-сервер расположен на компьютере с IP-адресом, например, 192.168.2.103, то обратиться к сценарию можно по адресу **http://192.168.2.103/mytestfile.php**.

П2.2. Установка PHP

Если же вы намерены применять PHP, то необходимо установить модуль PHP и настроить Web-сервер на взаимодействие с ним.

П2.2.1. Установка модуля PHP

Файлы модуля PHP находятся на сайте **www.php.net**. Зайдите на этот сайт и перейдите в раздел **Downloads** (Загрузки), а затем — в раздел **Windows Binaries**, чтобы получить модуль PHP для Windows. В этом разделе щелкните на одной из двух ссылок, соответствующих последней версии PHP:

- **PHP *версия* zip package** — полный пакет PHP (CGI и CLI);
- **PHP *версия* installer** — программа автоматической установки (инсталлятор) PHP CGI.

Здесь *версия* — номер версии модуля PHP. К моменту завершения работы над книгой новейшей версией была 5.2.5.

Модуль PHP CGI предназначен только для работы с Web-сервером, а модуль PHP CLI — для автономной работы с интерфейсом командной строки.

Программа автоматической установки PHP CGI имеет графический пользовательский интерфейс, позволяющий выбрать требуемые параметры. Здесь она не будет описываться. Рекомендую установить полный пакет, представленный на сайте архивным ZIP-файлом. В этом случае модуль PHP придется

устанавливать вручную, что совершенно не сложно, а полученные навыки пригодятся в дальнейшем. В этом разделе описана установка полного пакета.

Предположим, что в данный момент доступна версия 5.2.5. Для выбора полного пакета PHP этой версии щелкните на ссылке **PHP 5.2.5 zip package**. В результате будет выведен список зеркальных сайтов. Для России выберите **ru2.php.net**. Щелчок на этой ссылке приведет к открытию диалогового окна, в котором следует выполнить действия.

1. Выбрать режим сохранения файла.
2. Указать папку на вашем компьютере, где требуется сохранить ZIP-файл (архив исходных файлов PHP).
3. Щелкнуть на кнопке **Save** (Сохранить). Начнется перекачка файла в указанную папку. По окончании загрузки в этой папке появится файл с именем, например, php-5.2.5-win32.zip. Цифры 5.2.5 указывают номер версии PHP.

Теперь необходимо распаковать загруженный архивный файл в папку, в которой будет размещаться модуль PHP вместе с другими его файлами. Разархивируйте загруженный ZIP-файл, например, php-5.2.5-win32.zip в папку, в которой вы хотите установить модуль PHP, например, C:\php. Не рекомендуется устанавливать модуль PHP в папку, имя которой содержит пробелы, например, Programs Files. Папка, в которую вы извлекли файлы PHP, называется корневой папкой PHP. Если вы поместили файлы в папку C:\php, то именно она и будет корневой. В корневой папке PHP (например, C:\php) появятся несколько папок (dev, ext, extras, PEAR) и файлов. Модулю PHP CLI соответствует исполняемый файл php.exe, а модулю PHP CGI — php-cgi.exe.

Итак, модуль PHP с сопутствующими ему файлами размещен на диске компьютера. Теперь его необходимо настроить, т. е. установить параметры.

П2.2.2. Настройка модуля PHP

Параметры PHP имеют значения, установленные по умолчанию. В ряде простейших случаев они годятся, но в дальнейшем их наверняка придется корректировать.

Параметры настройки модуля PHP записаны в текстовом конфигурационном файле php.ini. В системе Windows в зависимости от ее версии этот файл должен находиться в папке:

- WINDOWS — в системах Windows 98/Me/Xp;
- WINNT — в системах Windows NT/2000.

При использовании инсталлятора файл php.ini создается автоматически. При ручной установке необходимо исходный конфигурационный файл php.ini-dist

из корневой папки модуля PHP скопировать под именем php.ini в нужную системную папку, в зависимости от версии Windows. Так например, если модуль PHP расположен в папке C:\php, то в системе Windows XP требуется скопировать файл C:\php\php.ini-dist в папку C:\Windows и там переименовать его в php.ini. В этом текстовом файле находятся параметры (директивы) модуля PHP и комментарии. Символом комментария является точка с запятой. Параметры записываются в виде *имя_параметра=значение*. Чтобы отключить параметр, следует поставить точку с запятой в начале строки, где этот параметр записан, а чтобы включить его, — удалить точку с запятой.

Если вы используете Web-сервер IIS, то необходимо изменить значение одного из параметров в файле php.ini. Для этого откройте этот файл в текстовом редакторе (например, Блокноте) и найдите в нем строку:

```
;cgi.force_redirect = 1.
```

В начале этой строки удалите точку с запятой (признак комментария) и замените значение 1 на 0. После этого, разумеется, следует сохранить изменения в файле.

В конфигурационном файле php.ini содержится много подобных строк. В дальнейшем вам придется изменять значения некоторых параметров.

Теперь модуль PHP установлен в своем базовом составе.

Чтобы сценарии, написанные на языке PHP, могли выполняться на стороне Web-сервера, последний необходимо настроить на использование модуля PHP. Для этого выполните последовательность действий.

1. Откройте консоль управления IIS (**Панель управления | Администрирование | Internet Information Services**).
2. Откройте окно свойств параметров своего Web-узла. Для этого в окне консоли IIS щелкните правой кнопкой мыши на имени Web-узла (это может быть **Веб-узел по умолчанию**) и в контекстном меню выберите опцию **Свойства**.
3. Перейдите на вкладку **Домашний каталог** и щелкните на кнопке **Настройка**. В результате откроется окно **Настройка приложения**. Перейдите на вкладку **Сопоставления** и щелкните на кнопке **Добавить**.
4. В поле **Исполняемый файл** введите путь к интерпретатору PHP, например, C:\php\php-cgi.exe. В поле **Расширение** введите `.php`. Установите флажок **Обработчик сценариев**. Аналогично можно добавить другие расширения файлов, которые должны ассоциироваться с интерпретатором PHP.
5. Щелкните на кнопке **ОК**. В результате в списке сопоставлений, расположенном в окне **Настройка приложения**, отобразится соответствующее сопоставление расширения файла программе, которая будет его обрабатывать.

П2.2.3. Установка расширений PHP

Интерпретатор PHP состоит из ядра (core), поддерживающего основную функциональность языка, и расширений (extensions). Некоторые расширения устанавливаются по умолчанию при установке PHP. Другие входят в дистрибутив, но их необходимо установить отдельно (например, расширение, содержащее функции работы с графическими изображениями, системами управления базами данных (СУБД), PDF-документами и др.). Чтобы узнать, какие расширения установлены, достаточно вызвать PHP-функцию `phpinfo()`, которая выводит сначала параметры ядра PHP, а затем список всех установленных расширений. Однако это можно сделать, если вы уже умеете работать с PHP.

Вот некоторые из наиболее популярных расширений, которые устанавливают вручную:

- расширения для работы с базами данных (СУБД) без использования интерфейса ODBC. Расширение для поддержки ODBC включено по умолчанию, а для каждой используемой СУБД требуется установить отдельное расширение (библиотеку функций). Далее приведены три такие расширения:
  - MySQL — php_mysql.dll;
  - MS SQL Server — php_mssql.dll;
  - PostgeeSQL — php_pgsql.dll;
- библиотека GD2 функций для работы с графическими изображениями — php_gd2.dll.

Примеры, приведенные в данной книге, потребуют установки графической библиотеки и одного из расширений для работы с СУБД.

Файлы расширений (динамически подключаемые библиотеки) находятся в папке ext, расположенной в корневой папке модуля PHP, например, C:\php\ext. Чтобы установить расширение, выполните следующие действия.

1. Скопируйте DLL-файл требуемого расширения из папки ext в корневую папку PHP, например, C:\php.
2. Внесите изменения в конфигурационном файле php.ini. Найдите в нем строку вида:

   `;extension=`*`файл_расширения`*.

   Например, если вы устанавливаете расширение для работы с СУБД MS SQL Server, то требуемая строка в php.ini имеет вид:

   `;extension=php_mssql.dll`.
3. В начале найденной строки удалите точку с запятой и сохраните изменения.

П2.2.4. Проверка работоспособности Web-сервера и обработчика PHP

Предположим, вы установили Web-сервер и модуль PHP на своем компьютере. Чтобы убедиться в том, что можно начинать работу, выполните следующие действия.

1. Откройте текстовый редактор, например, Блокнот Windows и введите следующий текст:

   ```
   <?php phpinfo() ?>.
   ```

2. Сохраните введенный текст в обычном текстовом файле test.php в домашнем каталоге Web-сервера. По умолчанию таким каталогом для IIS является C:\Inetpub\wwwroot. Если вы его не изменяли, то должен быть создан файл C:\Inetpub\wwwroot\test.php.

3. Откройте Web-браузер, например, Microsoft Internet Explorer и в адресной строке введите `http://localhost/test.php` (префикс http:// можно не указывать). Вместо `localhost` можно указать сетевое имя вашего компьютера либо его десятичный IP-адрес. Если Web-сервер установлен на другом компьютере в локальной сети, к которой вы подключены, то следует ввести сетевое имя или IP-адрес того компьютера, а не вашего. В результате в браузере должен отобразиться список параметров PHP.

Итак, мы проверили работоспособность взаимодействия Web-сервера и модуля PHP CGI. В данной книге работа с модулем PHP в режиме командной строки не рассматривается. Замечу только, что не все средства PHP CGI имеются в PHP CLI.

Литература

1. Гудман Д., Моррисон М. JavaScript. Библия для пользователя. 5-е изд.: Пер. с англ. — М.: Диалектика, 2006. — 1184 с.
2. Пауэлл Т., Шнайдер Ф. Полный справочник по JavaScript. 2-е изд.: Пер. с англ. — М.: Издательский дом "Вильямс", 2007. — 960 с.
3. Дунаев В. HTML, скрипты и стили. 2-е изд. — СПб.: БХВ-Петербург, 2008. — 1024 с.
4. Дунаев В. Сценарии для Web-сайта: PHP и JavaScript, 2-е издание. — СПб.: БХВ-Петербург, 2008. — 576 с.
5. Дунаев В. Самоучитель PHP. — СПб.: Питер, 2007. — 284 с.
6. Кузнецов М., Симдянов И., Голышев С. PHP 5. Практика разработки Web-сайтов. — СПб.: БХВ-Петербург, 2007. — 960 с.
7. Дунаев В. Самоучитель JavaScript. 2-е изд. — СПб.: Питер, 2005. — 395 с.
8. Дунаев В. Основы Web-дизайна. — СПб.: БХВ-Петербург, 2006. — 512 с.
9. Дунаев В. Базы данных. Язык SQL. 2-е изд. — СПб.: БХВ-Петербург, 2007. — 320 с.

Предметный указатель

A C

AJAX 307, 324
Cookie 272
CSV-файл 462

E, L, S

ECMAScript 70
LiveScript 66
SQL 487

А

Атрибут
  Language 72

В

Выражение 60
  элементарное 113, 118, 121, 123, 346, 347
Вычисление интеграла 186
Вычисление производной 189

Г, З

Гостевая книга 479
Зарезервированное слово 60
Знак операции 60

И

Идентификатор 60
Исключение 416
Исключительная ситуация 137, 217, 309

К

Календарь 199
Класс 53
Класс PHP 407
  private 413
  protected 413
  public 414
  базовый 407
  наследник 407
  ограничение доступа 412
Ключевое слово 60
Константа
  NaN 94
Константы PHP 341
  математические 372
  создание 341
Конструктор
  объекта 56, 208

Л

Лексема 60
Литерал 60

М

Массив 39, 164
  ассоциативный 166, 210
  двумерный 167
  длина 164
  индекс 164
  копирование 168
  многомерный 167
  одномерный 167
Массив PHP 381
  автоглобальный *См.* суперглобальный
  двумерный 384
  запись элементов в переменные 393
  копирование 387
  одномерный 384
  перемещение 390
  создание 381
  сортировка 387
  суперглобальный 397
Метод:
  close() 256
  window.open() 256
  alert() 82
  confirm() 83
  prompt() 84

О

Область видимости переменной 103
Объект 50, 52
  CssRule 264
  Date 191
  Document 229
  Function 146
  Global 140
  History 233
  Image 300
  Location 232
  Math 182
  Navigator 233
  Number 176
  Screen 235
  Style 264
  StyleSheet 264
  XMLHttpRequest 308

Объект PHP 408
клонирование 414
создание 411
удаление 414
Окно:
свойства 256
создание нового 255
модальное 83
Операнд 113
Оператор 60, 113
?: 126
delete 216
do-while 134
for 129
for ... in ... 214
if 123
in 214
instanceof 216
switch 127
typeof 112
var 102
while 132
width 214
арифметический 114
декремента 115
инкремента 115
комментария 113
логический (булевский) 121
логическое сложение (ИЛИ) 122
логическое умножение (И) 122
отрицания (НЕ) 121
побитовый 135
присваивания 110, 117
сравнения 118
условия 122
цикла *См.* for, while, do-while
Оператор JavaScript
break 354
continue 354
do-while 356
for 352
цикла *См.* for, while, do-while
Оператор PHP
echo 325
foreach 390
new 411
арифметический 343
декремента 344
инкремента 344
комментария 342
конкатенации 361
логический (булевский) 347
логическое сложение (ИЛИ) 348
логическое умножение (И) 348
отрицания (НЕ) 348
побитовый 349
приведения к типу 330
присваивания 331, 345
сравнения 346
строковый 345
управления 349

П

Переменная 32, 100
глобальная 144
локальная 144
Переменная PHP:
глобальная 339, 397
имя 330
локальная 339
область действия 338
предопределенная 397
статическая 339
Поиск экстремума 190
Принцип короткой обработки 122, 348
Прототип 210

Р

Расширение PHP 545
Регистровая зависимость 60
Решение квадратного уравнения 185

С

Сеанс 439
Сессия *См.* сеанс
Слабый контроль типов 38
СУБД 486
Счетчик посещений:
простой 461
распространяемый 472
сложный 470

Т

Тег:
<pre> 367
Тип данных 32
Boolean 95
Null 95
Number 95
String 94
Undefined 95
примитивный 88
составной 90

Ф

Файл:
установочный 257
Функции:
встроенные 140
вставки и замены подстрок 161
обработки числовых массивов 175
удаления пробелов 163
Функция 43, 139, 399
eval() 140, 302
isNaN() 98
parseFloat() 96
parseInt() 96
typeOf() 142
unescape() 142
escape() 141
пользовательская 142
рекурсивное определение 145

Функция PHP:
__construct() 409
arsort() 389
asort() 389
closedir() 461
copy() 432
date() 377
define() 341
defined() 342
die() 324
empty() 340
eval() 419, 468
exit() 324, 417
explode() 366
fclose() 455
feof() 457
fgetcsv() 463
fgets() 456
fgetss() 458
file() 457
file_exists() 455
file_get_contents() 457
fopen() 454
fread() 457
function_exists() 406
fwrite() 459
header() 435, 448
imagecreatefromgif() 475
imagegif() 475
imagettftext() 475
implode() 366
include() 324, 338, 429
is_dir() 455, 459
is_file() 455
is_uploaded_file() 432
isset() 340, 426
krsort() 390
ksort() 390
mkdir() 459
mktime() 377
move_uploaded_file() 432
opendir() 460
print_r() 337, 386
printf() 367
range() 382
readdir() 461
rsort() 389
session_destroy() 440
session_id() 440
session_start() 439
setcookie() 437
sort() 388
sprintf() 367
sqlite_close() 501
sqlite_current() 502
sqlite_exec() 500
sqlite_fetch_array() 501
sqlite_next() 502
sqlite_open() 500
sqlite_prev() 502
sqlite_query() 500
sqlite_rewind() 502
sqlite_seek() 502
strip_tags() 458
time() 377
unlink() 456
unset() 333, 383, 414
var_dump() 336, 386
встроенная 406
даты-времени 377
диапазона 382
математическая 371
переменная 405
перемещения указателя массива 392
пользовательская 399
преобразования массива в строку 394
преобразования строк 361
преобразования строки в массив 394
преобразования чисел 374
проверки существования 340
проверки типов 340
работы с базой данных 487
работы с графикой 447
форматирования строк 367
форматирования чисел 376