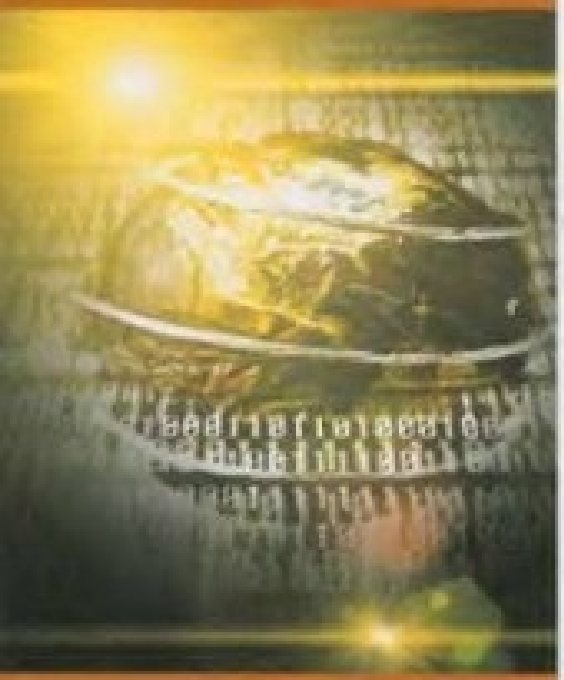


bhv

Андрей Попов

Windows Script Host для Windows 2000/XP

+Дискета

- Возможности Windows Script Host 5.6
- Вопросы безопасности при работе со сценариями
- Использование технологий WSC, ADO, ADSI и WMI
- Создание пользовательского интерфейса
- Обработка XML-файлов

МАСТЕР_СИСТЕМ

Annotation

В книге описывается сервер сценариев Windows Script Host (WSH) версии 5.6, который является одним из стандартных компонентов Windows XP, а также может быть установлен в более ранних версиях Windows. Приводятся сценарии на языках VBScript и JScript, иллюстрирующие применение стандартных объектов WSH 5.6, включая создание из сценариев дочерних процессов и запуск сценариев на удаленных компьютерах. В качестве более сложных примеров рассматриваются сценарии, работающие с базами данных, управляющие приложениями Microsoft Office, организующие графический пользовательский интерфейс, использующие технологии ADSI (Active Directory Service Interface) и WMI (Windows Management Instrumentation) для решения задач администрирования. Освещены вопросы практической работы с XML-файлами и создания COM-объектов. Особое внимание уделяется вопросам безопасности при работе со сценариями, приводятся конкретные примеры настроек политик безопасности. К книге прилагается дискета с исходными текстами большинства примеров.

- Введение
  -
  - Для кого предназначена эта книга
  - Структура книги
  - Принятые в книге соглашения
- Благодарности
- Глава 1
  -
  - Возможности технологии ActiveX
  - Нумерация версий WSH
  - Назначение и основные свойства WSH
  - Создание и запуск простейших сценариев JScript и VBScript
    -
    - Запуск сценария из командной строки в консольном режиме
    - Запуск сценария из командной строки в графическом режиме
    - Запуск сценария с помощью меню Пуск
    - Запуск сценария с помощью Проводника Windows (Windows Explorer)

- Установка и изменение свойств сценариев
  -
  - Свойства и параметры сценариев, выполняемых с помощью cscript.exe
  - Свойства и параметры сценариев, выполняемых с помощью wscript.exe
- Стандартные объекты WSH5.6
  -
  - Объект WScript
    -
    - Свойство Arguments
    - Свойства StdErr, StdIn, StdOut
    - Метод CreateObject
    - Метод ConnectObject
    - Метод DisconnectObject
    - Метод Echo
    - Метод Sleep
- Объекты-коллекции
  -
  - Объект WshArguments
  - Объект WshNamed
  - Объект WshUnnamed
  - Объект WshEnvironment
  - Объект WshSpecialFolders
- Работа с сетью и оболочкой Windows
  -
  - Объект WshNetwork
    -
    - Метод AddPrinterConnection
    - Метод AddWindowsPrinterConnection
    - Метод EnumNetworkDrives
    - Метод EnumPrinterConnections
    - Метод MapNetworkDrive
    - Метод RemoveNetworkDrive
    - Метод RemovePrinterConnection
    - Метод SetDefaultPrinter
  - Объект WshShell
    -
    - Метод AppActivate

        - Метод CreateShortcut
        - Метод Environment
        - Метод ExpandEnvironmentString
        - Метод LogEvent
        - Метод Popup
        - Метод RegDelete
        - Метод RegRead
        - Метод RegWrite
        - Метод Run
        - Метод SendKeys
    - Работа с ярлыками
        -
        - Объект WshShortcut
            -
            - Свойство Arguments
            - Свойство HotKey
            - Свойство IconLocation
            - Свойство WindowStyle
            - Свойство WorkingDirectory
        - Объект WshUrlShortcut
    - Запуск процессов на локальной и удаленной машине
        -
        - Объект WshScriptExec
            -
            - Свойство ProcessID
            - Свойство Status
            - Свойства StdOut, StdIn и StdErr
        - Объект WshController
        - Объект WshRemote
        - Объект WshRemoteError
- Глава 2
    -
    - Вывод на экран текстовых строк
        -
        - Метод Echo объекта WScript
        - Методы Write и WriteLine объекта WScript.StdOut
        - Функция MsgBox языка VBScript
        - Метод Popup объекта WshShell
    - Ввод строк текста

        -
        - Ввод строк в консольном режиме
        - Ввод строк в графическом режиме
    - Получение свойств WSH и запущенного сценария
    - Работа с параметрами командной строки сценария
    - Выход из сценария с определенным кодом завершения
    - Использование внешних объектов автоматизации (на Microsoft Word)
    - Запуск из сценариев внешних программ
        -
        - Запуск приложений Windows
        - Переключение между приложениями, имитация нажатий клавиш
        - Запуск независимых консольных приложений и команд DOS
        - Запуск дочерних консольных приложений и команд DOS, использование их входных и выходных потоков
    - Доступ к специальным папкам Windows XP
    - Создание ярлыков в специальных папках
    - Работа с системным реестром Windows
    - Работа с ресурсами локальной сети
        -
        - Определение имен рабочей станции, пользователя и домена
        - Получение списка подключенных сетевых дисков и принтеров
        - Подключение и отключение сетевых дисков и принтеров
    - Запуск сценариев на удаленных машинах. Контроль за ходом выполнения таких сценариев
- Глава 3
    -
    - Основные принципы XML
    - Схема WS XML
    - Элементы WS-файла
        -
        - Элементы <?xml?> и <![CDATA[]]>
        - Элемент <?job?>
        - Элемент <package>
        - Элемент <job>
        - Элемент <runtime>
        - Элемент <named>

        - Элемент <unnamed>
        - Элемент <description>
        - Элемент <example>
        - Элемент <resource>
        - Элемент <object>
        - Элемент <reference>
        - Элемент <script>
    - Примеры сценариев с разметкой XML
        -
        - Строгий режим обработки WS-файла
        - Несколько заданий в одном файле
        - Использование констант внешних объектов
        - Подключение внешних файлов
        - Два языка внутри одного задания (использование функции InputBox языка VBScript в сценариях JScript)
- Глава 4
    -
    - Шифрование сценариев
    - Цифровая подпись для сценариев WSH
        -
        - Использование цифровых сертификатов в Windows
        - Способы получения цифрового сертификата
        - Создание собственного сертификата
        - Управление сертификатами с помощью MMC
        - Добавление к сценарию цифровой подписи
        - Проверка цифровой подписи сценария
    - Политики безопасности для сценариев WSH
        -
        - Параметры реестра, влияющие на политику безопасности для WSH
        - Блокировка локальных и удаленных сценариев WSH. Пример административного шаблона
        - Три режима выполнения сценариев WSH
        - Протоколирование действий сценариев в журналах событий
        - Применение к сценариям WSH политики ограниченного использования программ
        - Блокировка сценария с заданным именем
        - Блокировка сценариев с заданной подписью
- Глава 5

- Выполнение основных операций с файловой системой
  - Объект FileSystemObject
    - Методы CopyFile и CopyFolder
    - Метод CreateTextFile
    - Методы DeleteFile и DeleteFolder
    - Метод DriveExists
    - Метод GetAbsolutePathName
    - Метод GetBaseName
    - Метод GetDrive
    - Метод GetParentFolderName
    - Метод GetSpecialFolder
    - Метод GetTempName
    - Методы MoveFile и MoveFolder
    - Метод OpenTextFile
  - Объект Drive
  - Коллекция Drives
  - Объект Folder
    - Метод Copy
    - Метод Delete
    - Метод Move
  - Коллекция Folders
  - Объект File
    - Метод Copy
    - Метод Delete
    - Метод Move
    - Метод OpenAsTextStream
  - Коллекция Files
  - Объект TextStream
- Примеры сценариев
  - Отчет об использовании дискового пространства
  - Удаление ненужных временных файлов с жесткого диска
  - Поиск файлов с использованием регулярных выражений
  - Перемещение файлов с ведением журнала действий

        - Разработка записной книжки в формате текстового файла
- Глава 6
    -
    - Записная книжка в формате XML
    - Просмотр XML-файла с помощью объектной модели Internet Explorer 4.0
        -
        - Описание объектной модели
        - Пример сценария
    - Использование XML DOM для просмотра и изменения XML-файла
        -
        - Описание модели XML DOM
        - Просмотр содержимого записной книжки
        - Добавление информации в записную книжку
        - Поиск и удаление записи из книжки
- Глава 7
    -
    - Многозадачный сценарий для работы с записной книжкой
        -
        - Обработка параметров командной строки
        - Организация диалога с помощью кнопочного меню
    - Однозадачный сценарий для работы с записной книжкой
        -
        - Использование Internet Explorer для создания диалоговых окон
        - Разработка HTML-формы для диалогового окна
        - Создание объекта для обмена данными между XML-файлом и формой
        - Вывод формы из сценария WSH
        - Обработка событий, генерируемых элементами управления формы
        - Окончательная доработка сценария IEPhoneBook.js
- Глава 8
    -
    - Объектные модели Microsoft Word и Excel
    - Вывод данных из записной книжки в документ Microsoft Word
        -
        - Вывод записей в виде обычного текста

        - Вывод записей в таблицу
    - Вывод данных из записной книжки в таблицу Microsoft Excel
- Глава 9
    -
    - Создание таблицы Phone.dbf в Microsoft Access
    - Настройка источника данных ODBC
    - Примеры сценариев
        -
        - Копирование данных из XML-файла в таблицу БД
        - Просмотр записей в таблице
        - Получение информации о полях и записях таблицы
        - Сортировка записей в таблице
        - Фильтрация записей в таблице
        - Перемещение в наборе записей
        - Доступ к БД без создания DSN
- Глава 10
    -
    - Технология Windows Script Components
    - Схема WSC XML
    - Элементы WSC-файла
        -
        - Элементы <?xml?> и <![CDATA[]]>
        - Элемент <package>
        - Элемент <component>
        - Элемент <registration>
        - Элемент <public>
        - Элемент <property>
        - Элемент <method>
        - Элемент <event>
        - Элементы <resource>, <object> и <reference>
        - Элемент <script>
    - Пример: COM-объект для архивирования файлов
        -
        - Создание макета файла DateArc.wsc с помощью Windows Script Component Wizard (JScript)
        - Доработка объекта-сценария DateArc.wsc (JScript)
        - Регистрация файла DateArc.wsc в качестве COM-объекта
        - Реализация объекта DateArc.wsc на VBScript
- Глава 11

-
- Использование службы каталогов Active Directory Service Interface (ADSI)
  -
  - Связывание с нужным объектом каталога
  - Список всех доступных доменов в локальной сети
  - Создание пользователя и группы на рабочей станции
  - Вывод информации о пользователе и смена его пароля
  - Удаление пользователя и группы на рабочей станции
  - Список всех групп на рабочей станции
  - Список всех пользователей в группе
  - Список всех групп, в которые входит пользователь
- Создание сценариев включения/выключения и входа/выхода
  -
  - Сценарии, выполняемые при загрузке операционной системы
  - Сценарии, выполняемые при завершении работы операционной системы
  - Сценарии входа для всех локальных пользователей
  - Сценарий выхода для всех локальных пользователей
  - Сценарий входа для одного пользователя
  - Примеры сценариев входа/выхода
    -
    - Подключение сетевых дисков и синхронизация времени при регистрации пользователей
    - Интерактивный выбор программ автозагрузки
    - Резервное копирование документов пользователя при окончании сеанса работы
- Вызов системных функций и стандартных диалоговых окон оболочки Windows
  -
  - Вызов модулей панели управления
    -
    - Запуск с помощью оболочки Windows
    - Запуск с помощью системных функций Windows
  - Открытие папки в Проводнике Windows
  - Вызов окна форматирования диска
  - Вызов окна запуска программ
  - Блокировка рабочей станции

        - Вызов окна выключения компьютера
    - Использование технологии Windows Management Instrumentation (WMI)
        -
        - Доступ к свойствам файла
        - Список всех запущенных процессов
        - Закрытие всех экземпляров запущенного приложения
- Заключение
- Приложение 1
    -
    - Строки кода и комментарии
    - Переменные
    - Типы данных
        -
        - Числа
        - Текстовые строки
        - Объекты
        - Логические данные
        - Null (пустой тип) и undefined (неопределенный тип)
    - Преобразование типов данных
    - Операторы
        -
        - Унарные операторы
        - Бинарные операторы
        - Операторы побитовых логических операций и сдвига
        - Операторы присваивания
        - Операторы отношения
        - Условные операторы
        - Операторы циклов
            -
            - Цикл for
            - Цикл for…in
            - Цикл while
            - Цикл do…while
        - Оператор break
        - Оператор continue
        - Прочие операторы
        - Обработка исключительных ситуаций
        - Порядок выполнения операторов

- Функции
  -
  - Встроенные функции
  - Функции пользователя
- Встроенные объекты (классы)
  -
  - Объект Array
  - Объект Date
  - Объект Enumerator
  - Объект Math
  - Объект String

- Приложение 2
  -
  - Строки кода и комментарии
  - Переменные
  - Подтипы данных
  - Константы
  - Массивы
  - Операторы
    -
    - Арифметические операторы
    - Операторы отношения и логические операторы
    - Условные операторы
    - Операторы циклов
      -
      - Цикл For…Next
      - Цикл For Each…Next
      - Цикл While…Wend
      - Цикл Do…Loop
    - Прочие операторы
    - Обработка исключительных ситуаций
  - Процедуры и функции
    -
    - Математические функции
    - Символьные функции
    - Функции для работы с массивами
    - Функции для работы с подтипами данных
    - Прочие функции
    - Функции и процедуры пользователя

- Приложение 3
  -
  - Создание и редактирование сценариев
    -
    - Редактор Primalscript
      -
      - Создание нового одиночного сценария
      - Открытие существующего одиночного сценария
      - Создание нового WS-файла
      - Открытие существующего WS-файла
      - Редактирование WS-файла на панели Workspace Nexus
      - Запуск одиночного сценария
      - Запуск задания из WS-файла
      - Подписывание сценариев
      - Возможности пользовательского интерфейса
    - Другие редакторы
  - Отладка сценариев в Microsoft Script Debugger
    -
    - Активизация отладчика
    - Команды отладчика
      -
      - Установка и удаление точек прерывания
      - Выполнение сценария
      - Просмотр стека вызовов
      - Просмотр и изменение значений переменных
- Приложение 4
- Приложение 5
- Источники информации
  - Список литературы
  - Ссылки на ресурсы Internet
    - Журналы и статьи
    - Сайты компании Microsoft
    - Зарубежные сайты
    - Российские сайты

Введение

Несколько лет назад компания Microsoft предложила в качестве инструмента разработки и выполнения специальных сценариев для операционной системы Windows сервер сценариев Windows Script Host (WSH), различные версии которого входят в стандартную поставку Windows 98/2000/XP (до этого на уровне операционной системы поддерживались только пришедшие из MS-DOS командные файлы). Сценарии WSH могут создаваться с помощью специализированных языков (например, Microsoft Visual Basic Script Edition (VBScript) или Microsoft JScript) и использовать любые объекты ActiveX, зарегистрированные в системе, что и определяет чрезвычайно мощные возможности таких сценариев.

Собственная объектная модель WSH позволяет из сценариев работать с файловой системой, системным реестром, специальными папками и ярлыками Windows, ресурсами локальной сети, а также запускать процессы и контролировать ход их выполнения.

Еще одна важная задача, которую решают сценарии WSH, — это организация взаимодействия с разработанными Microsoft современными ActiveX- технологиями:

□ ActiveX Data Object (ADO) — доступ к базам данных разных форматов;

□ Active Directory Service Interface (ADSI) — работа со службами каталогов (Active Directory для Windows 2000, Windows Directory Service для Windows NT 4.0 и т.д.);

□ Windows Management Instrumentation (WMI) — управление операционной системой Windows.

Надо сказать, что в то время как за рубежом сценариям WSH и используемым в них ActiveX-технологиям Microsoft посвящено много серьезных и объемных книг [10, 20–28], на русском языке подобных специальных книг пока немного (переводы книг Г. Борна [3] и Т. Экка [18], а также предыдущая книга автора [8]). В последнее время, правда, появилось довольно много публикаций о WSH, ADSI и WMI в журналах [1, 2, 6, 7, 9–16, 29]; краткое описание WSH и примеры сценариев приводятся в некоторых книгах, посвященных Windows 2000/XP (например, в [5] имеются примеры работы с системным реестром). Отметим также, что в последней версии WSH 5.6, которая является неотъемлемой частью

Windows XP, появились новые возможности (запуск сценариев на удаленных машинах, использование для сценариев политики ограниченного использования программ и т.д.), описания которых в русскоязычной литературе на момент написания книги автору не встречалось.

Итак, целью настоящей книги является решение следующих задач.

□ Детально описать объектные модели, использующиеся в WSH 5.6 и в технологии Windows Script Components (WSC), и дать примеры использования этих объектных моделей.

□ Привести практические примеры применения в сценариях WSH технологий ADO, ADSI и WMI, а также показать, каким образом из сценариев можно просматривать или изменять файлы в формате XML и управлять приложениями пакета Microsoft Office.

□ Обсудить проблемы безопасности, возникающие при работе с WSH, и описать способы решения этих проблем в Windows XP.

Для кого предназначена эта книга

Книга может быть полезна пользователям, программистам и администраторам Windows, которые желают получить систематизированную информацию о мощных возможностях ActiveX-сценариев WSH 5.6, а также об использовании в сценариях смежных технологий Microsoft (WMI, ADSI, ADO, WSC).

Информация, представленная в книге, позволяет применять ее и в качестве последовательного руководства по разработке сценариев Windows различной степени сложности (от простых JScript- или VBScript-сценариев, работающих без какого-либо вывода на экран, до многозадачных WS-файлов с XML-разметкой, предоставляющих пользователю полноценный графический интерфейс), и как справочник по объектам и XML-элементам, которые используются в WSH 5.6 (в том числе и для создания COM-объектов по технологии WSC).

Многие сценарии, приведенные в книге, могут применяться практически без изменений для решения конкретных практических задач по администрированию Windows (особенно это относится к примерам *главы 11*). Кроме этого, сведения, которые приведены в главе 4, помогут администраторам настроить политику безопасности для работы со сценариями WSH.

При изучении материала книги от читателя может потребоваться некоторое знакомство с языками JScript и VBScript (краткие справочники по этим языкам включены в приложения), а также понимание основ объектно-ориентированного программирования и COM-технологий.

Структура книги

В *главе 1* приводятся начальные сведения о назначении и возможностях WSH. Здесь показан процесс создания и запуска простейших сценариев на языках VBScript и JScript. Основная часть *главы 1* посвящена рассмотрению собственных объектов WSH. Подробно описаны свойства и методы этих объектов, с помощью которых в сценариях можно:

□ использовать внешние объекты ActiveX и ресурсы локальной сети;

□ выводить информацию в стандартный выходной поток или в окно Windows;

□ считывать данные из стандартного входного потока;

□ получать доступ к специальным папкам Windows и системному реестру;

□ создавать или изменять переменные среды и ярлыки Windows;

□ запускать процессы на локальной или удаленной рабочей станции.

В *главе 2* рассмотрены примеры сценариев, в которых используются стандартные объекты WSH (каждый из примеров реализован как на языке JScript, так и на языке VBScript). Среди прочих задач здесь освещены такие новые возможности WSH 5.6, как использование входных и выходных потоков дочерних приложений и контроль за ходом выполнения сценариев, запущенных на удаленном компьютере.

Глава 3 посвящена рассмотрению многозадачных сценариев WSH с разметкой XML. Здесь подробно описана объектная схема WS XML и приведены примеры сценариев, реализующих такие возможности этой схемы, как применение разных языков в одном задании или использование констант из библиотек типов внешних объектов.

В *главе 4* обсуждаются вопросы безопасности, возникающие при работе со сценариями WSH (незащищенность исходного кода сценария, простота написания и распространения сценариев-вирусов и т.д.). В этой главе описано, каким образом можно создавать зашифрованные файлы сценариев и добавлять к сценариям цифровые подписи для указания автора сценария. Кроме этого, рассмотрен процесс организации политик безопасности для сценариев WSH, позволяющих, например, запретить выполнение неподписанных сценариев любого типа или вообще заблокировать WSH для определенных пользователей.

В *главе 5* показано, каким образом можно получить полный доступ к файловой системе компьютера с помощью объекта `FileSystemObject`.

Здесь объяснено, какие свойства и методы нужно использовать для выполнения тех или иных файловых операций, а также приведено подробное описание этих свойств и методов. Также в этой главе приведены несколько полезных сценариев, позволяющих, например, производить поиск нужных файлов с использованием регулярных выражений или перемещать файлы из одного каталога в другой с ведением журнала действий.

В *главе 6* рассматриваются практические вопросы обработки из сценариев файлов в формате XML. В качестве примера здесь используется записная книжка, содержимое которой хранится в XML-файле. Приведенные в этой главе сценарии позволяют с помощью объектной модели XML DOM просматривать и изменять содержимое записной книжки.

В *главе 7* обсуждаются вопросы организации из сценариев диалога с пользователем, начиная от простого использования параметров командной строки и заканчивая созданием полноценного пользовательского интерфейса с помощью HTML-форм. Все рассматриваемые здесь сценарии предназначены для работы с записной книжкой в формате XML.

В *главе 8* показано, как из сценариев WSH организовать управление двумя самыми распространенными программами из пакета Microsoft Office: Word и Excel. Приведенные здесь сценарии позволяют организовать вывод информации из записной книжки в документ Word или таблицу Excel.

В *главе 9* рассматривается технология ADO, позволяющая работать из сценариев с базами данных различных форматов. Приведены примеры сценариев, с помощью которых можно просматривать и изменять записи в таблице DBF, содержимое которой переносится (также с помощью сценария) из записной книжки в XML-файле.

В *главе 10* обсуждается технология Windows Script Components (WSC), с помощью которой можно WSH-сценарии "упаковывать" в COM-объекты. Здесь описана объектная схема WSC XML и подробно рассмотрен процесс создания объекта-сценария, предназначенного для архивирования содержимого каталога в файлы с определенным именем.

В *главе 11* рассмотрены примеры сценариев, которые позволяют автоматизировать некоторые повседневные задачи администратора компьютерной системы (например, работу "с учетными записями пользователей, создание сценариев входа/выхода или настройку параметров операционной системы на нескольких рабочих станциях в сети). Для этого используются еще две мощные технологии Microsoft,

ADSI и WMI, возможности и основные принципы которых также описаны в этой главе.

В *приложениях 1–4* приведены справочники по языкам JScript и VBScript, дано описание специальных редакторов и отладчиков для сценариев (в частности, подробно рассмотрен мощный редактор Primalscript), а также указан список ошибок (с пояснениями), которые могут возникать при выполнении сценариев WSH 5.6.

Примеры сценариев, которые приведены в книге, можно набирать вручную или загружать с прилагаемой дискеты. Структура дискеты и инструкции по ее использованию приведены в *приложении 5*.

Принятые в книге соглашения

При описании операторов, функций и методов объектов мы использовали стандартные соглашения. Названия параметров и аргументов набраны курсивом, необязательные параметры заключены в квадратные скобки [], например:

```
CreateObject(strProgID [,strPrefix])
```

Если при вызове команды может быть указан только один параметр/ аргумент из нескольких возможных, то такие параметры/аргументы разделены знаком |, например:

```
<?job debug="true|false"?>
```

Благодарности

Я благодарю всех сотрудников издательства "БХВ-Петербург", в особенности Анатолия Николаевича Адаменко и Петра Анатольевича Науменко за их вклад в редактирование плана и текста книги, а также других людей, помогавших мне во время работы над книгой.

Особую признательность я хочу выразить Эдуарду Батаршину и Евгению Шикину за их ценные консультации и предложения.

Спасибо моей жене Татьяне и другим родственникам за их терпение и поддержку во всем.

Глава 1
Первое знакомство с Windows Script Host

В ранних версиях Windows стандартным средством для автоматизации однотипных повторяющихся задач служили командные (пакетные) файлы и утилиты пакета Resource Kit для соответствующей версии. Однако даже с помощью имеющегося в Windows NT/2000/XP усовершенствованного командного интерпретатора cmd.exe трудно написать какую-либо сложную программу-сценарий (script): отсутствует полноценная интерактивность, нельзя напрямую работать с рабочим столом Windows и системным реестром и т.д.

Для исправления этой ситуации компанией Microsoft был разработан сервер сценариев WSH — Windows Script Host, с помощью которого можно выполнять сценарии, написанные, в принципе, на любом языке (при условии, что для этого языка установлен соответствующий модуль (scripting engine), поддерживающий технологию *ActiveX Scripting*). В качестве стандартных языков поддерживаются Visual Basic Script Edition (VBScript) и JScript.

Вообще говоря, принцип работы сценариев, поддерживаемых WSH, состоит в использовании *объектов ActiveX*, поэтому вначале мы очень кратко опишем возможности самой технологии ActiveX компании Microsoft.

Возможности технологии ActiveX

Напомним, что в Windows с самого начала для обеспечения обмена данными между приложениями была разработана технология *связывания и внедрения объектов* (OLE, Object Linking and Embedding). Вначале технология OLE использовалась для создания составных документов, а затем для решения более общей задачи — предоставления приложениями друг другу собственных функций (служб) и правильного использования этих функций. Технология, позволяющая одному приложению *(клиенту автоматизации)*, вызывать функции другого приложения *(сервера автоматизации)*, была названа OLE Automation. В основе OLE и OLE Automation лежит разработанная Microsoft базовая "компонентная" технология COM (Component Object Model).

В общих словах, компонентное программное обеспечение — это способ разработки программ, при котором используются технологии создания программных модулей, подобные технологиям, применяемым для разработки аппаратных средств. Сложные элементные схемы собираются из стандартизированных микросхем, которые имеют четко определенные документированные функции. Разработчик может эффективно пользоваться такими микросхемами, не задумываясь об их внутренней структуре. В программных компонентах, написанных на каком-либо языке программирования, детали реализации используемых алгоритмов также скрыты внутри компонента (объекта), а на поверхности находятся общедоступные *интерфейсы*, которыми могут пользоваться и другие приложения, написанные на том же или другом языке.

В настоящее время, по заявлению Microsoft, термин OLE используется только по историческим причинам. Вместо него Microsoft с 1996 года применяет новый термин — *ActiveX*, первоначально обозначавший WWW (World Wide Web) компоненты (объекты), созданные на базе технологии COM.

Технология ActiveX до последнего времени являлась ключевой в продуктах Microsoft. Наиболее полное воплощение она нашла в программах Microsoft Office, Internet Explorer, IIS (Internet Information Service). В эти продукты для управления соответствующими объектами автоматизации были встроены интерпретаторы специальных языков сценариев: VBScript (используется в Microsoft Office, Internet Explorer, IIS) и JScript (используется в Internet Explorer, IIS). Однако непосредственно в

операционной системе, вне этих продуктов, выполнять сценарии, написанные на VBScript или JScript, было нельзя.

Сервер сценариев WSH является мощным инструментом, предоставляющим единый интерфейс (объектную модель) для специализированных языков (VBScript, JScript, PerlScript, REXX, TCL, Python и т.п.), которые, в свою очередь, позволяют использовать любые внешние объекты ActiveX. С помощью WSH сценарии могут быть выполнены непосредственно в операционной системе Windows, без встраивания в HTML-страницы.

Нумерация версий WSH

Скажем несколько слов относительно нумерации версий WSH. Самая первая версия WSH, входившая в качестве стандартного компонента в Windows 98, имела номер 1.0. С другой стороны, эта версия опиралась на языки сценариев JScript и VBScript версии 5.0, которые поддерживались Internet Explorer 5.0, поэтому можно к самой первой версии WSH относиться и как к WSH 5.0. Следующей версии WSH, входившей в поставку Windows 2000, был присвоен номер 2.0; в этой версии использовались языки JScript и VBScript версии 5.1. Наконец, номер последней версии WSH, являющейся неотъемлемой частью Windows XP, компания Microsoft решила все же привязать к номеру версии языков JScript и VBScript — таким образом появился сервер сценариев WSH 5.6.

Отметим также, что даже сама аббревиатура "WSH" сначала расшифровывалась Microsoft как "Windows Scripting Host", а затем было принято название "Windows Script Host".

Замечание

Несмотря на то, что WSH 5.6 является стандартной частью Windows XP, эту версию можно также установить и использовать во всех предыдущих 32-разрядных версиях Windows 95/98/ME/NT/2000. Для этого необходимо скачать инсталляционный файл для WSH 5.6 с сервера Microsoft (http://msdn.microsoft.com/scripting); там же можно найти и документацию по WSH 5.6.

Назначение и основные свойства WSH

WSH предъявляет минимальные требования к объему оперативной памяти и является очень удобным инструментом для автоматизации повседневных задач пользователей и администраторов операционной системы Windows. Используя сценарии WSH, можно непосредственно работать с файловой системой компьютера, а также управлять работой других приложений (серверов автоматизации). При этом возможности сценариев ограничены только средствами, которые предоставляют доступные серверы автоматизации.

Перечислим только наиболее очевидные задачи, для автоматизации которых прекрасно подходят сценарии WSH.

□ Организация резервного копирования на сетевой сервер файлов с локальной машины, которые отбираются по какому-либо критерию.

□ Быстрое изменение конфигурации рабочего стола Windows в зависимости от задач, выполняемых пользователем.

□ Автоматический запуск программ Microsoft Office, создание там сложных составных документов, распечатка этих документов и закрытие приложений.

□ Управление работой приложений, не являющихся серверами автоматизации, с помощью посылки в эти приложения нажатий клавиш.

□ Подключение и отключение сетевых ресурсов (дисков и принтеров).

□ Создание сложных сценариев регистрации для пользователей.

□ Выполнение задач администрирования локальной сети (например, добавление или удаление пользователей).

Создание и запуск простейших сценариев JScript и VBScript

Простейший WSH-сценарий, написанный на языке JScript или VBScript, — это обычный текстовый файл с расширением js или vbs соответственно, создать его можно в любом текстовом редакторе, способном сохранять документы в формате "Только текст".

Замечание

Специальные мощные программы, позволяющие создавать и отлаживать сценарии, описаны в *приложении 3.*

Размер сценария может изменяться от одной до тысяч строк, предельный размер ограничивается лишь максимальным размером файла в соответствующей файловой системе.

В качестве первого примера создадим JScript-сценарий, выводящий на экран диалоговое окно с надписью "Привет!" (рис. 1.1).

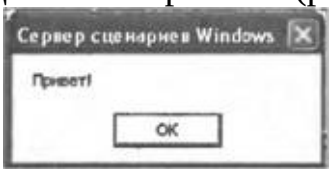


Рис. 1.1. Простое диалоговое окно

Для вывода такого окна достаточно с помощью, например, стандартного Блокнота Windows (notepad.exe) создать файл First.js, содержащий всего одну строку:

```
WScript.Echo("Привет!");
```

Тот же самый сценарий на языке VBScript, естественно, отличается синтаксисом и выглядит следующим образом:

```
WScript.Echo "Привет!"
```

Несмотря на то, что для работы наших сценариев достаточно всего одной строки, желательно сразу приучить себя к добавлению в начало файла информации о находящемся в нем сценарии: имя файла, используемый язык, краткое описание выполняемых действий. На языке JScript такая информация, оформленная в вице комментариев, может

выглядеть следующим образом:

```
/*****************************************************************
/*                                                          Имя:
First.js                                                   */
/*                                                         Язык:
JScript                                                  */
/*          Описание:          ВЫВОД          на          экран
приветствия                                  */
/*****************************************************************
```

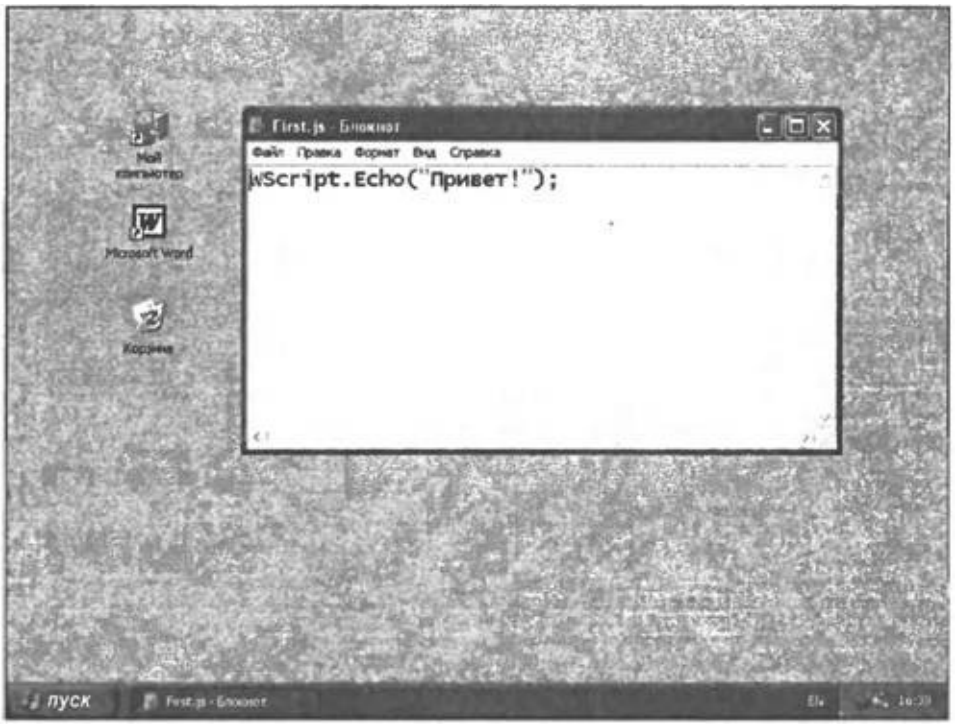


Рис. 1.2. Сценарий First.js

На языке VBScript то же самое выглядит следующим образом:

```
'*****************************************************************
' Имя: First.vbs
' Язык: VBScript
' Описание: Вывод на экран приветствия
'*****************************************************************
```

Для запуска сценариев WSH существует несколько способов.

Запуск сценария из командной строки в консольном режиме

Можно выполнить сценарий из командной строки с помощью консольной версии WSH cscript.exe. Например, чтобы запустить сценарий, записанный в файле C:\Script\First.js, нужно загрузить командное окно и

выполнить в нем команду

```
cscript C:\Script\First.js
```

В результате выполнения этого сценария в командное окно выведется строка "Привет!" (рис. 1.3).

Рис. 1.3. Результат выполнения First.js в консольном режиме (cscript.exe)

Запуск сценария из командной строки в графическом режиме

Сценарий можно выполнить из командной строки с помощью (оконной) графической версии WSH wscript.exe. Для нашего примера в этом случае нужно выполнить команду

```
wscript C:\Script\First.js
```

Тогда в результате выполнения сценария на экране появится нужное нам диалоговое окно (рис. 1.4).

Таким образом, мы видим, что при запуске сценария в консольном режиме вывод текстовой информации происходит в стандартный выходной поток (на экран), при запуске в графическом режиме — в диалоговое окно.

Рис. 1.4. Результат выполнения First.js в графическом режиме (wscript.exe)

Запуск сценария с помощью меню *Пуск*

Для запуска сценария с помощью пункта **Выполнить** (Run) меню **Пуск** (Start), достаточно написать полное имя этого сценария в поле **Открыть** (Open) (рис. 1.5).

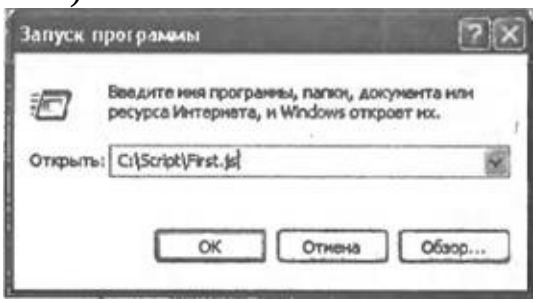


Рис. 1.5. Запуск сценария из меню Пуск в Windows XP

В этом случае по умолчанию сценарий будет выполнен с помощью wscript.exe, т.е. вывод информации будет вестись в графическое диалоговое окно.

Запуск сценария с помощью Проводника Windows (Windows Explorer)

Самым простым является запуск сценария в окнах Проводника Windows или на рабочем столе — достаточно просто выполнить двойной щелчок мышью на имени файла со сценарием или на его значке (рис. 1.6).

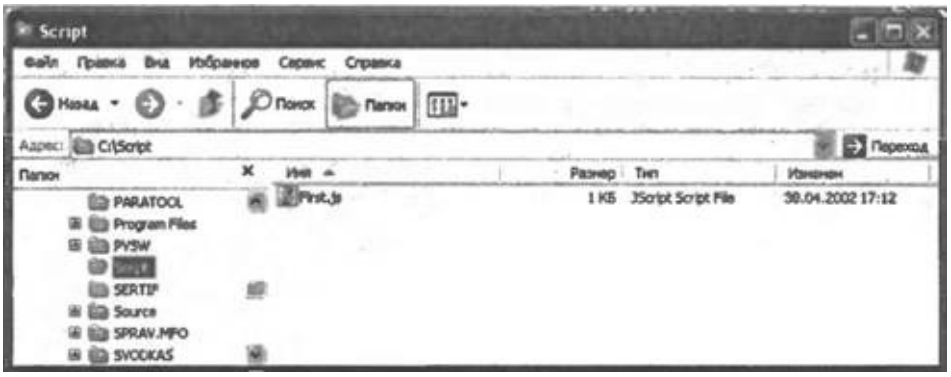


Рис. 1.6. Запуск сценария с помощью Проводника Windows

При этом, как и в случае запуска с помощью меню **Пуск** (Start), сценарий по умолчанию выполняется с помощью wscript.exe.

Установка и изменение свойств сценариев

В случае необходимости для сценариев можно задавать различные параметры, влияющие на ход их выполнения. Для консольной (cscript.exe) и графической (wscript.exe) версий сервера сценариев эти параметры задаются по-разному.

Свойства и параметры сценариев, выполняемых с помощью cscript.exe

В этом случае исполнение сценария контролируется с помощью параметров командной строки для cscript.exe (табл. 1.1), которые включают или отключают различные опции WSH (все эти параметры начинаются с символов //).

***Таблица** 1.1. Параметры командной строки cscript.exe*

Параметр	Описание
`//I`	Выключает пакетный режим (по умолчанию). При этом на экран будут выводиться все сообщения об ошибках в сценарии
`//B`	Включает пакетный режим. При этом на экран не будут выводиться никакие сообщения
`//T:nn`	Задает тайм-аут в секундах, т.е. сценарий будет выполняться nn секунд, после чего процесс прервется. По умолчанию время выполнения не ограничено
`//Logo`	Выводит (по умолчанию) перед выполнением сценария информацию о версии и разработчике WSH
`//Nologo`	Подавляет вывод информации о версии и разработчике WSH
`//H:CScript` или `//H:Wscript`	Делает cscript.exe или wscript.exe приложением для запуска сценариев по умолчанию. Если эти параметры не указаны, то по умолчанию подразумевается wscript.exe
`//S`	Сохраняет установки командной строки для текущего пользователя
`//?`	Выводит встроенную подсказку для параметров командной строки

`//E:engine`	Выполняет сценарий с помощью модуля, заданного параметром engine
`//D`	Включает отладчик
`//X`	Выполняет программу в отладчике
`//Job:<JobID>`	Запускает задание с индексом `JobID` из многозадачного WS-файла (структура WS-файлов будет описана в *главе 3)*
`//U`	Позволяет использовать при перенаправлении ввода/вывода с консоли кодировку Unicode

Например, команда

```
cscript //Nologo C:\Script\First.js
```

запустит сценарий First.js без информации о версии WSH (рис. 1.7).

Сценарий можно запускать с параметрами командной строки, которые указываются после имени этого сценария (процедура обработки таких параметров будет описана ниже, при рассмотрении объектов `WshArguments`, `WshNamed` и `WshUnnamed`). Например, команда

```
cscript //B C:\Script\First.js /a /b
```

запустит сценарий First.js в пакетном режиме, при этом `/a` и `/b` будут являться параметрами этого сценария, а `//B` — параметром приложения cscript.exe.

Рис. 1.7. Результат выполнения First.js в консольном режиме без информации о версии WSH

Свойства и параметры сценариев, выполняемых с помощью wscript.exe

При использовании для запуска модуля wscript.exe свойства сценария можно устанавливать с помощью вкладки **Сценарий** (Script) диалогового окна, задающего свойства файла в Windows (рис. 1.8).

После задания свойств сценария автоматически создается файл с именем этого сценария и расширением wsh, который имеет структуру наподобие ini-файла, например:

```
[ScriptFile]
Path=C:\Script\First.js
[Options]
Timeout=0
DisplayLogo=1
```

Если дважды щелкнуть в Проводнике Windows по wsh-файлу или запустить такой файл из командной строки, то соответствующий сервер сценариев (wscript.exe или cscript.exe) запустит сценарий, которому соответствует wsh- файл, с заданными **в** секции Options параметрами.

Замечание

Если wsh-файл не запускается, нужно проверить наличие на диске файла, указанного в секции ScriptFile.

Рис. 1.8. Установка свойств сценария First.js

При запуске сценариев с помощью wscript.exe для задания параметров

командной строки сценария можно использовать технологию drag-and-drop — если выделить в Проводнике Windows несколько файлов и перетащить их на ярлык сценария, то этот сценарий запустится, а имена выделенных файлов передадутся ему в качестве параметров.

Стандартные объекты WSH5.6

Перейдем теперь к описанию собственной объектной модели WSH 5.6. С помощью внутренних объектов этой версии WSH из сценариев можно выполнять следующие основные задачи:

□ выводить информацию в стандартный выходной поток (на экран) или в диалоговое окно Windows;

□ читать данные из стандартного входного потока (т. е. вводить данные с клавиатуры) или использовать информацию, выводимую другой командой;

□ использовать свойства и методы внешних объектов, а также обрабатывать события, которые генерируются этими объектами;

□ запускать новые независимые процессы или активизировать уже имеющиеся;

□ запускать дочерние процессы с возможностью контроля их состояния и доступа к их стандартным входным и выходным потокам;

□ работать с локальной сетью: определять имя зарегистрировавшегося пользователя, подключать сетевые диски и принтеры;

□ просматривать и изменять переменные среды;

□ получать доступ к специальным папкам Windows;

□ создавать ярлыки Windows;

□ работать с системным реестром.

В WSH 5.6 входят перечисленные ниже объекты:

□ `WScript`. Это главный объект WSH, который служит для создания других объектов или связи с ними, содержит сведения о сервере сценариев, а также позволяет вводить данные с клавиатуры и выводить информацию на экран или в окно Windows.

□ `WshArguments`. Обеспечивает доступ ко всем параметрам командной строки запущенного сценария или ярлыка Windows.

□ `WshNamed`. Обеспечивает доступ к именным параметрам командной строки запущенного сценария.

□ `WshUnnamed`. Обеспечивает доступ к безымянным параметрам командной строки запущенного сценария.

□ `WshShell`. Позволяет запускать независимые процессы, создавать ярлыки, работать с переменными среды, системным реестром и специальными папками Windows.

□ `WshSpecialFolders`. Обеспечивает доступ к специальным папкам

Windows.

□ `WshShortcut`. Позволяет работать с ярлыками Windows.

□ `WshUrlShortcut`. Предназначен для работы с ярлыками сетевых ресурсов.

□ `WshEnvironment`. Предназначен для просмотра, изменения и удаления переменных среды.

□ `WshNetwork`. Используется при работе с локальной сетью: содержит сетевую информацию для локального компьютера, позволяет подключать сетевые диски и принтеры.

□ `WshScriptExec`. Позволяет запускать консольные приложения в качестве дочерних процессов, обеспечивает контроль состояния этих приложений и доступ к их стандартным входным и выходным потокам.

□ `WshController`. Позволяет запускать сценарии на удаленных машинах.

□ `WshRemote`. Позволяет управлять сценарием, запущенным на удаленной машине.

□ `WshRemoteError`. Используется для получения информации об ошибке, возникшей в результате выполнения сценария, запущенного на удаленной машине.

Кроме этого, имеется объект FileSystemObject, обеспечивающий доступ к файловой системе компьютера (этот объект будет подробно описан в *главе 5)*.

Перейдем теперь к рассмотрению свойств и методов внутренних объектов WSH.

Замечание

Примеры всех сценариев, приведенных далее в этой главе, написаны на JScript. Так как эти сценарии только иллюстрируют свойства и методы WSH и не используют особенностей, характерных только для JScript, перевод их на VBScript не представляет никакого труда.

Объект *WScript*

Свойства объекта `WScript` позволяют получить полный путь к использующемуся серверу сценариев (wscript.exe или cscript.exe),

параметры командной строки, с которыми запущен сценарий, режим его работы (интерактивный или пакетный). Кроме этого, с помощью свойств объекта `WScript` можно выводить информацию в стандартный выходной поток и читать данные из стандартного входного потока. Также `WScript` предоставляет методы для работы внутри сценария с объектами автоматизации и вывода информации на экран (в текстовом режиме) или в окно Windows.

Отметим, что в сценарии WSH объект `WScript` можно использовать сразу, без какого-либо предварительного описания или создания, т. к. его экземпляр создается сервером сценариев автоматически. Для использования же всех остальных объектов нужно применять либо метод `CreateObject`, либо определенное свойство другого объекта.

Свойства объекта `WScript` представлены в табл. 1.2.

Таблица 1.2. *Свойства объекта* `WScript`

Свойство	**Описание**
`Application`	Предоставляет интерфейс `IDispatch` для объекта `WScript`
`Arguments`	Содержит указатель на коллекцию `WshArguments`, в которой находятся параметры командной строки для исполняемого сценария
`FullName`	Содержит полный путь к исполняемому файлу сервера сценариев (в Windows XP обычно это C:\WINDOWS\SYSTEM32\CSCRIPT.EXE или C:\WINDOWS\SYSTEM32\WSCRIPT.EXE)
`Name`	Содержит название объекта `Wscript` (Windows Script Host)
`Path`	Содержит путь к каталогу, в котором находится cscript.exe или wscript.exe (в Windows XP обычно это C:\WINDOWS\SYSTEM32)
`ScriptFullName`	Содержит полный путь к запущенному сценарию
`ScriptName`	Содержит имя запущенного сценария
`StdErr`	Позволяет запущенному сценарию записывать сообщения в стандартный поток для ошибок
`StdIn`	Позволяет запущенному сценарию читать информацию из стандартного входного потока
`StdOut`	Позволяет запущенному сценарию записывать информацию в стандартный выходной поток
`Version`	Содержит версию WSH

Опишем более подробно те свойства объекта WScript, которые требуют дополнительных пояснений.

Свойство *Arguments*

В следующем примере (листинг 1.1) с помощью цикла for на экран выводятся все параметры командной строки, с которыми был запущен сценарий.

Листинг 1.1. Вывод на экран всех параметров сценария

```
    /*****************************************************************
    /*                                                          Имя:
ShowArgs.js                                                  */
    /*                                                         Язык:
JScript                                                     */
    /*   Описание:   Вывод   на   экран   параметров   запущенного
сценария        */
    /*****************************************************************
    var i, objArgs;
    objArgs = WScript.Arguments; //Создаем объект WshArguments
    for (i=0; i<=objArgs.Count()-1; i++)
     WScript.Echo(objArgs(i)); //Выводим на экран i-й аргумент
    /*************                                              Конец
*******************************************/
```

Другие примеры работы с аргументами командной строки приведены в листингах 1.4, 1.5, 2.22 и 2.23.

Свойства *StdErr, StdIn, StdOut*

Доступ к стандартным входным и выходным потокам с помощью свойств `StdIn`, `StdOut` и `StdErr` можно получить только в том случае, если сценарий запускался в консольном режиме с помощью cscript.exe. Если сценарий был запущен с помощью wscript.exe, то при попытке обратиться к этим свойствам возникнет ошибка "`Invalid Handle`" (рис. 1.9).

Рис. 1.9. Ошибка при обращении к `StdIn` в графическом режиме

Работать с потоками `StdOut` и `StdErr` можно с помощью методов `Write`, `WriteLine`, `WriteBlankLines`, а с потоком `StdIn` — с помощью методов `Read`, `ReadLine`, `ReadAll`, `Skip`, `SkipLine`. Эти методы кратко описаны в табл. 1.3.

***Таблица** 1.3. Методы для работы с потоками*

Метод	**Описание**
`Read(n)`	Считывает из потока `StdIn` заданное параметром n число символов и возвращает полученную строку
`ReadAll()`	Читает символы из потока `StdIn` до тех пор, пока не встретится символ конца файла ASCII 26 (<Ctrl>+<Z>), и возвращает полученную строку
`ReadLine()`	Возвращает строку, считанную из потока `StdIn`
`Skip(n)`	Пропускает при чтении из потока `StdIn` заданное параметром n число символов
`SkipLine()`	Пропускает целую строку при чтении из потока `StdIn`
`Write(`*`string`*`)`	Записывает в поток `StdOut` или `StdErr` строку *`string`* (без символа конца строки)
`WriteBlankLines(n)`	Записывает в поток `StdOut` или `StdErr` заданное параметром n число пустых строк
`WriteLine`*`(string)`*	Записывает в поток `StdOut` или `StdErr` строку *`string`* (вместе с символом конца строки)

Напомним, что операционная система Windows поддерживает механизм *конвейеризации* (символ "|" в командной строке). Этот механизм делает возможным передачу данных от одной программы к другой. Таким образом, используя стандартные входные и выходные потоки, можно из сценария обрабатывать строки вывода другого приложения или перенаправлять выводимые сценарием данные на вход программ-фильтров

(`FIND` или `SORT`). Например, следующая команда будет сортировать строки вывода сценария example.js и выводить их в файл sort.txt:

```
cscript //Nologo example.js | sort > sort.txt
```

Опция //Nologo здесь нужна для того, чтобы в файл sort.txt не попадали строки с информацией о разработчике и номере версии WSH.

Кроме этого, с помощью методов, работающих с входным потоком StdIn, можно организовывать диалог с пользователем, т. е. создавать интерактивные сценарии. Пример такого сценария представлен в листинге 1.2.

Листинг 1.2. Пример интерактивного сценария

```
    /*****************************************************************
    /*                                                          Имя:
Interact.js                                                  */
    /*                                                         Язык:
JScript                                                     */
    /*     Описание:     Ввод/вывод     строк     в     консольном
режиме                  */
    /*****************************************************************
    var s;
    //Выводим строку на экран
    WScript.StdOut.Write("Введите число: ");
    //Считываем строку
    s = WScript.StdIn.ReadLine();
    //Выводим строку на экран
    WScript.StdOut.WriteLine("Вы ввели число " + s);
    /*************                                            Конец
*********************************************/
```

Объект `WScript` имеет несколько методов, которые описаны в табл. 1.4.

Таблица 1.4. *Методы объекта `WScript`*

Метод	**Описание**
`CreateObject(`*`strProgID [, strPrefix]`*`)`	Создает объект, заданный параметром *`strProgID`*
`ConnectObject(`*`strObject, strPrefix`*`)`	Устанавливает соединение с объектом *`strObject`*, позволяющее писать функции-обработчики его событий (имена этих функций должны начинаться с префикса *`strPrefix`*)
`DisconnectObject(`*`obj`*`)`	Отсоединяет объект *`obj`*, связь с которым была

	предварительно установлена в сценарии
`Echo(`*`[Arg1] [, Arg2] [, …]`*`)`	Выводит текстовую информацию на консоль или в диалоговое окно
`GetObject(`*`strPathname [, strProgID], [strPrefix]`*`)`	Активизирует объект автоматизации, определяемый заданным файлом (параметр *`strPathName`*), или объект, заданный параметром *`strProgID`*
`Quit(`*`[intErrorCode]`*`)`	Прерывает выполнение сценария с заданным параметром *`intErrorCode`* кодом выхода. Если параметр *`intErrorCode`* не задан, то объект `WScript` установит код выхода равным нулю
`Sleep(`*`intTime`*`)`	Приостанавливает выполнения сценария (переводит его в неактивное состояние) на заданное параметром *`intTime`* число миллисекунд

Приведем дополнительные пояснения и примеры использования для методов, приведенных в табл. 1.4.

Метод *CreateObject*

Строковый параметр *`strProgID`*, указываемый в методе `CreateObject`**,** называется *программным идентификатором* объекта (Programmic Identifier, ProgID).

Если указан необязательный параметр *`strPrefix`*, то после создания объекта в сценарии можно обрабатывать события, возникающие в этом объекте (естественно, если объект предоставляет интерфейсы для связи с этими событиями). Когда объект сообщает о возникновении определенного события, сервер сценариев вызывает функцию, имя которой состоит из префикса *`strPrefix`* и имени этого события. Например, если в качестве *`strPrefix`* указано "`MYOBJ_`", а объект сообщает о возникновении события "`OnBegin`", то будет запущена функция "`MYOBJ_OnBegin`", которая должна быть описана в сценарии.

В следующем примере метод `CreateObject` используется для создания объекта `WshNetwork`:

```
var WshNetwork = WScript.Createobject("WScript.Network");
```

Отметим, что объекты автоматизации из сценариев можно создавать и без помощи WSH. В JScript для этого используется объект `ActiveXObject`,

например:

```
var WshNetwork = new ActiveXObject("WScript.Network");
```

В VBscript для создания объектов может использоваться специальная функция `CreateObject`, например:

```
Set WshNetwork = CreateObject("WScript.Network")
```

Однако организовать в сценарии обработку событий создаваемого объекта можно только при использовании метода `WScript.CreateObject`.

Метод *ConnectObject*

Объект, соединение с которым осуществляется с помощью метода `ConnectObject`, должен предоставлять интерфейс к своим событиям.

В следующем примере в переменной `MyObject` создается абстрактный объект "`SomeObject`", затем из сценария вызывается метод `SomeMetod` этого объекта. После этого устанавливается связь с переменной `MyObject` и задается префикс "`MyEvent`" для процедур обработки события этого объекта. Если в объекте возникнет событие с именем "`Event`", то будет вызвана функция `MyEvent_Event`. Метод `DisconnectObject` объекта `WScript` производит отсоединение объекта `MyObject`.

```
var MyObject = WScript.CreateObject("SomeObject");
MyObject.SomeMethod();
WScript.ConnectObject(MyObject, "MyEvent");

function MyEvent_Event(strName) {
 WScript.Echo(strName);
}
WScript.DisconnectObject(MyObject);
```

Метод *DisconnectObject*

Если соединения с объектом *`obj`* не было установлено, то метод `DisconnectObject(`*`obj`*`)` не будет производить никаких действий. Пример применения `DisconnectObject` был приведен выше.

Метод *Echo*

Параметры *`Arg1`*, *`Arg2`* задают аргументы для вывода. Если сценарий был запущен с помощью wscript.exe, то метод `Echo` направляет вывод в диалоговое окно, если же для выполнения сценария применяется

cscript.exe, то вывод будет направлен на экран (консоль). Каждый из аргументов при выводе будет разделен пробелом. В случае использования cscript.exe вывод всех аргументов будет завершен символом новой строки. Если в методе Echo не задан ни один аргумент, то будет напечатана пустая строка.

Например, после выполнения сценария EchoExample.js (листинг 1.3) с помощью cscript.exe на экран будут выведены пустая строка, три числа и строка текста (рис. 1.10).

Листинг 1.3. Сценарий EchoExample.js

```
    /*****************************************************************
    /*                                                          Имя:
EchoExample.js                                                  */
    /*                                                         Язык:
JScript                                                        */
    /*           Описание:           Использование           метода
WScript.Echo                        */
    /*****************************************************************
    WScript.Echo();  //Выводим пустую строку
    WScript.Echo(1,2,3); //Выводим числа
    WScript.Echo("Привет!"); //Выводим строку
    /*************                                             Конец
*******************************************/
```

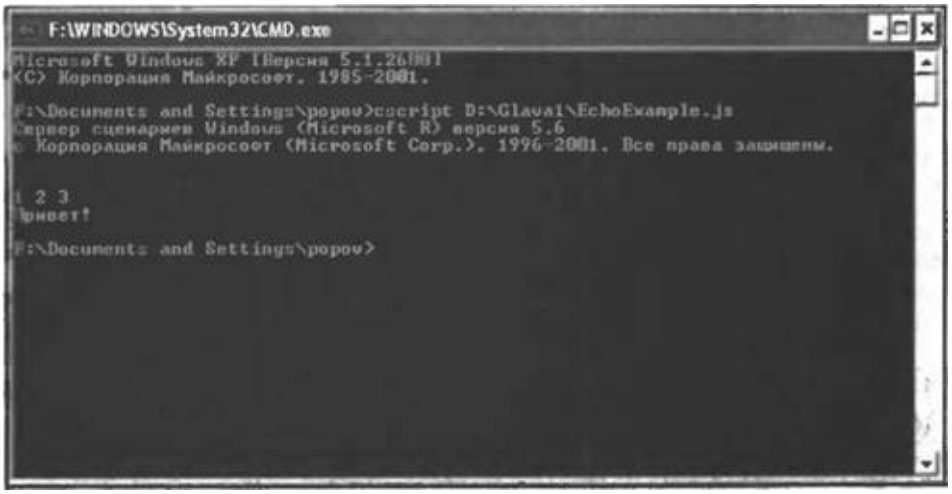


Рис. 1.10. Вывод информации с помощью метода Echo

Другие примеры использования метода `Echo` приведены в главе 2 (см. листинги 2.1, 2.2, 2.4 и 2.5).

Метод *Sleep*

В следующем примере сценарий переводится в неактивное состояние на 5 секунд:

```
WScript.Echo("Сценарий запущен, отдыхаем...");
WScript.Sleep(5000);
WScript.Echo("Выполнение завершено");
```

Метод `Sleep` необходимо применять при асинхронной работе сценария и какой-либо другой задачи, например, при имитации нажатий клавиш в активном окне с помощью метода `WshShell.SendKeys` (см. листинги 1.13, 2.31, 2.32).

Объекты-коллекции

В WSH входят объекты, с помощью которых можно получить доступ к коллекциям, содержащим следующие элементы:

□ параметры командной строки запущенного сценария или ярлыка Windows (объекты `WshArguments`, `WshNamed` и `WshUnnamed`);

□ значения переменных среды (объект `WshEnvironment`);

□ пути к специальным папкам Windows (объект `WshSpecialFolders`).

Объект *WshArguments*

Объект `WshArguments` содержит коллекцию всех параметров командной строки запущенного сценария или ярлыка Windows. Этот объект можно создать только **с** помощью свойства `Arguments` объектов `WScript` и `WshShortcut`.

В принципе, работать с элементами коллекции `WshArguments` можно стандартным для JScript образом — создать объект `Enumerator` и использовать его методы `moveNext`, `item` и `atEnd`. Например, вывести на экран все параметры командной строки, с которыми запущен сценарий, можно следующим образом (листинг 1.4).

Листинг 1.4. Вывод всех параметров сценария (стандартные коллекции JScript)

```
    /*****************************************************************
    /*                                                          Имя:
EnumArgs.js                                                   */
    /*                                                         Язык:
JScript                                                     */
    /*   Описание:   Вывод   на   экран   параметров   запущенного
сценария         */
    /*****************************************************************
    var objArgs, e, x;
    objArgs = WScript.Arguments; //Создаем объект WshArguments
    //Создаем объект Enumerator для коллекции objArgs
    e = new Enumerator(objArgs);
    for (;!e.atEnd();e.moveNext()) {
     x = e.item();  //Получаем значение элемента коллекции
```

```
     WScript.Echo(x); //Выводим значение параметра на экран
    }
    /*************                                          Конец
*******************************************/
```

Однако намного удобнее использовать методы `Count` и `Item` самого объекта `WshArguments` (метод `Item` имеется у всех коллекций WSH). Метод `Count` возвращает число элементов в коллекции, т. е. количество аргументов командной строки, а метод `Item(n)` — значение n-го элемента коллекции (нумерация начинается с нуля). Более того, чтобы получить значение отдельного элемента коллекции `WshArguments`, можно просто указать его индекс в круглых скобках после имени объекта.

Замечание

Число элементов в коллекции хранится и в свойстве `Length` объекта `WshArguments`.

Таким образом, предыдущий пример можно переписать более компактным образом (листинг 1.5).

Листинг 1.5. Вывод всех параметров сценария (методы WSH)

```
    /*************************************************************
    /*                                                        Имя:
ShowArgs.js                                                 */
    /*                                                       Язык:
JScript                                                   */
    /*   Описание:   Вывод   на   экран   параметров   запущенного
сценария        */
    /*************************************************************
    var i, objArgs;
    objArgs = WScript.Arguments; //Создаем объект WshArguments
    for (i=0; i<=objArgs.Count()-1; i++)
     WScript.Echo(objArgs(i)); //Выводим на экран i-й аргумент
    /*************                                          Конец
*******************************************/
```

С помощью объекта `WshArguments` можно также выделять и отдельно обрабатывать аргументы сценария, у которых имеются имена (например, `/Name:Andrey`) и безымянные аргументы. Ясно, что использование именных параметров более удобно, т. к. в этом случае нет необходимости

запоминать, в каком порядке должны быть записаны параметры при запуске того или иного сценария.

Для доступа к именным и безымянным аргументам используются соответственно два специальных свойства объекта `WshArguments`: `Named` и `Unnamed`.

Свойство `Named` содержит ссылку на коллекцию `WshNamed`, свойство `Unnamed` — на коллекцию `WshUnnamed`.

Таким образом, обрабатывать параметры командной строки запущенного сценария можно тремя способами:

□ просматривать полный набор всех параметров (как именных, так и безымянных) с помощью коллекции `WshArguments`;

□ выделить только те параметры, у которых есть имена (именные параметры) с помощью коллекции `WshNamed`;

□ выделить только те параметры, у которых нет имен (безымянные параметры) с помощью коллекции `WshUnnamed`.

У объекта `WshArguments` имеется еще один метод `ShowUsage`. Этот метод служит для вывода на экран информации о запущенном сценарии (описание аргументов командной строки, пример запуска сценария и т.д.). В свою очередь, подобную информацию можно задать только при использовании WSH-сценариев с разметкой XML; более подробно о применении метода `ShowUsage` идет речь в *главе 3* при рассмотрении элементов `<runtime>`, `<description>`, `<example>`, `<named>` и `<unnamed>`.

Объект *WshNamed*

Объект `WshNamed` содержит коллекцию параметров командной строки запущенного сценария, у которых имеется уникальное имя *(именные параметры)*. В WSH 5.6 именной параметр всегда начинается с символа "/", после чего приводится имя этого параметра, затем ставится двоеточие ":" и пишется значение параметра, например:

```
/Name:Andrey
```

В качестве значения именного параметра рассматривается набор символов, начинающихся после двоеточия и заканчивающихся перед первым встретившимся пробелом. Для того чтобы значением параметра командной строки была строка, состоящая из нескольких слов, необходимо заключить эту строку в кавычки. Например:

```
/Name:"Andrey Popov"
```

Создается объект WshNamed с помощью свойства `Named` коллекции `WshArguments`. Для того чтобы получить значение определенного аргумента,

его имя используется в качестве индекса коллекции.

Замечание

Узнать число именных параметров можно только с помощью свойства `Length` коллекции `WshNamed`; метода `Count` у этой коллекции нет.

Например, пусть сценарий MyScript.js запущен с двумя именными параметрами:

```
MyScript.js /User:Andrey /Computer:Server1
```

Тогда вывести на экран значение параметров Name и Computer можно двумя способами:

```
var objNamedArgs;
objNamedArgs=WScript.Arguments.Named;
WScript.Echo("Имя пользователя: "+objNamedArgs.Item("User"));
WScript.Echo("Имя компьютера: "+objNamedArgs.Item
("Computer"));
```

или просто

```
var objNamedArgs;
objNamedArgs=WScript.Arguments.Named;
WScript.Echo("Имя пользователя: "+objNamedArgs("User"));
WScript.Echo("Имя компьютера: "+objNamedArgs("Computer"));
```

Отметим также, что значением именного параметра, как и безымянного, может служить целая строка, состоящая из нескольких литералов и заключенная в кавычки, например:

```
MyScript.js /User:"Andrey Popov" /Computer:Server1
```

Тогда в результате выполнения в сценарии MyScript.js следующей строки:

```
WScript.Echo("Имя пользователя: "+
WScript.Arguments.Named("User"));
```

на экран будет выведено

```
Имя пользователя: Andrey Popov
```

Для того чтобы узнать, был ли указан при запуске сценария тот или иной именной параметр, используется метод `Exists` объекта `WshNamed`. Например,

```
if (WScript.Arguments.Named.Exists("User"))
        WScript.Echo("Имя пользователя: "+
WScript.Arguments.Named("User"));
```

Примеры, иллюстрирующие использование объекта WshNamed, приведены также в *главе 2* (см. листинги 2.22 и 2.23).

Объект *WshUnnamed*

В коллекции `WshUnnamed` содержатся параметры командной строки запущенного сценария, у которых нет имени *(безымянные параметры)*. Создается этот объект с помощью свойства `Unnamed` коллекции `WshArguments`. Для того чтобы получить значение определенного аргумента, его номер используется в качестве индекса коллекции (нумерация начинается с нуля).

Замечание

Узнать число безымянных параметров можно только с помощью свойства `Length` коллекции `WshUnnamed`; метода `Count` у этой коллекции нет.

Например, сценарий MyScript.js запущен с двумя безымянными параметрами:

```
MyScript.js "Andrey Popov" Server1
```

Тогда после выполнения в сценарии MyScript.js любого из следующих двух блоков:

```
var objUnnamedArgs;
obUnnamedArgs=WScript.Arguments.Unnamed;
WScript.Echo("Имя пользователя: "+objUnnamedArgs.Item(0));
WScript.Echo("Имя компьютера: "+objUnnamedArgs.Item(1));
```

или

```
var objUnnamedArgs;
obUnnamedArgs=WScript.Arguments.Unnamed;
WScript.Echo("Имя пользователя: "+objUnnamedArgs(0));
WScript.Echo("Имя компьютера: "+objUnnamedArgs(1));
```

на экран выведутся следующие строки:

```
Имя пользователя: Andrey Popov
Имя компьютера: Server1
```

Примеры, иллюстрирующие использование объекта `WshUnnamed`, приведены также в *главе 2* (см. листинги 2.22 и 2.23).

Объект *WshEnvironment*

Объект `WshEnvironment` позволяет получить доступ к коллекции, содержащей переменные среды заданного типа (переменные среды

операционной системы, переменные среды пользователя или переменные среды текущего командного окна). Этот объект можно создать с помощью свойства `Environment` объекта `WshShell` или одноименного его метода:

```
var WshShell=WScript.Createobject("WScript.Shell"),
 WshSysEnv=WshShell.Environment,
 WshUserEnv=WshShell.Environment("User");
```

Объект `WshEnvironment` имеет свойство `Length`, в котором хранится число элементов в коллекции (количество переменных среды), и методы `Count` и `Item`. Для того чтобы получить значение определенной переменной среды, в качестве аргумента метода `Item` указывается имя этой переменной в двойных кавычках. В следующем примере мы выводим на экран значение переменной среды `PATH`:

```
var WshShell=WScript.CreateObject("WScript.Shell"),
WshSysEnv=WshShell.Environment;
WScript.Echo("Системный путь:", WshSysEnv.Item("PATH"));
```

Можно также просто указать имя переменной в круглых скобках после имени объекта:

```
WScript.Echo("Системный путь:",WshSysEnv("PATH");
```

Кроме этого, у объекта WshEnvironment имеется метод Remove(strName), который удаляет заданную переменную среды. Например, в листинге 1.6 приведен сценарий, который удаляет две переменные (`EXAMPLE_1` и `EXAMPLE_2`) из окружения среды пользователя.

Замечание

Если в окружении среды пользователя нет переменных с именами `EXAMPLE_1` и `EXAMPLE_2`, то при вызове метода `Remove` произойдет ошибка.

Листинг 1.6. Удаление переменных среды

```
/*****************************************************************
/*                                                          Имя:
RemEnv.js                                                  */
/*                                                         Язык:
JScript                                                   */
/*        Описание:        Удаление        двух        переменных
среды                        */
/*****************************************************************
```

```
    //Создаем объект WshShell
    var WshShell = WScript.CreateObject("WScript.Shell");
    //Создаем объект WshEnvironment
    var WshUsrEnv = WshShell.Environment("User");
    //Удаляем переменные среды
    WshUsrEnv.Remove("EXAMPLE_1");
    WshUsrEnv.Remove("EXAMPLE_2");
    /*************                                          Конец
*******************************************/
```

Объект *WshSpecialFolders*

Объект `WshSpecialFolders` обеспечивает доступ к коллекции, содержащей пути к специальным папкам Windows (например, к рабочему столу или к меню **Пуск** (Start)); задание путей к таким папкам может быть необходимо, например, для создания непосредственно из сценария ярлыков на рабочем столе.

В Windows 9*x* поддерживаются следующие имена специальных папок:

- Desktop;
- Favorites;
- Fonts;
- MyDocuments;
- NetHood;
- PrintHood;
- Programs;
- Recent;
- SendTo;
- StartMenu;
- Startup;
- Templates.

В Windows NT/2000/XP дополнительно можно получить доступ еще к четырем папкам, которые хранят данные для всех пользователей:

- AllUsersDesktop;
- AllUsersStartMenu;
- AllUsersPrograms;
- AllUsersStartup.

Объект `WshSpecialFolders` создается с помощью свойства `SpecialFolders` объекта `WshShell`:

```
    var WshShell=WScript.CreateObject("WScript.Shell"),
     WshSpecFold=WshShell.SpecialFolders;
```

Как и почти все коллекции WSH, объект `WshSpecialFolders` имеет свойство `Length` и методы `Count` и `Item`. Доступ к отдельному элементу производится либо через имя соответствующей папки, либо через числовой индекс (Листинг 1.7).

Листинг 1.7. Обработка коллекции `WshSpecialFolders`

```
    /*****************************************************************
    /*                                                              Имя:
ShowSpecFold.js                                                  */
    /*                                                             Язык:
JScript                                                         */
    /*  Описание:  Вывод  на  экран  названий  специальных  папок
Windows     */
    /*                                                         (коллекция
WshSpecialFolders)                              */
    /*****************************************************************
    var WshShell, WshFldrs, i;
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("Wscript.Shell");
    //Создаем объект WshSpecialFolders
    WshFldrs = WshShell.SpecialFolders;
    WScript.Echo("Некоторые специальные папки...");
    //Выводим путь к папке Desktop
    WScript.Echo("Desktop="+ WshFldrs.item("Desktop"));
    //Выводим путь к папке Favorities
    WScript.Echo("Favorites="+ WshFldrs("Favorites"));
    //Выводим путь к папке Programs
    WScript.Echo("Programs="+ WshFldrs("Programs"));
    WScript.Echo("");
    WScript.Echo("Список всех специальных папок...");
    for (i=0;i<= WshFldrs.Count()-1;i++){
     //Выводим на экран i-й элемент коллекции WshFldrs
     WScript.Echo(WshFldrs(i));
    }
    /*************                                                 Конец
*******************************************/
```

Другие примеры работы со специальными папками Windows приведены в *главе 2* (см. листинги 2.39–2.42).

Работа с сетью и оболочкой Windows

Для работы с локальной сетью и оболочкой Windows (специальные папки, переменные среды, системный реестр) предназначены соответственно объекты `WshNetwork` и `WshShell`.

Объект *WshNetwork*

Объект `WshNetwork` предназначен для работы с ресурсами локальной сети; с помощью методов этого объекта можно подключать и отключать сетевые диски и принтеры.

Объект `WshNetwork` создается следующим образом:

```
var objNet=WScript.CreateObject("WScript.Network");
```

Свойства данного объекта приведены в табл. 1.5.

Таблица 1.5. *Свойства объекта* `WshNetwork`

Свойство	Описание
`ComputerName`	Содержит имя компьютера, на котором запущен сценарий
`UserDomain`	Содержит имя домена, в котором зарегистрировался пользователь
`UserName`	Содержит имя пользователя

Листинг 1.8. Пример использования объекта `WshNetwork`

```
    /*****************************************************************
    /*                                                          Имя:
ShowNetwork.js                                                  */
    /*                                                         Язык:
JScript                                                        */
    /*  Описание:  Вывод  на  экран  сетевого  имени  компьютера  и
имени      */
    /*
пользователя                                                  */
    /*****************************************************************
    var objNet;
    //Создаем объект WshNetwork
    objNet = WScript.CreateObject("WScript.Network");
```

```
    //Выводим на экран свойства ComputerName и UserName
    WScript.Echo("Имя машины:",objNet.ComputerName);
    WScript.Echo("Имя пользователя:",objNet.UserName);
    /*************                                          Конец
*****************************************/
```

Методы объекта `WshNetwork` описаны в табл. 1.6.

Таблица 1.6. *Методы объекта* `WshNetwork`

Метод	Описание
`AddPrinterConnection(`*`strLocalName, strRemoteName [ ,bUpdateProfile] [,strUser] [,strPassword]`*`)`	Подключает локальный порт компьютера к сетевому принтеру
Для Windows NT/2000/XP: `AddWindowsPrinterConnection(`*`strPrnPath`*`)` Для Windows 9*x*: `AddWindowsPrinterConnection(`*`strPrnPath, strDriverName[, strPort]`*`)`	Регистрирует принтер в Windows и подключает его к сетевому ресурсу. В отличие от `AddPrinterConnection`, этот метод позволяет создать связь с сетевым принтером без явного перенаправления вывода в локальный порт
`EnumNetworkDrives()`	Возвращает коллекцию, в которой хранятся буквы и сетевые пути ко всем подключенным сетевым дискам
`EnumPrinterConnections()`	Возвращает коллекцию, в которой хранятся данные обо всех подключенных сетевых принтерах
`MapNetworkDrive(`*`strLocalName, strRemoteName, [bUpdateProfile], [strUser], [strPassword]`*`)`	Подключает сетевой ресурс*`strRemoteName`*под локальным именем диска *`strLocalName`*
`RemoveNetworkDrive(`*`strName, [bForce], [bUpdateProfile]`*`)`	Отключает подключенный сетевой диск
`RemovePrinterConnection(`*`strName, [bForce], [bUpdateProfile]`*`)`	Отключает подключенный сетевой принтер

`SetDefaultPrinter(`*`strPrinterName`*`)`	Делает заданный сетевой принтер принтером по умолчанию

Опишем методы из табл. 1.6 более подробно.

Метод *AddPrinterConnection*

Если необязательный параметр *bUpdateProfile* равен `True`, то создаваемое сетевое подключение будет сохранено в профиле пользователя.

Параметры *strUser* (имя пользователя) и *strPassword* (пароль) нужны в том случае, когда вы подключаете сетевой принтер от имени пользователя, которое отличается от имени текущего пользователя, зарегистрированного в системе.

В следующем примере метод `AddPrinterConnection` применяется для подключения принтера с сетевым именем `\\Server1\Epson` к локальному порту `LPT1`:

```
var WshNetwork = CreateObject("WScript.Network");
WshNetwork.AddPrinterConnection("LPT1", "\\Server1\Epson");
```

Метод *AddWindowsPrinterConnection*

Параметр *strDriverName* указывает имя драйвера, необходимого для подключаемого принтера. Если принтер подключается в операционной системе Windows 9*x*, то нужный драйвер уже должен быть установлен на этой машине, иначе возникнет ошибка подключения. В Windows NT/2000/XP параметр *strDriverName* игнорируется.

Параметр *strPort* задает в явном виде порт, вывод в который будет перенаправлен на сетевой ресурс (по умолчанию это порт `LPT1`). В Windows NT/2000/XP параметр *strPort* игнорируется.

В следующем примере метод `AddWindowsPrinterConnection` применяется для подключения сетевого принтера к локальному порту `LPT1` (по умолчанию):

```
var WshNetwork=CreateObject("WScript.Network");
PrinterPath="\\printserv\DefaultPrinter";
PrinterDriver="Lexmark Optra S 1650";
WshNetwork.AddwindowsPrinterConnection(PrinterPath,
PrinterDriver);
```

Метод *EnumNetworkDrives*

Элементами возвращаемой коллекции являются буквы, обозначающие имеющиеся сетевые диски и сетевые имена ресурсов, к которым эти диски подключены. Первым элементом коллекции является буква, вторым — сетевое имя; эта последовательность сохраняется для всех сетевых дисков в коллекции.

В следующем примере на экран выводятся буквы, обозначающие все сетевые диски и имена ресурсов, к которым они подключены (листинг 1.9).

Листинг 1.9. Пример использования метода `EnumNetworkDrives`

```
    /*****************************************************************
    /*                                                           Имя:
ShowNetDrives.js                                              */
    /*                                                          Язык:
JScript                                                      */
    /*   Описание:   Вывод   на   экран   букв   сетевых   дисков   и
имен            */
    /*                                                 соответствующих
ресурсов                                  */
    /*****************************************************************
    var WshNetwork,oDrives,i;
    //Создаем объект WshNetwork
    WshNetwork = WScript.CreateObject("WScript.Network");
    //Создаем коллекцию с информацией о сетевых дисках
    oDrives = WshNetwork.EnumNetworkDrives();
    for (i=0; i<=oDrives.Count()-1; i++)
     WScript.Echo(oDrives.Item(i)); //Вывод i-го элемента коллекции
    /*************                                              Конец
*******************************************/
```

Метод *EnumPrinterConnections*

Элементами возвращаемой коллекции являются названия локальных портов и сетевые имена принтеров, связанных с этими портами. Сама коллекция организована так же, как и коллекция, возвращаемая методом `EnumNetworkDrives`.

В следующем примере на экран выводятся названия всех переназначенных портов и имена сетевых ресурсов, с которыми они

связаны (листинг 1.10).

Листинг 1.10. Пример использования метода `EnumPrinterConnections`

```
    /*****************************************************************
    /*                                                          Имя:
ShowNetPrn.js                                                  */
    /*                                                         Язык:
JScript                                                      */
    /*  Описание:  Вывод  на  экран  переназначенных  портов  и
имен          */
    /*                                                 соответствующих
ресурсов                                  */
    /*****************************************************************
    //Создаем объект WshNetwork
    var WshNetwork = WScript.CreateObject("WScript.Network");
    //Создаем коллекцию с информацией о подключенных принтерах
    var oPrinters = WshNetwork.EnumPrinterConnections();
    for (i=0; i<=oPrinters.Count()-1; i++)
       WScript.Echo(oPrinters.Item(i));   //Вывод   i-го   элемента
коллекции
    /*************                                             Конец
*********************************************/
```

Метод *MapNetworkDrive*

Если необязательный параметр *`bUpdateProfile`* равен `True`, то создаваемое сетевое подключение будет сохранено в профиле пользователя.

Параметры *`strUser`* (имя пользователя) и *`strPassword`* (пароль) нужны в том случае, когда вы подключаете сетевой диск от имени пользователя, которое отличается от имени текущего пользователя, зарегистрированного в системе.

В следующем примере диск "z" подключается к сетевому ресурсу \\Server1\Programs:

```
    var WshNetwork = WScript.CreateObject("WScript.Network");
    WshNetwork.MapNetworkDrive("Z:","\\Server1\Programs");
```

Метод *RemoveNetworkDrive*

В качестве параметра *`strName`* может быть указано либо локальное имя

(буква сетевого диска), либо сетевое имя (имя подключенного сетевого ресурса); это зависит от того, каким образом осуществлялось подключение. Если сетевому ресурсу сопоставлена буква локального диска, то параметр *strName* должен быть локальным именем. Если сетевому ресурсу не сопоставлена никакая буква, то параметр *strName* должен быть сетевым именем.

Если необязательный параметр *bForce* равен `True`, то отключение сетевого ресурса будет произведено вне зависимости от того, используется этот ресурс в настоящее время или нет.

Если необязательный параметр *bUpdateProfile* равен `True`, то отключаемое сетевое подключение будет удалено из профиля пользователя.

В следующем примере производится подключение диска "z" к сетевому ресурсу, а затем отключение этого ресурса (листинг 1.11).

Листинг 1.11. Пример подключения и отключения сетевого диска

```
    /*****************************************************************
    /*                                                          Имя:
MapDrive.js                                                  */
    /*                                                         Язык:
JScript                                                     */
    /*        Описание:        Подключение/отключение        сетевого
ресурса                */
    /*****************************************************************
    //Создаем объект WshNetwork
    var WshNetwork = WScript.CreateObject("WScript.Network");
    //Подключаем сетевой диск Z:\ к \\Server1\Programs
    WshNetwork.MapNetworkDrive("Z:","\\Server1\Programs");
    //Отключаем сетевой диск Z:\
    WshNetwork.RemoveNetworkDrive("Z:");
    /*************                                            Конец
*******************************************/
```

Метод *RemovePrinterConnection*

В качестве параметра *strName* может быть указано либо локальное имя (название порта), либо сетевое имя (имя подключенного сетевого принтера); это зависит от того, каким образом осуществлялось подключение. Если сетевому ресурсу явным образом сопоставлен локальный порт (например, LPT1), то параметр *strName* должен быть

локальным именем. Если сетевому принтеру не сопоставлен локальный порт, то параметр *strName* должен быть сетевым именем.

Параметры *bForce* и *bUpdateProfile* в этом методе имеют то же значение, что и одноимённые параметры в методе RemoveNetworkDrive.

В следующем примере отключается сетевой принтер, который был назначен на порт LPT1:

```
var WshNetwork = WScript.CreateObject("WScript.Network");
WshNetwork.RemovePrinterConnection("LPT1:");
```

Метод *SetDefaultPrinter*

Параметр *strName* задает сетевое имя принтера, который должен будет использоваться в системе по умолчанию.

В следующем примере с помощью метода AddPrinterConnection к порту LPT1: подключается сетевой принтер \\Server1\Epson, который затем устанавливается принтером по умолчанию (листинг 1.12).

Листинг 1.12. Пример использования метода SetDefaultPrinter

```
    /*****************************************************************
    /*                                                          Имя:
DefPrn.js                                                    */
    /*                                                         Язык:
JScript                                                    */
    /*        Описание:        Установка        принтера        по
умолчанию                      */
    /*****************************************************************
    //Создаем объект WshNetwork
    var WshNetwork = WScript.CreateObject("WScript.Network");
    //Подключаем к LPT1 сетевой принтер \\Server1\Epson
    WshNetwork.AddPrinterConnection("LPT1:","\\Server1\Epson");
    //Устанавливаем принтер по умолчанию
    WshNetwork.SetDefaultPrinter("\\Server1\Epson");
    /*************                                            Конец
*******************************************/
```

Другие примеры, иллюстрирующие использование объекта WshNetwork, приведены в *главе 2* (см. листинги 2.47–2.52).

Объект *WshShell*

С помощью объекта `WshShell` можно запускать новый процесс, создавать ярлыки, работать с системным реестром, получать доступ к переменным среды и специальным папкам Windows. Создается этот объект следующим образом:

```
var WshShell=WScript.CreateObject("WScript.Shell");
```

Объект `WshShell` имеет три свойства, которые приведены в табл. 1.7.

Таблица 1.7. *Свойства объекта* `WshShell`

Свойство	Описание
`CurrentDirectory`	Здесь хранится полный путь к текущему каталогу (к каталогу, из которого был запущен сценарий)
`Environment`	Содержит объект `WshEnvironment`, который обеспечивает доступ к переменным среды операционной системы для Windows NT/2000/XP или к переменным среды текущего командного окна для Windows 9x
`SpecialFolders`	Содержит объект `WshSpecialFolders` для доступа к специальным папкам Windows (рабочий стол, меню Пуск (Start) и т.д.)

Опишем теперь методы, имеющиеся у объекта `WshShell` (табл. 1.8).

Таблица 1.8. *Методы объекта* `WshShell`

Метод	Описание
`AppActivate(`*`title`*`)`	Активизирует заданное параметром *`title`* окно приложения. Строка *`title`* задает название окна (например, "calc" или "notepad") или идентификатор процесса (ProcessID, PID)
`CreateShortcut(`*`strPathname`*`)`	Создает объект `WshShortcut` для связи с ярлыком Windows (расширение lnk) или объект `WshUrlShortcut` для связи с сетевым ярлыком (расширение url). Параметр *`strPathname`* задает полный путь к создаваемому или изменяемому ярлыку

`Environment(`*`strType`*`)`	Возвращает объект `WshEnvironment`, содержащий переменные среды заданного вида
`Exec(`*`strCommand`*`)`	Создает новый дочерний процесс, который запускает консольное приложение, заданное параметром *`strCommand`*. В результате возвращается объект `WshScriptExec`, позволяющий контролировать ход выполнения запущенного приложения и обеспечивающий доступ к потокам `StdIn`, `StdOut` и `StdErr` этого приложения
`ExpandEnvironmentStrings(`*`strString`*`)`	Возвращает значение переменной среды текущего командного окна, заданной строкой *`strString`* (имя переменной должно быть окружено знаками "%")
`LogEvent(`*`intType, strMessage [ ,strTarget]`*`)`	Протоколирует события в журнале Windows NT/2000/XP или в файле WSH.log. Целочисленный параметр *`intType`* определяет тип сообщения, строка *`strMessage`* — текст сообщения. Параметр *`strTarget`* может задаваться только в Windows NT/2000/XP, он определяет название системы, в которой протоколируются события (по умолчанию это локальная система). Метод `LogEvent` возвращает `true`, если событие записано успешно и `false` в противном случае
	Выводит на экран информационное окно с сообщением, заданным параметром*`strText`*. Параметр

`Popup(strText, [nSecToWait] , [strTitle], [nType])`	*nSecToWait* задает количество секунд, по истечении которых окно будет автоматически закрыто, параметр*strTitle* определяет заголовок окна, параметр*nType* указывает тип кнопок и значка для окна
`RegDelete(strName)`	Удаляет из системного реестра заданный параметр или раздел целиком
`RegRead(strName)`	Возвращает значение параметра реестра или значение по умолчанию для раздела реестра
`RegWrite(strName, anyValue [,strType])`	Записывает в реестр значение заданного параметра или значение по умолчанию для раздела
`Run(strCommand, [intWindowStyle], [bWaitOnReturn])`	Создает новый независимый процесс, который запускает приложение, заданное параметром*strCommand*
`SendKeys(string)`	Посылает одно или несколько нажатий клавиш в активное окно (эффект тот же, как если бы вы нажимали эти клавиши на клавиатуре)
`SpecialFolders(strSpecFolder)`	Возвращает строку, содержащую путь к специальной папке Windows, заданной параметром *strSpecFolder*

Рассмотрим методы, приведенные в табл. 1.8, более подробно.

Метод *AppActivate*

Метод `AppActivate` активизирует уже запущенное указанное приложение (устанавливает на него фокус), но не производит никаких действий по изменению размеров его окна. Для того чтобы первоначально

запустить нужное приложение и определить вид его окна, следует использовать метод `Run` объекта `WshShell`. Для того чтобы определить, какое именно приложение необходимо активизировать, строка *`title`* сравнивается по очереди с названиями окон всех запущенных приложений. Если не найдено ни одного точного совпадения, будет производиться поиск того приложения, название окна которого начинается со строки *`title`*. Если и в этом случае не будет найдено ни одного подходящего приложения, то будет вестись поиск приложения, заголовок которого заканчивается на эту строку. Если будет найдено несколько подходящих окон, то произойдет активизация одного из них (окно выбирается произвольно).

В качестве примера использования метода `AppActivate` в листинге 1.13 приведен сценарий RunCalc.js, который запускает стандартный калькулятор Windows и выполняет в нем несколько простых арифметических действий (для этого используется метод `SendKeys`).

Листинг 1.13. Сценарий RunCalc.js

```
    /*****************************************************************
    /*                                                          Имя:
RunCalc.js                                                  */
    /*                                                         Язык:
JScript                                                   */
    /*   Описание:   Активизация   приложения   с   помощью   имени
окна         */
    /*****************************************************************
    //Создаем объект WshShell
    var WshShell = WScript.CreateObject("WScript.Shell");
    //Запускаем Калькулятор
    WshShell.Run("calc");
    //Приостанавливаем сценарий на 0,1 секунды
    WScript.Sleep(100);
    //Активизируем Калькулятор
    WshShell.AppActivate("Calculator");
    //Приостановка сценария на 0,1 секунды
    WScript.Sleep(100);
    //Посылаем нажатия клавиш в Калькулятор
    WshShell.SendKeys("1{+}");
    WScript.Sleep(500);
    WshShell.SendKeys("2");
    WScript.Sleep(500);
    WshShell.SendKeys("~");
    WScript.Sleep(2500);
```

```
    /*************                                                    Конец
*******************************************/
```

В *главе 2* приведены другие примеры использования метода `AppActivate` (см. листинги 2.31 и 2.32).

Метод *CreateShortcut*

Этот метод позволяет создать новый или открыть уже существующий ярлык для изменения его свойств.

В листинге 1.14 приведен пример сценария, в котором создаются два ярлыка — на сам выполняемый сценарий (объект `oShellLink` и на сетевой ресурс (`oUrlLink`).

Листинг 1.14. Пример использования метода `CreateShortcut`

```
    /*****************************************************************
    /*                                                              Имя:
MakeShortcuts.js                                              */
    /*                                                             Язык:
JScript                                                      */
    /*         Описание:         Создание         ярлыков         из
сценария                        */
    /*****************************************************************
    var WshShell,oShellLink,oUrlLink;
    //Создаем объект WshShell
    WshShell=WScript.CreateObject("WScript.Shell");
    //Создаем ярлык на файл
    oShellLink=WshShell.CreateShortcut("Current Script.lnk");
    //Устанавливаем путь к файлу
    oShellLink.TargetPath=WScript.ScriptFullName;
    //Сохраняем ярлык
    oShellLink.Save();
    //Создаем ярлык на сетевой ресурс
    oUrlLink = WshShell.CreateShortcut("Microsoft Web Site.URL");
    //Устанавливаем URL
    oUrlLink.TargetPath = "http://www.microsoft.com";
    //Сохраняем ярлык
    oUrlLink.Save();
    /*************                                                    Конец
*******************************************/
```

Примеры работы с ярлыками приведены в листингах 1.19–1.24, 2.43 и 2.44.

Метод *Environment*

Параметр *strType* задает вид переменных среды, которые будут записаны в коллекции WshEnvironment; возможными значениями этого параметра являются "System" (переменные среды операционной системы), "User" (переменные среды пользователя), "Volatile" (временные переменные) или "Process" (переменные среды текущего командного окна).

Замечание

Для Windows 9x единственным допустимым значением параметра *strType* является "Process".

В следующем примере мы распечатываем число процессоров, имеющихся в компьютере с операционной системой Windows NT/2000/XP (переменная NUMBER_OF_PROCESSORS), и путь к каталогу Windows (листинг 1.15).

Листинг 1.15. Печать значений переменных среды (объект WshShell.Environment)

```
    /*****************************************************************
    /*                                                          Имя:
ShowEnvir.js                                                  */
    /*                                                         Язык:
JScript                                                     */
    /*   Описание:   Получение   значений   некоторых   переменных
среды       */
    /*****************************************************************
    var WshShell,WshSysEnv;
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    //Создание коллекции WshEnvironment
    WshSysEnv = WshShell.Environment("SYSTEM");
    WScript.Echo(WshSysEnv("NUMBER_OF_PROCESSORS"));
    WScript.Echo(WshShell.Environment.Item("WINDIR"));
    /*************                                            Конец
*******************************************/
```

Метод *ExpandEnvironmentString*

В следующем примере на экран выводится путь к каталогу Windows (листинг 1.16).

Листинг 1.16. Печать значений переменных среды (`ExpandEnvironmertStrings`)

```
    /*****************************************************************
    /*                                                           Имя:
ExpEnvStr.js                                                  */
    /*                                                          Язык:
JScript                                                      */
    /*                                                       Кодировка:
Windows                                                  */
    /*   Описание:   Получение   значений   переменный   среды     с
помощью      */
    /*                                                         метода
ExpandEnvironmentString                          */
    /*****************************************************************
    var WS = WScript.CreateObject("WScript.Shell");
    WScript.Echo("Каталог
Windows:"+WS.ExpandEnvironmentStrings("%WinDir%"));
    /*************                                              Конец
*********************************************/
```

Метод *LogEvent*

В Windows NT/2000/XP события записываются в системном журнале, а в Windows 9x — в файле WSH.log, расположенном в каталоге пользователей Windows. Запись в WSH.log будет содержать время события, его тип и текст. Типы сообщений описаны в табл. 1.9.

Таблица 1.9. *Типы сообщений (параметр `intType`)*

Код	Значение	Код	Значение
0	`SUCCESS`	4	`INFORMATION`
1	`ERROR`	8	`AUDIT_SUCCESS`
2	`WARNING`	16	`AUDIT_FAILURE`

В следующем примере производится протоколирование работы сценария регистрации (здесь предполагается, что если этот сценарий отработал успешно, то функция `RunLoginScript` возвращает true, в противном случае — `false`):

```
    var WshShell = WScript.CreateObject("WScript.Shell");
    rc = RunLoginScript();
    if    (rc)    WshShell.LogEvent(0,"Logon    Script    Completed
Successfully");
    else WshShell.LogEvent(1,"Logon Script failed");
```

Метод *Popup*

Если в методе не задан параметр*`strTitle`*, то по умолчанию заголовком окна будет "Windows Script Host."

Параметр *`nType`* может принимать те же значения, что и в функции `MessageBox` из Microsoft Win32 API. В табл. 1.10 описаны некоторые возможные значения параметра nType и их смысл (полный список значений этого параметра можно посмотреть в описании функции MessageBox в документации по функциям Windows API).

Таблица 1.10. *Типы кнопок и иконок для метода* `Popup`

Значение *nType*	**Константа Visual Basic**	**Описание**
0	`vbOkOnly`	Выводится кнопка `OK`
1	`vbOkCancel`	Выводятся кнопки `OK` и `Отмена` (Cancel)
2	`vbAbortRetryIgnore`	Выводятся кнопки **Стоп** (Abort), **Повтор** (Retry) и **Пропустить** (Ignore)
3	`vbYesNoCancel`	Выводятся кнопки **Да** (Yes), **Нет** (No) и **Отмена** (Cancel)
4	`vbYesNo`	Выводятся кнопки **Да** (Yes) и **Нет** (No)
5	`vbRetryCancel`	Выводятся кнопки **Повтор** (Retry) и **Отмена** (Cancel)
16	`vbCritical`	Выводится значок **Stop Mark**
32	`vbQuestion`	Выводится значок **Question Mark**
48	`vbExclamation`	Выводится значок **Exclamation Mark**
64	`vbInformation`	Выводится значок **Information Mark**

Замечание

В сценариях, написанных на языке VBScript, можно непосредственно использовать именованные константы типа `vbOkCancel` без предварительного их объявления. Для того чтобы использовать такие константы в JScript-сценариях, их нужно предварительно объявить как переменные и присвоить нужные значения (например, `var vbOkCancel=1;`). Естественно, в любых сценариях вместо имен констант можно использовать их числовые значения.

В методе `Popup` можно комбинировать значения параметра, приведенные в табл. 1.10. Например, в результате выполнения следующего сценария:

```
var WshShell = WScript.CreateObject("WScript.Shell");
WshShell.Popup("Копирование завершено успешно", 5, "Ура",65);
```

на экран будет выведено информационное окно, показанное на рис. 1.11, которое автоматически закроется через 5 секунд.

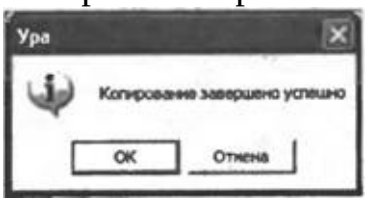


Рис. 1.11. Информационное окно, созданное методом `Popup`

Метод Popup возвращает целое значение, с помощью которого можно узнать, какая именно кнопка была нажата для выхода (табл. 1.11).

Таблица 1.11. *Возвращаемые методом `Popup` значения*

Значение	Константа Visual Basic	Описание
-1		Пользователь не нажал ни на одну из кнопок в течение времени, заданного параметром *`nSecToWait`*
1	`vbOk`	Нажата кнопка **OK**
2	`vbCancel`	Нажата кнопка **Отмена** (Cancel)
3	`vbAbort`	Нажата кнопка **Стоп** (Abort)
4	`vbRetry`	Нажата кнопка **Повтор** (Retry)
5	`vbIgnore`	Нажата кнопка **Пропустить** (Ignore)
6	`vbYes`	Нажата кнопка **Да** (Yes)

7	vbNo	Нажата кнопка **Нет** (No)

Примеры вывода информации с помощью метода Popup представлены в *главе 2* (см. листинги 2.13 и 2.14).

Метод *RegDelete*

Если параметр *strName* оканчивается символами \\, то этот метод удаляет ключ целиком (вместе со всеми параметрами внутри его), в противном случае удаляется только один заданный параметр. Параметр *strName* должен начинаться с одного из следующих корневых ключей (табл. 1.12).

Таблица 1.12. *Названия ключей*

Краткое название	Длинное название
HCKU	HKEY_CURRENT_USER
HKLM	HKEY_LOCAL_MACHINE
HKCR	HKEY_CLASSES_ROOT
	HKEY_USERS
	HKEY_CURRENT_CONFIG

Пример, иллюстрирующий применение метода RegDelete, приведен в *главе 2* (см. листинги 2.45 и 2.46).

Метод *RegRead*

С помощью этого метода можно прочитать следующие типы данных:

REG_SZ, REG_EXPAND_SZ, REG_DWORD, REG_BINARY и REG_MULTI_SZ. Если в реестре содержатся данные других типов, то метод RegRead вернет значение DISP_E_TYPEMISMATCH.

Если параметр *strName* оканчивается символами \\, то этот метод считывает значение по умолчанию для раздела (если оно установлено), в противном случае читается значение параметра.

В следующем примере на экран выводятся считанные из реестра с помощью метода RegRead значение параметра и значение по умолчанию для раздела реестра (листинг 1.17).

Листинг 1.17. Чтение значений параметра и раздела системного реестра

```
    /*****************************************************************
    /*                                                          Имя:
RegRead.js                                                    */
    /*                                                         Язык:
JScript                                                     */
    /* Описание: Чтение значений параметра и раздела системного
реестра */
    /*****************************************************************
    var WS,s;
    //Создаем объект WshShell
    WS = WScript.CreateObject("WScript.Shell");
    s="Значение параметра\n";
    s+="HKCU\\Control Panel\\Keyboard\\KeyboardSpeed = ";
    //Читаем значение параметра реестра
    s+=WS.RegRead("HKCU\\Control
Panel\\Keyboard\\KeyboardSpeed")+"\n\n";
    s+="Значение по умолчанию для раздела\n";
    s+="HKCU\\Control Panel\\Keyboard\\ = ";
    //Читаем значение по умолчанию для раздела реестра
    s+=WS.RegRead("HKCU\\Control Panel\\Keyboard\\");
    //Вывод на экран сформированной строки
    WScript.Echo(s);
    /*************                                            Конец
**********************************************/
```

Метод *RegWrite*

Если параметр *strName* оканчивается символами \\, то этот метод записывает раздел; если такого раздела нет, он будет создан. Если в конце *strName* отсутствуют символы \\, то производится запись указанного параметра в нужный раздел; если такого параметра нет, он будет создан с указанным именем и значением.

Параметр *anyValue* метода задает значение, которое нужно записать в заданный параметром *strName* параметр или раздел реестра. Необязательный параметр *strType* определяет тип записываемого значения. В качестве *strType* можно указывать "REG_SZ", "REG_EXPAND_SZ", "REG_DWORD" и "REG_BINARY". Если в качестве параметра *strType* передается другое значение, то метод RegWrite вернет значение E_INVALIDARG.

В случае, когда *strType* установлено в "REG_SZ" или "REG_EXPAND_SZ",

метод `RegWrite` автоматически конвертирует параметр *`anyValue`* в строку. Если значение *`strType`* равно "`REG_DWORD`", то *`anyValue`* переводится в целый формат. Если *strType* равно "`REG_BINARY`", то *`anyValue`* должно быть целым числом.

Пример, иллюстрирующий применение метода RegWrite, приведен в *главе 2* (см. листинги 2.45 и 2.46).

Метод *Run*

Параметр *`intWindowStyle`* устанавливает вид окна для запускаемого приложения (табл. 1.13).

***Таблица 1.13**. Типы окна (`intWindowStyle`)*

Параметр	Константа Visual Basic	Описание
0	`vbHide`	Прячет текущее окно и активизирует другое окно (показывает его и передает ему фокус)
1	`vbNormalFocus`	Активизирует и отображает окно. Если окно было минимизировано или максимизировано, система восстановит его первоначальное положение и размер. Этот флаг должен указываться сценарием во время первого отображения окна
2	`vbMinimizedFocus`	Активизирует окно и отображает его в минимизированном (свернутом) виде
3	`vbMaximizedFocus`	Активизирует окно и отображает его в максимизированном (развернутом) виде
4	`vbNormalNoFocus`	Отображает окно в том виде, в котором оно находилось последний раз. Активное окно при этом остается активным
5		Активизирует окно и отображает его в текущем состоянии
6	vbMinimizedNoFocus	Минимизирует заданное окно и активизирует следующее (в Z-порядке) окно

7	Отображает окно в свернутом виде. Активное окно при этом остается активным
8	Отображает окно в его текущем состоянии. Активное окно при этом остается активным
9	Активизирует и отображает окно. Если окно было минимизировано или максимизировано, система восстановит его первоначальное положение и размер. Этот флаг должен указываться, если производится восстановление свернутого окна (его нельзя использовать в методе `Run`)
10	Устанавливает режим отображения, опирающийся на режим программы, которая запускает приложение

Замечание

В сценариях, написанных на языке VBScript, можно непосредственно использовать именованные константы типа `vbHide` без предварительного их объявления. Для того чтобы использовать такие константы в JScript-сценариях, их нужно предварительно объявить как переменные и присвоить нужные значения (например, `var vbHide=0;`). Естественно, в любых сценариях вместо имен констант можно использовать их числовые значения.

Необязательный параметр *`bWaitOnReturn`* является логической переменной, дающей указание ожидать завершения запущенного процесса. Если этот параметр не указан или установлен в `false`, то после запуска из сценария нового процесса управление сразу же возвращается обратно в сценарий (не дожидаясь завершения запущенного процесса). Если же *`bWaitOnReturn`* установлен в `true`, то сценарий возобновит работу только после завершения вызванного процесса.

При этом если параметр *bWaitOnReturn* равен `true`, то метод `Run` возвращает код выхода вызванного приложения. Если же *bWaitOnReturn* равен `false` или не задан, то метод `Run` всегда возвращает ноль.

В следующем примере мы запускаем Блокнот (notepad.exe) и открываем в нем файл с выполняемым сценарием:

```
var WshShell = WScript.CreateObject("WScript.Shell");
WshShell.Run("%windir%\\notepad" + WScript.ScriptFullName);
```

Следующий сценарий печатает код выхода вызванного приложения (листинг 1.18).

Листинг 1.18. Вывод кода выхода запущенного приложения

```
/*****************************************************************
/*                                                          Имя:
RetCode.js                                                  */
/*                                                         Язык:
JScript                                                    */
/*     Описание:     Вывод     кода     выхода     запущенного
приложения                */
/*****************************************************************
//Создаем объект WshShell
var WshShell = WScript.CreateObject("WScript.Shell");
//Запускаем Блокнот и ожидаем завершения его работы
Return = WshShell.Run("notepad " + WScript.ScriptFullName, 1,
true);
//Печатаем код возврата
WScript.Echo("Код возврата:", Return);
/*************                                            Конец
**********************************************/
```

Другие примеры запуска приложений с помощью метода Run приведены в *главе 2* (см. листинги 2.31 и 2.32).

Метод *SendKeys*

Каждая клавиша задается одним или несколькими символами. Например, для того чтобы задать нажатие друг за другом букв А, Б и В, нужно указать в качестве параметра для `SendKeys` строку "АБВ": `string="АБВ"`.

Несколько символов имеют в методе SendKeys специальное значение: `+`, `^`, `%`, `~`, `(`, `)`. Для того чтобы задать один из этих символов, их нужно заключить в фигурные скобки `{}`. Например, для задания знака плюс

используется {+}. Квадратные скобки [] хотя и не имеют в методе SendKeys специального смысла, их также нужно заключать в фигурные скобки. Кроме этого, для задания самих фигурных скобок следует использовать следующие конструкции: {{} (левая скобка) и {}} (правая скобка).

Для задания неотображаемых символов, таких как <Enter> или <Tab> и специальных клавиш, в методе SendKeys используются коды, представленные в табл. 1.14.

Таблица 1.14. *Коды специальных клавиш для SendKeys*

Названия клавиш	**Код**	**Названия клавиш**	**Код**
<Backspace>	{BACKSPACE}, {BS} или {BKSP}	<→>	{RIGHT}
<Break>	{BREAK}	<F1>	{F1}
<Caps Lock>	{CAPSLOCK}	<F2>	{F2}
<Del> или <Delete>	{DELETE} или {DEL}	<F3>	{F3}
<End>	{END}	<F4>	{F4}
<Enter>	{ENTER} ИЛИ ~	<F5>	{F5}
<Esc>	{ESC}	<F6>	{F6}
<Home>	{HELP}	<F7>	{F7}
<Ins> или <Insert>	{INSERT} или {INS}	<F8>	{F8}
<Num Lock>	{NUMLOCK}	<F9>	{F9}
<Page Down>	{PGDN}	<F10>	{F10}
<Page Up>	{PGUP}	<F11>	{F11}
<Print Screen>	{PRTSC}	<F12>	{F12}
<Scroll Lock>	{SCROLLLOCK}	<F13>	{F13}
<Tab>	{TAB}	<F14>	{F14}
<↑>	{UP}	<F15>	{F15}
<←>	{LEFT}	<F16>	{F16}
<↓>	{DOWN}		

Для задания комбинаций клавиш с <Shift>, <Ctrl> или <Alt>, перед соответствующей клавишей нужно поставить один или несколько кодов из

табл. 1.15.

Таблица 1.15. *Коды клавиш <Shift>, <Ctrl> и <Alt>*

Клавиша	Код
<Shift>	+
<Ctrl>	^
<Alt>	%

Для того чтобы задать комбинацию клавиш, которую нужно набирать, удерживая нажатыми клавиши <Shift>, <Ctrl> или <Alt>, нужно заключить коды этих клавиш в скобки. Например, если требуется сымитировать нажатие клавиш <G> и <S> при нажатой клавише <Shift>, следует использовать последовательность "+(GS)". Для того же, чтобы задать одновременное нажатие клавиш <Shift>+<G>, а затем <S> (уже без <Shift>), используется "+GS".

В методе SendKeys можно задать несколько нажатий подряд одной и той же клавиши. Для этого необходимо в фигурных скобках указать код нужной клавиши, а через пробел — число нажатий. Например, {LEFT 42} означает нажатие клавиши <←> 42 раза подряд; {h 10} означает нажатие клавиши <h> 10 раз подряд.

Замечание

Метод SendKeys не может быть использован для посылки нажатий клавиш для приложений, которые не были разработаны специально для запуска в Microsoft Windows (например, для приложений MS-DOS).

Примеры, иллюстрирующие использование SendKeys, приведены в листингах 1.13, 2.31, 2.32.

Работа с ярлыками

Свойства и методы для работы с ярлыками Windows предоставляют два объекта WSH: `WshShortcut` и `WshUrlShortcut`.

Объект *WshShortcut*

С помощью объекта `WshShortcut` можно создать новый ярлык Windows или изменить свойства уже существующего ярлыка. Этот объект можно создать только с помощью метода `CreateShortcut` объекта `WshShell`. В листинге 1.19 представлен пример сценария, в котором создается ярлык на этот самый сценарий (ярлык будет находиться в текущем каталоге).

Листинг 1.19. Создание ярлыка на выполняемый сценарий

```
    /*****************************************************************
    /*                                                          Имя:
MakeShortcut1.js                                          */
    /*                                                         Язык:
JScript                                                  */
    /*     Описание:     Создание     ярлыка     на     выполняемый
сценарий             */
    /*****************************************************************
    var WshShell,oShellLink;
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    //Создаем ярлык в текущем каталоге
    oShellLink = WshShell.CreateShortcut("Current Script.lnk");
    //Устанавливаем путь к файлу
    oShellLink.TargetPath = WScript.ScriptFullName;
    //Сохраняем ярлык
    oShellLink.Save();
    /*************                                            Конец
*******************************************/
```

Свойства объекта `WshShortcut` описаны в табл. 1.16.

Таблица 1.16. *Свойства объекта* `WshShortcut`

Свойство	Описание
`Arguments`	Содержит строку, задающую параметры командной

	строки для ярлыка
`Description`	Содержит описание ярлыка
`FullName`	Содержит строку с полным путем к ярлыку
`HotKey`	Задает "горячую" клавишу для ярлыка, т.е. определяет комбинацию клавиш, с помощью которой можно запустить или сделать активной программу, на которую указывает заданный ярлык
`IconLocation`	Задает путь к значку ярлыка
`TargetPath`	Устанавливает путь к файлу, на который указывает ярлык
`WindowStyle`	Определяет вид окна для приложения, на которое указывает ярлык
`WorkingDirectory`	Задает рабочий каталог для приложения, на которое указывает ярлык

Приведем необходимые пояснения и примеры использования свойств объекта `WshShortcut`.

Свойство *Arguments*

В листинге 1.20 приведен пример сценария, создающего ярлык на этот самый сценарий с двумя параметрами командной строки.

Листинг 1.20. Создание ярлыка на выполняемый сценарий с аргументами командной строки

```
    /*****************************************************************
    /*                                                          Имя:
MakeShortcut2.js                                            */
    /*                                                         Язык:
JScript                                                   */
    /*   Описание:   Создание   ярлыка   на   выполняемый   сценарий
с          */
    /*                                        аргументами    командной
строки                          */
    /*****************************************************************
    var WshShell,oShellLink;
    //Создаем объект WshShell
```

```
    WshShell = WScript.CreateObject("WScript.Shell");
    //Создаем ярлык в текущем каталоге
    oShellLink = WshShell.CreateShortcut("Current Script.lnk");
    //Устанавливаем путь к файлу
    oShellLink.TargetPath = WScript.ScriptFullName;
    //Указываем аргументы командной строки
    oShellLink.Arguments = "-a abc.txt";
    //Сохраняем ярлык
    oShellLink.Save();
    /*************                                                    Конец
*****************************************/
```

Свойство *HotKey*

Для того чтобы назначить ярлыку "горячую" клавишу, необходимо в свойство `HotKey` записать строку, содержащую названия нужных клавиш, разделенные символом "+".

Замечание

"Горячие" клавиши могут быть назначены только ярлыкам, которые расположены на рабочем столе Windows или в меню Пуск (Start). Для того чтобы нажатия "горячих" клавиш срабатывали, необходимо, чтобы языком по умолчанию в операционной системе был назначен английский.

В следующем примере (листинг 1.21) на рабочем столе создается ярлык для Блокнота, которому назначается комбинация "горячих" клавиш <Ctrl>+ +<Alt>+<D>.

Листинг 1.21. Создание ярлыка на Блокнот с комбинацией "горячих" клавиш

```
    /*****************************************************************
    /*                                                          Имя:
MakeShortcut3.js                                              */
    /*                                                          Язык:
JScript                                                      */
    /*  Описание:  Создание  ярлыка  на  Блокнот  с  комбинацией
горячих    */
```

```
    /*
клавиш                                                          */
    /*****************************************************************
    var WshShell,strDesktop,oMyShortcut;
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    //Определяем путь к рабочему столу
    strDesktop = WshShell.SpecialFolders("Desktop");
    //Создаем ярлык в текущем каталоге
    oMyShortcut                                                     =
WshShell.CreateShortcut(strDesktop+"\\a_key.lnk");
    //Устанавливаем путь к файлу
    oMyShortcut.TargetPath =
     WshShell.ExpandEnvironmentStrings("%windir%\\notepad.exe");
    //Назначаем комбинацию горячих клавиш
    oMyShortcut.Hotkey = "CTRL+ALT+D";
    //Сохраняем ярлык
    oMyShortcut.Save();
    WScript.Echo("Горячие        клавиши        для        ярлыка:
"+oMyShortcut.Hotkey);
    /*************                                              Конец
*****************************************/
```

Свойство *IconLocation*

Для того чтобы задать значок для ярлыка, необходимо в свойство `IconLocation` записать строку следующего формата: *"путь, индекс"*. Здесь параметр *путь* определяет расположение файла, содержащего нужный значок, а параметр *индекс* — номер этого значка в файле (номера начинаются с нуля).

В следующем примере (листинг 1.22) создается ярлык на выполняющийся сценарий с первым значком (индекс 0) из файла notepad.exe.

Листинг 1.22. Создание ярлыка на выполняемый сценарий со значком из notepad.exe

```
    /*****************************************************************
    /*                                                          Имя:
MakeShortcut4.js                                             */
    /*                                                          Язык:
JScript                                                     */
```

```
    /* Описание: Создание ярлыка на выполняемый сценарий с иконкой */
    /* из notepad.exe */
    /*****************************************************************
    var WshShell,oShellLink;
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    //Создаем ярлык в текущем каталоге
    oShellLink = WshShell.CreateShortcut("Current Script.lnk");
    //Устанавливаем путь к файлу
    oShellLink.TargetPath = WScript.ScriptFullName;
    //Выбираем иконку из файла notepad.exe
    oShellLink.IconLocation = "notepad.exe, 0";
    //Сохраняем ярлык
    oShellLink.Save();
    /************* Конец ********************************************/
```

Свойство *WindowStyle*

Значением свойства `WindowStyle` является целое число `intWindowStyle`, которое может принимать значения, приведенные в табл. 1.17.

Таблица 1.17. *Значения параметра* `intWindowStyle`

`IntWindowStyle`	**Описание**
1	Стандартный размер окна. Если окно было минимизировано или максимизировано, то будут восстановлены его первоначальные размеры и расположение на экране
3	Окно при запуске приложения будет развернуто на весь экран (максимизировано)
7	Окно при запуске приложения будет свернуто в значок (минимизировано)

Свойство *WorkingDirectory*

В следующем примере (листинг 1.23) создается ярлык для Блокнота, причем в качестве рабочего каталога указан корневой каталог диска C:.

Листинг 1.23. Создание ярлыка на Блокнот с комбинацией горячих клавиш

```
    /*****************************************************************
    /*                                                            Имя:
MakeShortcut5.js                                              */
    /*                                                           Язык:
JScript                                                     */
    /*  Описание:  Создание  ярлыка  на  Блокнот  с  изменением
рабочего    */
    /*
каталога                                                  */
    /*****************************************************************
    var WshShell,oShellLink;
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    //Создаем ярлык в текущем каталоге
    oShellLink = WshShell.CreateShortcut("Notepad.lnk");
    //Устанавливаем путь к файлу
    oShellLink.TargetPath = "notepad.exe";
    //Назначаем рабочий каталог
    oShellLink.WorkingDirectory = "c:\\";
    //Сохраняем ярлык
    oShellLink.Save();
    /*************                                              Конец
*******************************************/
```

Объект `WshShortcut` имеет единственный метод `Save`, который сохраняет заданный ярлык в каталоге, указанном в свойстве `FullName`.

Объект *WshUrlShortcut*

С помощью объекта `WshUrlShortcut` можно создать новый ярлык для сетевых ресурсов или изменить свойства уже существующего ярлыка. Этот объект, как и `WshShortcut`, можно создать только с помощью метода `CreateShortcut` объекта `WshShell`.

В следующем примере (листинг 1.24) создается сетевой ярлык для сайта www.microsoft.com.

Листинг 1.24. Создание сетевого ярлыка

```
    /*****************************************************************
```

```
    /*                                                          Имя:
MakeShortcut6.js                                                */
    /*                                                          Язык:
JScript                                                         */
    /*      Описание:      Создание      сетевого      ярлыка      для
www.microsoft.com      */
    /************************************************************
    var WshShell,oUrlLink;
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    //Создаем ярлык в текущем каталоге
    oUrlLink = WshShell.CreateShortcut("Microsoft Web Site.URL");
    //Устанавливаем путь к сайту
    oUrlLink.TargetPath = "http://www.microsoft.com";
    //Сохраняем ярлык
    oUrlLink.Save();
    /*************                                               Конец
*****************************************/
```

Объект `WshUrlShortcut` имеет два свойства: `FullName` и `TargetPath`, которые полностью аналогичны одноименным свойствам рассмотренного выше объекта `WshShortcut`.

Также у объекта `WshUrlShortcut` имеется метод `Save`, с помощью которого ярлык сохраняется в каталоге, указанном в свойстве `FullName`.

Другие примеры работы с ярлыками с помощью объекта `WshShortcut` приведены в *главе 2* (см. листинги 2.43 и 2.44).

Запуск процессов на локальной и удаленной машине

Из сценариев WSH 5.6 можно на локальной машине запускать дочерние процессы, имея при этом доступ к их стандартным входным/выходным потокам и контролируя ход выполнения этих процессов. Для этих целей предназначен объект `WshScriptExec`.

Кроме этого, имеется возможность запустить сценарий, файл с которым находится на локальной машине, на другой удаленной машине. Для выполнения сценариев на удаленных машинах и обработки ошибок, возникающих в таких сценариях, используются объекты `WshController`, `WshRemote` и `WshRemoteError`.

Объект *WshScriptExec*

В WSH 5.6 появилась возможность при помощи метода `WshShell.Exec` запускать консольное приложение или сценарий как *дочерний процесс* выполняемого сценария, т.е. с теми же переменными среды, что и у процесса-родителя. Метод `WshShell.Exec` выполняет командную строку, указанную в качестве его параметра, и возвращает объект `WScriptExec`, свойства и методы которого предоставляют информацию о запущенной задаче и обеспечивают доступ к ее стандартным потокам ввода/вывода и ошибок (обработка этих потоков необходима в силу того, что непосредственно на экране строки, выводимые дочерним приложением, не появляются).

Отметим также, что с помощью метода `WshShell.Exec` можно запускать и графические оконные Windows-приложения. В этом случае создаваемый объект `WshScriptExec` полезен тем, что он позволяет получить идентификатор запущенного процесса (Process ID, PID), который затем можно использовать для активизации задачи при помощи метода `WshShell.AppActivate`.

Объект `WScriptExec` имеет единственный метод `Terminate`, с помощью которого можно прервать выполнение дочернего процесса.

Например:

```
var WshShell=WScript.CreateObject("WScript.Shell");
var ChildJob = WshShell.Exec("cscript ChildScript.js");
ChildJob.Terminate();
```

Метод `Terminate` пытается закрыть приложение, посылая ему сообщение `WM_CLOSE`. Если это не срабатывает, задача завершается принудительно. Методом `Terminate` нужно пользоваться только в крайнем случае, т.к. некоторые приложения, завершенные таким способом, не полностью освобождают ресурсы. Поэтому, как правило, лучше дождаться, когда запущенная задача сама закончит свою работу.

Свойства объекта `WshScriptExec` описаны в табл. 1.18.

Таблица 1.18. *Свойства объекта `WshScriptExec`*

Свойство	Описание
`ExitCode`	Содержит код выхода, устанавливаемый дочерней задачей при завершении выполнения
`ProcessID`	Содержит идентификатор процесса (ProcessID, PID), которому соответствует объект `WshScriptExec`
`Status`	Содержит информацию о ходе выполнения дочерней задачи
`StdOut`	Позволяет сценарию-родителю считывать информацию из стандартного выходного потока запущенной дочерней задачи
`StdIn`	Позволяет сценарию-родителю записывать информацию в стандартный входной поток запущенной дочерней задачи
`StdErr`	Позволяет сценарию-родителю считывать информацию из стандартного потока ошибок запущенной дочерней задачи

Свойство *ProcessID*

В следующем примере (сценарий MakeCalc.js) свойство `ProcessID` используется для активизации стандартного калькулятора Windows. Напомним, что для этой цели также можно при вызове метода `WshShell.AppActivate` использовать название окна "Calculator".

Листинг 1.25. Активизация приложений с помощью PID

```
    /*****************************************************************
    /*                                                          Имя:
MakeCalc.js                                                  */
    /*                                                         Язык:
JScript                                                     */
    /*     Описание:     Активизация     приложений     с     помощью
```

```
PID                */
    /*************************************************************
    var WshShell, theCalculator;
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    //Запускаем калькулятор
    theCalculator = WshShell.Exec("calc");
    //Приостанавливаем выполнение сценария для того, чтобы окно
    //калькулятора появилось на экране
    WScript.Sleep(500);
    //Активизируем окно калькулятора
    WshShell.AppActivate(theCalculator.ProcessID);
    //Посылаем нажатия клавиш в окно калькулятора
    WshShell.SendKeys("1{+}");
    WScript.Sleep(500);
    WshShell.SendKeys("2");
    WScript.Sleep(500);
    WshShell.SendKeys("~");
    /*************                                              Конец
*****************************************/
```

Свойство *Status*

После запуска дочернего процесса сценарий-родитель продолжает выполняться асинхронно, поэтому необходимо уметь определять, выполняется ли еще запущенная задача, или она уже завершена. Для этой цели используется свойство `Status`: если значение `Status` равно 0, то это означает, что дочерний процесс находится в стадии выполнения, если `Status` равно 1, то запущенная задача уже завершена. Например, в результате выполнения приведенного в листинге 1.26 сценария ChildStatus.js на экран выведется несколько строк "Команда еще выполняется" (рис. 1.12).

Листинг 1.26. Контроль состояния дочернего процесса

```
    /*************************************************************
    /*                                                      Имя:
ChildStatus.js                                          */
    /*                                                      Язык:
JScript                                                 */
    /*      Описание:      Контроль      состояния      дочернего
процесса              */
```

```
    /*****************************************************************
    var WshShell,theJob;
    //Создаем объект WshShell
    WshShell=WScript.CreateObject("WScript.Shell");
    //Запускаем дочернее приложение
    theJob = WshShell.Exec("xcopy /?");
    for (;;) {
       if  (theJob.status==1)  //Проверяем  завершение  дочернего
процесса
      break;  //Выходим из цикла
     else WScript.Echo("Команда еще выполняется");
    }
    WScript.Echo("Выполнение завершено");
    /*************                                          Конец
*****************************************/
```

Рис. 1.12. Результат выполнения сценария ChildStatus.js

Свойства *StdOut, StdIn* и *StdErr*

Работать с потоками `StdOut`, `StdIn` и `StdErr` объекта `WshScriptExec` можно с помощью тех же методов, которые применяются в объекте `WScript` для доступа к соответствующим стандартным потокам (см. табл. 1.3). Например, запустив приведенный в листинге 1.27 сценарий ConToWin.js с помощью wscript.exe, мы выведем в графическое окно информацию о ключах программы cscript.exe (рис. 1.13).

Рис. 1.13. Результат выполнения сценария ConToWin.js

Отметим, что запускаемое консольное приложение cscript.exe выводит символы кириллицы в DOS-кодировке, поэтому для вывода таких символов в графическое окно их нужно преобразовать в Windows-кодировку. В рассматриваемом сценарии это делается с помощью функции DosToWin, которая преобразует переданную в качестве параметра строку следующим образом: все символы кириллицы в этой строке переводятся в Windows-кодировку, остальные символы остаются без изменений:

```
    function DosToWin(s) {
     var i,ss;  //Объявляем переменные
     //Проверяем, создан ли объект RusDict
     if (typeof(RusDict)=="undefined")
      //Если объект RusDict не создан, создаем его
      MakeRusDict();
     ss="";
     for (i=0;i<s.length;i++) {  //Цикл по всем символам в строке
      if (RusDict.Exists(s.charAt(i)))  //Проверяем наличие символа
в словаре
       //Преобразуем i-й символ в Windows-кодировку
       ss+=RusDict.Item(s.charAt(i));
      else ss+=s.charAt(i);
     }
     return ss;
    }
```

Основным в функции `DosToWin` является использование объекта `Dictionary` с именем `RusDict`. Этот объект формируется в функции `MakeRusDict` и содержит пары "ключ"–"значение" для всех букв русского алфавита, причем в качестве ключа указывается буква в DOS-кодировке, а в качестве значения — символ с кодом, который соответствует этой букве в

Windows-кодировке.

```
    //Функция для создания объекта Dictionary с парами "ключ-значение", где
    //"ключ"-буква в DOS-кодировке, "значение"- символ, соответствующий этой
    //букве в Windows-кодировке
    function MakeRusDict() {
     //Создаем объект Dictionary
     RusDict = WScript.CreateObject("Scripting.Dictionary");
      //Заполняем пары "ключ" (символ в DOS-кодировке)- "значение" (символ в
     //Window-кодировке) для всех букв русского алфавита
     RusDict.add("Ђ", "А"); RusDict.add("Ѓ", "Б"); RusDict.add(",", "В");
     RusDict.add("ѓ", "Г"); RusDict.add("„", "Д"); RusDict.add("…", "Е");
     RusDict.add("р", "Ё"); RusDict.add("†", "Ж"); RusDict.add("‡", "З");
     RusDict.add("€", "И"); RusDict.add("‰", "Й"); RusDict.add("Љ", "К");
     RusDict.add("‹", "Л"); RusDict.add("Њ", "М"); RusDict.add("Ќ", "Н");
     RusDict.add("Ћ", "О"); RusDict.add("Џ", "П"); RusDict.add("ђ", "Р");
     RusDict.add("‘", "С"); RusDict.add("’", "Т"); RusDict.add("“", "У");
     RusDict.add("”", "Ф"); RusDict.add("•", "Х"); RusDict.add("–", "Ц");
      RusDict.add("—", "Ч"); RusDict.add("", "Ш"); RusDict.add("™", "Щ");
     RusDict.add("љ", "Ъ"); RusDict.add("›", "Ы"); RusDict.add("њ", "Ь");
     RusDict.add("ќ", "Э"); RusDict.add("ћ", "Ю"); RusDict.add("џ", "Я");
     RusDict.add(" ", "а"); RusDict.add("Ў", "б"); RusDict.add("ў", "в");
     RusDict.add("J", "г"); RusDict.add("¤", "д"); RusDict.add("Ґ", "е");
     RusDict.add("с", "ё"); RusDict.add("¦", "ж"); RusDict.add("§", "з");
     RusDict.add("Ё", "и"); RusDict.add("©", "й"); RusDict.add("Є", "к");
      RusDict.add("«", "л"); RusDict.add("¬", "м"); RusDict.add("", "н");
     RusDict.add("®", "о"); RusDict.add("Ї", "п"); RusDict.add("а", "р");
     RusDict.add("б", "с"); RusDict.add("в", "т"); RusDict.add("г",
```

```
"у");
     RusDict.add("д", "ф"); RusDict.add("е", "х"); RusDict.add("ж",
"ц");
     RusDict.add("з", "ч"); RusDict.add("и", "ш"); RusDict.add("й",
"щ");
     RusDict.add("к", "ъ"); RusDict.add("л", "ы"); RusDict.add("м",
"ь");
     RusDict.add("н", "э"); RusDict.add("о", "ю"); RusDict.add("п",
"я");
    }
```

Листинг 1.27. Доступ к потоку StdOut дочернего процесса

```
    /*****************************************************************
    /*                                                          Имя:
ConToWin.js                                                  */
    /*                                                         Язык:
JScript                                                    */
    /* Кодировка: DOS                                             */
    /*    Описание:    Доступ    к    потоку    StdOut    дочернего
процесса           */
    /*****************************************************************
    var WshShell,theJob,s,IsBreak,RusDict;  //Объявляем переменные
    //Функция  для  создания  объекта  Dictionary  с  парами  "ключ-
значение", где
    //"ключ"-буква    в    DOS-кодировке,    "значение"-    символ,
соответствующий этой
    //букве в Windows-кодировке
    function MakeRusDict() {
     //Создаем объект Dictionary
     RusDict = WScript.CreateObject("Scripting.Dictionary");
     //Заполняем пары "ключ" (символ в DOS-кодировке)- "значение"
(символ в
     //Window-кодировке) для всех букв русского алфавита
     RusDict.add("Ђ", "А"); RusDict.add("Ѓ", "Б"); RusDict.add(",",
"В");
     RusDict.add("ѓ", "Г"); RusDict.add("„", "Д"); RusDict.add("…",
"Е");
     RusDict.add("р", "Ё"); RusDict.add("†", "Ж"); RusDict.add("‡",
"З");
     RusDict.add("€", "И"); RusDict.add("‰", "Й"); RusDict.add("Љ",
"К");
     RusDict.add("‹", "Л"); RusDict.add("Њ", "М"); RusDict.add("Ќ",
"Н");
     RusDict.add("Ћ", "О"); RusDict.add("Џ", "П"); RusDict.add("ђ",
```

```
"Р");
     RusDict.add("‘", "С"); RusDict.add("’", "Т"); RusDict.add("“",
"У");
     RusDict.add("”", "Ф"); RusDict.add("•", "Х"); RusDict.add("–",
"Ц");
     RusDict.add("—", "Ч"); RusDict.add("", "Ш"); RusDict.add("™",
"Щ");
     RusDict.add("љ", "Ъ"); RusDict.add("›", "Ы"); RusDict.add("њ",
"Ь");
     RusDict.add("ќ", "Э"); RusDict.add("ћ", "Ю"); RusDict.add("џ",
"Я");
     RusDict.add(" ", "а"); RusDict.add("Ў", "б"); RusDict.add("ў",
"в");
     RusDict.add("J", "г"); RusDict.add("¤", "д"); RusDict.add("Ґ",
"е");
     RusDict.add("с", "ё"); RusDict.add("¦", "ж"); RusDict.add("§",
"з");
     RusDict.add("Ё", "и"); RusDict.add("©", "й"); RusDict.add("Є",
"к");
     RusDict.add("«", "л"); RusDict.add("¬", "м"); RusDict.add("",
"н");
     RusDict.add("®", "о"); RusDict.add("Ї", "п"); RusDict.add("а",
"р");
     RusDict.add("б", "с"); RusDict.add("в", "т"); RusDict.add("г",
"у");
     RusDict.add("д", "ф"); RusDict.add("е", "х"); RusDict.add("ж",
"ц");
     RusDict.add("з", "ч"); RusDict.add("и", "ш"); RusDict.add("й",
"щ");
     RusDict.add("к", "ъ"); RusDict.add("л", "ы"); RusDict.add("м",
"ь");
     RusDict.add("н", "э"); RusDict.add("о", "ю"); RusDict.add("п",
"я");
    }
    //Функция для перевода строки из DOS- в Windows-кодировку
    function DosToWin(s) {
     var i,ss;  //Объявляем переменные
     //Проверяем, создан ли объект RusDict
     if (typeof(RusDict)=="undefined")
      //Если объект RusDict не создан, создаем его
      MakeRusDict();
     ss="";
     for (i=0;i<s.length;i++) {  //Цикл по всем символам в строке
      if (RusDict.Exists(s.charAt(i)))  //Проверяем наличие символа
в словаре
       //Преобразуем i-й символ в Windows-кодировку
       ss+=RusDict.Item(s.charAt(i));
```

```
      else ss+=s.charAt(i);
     }
     return ss;
    }
    /*************                                                Начало
*****************************************/
    //Создаем объект WshShell
    WshShell=WScript.CreateObject("WScript.Shell");
    //Запускаем дочернее приложение
    theJob = WshShell.Exec("cscript");
    IsBreak=false;
    for (;;) {
     if (!theJob.StdOut.AtEndOfStream)
      //Считываем всю информацию, находящуюся в потоке StdOut
      //дочернего процесса
      s+=theJob.StdOut.ReadAll();
     if (IsBreak) break;
       if  (theJob.status==1)  //Проверяем,  не  завершилась  ли
запущенная задача
      IsBreak=true;
     else WScript.Sleep(100);
    }
    //Преобразуем сформированные строки в Windows-кодировку
    //и выводим их на экран
    WScript.Echo(DosToWin(s));
    /*************                                                Конец
*****************************************/
```

Таким образом, можно с помощью метода Exec запустить утилиту командной строки, передавать ей нужную входную информацию с помощью свойства StdIn и с помощью свойства StdOut получать и анализировать выдаваемые этой утилитой строки (соответствующие примеры приведены также в листингах 2.37 и 2.38).

Объект *WshController*

Объект `WshController` имеет единственный метод `CreateScript` и предназначен для создания объекта-сценария на удаленной машине.

> **Замечание**
>
> В силу соображений безопасности удаленные сценарии можно запускать только с машин, на которых установлена операционная система Windows NT/2000/XP; то же самое требование предъявляется к машинам, на которых должны

выполняться такие сценарии. Кроме этого, после начальной установки WSH по умолчанию выполнение удаленных сценариев запрещено; действия, которые необходимо произвести для разрешения выполнения таких сценариев, описаны в *главе 2*.

Создается объект WshController следующим образом:

```
var WshController=WScript.CreateObject("WshController");
```

Замечание

Обратите внимание, что для объекта `WshController` программным идентификатором (ProgID) является именно строка "`WshController`", а не строка "`WScript.WshController`", как указано в бета-версии документации на WSH 5.6.

Метод `CreateScript` возвращает указатель на объект `WshRemote`, с помощью которого можно контролировать состояние удаленного сценария и управлять его выполнением. При выполнении этого метода WSH последовательно производит следующие действия:

□ подготавливает файл со сценарием для пересылки на удаленную станцию;

□ с помощью протокола DCOM создает экземпляр объекта `WshRemote` на удаленной машине;

□ пересылает сценарий на удаленную станцию для последующего выполнения с помощью метода `Execute` объекта `WshRemote`.

Синтаксис метода `CreateScript`:

`CreateScript(`*`CommandLine, [MachineName]`*`)`

Параметр *`CommandLine`* содержит командную строку для запуска сценария, который должен быть выполнен на удаленной станции. Путь к сценарию должен быть указан относительно локальной машины, а не относительно той станции, на которой должен выполняться сценарий. Чаще всего этот путь указывает на файл, находящийся непосредственно на локальном компьютере, хотя можно указать путь к сценарию, записанному на каком-либо общем сетевом ресурсе (это делает возможным, работая на одном компьютере, загружать сценарий с другого и выполнять этот сценарий на третьем компьютере в сети).

Второй необязательный параметр *`MachineName`* задает имя компьютера, на котором должен быть запущен сценарий. Если этот параметр не указан,

то сценарий будет выполняться на локальной машине.

Объект *WshRemote*

Объект WshRemote необходим для контроля состояния сценариев, которые запущены на удаленной машине. В результате запуска такого сценария на удаленной машине создается процесс, поэтому можно сказать, что экземпляром объекта WshRemote, соответствующего выполняющемуся сценарию, является процесс. Создается WshRemote с помощью метода CreateScript объекта WshController:

```
    var Controller,RemoteScript;
    Controller=WScript.CreateObject("WshController");
    RemoteScript=Controller.CreateScript("d: WscriptsWMyScript.js",
"Server1");
```

Объект WshRemote имеет два свойства: Error и Status.

В свойстве Error хранится ссылка на объект WshRemoteError, который содержит информацию об ошибке, приведшей к аварийному завершению работы удаленного сценария.

Числовое свойство Status позволяет определить состояние сценария, работающего асинхронно на удаленной машине. Возможные значения свойства Status приведены в табл. 1.19.

Таблица 1.19. *Значения параметра* `Status`

Значение	Числовое значение	Описание
`NoTask`	0	Объект `WshRemote`, соответствующий удаленному сценарию, создан, однако сценарий еще не запущен
`Running`	1	Выполнение удаленного сценария продолжается
`Finished`	2	Удаленный сценарий завершен

Два имеющихся у объекта WshRemote метода позволяют соответственно запустить удаленный сценарий (метод Execute) или принудительно завершить его выполнение (метод Terminate); оба эти метода не имеют параметров. Метод Terminate, подобно одноименному методу объекта WshScriptExec, пытается закрыть приложение, посылая ему сообщение WM_CLOSE (если это не срабатывает, процесс завершается принудительно).

Кроме свойств и методов, объект WshRemote может генерировать три

события, которые описаны в табл. 1.20.

Таблица 1.20. *События объекта* `WshRemote`

Событие	Описание
`Start`	Возникает при вызове метода `Execute` и сигнализирует серверу сценариев о начале выполнения сценария на удаленной машине
`Error`	Возникает в том случае, когда выполнение сценария на удаленной машине завершается аварийно
`End`	Возникает при завершении (нормальном или аварийном) работы сценария на удаленной машине

Для обработки в сценариях событий, приведенных в табл. 1.20, необходимо подключиться к объекту WshRemote с помощью метода ConnectObject объекта WScript (листинг 1.28).

Листинг 1.28. Обработка событий объекта WshRemote (JScript)

```
    /*****************************************************************
    /*                                                              Имя:
RemoteEvents.js                                                    */
    /*                                                              Язык:
JScript                                                           */
    /*  Описание:  Обработка  событий,  возникающих  при  выполнении
удаленного */
    /*
сценария                                                    */
    /*****************************************************************
    Var Controller,RemScript,IsQuit;  //Объявляем переменные
    //Создаем объект WshController
    Controller = WScript.CreateObject("WshController");
    //Создаем сценарий на удаленной машине (объект WshRemote)
    RemScript   =   Controller.CreateScript("D:\RemoteScript.js   ",
"stand");
    //Устанавливаем соединение с объектом WshRemote
    WScript.ConnectObject(RemScript, "RemoteScript_");
    RemScript.Execute();  //Запускаем удаленный сценарий
    IsQuit = False;
    while   (!IsQuit)   WScript.Sleep(100);      //Приостанавливаем
сценарий на 0,1 сек
    WScript.Quit();  //Выходим из сценария
    /***************              Функции-обработчики       событий
```

```
**********************/
    function RemoteScript_End { //Событие End
     WScript.Echo("Выполнение удаленного сценария завершено");
     IsQuit = True;
    }
    function RemoteScript_Error { //Событие Error
     //Выводим на экран описание возникшей ошибки
     WScript.Echo("Ошибка при выполнении удаленного сценария: " +
      RemScript.Error.Description);
     IsQuit = True;
    }
    function RemoteScript_Start { //Событие Start
     WScript.Echo("Удаленный сценарий запущен");
    }
    /*************                                          Конец
******************************************/
```

Объект *WshRemoteError*

Объект `WshRemoteError` создается автоматически при возникновении ошибки во время выполнения сценария на удаленной машине и содержит информацию об этой ошибке. Ссылка на объект `WshRemoteError` хранится в свойстве `Error` соответствующего объекта `WshRemote`.

Свойства объекта `WshRemoteError` описаны в табл. 1.21 (методов у этого объекта нет).

Таблица 1.21. *Свойства объекта* `WshRemoteError`

Свойство	Описание
`Description`	Содержит краткое описание ошибки, которая привела к аварийному завершению работы сценария. Если для какой-либо ошибки описание не предусмотрено, `Description` содержит пустую строку
`Line`	Определяет номер строки в файле сценария, в которой произошла ошибка. Если для ошибки нельзя определить номер строки, в которой она произошла, в свойство `Line` записывается 0
`Character`	Определяет номер символа в строке, в котором произошла ошибка. Если для ошибки нельзя определить точную позицию, в которой она возникла, в свойство `Character` записывается 0

`Number`	Содержит числовой код ошибки
`SourceText`	Содержит в текстовом виде строку сценария, в которой возникла ошибка. Так как не всегда возможно точно определить строку, в которой произошла ошибка, то иногда значением свойства `SourceText` может быть пустая строка
`Source`	Содержит в символьном виде название COM-объекта, обращение к которому послужило источником ошибок

Для получения информации о возникшей при выполнении удаленного сценария ошибке можно использовать обработчик события `Error` объекта `WshRemote`; соответствующие примеры приведены в листингах 1.28, 2.56 и 2.57.

Глава 2
Примеры использования стандартных объектов WSH (JScript и VBScript)

В этой главе мы на примерах подробно рассмотрим, как с помощью стандартных объектов WSH 5.6, описание которых приведено в *главе 1*, можно решать некоторые практические задачи, связанные, в частности, с выводом и вводом текстовой информации, запуском других приложений (как на локальной, так и на удаленной машине), созданием ярлыков в различных папках, работой с системным реестром и локальной сетью. Практически все сценарии приведены как на языке JScript, так и на VBScript, и снабжены подробными комментариями.

Вывод на экран текстовых строк

Сформированные в сценарии строки текста можно выводить в стандартный выходной поток (в консольном режиме) или в графическое диалоговое окно несколькими способами:

- с помощью метода `Echo` объекта `WScript`;
- с помощью методов `Write` и `WriteLine` объекта `WScript.StdOut`;
- с помощью функции `MsgBox` языка VBScript;
- с помощью метода `Popup` объекта `WshShell`.

Метод *Echo* объекта *WScript*

Примеры использования метода `WScript.Echo` в сценариях, написанных на языках JScript и VBScript, представлены соответственно в листингах 2.1 и 2.2.

Замечание

Для корректного отображения с помощью метода Echo символов кириллицы, эти символы должны быть представлены в Windows-кодировке (CP 1251).

Листинг 2.1. Вывод строк в Win-кодировке с помощью метода `WScript.Echo` (JScript)

```
    /*****************************************************************
    /*                                                          Имя:
Echo1.js                                                    */
    /*                                                         Язык:
JScript                                                    */
    /*       Описание:       Пример       использования       метода
WScript.Echo              */
    /*****************************************************************
    //Печатаем строку текста (кириллица)
    WScript.Echo("Использование метода Echo (Win-кодировка)");
    //Печатаем строку текста и результат вычислений
    WScript.Echo("Например, 1+2=",1+2);
```

```
    /*************                                                    Конец
*****************************************/
```

Листинг 2.2. Вывод строк в Win-кодировке с помощью метода WScript.Echo (VBScript)

```
    '*************************************************************
    ' Имя: Echo1.vbs
    ' Язык: VBScript
    ' Описание: Пример использования метода WScript.Echo
    '*************************************************************
    ' Печатаем строку текста (кириллица)
    WScript.Echo "Использование метода Echo (Win-кодировка)"
    ' Печатаем строку текста и результат вычислений
    WScript.Echo "Например, 1+2=",1+2
    '*************                                                  Конец
*****************************************
```

Если сценарий Echo1.js (Echo1.vbs) был запущен с помощью cscript.exe, то строки выводятся в командное окно (рис. 2.1).

Если же этот сценарий выполнялся с помощью wscript.exe, то строки по очереди выводятся в диалоговые окна с единственной кнопкой **OK** (рис. 2.2).

Часто бывает необходимо выводить в диалоговое окно не по одной строке текста, а сразу несколько таких строк (рис. 2.3). Для этого нужно формировать строки, содержащие символы перевода строки: escape-последовательность `"\n"` для JScript и предопределенная именованная константа `vbCrLf` для VBScript (соответствующие примеры сценариев приведены в листингах 2.3 и 2.4).

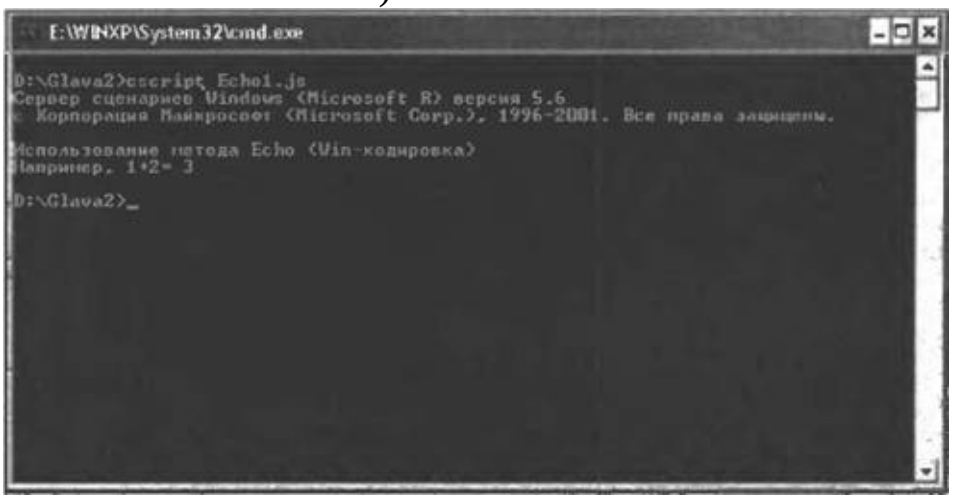


Рис. 2.1. Результат выполнения Echo1.js с помощью cscript.exe

Рис. 2.2. Результат выполнения Echo1.js с помощью wscript.exe

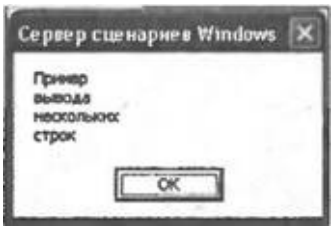


Рис. 2.3. Диалоговое окно с несколькими строками текста

Листинг 2.3. Вывод в диалоговое окно нескольких строк (JScript)

```
    /*****************************************************************
    /*                                                          Имя:
Echo2.js                                                    */
    /*                                                         Язык:
JScript                                                    */
    /*      Описание:      Вывод      сразу      нескольких      строк
(WScript.Echo)           */
    /*****************************************************************
    var s; //Объявляем переменную
    s="Пример\nвывода\nнескольких\nстрок"; //Формируем строки
    WScript.Echo(s); //Печатаем строки
    /*************                                             Конец
*******************************************/
```

Листинг 2.4. Вывод в диалоговое окно нескольких строк (VBScript)

```
    '*****************************************************************
    ' Имя: Echo2.vbs
    ' Язык: VBScript
    ' Описание: Вывод сразу нескольких строк (WScript.Echo)
    '*****************************************************************
    Option Explicit
    Dim s  ' Объявляем переменную
    ' Формируем строки
```

```
    s="Пример"&vbCrLf&"вывода"&vbCrLf&"нескольких"&vbCrLf&"строк"
    WScript.Echo s  ' Печатаем строки
    '*************                                              Конец
*****************************************
```

Методы *Write* и *WriteLine* объекта *WScript.StdOut*

Для вывода строк в сценариях, выполняющихся в консольном режиме, можно использовать стандартный выходной поток `WScript.StdOut` (листинги 2.5 и 2.6). Напомним, что запускать сценарий, обращающийся к потоку `StdOut`, можно только в консольном режиме с помощью cscript.exe. Если же попробовать выполнить, например, сценарий StdOut1.js с помощью wscript.exe, то произойдет ошибка (рис. 2.4).

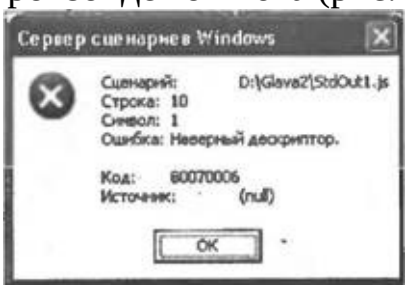


Рис. 2.4. Ошибка, возникающая при обращении к `StdOut` в графическом режиме

Листинг 2.5. Вывод строк в стандартный выходной поток (JScript)

```
    /*****************************************************************
    /*                                                          Имя:
StdOut1.js                                                    */
    /*                                                          Язык:
JScript                                                      */
    /*   Описание:   Пример   использования   методов   StdOut.Write
и          */
    /*
StdOut.WriteLine                                            */
    /*****************************************************************
    var n;  //Объявляем переменную
    n=1+2;
    //Печать без перевода строки
    WScript.StdOut.Write("Использование метода ");
    //Выводим строку с текущей позиции курсора
    WScript.StdOut.WriteLine("StdOut.WriteLine");
```

```
    //Печатаем строку и значение переменной
    WScript.StdOut.WriteLine("Например, 1+2="+n);
    /*************                                          Конец
*******************************************/
```

Листинг 2.6. Вывод строк в стандартный выходной поток (VBScript)

```
    '*****************************************************************
    ' Имя: StdOut1.vbs
    ' Язык: VBScript
    '  Описание:  Пример  использования  методов  StdOut.Write  и
StdOut.WriteLine
    '*****************************************************************
    Option Explicit
    Dim n   ' Объявляем переменную
    n=1+2
    ' Печать без перевода строки
    WScript.StdOut.Write "Использование метода "
    ' Выводим строку с текущей позиции курсора
    WScript.StdOut.WriteLine "StdOut.WriteLine"
```

Замечание

В Windows XP символы кириллицы, посылаемые из сценария в стандартный выходной поток, должны быть представлены в Windows-кодировке (CP 1251). В предыдущих версиях Windows для корректного отображения на экране символы кириллицы при использовании потока `WScript.StdOut` должны быть в DOS-кодировке (OEM 866).

Как и при использовании метода `WScript.Echo`, в качестве параметра метода `WriteLine` можно указывать строки, содержащие символы перевода строки (листинги 2.7 и 2.8).

Листинг 2.7. Вывод в `StdOut` сразу нескольких строк (JScript)

```
    /*****************************************************************
    /*                                                          Имя:
StdOu2.js                                                   */
```

```
    /*                                                          Язык:
JScript                                                       */
    /*      Описание:      Вывод      сразу      нескольких      строк
(StdOut.WriteLine)       */
    /************************************************************
    var s;  //Объявляем переменную
    s="Пример\nвывода\nнескольких\nстрок"; //Формируем строки
    WScript.StdOut.WriteLine(s);  //Выводим строки
    /*************                                               Конец
*****************************************/
```

Листинг 2.8. Вывод в `StdOut` сразу нескольких строк (VBScript)

```
    '************************************************************
    ' Имя: StdOut2.vbs
    ' Язык: VBScript
    ' Описание: Вывод сразу нескольких строк (StdOut.WriteLine)
    '************************************************************
    Option Explicit
    Dim s  ' Объявляем переменную
    ' Формируем строки
    s="Пример"&vbCrLf&"вывода"&vbCrLf&"нескольких"&vbCrLf&"строк"
    WScript.StdOut.WriteLine s  ' Выводим строки
    '*************                                               Конец
*****************************************
```

Для создания более компактного текста сценария можно сразу сохранить ссылку на стандартный выходной поток `WScript.StdOut` в отдельную переменную и затем при вызове методов `Write` и `WriteLine` использовать эту переменную (листинги 2.9 и 2.10).

Листинг 2.9. Сохранение ссылки на поток `StdOut` в переменной (JScript)

```
    /************************************************************
    /*                                                           Имя:
StdOut3.js                                                     */
    /*                                                          Язык:
JScript                                                       */
    /*       Описание:       Пример       использования       метода
StdOut.WriteLine          */
    /************************************************************
    var n,StdOut;  //Объявляем переменные
    n=1+2;
```

```
    StdOut=WScript.StdOut;     //Сохраняем ссылку на StdOut в
переменной
    //Выводим строки в StdOut
    StdOut.WriteLine("Пример          использования          метода
StdOut.WriteLine() ...");
    StdOut.WriteLine("1+2="+n);
    /*************                                              Конец
*****************************************/
```

Листинг 2.10. Сохранение ссылки на поток StdOut в переменной (VBScript)

```
    '*************************************************************
    ' Имя: StdOut3.vbs
    ' Язык: JScript
    ' Описание: Пример использования метода StdOut.WriteLine
    '*************************************************************
    Option Explicit
    Dim n,StdOut  ' Объявляем переменные
    n=1+2
    Set  StdOut=WScript.StdOut    '  Сохраняем  ссылку  на  StdOut  в
переменной
    ' Выводим строки в StdOut
    StdOut.WriteLine     "Это     пример     использования     метода
StdOut.WriteLine() ..."
    StdOut.WriteLine "1+2=" & n
    '*************                                              Конец
*****************************************
```

Функция *MsgBox* языка VBScript

В языке VBScript существует специальная функция `MsgBox`, с помощью которой можно выводить информацию в диалоговое окно с несколькими кнопками; также в этом окне можно задавать заголовок и значок (рис. 2.5).

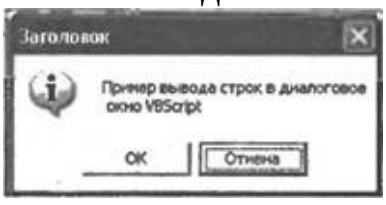


Рис. 2.5. Диалоговое окно, созданное с помощью функции `MsgBox`

Пример сценария, создающего такое диалоговое окно, приведен в

листинге 2.11.

Замечание

В языке JScript аналога функции `MsgBox` нет.

Листинг 2.11. Создание диалогового окна с помощью функции MsgBox (VBScript)

```
    '*************************************************************
    ' Имя: MsgBox.vbs
    ' Язык: VBScript
    ' Описание: Пример использования функции MsgBox
    '*************************************************************
    Dim Res,Text,Title  ' Объявляем переменные
    Text="Пример вывода строк в диалоговое" & vbCrLf & " окно
VBScript"
    Title="Заголовок"
    ' Выводим диалоговое окно на экран
    Res=MsgBox(Text,vbOkCancel+vbInformation+vbDefaultButton2,Title)
    ' Определяем, какая из кнопок была нажата в диалоговом окне
    If Res=vbOk Then
     MsgBox "Нажата кнопка OK"
    Else
     MsgBox "Нажата кнопка Отмена"
    End If
    '*************                                          Конец
*****************************************
```

Подробное описание функции `MsgBox` приведено в *приложении 1*. Здесь же мы отметим только то, что значением функции `MsgBox` является константа, соответствующая нажатой в диалоговом окне кнопки (в нашем примере такими константами являются `vbOk` и `vbCancel`). Таким образом, `MsgBox` может использоваться в сценариях для организации выбора пользователем одного из возможных вариантов, однако это не совсем удобно, т.к. надписи на кнопках нельзя задавать произвольным образом (можно указать только **OK**, **Отмена**, **Стоп**, **Повтор**, **Пропустить**, **Да** и **Нет**).

Метод *Popup* объекта *WshShell*

С помощью метода Popup (подробное описание метода приведено в *главе 1*) можно создавать такие же диалоговые окна, как и при помощи функции MsgBox, причем этот метод можно использовать как в VBScript-, так и в JScript-сценариях (листинги 2.12 и 2.13).

Листинг 2.12. Создание диалогового окна с помощью метода Popup (JScript)

```
    /*************************************************************
    /* Имя: Popup.js
   */
    /*                                                      Язык:
JScript                                                    */
    /*       Описание:       Пример       использования       метода
WshShell.Popup            */
    /*************************************************************
    var WshShell,Res,Text,Title; //Объявляем переменные
    //Инициализируем константы для диалоговых окон
    var vbOkCancel=1,vbOk=1;
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    Text="Пример вывода строк в диалоговое\nокно WScript";
    Title="Заголовок"
    //Выводим диалоговое окно на экран
    Res=WshShell.Popup(Text,0,Title,vbOkCancel);
    // Определяем, какая из кнопок была нажата в диалоговом окне
    if (Res==vbOk) WshShell.Popup("Нажата кнопка OK");
    else WshShell.Popup("Нажата кнопка Отмена");
    /*************                                          Конец
*****************************************/
```

Листинг 2.13. Создание диалогового окна с помощью метода Popup (VBScript)

```
    '*************************************************************
    ' Имя: Popup.vbs
    ' Язык: VBcript
    ' Описание: Пример использования метода WshShell.Popup
    '*************************************************************
    Option Explicit
    Dim WshShell,Res,Text,Title  ' Объявляем переменные
    ' Создаем объект WshShell
```

```
    Set WshShell = WScript.CreateObject("WScript.Shell")
    Text="Пример вывода строк в диалоговое" & vbCrLf & "окно
WScript"
    Title="Заголовок"
    ' Выводим диалоговое окно на экран
    Res=WshShell.Popup(Text,0,Title,vbOkCancel)
    ' Определяем, какая из кнопок была нажата в диалоговом окне
    If (Res=vbOk) Then
     WshShell.Popup "Нажата кнопка OK"
    Else
     WshShell.Popup "Нажата кнопка Отмена"
    End If
    '*************                                              Конец
*****************************************
```

Главным отличием метода `Popup` от функции `MsgBox` является наличие параметра *`nSecToWait`*, задающего время (в секундах), по истечении которого диалоговое окно будет автоматически закрыто. Если этот параметр равен нулю, как в приведенных выше примерах, то окно будет закрыто только после нажатия какой-либо кнопки в нем.

Ввод строк текста

Для организации в сценариях диалога с пользователем необходимо уметь принимать вводимые с клавиатуры строки текста. В консольном и графическом режимах ввод информации осуществляется по-разному: при запуске сценария с помощью cscript.exe мы имеем доступ к стандартному входному потоку `StdOut`, при использовании wscript.exe можно применять функцию `InputBox` языка VBScript.

Ввод строк в консольном режиме

Самый простой способ ввести строку в консольном режиме предоставляет метод `WScript.StdIn.ReadLine`, при использовании этого метода ввод завершается нажатием клавиши <Enter>.

Отметим, что при использовании стандартного входного потока `WScript.StdIn` в Windows XP (по крайней мере в той версии, которой пользовался автор) возникает проблема, связанная с кодировкой символов кириллицы. Дело в том, что метод `WScript.StdIn.ReadLine` возвращает строку в DOS-кодировке, а для вывода на экран с помощью методов `WScript.StdOut.WriteLine` или `WScript.Echo` строка должна быть в Windows-кодировке (в предыдущих версиях Windows метод `WScript.StdOut.WriteLine` требовал строку в DOS-кодировке). Поэтому для корректного отображения символов кириллицы на экране приходится применять дополнительные функции конвертации из DOS- в Windows-кодировку. Стандартных методов или функций, предназначенных для этой цели, в языках JScript и VBScript нет, поэтому такие функции следует написать самостоятельно.

Рассмотрим сначала написанную на JScript функцию конвертации DosToWin из листинга 2.14:

```
    function DosToWin(s) {
     var i,ss;  //Объявляем переменные
     //Проверяем, создан ли объект RusDict
     if (typeof(RusDict)=="undefined")
      //Если объект RusDict не создан, создаем его
      MakeRusDict();
     ss="";
     for (i=0;i<s.length;i++) {  //Цикл по всем символам в строке
      if (RusDict.Exists(s.charAt(i)))  //Проверяем наличие символа
в словаре
```

```
      //Преобразуем i-й символ в Windows-кодировку
      ss+=RusDict.Item(s.charAt(i));
     else ss+=s.charAt(i);
    }
    return ss;
   }
```

Как мы видим, эта функция преобразует переданную в качестве параметра строку следующим образом: все символы кириллицы в этой строке переводятся в Windows-кодировку, остальные символы остаются без изменений. Основным в функций `DosToWin` является использование объекта `Dictionary` (аналог ассоциативного массива) с именем `RusDict`. Этот объект формируется в функции `MakeRusDict` и содержит пары "ключ"–"значение" для всех букв русского алфавита, причем в качестве ключа указывается буква в DOS-кодировке, а в качестве значения — символ с кодом, который соответствует этой букве в Windows-кодировке:

```
   function MakeRusDict() {
    //Создаем объект Dictionary
    RusDict = WScript.CreateObject("Scripting.Dictionary");
     //Заполняем пары "ключ" (символ в DOS-кодировке)-"значение"
(символ в
    //Window-кодировке) для всех букв русского алфавита
    RusDict.add("Ъ", "А"); RusDict.add("Ѓ", "Б"); RusDict.add(",",
"В");
    RusDict.add("ѓ", "Г"); RusDict.add("„", "Д"); RusDict.add("…",
"Е");
    RusDict.add("р", "Ё"); RusDict.add("†", "Ж"); RusDict.add("‡",
"З");
    RusDict.add("€", "И"); RusDict.add("‰", "Й"); RusDict.add("Љ",
"К");
    RusDict.add("‹", "Л"); RusDict.add("Њ", "М"); RusDict.add("Ќ",
"Н");
    RusDict.add("Ћ", "О"); RusDict.add("Џ", "П"); RusDict.add("ђ",
"Р");
    RusDict.add("‘", "С"); RusDict.add("’", "Т"); RusDict.add("“",
"У");
    RusDict.add("”", "Ф"); RusDict.add("•", "Х"); RusDict.add("–",
"Ц");
     RusDict.add("—", "Ч"); RusDict.add("", "Ш"); RusDict.add("™",
"Щ");
    RusDict.add("љ", "Ъ"); RusDict.add("›", "Ы"); RusDict.add("њ",
"Ь");
    RusDict.add("ќ", "Э"); RusDict.add("ћ", "Ю"); RusDict.add("џ",
"Я");
    RusDict.add(" ", "а"); RusDict.add("Ў", "б"); RusDict.add("ў",
"в");
```

```
     RusDict.add("J", "г"); RusDict.add("¤", "д"); RusDict.add("Ґ",
"е");
     RusDict.add("c", "ё"); RusDict.add("¦", "ж"); RusDict.add("§",
"з");
     RusDict.add("Ё", "и"); RusDict.add("©", "й"); RusDict.add("Є",
"к");
     RusDict.add("«", "л"); RusDict.add("¬", "м"); RusDict.add("",
"н");
     RusDict.add("®", "о"); RusDict.add("Ї", "п"); RusDict.add("a",
"р");
     RusDict.add("б", "с"); RusDict.add("в", "т"); RusDict.add("г",
"у");
     RusDict.add("д", "ф"); RusDict.add("e", "х"); RusDict.add("ж",
"ц");
     RusDict.add("з", "ч"); RusDict.add("и", "ш"); RusDict.add("й",
"щ");
     RusDict.add("к", "ъ"); RusDict.add("л", "ы"); RusDict.add("м",
"ь");
     RusDict.add("н", "э"); RusDict.add("о", "ю"); RusDict.add("п",
"я");
    }
```

В функции `DosToWin` из VBScript-сценария StdIn1.vbs (листинг 2.15) реализован другой подход к переводу строки в Windows-кодировку, связанный с преобразованием ANSI-кодов символов:

```
Function DosToWin(s)
Dim i,k,ss
 ss=""
 For i=1 To Len(s)  ' Цикл по всем символам в строке
  k = Asc(Mid(s,i,1))  ' Определяем ANSI-код i-го символа
  ' Изменяем код k на код соответствующего символа в
  ' Windows-кодировке
  If (128 <= k) And (k <= 175) Then
   k=k+64
  ElseIf (224 <= k) And (k <= 239) Then
   k=k+16
  ElseIf k = 240 Then
   k=168
  ElseIf k = 241 Then
   k=184
  End If
  ss=ss+Chr(k)  ' Возвращаем преобразованную строку
 Next
 DosToWin=ss
End Function
```

Весь алгоритм этой функции состоит в вычислении по ANSI-коду буквы русского алфавита в DOS-кодировке кода символа в Windows-

кодировке, соответствующего этой букве.

Листинг 2.14. Ввод одной строки с помощью метода `StdIn.ReadLine` (JScript)

```
    /*****************************************************************
    /*                                                           Имя:
StdIn1.js                                                    */
    /*                                                          Язык:
JScript                                                    */
    /*       Описание:       Пример       использования       метода
StdIn.ReadLine            */
    /*****************************************************************
    var s,RusDict;  //Объявляем переменные
    //Функция  для  создания  объекта  Dictionary  с  парами  "ключ-
значение", где
    //"ключ"-буква    в    DOS-кодировке,    "значение"-    символ,
соответствующий этой
    //букве в Windows-кодировке
    function MakeRusDict() {
      //Создаем объект Dictionary
     RusDict = WScript.CreateObject("Scripting.Dictionary");
      //Заполняем  пары  "ключ"  (символ  в  DOS-кодировке)-"значение"
(символ в
     //Window-кодировке) для всех букв русского алфавита
     RusDict.add("Ђ", "А"); RusDict.add("Ѓ", "Б"); RusDict.add(",",
"В");
     RusDict.add("ѓ", "Г"); RusDict.add("„", "Д"); RusDict.add("…",
"Е");
     RusDict.add("р", "Ё"); RusDict.add("†", "Ж"); RusDict.add("‡",
"З");
     RusDict.add("€", "И"); RusDict.add("‰", "Й"); RusDict.add("Љ",
"К");
     RusDict.add("‹", "Л"); RusDict.add("Њ", "М"); RusDict.add("Ќ",
"Н");
     RusDict.add("Ћ", "О"); RusDict.add("Џ", "П"); RusDict.add("ђ",
"Р");
     RusDict.add("‘", "С"); RusDict.add("’", "Т"); RusDict.add("“",
"У");
     RusDict.add("”", "Ф"); RusDict.add("•", "Х"); RusDict.add("–",
"Ц");
     RusDict.add("—", "Ч"); RusDict.add("", "Ш"); RusDict.add("™",
"Щ");
     RusDict.add("љ", "Ъ"); RusDict.add("›", "Ы"); RusDict.add("њ",
"Ь");
```

```
     RusDict.add("ќ", "Э"); RusDict.add("ћ", "Ю"); RusDict.add("џ",
"Я");
     RusDict.add(" ", "а"); RusDict.add("Ў", "б"); RusDict.add("ў",
"в");
     RusDict.add("J", "г"); RusDict.add("¤", "д"); RusDict.add("Ґ",
"е");
     RusDict.add("с", "ё"); RusDict.add("¦", "ж"); RusDict.add("§",
"з");
     RusDict.add("Ё", "и"); RusDict.add("©", "й"); RusDict.add("Є",
"к");
      RusDict.add("«", "л"); RusDict.add("¬", "м"); RusDict.add("",
"н");
     RusDict.add("®", "о"); RusDict.add("Ї", "п"); RusDict.add("а",
"р");
     RusDict.add("б", "с"); RusDict.add("в", "т"); RusDict.add("г",
"у");
     RusDict.add("д", "ф"); RusDict.add("е", "х"); RusDict.add("ж",
"ц");
     RusDict.add("з", "ч"); RusDict.add("и", "ш"); RusDict.add("й",
"щ");
     RusDict.add("к", "ъ"); RusDict.add("л", "ы"); RusDict.add("м",
"ь");
     RusDict.add("н", "э"); RusDict.add("о", "ю"); RusDict.add("п",
"я");
    }
    //Функция для перевода строки из DOS- в Windows-кодировку
    function DosToWin(s) {
     var i,ss;  //Объявляем переменные
     //Проверяем, создан ли объект RusDict
     if (typeof(RusDict)=="undefined")
      //Если объект RusDict не создан, создаем его
      MakeRusDict();
     ss="";
     for (i=0;i<s.length;i++) {  //Цикл по всем символам в строке
      if (RusDict.Exists(s.charAt(i)))  //Проверяем наличие символа
в словаре
       //Преобразуем i-й символ в Windows-кодировку
       ss+=RusDict.Item(s.charAt(i));
      else ss+=s.charAt(i);
     }
     return ss;
    }
    /*************                                          Начало
*****************************************/
    //Печатаем приглашение для ввода
    WScript.StdOut.Write("Введите одну строку: ");
    s = WScript.StdIn.ReadLine();  //Вводим строку с клавиатуры
```

```
    WScript.StdOut.WriteBlankLines(1);  //Печатаем пустую строку
    WScript.StdOut.Write("Было введено: ");
    //Преобразовываем введенную строку в Windows-кодировку
    //и выводим ее на экран
    WScript.StdOut.WriteLine(DosToWin(s));
    /*************                                          Конец
*****************************************/
```

Листинг 2.15. Ввод одной строки с помощью метода StdIn ReadLine (VBScript)

```
    '*************************************************************
    ' Имя: StdIn1.vbs
    ' Язык: VBScript
    ' Описание: Пример использования метода StdIn.WriteLine
    '*************************************************************
    ' Функция для перевода строки из DOS- в Windows-кодировку
    Function DosToWin(s)
    Dim i,k,ss
     ss=""
     For i=1 To Len(s)  ' Цикл по всем символам в строке
      k = Asc(Mid(s,i,1))  ' Определяем ANSI-код i-го символа
      ' Изменяем код k на код соответствующего символа в
      ' Windows-кодировке
      If (128 <= k) And (k <= 175) Then
       k=k+64
      ElseIf (224 <= k) And (k <= 239) Then
       k=k+16
      ElseIf k = 240 Then
       k=168
      ElseIf k = 241 Then
       k=184
      End If
      ss=ss+Chr(k)  ' Возвращаем преобразованную строку
     Next
     DosToWin=ss
    End Function
    '*************                                          Начало
*****************************************
    Dim s
    ' Печатаем приглашение для ввода
    WScript.StdOut.Write "Введите одну строку: "
    s = WScript.StdIn.ReadLine  ' Вводим строку с клавиатуры
    WScript.StdOut.WriteBlankLines 1   ' Печатаем пустую строку
    WScript.StdOut.Write "Было введено: "
```

```
    ' Преобразовываем введенную строку в Windows-кодировку
    ' и выводим ее на экран
    WScript.StdOut.WriteLine DosToWin(s)
    '*************                                          Конец
*******************************************
```

Используя метод `WScript.StdIn.ReadAll`, можно ввести сразу несколько строк подряд, ввод при этом прекращается после нажатия клавиш <Ctrl>+<Z>. Из введенной таким образом переменной можно затем сформировать массив, содержащий все строки. Для этого в JScript применяется метод `split` объекта `string`, а в VBScript — одноименная внутренняя функция `Split` (листинги 2.16 и 2.17).

Листинг 2.16. Ввод нескольких строк с помощью метода `StdIn.ReadAll` (JScript)

```
    /*****************************************************************
    /*                                                          Имя:
StdIn2.js                                                    */
    /*                                                         Язык:
JScript                                                    */
    /*       Описание:       Пример       использования       метода
StdIn.ReadAll              */
    /*****************************************************************
    var RusDict;
    //Функция  для  создания  объекта  Dictionary  с  парами  "ключ-
значение", где
    //"ключ"-буква    в    DOS-кодировке,    "значение"-    символ,
соответствующий этой
    //букве в Windows-кодировке
    function MakeRusDict() {
     //Создаем объект Dictionary
     RusDict = WScript.CreateObject("Scripting.Dictionary");
      //Заполняем  пары  "ключ"  (символ  в  DOS-кодировке)-"значение"
(символ в
     //Window-кодировке) для всех букв русского алфавита
     RusDict.add("Ђ", "А"); RusDict.add("Ѓ", "Б"); RusDict.add(",",
"В");
     RusDict.add("ѓ", "Г"); RusDict.add("„", "Д"); RusDict.add("…",
"Е");
     RusDict.add("р", "Ё"); RusDict.add("†", "Ж"); RusDict.add("‡",
"З");
     RusDict.add("€", "И"); RusDict.add("‰", "Й"); RusDict.add("Љ",
```

```
"К");
     RusDict.add("‹", "Л"); RusDict.add("Њ", "М"); RusDict.add("Ќ",
"Н");
     RusDict.add("Ћ", "О"); RusDict.add("Џ", "П"); RusDict.add("ђ",
"Р");
     RusDict.add("‘", "С"); RusDict.add("’", "Т"); RusDict.add("“",
"У");
     RusDict.add("”", "Ф"); RusDict.add("•", "Х"); RusDict.add("–",
"Ц");
     RusDict.add("—", "Ч"); RusDict.add("", "Ш"); RusDict.add("™",
"Щ");
     RusDict.add("љ", "Ъ"); RusDict.add("›", "Ы"); RusDict.add("њ",
"Ь");
     RusDict.add("ќ", "Э"); RusDict.add("ћ", "Ю"); RusDict.add("џ",
"Я");
     RusDict.add(" ", "а"); RusDict.add("Ў", "б"); RusDict.add("ў",
"в");
     RusDict.add("J", "г"); RusDict.add("¤", "д"); RusDict.add("Ґ",
"е");
     RusDict.add("c", "ё"); RusDict.add("¦", "ж"); RusDict.add("§",
"з");
     RusDict.add("Ё", "и"); RusDict.add("©", "й"); RusDict.add("Є",
"к");
     RusDict.add("«", "л"); RusDict.add("¬", "м"); RusDict.add("",
"н");
     RusDict.add("®", "о"); RusDict.add("Ї", "п"); RusDict.add("a",
"р");
     RusDict.add("б", "с"); RusDict.add("в", "т"); RusDict.add("г",
"у");
     RusDict.add("д", "ф"); RusDict.add("е", "х"); RusDict.add("ж",
"ц");
     RusDict.add("з", "ч"); RusDict.add("и", "ш"); RusDict.add("й",
"щ");
     RusDict.add("к", "ъ"); RusDict.add("л", "ы"); RusDict.add("м",
"ь");
     RusDict.add("н", "э"); RusDict.add("о", "ю"); RusDict.add("п",
"я");
    }
    //Функция для перевода строки из DOS- в Windows-кодировку
    function DosToWin(s) {
     var i,ss;  //Объявляем переменные
     //Проверяем, создан ли объект RusDict
     if (typeof(RusDict)=="undefined")
      //Если объект RusDict не создан, создаем его
      MakeRusDict();
     ss="";
     for (i=0;i<s.length;i++) {  //Цикл по всем символам в строке
```

```
      if (RusDict.Exists(s.charAt(i)))  //Проверяем наличие символа
в словаре
       //Преобразуем i-й символ в Windows-кодировку
       ss+=RusDict.Item(s.charAt(i));
      else ss+=s.charAt(i);
     }
     return ss;
    }
    /*************                                                Начало
*****************************************/
    var s,ArrS,i;  //Объявляем переменные
    //Печатаем приглашение для ввода
    WScript.StdOut.WriteLine("Вводите строки:");
    s = WScript.StdIn.ReadAll();  //Вводим строки с клавиатуры
    WScript.StdOut.WriteBlankLines(3);  //Печатаем пустые строки
    ArrS=s.split("\n");  //Формируем массив из введенных строк
    WScript.StdOut.WriteLine("Всего ведено строк: "+ArrS.length);
    for (i=1;i<=ArrS.length;i++)
     //Преобразовываем введенные строки в Windows-кодировку
     //и выводим их на экран
     WScript.StdOut.WriteLine(i+": "+DosToWin(ArrS[i-1]));
    /*************                                                 Конец
*****************************************/
```

Листинг 2.17. Ввод нескольких строк с помощью метода `StdIn.ReadAll` (VBScript)

```
    '*************************************************************
    ' Имя: StdIn2.vbs
    ' Язык: VBScript
    ' Описание: Пример использования метода StdIn.ReadAll
    '*************************************************************
    Option Explicit
    ' Функция для перевода строки из DOS- в Windows-кодировку
    Function DosToWin(s)
    Dim i,k,ss
     ss=""
     For i=1 To Len(s)  ' Цикл по всем символам в строке
      k = Asc(Mid(s,i,1))  ' Определяем ANSI-код i-го символа
      ' Изменяем код k на код соответствующего символа в
      ' Windows-кодировке
      If (128 <= k) And (k <= 175) Then
       k=k+64
      ElseIf (224 <= k) And (k <= 239) Then
       k=k+16
```

```
      ElseIf k = 240 Then
       k=168
      ElseIf k = 241 Then
       k=184
      End If
      ss=ss+Chr(k)
     Next
     DosToWin=ss   ' Возвращаем преобразованную строку
    End Function
    '*************                                                Начало
*****************************************
    Dim s,ArrS,i,ColStr   ' Объявляем переменные
    ' Печатаем приглашение для ввода
    WScript.StdOut.WriteLine "Вводите строки:"
    s = WScript.StdIn.ReadAll  ' Вводим строки с клавиатуры
    WScript.StdOut.WriteBlankLines 3  ' Печатаем пустые строки
    ArrS=Split(s,vbCrLf)  ' Формируем массив из введенных строк
    ColStr=UBound(ArrS)+1
    ' Печатаем введенные строки
    WScript.StdOut.WriteLine "Всего ведено строк: " & ColStr
    For i=1 To ColStr
     ' Преобразовываем введенные строки в Windows-кодировку
     ' и выводим их на экран
     WScript.StdOut.WriteLine i & ": " & DosToWin(ArrS(i-1))
    Next
    '*************                                                Конец
*****************************************/
```

Ввод строк в графическом режиме

В сценариях VBScript в графическом режиме информацию можно вводить с помощью диалогового окна, создаваемого внутренней функцией `InputBox` (рис. 2.6).

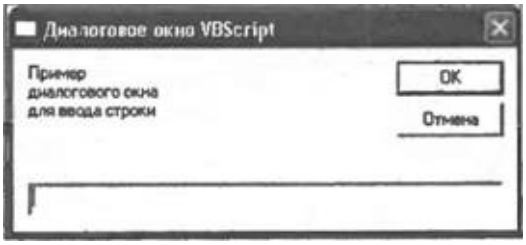


Рис. 2.6. Диалоговое окно со строкой ввода

Пример сценария, использующего функцию `InputBox`, представлен в листинге 2.18 (подробное описание параметров функции InputBox см. в *приложении 1*).

Листинг 2.18. Ввод одной строки с помощью функции InputBox (VBScript)

```
    '*************************************************************
    ' Имя: InpBox.vbs
    ' Язык: VBScript
    ' Описание: Пример использования функции InputBox
    '*************************************************************
    Option Explicit
    Dim s,s1  ' Объявляем переменные
    s1="Пример" & vbCrLf & "диалогового окна" & vbCrLf & "для ввода
строки"
    ' Выводим диалоговое окно со строкой ввода на экран
    s=InputBox(s1,"Диалоговое окно VBScript")
    ' Выводим диалоговое окно с введенной строкой
    MsgBox "Было введено: " & s
    '*************                                          Конец
*****************************************/
```

К сожалению, ни в языке JScript, ни в объектной модели WSH нет функции или метода, позволяющих напрямую создавать диалоговые окна со строкой ввода. Однако при помощи файлов сценариев с XML-разметкой, описанных в *главе 3*, функции языка VBScript (`InputBox` в частности) можно использовать внутри JScript-сценария (соответствующий пример приведен в листинге 3.11).

Получение свойств WSH и запущенного сценария

На практике часто бывает необходимо знать определенные атрибуты WSH (например, с помощью какого приложения-сервера был запущен сценарий) и сценария, работающего в данный момент (например, имя этого сценария или путь к каталогу, в котором он находится). Некоторые параметры WSH и исполняемого сценария можно определить непосредственно с помощью соответствующих методов объекта `WScript`:

- полный путь к приложению-серверу (cscript.exe или wscript.exe);
- имя каталога, в котором находится приложение-сервер;
- номер используемой версии WSH;
- полный путь к исполняемому сценарию;
- имя исполняемого сценария.

Для проверки режима, в котором был запущен сценарий, можно предложить функцию `IsCScript` (ниже приведена реализация этой функции на языке JScript), которая будет возвращать `true`, если использовался хост cscript.exe (консольный режим), и `false`, если использовался wscript.exe (графический режим):

```
function IsCScript() {
 //Проверка режима, в котором запущен сценарий
                              return                              ("c"==
WScript.FullName.toLowerCase().charAt(WScript.FullName.length      -
11));
}
```

Как мы видим, вся работа функции `IsCScript` состоит в определении того, с какой буквы начинается имя приложения-сервера ("c" для cscript.exe или "w" для wscript.exe).

Полный путь к текущему каталогу, т.е. к каталогу, из которого был запущен сценарий, хранится в свойстве `CurrentDirectory` объекта `WshShell`.

Если сценарий был запущен не из того каталога, в котором находится сам файл со сценарием, то текущий каталог не будет совпадать с каталогом сценария. Для того чтобы получить путь к каталогу сценария, нужно выделить этот путь из свойства `WScript.ScriptFullName`, содержащему полный путь к выполняемому сценарию (включая имя файла). На языке JScript это можно реализовать с помощью функции `GetScriptDir` следующего содержания:

```
function GetScriptDir() {
```

```
     var ScriptDir;
     ScriptDir = WScript.ScriptFullName;
               ScriptDir          =          ScriptDir.substring(0,
ScriptDir.lastIndexOf("\\"));
     return ScriptDir;
    }
```

Полные тексты сценариев на языках JScript (PropScript.js) и VBScript (PropScript.vbs), выводящих на экран сведения о свойства WSH и запущенного сценария, приведены в листингах 2.19 и 2.20 соответственно; результат работы сценария PropScript.js представлен на рис. 2.7.

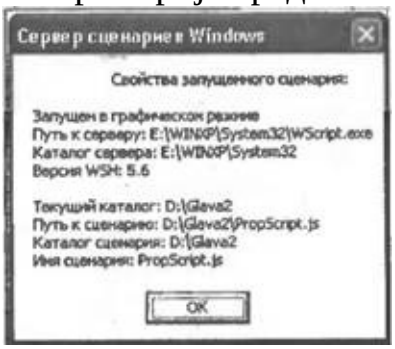


Рис. 2.7. Результаты выполнения сценария PropScript.js в графическом режиме

Листинг 2.19. Вывод свойств WSH и запущенного сценария (JScript)

```
    /*****************************************************************
    /*                                                          Имя:
PropScript.js                                                  */
    /*                                                          Язык:
JScript                                                        */
    /*       Описание:       Вывод       свойств       запущенного
сценария                    */
    /*****************************************************************
    //Проверка режима, в котором запущен сценарий
    function IsCScript() {
                              return                          ("c"==
WScript.FullName.toLowerCase().charAt(WScript.FullName.length      -
11));
    }
    //Возвращает каталог, содержащий запущенный сценарий
    function GetScriptDir() {
     var ScriptDir;
     ScriptDir = WScript.ScriptFullName;
```

```
              ScriptDir          =          ScriptDir.substring(0,
ScriptDir.lastIndexOf("\\"));
     return ScriptDir;
    }
    /*******************                                          Начало
*********************************/
    var WshShell,s; //Объявляем переменные
    //Создаем объект WshShell
    WshShell=WScript.CreateObject("WScript.Shell");
    s="                    Свойства запущенного сценария:\n\n";
    //Проверяем, в каком режиме был запущен сценарий
    if (IsCScript()) s+="Запущен в консольном режиме\n";
    else s+="Запущен в графическом режиме\n";
    //Определяем остальные параметры
    s+="Путь к серверу: "+WScript.FullName+"\n";
    s+="Каталог сервера: "+WScript.Path+"\n";
    s+="Версия WSH: "+WScript.Version+"\n\n";
    s+="Текущий каталог: "+ WshShell.CurrentDirectory+"\n";
    s+="Путь к сценарию: "+WScript.ScriptFullName+"\n";
    s+="Каталог сценария: "+GetScriptDir()+"\n";
    s+="Имя сценария: "+WScript.ScriptName+"\n";
    WScript.Echo(s);  //Выводим сформированные строки
    /*************                                               Конец
*******************************************/
```

Листинг 2.20. Вывод свойств WSH и запущенного сценария (VBScript)

```
    '************************************************************
    ' Имя: PropScript.vbs
    ' Язык: VBScript
    ' Описание: Вывод свойств запущенного сценария
    '************************************************************
    Option Explicit
    ' Проверка режима, в котором запущен сценарий
    Function IsCScript()
                                                          IsCScript=
("c"=Mid(LCase(WScript.FullName),Len(WScript.FullName)-10,1))
    End Function
    ' Возвращает каталог, содержащий запущенный сценарий
    Function GetScriptDir()
     Dim ScriptDir
     ScriptDir = WScript.ScriptFullName
     ScriptDir = Left(ScriptDir, InstrRev(ScriptDir,"\")-1)
     GetScriptDir=ScriptDir
    End Function
```

```
    '*******************                                                Начало
*********************************/
    Dim WshShell,s   ' Объявляем переменные
    ' Создаем объект WshShell
    Set WshShell=WScript.CreateObject("WScript.Shell")
    s="                     Свойства запущенного сценария:" & vbCrLf
& vbCrLf
    ' Проверяем, в каком режиме был запущен сценарий
    If IsCScript() Then
     s=s & "Запущен в консольном режиме" & vbCrLf
    Else
     s=s & "Запущен в графическом режиме" & vbCrLf
    End If
    ' Определяем остальные параметры
    s=s & "Путь к серверу: " & WScript.FullName & vbCrLf
    s=s & "Каталог сервера: " & WScript.Path & vbCrLf
    s=s & "Версия WSH: " & WScript.Version & vbCrLf & vbCrLf
    s=s & "Текущий каталог: "+ WshShell.CurrentDirectory & vbCrLf
    s=s & "Путь к сценарию: " & WScript.ScriptFullName & vbCrLf
    s=s & "Каталог сценария: " & GetScriptDir()  & vbCrLf
    s=s & "Имя сценария: " & WScript.ScriptName & vbCrLf
    WScript.Echo s   ' Выводим сформированные строки
    '*************                                                    Конец
*******************************************
```

Работа с параметрами командной строки сценария

Используя аргументы командной строки, в сценарии можно передавать различную информацию, скажем, те или иные переключатели или имена пользователей и рабочих станций. При задании аргумента можно указать либо только его значение, либо имя вместе со значением в следующем формате: *"Имя_аргумента:Значение"*.

Замечание

Как в имени аргумента, так и в его значении могут использоваться символы кириллицы.

Например, выполнив в командном окне строку

```
cscript Example.js /Имя:"Андрей Попов" /Возраст:30
```

или

```
cscript Example.js /Возраст:30 /Имя:"Андрей Попов"
```

мы передадим в сценарий Example.js два параметра: "Имя" со значением "Андрей Попов" и "Возраст" со значением "30". Значения этих параметров можно было передать и как безымянные параметры:

```
cscript Example.js "Андрей Попов" 30
```

Однако в последнем случае при задании безымянных аргументов будет важен порядок их указания в командной строке.

В WSH для обработки параметров командной строки служат следующие объекты-коллекции:

- `WshArguments` (содержит все параметры, как именные, так и безымянные);
- `WshNamed` (содержит только именные параметры);
- `WshUnnamed` (содержит только безымянные параметры).

Замечание

Описание аргументов командной строки в сценариях можно задавать с помощью XML-элементов `<runtime>`, `<named>` и

<unnamed> *(см. главу 3).*

Для доступа к коллекциям, содержащим аргументы командной строки, в сценарии сначала нужно создать переменную-экземпляр объекта `WshArguments`; для этого используется свойство `Arguments` объекта `WScript`. Пример на языке JScript:

```
var objArgs=WScript.Arguments;
```

Для создания экземпляров коллекций `WshNamed` и `WshUnnamed` используются соответственно методы `Named` и `Unnamed` объекта `WshArguments`. Например:

```
var objNamedArgs=objArgs.Named;
var objUnnamedArgs=objArgs.Unnamed;
```

Методы и свойства коллекций `WshArguments`, `WshNamed` и `WshUnnamed` подробно описаны в *главе 1*. Отметим здесь только, что для корректной работы с параметрами командной строки, имена которых содержат символы кириллицы, эти имена в сценарии должны быть написаны в кодировке Windows.

В листингах 2.21 и 2.22 приведены примеры сценариев на языках JScript и VBScript, которые выводят на экран общее количество параметров командной строки, количество именных и безымянных аргументов, а также значения каждой из этих групп параметров. Результат работы этих сценариев, запущенных в консольном режиме, представлен на рис. 2.8.

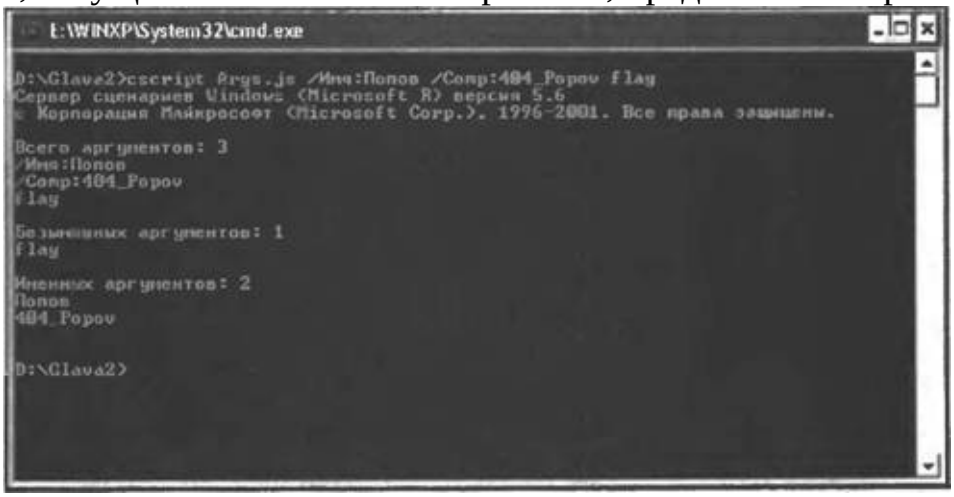


Рис. 2.8. Результат работы сценария Args.js

Листинг 2.21. Доступ к параметрам командной строки запущенного сценария (JScript)

```
    /*****************************************************************
    /*                                                              Имя:
Args.js                                                          */
    /*                                                             Язык:
JScript                                                         */
    /*     Описание:     Работа     с     аргументами     запущенного
сценария              */
    /*****************************************************************
    var
        i,objArgs,s,objNamedArgs,objUnnamedArgs;        //Объявляем
переменные
    objArgs = WScript.Arguments;  //Создаем объект WshArguments
    //Определяем общее количество аргументов
    s="Всего аргументов: "+objArgs.Count()+"\n";
    for (i=0; i<=objArgs.Count()-1; i++)
       s+=objArgs(i)+"\n";   //Формируем   строки   со   значениями
аргументов
    objUnnamedArgs=objArgs.Unnamed;  //Создаем объект WshUnnamed
    //Определяем количество безымянных аргументов
    s+="\nБезымянных аргументов: "+objUnnamedArgs.length+"\n";
    for (i=0; i<=objUnnamedArgs.length-1; i++)
     //Формируем строки со значениями безымянных аргументов
     s+=objUnnamedArgs(i)+"\n";
    objNamedArgs=objArgs.Named; //Создаем объект WshNamed
    //Определяем количество именных аргументов
    s+="\nИменных аргументов: "+objNamedArgs.length+"\n";
    //Проверяем, существует ли аргумент /Имя:
    if (objNamedArgs.Exists("Имя")) s+=objNamedArgs("Имя")+"\n";
    //Проверяем, существует ли аргумент /Comp:
    if (objNamedArgs.Exists("Comp")) s+=objNamedArgs("Comp")+"\n";
    WScript.Echo(s);  //Выводим сформированные строки
    /*************                                              Конец
*******************************************/
```

Листинг 2.22. Доступ к параметрам командной строки запущенного сценария (VBScript)

```
    '*****************************************************************
    ' Имя: Args.vbs
    ' Язык: VBScript
    ' Описание: Работа с аргументами запущенного сценария
    '*****************************************************************
    Option Explicit
    Dim  i,Arg,objArgs,s,objNamedArgs,objUnnamedArgs    '  Объявляем
переменные
```

```
    Set objArgs = WScript.Arguments  ' Создаем объект WshArguments
    ' Определяем общее количество аргументов
    s="Всего аргументов: " & objArgs.Count() & vbCrLf
    For Each Arg In objArgs
      s=s  &  Arg  &  vbCrLf    '  Формируем  строки  со  значениями
аргументов
    Next
    Set objUnnamedArgs=objArgs.Unnamed  ' Создаем объект WshUnnamed
    ' Определяем количество безымянных аргументов
    s=s    &    vbCrLf    &    "Безымянных    аргументов:    "    &
objUnnamedArgs.length & vbCrLf
    For Each Arg In objUnnamedArgs
     ' Формируем строки со значениями безымянных аргументов
     s=s & Arg & vbCrLf
    Next
    Set objNamedArgs=objArgs.Named  ' Создаем объект WshNamed
    ' Определяем количество именных аргументов
    s=s & vbCrLf & "Именных аргументов: " & objNamedArgs.Length &
vbCrLf
    ' Проверяем, существует ли аргумент /Имя:
    If objNamedArgs.Exists("Имя") Then
     s=s & objNamedArgs("Имя") & vbCrLf
    End If
    ' Проверяем, существует ли аргумент /Comp:
    If objNamedArgs.Exists("Comp") Then
     s=s & objNamedArgs("Comp") & vbCrLf
    End If
    WScript.Echo s   ' Выводим сформированные строки
    '*************                                                  Конец
*****************************************
```

Выход из сценария с определенным кодом завершения

Любое приложение при завершении своей работы может возвращать операционной системе целое число — код выхода (обычно ненулевое значение этого кода указывает на то, что выполнение программы прервалось в силу той или иной ошибки).

Замечание

Сама операционная система Windows не проверяет код завершения приложений.

В WSH код выхода из сценария задается с помощью параметра метода `Quit` объекта `WScript`. В листингах 2.23 и 2.24 приведены сценарии, в которых код завершения выбирается в зависимости от того, какая кнопка нажата в диалоговом окне (рис. 2.9): кнопке **OK** соответствует код 1, кнопке **Отмена** — код 0.

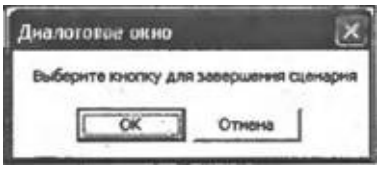


Рис. 2.9. Диалоговое окно, создаваемое в сценарии Quit.js

Листинг 2.23. Выход из сценария с заданным кодом завершения (JScript)

```
    /*****************************************************************
    /*                                                          Имя:
Quit.js                                                    */
    /*                                                         Язык:
JScript                                                   */
    /*   Описание:   Выход   из   сценария   с   заданным   кодом
завершения         */
    /*****************************************************************
    var WshShell,Res,Text,Title;  //Объявляем переменные
    var   vbOkCancel=1,vbOk=1;     //Инициализируем   константы   для
```

```
диалоговых окон
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    Text="Выберите кнопку для завершения сценария";
    Title="Диалоговое окно";
    //Выводим диалоговое окно на экран
    Res=WshShell.Popup(Text,0,Title,vbOkCancel);
    if (Res==vbOk) WScript.Quit(1);
    else WScript.Quit(0);
    /*************                                                Конец
*****************************************/
```

Листинг 2.24. Выход из сценария с заданным кодом завершения (VBScript)

```
    '*************************************************************
    ' Имя: Quit.vbs
    ' Язык: VBScript
    ' Описание: Выход из сценария с заданным кодом завершения
    '*************************************************************
    Option Explicit
    Dim WshShell,Res,Text,Title  ' Объявляем переменные
    ' Создаем объект WshShell
    Set WshShell = WScript.CreateObject("WScript.Shell")
    Text="Выберите кнопку для завершения сценария"
    Title="Диалоговое окно"
    ' Выводим диалоговое окно на экран
    Res=WshShell.Popup(Text,0,Title,vbOkCancel)
    If Res=1 Then
     WScript.Quit 1
    Else
     WScript.Quit 0
    End If
    '*************                                                Конец
*****************************************
```

Если сценарий запускался с помощью командного файла, то код выхода можно проанализировать с помощью оператора `IF ERRORLEVEL`.

Пример подобного BAT-файла приведен в листинге 2.25. Здесь сценарий Quit.js запускается с помощью команды `START` с ключом `/WAIT`, указывающим на то, что выполнение BAT-файла должно быть приостановлено до окончания работы Quit.js. После этого, если код завершения равен 1 (в диалоговом окне сценария была нажата кнопка **OK**), происходит переход к метке `:Ok` и выдача с помощью команды `ECHO`

соответствующего сообщения на экран.

Замечание

Для корректного отображения на экране символов кириллицы в BAT-файлах должна использоваться DOS-кодировка.

Если же код завершения сценария Quit.js был равен 0 (в диалоговом окне была нажата кнопка **Отмена**), то управление перейдет к строке

```
ECHO Для выхода из Quit.js была нажата кнопка Отмена
```

Листинг 2.25. Анализ кода выхода сценария Quit.js (Check.bat)

```
@ECHO OFF
REM
**************************************************************
REM Имя: check.bat
REM Язык: BAT-файл
REM Кодировка: DOS
REM Описание: Определение кода завершения для сценария Quit.js
REM
**************************************************************
@ECHO OFF
ECHO Запускаем сценарий Quit.js...
START /W Quit.js
REM Определяем код завершения для сценария Quit.js
IF ERRORLEVEL 1 GOTO :Ok
ECHO Для выхода из Quit.js была нажата кнопка Отмена
GOTO :end
:Ok
ECHO Для выхода из Quit.js была нажата кнопка Ok
:end
```

Использование внешних объектов автоматизации (на Microsoft Word)

Для того чтобы из сценария получить доступ к свойствам или методам внешнего сервера автоматизации, вначале надо "создать" соответствующий объект, т.е. загрузить в память экземпляр нужного COM-объекта и сохранить в переменной ссылку на этот объект. Напомним, что объект в сценарии может создаваться несколькими способами:

□ с помощью метода `CreateObject` объекта `WScript` (объектная модель WSH);

□ с помощью конструкции `new ActiveXObject` (язык JScript);

□ с помощью функции `CreateObject` (язык VBScript).

В любом случае в используемый метод или функцию в качестве параметра передается *программный идентификатор* объекта (ProgID), заключенный в скобки. Пример на языке JScript:

```
var WA=WScript.CreateObject("Word.Application");
```

То же самое на VBScript:

```
Set WA=WScript.CreateObject("Word.Application")
```

Перед точкой в ProgID стоит имя *библиотеки типов* (type library) для объекта, которая может существовать как в виде отдельного файла с расширением tlb, так и в виде части файла с исполняемым кодом объекта (библиотека типов, содержащая сведения о COM-объекте, регистрируется в системном реестре при установке приложения, использующего этот объект). После точки в ProgID указывается имя класса, содержащего свойства и методы, доступные для использования другими приложениями.

Выполняя метод `CreateObject`, интерпретатор сценария через ProgID получает из системного реестра путь к файлам нужной библиотеки типов. Затем с помощью этой библиотеки в память загружается экземпляр запрашиваемого объекта, и его интерфейсы становятся доступными для использования в сценарии. Ссылка на созданный объект сохраняется в переменной; в дальнейшем, используя эту переменную, мы получаем доступ к свойствам и методам объекта, а также к его вложенным объектам (если они имеются).

Для примера рассмотрим, каким образом из сценария можно управлять работой Microsoft Word, который является сервером автоматизации (листинги 2.26 и 2.27).

Замечание

Более подробно объектная схема Microsoft Word описывается в *главе 9*.

Сначала создается главный объект `Word.Application`, который запускает приложение Microsoft Word:

```
WA=WScript.CreateObject("Word.Application");
```

Затем создается новый пустой документ, в результате в переменную WD заносится ссылка на объект `Document`:

```
WD=WA.Documents.Add();
```

Наконец, в переменную `Sel` заносится ссылка на объект `Selection`, с помощью которого можно задать тип и размер шрифта, тип выравнивания абзацев и напечатать в документе строки текста:

```
Sel=WA.Selection;
```

В результате выполнения сценариев PrintInWord.js или PrintInWord.vbs в новом документе Word печатаются две строки текста (рис. 2.10), после чего с помощью метода `PrintOut` объекта `Document` содержимое документа выводится на принтер:

```
WD.PrintOut();
```

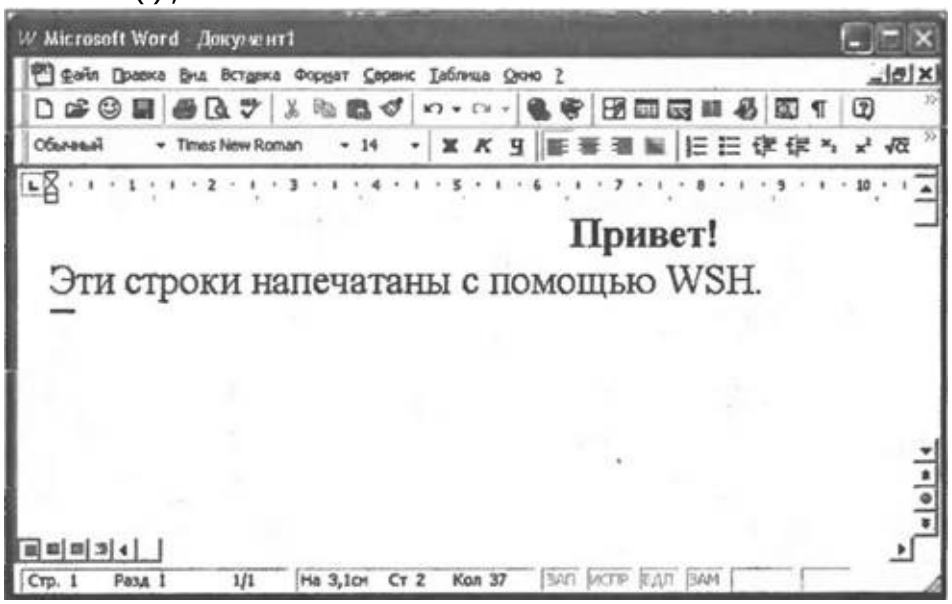


Рис. 2.10. Результат выполнения сценариев PrintInWord.js

Листинг 2.26. Использование сервера автоматизации Microsoft Word (JScript)

```
    /*****************************************************************
    /*                                                              Имя:
PrintInWord.js                                                  */
    /*                                                             Язык:
JScript                                                        */
    /*    Описание:    Использование    из    сценария    внешнего
объекта            */
    /*                                    автоматизации   (Microsoft
Word)                        */
    /*****************************************************************
    var WA,WD,Sel;  //Объявляем переменные
    //Создаем объект--приложение Microsoft Word
    WA=WScript.CreateObject("Word.Application");
    //Можно было использовать конструкцию
    //WA=new ActiveXObject("Word.Application");
    WD=WA.Documents.Add();     //Создаем   новый   документ   (объект
Document)
    WA.Visible=true;  //Делаем Word видимым
    Sel=WA.Selection;  //Создаем объект Selection
    Sel.Font.Size=14;  //Устанавливаем размер шрифта
    Sel.ParagraphFormat.Alignment=1;  //Выравнивание по центру
    Sel.Font.Bold=true;  //Устанавливаем полужирный шрифт
    Sel.TypeText("Привет!\n"); //Печатаем строку текста
    Sel.Font.Bold=false; //Отменяем полужирный шрифт
    Sel.ParagraphFormat.Alignment=0;  //Выравнивание по левому краю
    //Печатаем строку текста
    Sel.TypeText("Эти строки напечатаны с помощью WSH.");
    WD.PrintOut();   //Выводим документ на принтер
    /*************                                                 Конец
*******************************************/
```

Листинг 2.27. Использование сервера автоматизации Microsoft Word (VBScript)

```
    '*****************************************************************
    ' Имя: PrintInWord.vbs
    ' Язык: VBScript
    ' Описание: Использование из сценария внешнего объекта
    '           автоматизации (Microsoft Word)
    '*****************************************************************
    Option Explicit
    Dim WA,WD,Sel  ' Объявляем переменные
    'Создаем объект--приложение Microsoft Word
    Set WA=WScript.CreateObject("Word.Application")
    ' Можно было использовать конструкцию
```

```
    ' Set WA=CreateObject("Word.Application")
    Set  WD=WA.Documents.Add    'Создаем  новый  документ  (объект
Document)
    WA.Visible=true  ' Делаем Word видимым
    Set Sel=WA.Selection  'Создаем объект Selection
    Sel.Font.Size=14 'Устанавливаем размер шрифта
    Sel.ParagraphFormat.Alignment=1  'Выравнивание по центру
    Sel.Font.Bold=true  'Устанавливаем полужирный шрифт
    Sel.TypeText "Привет!" & vbCrLf  'Печатаем строку текста
    Sel.Font.Bold=false  'Отменяем полужирный шрифт
    Sel.ParagraphFormat.Alignment=0  'Выравнивание по левому краю
    'Печатаем строку текста
    Sel.TypeText "Эти строки напечатаны с помощью WSH."
    WD.PrintOut   'Выводим документ на принтер
    '*************                                          Конец
*****************************************
```

Запуск из сценариев внешних программ

Внешние программы и команды можно запускать из сценариев различными способами.

Запуск приложений Windows

Запустить из сценария WSH другое приложение можно с помощью методов `Run` или `Exec` объекта `WshShell`.

При использовании метода `Run` для запускаемого приложения можно задать тип окна (при условии, что приложение поддерживает этот тип). Например, в результате выполнения следующих двух строк JScript-кода:

```
var WshShell = WScript.CreateObject("WScript.Shell");
WshShell.Run("notepad", 3);
```

программа Блокнот (notepad.exe) будет запущена в максимизированном (распахнутом на весь экран) окне (список всех возможных значений параметров метода `Run` приведен в табл. 1.13).

Замечание

Метод `Run` всегда создает новый экземпляр запускаемого процесса, с его помощью нельзя ни повторно активизировать окно запущенного приложения (для этого используется метод `AppActivate`), ни свернуть или развернуть его.

Другим вариантом запуска из сценария приложения Windows является применение метода `Exec`. Этот метод запускает приложение, путь к которому указан как параметр метода, и возвращает объект `WshScriptExec`.

Например:

```
var WshShell = WScript.CreateObject("WScript.Shell");
var theNotepad = WshShell.Exec("calc");
```

Замечание

При подобном запуске приложения, в отличие от метода `Run`, нельзя задать тип окна.

Объект `WshScriptExec` позволяет контролировать ход выполнения запущенного приложения с помощью свойства `Status` — если `Status` равен 0, то приложение выполняется, если `Status` равен 1, то приложение завершено. Кроме этого, используя метод `Terminate`, можно принудительно завершить работу того приложения, которому соответствует объект `WshScriptExec`.

В листинге 2.28 приведен сценарий на языке JScript, в котором с помощью метода Exec запускается Блокнот (notepad.exe); ссылка на соответствующий объект `WshScriptExec` сохраняется в переменной `theNotepad`:

```
theNotepad = WshShell.Exec("notepad");
```

После этого выполнение сценария приостанавливается на 1 секунду (пауза необходима для того, чтобы окно Блокнота успело появиться на экране), после чего выводится диалоговое окно с информацией о статусе запущенного приложения и вопросом о необходимости закрытия Блокнота (рис. 2.11):

```
WScript.Sleep(1000);
Text="Блокнот    запущен(Status="+theNotepad.Status+")\nЗакрыть
Блокнот?";
Title="";
Res=WshShell.Popup(Text, 0, Title, vbQuestion+vbYesNo);
```

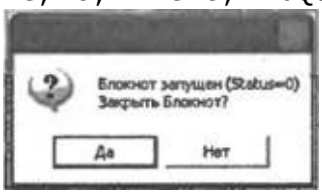


Рис. 2.11. Диалоговое окно, формируемое в сценарии ExecWinApp.js

В случае утвердительного ответа происходит закрытие Блокнота с помощью метода **Terminate:**

```
if (Res==vbYes) {
 theNotepad.Terminate();
 WScript.Sleep(100);
 WScript.Echo("Блокнот закрыт (Status="+theNotepad.Status+")");
}
```

Листинг 2.28. Запуск и принудительное закрытие приложения (JScript)

```
/*****************************************************************
/*                                                          Имя:
ExecWinApp.js                                              */
```

```
    /*                                                          Язык:
JScript                                                          */
    /*   Описание:   Запуск   и   закрытие   приложения   (объект
WshScriptExec)   */
    /*************************************************************
    var WshShell,theNotepad,Res,Text,Title;  //Объявляем переменные
    //Инициализируем константы для диалоговых окон
    var vbYesNo=4,vbQuestion=32,vbYes=6,vbNo=7;
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    WScript.Echo("Запускаем Блокнот");
    //Запускаем приложение (создаем объект WshScriptExec)
    theNotepad = WshShell.Exec("notepad");
    WScript.Sleep(1000);  //Приостанавливаем выполнение сценария
    Text="Блокнот  запущен  (Status="+theNotepad.Status+")\nЗакрыть
Блокнот?";
    Title="";
    //Выводим диалоговое окно на экран
    Res=WshShell.Popup(Text,0,Title,vbQuestion+vbYesNo);
    //Определяем, какая кнопка нажата в диалоговом окне
    if (Res==vbYes) {
     theNotepad.Terminate();  //Прерываем работу Блокнота
     //Приостанавливаем выполнение сценария для того, чтобы Блокнот
     //успел закрыться
     WScript.Sleep(100);
     WScript.Echo("Блокнот закрыт (Status="+theNotepad.Status+")");
    }
    /*************                                               Конец
*****************************************/
```

Тот же самый пример на языке VBScript приведен в листинге 2.29.

Листинг 2.29. Запуск и принудительное закрытие приложения (VBScript)

```
    '************************************************************
    ' Имя: ExecWinApp.vbs
    ' Язык: VBScript
    ' Описание: Запуск и закрытие приложение (объект WshScriptExec)
    '************************************************************
    Option Explicit
    Dim WshShell,theNotepad,Res,Text,Title   ' Объявляем переменные
    ' Создаем объект WshShell
    Set WshShell = WScript.CreateObject("WScript.Shell")
    WScript.Echo "Запускаем Блокнот"
    ' Запускаем приложение (создаем объект WshScriptExec)
```

```
    Set theNotepad = WshShell.Exec("notepad")
    WScript.Sleep 500   ' Приостанавливаем выполнение сценария
    Text="Блокнот запущен (Status=" & theNotepad.Status & ")" &
vbCrLf _
     & "Закрыть Блокнот?"
    Title=""
    ' Выводим диалоговое окно на экран
    Res=WshShell.Popup(Text,0,Title,vbQuestion+vbYesNo)
    ' Определяем, какая кнопка нажата в диалоговом окне
    If Res=vbYes Then
     theNotepad.Terminate ' Прерываем работу Блокнота
     ' Приостанавливаем выполнение сценария для того, чтобы Блокнот
     ' успел закрыться
     WScript.Sleep 100
      WScript.Echo "Блокнот закрыт (Status=" & theNotepad.Status &
")"
    End If
    '*************                                                Конец
*****************************************/
```

Переключение между приложениями, имитация нажатий клавиш

Производить переключение между окнами нескольких запущенных приложений позволяет метод `AppActivate` объекта `WshScript`. В качестве аргумента этого метода нужно указывать либо заголовок активизируемого окна, либо программный идентификатор (PID) процесса, который запущен в этом окне. Предпочтительным является использование PID, который можно получить с помощью свойства `ProcessID` объекта `WshScriptExec`, соответствующего активизируемому приложению. Недостатки применения в методе `AppActivate` заголовка окна:

□ при написании сценария необходимо знать точное название заголовка;

□ само приложение может изменить текст в заголовке окна;

□ в случае нескольких окон с одинаковыми заголовками `AppActivate` всегда будет активизировать один и тот же экземпляр, доступ к другим окнам получить не удастся.

Активизировав то или иное окно, в котором выполняется приложение Windows, можно из сценария сымитировать нажатия клавиш в этом окне. Для этого используется метод `SendKeys` объекта `WshShell` (подробное описание этого метода приведено в *главе 1*).

Замечание

Для нормальной работы метода `SendKeys` необходимо, чтобы языком по умолчанию в операционной системе был назначен английский язык.

Рассмотрим пример сценария Run&ExecWinApp.js (листинг 2.30), в котором запускается **Калькулятор** (calc.exe), и в его окно с помощью `SendKeys` последовательно посылаются нажатия клавиш <1>, <+>, <2> и <Enter>:

```
theCalculator = WshShell.Exec("calc");
WScript.Sleep(1000);
WshShell.AppActivate(theCalculator.ProcessID);
WshShell.SendKeys("1{+}");
WshShell.SendKeys("2");
WshShell.SendKeys("~"); //Клавиша <Enter>
```

Затем выполнение сценария приостанавливается на 1 секунду, чтобы результат вычислений был виден на экране:

```
WScript.Sleep(1000);
```

после чего результат вычислений (символ "3") копируется в буфер с помощью "нажатия" клавиш <Ctrl>+<C>:

```
WshShell.SendKeys ("^c");
```

После этого на экран выводится сообщение о том, что Калькулятор будет закрыт:

```
WScript.Echo("Закрываем калькулятор");
```

в результате чего окно Калькулятора теряет фокус. Для того чтобы вновь активизировать это окно, используется метод AppActivate, параметром которого служит PID Калькулятора:

```
WshShell.AppActivate(theCalculator.ProcessID);
```

Для того чтобы закрыть окно Калькулятора, в него посылаются нажатия клавиш <Alt>+<F4>:

```
WshShell.SendKeys("%{F4}");
```

После закрытия Калькулятора запускается Блокнот (notepad.exe) и в него записываются результаты работы Калькулятора (вставка из буфера вычисленной суммы производится с помощью нажатий <Ctrl>+<V>):

```
WshShell.Run("notepad");
WScript.Sleep(1000);
WshShell.AppActivate("notepad");
WshShell.SendKeys("1{+}2=");
WshShell.SendKeys("^v");
WshShell.SendKeys(" {(}c{)} Calculator");
```

В результате в Блокноте отображается текст, показанный на рис. 2.12.

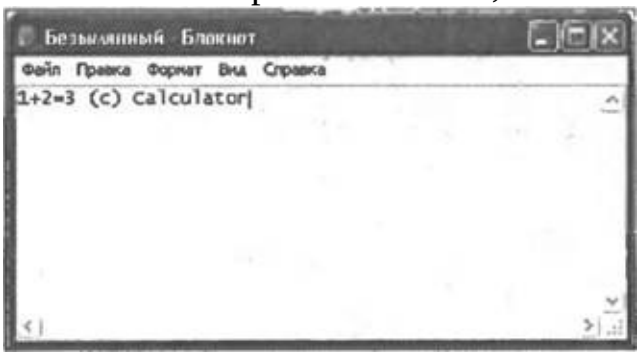


Рис. 2.12. Результат работы сценария Run&ExecWinApp.js

Листинг 2.30. Запуск двух приложений и обмен данными между ними (JScript)

```
    /*****************************************************************
    /*                                                          Имя:
Run&ExecWinApp.js                                              */
    /*                                                         Язык:
JScript                                                       */
    /* Описание: Запуск двух приложений и обмен данными между
ними     */
    /*****************************************************************
    var WshShell, theCalculator;  //Объявляем переменные
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    WScript.Echo("Запускаем калькулятор и\n считаем 1+2");
    //Создаем объект WshScript (запускаем Калькулятор)
    theCalculator = WshShell.Exec("calc");
    //Приостанавливаем выполнение сценария, для того, чтобы
    //окно Калькулятора успело появиться на экране
    WScript.Sleep(1000);
    //Активизируем окно Калькулятора
    WshShell.AppActivate(theCalculator.ProcessID);
    //Посылаем нажатия клавиш в окно Калькулятора
    WshShell.SendKeys("1{+}");
    WshShell.SendKeys("2");
    WshShell.SendKeys("~");  //Клавиша <Enter>
    WScript.Sleep(1000);
    //Копируем результат вычисления в буфер Windows (<Ctrl>+C)
    WshShell.SendKeys("^c");
    //Выводим сообщение (активное окно меняется)
    WScript.Echo("Закрываем калькулятор");
```

```
    //Активизируем окно Калькулятора
    WshShell.AppActivate(theCalculator.ProcessID);
    //Закрываем окно Калькулятора (<Alt>+<F4>)
    WshShell.SendKeys("%{F4}");
    WScript.Echo("Запускаем Блокнот и копируем туда результат");
    WshShell.Run("notepad"); //Запускаем Блокнот
    //Приостанавливаем выполнение сценария, для того, чтобы
    //окно Блокнота успело появиться на экране
    WScript.Sleep(1000);
    WshShell.AppActivate("notepad"); //Активизируем окно Блокнота
    //Посылаем нажатия клавиш в окно Блокнота
    WshShell.SendKeys("1{+}2=");
    //Вставляем содержимое буфера Windows (<Ctrl>+V)
    WshShell.SendKeys("^v");
    //Выводим в окно Блокнота оставшуюся информацию
    WshShell.SendKeys(" {(}c{)} Calculator");
    /*************                                                Конец
*****************************************/
```

Тот же пример, реализованный в виде VBScript-сценария, приведен в листинге 2.31.

Листинг 2.31. Запуск двух приложений и обмен данными между ними (VBScript)

```
    '*************************************************************
    ' Имя: Run&ExecWinApp.vbs
    ' Язык: VBScript
    ' Описание: Запуск двух приложений и обмен данными между ними
    '*************************************************************
    Option Explicit
    Dim WshShell, theCalculator  ' Объявляем переменные
    ' Создаем объект WshShell
    Set WshShell = WScript.CreateObject("WScript.Shell")
    WScript.Echo("Запускаем калькулятор и" & vbCrLf & "считаем
1+2")
    ' Создаем объект WshScript (запускаем Калькулятор)
    Set theCalculator = WshShell.Exec("calc")
    ' Приостанавливаем выполнение сценария, для того, чтобы
    ' окно Калькулятора успело появиться на экране
    WScript.Sleep 500
    ' Активизируем окно Калькулятора
    WshShell.AppActivate theCalculator.ProcessID
    ' Посылаем нажатия клавиш в окно Калькулятора
    WshShell.SendKeys "1{+}"
```

```
WshShell.SendKeys "2"
WshShell.SendKeys "~"    ' Клавиша <Enter>
WScript.Sleep 500
' Копируем результат вычисления в буфер Windows (<Ctrl>+C)
WshShell.SendKeys "^c"
' Выводим сообщение (активное окно меняется)
WScript.Echo "Закрываем калькулятор"
' Активизируем окно Калькулятора
WshShell.AppActivate theCalculator.ProcessID
' Закрываем окно Калькулятора (<Alt>+<F4>)
WshShell.SendKeys "%{F4}"
WScript.Echo "Запускаем Блокнот и копируем туда результат"
WshShell.Run "notepad"     ' Запускаем Блокнот
' Приостанавливаем выполнение сценария, для того, чтобы
' окно Блокнота успело появиться на экране
WScript.Sleep 1000
WshShell.AppActivate "notepad"    ' Активизируем окно Блокнота
' Посылаем нажатия клавиш в окно Блокнота
WshShell.SendKeys "1{+}2="
' Вставляем содержимое буфера Windows (<Ctrl>+V)
WshShell.SendKeys "^v"
' Выводим в окно Блокнота оставшуюся информацию
WshShell.SendKeys " {(}c{)} Calculator"
'*************                                          Конец
*******************************************
```

Запуск независимых консольных приложений и команд DOS

Для запуска независимых, т.е. работающих в отдельном адресном пространстве и использующих свою копию переменных среды, консольных приложений или внешних (представленных исполняемыми файлами на жестком диске) команд DOS используется метод `Run` объекта `WshShell`. При этом выполнение сценария можно приостановить до окончания работы запущенного приложения, а затем проанализировать код выхода этого приложения (для этого третий параметр метода `Run` должен равняться `true`). Соответствующие примеры сценариев на языках JScript и VBScript приведены в листингах 2.32 и 2.33 соответственно.

Листинг 2.32. Запуск независимого консольного приложения (JScript)

```
/*************************************************************
/*                                                        Имя:
```

```
RunConApp.js                                                    */
    /*                                                              Язык:
JScript                                                         */
    /*   Описание:   Запуск   независимого   консольного   приложения
и          */
    /*                                        определение    его    кода
выхода                              */
    /*****************************************************************
    var WshShell, Code;  //Объявляем переменные
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    //Запускаем утилиту xcopy с ключом "/?" и ожидаем окончания ее
работы
    Code=WshShell.Run("xcopy /?",1,true);
    //Печатаем полученный код возврата
    WScript.Echo("Код возврата: ", Code);
    /*************                                                 Конец
*********************************************/
```

Листинг 2.33. Запуск независимого консольного приложения (VBScript)

```
    '*****************************************************************
    ' Имя: RunConApp.vbs
    ' Язык: VBScript
    ' Описание: Запуск независимого консольного приложения и
    '           определение его кода выхода
    '*****************************************************************
    Option Explicit
    Dim WshShell, Code   ' Объявляем переменные
    ' Создаем объект WshShell
    Set WshShell = WScript.CreateObject("WScript.Shell")
    ' Запускаем утилиту xcopy с ключом "/?" и ожидаем окончания ее
работы
    Code=WshShell.Run("xcopy /?",1,true)
    ' Печатаем полученный код возврата
    WScript.Echo "Код возврата: ", Code
    '*************                                                 Конец
*********************************************/
```

Для выполнения внутренней команды DOS нужно запустить командный интерпретатор (в Windows NT/2000/XP это файл cmd.exe, в Windows*9x* — command.com) и передать ему в качестве параметра нужную команду. Для того чтобы при вызове командного интерпретатора не заботиться о полном пути к cmd.exe, нужно использовать переменную среды `COMSPEC`.

Замечание

Для получения значения переменной среды ее имя нужно окружить знаками "%" (например, `%COMSPEC%`).

В листингах 2.34 и 2.35 приведены сценарии на языках JScript и VBScript, в которых запускаются внутренние команды `COPY /?` (вызов встроенной справки для copy) и `DIR %WINDIR%` (вывод содержимого системного каталога Windows).

При этом окно, в котором выполняется команда `COPY /?`, не закрывается после завершения этой команды, т.к. при запуске командного интерпретатора был указан ключ /k, а информация, выводимая командой DIR %WINDIR%, перенаправляется в файл windir.txt, после чего командное окно закрывается, т.к. для командного интерпретатора в этом случае был указан ключ `/c`.

Листинг 2.34. Доступ к параметрам командной строки запущенного сценария (JScript)

```
    /*****************************************************************
    /*                                                          Имя:
RunDOSCom.js                                                  */
    /*                                                         Язык:
JScript                                                     */
    /*       Описание:       Выполнение       внутренних       команд
DOS                        */
    /*****************************************************************
    var WshShell, Code; //Объявляем переменные
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    //Запускаем внутреннюю команду COPY
    WshShell.Run("%COMSPEC% /k copy /?",1);
    //Запускаем внутреннюю команду DIR
    WshShell.Run("%COMSPEC% /c dir %WINDIR% > windir.txt",1);
    /*************                                            Конец
*******************************************/
```

Листинг 2.35. Доступ к параметрам командной строки запущенного сценария (VBScript)

```
    '*************************************************************
    ' Имя: RunDOSCom.vbs
    ' Язык: VBScript
    ' Описание: Выполнение внутренних команд DOS
    '*************************************************************
    Option Explicit
    Dim WshShell, Code   ' Объявляем переменные
    ' Создаем объект WshShell
    Set WshShell = WScript.CreateObject("WScript.Shell")
    ' Запускаем внутреннюю команду COPY
    WshShell.Run "%COMSPEC% /k copy /?",1
    ' Запускаем внутреннюю команду DIR
    WshShell.Run "%COMSPEC% /c dir %WINDIR% > windir.txt",1
    '*************                                          Конец
*****************************************/
```

Запуск дочерних консольных приложений и команд DOS, использование их входных и выходных потоков

Консольное приложение или команду DOS можно запустить из сценария как дочернюю задачу, т.е. с теми же переменными среды, что у процесса-родителя. При этом информация, выводимая дочерним процессом, на экран дублироваться не будет, однако из родительского сценария можно считывать информацию из выходного потока и посылать данные во входной поток дочерней задачи (это напоминает конвейеризацию команд DOS, при которой данные выходного потока одной команды поступают во входной поток другой команды, например `DIR | MORE`). Таким образом, из сценария можно запускать ту или иную утилиту командной строки и обрабатывать выводимые ей данные; иногда таким образом получить нужную информацию бывает проще и быстрее, чем при использовании объектной модели WSH или другого сервера автоматизации.

В качестве примера рассмотрим сценарий ExecConApp.js (листинг 2.36), который выводит на экран общее количество файлов в текущем каталоге и их имена (рис. 2.13).

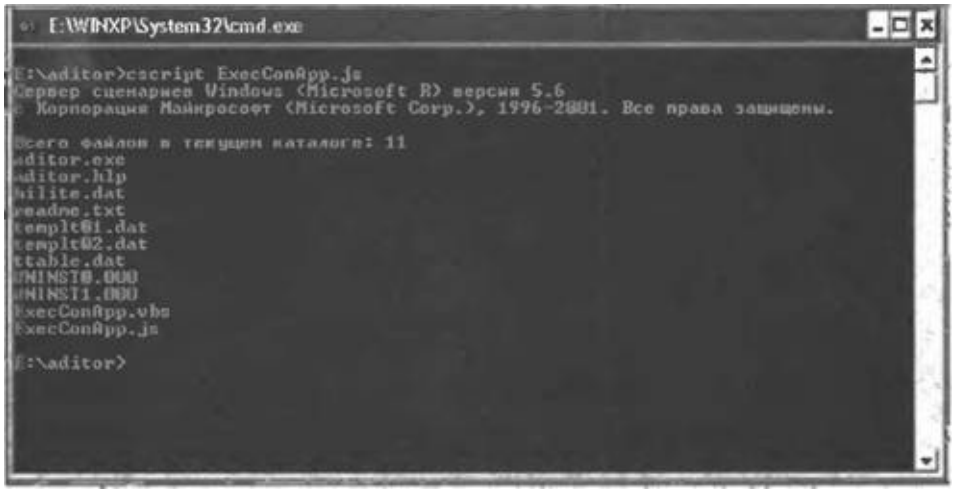


Рис. 2.13. Результат выполнения сценария ExecConApp.js

Как нетрудно заметить, имена файлов выводятся на экран в том же виде, что и при использовании команды `DIR /B` (рис. 2.14).

Таким образом, для получения нужной информации необходимо запустить в качестве дочернего процесса команду `DIR` с ключом `/B`:

```
ObjExec=WshShell.Exec("%COMSPEC% /c dir /b");
```

и полностью считать данные, появляющиеся в выходном потоке этого процесса. Для этого в цикле вызывается метод ReadAll, считывающий всю информацию, имеющуюся к тому времени в потоке StdOut объекта ObjExec в переменную `s`:

```
IsBreak=false;
for (;;) { //Бесконечный цикл
 //Проверяем, достигнут ли конец выходного потока команды DIR
 if (!ObjExec.StdOut.AtEndOfStream)
  //Считываем полностью выходной поток команды DIR
  s+=ObjExec.StdOut.ReadAll();
 if (IsBreak) break; //Выходим из цикла
  if (ObjExec.Status==1) //Проверяем, не завершилось ли
выполнение DIR
  IsBreak=true;
 else WScript.Sleep(100); //Приостанавливаем сценарий на 0,1
сек
}
```

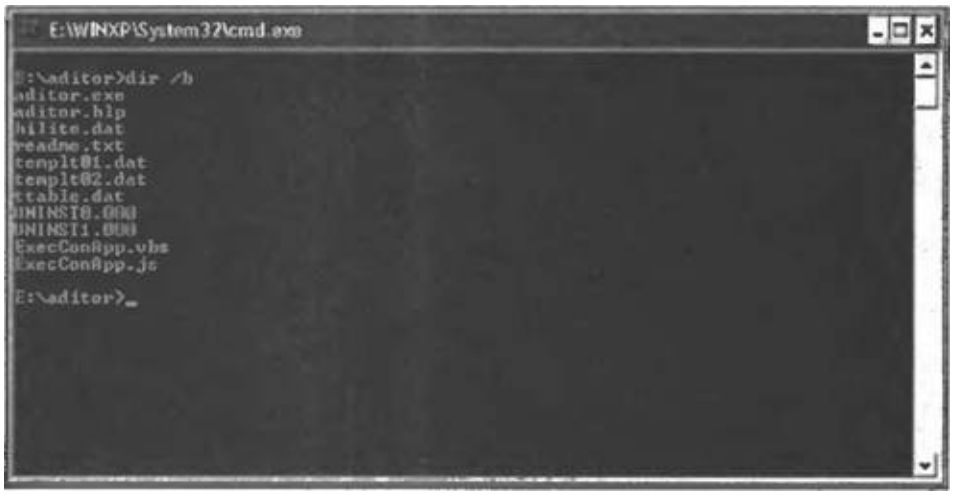


Рис. 2.14. Результат выполнения команды `DIR /B`

Родительский и дочерний процессы работают асинхронно, поэтому пока команда `DIR` не перестанет выдавать данные, т.е. пока свойство `Status` объекта `ObjExec` не станет равным 1, выполнение сценария с помощью метода `WScript.Sleep` периодически приостанавливается на 0,1 секунды.

После того как считаны все данные из выходного потока команды `DIR` (свойство `ObjExec.StdOut.AtEndOfStream` равно `true`), происходит выход из цикла и формирование из переменной s массива выведенных строк:

```
ArrS=s.split("\n");
```

После этого только остается подсчитать количество файлов в каталоге, которое на единицу меньше количества строк в массиве `ArrS`:

```
ColFiles=ArrS.length-1;
```

и вывести нужные строки на экран:

```
WScript.StdOut.WriteLine("Всего файлов в текущем каталоге: "+ColFiles);
for (i=0;i<=ColFiles-1; i++ )
 WScript.StdOut.WriteLine(ArrS[i]); //Выводим строки на экран
```

Замечание

В дочернем консольном приложении вывод строк в выходной поток происходит в DOS-кодировке, поэтому при наличии символов кириллицы эти строки нужно преобразовывать в кодировку Windows (примеры соответствующих функций конвертации на языках JScript и VBScript приведены в листингах 2.14 и 2.15).

Листинг 2.36. Запуск дочернего консольного приложения (JScript)

```
    /*****************************************************************
    /*                                                          Имя:
ExecConApp.js                                                  */
    /*                                                         Язык:
JScript                                                       */
    /*       Описание:       Запуск       дочернего       консольного
приложения                */
    /*****************************************************************
    //Объявляем переменные
    var ObjExec,WshShell,s,IsBreak,ArrS,ColStr,ColFiles,i;
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    //Запускаем команду DIR
    ObjExec=WshShell.Exec("%COMSPEC% /c dir /b");
    s="";
    IsBreak=false;
    for (;;) {  //Бесконечный цикл
     //Проверяем, достигнут ли конец выходного потока команды DIR
     if (!ObjExec.StdOut.AtEndOfStream)
      //Считываем полностью выходной поток команды DIR
      s+=ObjExec.StdOut.ReadAll();
     if (IsBreak) break; //Выходим из цикла
      if  (ObjExec.Status==1)    //Проверяем,  не  завершилось  ли
выполнение DIR
      IsBreak=true;
     else WScript.Sleep(100);  //Приостанавливаем сценарий на 0,1
сек
    }
    ArrS=s.split("\n");  //Формируем массив строк
    ColFiles=ArrS.length-1;      //  Количество  файлов  в  текущем
каталоге
    WScript.StdOut.WriteLine("Всего  файлов  в  текущем  каталоге:
"+ColFiles);
    for (i=0;i<=ColFiles-1;i++)
     WScript.StdOut.WriteLine(ArrS[i]); //Выводим строки на экран
    /*************                                             Конец
*******************************************/
```

Аналогичный сценарий на языке VBScript приведен в листинге 2.37.

Листинг 2.37. Запуск дочернего консольного приложения (VBScript)

```
    '*****************************************************************
```

```
    ' Имя: ExecConApp.vbs
    ' Язык: VbScript
    ' Описание: Запуск дочернего консольного приложения
    '*************************************************************
    Option Explicit
    ' Объявляем переменные
    Dim ObjExec,WshShell,s,IsBreak,ArrS,ColStr,ColFiles,i
    ' Создаем объект WshShell
    Set WshShell = WScript.CreateObject("WScript.Shell")
    ' Запускаем команду DIR
    Set ObjExec=WshShell.Exec("%COMSPEC% /c dir /b")
    s=""
    IsBreak=False
    Do While True  ' Бесконечный цикл
     ' Проверяем, достигнут ли конец выходного потока команды DIR
     If (Not ObjExec.StdOut.AtEndOfStream) Then
      ' Считываем полностью выходной поток команды DIR
      s=s+ObjExec.StdOut.ReadAll
     End If
     If IsBreak Then
      Exit Do  ' Выходим из цикла
     End If
     ' Проверяем, не завершилось ли выполнение DIR
     If ObjExec.Status=1 Then
      IsBreak=True
     Else
      WScript.Sleep 100 ' Приостанавливаем сценарий на 0,1 сек
     End If
    Loop
    ArrS=Split(s,vbCrLf)  ' Формируем массив строк
    ColFiles=UBound(ArrS)  ' Количество файлов в текущем каталоге
    WScript.StdOut.WriteLine "Всего файлов в текущем каталоге: " &
ColFiles
    For i=0 To ColFiles-1
     WScript.StdOut.WriteLine ArrS(i)  ' Выводим строки на экран
    Next
    '*************                                          Конец
*****************************************
```

Доступ к специальным папкам Windows XP

При установке Windows всегда автоматически создаются несколько специальных папок (например, папка для рабочего стола (Desktop) или папка для меню **Пуск** (Start)), путь к которым впоследствии может быть тем или иным способом изменен. С помощью свойства `SpecialFolders` объекта `WshShell` можно создать объект `WshSpecialFolders`, который является коллекцией, содержащей пути ко всем специальным папкам, имеющимся в системе (список названий этих папок приведен в *главе 1* при описании объекта `WshSpecialFolders`).

В листингах 2.38 и 2.39 приводятся сценарии на языках JScript и VBScript соответственно, которые формируют список всех имеющихся в системе специальных папок (рис. 2.15).

Рис. 2.15. Пути для всех специальных папок в Windows XP

Листинг 2.38. Формирование списка всех специальных папок (JScript)

```
    /*****************************************************************
    /*                                                          Имя:
SpecFold1.js                                                  */
    /*                                                         Язык:
JScript                                                     */
    /*   Описание:   Вывод   названий   всех   специальных   папок
Windows        */
    /*****************************************************************
```

```
    var WshShell, WshFldrs, i, s;  //Объявляем переменные
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("Wscript.Shell");
    //Создаем объект WshSpecialFolders
    WshFldrs = WshShell.SpecialFolders;
    s="Список всех специальных папок:\n\n";
    //Перебираем все элементы коллекции WshFldrs
    for (i=0;i<= WshFldrs.Count()-1;i++) {
     //Формируем строки с путями к специальным папкам
     s+=WshFldrs(i)+"\n";
    }
    WScript.Echo(s);
    /*************                                          Конец
*****************************************/
```

Листинг 2.39. Формирование списка всех специальных папок (VBScript)

```
    '*************************************************************
    ' Имя: SpecFold1.vbs
    ' Язык: VBScript
    ' Описание: Вывод названий всех специальных папок Windows
    '*************************************************************
    Option Explicit
    Dim WshShell, WshFldrs, SpecFldr, s  ' Объявляем переменные
    ' Создаем объект WshShell
    Set WshShell = WScript.CreateObject("Wscript.Shell")
    ' Создаем объект WshSpecialFolders
    Set WshFldrs = WshShell.SpecialFolders
    s="Список всех специальных папок:" & vbCrLf & vbCrLf
    ' Перебираем все элементы коллекции WshFldrs
    For Each SpecFldr In WshFldrs
     ' Формируем строки с путями к специальным папкам
     s=s & SpecFldr & vbCrLf
    Next
    WScript.Echo s
    '*************                                          Конец
*****************************************/
```

Объект `WshSpecialFolders` также позволяет получить путь к конкретно заданной специальной папке. Например, в сценарии SpecFold2.js (листинг 2.40) на экран выводятся пути к папкам рабочего стола (Desktop), избранных ссылок (Favorites) и раздела **Программы** (Programs) меню **Пуск** (Run) — рис. 2.16.

Рис. 2.16. Пути для некоторых специальных папок

Листинг 2.40. Доступ к определенным специальным папкам (JScript)

```
    /*****************************************************************
    /*                                                          Имя:
SpecFold2.js                                                  */
    /*                                                         Язык:
JScript                                                     */
    /*  Описание:  Вывод  названий  заданных  специальных  папок
Windows    */
    /*****************************************************************
    var WshShell, WshFldrs, s;   //Объявляем переменные
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("Wscript.Shell");
    //Создаем объект WshSpecialFolders
    WshFldrs = WshShell.SpecialFolders;
    //Формируем строки с путями к конкретным специальным папкам
    s="Некоторые специальные папки:\n\n";
    s+="Desktop:\t"+WshFldrs("Desktop")+"\n";
    s+="Favorites:\t"+WshFldrs("Favorites")+"\n";
    s+="Programs:\t"+WshFldrs("Programs");
    WScript.Echo(s);  //Выводим сформированные строки на экран
    /*************                                            Конец
*******************************************/
```

Реализация того же сценария на языке VBScript приведена в листинге 2.41.

Листинг 2.41. Доступ к определенным специальным папкам (VBScript)

```
    '*****************************************************************
    ' Имя: SpecFold2.vbs
    ' Язык: VBScript
    ' Описание: Вывод названий заданных специальных папок Windows
    '*****************************************************************
```

```
    Option Explicit
    Dim WshShell, WshFldrs, s   ' Объявляем переменные
    ' Создаем объект WshShell
    Set WshShell = WScript.CreateObject("Wscript.Shell")
    ' Создаем объект WshSpecialFolders
    Set WshFldrs = WshShell.SpecialFolders
    ' Формируем строки с путями к конкретным специальным папкам
    s="Некоторые специальные папки:" & vbCrLf & vbCrLf
    s=s+"Desktop:"+WshFldrs("Desktop") & vbCrLf
    s=s+"Favorites:"+WshFldrs("Favorites") & vbCrLf
    s=s+"Programs:"+WshFldrs("Programs")
    WScript.Echo s   ' Выводим сформированные строки на экран
    '*************                                          Конец
*****************************************/
```

Создание ярлыков в специальных папках

Для того чтобы из сценария создать ярлык в специальной папке (рабочий стол, меню **Пуск** (Start) и т.п.) или изменить свойства уже существующего ярлыка, необходимо:

1. Используя коллекцию `WshSpecialFolders`, узнать путь к нужной специальной папке.

2. С помощью метода `CreateShortcut` объекта `WshShell` создать объект `WshShortcut` (`WshUrlShortcut`) для связи с ярлыком в этой папке.

3. Задать или изменить свойства ярлыка с помощью соответствующих методов объекта `WshShortcut` (`WshUrlShortcut`).

4. Сохранить ярлык с помощью метода Save объекта WshShortcut (WshUrlShortcut).

Объект `WshShortcut` предоставляет доступ к следующим свойствам ярлыков (рис. 2.17):

□ **Объект** (Target);

□ **Рабочая папка** (Start in);

□ **Быстрый вызов** (Shortcut key);

□ **Окно** (Run);

□ **Комментарий** (Comment).

Кроме этого, с помощью объекта `WshShortcut` можно сменить значок, который соответствует ярлыку.

Рис. 2.17. Свойства ярлыка в Windows XP

Остальных свойств, имеющихся у ярлыков в Windows XP, объект `WshShortcut` не поддерживает (например, нельзя установить или сбросить флажок, позволяющий запускать процесс в отдельном адресном пространстве или под другой учетной записью пользователя).

В качестве примера ниже приведен сценарий Shortcut.js (листинг 2.42), в котором создается ярлык "Мой ярлык.lnk" на Блокнот (notepad.exe), причем этот ярлык может быть сохранен либо в меню **Программы** (Programs) работающего пользователя, либо на его рабочем столе. Выбор специальной папки в сценарии производится с помощью диалогового окна, которое создается методом `Popup` объекта `WshShell` (рис. 2.18).

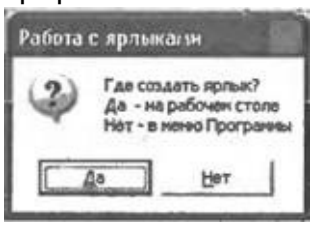


Рис. 2.18. Диалоговое окно для выбора специальной папки

Рис. 2.10. Свойства ярлыка "Мой ярлык.lnk"

Для создаваемого ярлыка выбирается значок из файла Shell32.dll, находящегося в подкаталоге System каталога Windows (в Windows 95/98

этот файл находится в подкаталоге System), назначается комбинация горячих клавиш <Ctrl>+<Alt>+<N> и устанавливается максимизированный тип окна (рис. 2.19).

Листинг 2.42. Доступ к определенным специальным папкам (JScript)

```
    /*****************************************************************
    /*                                                          Имя:
Shortcut.js                                                  */
    /*                                                         Язык:
JScript                                                     */
    /*     Описание:     Создание     ярлыков     в     специальных
папках                */
    /*****************************************************************
    //Объявляем переменные
    var WshShell,MyShortcut,PathTarg,PathIcon,Res,PathShortcut;
    //Инициализируем константы для диалоговых окон
    var vbYesNo=4,vbQuestion=32,vbYes=6;
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    //Выводим запрос для выбора папки, в которой будет создан ярлык
    Res=WshShell.Popup("Где создать ярлык?\nДа  - на рабочем столе
\nНет - в меню Программы",0,
     "Работа с ярлыками",vbQuestion+vbYesNo);
    if (Res==vbYes) //Нажата кнопка Да
     //Определяем путь к рабочему столу
     PathShortcut = WshShell.SpecialFolders("Desktop");
    else
     //Определяем путь к меню Программы
     PathShortcut = WshShell.SpecialFolders("Programs");
    //Создаем объект-ярлык
    MyShortcut     =     WshShell.CreateShortcut(PathShortcut+"\\Мой
ярлык.lnk");
    //Устанавливаем путь к файлу
    PathTarg=WshShell.ExpandEnvironmentStrings("%windir%\\notepad.ex
    MyShortcut.TargetPath = PathTarg;
    //Назначаем комбинацию горячих клавиш
    MyShortcut.Hotkey = "CTRL+ALT+N";
    //Выбираем иконку из файла SHELL32.dll
    PathIcon=
     WshShell.ExpandEnvironmentStrings("%windir%\\system32\\SHELL32.
    MyShortcut.IconLocation = PathIcon+", 1";
    MyShortcut.WindowStyle=3;          //Устанавливаем   тип   окна
(максимизировано)
```

```
    MyShortcut.Save();  //Сохраняем ярлык
    WScript.Echo("Ярлык создан|");
    /*************                                                Конец
*****************************************/
```

Реализация того же сценария на языке VBScript приведена **в** листинге 2.43.

Листинг 2.43. Доступ к определенным специальным папкам (VBScript)

```
    '*************************************************************
    ' Имя: Shortcut.vbs
    ' Язык: JScript
    ' Описание: Создание ярлыков в специальных папках
    '*************************************************************
    Option Explicit
    ' Объявляем переменные
    Dim WshShell,MyShortcut,PathTarg,PathIcon,Res,PathShortcut
    ' Создаем объект WshShell
    Set WshShell = WScript.CreateObject("WScript.Shell")
    ' Выводим запрос для выбора папки, в которой будет создан ярлык
    Res=WshShell.Popup("Где создать ярлык?" & vbCrLf & "Да   - на
рабочем столе" & vbCrLf & _
          "Нет     -     в     меню     Программы",0,"Работа     с
ярлыками",vbQuestion+vbYesNo)
    If Res=vbYes Then  ' Нажата кнопка Да
     ' Определяем путь к рабочему столу
     PathShortcut = WshShell.SpecialFolders("Desktop")
    Else
     ' Определяем путь к меню Программы
     PathShortcut = WshShell.SpecialFolders("Programs")
    End If
    ' Создаем объект-ярлык
    Set  MyShortcut  =  WshShell.CreateShortcut(PathShortcut+"\Мой
ярлык.lnk")
    ' Устанавливаем путь к файлу
    PathTarg=WshShell.ExpandEnvironmentStrings("%windir%\\notepad.ex
    MyShortcut.TargetPath = PathTarg
    ' Назначаем комбинацию горячих клавиш
    MyShortcut.Hotkey = "CTRL+ALT+N"
    ' Выбираем иконку из файла SHELL32.dll
    PathIcon = _
     WshShell.ExpandEnvironmentStrings("%windir%\system32\SHELL32.dl
    MyShortcut.IconLocation = PathIcon & ", 1"
    MyShortcut.WindowStyle=3       '   Устанавливаем   тип   окна
(максимизировано)
    MyShortcut.Save   ' Сохраняем ярлык
    WScript.Echo "Ярлык создан|"
    '*************                                                Конец
```

****

Работа с системным реестром Windows

Во всех версиях Windows системный реестр — это база данных, в которой хранится информация о конфигурации компьютера и операционной системы. С точки зрения пользователя, реестр является иерархическим деревом разделов, подразделов и параметров. Работать с этим деревом можно с помощью стандартного редактора реестра regedit.exe (рис. 2.20).

Рис. 2.20. Редактор реестра regedit.exe

С помощью методов объекта `WshShell` из сценариев WSH можно:

- создавать новые разделы и параметры (метод `RegWrite`);
- изменять значения параметров и разделов (метод `RegWrite`);
- считывать значения параметров и разделов (метод `RegRead`);
- удалять параметры и разделы (метод `RegDelete`).

Замечание

В Windows XP для работы с системным реестром сценарий должен иметь разрешение на доступ к разделам реестра, которым обладает администратор.

В листинге 2.44 представлен сценарий Registry.js, который производит манипуляции внутри корневого раздела `HKEY_CURRENT_USER`, причем каждая операция выполняется только после утвердительного ответа на соответствующий запрос, формируемый в диалоговом окне.

Сначала в разделе `HKEY_CURRENT_USER` создается подраздел `ExampleKey`, в который затем записывается строковый параметр `ExampleValue` со

значением "Value from WSH" (рис. 2.21).

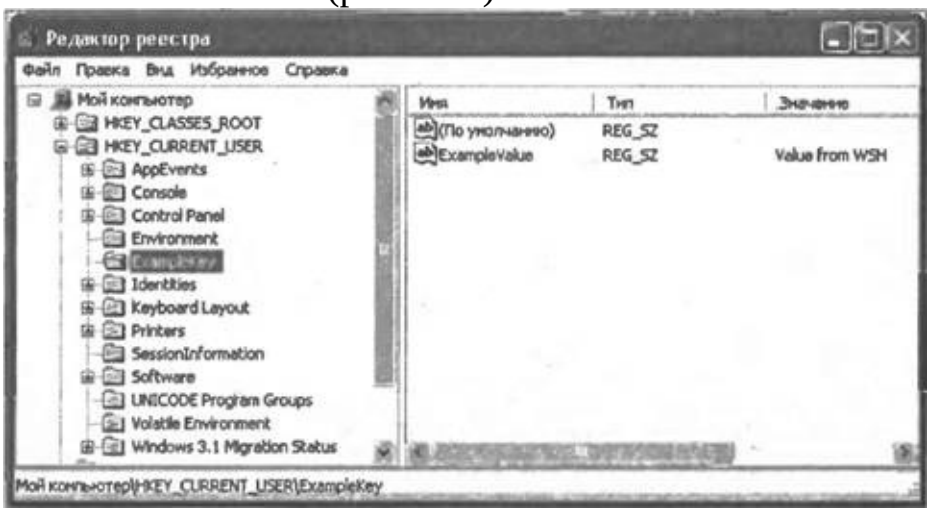


Рис. 2.21. Элементы системного реестра, создаваемые сценарием Registry.js

После этого параметр `ExampleValue` и раздел `ExampleKey` последовательно удаляются из реестра.

Листинг 2.44. Работа с системным реестром (JScript)

```
    /*****************************************************************
    /*                                                          Имя:
Registry.js                                                    */
    /*                                                         Язык:
JScript                                                      */
    /*          Описание:          Работа          с          системным
реестром                                  */
    /*****************************************************************
    //Объявляем переменные
    var WshShell,Root,Key,Res,SValue,ValueName,SRegValue;
    //Инициализируем константы для диалоговых окон
    var
vbYesNo=4,vbQuestion=32,vbInformation=64,vbYes=6,vbOkOnly=0;
    Root="HKEY_CURRENT_USER";  //Корневой ключ
    Key="\\ExampleKey\\";      //Новый ключ
    ValueName="ExampleValue";  //Имя нового параметра
    SValue="Value from WSH";   //Значение нового параметра
    //Создаем объект WshShell
    WshShell=WScript.CreateObject("WScript.Shell");
    //Запрос на создание нового ключа
    Res=WshShell.Popup("Создать ключ\n"+Root+Key+"?",0,
```

```
     "Работа с реестром",vbQuestion+vbYesNo);
    if (Res==vbYes) { //Нажата кнопка Да
     //Записываем новый ключ
     WshShell.RegWrite(Root+Key,"");
     WshShell.Popup("Ключ\n"+Root+Key+" создан!",0,
      "Работа с реестром",vbInformation+vbOkOnly);
    }
    //Запрос на запись нового параметра
    Res=WshShell.Popup("Записать                                    параметр
\n"+Root+Key+ValueName+"?",0,
     "Работа с реестром",vbQuestion+vbYesNo);
    if (Res==vbYes) { //Нажата кнопка Да
     //Записываем новый строковый параметр
     WshShell.RegWrite(Root+Key+ValueName,SValue,"REG_SZ");
     WshShell.Popup("Параметр\n"+Root+Key+ValueName+" записан!",0,
      "Работа с реестром",vbInformation+vbOkOnly);
     //Считываем значение созданного параметра
     SRegValue=WshShell.RegRead(Root+Key+ValueName);
     //Выводим на экран полученное значение
     WshShell.Popup(Root+Key+ValueName+"="+SRegValue,0,
      "Работа с реестром",vbInformation+vbOkOnly);
    }
    //Запрос на удаление параметра
    Res=WshShell.Popup("Удалить                                     параметр
\n"+Root+Key+ValueName+"?",0,
     "Работа с реестром",vbQuestion+vbYesNo);
    if (Res==vbYes) { //Нажата кнопка Да
     //Удаляем параметр
     WshShell.RegDelete(Root+Key+ValueName);
     WshShell.Popup("Параметр\n"+Root+Key+ValueName+" удален!",0,
      "Работа с реестром",vbInformation+vbOkOnly);
    }
    //Запрос на удаление раздела
    Res=WshShell.Popup("Удалить раздел\n"+Root+Key+"?",0,
     "Работа с реестром",vbQuestion+vbYesNo);
    if (Res==vbYes) {  //Нажата кнопка Да
     //Удаляем раздел
     WshShell.RegDelete(Root+Key);
     WshShell.Popup("Раздел\n"+Root+Key+" удален!",0,
      "Работа с реестром",vbInformation+vbOkOnly);
    }
    /*************                                          Конец
*****************************************/
```

Реализация того же сценария на языке VBScript приведена в листинге 2.45.

Листинг 2.45. Работа с системным реестром (VBScript)

```
'*************************************************************
' Имя: Registry.vbs
' Язык: VBScript
' Описание: Работа с системным реестром
'*************************************************************
Option Explicit
'Объявляем переменные
Dim WshShell,Root,Key,Res,SValue,ValueName,SRegValue
Root="HKEY_CURRENT_USER"   'Корневой ключ
Key="\ExampleKey\"         'Новый ключ
ValueName="ExampleValue"   'Имя нового параметра
SValue="Value from WSH"    'Значение нового параметра
'Создаем объект WshShell
Set WshShell=WScript.CreateObject("WScript.Shell")
'Запрос на создание нового ключа
Res=WshShell.Popup("Создать ключ" & vbCrLf & Root & Key &
"?",0,_
 "Работа с реестром",vbQuestion+vbYesNo)
If Res=vbYes Then   'Нажата кнопка Да
 'Записываем новый ключ
 WshShell.RegWrite Root & Key, ""
 WshShell.Popup "Ключ" & vbCrLf & Root & Key & " создан!",0,_
  "Работа с реестром",vbInformation+vbOkOnly
End If
'Запрос на запись нового параметра
Res=WshShell.Popup("Записать параметр" & vbCrLf & Root & Key &
_
 ValueName & "?",0,"Работа с реестром",vbQuestion+vbYesNo)
If Res=vbYes Then   'Нажата кнопка Да
 'Записываем новый строковый параметр
 WshShell.RegWrite Root & Key & ValueName,SValue,"REG_SZ"
 WshShell.Popup "Параметр" & vbCrLf & Root & Key & _
           ValueName     &     "     записан!",0,"Работа     с
реестром",vbInformation+vbOkOnly
 'Считываем значение созданного параметра
 SRegValue=WshShell.RegRead(Root & Key & ValueName)
 'Выводим на экран полученное значение
 WshShell.Popup Root & Key & ValueName & "=" & SRegValue,0,_
  "Работа с реестром",vbInformation+vbOkOnly
End If
'Запрос на удаление параметра
Res=WshShell.Popup("Удалить параметр" & vbCrLf & Root & Key & _
 ValueName & "?",0,"Работа с реестром",vbQuestion+vbYesNo)
If Res=vbYes Then   'Нажата кнопка Да
```

```
      'Удаляем параметр
      WshShell.RegDelete Root & Key & ValueName
      WshShell.Popup "Параметр" & vbCrLf & Root & Key & _
                ValueName     &     "     удален!",0,"Работа     с
реестром",vbInformation+vbOkOnly
     End If
     'Запрос на удаление раздела
     Res=WshShell.Popup("Удалить раздел" & vbCrLf & Root & Key & _
      "?",0,"Работа с реестром",vbQuestion+vbYesNo)
     If Res=vbYes Then   'Нажата кнопка Да
      'Удаляем раздел
      WshShell.RegDelete Root & Key
      WshShell.Popup "Раздел" & vbCrLf & Root & Key & " удален!",0,_
       "Работа с реестром",vbInformation+vbOkOnly
     End If
     '*************                                                Конец
*****************************************
```

Работа с ресурсами локальной сети

Стандартным объектом, позволяющим выполнять типовые операции с локальной сетью, является `WshNetwork`. С помощью этого объекта можно:

□ узнать сетевое имя компьютера, имя текущего пользователя и название домена, в котором он зарегистрировался;

□ получить список всех сетевых дисков и всех сетевых принтеров, подключенных к рабочей станции;

□ подключить или отключить сетевой диск и принтер;

□ установить сетевой принтер в качестве принтера, используемого по умолчанию.

Замечание

Для решения более сложных задач, связанных с администрированием локальной сети, можно применять имеющиеся в Windows XP технологии ADSI — Active Directory Service Interface и WMI — Windows Management Instrumentation.

Определение имен рабочей станции, пользователя и домена

Для того чтобы из сценария узнать имя текущего пользователя, домена и компьютера в сети, можно использовать соответствующие свойства объекта `WshNetwork`: `UserName`, `Domain` и `ComputerName`. Примеры сценариев на языках JScript и VBScript, которые выводят на экран такую информацию, приведены в листингах 2.46 и 2.47.

Листинг 2.46. Вывод сетевых параметров станции (JScript)

```
    /*****************************************************************
    /*                                                          Имя:
NetworkParam.js                                                 */
    /*                                                         Язык:
JScript                                                        */
    /*        Описание:        Вывод        сетевых        параметров
станции                        */
```

```
    /*****************************************************************
    var WshNetwork,s; //Объявляем переменные
    //Создаем объект WshNetwork
    WshNetwork = WScript.CreateObject("WScript.Network");
    s="Сетевые параметры станции:\n\n";
    //Выводим на экран свойства ComputerName, UserName и UserDomain
    s+="Имя машины: "+WshNetwork.ComputerName+"\n";
    s+="Имя пользователя: "+WshNetwork.UserName+"\n";
    s+="Домен: "+WshNetwork.UserDomain;
    WScript.Echo(s);
    /*************                                                Конец
*****************************************/
```

Листинг 2.47. Вывод сетевых параметров станции и списков подключенных сетевых ресурсов (VBScript)

```
    '*****************************************************************
    ' Имя: NetworkParam.vbs
    ' Язык: VBScript
    ' Описание: Вывод сетевых параметров станции
    '*****************************************************************
    Option Explicit
    Dim   WshNetwork,s,NetwDrives,i,NetwPrinters      '   Объявляем
переменные
    ' Создаем объект WshNetwork
    Set WshNetwork = WScript.CreateObject("WScript.Network")
    s="Сетевые параметры станции:" & vbCrLf & vbCrLf
    ' Выводим на экран свойства ComputerName, UserName и UserDomain
    s=s & "Имя машины: " & WshNetwork.ComputerName & vbCrLf
    s= s & "Имя пользователя: " & WshNetwork.UserName & vbCrLf
    s= s & "Домен: " & WshNetwork.UserDomain
    WScript.Echo s
    '*************                                                Конец
*****************************************
```

Получение списка подключенных сетевых дисков и принтеров

У объекта `WshNetwork` имеются методы `EnumNetworkDrives` и `EnumPrinterConnections`, с помощью которых можно создать коллекции, содержащие, соответственно, сведения о всех подключенных к локальной станции сетевых дисках и сетевых принтерах. Эти коллекции устроены

следующим образом: первым элементом является буква диска или название порта, вторым — сетевое имя ресурса, с которым связан этот диск или принтер. Та же последовательность сохраняется для всех элементов коллекции.

В листингах 2.48 и 2.49 приведены сценарии на языках JScript и VBScript соответственно, в которых на экран выводятся диалоговые окна, содержащие информацию о сетевых дисках и сетевых принтерах, подключенных к рабочей станции (рис. 2.22).

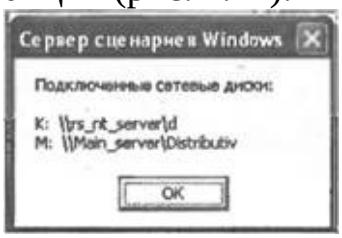


Рис. 2.22. Выводимая сценарием ListNetworkResources.js информация о подключенных сетевых ресурсах

Листинг 2.48. Вывод списка подключенных сетевых ресурсов (JScript)

```
    /*****************************************************************
    /*                                                          Имя:
ListNetworkResources.js                                      */
    /*                                                         Язык:
JScript                                                     */
    /*  Описание:  Вывод  подключенных  сетевых  ресурсов  (диски  и
принтеры) */
    /*****************************************************************
    var     WshNetwork,s,NetwDrives,i,NetwPrinters;     //Объявляем
переменные
    //Создаем объект WshNetwork
    WshNetwork = WScript.CreateObject("WScript.Network");
    /*****   Вывод списка всех подключенных сетевых дисков  ******/
    s="Подключенные сетевые диски:\n\n";
    //Создаем коллекцию с данными о подключенных дисках
    NetwDrives = WshNetwork.EnumNetworkDrives();
    i=0;
    while   (i<=NetwDrives.Count()-2)   {     //Перебираем   элементы
коллекции
     //В первом элементе коллекции содержится буква диска,
     //во втором - сетевое имя ресурса и т.д.
     s+=NetwDrives(i)+"  "+NetwDrives(i+1)+"\n";
     i=i+2;
```

```
    }
    WScript.Echo(s);  //Выводим сформированные строки на экран
    /******     Вывод списка всех подключенных сетевых принтеров
******/
    s="Подключенные сетевые принтеры:\n\n";
    //Создаем коллекцию с данными о подключенных принтерах
    NetwPrinters = WshNetwork.EnumPrinterConnections();
    i=0;
    while  (i<=NetwPrinters.Count()-2)  {    //Перебираем  элементы
коллекции
     //В первом элементе коллекции содержится названия локальных
портов,
     //во втором - сетевое имя принтера и т.д.
     s+=NetwPrinters(i)+"  "+NetwPrinters(i+1)+"\n";
     i=i+2;
    }
    WScript.Echo(s); //Выводим сформированные строки на экран
    /*************                                                Конец
*****************************************/
```

Листинг 2.49. Вывод списка подключенных сетевых ресурсов (VBScript)

```
    '************************************************************
    '                                                      Имя:
ListNetworkResources.vbs
    '                                                     Язык:
JScript
    '  Описание:  Вывод  подключенных  сетевых  ресурсов  (диски  и
принтеры)
    '************************************************************
    Option Explicit
    Dim   WshNetwork,s,NetwDrives,i,NetwPrinters      '   Объявляем
переменные
    ' Создаем объект WshNetwork
    Set WshNetwork = WScript.CreateObject("WScript.Network")
    '********     Вывод списка всех подключенных сетевых дисков
*********
    s="Подключенные сетевые диски:" & vbCrLf & vbCrLf
    ' Создаем коллекцию с данными о подключенных дисках
    Set NetwDrives = WshNetwork.EnumNetworkDrives()
    i=0
    While i<=NetwDrives.Count()-2  ' Перебираем элементы коллекции
     ' В первом элементе коллекции содержится буква диска,
     ' во втором - сетевое имя ресурса и т.д.
      s=s  &  NetwDrives.Item(i)  &  "   "  &  NetwDrives.Item(i+1)  &
```

```
vbCrLf
     i=i+2
    Wend
    WScript.Echo s   ' Выводим сформированные строки на экран
    '********          Вывод списка всех подключенных сетевых
принтеров    *******
    s="Подключенные сетевые принтеры:" & vbCrLf & vbCrLf
    ' Создаем коллекцию с данными о подключенных принтерах
    Set NetwPrinters = WshNetwork.EnumPrinterConnections()
    i=0
    While  i<=NetwPrinters.Count()-2       '  Перебираем элементы
коллекции
     ' В первом элементе коллекции содержится названия локальных
портов,
     ' во втором - сетевое имя принтера и т.д.
     s=s & NetwPrinters.Item(i) & "   " & NetwPrinters.Item(i+1) &
vbCrLf
     i=i+2
    Wend
    WScript.Echo s  'Выводим сформированные строки на экран
    '*************                                          Конец
*****************************************
```

Подключение и отключение сетевых дисков и принтеров

Имеющиеся в локальной сети общедоступные ресурсы (диски и принтеры) можно посредством сценария подключить к рабочей станции для совместного использования. Подключаемому сетевому диску при этом нужно поставить в соответствие незанятую букву локального диска (например, если в системе уже имеются диски C:, D: и E: (локальные или сетевые), то сетевой диск можно подключить под буквой F: или K:, но не E:). В случае подключения сетевого принтера можно либо напрямую соединиться с этим принтером (для печати из приложений Windows), либо поставить в соответствие удаленному принтеру локальный порт (для печати из старых приложений MS-DOS).

Замечание

Сетевые диски и принтеры также можно подключить с помощью Проводника Windows или выполнив соответствующую команду `NET USE`.

В качестве примера рассмотрим JScript-сценарий MapResources.js (листинг 2.50), в котором производится подключение диска K: к сетевому ресурсу \\RS_NT_Server\d и установка связи локального порта LPT1 с сетевым принтером \\104_Stepankova\HP.

Сначала нужно создать экземпляры объектов WshNetwork и WshShell:

```
WshNetwork = WScript.CreateObject("WScript.Network");
WshShell = WScript.CreateObject("WScript.Shell");
```

Для того чтобы подключить сетевой диск к устройству K:, нужно быть уверенным, что с этой буквой уже не связан сетевой диск (иначе произойдет ошибка). Поэтому предварительно отключается сетевой диск с помощью метода RemoveNetworkDrive:

```
WshNetwork.RemoveNetworkDrive(Drive);
```

(переменной Drive заранее было присвоено значение "K:"). При выполнении этой команды может произойти ошибка времени выполнения (например, диск K: не существует или возникла ошибка при отключении связанного с ним сетевого ресурса), поэтому вызов метода `RemoveNetworkDrive` помещается внутрь блока try конструкции `try…catch` языка JScript, которая позволяет обрабатывать такие ошибки:

```
try {
 //Отключаем сетевой диск
 WshNetwork.RemoveNetworkDrive(Drive);
} catch (e) { //Обрабатываем возможные ошибки
if (e.number != 0) {
 //Выводим сообщение об ошибке
 IsError=true;
 Mess="Ошибка при отключении диска "+Drive + "\nКод ошибки: "+
  e.number+"\nОписание: " + e.description;
   WshShell.Popup(Mess,   0,   "Отключение   сетевого   диска",
vbCritical);
 }
}
```

Теперь в случае возникновения ошибки при работе метода `RemoveNetworkDrive` управление передастся внутрь блока `catch`, а в полях переменной-объекта e будет содержаться информация о произошедшей ошибке (`e.number` — числовой код ошибки, `e.description` — краткое описание ошибки); эта информация отображается в диалоговом окне (рис. 2.23).

Рис. 2.23. Информация об ошибке, произошедшей при отключении диска K:

Если же отключение диска K: прошло успешно, на экран выводится диалоговое окно с информацией об этом (рис. 2.24):

```
    if (!IsError) { //Все в порядке
     Mess="Диск "+Drive+" отключен успешно";
       WshShell.Popup(Mess,  0,  "Отключение  сетевого  диска",
vbInformation);
    }
```

Рис. 2.24. Информация об успешном отключении диска K:

Аналогичный блок `try…catch` используется и при подключении сетевого диска:

```
    try {
     //Подключаем сетевой диск
     WshNetwork.MapNetworkDrive(Drive, NetPath);
    } catch (e) {
     //Обрабатываем возможные ошибки
     if (e != 0) {
      //Выводим сообщение об ошибке
      IsError=true;
       Mess="Ошибка  при  подключении  диска  "  +  Drive  +  "  к  "  +
NetPath+
       "\nКод ошибки: "+e.number + "\nОписание: "+e.description;
          WshShell.Popup(Mess,  0,  "Подключение  сетевого  диска",
vbCritical);
```

Если, например, пользователь, который подключает сетевой диск, не имеет соответствующих прав доступа к сетевому ресурсу, то на экран выведется диалоговое окно, изображенное на рис. 2.25.

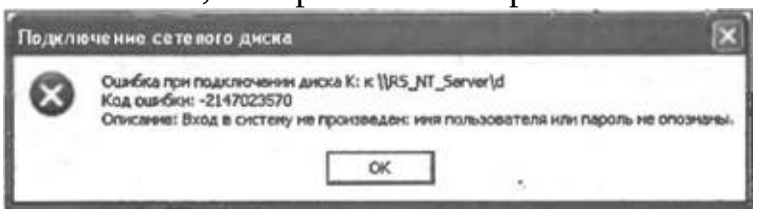


Рис. 2.25. Информация об ошибке, произошедшей при подключении диска K:

Освобождение локального порта (метод `RemovePrinterConnection`), подключение сетевого принтера к этому порту (метод `AddPrinterConnection`) и обработка ошибок времени выполнения, которые могут возникнуть при этих действиях, производится в сценарии аналогичным образом.

Листинг 2.50. Отключение и подключение сетевых ресурсов (JScript)

```
    /*****************************************************************
    /*                                                          Имя:
MapResources.js                                                 */
    /*                                                         Язык:
JScript                                                        */
    /*  Описание:  Отключение  и  подключение  сетевых  дисков  и
принтеров    */
    /*****************************************************************
    //Объявляем переменные
    var
WshNetwork,WshShell,Drive,NetPath,Port,NetPrinter,Mess,IsError;
    //Инициализируем константы для диалоговых окон
    var vbCritical=16,vbInformation=64;
    Drive="K:";      //Буква диска
    //NetPath="\\\\RS_NT_Server\\d";       //Сетевой    путь    для
подключения диска
    NetPath="\\\\RS_NT_Server\\d";   //Сетевой путь для подключения
диска
    Port="LPT1";   //Название локального порта
    //Сетевой путь для подключения принтера
    NetPrinter="\\\\104_Stepankova\\HP";
    //Создаем объект WshNetwork
    WshNetwork = WScript.CreateObject("WScript.Network");
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    /*************             Отключение      сетевого      диска
**********************/
    IsError=false;
    try {
     //Отключаем сетевой диск
     WshNetwork.RemoveNetworkDrive(Drive);
    } catch (e) {  //Обрабатываем возможные ошибки
     if (e != 0) {
      //Выводим сообщение об ошибке
      IsError=true;
      Mess="Ошибка при отключении диска "+Drive+"\nКод ошибки: "+
```

```
       e.number+"\nОписание: "+e.description;
                    WshShell.Popup(Mess,0,"Отключение       сетевого
диска",vbCritical);
     }
    }
    if (!IsError) {
     //Все в порядке
     Mess="Диск "+Drive+" отключен успешно";
                WshShell.Popup(Mess,0,"Отключение           сетевого
диска",vbInformation);
    }
    /*************            Подключение      сетевого      диска
**********************/
    IsError=false;
    try {
     //Подключаем сетевой диск
     WshNetwork.MapNetworkDrive(Drive,NetPath);
    } catch (e) { //Обрабатываем возможные ошибки
     if (e != 0) {
      //Выводим сообщение об ошибке
      IsError=true;
       Mess="Ошибка  при  подключении  диска  "  +  Drive  +  "  к  "  +
NetPath+
       "\nКод ошибки: "+e.number+"\nОписание: "+e.description;
                    WshShell.Popup(Mess,0,"Подключение       сетевого
диска",vbCritical);
     }
    }
    if (!IsError) {
     //Все в порядке
     Mess="Диск "+Drive+" успешно подключен к "+NetPath;
                WshShell.Popup(Mess,0,"Подключение           сетевого
диска",vbInformation);
    }
    /*************          Освобождение     локального     порта
**********************/
    IsError=false;
    try {
     //Разрываем связь с сетевым принтером
     WshNetwork.RemovePrinterConnection(Port);
    } catch (e) {
     if (e != 0) {  //Обрабатываем возможные ошибки
      //Выводим сообщение об ошибке
      IsError=true;
      Mess="Ошибка при отключении порта "+Port+"\nКод ошибки: "+
       e.number+"\nОписание: "+e.description;
          WshShell.Popup(Mess,0,"Отключение   локального   порта   от
```

```
сетевого ресурса",vbCritical);
     }
    }
    if (!IsError) {
     //Все в порядке
     Mess="Порт "+Port+" отключен успешно";
     WshShell.Popup(Mess,0,"Отключение локального порта от сетевого
ресурса",vbInformation);
    }
    /*****   Подключение локального порта   к сетевому принтеру
*********/
    IsError=false;
    try {
     //Подключаем сетевой принтер к локальному порту
     WshNetwork.AddPrinterConnection(Port,NetPrinter);
    } catch (e) {  //Обрабатываем возможные ошибки
     if (e != 0) {
      //Выводим сообщение об ошибке
      IsError=true;
         Mess="Ошибка  при  переназначении  порта  "+Port+  "  на
"+NetPrinter+
       "\nКод ошибки: "+e.number+"\nОписание: "+e.description;
         WshShell.Popup(Mess,0,"Подключение  локального  порта  к
сетевому ресурсу",vbCritical);
     }
    }
    if (!IsError) {
     //Все в порядке
     Mess="Порт "+Port+" успешно подключен к "+NetPrinter;
     WshShell.Popup(Mess,0,"Подключение локального порта к сетевому
ресурсу",vbInformation);
    }
    /*************                                                Конец
*****************************************/
```

Реализация того же сценария на языке VBScript представлена в листинге 2.51. Главное отличие здесь состоит в способе обработки возможных ошибок времени выполнения. В VBScript для этой цели предназначен оператор `On Error Resume Next` — при возникновении ошибки после выполнения этого оператора сценарий не прервется, а просто перейдет к выполнению следующей строки кода. Проанализировать же возникшую ошибку можно с помощью специального объекта `Err`, в полях `Number` и `Description` которого будут соответственно содержаться код и описание ошибки.

Листинг 2.51. Отключение и подключение сетевых ресурсов (VBScript)

```
'*************************************************************
' Имя: MapResources.vbs
' Язык: VBScript
' Описание: Отключение и подключение сетевых дисков и принтеров
'*************************************************************
Option Explicit
' Объявляем переменные
Dim WshNetwork,Drive,NetPath,Port,NetPrinter
Drive="K:"   ' Буква диска
NetPath="\\RS_NT_Server\d"      ' Сетевой путь для подключения
диска
Port="LPT1"   ' Название локального порта
' Сетевой путь для подключения принтера
NetPrinter="\\104_Stepankova\HP"
' Создаем объект WshNetwork
Set WshNetwork = WScript.CreateObject("WScript.Network")
' Создаем объект WshShell
Set WshShell = WScript.CreateObject("WScript.Shell")
On  Error  Resume  Next   ' Включаем обработку ошибок времени
выполнения
'*************            Отключение      сетевого      диска
**********************
' Отключаем сетевой диск
WshNetwork.RemoveNetworkDrive Drive
If Err.Number<>0 Then
 Mess="Ошибка при отключении диска " & Drive & vbCrLf & _
  "Код ошибки: " &  e.number & vbCrLf &+ _
  "Описание: " & e.description
 WshShell.Popup Mess,0,"Отключение сетевого диска",vbCritical
Else
 ' Все в порядке
 Mess="Диск " & Drive & " отключен успешно"
       WshShell.Popup        Mess,0,"Отключение        сетевого
диска",vbInformation
End If
'*************            Подключение      сетевого      диска
**********************
' Подключаем сетевой диск
WshNetwork.MapNetworkDrive Drive,NetPath
If Err.Number<>0 Then
 Mess="Ошибка при подключении диска " & Drive & " к " & NetPath
&_
  "Код ошибки: " & e.number & "Описание: " & e.description
 WshShell.Popup Mess,0,"Подключение сетевого диска",vbCritical
```

```
    Else
     ' Все в порядке
     Mess="Диск " & Drive & " успешно подключен к " & NetPath
           WshShell.Popup       Mess,0,"Подключение       сетевого
диска",vbInformation
    End If
    '*************          Освобождение     локального     порта
**********************
    ' Разрываем связь с сетевым принтером
    WshNetwork.RemovePrinterConnection Port
    If Err.Number<>0 Then
     Mess="Ошибка при отключении порта " & Port & "Код ошибки: " &_
      e.number & "Описание: " & e.description
       WshShell.Popup   Mess,0,"Отключение   порта   от   сетевого
ресурса",vbCritical
    Else
     ' Все в порядке
     Mess="Порт " & Port & " отключен успешно"
     WshShell.Popup Mess,0,"Отключение порта от сетевого ресурса",_
      vbInformation
    End If
    '*****   Подключение  локального  порта   к  сетевому  принтеру
*********
    ' Подключаем сетевой принтер к локальному порту
    WshNetwork.AddPrinterConnection Port,NetPrinter
    If Err.Number<>0 Then
     Mess="Ошибка  при  переназначении  порта  " & Port & " на " &
NetPrinter &_
      "Код ошибки: " & e.number & "Описание: " & e.description
       WshShell.Popup   Mess,0,"Подключение   порта   к   сетевому
ресурсу",vbCritical
    Else
     ' Все в порядке
     Mess="Порт " & Port & " успешно подключен к " & NetPrinter
     WshShell.Popup Mess,0,"Подключение порта к сетевому ресурсу",
      vbInformation
    End If
    '*************                                              Конец
*******************************************
```

Запуск сценариев на удаленных машинах. Контроль за ходом выполнения таких сценариев

Начиная с версии 5.6 сценарии WSH можно запускать не только на локальной машине, но и на других компьютерах, имеющихся в сети (это может быть очень удобно для централизованного администрирования удаленных рабочих станций).

Такие WSH-сценарии называются *удаленными сценариями (remote scripts)*. При этом файл со сценарием может находиться либо на локальной машине, либо на общедоступном сетевом ресурсе. На жесткий диск удаленной машины файл сценария копироваться не будет — вместо этого текст сценария по коммуникационному протоколу DCOM — Distributed COM (распределенный COM) передается непосредственно в память процесса, запускаемого на этой машине.

Естественно, такой механизм запуска сценариев является весьма мощным средством удаленного администрирования, однако он также может быть использован для быстрого и практически незаметного распространения по сети вирусов. Поэтому при использовании удаленных сценариев WSH предусмотрены довольно строгие меры безопасности.

Во-первых, и на локальной и на удаленной машинах должны быть установлены операционные системы Windows NT (SP 3)/Windows 2000/Windows XP (системы Windows 95/98/ME не поддерживаются).

Во-вторых, пользователь, который запускает сценарий, должен входить в группу локальных администраторов на той машине, где должен выполняться сценарий.

В-третьих, удаленная машина должна быть предварительно настроена для выполнения удаленных сценариев (по умолчанию после первоначальной установки выполнение таких сценариев запрещено). Для этого необходимо записать 1 в следующий параметр системного реестра:

`HKLM\Software\Microsoft\Windows Script Host\Settings\Remote` **(А)**

(если этот параметр не существует, его нужно создать). Если значением этого параметра является 0, то это означает, что выполнение удаленных сценариев на машине запрещено.

Для того чтобы разрешить выполнение удаленных сценариев на уровне пользователя, необходимо создать параметр

`HKCU\Software\Microsoft\Windows Script Host\Settings\Remote` **(Б)**

и также записать в него 1. Если значением этого параметра является 0,

то это означает, что выполнение удаленных сценариев для текущего пользователя запрещено.

Также при настройке режима выполнения удаленных сценариев нужно проверить значение параметра

`HKLM\Software\Microsoft\Windows Script Host\Settings\IgnoreUserSettings` **(В)**

Если значением этого параметра является 1, то параметр **(Б)** игнорируется и проверяется только значение параметра **(А)**. Если же значением параметра **(В)** является 0, то WSH сначала проверяет параметр **(Б)** и только в случае его отсутствия принимается во внимание значение параметра **(А)**.

Если удаленные сценарии нужно выполнять на машине с операционной системой Windows XP, то на этой машине нужно перерегистрировать сервер wscript.exe с помощью следующей команды:

```
wscript.exe -regserver
```

Удаленные сценарии всегда запускаются с помощью сервера wscript.exe, причем в этих сценариях не поддерживается вывод на экран удаленного компьютера никаких элементов пользовательского интерфейса (не выводятся даже диалоговые окна с сообщениями о возникающих в ходе выполнения ошибках). Другими словами, в удаленных сценариях по умолчанию нельзя использовать методы `WScript.Echo` или `WshShell.Popup` (это может привести к непредсказуемым результатам).

Для примера рассмотрим сценарий RemoteShortcut.js (листинг 2.52), который создает ярлык в специальной папке AllUserDesktop (рабочий стол для всех пользователей). Предположим, что этот сценарий находится в корневом каталоге диска D:, а запустить сценарий необходимо на компьютере \\Stand.

Листинг 2.52. Сценарий для запуска на удаленной машине (JScript)

```
/*****************************************************************
/*                                                          Имя:
RemoteShortcut.js                                          */
/*      Описание:      Создание      ярлыка      на      рабочем
столе                    */
/*****************************************************************
var WshShell,MyShortcut,PathTarg,PathShortcut;
//Создаем объект WshShell
WshShell = WScript.CreateObject("WScript.Shell");
//Определяем путь к папке "AllUsersDesktop" (рабочий стол
```

```
    //всех пользователей)
    PathShortcut = WshShell.SpecialFolders("AllUsersDesktop");
    //Создаем объект-ярлык
    MyShortcut   =   WshShell.CreateShortcut(PathShortcut+   "\\From
Remote WSH.lnk");
    //Устанавливаем путь к файлу
    PathTarg=WshShell.ExpandEnvironmentStrings("%windir%\\notepad.ex
    MyShortcut.TargetPath = PathTarg;
    MyShortcut.Save();  //Сохраняем ярлык
    /*************                                          Конец
*******************************************/
```

Для запуска сценария RemoteShortcut.js на удаленном компьютере \\Stand нужно создать другой сценарий RunRemoteScript.js (листинг 2.53). Здесь вначале создается объект `WshController`:

```
    Controller = WScript.CreateObject("WshController");
```

Затем мы получаем ссылку на экземпляр объекта `WshRemote` на машине \\Stand, соответствующий сценарию с текстом, взятым из файла D:\RemoteScript.js:

```
    RemScript    =    Controller.CreateScript("D:\\RemoteScript.js",
"stand");
```

Запускается удаленный сценарий с помощью метода `Execute`:

```
    RemScript.Execute();
```

После этого нужно дождаться окончания работы сценария на удаленной машине, что делается путем контроля в цикле `while` свойства `Status` объекта `WshRemote` (значение свойства status, равное 2, говорит о том, что выполнение удаленного сценария завершено):

```
    while (RemScript.Status != 2)
     //Цикл выполняется до завершения удаленного сценария
     WScript.Sleep(100); //Приостанавливаем сценарий на 0,1 сек
```

Метод `Sleep` объекта `WScript` вызывается в цикле для того, чтобы освободить процессор во время ожидания завершения удаленного сценария (листинг 2.53).

Листинг 2.53. Запуск удаленного сценария (JScript)

```
    /*************************************************************
    /*                                                        Имя:
RunRemoteScript.js                                          */
    /*                                                       Язык:
JScript                                                    */
    /*            Описание:            Запуск            удаленного
сценария                            */
```

```
    /*************************************************************
    var Controller, RemScript;  //Объявляем переменные
    //Создаем объект WshController
    Controller = WScript.CreateObject("WshController");
    //Создаем сценарий на удаленной машине (объект WshRemote)
    RemScript    =    Controller.CreateScript("D:\\RemoteScript.js",
"stand");
    RemScript.Execute();  //Запускаем удаленный сценарий
    WScript.Echo("Удаленный сценарий запущен");
    while (RemScript.Status != 2)
     //Цикл выполняется до завершения удаленного сценария
     WScript.Sleep(100);  //Приостанавливаем сценарий на 0,1 сек
    WScript.Echo("Выполнение удаленного сценария завершено");
    /*************                                          Конец
*****************************************/
```

В листинге 2.54 приведен аналог сценария RunRemoteScript.js на языке VBScript.

Листинг 2.54. Запуск удаленного сценария (VBScript)

```
    '************************************************************
    ' Имя: RunRemoteScript.vbs
    ' Язык: VBScript
    ' Описание: Запуск удаленного сценария
    '************************************************************
    Option Explicit
    Dim Controller, RemScript  ' Объявляем переменные
    ' Создаем объект WshController
    Set Controller = WScript.CreateObject("WshController")
    ' Создаем сценарий на удаленной машине (объект WshRemote)
    Set  RemScript  =  Controller.CreateScript("D:\\RemoteScript.js",
"stand")
    RemScript.Execute  ' Запускаем удаленный сценарий
    WScript.Echo "Удаленный сценарий запущен"
    While RemScript.Status <> 2
     ' Цикл выполняется до завершения удаленного сценария
     WScript.Sleep 100  ' Приостанавливаем сценарий на 0,1 сек
    Wend
    WScript.Echo "Выполнение удаленного сценария завершено"
    '*************                                          Конец
*****************************************
```

Контролировать ход выполнения удаленных сценариев можно не только путем анализа свойства `Status`, но и с помощью обработки событий `Start` (запуск сценария), `Error` (ошибка при выполнении сценария) и `End`

(завершение работы сценария) объекта WshRemote; соответствующие примеры сценариев на языках JScript и VBScript приведены в листингах 2.55 и 2.56.

Напомним, что для обработки событий объекта нужно в сценарии сначала создать экземпляр этого объекта, а затем соединиться с ним при помощи метода `ConnectObject`, указав нужный префикс для функций-обработчиков:

```
    Controller = WScript.CreateObject("WshController");
    RemScript  =  Controller.CreateScript("D:\\RemoteScript.js  ",
"stand");
    WScript.ConnectObject(RemScript, "RemoteScript_");
```

Затем в тексте сценария описываются функции `RemoteScript_Start`, `RemoteScript_Error` и `RemoteScript_End`, управление в которые будет передаваться при наступлении соответствующих событий.

Листинг 2.55. Обработка событий объекта `WshRemote` (JScript)

```
    /*****************************************************************
    /*                                                          Имя:
RemoteEvents.js                                                 */
    /*                                                          Язык:
JScript                                                        */
    /*  Описание:  Обработка  событий,  возникающих  при  выполнении
удаленного */
    /*
сценария                                                  */
    /*****************************************************************
    var Controller,RemScript,IsQuit;  //Объявляем переменные
    //Создаем объект WshController
    Controller = WScript.CreateObject("WshController");
    //Создаем сценарий на удаленной машине (объект WshRemote)
    RemScript  =  Controller.CreateScript("D:\\RemoteScript.js  ",
"stand");
    //Устанавливаем соединение с объектом WshRemote
    WScript.ConnectObject(RemScript, "RemoteScript_");
    RemScript.Execute();  //Запускаем удаленный сценарий
    IsQuit = false;
    while   (!IsQuit)   WScript.Sleep(100);      //Приостанавливаем
сценарий на 0,1 сек
    WScript.Quit();  //Выходим из сценария
    /***************              Функции-обработчики       событий
**********************/
    function RemoteScript_End() { //Событие End
```

```
     WScript.Echo("Выполнение удаленного сценария завершено");
     IsQuit = true;
    }
    function RemoteScript_Error() { //Событие Error
     //Выводим на экран описание возникшей ошибки
     WScript.Echo("Ошибка при выполнении удаленного сценария: " +
      RemScript.Error.Description);
     IsQuit = true;
    }
    function RemoteScript_Start() { //Событие Start
     WScript.Echo("Удаленный сценарий запущен");
    }
    /*************                                                Конец
*****************************************/
```

Листинг 2.56. Обработка событий объекта WshRemote (VBScript)

```
    '*************************************************************
    ' Имя: RemoteEvents.vbs
    ' Язык: VBScript
    '  Описание:  Обработка  событий,  возникающих  при  выполнении
удаленного
    '           сценария
    '*************************************************************
    Option Explicit
    Dim Controller,RemScript,IsQuit  ' Объявляем переменные
    ' Создаем объект WshController
    Set Controller = CreateObject("WshController")
    ' Создаем сценарий на удаленной машине (объект WshRemote)
    Set  RemScript  =  Controller.CreateScript("D:\RemoteScript.js ",
"stand")
    ' Устанавливаем соединение с объектом WshRemote
    WScript.ConnectObject RemScript, "RemoteScript_"
    RemScript.Execute  ' Запускаем удаленный сценарий
    IsQuit = False
    While Not IsQuit
     WScript.Sleep 100  ' Приостанавливаем сценарий на 0,1 сек
    Wend
    WScript.Quit ' Выходим из сценария
    '***************             Функции-обработчики       событий
**********************
    Function RemoteScript_End()  ' Событие End
     WScript.Echo "Выполнение удаленного сценария завершено"
     IsQuit = True
    End Function
```

```
    Function RemoteScript_Error()  ' Событие Error
     ' Выводим на экран описание возникшей ошибки
     WScript.Echo "Ошибка при выполнении удаленного сценария: " & _
      RemScript.Error.Description
     IsQuit = True
    End Function
    Function RemoteScript_Start()  ' Событие Start
     WScript.Echo "Удаленный сценарий запущен"
    End Function
    '*************                                                Конец
*********************************************
```

Замечание

При контроле за ходом выполнения удаленного сценария с помощью обработки событий объекта `WshRemote` затрачивается больше ресурсов компьютера по сравнению с простой проверкой свойства `Status`. Кроме этого, при обработке событий увеличивается сетевой трафик между локальной и удаленной машинами.

Глава 3
Сценарии WSH как приложения XML

До сих пор мы рассматривали простые одиночные файлы сценариев, в которых мог использоваться язык JScript или VBScript. В версии WSH 1.0 это был единственный поддерживаемый тип сценариев, причем используемый язык определялся по расширению файла: js для JScript и vbs для VBScript. Начиная с WSH 2.0 появилась возможность создавать сценарии, в которых можно применять оба языка одновременно. Для таких сценариев в операционной системе регистрируется расширение wsf; wsf-файлы мы будем далее называть просто WS-файлами. Новый тип сценариев (WS-файл) имеет еще несколько важных преимуществ перед одиночными файлами сценариев WSH 1.0:

□ поддерживаются вложенные файлы;

□ возможен доступ из сценария к внешним мнемоническим константам, которые определены в библиотеках типов используемых объектов ActiveX;

□ в одном WS-файле можно хранить несколько отдельных, независимых друг от друга, сценариев;

□ сценарий становится самодокументируемым, т.е. вывод информации об использовании сценария и его синтаксисе происходит автоматически.

Понятно, что для обеспечения новых возможностей необходимо иметь больше информации, чем ее может предоставить отдельный сценарий. В самом файле сценария должна присутствовать некоторая дополнительная информация, скажем, имя этого сценария (подобная информация содержится, например, в заголовках HTML-страниц). Другими словами, для сценариев WSH должен использоваться уже некий специальный формат, а не просто отдельные js- или vbs-файлы. В качестве такого формата разработчики Microsoft выбрали язык XML — Extensible Markup Language, который уже использовался ими для определения информационной модели в технологии WSC — Windows Script Components, которая позволяет с помощью языков сценариев создавать и регистрировать полноценные COM-объекты.

Таким образом, теперь сценарии WSH не просто содержат в текстовом виде ActiveX-совместимый сценарий, а являются XML-приложениями, поддерживающими схему WS XML — Windows Script XML, которая, в

свою очередь, опирается на схему WSC XML. Поэтому для понимания двух технологий (WSC и WSH) достаточно освоить одну схему XML.

WS-файл рассматривается сервером сценариев как файл с разметкой XML, который должен соответствовать схеме WS XML. Новый тип файла и формат XML обеспечивают более мощную среду для написания сценариев.

Для того чтобы использовать язык XML в сценариях WSH, вовсе не обязательно вникать во все тонкости этого языка, однако основные принципы XML понимать, конечно, нужно.

Основные принципы XML

Проявляемый в настоящее время большой интерес к языку XML объясняется тем, что он предоставляет возможности, позволяющие в текстовой форме описывать структурированные данные. Точнее говоря, XML является метаязыком для создания различных языков разметки, которые способны определять произвольные структуры данных — двоичные данные, записи в базе данных или сценарии. Прежде всего, XML используется в Internet- приложениях при работе браузеров, которые отображают информацию, находящуюся на Web-серверах. При этом пользователю отдельно передаются данные в виде XML-документа, и отдельно — правила интерпретации этих данных для отображения с помощью, например, языков сценариев JScript или VBScript.

Как и HTML, XML является независимым от платформы промышленным стандартом. Полные спецификации XML и связанных с ним языков доступны на официальной странице консорциума W3C — World Wide Web Consortium по адресу http://www.w3c.org/xml.

Внешне XML-документ похож на HTML-документ, т.к. XML-элементы также описываются с помощью *тегов*, т.е. ключевых слов. Однако, в отличие от HTML, в XML пользователь может создавать собственные элементы, поэтому набор тегов не является заранее предопределенным. Еще раз повторим, что теги XML определяют структурированную информацию и, в отличие от тегов HTML, не влияют на то, как браузер отобразит эту информацию. Ниже перечислены несколько основных правил формирования корректного XML-документа:

□ документ XML состоит из элементов разметки (markup) и непосредственно данных (content);

□ все XML-элементы описываются с помощью тегов;

□ в заголовке документа с помощью специальных тегов помещается дополнительная информация (используемый язык разметки, его версия и т.д.);

□ каждый открывающий тег, который определяет область данных, должен иметь парный закрывающий тег (в HTML некоторые закрывающие теги можно опускать);

□ в XML, в отличие от HTML, учитывается регистр символов;

□ все значения атрибутов, используемых в определении тегов, должны быть заключены в кавычки;

□ вложенность элементов в документе XML строго контролируется.

Рассмотрим теперь структуру и синтаксис WS-файлов, использующих схему WS XML.

Схема WS XML

Синтаксис элементов, составляющих структуру WS-файла, в общем виде можно представить следующим образом:

```
<element [attribute1="value1" [attribute2="value2" ... ]]>
 Содержимое (content)
</element>
```

Открывающий тег элемента состоит из следующих компонентов:

□ открывающей угловой скобки "<";

□ названия элемента, написанного строчными буквами;

□ необязательного списка атрибутов со значениями (названия атрибутов пишутся строчными буквами, значения заключаются в двойные кавычки);

□ закрывающей угловой скобки ">".

Например, тег начала элемента

```
<script language="JScript">
```

имеет имя тега `script` и определяет атрибут `language` со значением "`JScript`". Атрибуты предоставляют дополнительную информацию о соответствующем теге или последующем содержимом элемента. В нашем примере атрибут указывает на то, что содержимым элемента является текст сценария на языке JScript.

Закрывающий тег элемента состоит из следующих компонентов:

□ открывающей угловой скобки "<";

□ символа "/";

□ названия элемента, написанного строчными буквами;

□ закрывающей угловой скобки ">".

Таким образом, тег конца элемента не имеет атрибутов, например, `</script>`.

Если у элемента нет содержимого, то он имеет следующий вид:

```
<element [attribute1="value1" [attribute2="value2" ... ]]/>
```

То есть в этом случае элемент состоит из следующих компонентов:

□ открывающей угловой скобки "<";

□ названия элемента, написанного строчными буквами;

□ необязательного списка атрибутов со значениями (названия атрибутов пишутся строчными буквами, значения заключаются в двойные кавычки);

□ символа"/";

□ закрывающей угловой скобки ">".

Пример такого элемента:

```
<script language="JScript" src="tools.js"/>
```

Представленная в листинге 3.1 схема WS XML — это модель данных, определяющая элементы и соответствующие атрибуты, а также связи элементов друг с другом и возможную последовательность появления элементов. Также эта схема может задавать значения атрибутов по умолчанию.

Листинг 3.1. Схема WS XML

```
<?xml version="1.0" standalone="yes"?>
<package>
 <job [id="JobID"]>
  <?job debug="true|false"?>
  <runtime>
            <named   name="NamedName"   helpstring="HelpString"
type="string|boolean|simple" required="true|false" />
          <unnamed   name="UnnamedName"   helpstring="HelpString"
many="true|false" required="true|false" />
   <description> Описание сценария </description>
   <example> Пример запуска сценария </example>
  </runtime>
  <resource id="ResourceID"> Строка или число </resource>
  <object id="ObjID" [classId="clsid:GUID"|progid="ProgID"]/>
      <reference   [object="ProgID"   |   guid=""typelibGUID"]
[version="version"]/>
  <script language="language" [src="strFileURL"]\>
  <script language="language" >
   <![CDATA[
    Код сценария
   ]]>
  </scriipt>
 </job>
 Другие задания
</package>
```

Таким образом, из листинга 3.1 видно, что:

□ элемент <package> может содержать один или несколько элементов <job>;

□ элемент <job> может содержать один или несколько элементов <runtime>, <resource>, <object>, <reference> или <script>;

□ элемент <runtime> может содержать один или несколько элементов <named> и <unnamed>, а также элементы <description> и <example>.

Обязательными для создания корректного сценария являются только элементы `<job>` и `<script>`. Сам код сценария всегда располагается внутри элемента `<script>`.

Опишем теперь элементы XML, использующиеся в сценариях WSH, более подробно.

Элементы WS-файла

В WS-файл можно вставлять комментарии независимо от разметки XML. Сделать это можно двумя способами: с помощью элемента `<!-- -->` или элемента `<comment>`. Например:

```
<!-- Первый комментарий -->
```

или

```
<comment>
Второй комментарий
</comment>
```

Элементы <*?xml?*> и <*![CDATA[]]*>

Эти элементы являются стандартными для разметки W3C XML 1.0. В сценариях WSH они определяют способ обработки WS-файла. Всего существует два режима обработки сценария: нестрогий (loose) и строгий (strict).

При нестрогой обработке (элемент `<?xml?>` отсутствует) не предполагается выполнение всех требований стандарта XML. Например, не требуется различать строчные и заглавные буквы и заключать значения атрибутов в двойные кавычки. Кроме этого, в процессе нестрогой обработки считается, что все содержимое между тегами `<script>` и `</script>` является исходным кодом сценария. Однако при таком подходе может произойти ошибочная интерпретация вложенных в сценарий зарезервированных для XML символов или слов как разметки XML. Например, имеющиеся в коде сценария знаки "меньше" (<) и "больше" (>) могут привести к прекращению разбора и выполнения сценария.

Для того чтобы задать режим строгой обработки сценария, нужно поместить элемент `<?xml?>` в самой первой строке сценария — никаких других символов или пустых строк перед ним быть не должно. При такой обработке WS-файла нужно четко следовать всем правилам стандарта XML. Код сценария должен быть помещен в секцию `CDATA`, которая начинается с символов "`<![CDATA[`" и заканчивается символами "`]]>`".

Замечание

В WSH 5.6 названия и значения атрибутов в элементе `<?xml?`

> должны быть именно такими, как в листинге 3.1 (version="1.0" и standalone="yes").\

Элемент <*?job?*>

Элемент `<?job?>` задает режим отладки при выполнении WS-файла. Если значение атрибута `debug` равно `true`, то задание может быть выполнено во внешнем отладчике *(см. приложение 3)*. Если же значение атрибута `debug` равно `false`, то отладчик для этого задания применен быть не может. По умолчанию `debug` имеет значение `false`.

Элемент <*package*>

Этот элемент необходим в тех WS-файлах, в которых с помощью элементов `<job>` определено более одного задания. В этом случае все эти задания должны находиться внутри пары тегов `<package>` и `</package>` (см. листинг 3.1). Другими словами, `<package>` является контейнером для элементов `<job>`.

Если же в WS-файле определено только одно задание, то элемент `<package>` можно не использовать.

Элемент <*job*>

Элементы `<job>` позволяют определять несколько заданий (независимо выполняющихся частей) в одном WS-файле. Иначе говоря, между тегами `<job>` и `</job>` будет находиться отдельный сценарий (который, в свою очередь, может состоять из нескольких частей, написанных, возможно, на разных языках).

У элемента `<job>` имеется единственный атрибут `id`, который определяет уникальное имя задания. Например, в сценарии two_jobs.wsf определяются два задания с именами "`Task1`" и "`Task2`" (листинг 3.2).

Листинг 3.2. Файл two_jobs.wsf

```
<package>
<job id="Task1">
<!-- Описываем первое задание (id="Task1") -->
<script language="VBScript">
```

```
 WScript.Echo "Выполняется первое задание (VBScript)"
</script>
</job>
<job id="Task2">
<!-- Описываем второе задание (id="Task1") -->
<script language="JScript">
 WScript.Echo "Выполняется второе задание (JScript)"
</script>
</job>
</package>
```

Для того чтобы запустить конкретное задание из многозадачного WS-файла, нужно воспользоваться параметром `//job:"JobID"` в командной строке WSH. Например, следующая команда:

```
cscript //job:"Task1" two_jobs.wsf
```

запускает с помощью cscript.exe задание с именем "Task1" из файла two_jobs.wsf.

Замечание

Если параметр `//job` не указан, то по умолчанию из многозадачного WS-файла запускается первое задание.

Если в WS-файле имеется несколько заданий, то они должны находиться внутри элемента `<package>`. Элемент `<job>` является одним из двух обязательных элементов в сценариях WSH с разметкой XML.

Элемент <*runtime*>

При запуске почти всех стандартных команд или утилит командной строки Windows с ключом `/?` на экран выводится встроенная справка, в которой кратко описываются назначение и синтаксис этой команды или утилиты (рис. 3.1).

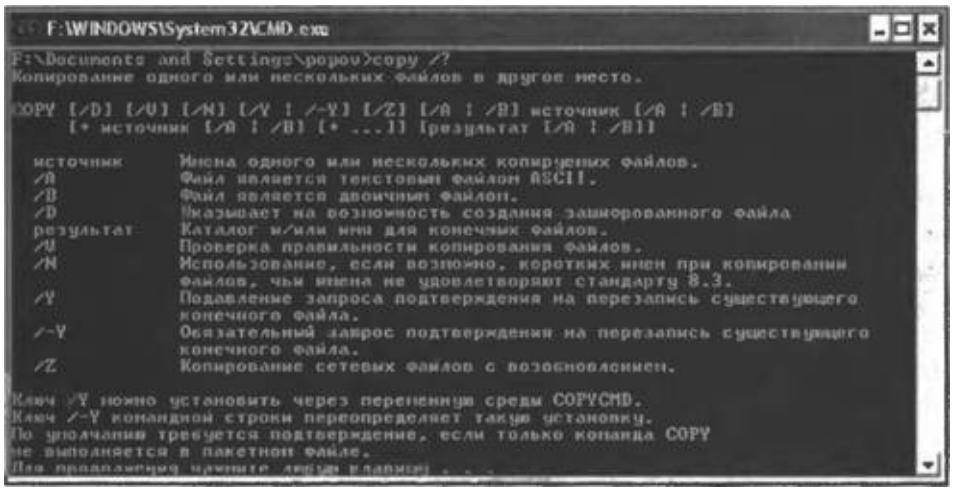


Рис. 3.1. Встроенная справка для команды `COPY`

Хорошим тоном считается создание такой справки и для разрабатываемых сценариев WSH. Понятно, что добавление в сценарий функции вывода информации о назначении, синтаксисе и аргументах этого сценария потребовало бы написания довольно большого количества кода: необходимо следить за ключом `/?` в командной строке, а при добавлении нового параметра командной строки возникнет необходимость изменения функции, отвечающей за вывод информации на экран.

Элемент `<runtime>` позволяет сделать сценарий самодокументируемым, т.е. в этом случае при задании в командной строке ключа `/?` на экран будет автоматически выводиться информация об использовании сценария, о его синтаксисе и аргументах (именных и безымянных), а также пример запуска сценария с конкретными значениями аргументов.

При этом сам элемент `<runtime>` является лишь контейнером, а содержимое для вывода информации хранится в элементах `<named>` (описание именных параметров командной строки), `<unnamed>` (описание безымянных параметров командной строки), `<description>` (описание самого сценария) и `<example>` (пример запуска сценария), которые находятся внутри `<runtime>`.

Замечание

Элемент `<runtime>` является дочерним относительно `<job>`, поэтому все описания, приведенные внутри `<runtime>`, относятся только к текущему заданию.

Элемент <*named*>

С помощью элементов <named> можно описывать (документировать) именные параметры командной строки сценария. В табл. 3.1 приведено описание аргументов элемента <named>.

Таблица 3.1. *Аргументы элемента* <named>

Аргумент	Описание
name	Задает имя параметра командной строки
helpstring	Строка, содержащая описание параметра командной строки
type	Определяет тип параметра командной строки. Может принимать значения "string" (символьный тип), "boolean" (логический тип), "simple" (в сценарий передается только имя параметра без дополнительного значения). По умолчанию используется тип "simple"
required	Используется для того, чтобы показать, является ли параметр командной строки обязательным. Может принимать значения "true" (параметр нужно указывать обязательно) и "false" (параметр можно не указывать)

Информация, которая указывается для объявляемого в элементе <named> параметра командной строки, используется только для самодокументируемости сценария и никак не влияет на реальные значения, которые будут указаны в командной строке при запуске сценария. Например, если параметр объявлен как обязательный (required="true"), но в действительности не был указан при запуске сценария, то никакой ошибки во время работы не произойдет.

Если для аргумента командной строки сценария указан тип "string", то предполагается, что этот аргумент имеет имя и значение, разделенные символом ":", например:

```
/Имя:"Андрей Попов" /Возраст:30
```

Если в качестве типа параметра командной строки используется "simple", то для этого параметра в командной строке указывается только его имя без значения:

```
/Имя /Возраст
```

Для того чтобы передать в сценарий аргумент командной строки типа "boolean", нужно после имени этого аргумента указать символ "+" (соответствует логическому значению "истина") или "-" (соответствует

значению "ложь"). Например:

```
/Запись+ /ReWrite-
```

В листинге 3.3 приведен сценарий named.wsf, в котором в блоке `<runtime>` описываются три именных аргумента командной строки:

- `/Имя` (обязательный аргумент символьного типа);
- `/Компьютер` (необязательный аргумент символьного типа);
- `/Новый` (обязательный аргумент логического типа).

После запуска с помощью wscript.exe в сценарии named.wsf сначала вызывается метод `WScript.Arguments.Usage`, в результате чего на экран выводится диалоговое окно с информацией о сценарии и параметрах командной строки (рис. 3.2).

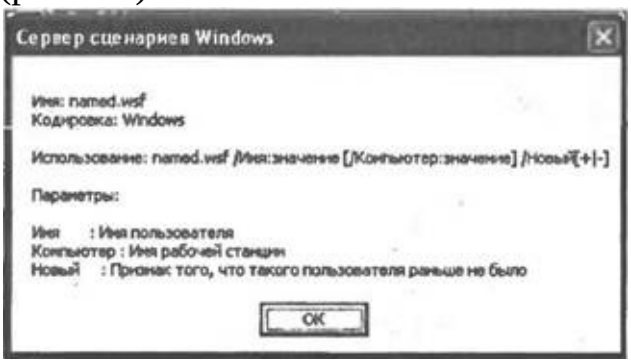


Рис. 3.2. Диалоговое окно с информацией о параметрах сценария named.wsf

Затем в сценарии проверяется, какие именно аргументы командной строки были подставлены при запуске, и выделяются значения этих аргументов. Для этого создается объект WshNamed, являющийся коллекцией именных аргументов командной строки, и используется метод Exists этого объекта:

```
//Создаем объект WshNamed – коллекция именных аргументов сценария
objNamedArgs= WScript.Arguments.Named;
s="";
//Проверяем, существует ли аргумент /Имя:
if (objNamedArgs.Exists("Имя"))
 //Получаем значение символьного аргумента /Имя
 s+="Имя: "+objNamedArgs("Имя") +"\n";
//Проверяем, существует ли аргумент /Компьютер:
if (objNamedArgs.Exists("Компьютер"))
 //Получаем значение символьного аргумента /Компьютер
 s+="Машина: "+objNamedArgs("Компьютер") + "\n";
```

Значением параметра `/Новый` является константа логического типа (`true` или `false`), поэтому для формирования строки, соответствующей

этому значению, используется условный оператор языка JScript:

```
    //Проверяем, существует ли аргумент /Новый
    if (objNamedArgs.Exists("Новый"))
     //Получаем с помощью условного оператора значение
     //логического аргумента /Новый
      s+="Новый пользователь: "+(objNamedArgs("Новый") ? "Да" :
"Нет");
```

Если запустить сценарий named.wsf следующим образом:

```
wscript.exe named.wsf /Имя:Popov /Компьютер:404_Popov /Новый+
```

то на экран будет выведено диалоговое окно, показанное на рис. 3.3.

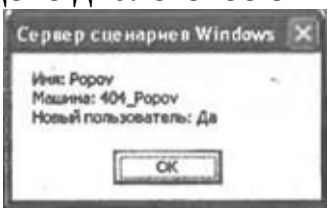


Рис. 3.3. Значения именных аргументов командной строки, переданных в named.wsf

Листинг 3.3. Файл named.wsf

```
    <job id="Named">
     <runtime>
      <description>
      Имя: named.wsf
      Кодировка: Windows
      </description>
      <named
       name="Имя"
       helpstring="Имя пользователя"
       type="string" required="true"/>
      <named
       name="Компьютер"
       helpstring="Имя рабочей станции"
       type="string" required="false"/>
      <named
       name="Новый"
        helpstring="Признак того, что такого пользователя раньше не
было"
       type="boolean" required="true"/>
     </runtime>
     <script language="JScript">
      var objNamedArgs,s;
```

```
      s="";
       //Вызываем метод ShowUsage для вывода на экран описания
сценария
      WScript.Arguments.ShowUsage();
       //Создаем объект WshNamed - коллекция именных аргументов
сценария
      objNamedArgs= WScript.Arguments.Named;
      //Проверяем, существует ли аргумент /Имя:
      if (objNamedArgs.Exists("Имя"))
       //Получаем значение символьного аргумента /Имя
       s+="Имя: "+objNamedArgs("Имя")+"\n";
      //Проверяем, существует ли аргумент /Компьютер:
      if (objNamedArgs.Exists("Компьютер"))
       //Получаем значение символьного аргумента /Компьютер
       s+="Машина: "+objNamedArgs("Компьютер")+"\n";
      //Проверяем, существует ли аргумент /Новый
      if (objNamedArgs.Exists("Новый"))
       //Получаем с помощью условного оператора значение
       //логического аргумента /Новый
        s+="Новый пользователь: "+(objNamedArgs("Новый") ? "Да" :
"Нет");
      //Выводим полученные строки на экран
      WScript.Echo(s);
     </script>
    </job>
```

Элемент <*unnamed*>

С помощью элементов <unnamed> можно описывать (документировать) безымянные параметры командной строки сценария. В табл. 3.2 приведено описание аргументов элемента <unnamed>.

Таблица 3.2. *Аргументы элемента* <unnamed>

Аргумент	Описание
name	Задает имя, которое будет указано для описываемого параметра командной строки при выводе информации о сценарии
helpstring	Строка, содержащая описание параметра командной строки
many	Определяет, сколько раз может быть указан безымянный параметр в командной строке. Значение, равное "true" (используется по умолчанию), означает, что безымянный параметр может встретиться в командной строке более одного раза. Значение, равное "false", означает, что безымянный

	параметр должен быть указан только один раз
`required`	Определяет, является ли безымянный параметр командной строки обязательным. Может принимать значения "`true`", "`on`" или 1 (параметр нужно указывать обязательно), "`false`", "`off`" или 0 (параметр можно не указывать). Также значением аргумента "required" может быть целое число, которое показывает, сколько раз безымянный параметр должен обязательно быть указан в командной строке

Информация, которая указывается для объявляемого в элементе `<unnamed>` параметра командной строки, используется, как и в случае элемента `<named>`, только для самодокументируемости сценария и никак не влияет на реальные значения, которые будут указаны в командной строке при запуске сценария. Например, если безымянный параметр объявлен как обязательный (`required="true"`), но в действительности не был указан при запуске сценария, то никакой ошибки во время работы не произойдет.

Рассмотрим в качестве примера сценарий unnamed.wsf, в который в качестве параметров командной строки должны передаваться расширения файлов, причем обязательно должны быть указаны хотя бы два таких расширения (листинг 3.4).

Для создания информации об использовании этого сценария создается элемент `<unnamed>` следующего вида:

```
<unnamed name="Расш" helpstring="Расширения файлов" many="true" required=2/>
```

После запуска с помощью wscript.exe в сценарии unnamed.wsf сначала вызывается метод `WScript.Arguments.Usage`, в результате чего на экран выводится диалоговое окно с информацией о сценарии и параметрах командной строки (рис. 3.4).

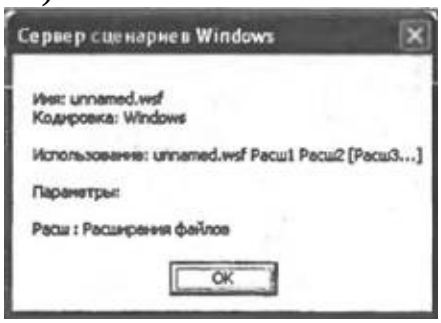


Рис. 3.4. Диалоговое окно с информацией о параметрах сценария unnamed.wsf

Затем в сценарии создается коллекция `objUnnamedArgs` (объект

WshUnnamed), которая содержит все безымянные аргументы командной строки, реально переданные в сценарий:

```
    objUnnamedArgs=WScript.Arguments.Unnamed;    //Создаем объект
WshUnnamed
```

После этого определяется общее число реально переданных в сценарий параметров командной строки (свойство length) и в цикле while организуется перебор всех элементов коллекции objUnnamedArgs.

```
    //Определяем количество безымянных аргументов
    s="Передано в сценарий безымянных аргументов:
"+objUnnamedArgs.length;
    for (i=0; i<=objUnnamedArgs.length-1; i++)
     //Формируем строки со значениями безымянных аргументов
     s+="\n"+objUnnamedArgs(i);
    //Выводим полученные строки на экран
    WScript.Echo(s);
```

Если запустить сценарий unnamed.wsf следующим образом:

```
wscript.exe unnamed.wsf vbs js
```

то на экран будет выведено диалоговое окно, показанное на рис. 3.5.

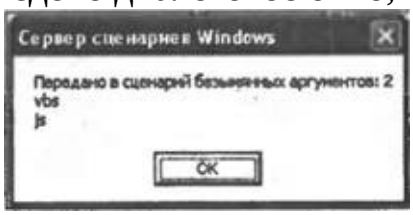


Рис. 3.5. Значения безымянных аргументов командной строки, переданных в unnamed.wsf

Листинг 3.4. Файл unnamed.wsf

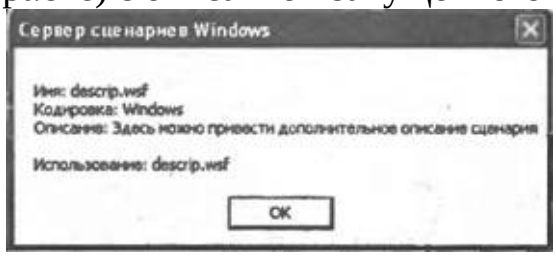
```
    <job id="Unnamed">
     <runtime>
      <description>
      Имя: unnamed.wsf
      Кодировка: Windows
      </description>
          <unnamed   name="Расш"   helpstring="Расширения   файлов"
many="true" required="2"/>
     </runtime>
     <script language="JScript">
     var objUnnamedArgs,s;
      //Вызываем метод ShowUsage для вывода на экран описания
сценария
```

```
     WScript.Arguments.ShowUsage();
      objUnnamedArgs=WScript.Arguments.Unnamed;    //Создаем объект
WshUnnamed
     //Определяем количество безымянных аргументов
        s="Передано    в    сценарий    безымянных    аргументов:
"+objUnnamedArgs.length;
     for (i=0; i<=objUnnamedArgs.length-1; i++)
      //Формируем строки со значениями безымянных аргументов
      s+="\n"+objUnnamedArgs(i);
     //Выводим полученные строки на экран
     WScript.Echo(s);
     </script>
    </job>
```

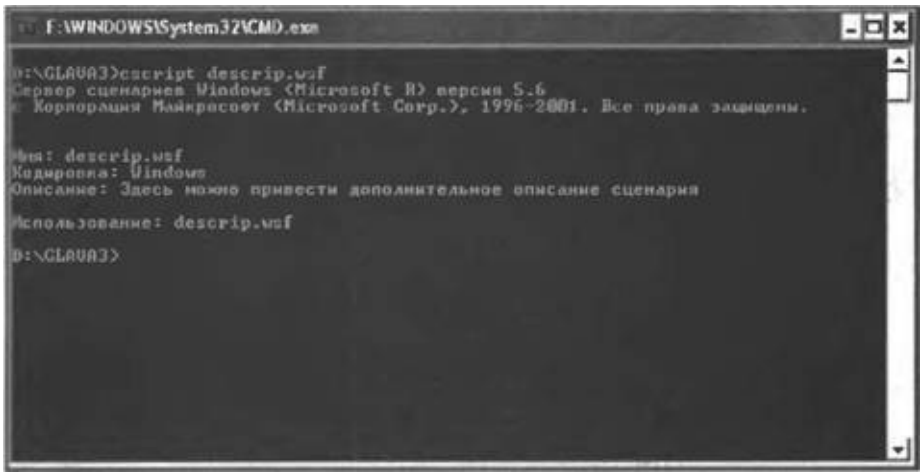
Элемент <*description*>

Внутри элемента `<description>` помещается текст (без дополнительных кавычек), описывающий назначение сценария. Как и все элементы внутри `<runtime>`, этот текст выводится на экран, если сценарий был запущен с ключом `/?` в командной строке или если в сценарии встретился вызов метода `ShowUsage` объекта `WshArguments`. При выводе текста на экран учитываются все имеющиеся в нем пробелы, символы табуляции и перевода строки.

Пример использования элемента <description> и метода ShowUsage представлен в сценарии descrip.wsf (листинг 3.5). Здесь сразу вызывается метод WScript.Arguments.ShowUsage, в результате чего на экран выводится диалоговое окно (в случае запуска сценария с помощью wscript.exe) (рис. 3.6, *а)* или просто строки текста (в случае запуска сценария с помощью cscript.exe) с описанием запущенного сценария (рис. 3.6, б).



а



б

Рис. 3.6. Вывод текста, описывающего сценарий: а — в графическом режиме; б — в консольном режиме

Листинг 3.5. Файл descrip.wsf

```
<job id="Descrip">
 <runtime>
  <description>
  Имя: descrip.wsf
  Кодировка: Windows
    Описание: Здесь можно привести дополнительное описание
сценария
  </description>
 </runtime>
 <script language="JScript">
 //Вызываем метод ShowUsage
 WScript.Arguments.ShowUsage();
 </script>
</job>
```

Элемент <*example*>

Внутри элемента `<example>` приводится текст из одной или нескольких строк, в котором можно описать примеры запуска сценария. Если сценарий был запущен с ключом `/?` в командной строке или в сценарии встретился вызов метода `ShowUsage` объекта `WshArguments`, то этот текст выводится в графическое диалоговое окно (при использовании wscript.exe) или на экран (в консольном режиме при использовании cscript.exe). При выводе текста

на экран учитываются все имеющиеся в нем пробелы, символы табуляции и перевода строки, при этом строки из элемента <example> выводятся после строк из элемента <description> (рис. 3.7).

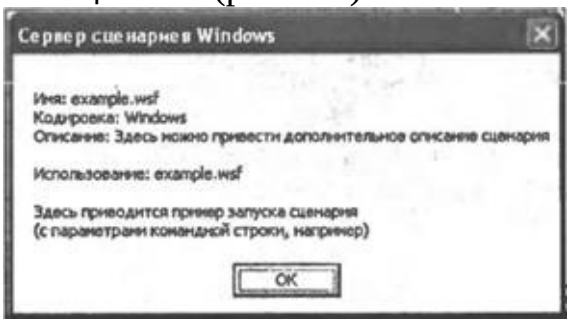


Рис. 3.7. Диалоговое окно, формируемое элементами <description> и <example>

Сценарий example.wsf, диалоговое окно с описанием которого показано на рис. 3.7, приведен в листинге 3.6.

Листинг 3.6. Файл example.wsf

```
<job id="Example">
 <runtime>
  <description>
  Имя: example.wsf
  Кодировка: Windows
    Описание: Здесь можно привести дополнительное описание
сценария
  </description>
  <example>

  Здесь приводится пример запуска сценария
  (с параметрами командной строки, например)
  </example>
 </runtime>
 <script language="JScript">
 //Вызываем метод ShowUsage
 WScript.Arguments.ShowUsage();
 </script>
</job>
```

Элемент <*resource*>

Элемент <resource> позволяет отделить символьные или числовые

константы (ресурсы) от остального кода сценария. Например, таким образом удобно собрать в одном месте строки, которые используются в сценарии для вывода каких-либо стандартных сообщений. Если после этого понадобится изменить сообщения в сценарии (например, перевести их на другой язык), то достаточно будет внести соответствующие корректировки в строки, описанные в элементах <resource>.

Для получения значения ресурса в сценарии нужно вызвать метод getResource, передав в качестве параметра символьный идентификатор ресурса (значение атрибута id).

В листинге 3.7 представлен пример сценария resource.wsf, в котором определяется ресурсная строка с идентификатором "MyName":

```
<resource id="MyName"> Меня зовут Андрей Попов </resource>
```

Значение этого ресурса затем выводится на экран с помощью метода Echo **о**бъекта WScript и метода getResource:

```
WScript.Echo(getResource("MyName"));
```

Листинг 3.7. Файл resource.wsf

```
<job id="Resource">
 <runtime>
  <description>
  Имя: resource.wsf
  Описание: Пример использования в сценарии ресурсных строк
  </description>
 </runtime>
 <resource id="MyName">
 Меня зовут Андрей Попов
 </resource>
 <script language="JScript">
 //Выводим на экран значение ресурса "MyName"
 WScript.Echo(getResource("MyName"));
 </script>
</job>
```

Элемент <*object*>

Элемент <object> предлагает еще один способ создания экземпляра COM-объектов для использования их внутри сценариев. Напомним, что ранее для этого мы использовали методы CreateObject и GetObject объекта WScript, объект ActiveXObject и функцию GetObject языка JScript, а также

функцию `CreateObject` языка VBScript. Элемент `<object>` может заменить эти средства.

Атрибут `id` в `<object>` — это имя, применяемое для обращения к объекту внутри сценария. Отметим, что объект, создаваемый с помощью тега `<object>`, будет глобальным по отношению к тому заданию, в котором он определен. Другими словами, этот объект может использоваться во всех элементах `<script>`, находящихся внутри элемента `<job>`, содержащего описание объекта.

Атрибуты `classid` и `progid` используются в `<object>` соответственно для указания *глобального кода* создаваемого объекта (GUID, Globally Unique ID) или *программного кода* объекта (Programmic Identifier). Из этих двух необязательных атрибутов может быть указан только один. Например, создать объект `FileSystemObject (GUID="0D43FE01-F093-11CF-8940-00A0C9054228")` можно двумя способами:

```
<object id="fso" classid="clsid:0D43FE01-F093-11CF-8940-00A0C9054228"/>
```

или

```
<object id="fso" progid="Scripting.FileSystemObject"/>
```

В обычном js-файле или внутри элемента `<script>` этот объект мы бы создали следующим образом:

```
var fso = WScript.CreateObject("Scripting.FileSystemObject");
```

или

```
var fso = new ActiveXObject("Scripting.FileSystemObject");
```

Элемент <*reference*>

При вызове многих методов внешних объектов, которые используются внутри сценариев, требуется указывать различные числовые или строковые константы, определенные в этих внешних объектах. Например, для того, чтобы открыть текстовый файл с помощью метода `OpenTextFile` объекта `FileSystemObject`, может потребоваться указать параметр, который определяет режим ввода/вывода (возможные значения констант `ForReading=1`, `ForWriting=2` и `ForAppending=8`) открываемого файла. Ясно, что запомнить все значения констант различных объектов очень трудно, поэтому при их использовании приходится постоянно обращаться к справочной информации.

К счастью, большинство объектов предоставляет информацию об именах используемых ими констант в своей библиотеке типов (как уже отмечалось в *главе* 2, библиотека типов регистрируется в системном

реестре при установке COM-объекта и может существовать как в виде отдельного файла с расширением tlb, так и в виде части файла с исполняемым кодом объекта). Элемент `<reference>` как раз обеспечивает доступ к мнемоническим константам, определенным в библиотеке типов объекта (экземпляр объекта при этом не создается).

Для того чтобы воспользоваться константами определенного объекта, нужно в теге `<reference>` указать программный код этого объекта (атрибут `object`) или *глобальный код* его библиотеки типов (атрибут `guid`), а также, при необходимости, номер версии объекта (атрибут `version`).

Например, доступ к константам объекта `FileSystemObject` организуется следующим образом:

```
<reference object="Scripting.FileSystemObject"/>
```

После этого в сценариях можно просто использовать константы с именами `ForReading` или `ForAppending`, не заботясь об их числовых значениях (соответствующий пример приведен в листинге 3.10).

Элемент <*script*>

Элемент `<script>` с помощью атрибута `language` позволяет определить язык сценария (`language="JScript"` для языка JScript и `language="VBScript"` для языка VBScript). Это делает возможным использовать в одном задании сценарии, написанные на разных языках, что иногда бывает очень удобно. Предположим, что у вас имеются сценарии на JScript и VBScript, функции которых необходимо объединить. Для этого не нужно переписывать один из сценариев на другой язык — используя WS-файл, можно из сценария JScript спокойно вызывать функции, написанные на VBScript, и наоборот! Пример подобного сценария приведен в листинге 3.12.

Атрибут `src` позволяет подключить к выполняющемуся сценарию внешний файл с другим сценарием. Например, задание элемента

```
<script language="JScript" src="tools.js"/>
```

приведет к такому же результату, как если бы содержимое файла tools.js было расположено между тегами `<script>` и `</script>`:

```
<script language="JScript">
 Содержимое файла tools.js
</script>
```

Таким образом, можно выделить код, который должен использоваться в нескольких сценариях, поместить его в один или несколько внешних файлов, а затем по мере необходимости просто подключать с помощью

атрибута `src` эти файлы к другим сценариям.

Замечание

Элемент `<script>` является вторым обязательным элементом в сценариях WSH с разметкой XML.

Примеры сценариев с разметкой XML

Приведем примеры сценариев, иллюстрирующие основные свойства WS-файлов.

Строгий режим обработки WS-файла

Напомним, что здесь обязательными являются элементы <?xml?> и <![CDATA[]]>. Соответствующий пример сценария strict.wsf приведен в листинге 3.8.

Листинг 3.8. Файл strict.wsf

```
<?xml version="1.0" standalone="yes" encoding="windows-1251"?>
<job id="JS">
 <runtime>
  <description>
  Имя: strict.wsf
  Кодировка: Windows
  Описание: Пример строгого режима обработки WS-файла
  </description>
 </runtime>
 <script language="JScript">
  <![CDATA[
   WScript.Echo("Всем привет!");
  ]]>
 </script>
</job>
```

Несколько заданий в одном файле

Каждое отдельное задание в WS-файле должно находиться внутри элементов <job> и </job>. В свою очередь, все элементы <job> являются дочерними элементами контейнера <package>.

В качестве примера рассмотрим сценарий multijob.wsf, приведенный в листинге 3.9. Здесь описываются два задания с идентификаторами "VBS" (сценарий на языке VBScript) и "JS" (сценарий на языке JScript).

Листинг 3.9. Файл multijob.wsf

```
<package>
 <job id="VBS">
  <!-- Описываем первое задание (id="VBS") -->
  <runtime>
   <description>
   Имя: multijob.wsf
   Кодировка: Windows
   Описание: Первое задание из example.wsf
   </description>
  </runtime>
  <script language="VBScript">
   WScript.Echo "Первое задание (VBScript)"
  </script>
 </job>
 <job id="JS">
  <!-- Описываем второе задание (id="JS") -->
  <runtime>
   <description>
   Имя: example.wsf
   Кодировка: Windows
   Описание: Второе задание из example.wsf
   </description>
  </runtime>
  <script language="JScript">
   WScript.Echo("Второе задание (JScript)");
  </script>
 </job>
</package>
```

Для того чтобы выполнить первое задание сценария multijob.wsf, которое выведет на экран строку "Первое задание (VBScript)", нужно выполнить одну из следующих команд:

```
cscript //job:"VBS" multijob.wsf
cscript multijob.wsf
wscript //job:"VBS" multijob.wsf
wscript multijob.wsf
```

Для запуска второго задания, выводящего на экран строку "Второе задание (JScript)", нужно явно указывать идентификатор этого задания, поэтому используется одна из двух команд:

```
cscript //job:"JS" multijob.wsf
wscript //job:"JS" multijob.wsf
```

Использование констант внешних объектов

Для того чтобы в сценарии обращаться по имени к константам, определенным во внешних объектах, не создавая экземпляров самих объектов, необходимо сначала получить ссылку на эти объекты с помощью элемента `<reference>`.

В листинге 3.10 приведен сценарий refer.wsf, в котором с помощью элемента `<reference>` производится доступ к трем константам объекта `FileSystemObject` (`ForReading`, `ForWriting` и `ForAppending`), которые определяют режим работы из сценария с внешним текстовым файлом.

Листинг 3.10. Использование в сценарии констант внешних объектов (файл refer.wsf)

```
<job id="Example">
 <runtime>
  <description>
  Имя: refer.wsf
  Кодировка: Windows
  Описание: Использование констант внешних объектов
  </description>
 </runtime>
 <!-- Получаем ссылку на объект FileSystemObject -->
 <reference object="Scripting.FileSystemObject"/>
 <script language="JScript">
  var s;
  s="Значения констант объекта FileSystemObject:\n\n";
  //Получаем значение константы ForReading
  s+="ForReading="+ForReading+"\n";
  //Получаем значение константы ForWriting
  s+="ForWriting="+ForWriting+"\n";
  //Получаем значение константы ForAppending
  s+="ForAppending="+ForAppending;
  //Выводим полученные строки на экран
  WScript.Echo(s);
 </script>
</job>
```

В результате выполнения сценария refer.wsf на экран выведется диалоговое окно с информацией о значениях констант объекта `FileSystemObject` (рис. 3.8).

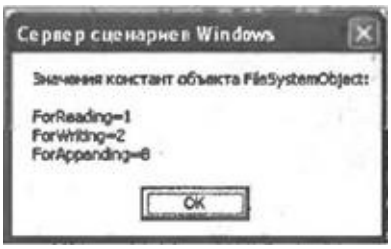


Рис. 3.8. Результат работы сценария refer.wsf

Подключение внешних файлов

К WS-файлу можно подключать "обычные" JScript- или VBScript-сценарии, которые находятся во внешних файлах. Для этого нужно указать путь к этому внешнему файлу в атрибуте `src` элемента `<script>`.

Для примера создадим файл inc.js, в который запишем строку

```
WScript.Echo("Здесь выполняется сценарий inc.js");
```

и файл main.wsf, содержание которого приведено в листинге 3.11.

Листинг 3.11. Подключение внешнего сценария (файл main wsf)

```
<job id="Example">
 <runtime>
  <description>
  Имя: main.wsf
  Кодировка: Windows
  Описание: Подключение сценария, находящегося во внешнем файле
  </description>
 </runtime>
 <!-- Подключаем сценарий из файла inc.js -->
 <script language="JScript" src="inc.js"/>
 <!-- Определяем основной сценарий -->
 <script language="JScript">
  WScript.Echo("Здесь выполняется основной сценарий");
 </script>
</job>
```

Если запустить main.wsf с помощью cscript.exe, то на экран выведутся две строки:

```
Здесь выполняется сценарий inc.js
Здесь выполняется основной сценарий
```

Два языка внутри одного задания (использование функции InputBox языка VBScript в сценариях JScript)

Как уже отмечалось в *главе 2*, ни в WSH, ни в JScript нет метода или функции, которые позволяли бы в графическом режиме создать диалоговое окно для ввода текста. Однако в языке VBScript имеется функция `InputBox`, предназначенная как раз для этой цели; используя разметку XML, мы можем легко использовать эту функцию в сценариях JScript. Соответствующий пример приведен в сценарии multilang.wsf (листинг 3.12).

Сначала в этом сценарии на языке VBScript описывается функция `InputName`, которая возвращает строку, введенную с помощью функции `InputBox`:

```
<script language="VBScript">
Function InputName
  InputName = InputBox("Введите Ваше имя:", "Окно ввода VBScript")
End Function
</script>
```

Затем в следующем разделе `<script>` приводится JScript-сценарий, в котором происходит вызов функции `InputName` и сохранение возвращаемого ею значения в переменной s:

```
var s;
s = InputName();
```

Значение полученной таким образом переменной s выводится затем на экран:

```
WScript.Echo("Здравствуйте, "+s+"!");
```

Таким образом, после запуска сценария multilang.wsf на экран выводится диалоговое окно для ввода имени пользователя, показанное на рис. 3.9.

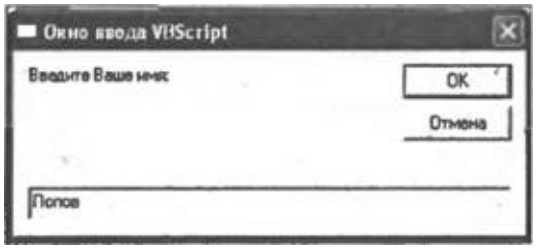


Рис. 3.9. Окно ввода (функция `InputBox` языка VBScript)

После ввода информации на экран выводится окно, показанное на рис. 3.10.

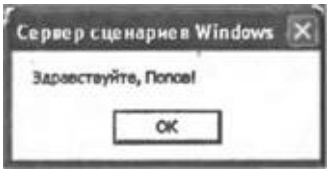


Рис. 3.10. Стандартное окно вывода WSH

Листинг 3.12. Использование различных языков внутри одного задания (файл multilang.wsf)

```
    <job id="Example">
     <runtime>
      <description>
      Имя: multilang.wsf
      Кодировка: Windows
      Описание: Использование функции InputBox в JScript-сценарии
      </description>
     </runtime>
     <script language="VBScript">
      Function InputName  ' Описываем функцию на языке VBScript
       ' Вводим имя в диалоговом окне
          InputName  =  InputBox("Введите  Ваше  имя:",  "Окно  ввода
VBScript")
      End Function
     </script>
     <script language="JScript">
      var s;
      s = InputName();  //Вызываем функцию InputName
      //Выводим значение переменной s на экран
      WScript.Echo("Здравствуйте, "+s+"!");
     </script>
    </job>
```

Глава 4
Безопасность при работе со сценариями WSH

Одним из главных преимуществ WSH является возможность запуска программ-сценариев, которые хранятся в виде исходного текста, что максимально упрощает процессы написания и распространения программ — не нужны ни дополнительные компиляторы для создания исполняемого кода, ни специальные утилиты для установки и регистрации сценариев в операционной системе. Однако при использовании таких сценариев в силу той же простоты сразу возникает несколько проблем.

Во-первых, исходный код сценария является незащищенным — любой пользователь, запускающий сценарий, может модифицировать его и использовать в дальнейшем как свой собственный (нарушаются авторские права автора).

Во-вторых, простота распространения и выполнения сценариев открывает широкие возможности для написания вредоносных сценариев-вирусов, которые могут, например, рассылаться по электронной почте, как широко известный вирус "I Love You".

Поэтому при использовании сценариев WSH вопросы безопасности имеют весьма большое значение. В этой главе описано, каким образом можно создавать зашифрованные файлы сценариев и добавлять к сценариям цифровые подписи для указания автора сценария. Кроме этого, рассмотрен процесс организации политик безопасности для сценариев WSH, позволяющих, например, запретить выполнение неподписанных сценариев любого типа или вообще заблокировать WSH для определенных пользователей.

Шифрование сценариев

Начиная с версии 2.0, в WSH появилась возможность скрыть от пользователя исходный текст сценария, преобразовав (зашифровав) его с помощью программы Microsoft Script Encoder, которую можно свободно скачать по адресу **http://msdn.microsoft.com/scripting/vbscript/download/x86/sce10en.exe.**

Программа Script Encoder может применяться для шифрования сценариев JScript (файлы *.js), VBScript (файлы *.vbs) и WS-файлов (расширение wsf), а также сценариев, содержащихся в гипертекстовых файлах HTML.

Замечание

Шифрование с помощью Script Encoder не следует рассматривать как надежное средство сохранения в тайне исходного кода сценария — программа просто преобразует текст сценария в кодировку, непригодную для чтения, и профессионал сможет из него восстановить первоначальное содержимое. Однако для защиты сценария от изменений обычными пользователями подобного шифрования вполне достаточно.

Для запуска программы Script Encoder служит файл screnc.exe; по умолчанию установка исполняемого файла и файла помощи производится в каталог Program Files\Windows Script Encoder. Программа srcenc.exe запускается из командной строки, в качестве ее обязательных параметров указываются имена исходного файла сценария и файла, в котором будет содержаться этот сценарий в зашифрованном виде.

Замечание

В системе зарегистрированы специальные расширения для файлов с зашифрованными сценариями WSH: jse для сценариев JScript и vbe для сценариев VBScript

Рассмотрим пример. Пусть в файле ForEncode.js находится простой JScript-сценарий (листинг 4.1).

Листинг 4.1. Исходный текст сценария ForEncode.js

```
    /*****************************************************************
    /*                                                           Имя:
ForEncode.js                                                  */
    /*                                                          Язык:
JScript                                                     */
    /*              Описание:              Исходный              текст
сценария                                 */
    /*****************************************************************
    WScript.Echo("Привет!");
    /*************                                              Конец
*******************************************/
```

Тогда после выполнения команды

```
    screnc ForEncode.js Encoded.jse
```

создастся файл Encoded.jse, содержащий зашифрованный текст сценария ForEncode.js (листинг 4.2).

Листинг 4.2. Зашифрованный сценарий Encoded.jse

```
    #@~^0QEAAA==&CeMCeCeeCeCMeCeMeCeMMCeeCMeCeeCMMeCeCeMeMMCeMeCMeCe
N/P,~P,PP,~~P,P,P~P~~,P~P,~P,P~~,PP~~,P~P,~,P~,P,ez@#@&JMPЯзык=PB?
1Dr2DPP,~P,PP,~~P,P,P~P~~,P~P,~P,P~~,PP~~,P~P,~,P~,P,PP,eJ@#@&Je~Опи
```

Сценарии VBScript, записанные в файлах с расширением vbs, шифруются точно так же:

```
    screnc ForEncode.vbs Encoded.vbe
```

Исходный сценарий ForEncode.vbs приведен в листинге 4.3, зашифрованный сценарий Encoded.vbe — в листинге 4.4.

Листинг 4.3. Исходный текст сценария ForEncode.vbs

```
    '*****************************************************************
    '                                                           Имя:
ForEncode.vbs
    '                                                          Язык:
VBScript
```

```
'                Описание:                Исходный                текст
сценария
'*****************************************************************
WScript.Echo "Привет!"
'*************                                                Конец
*********************************************
```

Листинг 4.4. Зашифрованный сценарий Encoded.vbe

```
#@~^xQEAAA==vCeMCeCeeCeCMeCeMeCeMMCeeCMeCeeCMMeCeCeMeMMCeMeCMeCe
8kPP,~P,PP,~~P,P,P~P~~,P~P,~P,P~~,PP~~,P~P,~,P~,@#@&BPЯзык=Pj$?
1DrwDP~~,PP,~P,PP,~~P,P,P~P~~,P~P,~P,P~~,PP~~,P~P,~,P~,P@#@&B,Описан
```

Как видно из листингов 4.3 и 4.4, символы кириллицы остаются в зашифрованных сценариях без изменения.

Зашифрованные файлы Encoded.jse и Encoded.vbe можно запускать с помощью cscript.exe или wscript.exe, выполняться они будут точно так же, как и исходные сценарии (рис. 4.1).

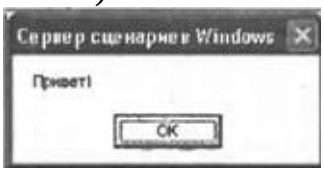


Рис. 4.1. Результат выполнения зашифрованного сценария Encoded.jse

Еще одной весьма полезной особенностью сценариев, зашифрованных с помощью Script Encoder, является то, что при запуске такого сценария автоматически производится контроль целостности файла. Например, если в файле Encoded.jse убрать или добавить букву в слово "Привет", то при запуске будет выведено сообщение об ошибке (рис. 4.2) и сценарий выполняться не будет.

Рис. 4.2. Сообщение об ошибке, выводимое при запуске модифицированного файла Encoded.jse

Содержимое зашифрованных сценариев с расширениями jse и vbe можно вставлять в WS-файлы внутрь элементов `<script>`, при этом в

качестве значения аргумента `language` должно быть указано "`JScript.Encode`" (зашифрованный сценарий на языке JScript) или "`VBScript.Encode`" (зашифрованный сценарий на языке VBScript). Пример такого WS-файла Encoded.wsf приведен в листинге 4.5, исходный файл ForEncode.wsf — в листинге 4.6.

Листинг 4.5. Зашифрованный сценарий Encoded.wsf

```
<job id="Encoded">
<runtime>
<description>
Имя: Encoded.wsf
Описание: WS-файл с зашифрованными сценариями
</description>
</runtime>
<script language="VBScript.Encode">
#@~^FgAAAA== Um.bwDR21tK~JПриветеJ/gQAAA==^#~@
</script>
<script language="JScript.Encode">
#@~^FgAAAA== Um.bwDR21tKcJПокаеJbiagUAAA==^#~@
</script>
</job>
```

Листинг 4.6. Исходный сценарий ForEncode.wsf

```
<job id="Encoded">
<runtime>
<description>
Имя: Encoded.wsf
Описание: WS-файл с зашифрованными сценариями
</description>
</runtime>
<script language="VBScript.Encode">
WScript.Echo "Привет!"
</script>
<script language="JScript.Encode">
WScript.Echo("Пока!");
</script>
</job>
```

Однако если попытаться зашифровать исходный файл ForEncode.wsf с помощью команды

```
screnc ForEncode.wsf Encoded.wsf
```

то возникнет ошибка, т.к. Script Encoder "не понимает" расширения wsf. Поэтому для шифрования WS-файлов нужно при вызове screnc.exe в командной строке использовать дополнительный ключ:

```
screnc /e htm ForEncode.wsf Encoded.wsf
```

Параметр `/e htm` здесь указывает на то, что исходный файл ForEncode.wsf является файлом с HTML-разметкой (расширение htm), при этом Script Encoder шифрует только содержимое элементов `<script>`, оставляя весь остальной текст файла без изменения.

Описание других ключей программы Script Encoder, которые могут применяться при шифровании сценариев WSH, приведено в табл. 4.1.

Таблица 4.1. *Параметры командной строки screnc.exe*

Параметр	**Описание**
`/s`	Подавляет все сообщения программы. Если этот ключ не указан, то по умолчанию сообщения программы выводятся на экран
`/f`	Перезаписывает исходный файл (зашифрованный файл будет иметь то же самое имя, что и исходный сценарий)
`/l` *`defLanguage`*	Явно указывает язык сценария в шифруемом файле. Например, `/l Jscript`

Цифровая подпись для сценариев WSH

Сценарии WSH можно защищать с помощью *цифровой подписи*, используя которую, можно определить происхождение сценария и гарантировать его целостность, т.е. отсутствие в этом сценарии несанкционированных изменений. Если источник, из которого поступил этот сценарий, является *надежным (trusted source)*, т.е. вы доверяете этому источнику, сценарий можно выполнить. В случае же ненадежности создателя или распространителя сценария можно (при соответствующей настройке политик безопасности Windows) отказаться от его запуска.

Таким образом, разработчик должен идентифицировать свои сценарии цифровой подписью, чтобы пользователи, получающие такие сценарии, знали, откуда они к ним попали и кто их создал. После добавления цифровой подписи к сценарию он может свободно запускаться (но не модифицироваться!) всеми, кто указал автора программы как надежного источника сценариев.

Цифровая подпись в Windows создается с помощью цифровых сертификатов, поэтому сначала кратко рассмотрим, каким образом можно получить цифровой сертификат и установить доверие к нему.

Использование цифровых сертификатов в Windows

Цифровой сертификат — это электронный документ, который однозначно идентифицирует его владельца. Сертификаты различных типов широко используются во многих службах безопасности Windows для разных целей, например:

□ проверка подлинности пользователей Интернета или Web-сервера (пользователи должны иметь возможность доказать свою подлинность тем, с кем они соединяются в компьютерной сети, и должны иметь возможность проверить подлинность других пользователей);

□ шифрование и защита от искажений данных, которые передаются по локальной сети или при помощи электронной почты (при обмене зашифрованными сообщениями электронной почты никто не сможет прочитать сообщение в процессе его доставки, а расшифровать и прочитать сообщение сможет только получатель, которому оно предназначено);

□ подписание электронных писем (получатель сообщения может проверить, что сообщение не было изменено в процессе доставки и что

сообщение пришло именно от отправителя);

□ проверка подлинности программного обеспечения, которое загружается из Интернета, устанавливается из локальной сети организации или с компакт-диска (неподписанное программное обеспечение, т.е. не имеющее действительного сертификата издателя, может представлять опасность для компьютера и сохраняемой на нем информации).

Способы получения цифрового сертификата

Различаются цифровые сертификаты трех типов: созданные разработчиком, выданные разработчику организацией и полученные от центра сертификации.

Цифровой сертификат, созданный разработчиком, обычно используют те пользователи, которые доверяют этому разработчику. Например, администратор локальной сети может создать свой собственный сертификат для подписи сценариев WSH, которые он создает и распространяет внутри своей организации.

В организации может быть организован свой сервер выдачи цифровых сертификатов (соответствующая служба имеется, например, в операционной системе Windows 2000 Server). Таким образом, организация распространяет сертификаты среди собственных разработчиков, не прибегая к услугам коммерческих центров сертификации.

В Интернете имеются сайты коммерческих центров сертификации, которые выдают сертификаты как организациям, так и частным лицам; одним из самых известных таких центров является VeriSign **(http://www.verisign.com).** Естественно, большинство услуг центров сертификации являются платными, причем цена на сертификат зависит от цели его приобретения (личный цифровой сертификат для индивидуального распространения программ стоит намного дешевле, чем сертификат для организаций, занимающихся разработкой и продажей программного обеспечения).

Понятно, что для того, чтобы попрактиковаться в применении сертификатов для подписания сценариев WSH, проще всего создать цифровой сертификат самостоятельно, поэтому в дальнейшем мы будем рассматривать только такие сертификаты.

Создание собственного сертификата

Наиболее быстрым способом создания собственного цифрового сертификата является использование программы SelfCert.exe, входящей в состав Microsoft Office 2000/XP. Запустив эту утилиту, мы получим диалоговое окно, позволяющее задать имя создаваемого сертификата (рис. 4.3). В качестве этого имени можно использовать, например, свое имя или название организации.

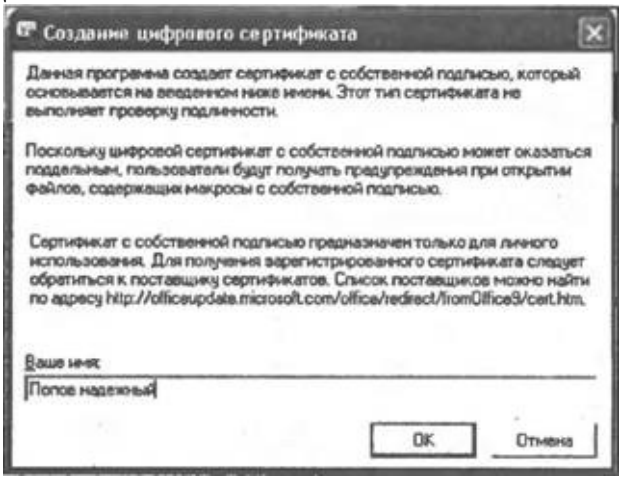


Рис. 4.3. Создание нового личного сертификата с помощью SelfCert.exe

Рис. 4.4. Успешное создание личного сертификата "Попов надежный"

Для дальнейших экспериментов нам понадобится создать два сертификата с именами "Попов надежный" и "Попов ненадежный". После нажатия кнопки **OK** в случае успешного создания сертификата на экран выводится диалоговое окно с информацией об этом (рис. 4.4).

Управление сертификатами с помощью MMC

Для того чтобы посмотреть свойства созданных сертификатов, нам потребуется запустить *консоль управления Microsoft Management Console (MMC)* — инструмент для создания, сохранения и открытия средств администрирования (называемых *консолями* (Snap-in) MMC), которые управляют оборудованием, программными и сетевыми компонентами операционной системы Windows. Для того чтобы загрузить MMC, нужно

выполнить команду `mmc` либо в командной строке, либо с помощью пункта **Выполнить** (Run) меню **Пуск** (Start). В результате на экране появится окно новой консоли (рис. 4.5).

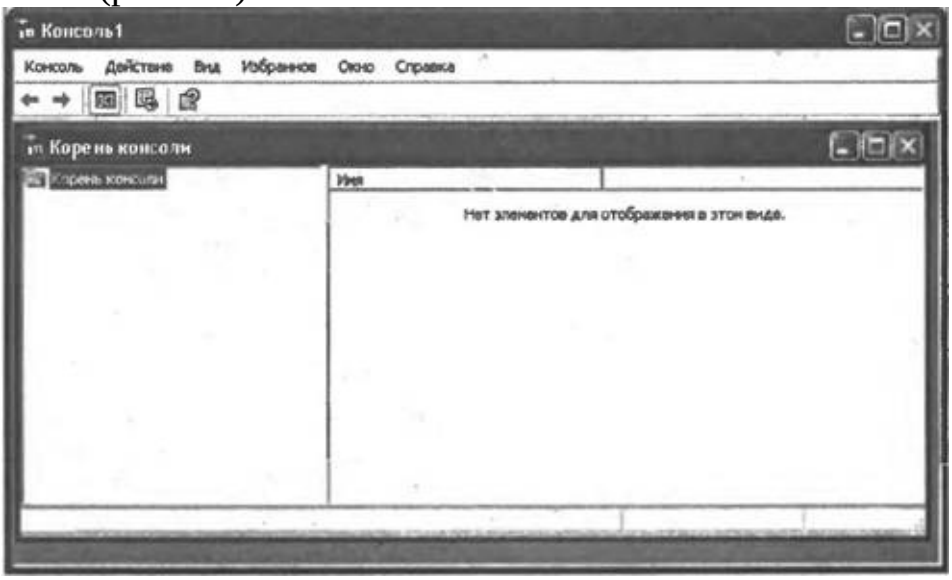


Рис. 4.5. Новая консоль MMC

Теперь добавим в консоль оснастку **Сертификаты** (Certificates). Для этого нужно в меню **Консоль** (Console) выбрать пункт **Добавить/удалить оснастку** (Add/Remove Snap-in), после чего появится диалоговое окно, показанное на рис. 4.6.

Рис. 4.6. Диалоговое окно добавления/удаления оснасток для консоли MMC

После нажатия кнопки **Добавить** (Add) выводится список всех имеющихся оснасток (рис. 4.7).

Из этого списка нужно выбрать **Сертификаты** (Certificates) и нажать кнопку **Добавить** (Add). После этого в новом диалоговом окне отмечаем, что добавляемая оснастка будет управлять сертификатами для своей учетной записи (рис. 4.8) и нажимаем кнопку **Готово** (Finish).

Никаких других оснасток в окно консоли мы добавлять не будем, поэтому нажимаем кнопку **Закрыть** (Close) в списке оснасток и кнопку **OK** в окне добавления/удаления оснасток. Созданные нами сертификаты будут находиться в разделе **Личные\Сертификаты** (Personal\Sertificates) (рис. 4.9).

Выделив нужный сертификат в списке и нажав клавишу <Enter>, мы выведем окно с описанием свойств сертификата (рис. 4.10).

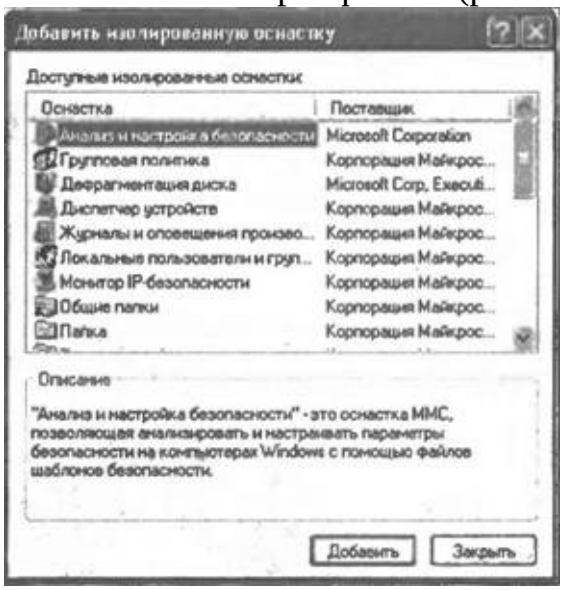


Рис. 4.7. Список всех оснасток

Рис. 4.8. Выбор типа сертификатов, которым будет управлять добавляемая оснастка **Сертификаты**

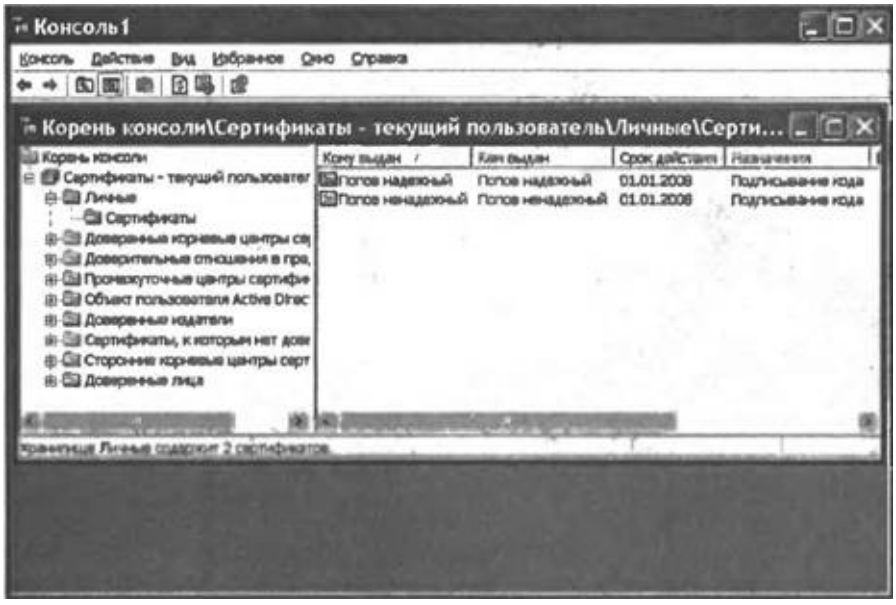


Рис. 4.9. Расположение личных сертификатов

Рис. 4.10. Свойства сертификата "Попов надежный"

Как мы видим, для обоих новых сертификатов не установлено доверие. Для того чтобы установить доверие к одному из сертификатов ("Попов надежный"), достаточно просто перетащить при помощи мыши этот сертификат **в** раздел **Доверенные корневые центры сертификации | Сертификаты** (Trusted Root Certification Authorities) (рис. 4.11).

Замечание

Если при перетаскивании сертификата в новое место держать нажатой клавишу <Ctrl>, то сертификат будет не перемещен, а скопирован.

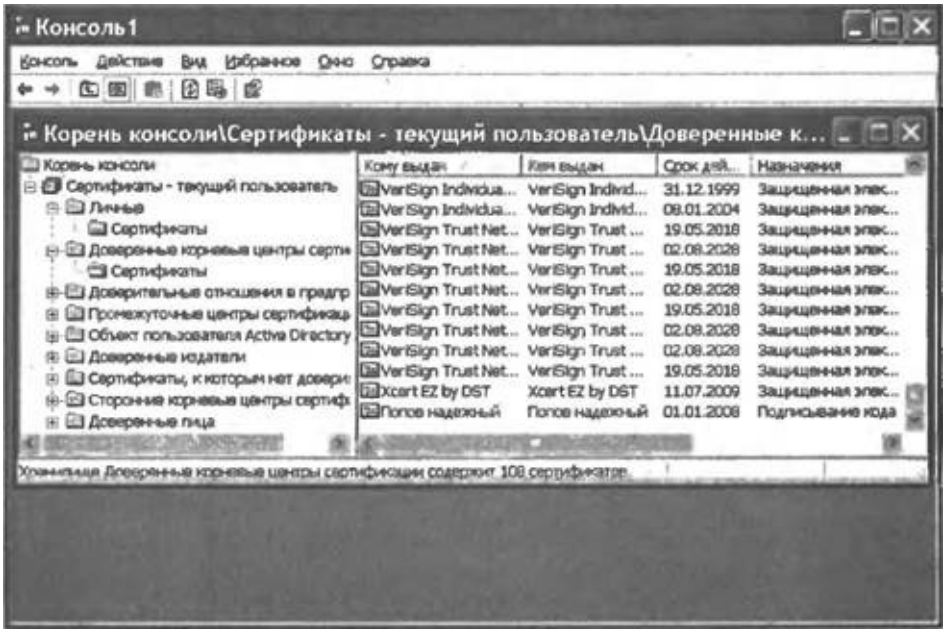


Рис. 4.11. Сертификаты, к которым установлено доверие

Свойства сертификата "Попов надежный" после установки доверия к нему показаны на рис. 4.12.

Выполним теперь экспорт в файл сертификата "Попов ненадежный" (это понадобится нам в дальнейшем при построении политики ограниченного использования программ). Для этого следует выделить нужный сертификат и выбрать в меню **Действие | Все задачи** (Action | All Tasks) пункт **Экспорт** (Export), после чего запустится мастер экспорта сертификатов, в котором нужно согласиться со всеми настройками, предлагаемыми по умолчанию, а в качестве имени экспортируемого файла указать "C:\Script\Попов.cer".

Рис. 4.12. Новые свойства сертификата "Попов надежный"

Добавление к сценарию цифровой подписи

Подписать файл со сценарием WSH можно двумя способами. Во-первых, можно использовать программу SignCode.exe, с помощью которой производится подпись любого исполняемого кода. Однако эта программа не является стандартной частью операционной системы Windows, ее нужно устанавливать отдельно. Поэтому далее мы будем рассматривать второй способ подписи сценариев — с помощью метода `SignFile` объекта `Scripting.Signer`, который регистрируется в системе при установке WSH 5.6.

В листинге 4.7 приведен сценарий SignScript.wsf, с помощью которого мы будем подписывать файлы сценариев различных типов, используя личные сертификаты "Попов надежный" и "Попов ненадежный". При запуске этого сценария нужно указать два именных параметра командной строки: `/file` для задания пути к подписываемому файлу и `/cert`, содержащий название цифрового сертификата, на основе которого будет создаваться подпись. Эти параметры в сценарии подставляются в качестве аргументов метода `SignFile` объекта `Scripting.Signer`:

```
//Создаем объект Scripting.Signer
Signer = WScript.CreateObject("Scripting.Signer");
File = WScript.Arguments.Named("file"); //Имя файла
```

```
Cert = WScript.Arguments.Named("cert"); //Название сертификата
Signer.SignFile(File, Cert); //Подписываем файл
```

Замечание

В методе `SignFile` может быть указан третий необязательный параметр, задающий путь к хранилищу сертификатов. В сценарии SignScript.wsf этот параметр не используется, т.к. для создания подписи применяются личные сертификаты, находящиеся на локальной машине.

Листинг 4.7. Сценарий SignScript.wsf

```
<job>
 <runtime>
    <named name="file" helpstring="Путь к файлу сценария"
required="true" type="string"/>
   <named name="cert" helpstring="Имя цифрового сертификата"
required="true" type="string"/>
  <description>
  Имя: SignScript.wsf
  Описание: Добавление цифровой подписи к файлам
            со сценариями WSH
  </description>
  <example>
  Пример:
  SignScript.wsf  /file:Signed.wsf /cert:"Попов надежный"
  </example>
 </runtime>
 <script language="JScript">
  var Signer, File, Cert, Store;
        if    (!(WScript.Arguments.Named.Exists("cert")    &&
WScript.Arguments.Named.Exists("file"))) {
   WScript.Arguments.ShowUsage();
   WScript.Quit();
  }
  Signer = WScript.CreateObject("Scripting.Signer");
  File = WScript.Arguments.Named("file");
  Cert = WScript.Arguments.Named("cert");
  Store = "";
  Signer.SignFile(File, Cert);
 </script>
```

```
</job>
```

Цифровые подписи добавляются в конец файлов, содержащих сценарии, причем в обычных JScript- и VBScript-файлах подпись находится в блоке комментария, а в WS-файлах — внутри элемента `<signature>`. Это делает возможным запуск подписанных сценариев в предыдущих версиях WSH, т.к. здесь закомментированные или находящиеся внутри `<signature>` части сценариев будут при выполнении проигнорированы.

В листингах 4.8–4.10 приведены примеры сценариев различных типов, которые были подписаны с использованием сертификата "Попов надежный".

Листинг 4.8. Подписанный JScript-файл

```
    /**************************************************************
    /*                                                      Имя:
Signed.js                                                    */
    /*                                                      Язык:
JScript                                                     */
    /*         Описание:         Сценарий         с         цифровой
подписью                         */
    /**************************************************************
    WScript.Echo("Привет!");
    // SIG // Begin signature block
    // SIG // MIIEMAYJKoZIhvcNAQcCoIIEITCCBB0CAQExDjAMBggq
    // SIG // hkiG9w0CBQUAMGYGCisGAQQBgjcCAQSgWDBWMDIGCisG
    // SIG // AQQBgjcCAR4wJAIBAQQQEODJBs441BGiowAQS9NQkAIB
    // SIG // AAIBAAIBAAIBAAIBADAgMAwGCCqGSIb3DQIFBQAEENmN
    // SIG // vsHhmXojm79+NmBDE0qgggJIMIICRDCCAa2gAwIBAgIQ
    // SIG // sGqcGTDiWZBINKde3DiDqDANBgkqhkiG9w0BAQQFADAn
    // SIG // MSUwIwYDVQQDHhwEHwQ+BD8EPgQyACAEPQQwBDQENQQ2
    // SIG // BD0ESwQ5MB4XDTAxMTIzMTIwMDAwMFoXDTA3MTIzMTIw
    // SIG // MDAwMFowJzElMCMGA1UEAx4cBB8EPgQ/BD4EMgAgBD0E
    // SIG // MAQ0BDUENgQ9BEsEOTCBnzANBgkqhkiG9w0BAQEFAAOB
    // SIG // jQAwgYkCgYEAsNlPby/9ax7Ky75RO8xI+jrNU/u842T2
    // SIG // 6Md730yxbVD9+54SVNtsCoYOR7OYgQMb9TtiirFpk0bJ
    // SIG // PH7WuCjGlrMOhDt86Vq++er67d87p0rytm0R7m1/FOkw
    // SIG // GxxjycUEKS2w65FXfdpngRKiu3NIDb6tsupKiHex3XEo
    // SIG // 1n0O++kCAwEAAaNxMG8wEwYDVR0lBAwwCgYIKwYBBQUH
    // SIG // AwMwWAYDVR0BBFEwT4AQ0LHfQjJYgcR4nBGbd2fwEKEp
    // SIG // MCcxJTAjBgNVBAMeHAQfBD4EPwQ+BDIAIAQ9BDAENAQ1
    // SIG // BDYEPQRLBDmCELBqnBkw4lmQSDSnXtw4g6gwDQYJKoZI
    // SIG // hvcNAQEEBQADgYEAi0hytKDNpUwc8/MpidjPFaE3xu9g
    // SIG // SSVXbietJ5kW7uJR1IXAA1CaEHDCe0JIxCdHhfKTcWrK
```

```
    // SIG // Q4t0mG6fBFO180xKHcyOCpVuFywnkMI8AWKzqNYdaDUZ
    // SIG // sNtaw4R4XjX9koovYmfFo4i75A9zd4qXB93AaWUJMEt7
    // SIG // g+JQ2Blzn4gxggFSMIIBTgIBATA7MCcxJTAjBgNVBAMe
    // SIG // HAQfBD4EPwQ+BDIAIAQ9BDAENAQ1BDYEPQRLBDkCELBq
    // SIG // nBkw4lmQSDSnXtw4g6gwDAYIKoZIhvcNAgUFAKBsMBAG
    // SIG // CisGAQQBgjcCAQwxAjAAMBkGCSqGSIb3DQEJAzEMBgor
    // SIG // BgEEAYI3AgEEMBwGCisGAQQBgjcCAQsxDjAMBgorBgEE
    // SIG // AYI3AgEVMB8GCSqGSIb3DQEJBDESBBB3QmVqnKzYLdaa
    // SIG // EXTFBkJhMA0GCSqGSIb3DQEBAQUABIGAAMkO/8PM8+Ay
    // SIG // zG8FeUtsDIUy56sSmYd9W34axmFbGktiseSshNxemfbv
    // SIG // 7TucEbZ40zyMFdpaKDqwwCNf1M3k59eu4jk0wX1v1VBd
    // SIG // +mwQ3JNg3WpvAL6OFrAko+ksiZ2ndjKJfJXwDUSITFmy
    // SIG // aGrGNkC3RYRhp6dPdL8Tb0PVvcU=
    // SIG // End signature block
```

Листинг 4.9. Подписанный VBScript-файл

```
    '*************************************************************
    ' Имя: Signed.vbs
    ' Язык: VBScript
    ' Описание: Сценарий с цифровой подписью
    '*************************************************************
    WScript.Echo "Привет!"
    '*************                                          Конец
*****************************************
    '' SIG '' Begin signature block
    '' SIG '' MIIEMAYJKoZIhvcNAQcCoIIEITCCBB0CAQExDjAMBggq
    '' SIG '' hkiG9w0CBQUAMGYGCisGAQQBgjcCAQSgWDBWMDIGCisG
    '' SIG '' AQQBgjcCAR4wJAIBAQQQTvApFpkntU2P5azhDxfrqwIB
    '' SIG '' AAIBAAIBAAIBAAIBADAgMAwGCCqGSIb3DQIFBQAEELv8
    '' SIG '' ImJefHuyhVFY7TYWddigggJIMIICRDCCAa2gAwIBAgIQ
    '' SIG '' 75BSQe/4a5lLJ/s5SPVp+zANBgkqhkiG9w0BAQQFADAn
    '' SIG '' MSUwIwYDVQQDHhwEHwQ+BD8EPgQyACAEPQQwBDQENQQ2
    '' SIG '' BD0ESwQ5MB4XDTAxMTIzMTIwMDAwMFoXDTA3MTIzMTIw
    '' SIG '' MDAwMFowJzElMCMGA1UEAx4cBB8EPgQ/BD4EMgAgBD0E
    '' SIG '' MAQ0BDUENgQ9BEsEOTCBnzANBgkqhkiG9w0BAQEFAAOB
    '' SIG '' jQAwgYkCgYEAsNlPby/9ax7Ky75RO8xI+jrNU/u842T2
    '' SIG '' 6Md730yxbVD9+54SVNtsCoYOR7OYgQMb9TtiirFpk0bJ
    '' SIG '' PH7WuCjGlrMOhDt86Vq++er67d87p0rytm0R7m1/FOkw
    '' SIG '' GxxjycUEKS2w65FXfdpngRKiu3NIDb6tsupKiHex3XEo
    '' SIG '' 1n0O++kCAwEAAaNxMG8wEwYDVR0lBAwwCgYIKwYBBQUH
    '' SIG '' AwMwWAYDVR0BBFEwT4AQ0LHfQjJYgcR4nBGbd2fwEKEp
    '' SIG '' MCcxJTAjBgNVBAMeHAQfBD4EPwQ+BDIAIAQ9BDAENAQ1
    '' SIG '' BDYEPQRLBDmCEO+QUkHv+GuZSyf7OUj1afswDQYJKoZI
    '' SIG '' hvcNAQEEBQADgYEAe3+Qek5z1V/kzjxcYB4nrGs0jN+5
```

```
'' SIG '' uRTY5+PtPyjz85i2y3YbH08HitGUYhZA3ImslbzMOCGg
'' SIG '' vIJAfzvhlIve+kKG5pG9EDFUpJ/eyHEizAmxjSChgZlz
'' SIG '' 2V++7VF6hIevX4VcUKCcCoGXb88Tp/XwY1arFFYzSUZg
'' SIG '' MJl3tYcHVrMxggFSMIIBTgIBATA7MCcxJTAjBgNVBAMe
'' SIG '' HAQfBD4EPwQ+BDIAIAQ9BDAENAQ1BDYEPQRLBDkCEO+Q
'' SIG '' UkHv+GuZSyf7OUj1afswDAYIKoZIhvcNAgUFAKBsMBAG
'' SIG '' CisGAQQBgjcCAQwxAjAAMBkGCSqGSIb3DQEJAzEMBgor
'' SIG '' BgEEAYI3AgEEMBwGCisGAQQBgjcCAQsxDjAMBgorBgEE
'' SIG '' AYI3AgEVMB8GCSqGSIb3DQEJBDESBBDWCooVTqYEryt6
'' SIG '' BNyCqxsxMA0GCSqGSIb3DQEBAQUABIGASPcIufHgECN+
'' SIG '' r9VNOAo4NJnI610cZS73Unu29n27/LISI5zrHqC/AR2n
'' SIG '' ts04Ah3j79usRJEaNJySDdTB3T5CyEue0XBS1FOTj4nt
'' SIG '' kYZBWXVxM0+Z1oGZNCTFJFtqPnEGhI2WwpKMz00luw09
'' SIG '' qnpjkM9dxsgPlumnkf5dpleZtBU=
'' SIG '' End signature block
```

Листинг 4.10. Подписанный WS-файл

```
<job>
<runtime>
<description>
Имя: Signed.wsf
Описание: Сценарий с цифровой подписью
</description>
</runtime>
<script language="JScript">
WScript.Echo("Привет!");
</script>
<signature>
** SIG ** MIIEMAYJKoZIhvcNAQcCoIIEITCCBB0CAQExDjAMBggq
** SIG ** hkiG9w0CBQUAMGYGCisGAQQBgjcCAQSgWDBWMDIGCisG
** SIG ** AQQBgjcCAR4wJAIBAQQQcAVhGs441BGiowAQS9NQkAIB
** SIG ** AAIBAAIBAAIBAAIBADAgMAwGCCqGSIb3DQIFBQAEEANf
** SIG ** TmfjlqP6LHKR3b45MWagggJIMIICRDCCAa2gAwIBAgIQ
** SIG ** 75BSQe/4a5lLJ/s5SPVp+zANBgkqhkiG9w0BAQQFADAn
** SIG ** MSUwIwYDVQQDHhwEHwQ+BD8EPgQyACAEPQQwBDQENQQ2
** SIG ** BD0ESwQ5MB4XDTAxMTIzMTIwMDAwMFoXDTA3MTIzMTIw
** SIG ** MDAwMFowJzElMCMGA1UEAx4cBB8EPgQ/BD4EMgAgBD0E
** SIG ** MAQ0BDUENgQ9BEsEOTCBnzANBgkqhkiG9w0BAQEFAAOB
** SIG ** jQAwgYkCgYEAsNlPby/9ax7Ky75RO8xI+jrNU/u842T2
** SIG ** 6Md730yxbVD9+54SVNtsCoYOR7OYgQMb9TtiirFpk0bJ
** SIG ** PH7WuCjGlrMOhDt86Vq++er67d87p0rytm0R7m1/FOkw
** SIG ** GxxjycUEKS2w65FXfdpngRKiu3NIDb6tsupKiHex3XEo
** SIG ** 1n0O++kCAwEAAaNxMG8wEwYDVR0lBAwwCgYIKwYBBQUH
** SIG ** AwMwWAYDVR0BBFEwT4AQ0LHfQjJYgcR4nBGbd2fwEKEp
```

```
** SIG ** MCcxJTAjBgNVBAMeHAQfBD4EPwQ+BDIAIAQ9BDAENAQ1
** SIG ** BDYEPQRLBDmCEO+QUkHv+GuZSyf7OUj1afswDQYJKoZI
** SIG ** hvcNAQEEBQADgYEAe3+Qek5z1V/kzjxcYB4nrGs0jN+5
** SIG ** uRTY5+PtPyjz85i2y3YbH08HitGUYhZA3ImslbzMOCGg
** SIG ** vIJAfzvhlIve+kKG5pG9EDFUpJ/eyHEizAmxjSChgZlz
** SIG ** 2V++7VF6hIevX4VcUKCcCoGXb88Tp/XwY1arFFYzSUZg
** SIG ** MJl3tYcHVrMxggFSMIIBTgIBATA7MCcxJTAjBgNVBAMe
** SIG ** HAQfBD4EPwQ+BDIAIAQ9BDAENAQ1BDYEPQRLBDkCEO+Q
** SIG ** UkHv+GuZSyf7OUj1afswDAYIKoZIhvcNAgUFAKBsMBAG
** SIG ** CisGAQQBgjcCAQwxAjAAMBkGCSqGSIb3DQEJAzEMBgor
** SIG ** BgEEAYI3AgEEMBwGCisGAQQBgjcCAQsxDjAMBgorBgEE
** SIG ** AYI3AgEVMB8GCSqGSIb3DQEJBDESBBC8sjXscmpwRH4l
** SIG ** +m0CC76kMA0GCSqGSIb3DQEBAQUABIGAipV3It04p5Mz
** SIG ** h1Mg/dssx9PjnOTY2AWZjCYnlt8XSaGQTna1P780Krul
** SIG ** uIa3ZNbeqeSELrkZEaQdzhcw6lfUOJxVWOXKdLsss8sM
** SIG ** V9HjQ00ggKeDDtsSC+twoz7TMWqLsvEgHz7ARzL9V6lQ
** SIG ** juR0pw/g4E4cfiBQ7fvtnLh+s9o=
</signature>
</job>
```

Проверка цифровой подписи сценария

Применяя соответствующие политики безопасности, можно установить режим, при котором все запускаемые сценарии автоматически будут проверяться на предмет корректности их цифровой подписи (процесс настройки подобных политик безопасности описан ниже).

Кроме этого, можно самостоятельно из сценария WSH проверить достоверность цифровой подписи, которой снабжен тот или иной файл, и выяснить, входит ли сертификат создателя подписи в число сертификатов, к которым установлено доверие. Для такой проверки служит метод VerifyFile объекта Scripting.Signer. Данный метод имеет два параметра (File и ShowUI), первый из которых задает имя проверяемого файла, а второй является логическим флагом, позволяющим выводить или не выводить на экран диалоговое окно с информацией о состоянии сертификата, при помощи которого была создана цифровая подпись для этого файла. Если цифровая подпись проверяемого сценария является корректной, содержимое файла после создания подписи не изменялось, а к сертификату создателя сценария установлено доверие, то метод VerifyFile возвращает значение true, в противном случае — false.

В качестве примера в листинге 4.11 приведен сценарий Check.js, который проверяет цифровую подпись файла Signed.js.

Листинг 4.11. Проверка подлинности цифровой подписи сценария

```
    /*****************************************************************
    /*                                                              Имя:
Check.js                                                        */
    /*                                                             Язык:
JScript                                                       */
    /*     Описание:     Проверка     цифровой     подписи     файла
Signed.js              */
    /*****************************************************************
    var Signer, File, ShowUI, FileOK; //Объявляем переменные
    //Создаем объект Scripting.Signer
    Signer = WScript.CreateObject("Scripting.Signer");
    File = "Signed.js";  //Имя проверяемого файла
    ShowUI = false;
    //Проверяем подпись в файле
    FileOK = Signer.VerifyFile(File, ShowUI);
    if    (FileOK)    WScript.Echo("Сценарий    "+File+"    является
надежным.");
    else WScript.Echo("Сценарий "+File+" НЕ является надежным.");
    /*************                                                 Конец
*********************************************/
```

Политики безопасности для сценариев WSH

Процесс организации политики безопасности для сценариев WSH заключается в задании тех или иных ограничений на запуск и выполнение этих сценариев. При этом могут применяться два подхода.

Первый подход может использоваться в операционной системе Windows любой версии, начиная с Windows 95. Смысл здесь состоит в применении специальных параметров системного реестра, которые позволяют:

□ запретить выполнение на компьютере сценариев, запускаемых локально или с другой машины;

□ задать режим работы WSH, при котором перед запуском всех сценариев проверяется их цифровая подпись и в соответствии с этим принимается решение о возможности выполнения этих сценариев;

□ вести в журнале событий аудит успехов и отказов для сценариев WSH.

Второй подход может использоваться только в Windows XP. Здесь для сценариев WSH, как и для всех других исполняемых программ, может применяться специальная *политика ограниченного использования программ* (SRP, Software Restriction Policies), с помощью которой можно, например, запретить запуск сценария, имеющего определенное имя или цифровую подпись.

Параметры реестра, влияющие на политику безопасности для WSH

Режим выполнения сценариев WSH зависит от нескольких параметров системного реестра, которые могут быть записаны в двух разделах:

`HKLM\Software\Microsoft\Windows Script Host\Settings` **(А)**

или

`HKCU\Software\Microsoft\Windows Script Host\Settings` **(Б)**

В разделе **(А)** хранятся установки WSH для всех пользователей, запускающих сценарии на данной машине, а в разделе **(Б)** — для текущего пользователя, зарегистрированного в системе. При этом строковый параметр `IgnoreUserSettings` из раздела **(А)** определяет, откуда именно будут браться параметры: если `IgnoreUserSettings` равен 0 или вообще не задан, то на политику безопасности для сценариев WSH будут влиять

параметры из раздела **(А)**. Если же `IgnoreUserSettings` равен 1, то параметры берутся из раздела **(Б)**.

Параметры реестра, определяющие политику безопасности при использовании сценариев WSH, описаны в табл. 4.2.

Таблица 4.2. *Параметры реестра для WSH*

Параметр	**Тип**	**Описание**
`Enabled`	Строковый (`REG_SZ`)	Если значение равно 0, то выполнение любых сценариев WSH запрещено. Если значение равно 1, то на машине могут выполняться локальные сценарии WSH. Значением по умолчанию является 1
`Remote`	Строковый (`REG_SZ`)	Если значение равно 0, то выполнение удаленных сценариев WSH запрещено. Если значение равно 1, то на машине могут выполняться удаленные (т. е. запускаемые с других компьютеров) сценарии WSH. Значением по умолчанию является 0
`TrustPolicy`	Целый (`REG_DWORD`)	Если значение равно 0, то все сценарии запускаются без проверки их цифровой подписи. Если значение равно 1, то перед запуском неподписанных сценариев или сценариев с подписью, которой соответствует цифровой сертификат, не входящий в число доверяемых, будет выводиться диалоговое окно с предупреждением о возможной опасности такого сценария (при этом есть возможность отказаться от выполнения сценария). Если значение равно 2, то будут запускаться только сценарии, которые подписаны цифровой подписью, и к сертификату, с помощью которого создана эта подпись, установлено доверие. Значением по умолчанию является 0

`UseWINSAFER`	Строковый (`REG_SZ`)	В операционных системах Windows 9x/ ME/ NT/2000 этот параметр не используется. В Windows XP параметр определяет, следует ли к сценариям WSH применять политики ограниченного использования программ (SRP). Если значение равно 0, то к сценариям WSH политики ограниченного использования программ не применяются, а режим выполнения сценариев определяется параметром `TrustPolicy`. Если значение равно 1, то режим выполнения сценариев задается политикой ограниченного использования программ, а параметр `TrustPolicy` игнорируется. Значением по умолчанию является 0
`LogSecurityFailures`	Строковый (`REG_SZ`)	Если значение равно 1, то информация о всех ошибках, которые возникают при выполнении сценариев и связаны с вопросами безопасности, заносится в журнал событий системы (System Event Log). Другими словами, ведется аудит отказов для сценариев WSH. Если значение равно 0, то информация об ошибках в сценариях, которые связаны с нарушениями установленных политик безопасности, не заносится в журнал событий. Значением по умолчанию является 0
`LogSecuritySuccesses`	Строковый (`REG_SZ`)	Если значение равно 1, то ведется аудит успехов для сценариев WSH, т.е. в в журнал событий системы (System Event Log) записывается информация о каждом успешно запущенном сценарии. Если значение равно 0, то аудит успехов не ведется. Значением по умолчанию

является 0

Устанавливая определенным образом значения параметров системного реестра из табл. 4.2, можно настроить политику безопасности при использовании сценариев WSH для конкретного пользователя или рабочей станции (например, можно разрешить выполнение сценариев только администраторам домена). В сетях Windows NT или на автономной машине для этого используется *редактор системной политики* (Poledit.exe), а в сетях с установленной службой каталогов Active Directory применяется *групповая политика* (Group Policy), доступная с помощью оснастки **Active Directory — пользователи и компьютеры** (Active Directory — users and computers) консоли управления MMC. На автономном компьютере с операционной системой Windows XP/2000 для настройки политики безопасности WSH можно также изменять локальную групповую политику, которая позволяет задать одинаковые параметры безопасности для всех пользователей данного компьютера.

В любом случае для более удобного и безопасного изменения значений параметров реестра следует применять *административные шаблоны* (administrative templates) — текстовые файлы с расширением adm, позволяющие создавать удобный пользовательский интерфейс для редактирования заданных параметров системного реестра в редакторе системной политики или при работе с оснасткой **Групповая политика** (Group Policy) в MMC (пример такого adm-файла, позволяющего блокировать выполнение сценариев WSH, приведен в листинге 4.12).

Замечание

Более подробную информацию о редакторе системной политики Poledit.exe, настройке групповой политики в Windows 2000/XP и администраторских шаблонах можно найти в справочной системе Windows XP и документации MSDN.

Блокировка локальных и удаленных сценариев WSH. Пример административного шаблона

Как уже было указано в табл. 4.2, за блокировку локальных и удаленных сценариев WSH отвечают соответственно параметры реестра

Enabled и Remote: если Enabled равно "0", то на машине вообще нельзя выполнять сценарии, если Remote равен "0", то нельзя выполнять сценарии, запускаемые с другого компьютера (удаленные сценарии). При такой блокировке устанавливается запрет на выполнение сценариев всех типов — при попытке запуска любого файла с расширениями js, vbs, jse, vbe или wsf будет выведено диалоговое окно, показанное на рис. 4.13, а сценарий выполнен не будет.

В листинге 4.12 приведен административный шаблон wsh.adm, с помощью которого можно запрещать/разрешать выполнение локальных или удаленных сценариев. Как мы видим, в административном шаблоне описывается путь к разделу реестра (параметр KEYNAME), содержащего нужный параметр, название (параметр VALUENAME) и список возможных значений (параметры VALUEON и VALUEOFF) этого параметра, а также приводятся строки, поясняющие назначение редактируемых параметров (секция [STRINGS]).

Рис. 4.13. Блокировка выполнения сценариев WSH

Листинг 4.12. Файл WSH.adm

```
CLASS MACHINE
  CATEGORY "Локальные и удаленные сценарии WSH"
    POLICY !!LocalEnabledPolicy
      KEYNAME "Software\Microsoft\Windows Script Host\Settings"
        EXPLAIN !!LocalEnabledPolicyHelp
      VALUENAME Enabled
        VALUEON "1" VALUEOFF "0"
    END POLICY
    POLICY !!MachRemotePolicy
      KEYNAME "Software\Microsoft\Windows Script Host\Settings"
        EXPLAIN !!MachRemotePolicyHelp
      VALUENAME Remote
        VALUEON "1" VALUEOFF "0"
    END POLICY
    POLICY !!IgnoreUserSettingsPolicy
      KEYNAME "Software\Microsoft\Windows Script Host\Settings"
```

```
        EXPLAIN !!IgnoreUserHelp
      VALUENAME IgnoreUserSettings
        VALUEON "1" VALUEOFF "0"
    END POLICY
  END CATEGORY

CLASS USER
  CATEGORY "Локальные и удаленные сценарии WSH"
    POLICY !!LocalEnabledPolicy
      KEYNAME "Software\Microsoft\Windows Script Host\Settings"
        EXPLAIN !!LocalEnabledPolicyHelp
      VALUENAME Enabled
        VALUEON "1" VALUEOFF "0"
    END POLICY
    POLICY !!MachRemotePolicy
      KEYNAME "Software\Microsoft\Windows Script Host\Settings"
        EXPLAIN !!MachRemotePolicyHelp
      VALUENAME Remote
        VALUEON "1" VALUEOFF "0"
    END POLICY
  END CATEGORY
[STRINGS]
LocalEnabledPolicy="Разрешить локальный WSH"
LocalEnabledPolicyHelp="Если значение равно 1, то локальный WSH
разрешен"
MachRemotePolicy="Разрешить удаленный WSH"
MachRemotePolicyHelp="Если значение равно 1, то удаленный WSH
разрешен"
IgnoreUserSettingsPolicy="Игнорировать установки пользователя"
IgnoreUserHelp="Должен быть установлен для того, чтобы
применять ко всем пользователям политику для машины"
```

Для применения политик безопасности, которые описаны в шаблоне wsh.adm, к различным пользователям и рабочим станциям, можно воспользоваться либо редактором системной политики Poledit.exe (в сетях Windows NT и на автономных машинах), либо оснасткой **Групповая политика** (Group Policy) в MMC (в сетях с установленной службой каталогов Active Directory и на автономных машинах).

Рассмотрим сначала вариант, связанный с использованием редактора системной политики Poledit.exe. После запуска этой программы надо добавить wsh.adm в список текущих шаблонов политик. Для этого нужно в меню **Параметры** (Options) выбрать пункт **Шаблон политики** (Policy Template) и воспользоваться кнопкой **Добавить** (Add) в диалоговом окне **Параметры шаблона политики** (Policy Template Options) (рис. 4.14).

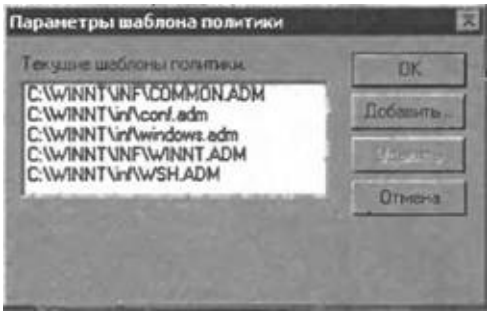


Рис. 4.14. Добавление файла wsh.adm в список текущих шаблонов политик

После подключения шаблона wsh.adm можно создать новую политику (пункт **Новая политика** (New Policy) в меню **Файл** (File)) (рис. 4.15).

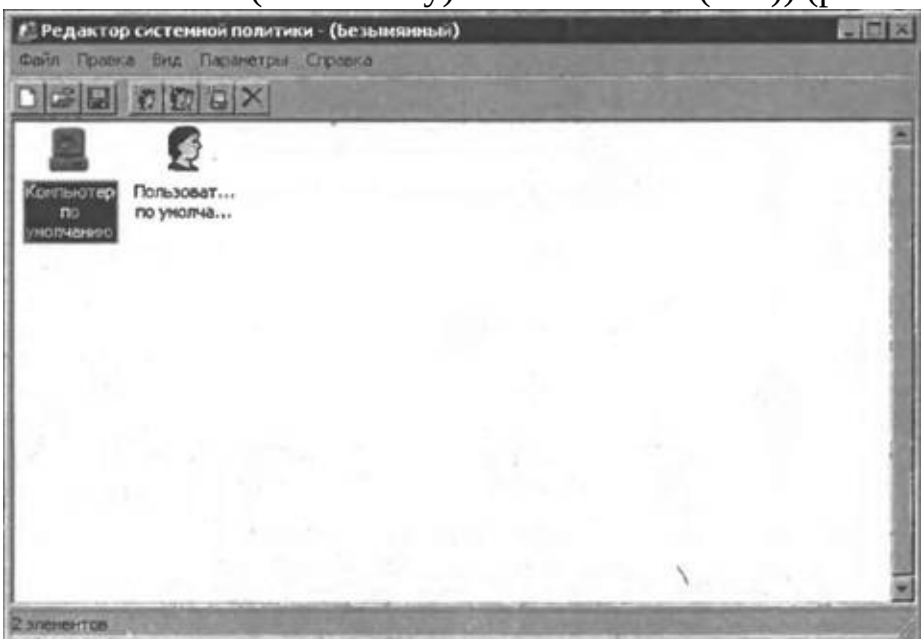


Рис. 4.15. Создание новой системной политики

В свойствах пользователя и компьютера появится новый раздел **Локальные и удаленные сценарии WSH** с соответствующими параметрами политики безопасности, которые описаны в wsh.adm (рис. 4.16).

Рис. 4.16. Параметры политики безопасности для WSH, взятые из шаблона wsh.adm

В Windows 2000/XP, имея соответствующие администраторские права, можно подключить административный шаблон wsh.adm к оснастке **Групповая политика** (Group Policy) в MMC. Рассмотрим, каким образом это делается для автономного компьютера, не подключенного к сети.

Сначала загрузим MMC, выполнив команду mmc либо в командной строке, либо с помощью пункта **Выполнись** (Run) меню **Пуск** (Start). Затем выберем пункт **Добавить или удалить оснастку** (Add/Remove Snap-in) в меню **Консоль** (Console) и нажмем кнопку **Добавить** (Add). В появившемся списке всех имеющихся оснасток нужно выбрать пункт **Групповая политика** (Group Policy) и нажать кнопку **Добавить** (Add). После этого запустится мастер групповой политики, в котором нужно указать, что объектом групповой политики является локальный компьютер, и нажать кнопку **Готово** (Finish) (рис. 4.17).

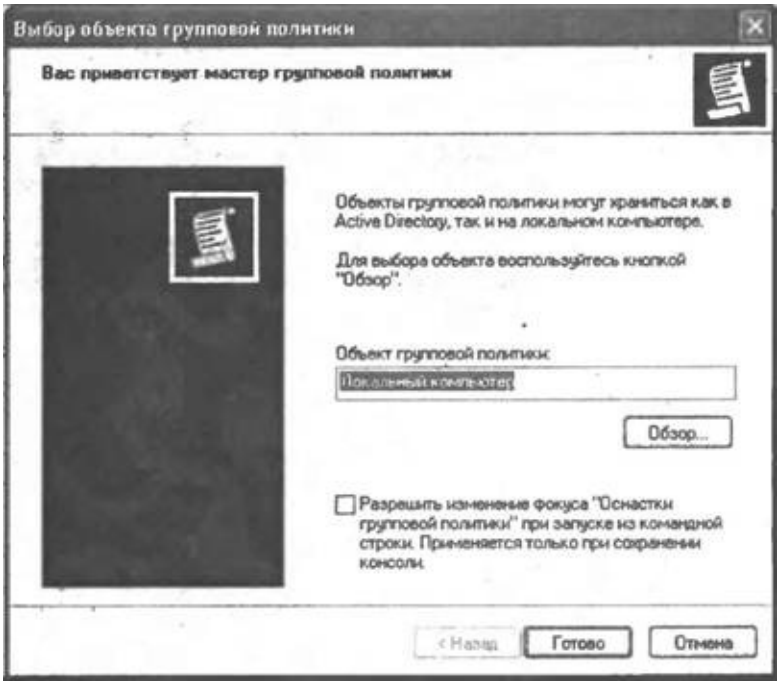


Рис. 4.17. Мастер групповой политики

Никаких других оснасток в окно консоли мы добавлять не будем, поэтому нажимаем кнопку **Закрыть** (Close) в списке оснасток и кнопку **OK** в окне добавления/удаления оснасток. После этого мы можем в окне консоли изменять групповую политику для локального компьютера (рис. 4.18).

Для того чтобы добавить в оснастку нужный нам административный шаблон, следует выделить раздел **Конфигурация компьютера | Административные шаблоны** (Computer Configuration | Administrative Templates) и в меню **Действие** (Action) выбрать пункт **Добавление/ удаление шаблонов** (Add/Remove Templates). После этого на экран будет выведено диалоговое окно со списком всех шаблонов, подключенных к оснастке **Политика "Локальный компьютер"** (Local Computer Policy) (рис. 4.19).

Для добавления нового административного шаблона в этот список нужно нажать кнопку **Добавить** (Add) и выбрать нужный файл с расширением adm (в нашем случае это wsh.adm). После этого список подключенных шаблонов следует закрыть. В результате в каждом из разделов **Конфигурация компьютера | Административные шаблоны** (Computer Configuration | Administrative Templates) и **Конфигурация пользователя | Административные шаблоны** (User Configuration I Administrative Templates) создастся подраздел **Локальные и удаленные**

сценарии WSH (рис. 4.20).

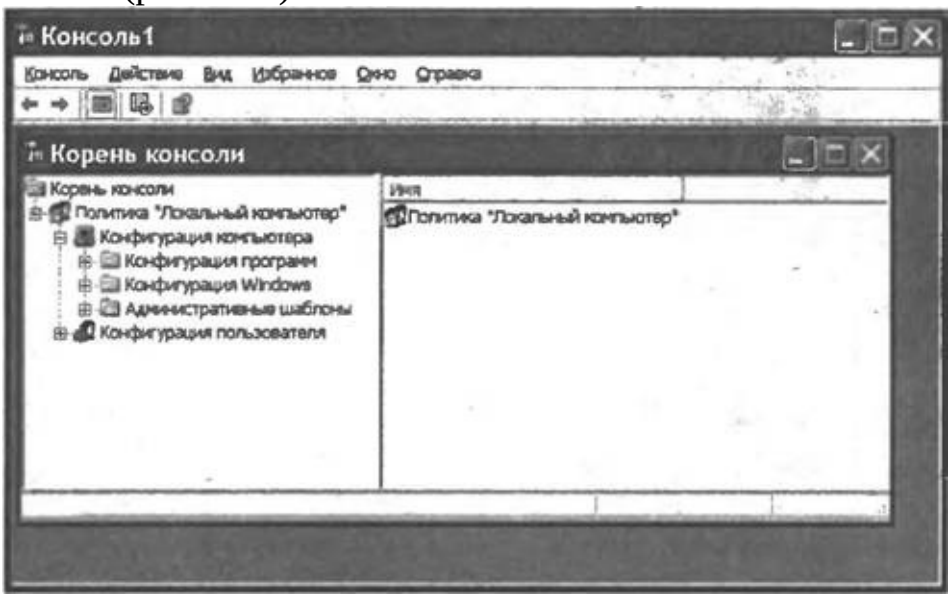


Рис. 4.18. Настройки групповой политики для локального компьютера

Рис. 4.19. Административные шаблоны, подключенные к оснастке **Групповая политика**

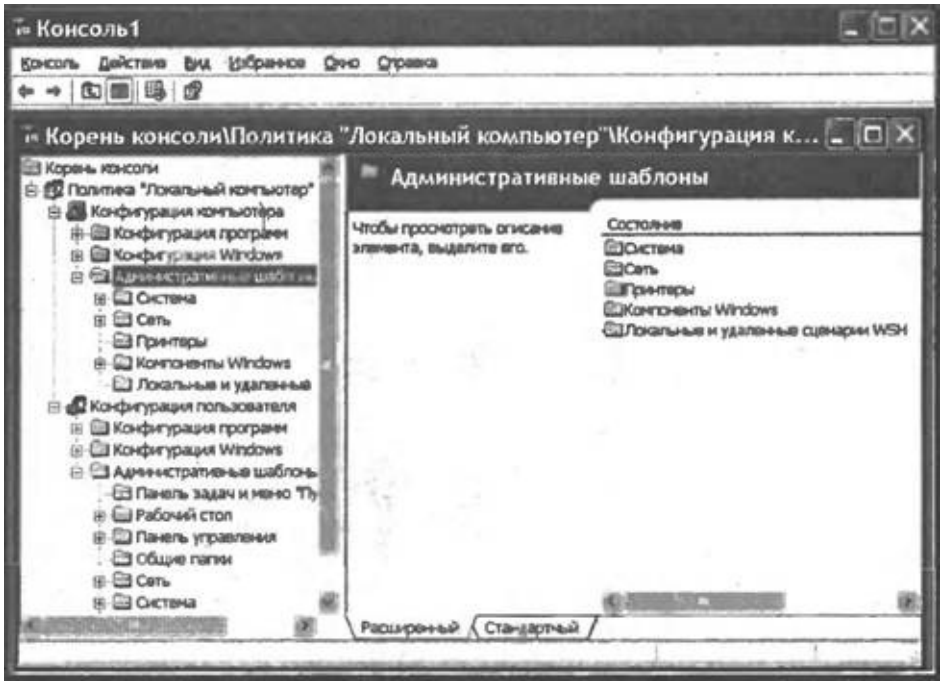


Рис. 4.20. Новый подраздел **Локальные и удаленные сценарии WSH**

Рис. 4.21. Параметры фильтрации административных шаблонов политик

Для того чтобы описанные в wsh.adm параметры можно было редактировать, нужно выделить подраздел **Локальные и удаленные сценарии WSH**, выбрать в меню **Вид** (View) пункт **Фильтрация** (Filter), снять флажок **Показывать только управляемые параметры политики** (Show controlled policy parameters only) в диалоговом окне **Фильтрация** (Filter) и нажать кнопку **OK** (рис. 4.21).

После этого в подразделе **Локальные и удаленные сценарии WSH**

будут показаны описанные в wsh.adm параметры политики безопасности для WSH (рис. 4.22).

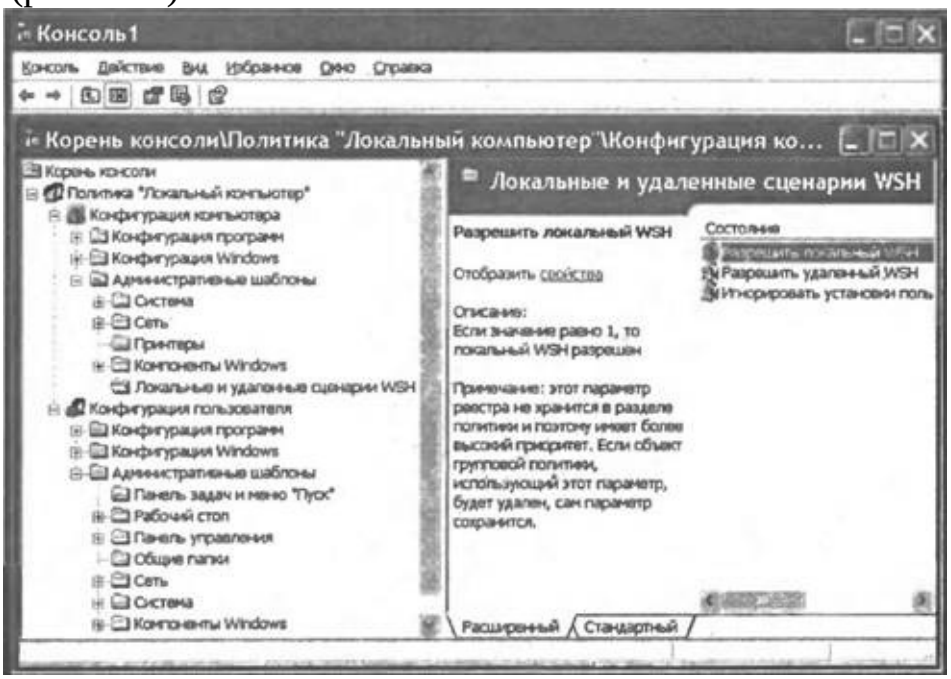


Рис. 4.22. Параметры политики безопасности для WSH, взятые из шаблона wsh.adm

В правой части окна **Консоль1** отображаются состояния локальных и удаленных сценариев WSH. Выбирая соответствующую вкладку, можно переключаться между расширенным и стандартным отображением параметров.

Выбрав с помощью нажатия клавиши <Enter> нужный параметр, можно установить его значение (рис. 4.23).

Рис. 4.23. Изменение значения параметра политики безопасности для WSH

Три режима выполнения сценариев WSH

Для сценариев WSH можно задать один из трех режимов их выполнения:

1. Режим безопасности отключен. Запускаются все сценарии, вне зависимости от того, имеется ли у этих сценариев цифровая подпись или нет.

2. Средний уровень безопасности. Надежные сценарии, т.е. имеющие цифровую подпись, к которой установлено доверие, запускаются сразу, без дополнительных сообщений. При попытке запуска неподписанных сценариев или сценариев, подписанных с помощью цифрового сертификата, к которому не установлено доверие, а также сценариев, содержимое которых было изменено после подписания, выводится диалоговое окно с кратким описанием возникшей ситуации. Данное диалоговое окно позволяет либо проигнорировать предупреждение о возможной небезопасности сценария и запустить его, либо отказаться от выполнения сценария.

3. Сильный уровень безопасности. Надежные сценарии запускаются сразу, без дополнительных сообщений. Всем остальным сценариям будет отказано в запуске.

Как следует из табл. 4.2, для установки той или иной политики безопасности служит параметр реестра целого типа `TrustPolicy`. Значения этого параметра, равные 0, 1 и 2, соответствуют пунктам 1, 2 и 3 вышеприведенного списка.

Замечание

Для установки политики безопасности WSH с помощью `TrustPolicy`, необходимо, чтобы значением параметра `UseWINSAFER` был 0 (либо этот параметр не был указан совсем).

В качестве примера запустим сценарий Signed.vbs с подписью, основанной на цифровом сертификате "Попов ненадежный". Если `TrustPolicy` равно 1, то на экран выведется диалоговое окно, показанное на рис. 4.24.

Рис. 4.24. Предупреждение о безопасности при запуске ненадежного сценария (`TrustPolicy=1`)

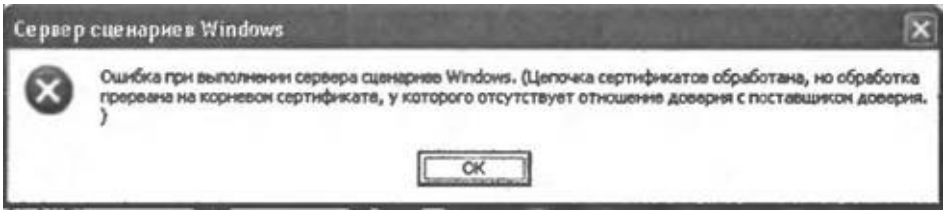


Рис. 4.25. Отказ при запуске ненадежного сценария (`TrustPolicy=2`)

Если же установить значение параметра `TrustPolicy` равным 2 и попытаться выполнить Signed.vbs, то сценарий запущен не будет, а на экран

будет выведено диалоговое окно, показанное рис. 4.25.

Протоколирование действий сценариев в журналах событий

При использовании WSH имеется возможность вести аудит успехов и отказов для сценариев, т.е. автоматически заносить в журнал событий системы информацию об успешных или неуспешных (с точки зрения безопасности) попытках запуска сценариев. Для этой цели служат два параметра системного реестра (`LogSecuritySuccesses` и `LogSecurityFailures`), которые были описаны в табл. 4.2.

Замечание

Для просмотра журнала событий можно воспользоваться соответствующей оснасткой в MMC или выбрать в меню **Пуск** (Start) пункт Все **программы | Администрирование | Просмотр событий** (All Programs | Administrative Tools | Event Viewer).

Рассмотрим пример. Установим режим безопасности так, чтобы для сценариев WSH велся как аудит успехов (`LogSecuritySuccesses="1"`), так и аудит отказов (`LogSecurityFailures="1"`). Если теперь заблокировать сценарии WSH (`Enabled="0"`) и попытаться запустить какой-либо сценарий, то в журнал событий системы будет добавлена запись об отказе (рис. 4.26).

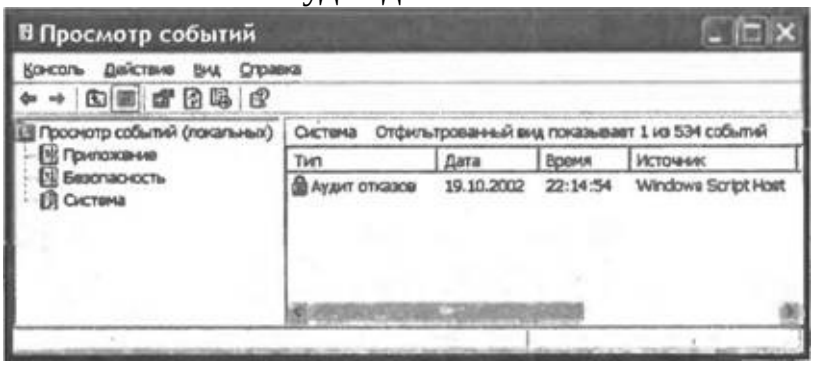


Рис. 4.26. Информация об отказе для сценария WSH в журнале событий системы

Разрешим теперь выполнение сценариев WSH (`Enabled="1"`) и установим режим безопасности, при котором будут запускаться только сценарии с цифровыми подписями, для которых установлено доверие (`TrustPolicy=2`). Если запустить какой-либо сценарий с такой подписью

(например, Signed.js с подписью, созданной на основе сертификата "Попов надежный"), то в журнале событий системы появится запись об успехе (рис. 4.27).

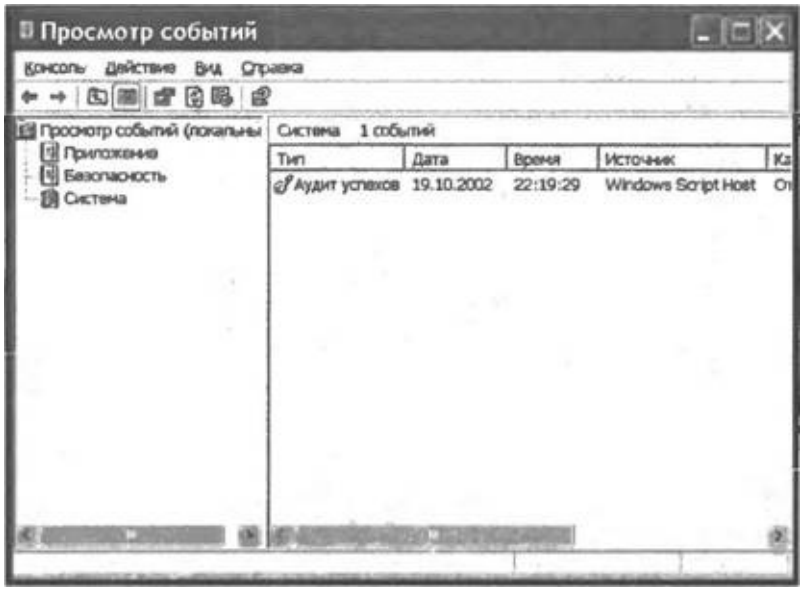


Рис. 4.27. Информация об успехе для сценария WSH в журнале событий системы

Применение к сценариям WSH политики ограниченного использования программ

В Windows XP встроены политики ограниченного использования программ (SRP, Software Restriction Policies), с помощью которых можно управлять возможностью выполнения программного обеспечения на компьютере. Политики SRP могут применяться и к сценариям WSH любого типа, для этого необходимо, чтобы параметр системного реестра `UseWINSAFER`, который описан в табл. 4.2, имел значение 1.

Для применения политик SRP к сценариям WSH на локальной машине необходимо сначала загрузить в MMC оснастку **Политика "Локальный компьютер"** (Local Computer Policy) (рис. 4.28).

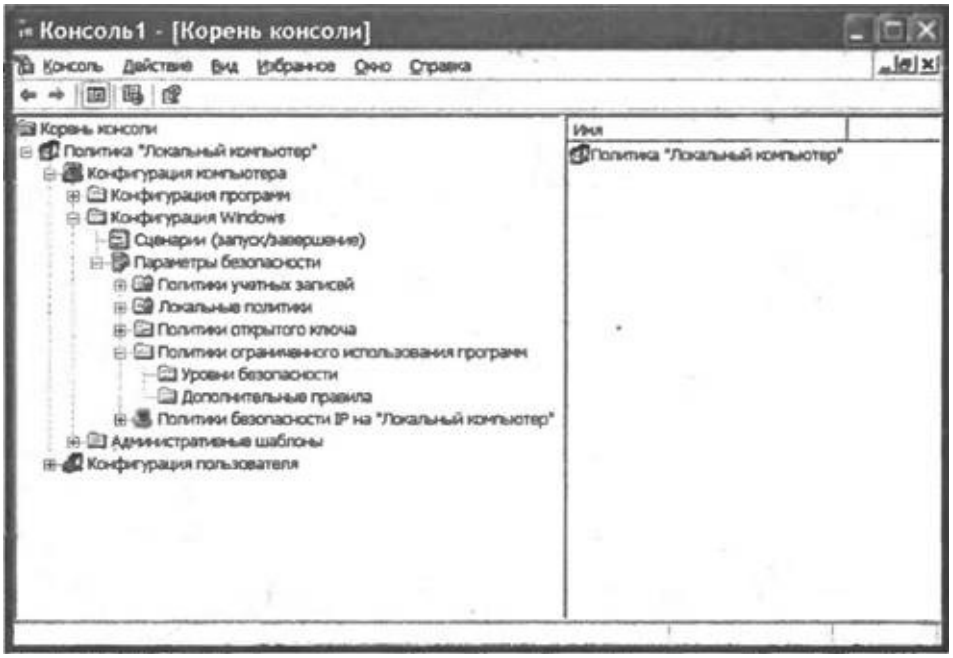


Рис. 4.28. Установка политик SRP с помощью оснастки **Политика "Локальный компьютер"**

Замечание

Процесс загрузки этой оснастки был подробно описан в разд. *"Блокировка локальных и удаленных сценариев WSH. Пример административного шаблона" этой главы*, поэтому здесь мы останавливаться на нем не будем.

После этого нужно перейти в раздел **Конфигурация компьютера | Конфигурация Windows | Параметры безопасности | Политики ограниченного использования программ | Дополнительные правила** (Computer Configuration | Windows Configuration | Security Settings | Software Restriction Policies | Additional Rules).

Создание и изменение дополнительных правил (Additional Rules) в SRP и является механизмом определения политик безопасности для сценариев WSH.

Блокировка сценария с заданным именем

Для того чтобы, пользуясь SRP, запретить выполнение сценариев с

определенными именами, нужно создать новое *правило для пути* (Path Rule), которое позволяет идентифицировать программы по пути к ним. Рассмотрим, например, каким образом можно запретить выполнение всех сценариев, написанных на языке VBScript (т.е. запретить запуск всех файлов с расширением vbs). Для этого нужно, находясь в разделе **Дополнительные правила** (Additional Rules), выбрать в меню **Действие** (Action) пункт **Создать правило для пути** (New Path Rule), после чего на экран будет выведено диалоговое окно, в котором нужно описать создаваемое правило. Здесь в поле **Путь** (Path) укажем маску "*.vbs" для всех VBScript-сценариев, а в раскрывающемся списке **Уровень безопасности** (Security level) выберем значение "Не разрешено" ("Disallowed") (рис. 4.29).

Рис. 4.29. Диалоговое окно для создания нового правила для пути

После ввода такого дополнительного правила SRP на машине нельзя будет запускать сценарии с расширением vbs. При попытке выполнения VBScript- сценария будет выведено диалоговое окно с информацией о невозможности его запуска (рис. 4.30).

Также в разделе **Дополнительные правила** (Additional rules) имеется возможность создать *правило для хеша* (хеш — Это серия байтов фиксированной длины, которая рассчитывается по специальному алгоритму и однозначно идентифицирует содержимое файла). С помощью этого правила можно запретить выполнение файла, имеющего заданный хеш. Однако для сценариев, в отличие от компилированных исполняемых

модулей, это не является подходящим ограничением, т.к. хеш файла со сценарием можно изменить простым добавлением пустой строки в текст сценария.

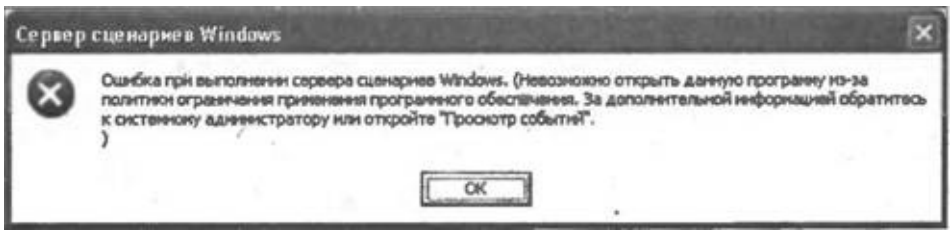


Рис. 4.30. Блокировка сценариев WSH с помощью дополнительных правил SRP

Блокировка сценариев с заданной подписью

Еще одним возможным ограничением является запрет на выполнение файлов, подписанных с помощью определенного цифрового сертификата ("Попов ненадежный", например).

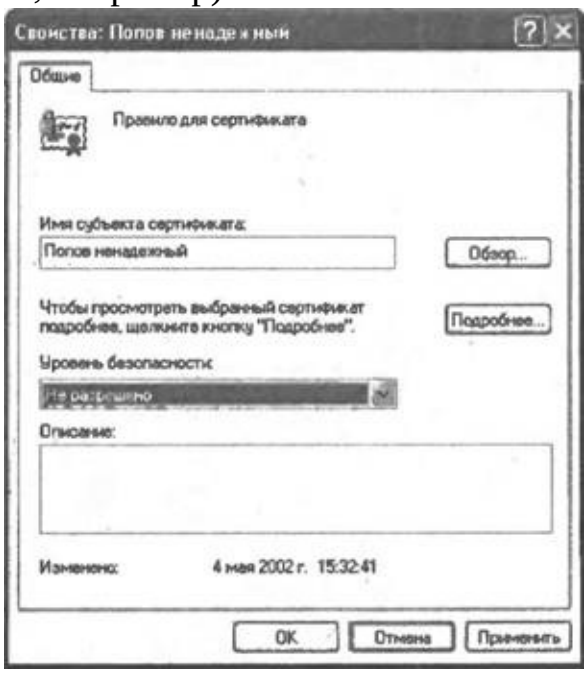


Рис. 4.31. Диалоговое окно для создания нового правила для сертификата

Для этого нужно в разделе **Дополнительные правила** (Additional rules) создать новое *правило для сертификата* (Certificate Rule) (пункт **Создать правило для сертификата** (New Certificate Rule) в меню **Действие** (Action)). В диалоговом окне **Создание правила для**

сертификата (New Certificate Rule) в качестве имени субъекта сертификата (Certificate subject name) укажем с помощью кнопки **Обзор** (Browse) файл C:\Script\Попов.cer (процесс создания этого файла описан в *разд. "Управление сертификатами с помощью MMC"*), а в раскрывающемся списке Уровень безопасности (Security level) выберем значение "Не разрешено" ("Disallowed") — рис. 4.31.

После ввода этого дополнительного правила SRP на машине нельзя будет запускать никакие исполняемые файлы (сценарии WSH в том числе), которые подписаны с использованием сертификата "Попов ненадежный".

Глава 5
Доступ из сценариев к файловой системе

Сценарии WSH позволяют получить полный доступ к файловой системе компьютера, в отличие от JScript- или VBScript-сценариев, внедренных в HTML-страницы, где в зависимости от уровня безопасности, который устанавливается в настройках браузера, те или иные операции могут быть запрещены.

Выполнение основных операций с файловой системой

Для работы с файловой системой из сценариев WSH предназначены восемь объектов, главным из которых является `FileSystemObject`. С помощью методов объекта `FileSystemObject` можно выполнять следующие основные действия:

- □ копировать или перемещать файлы и каталоги;
- □ удалять файлы и каталоги;
- □ создавать каталоги;
- □ создавать или открывать текстовые файлы;
- □ создавать объекты `Drive`, `Folder` и `File` для доступа к конкретному диску, каталогу или файлу соответственно.

С помощью свойств объектов `Drive`, `Folder` и `File` можно получить детальную информацию о тех элементах файловой системы, с которыми они ассоциированы. Объекты `Folder` и `File` также предоставляют методы для манипулирования файлами и каталогами (создание, удаление, копирование, перемещение); эти методы в основном копируют соответствующие методы объекта `FileSystemObject`.

Кроме этого, имеются три объекта-коллекции: `Drives`, `Folders` и `Files`. Коллекция `Drives` содержит объекты `Drive` для всех имеющихся в системе дисков, `Folders` — объекты `Folder` для всех подкаталогов заданного каталога, `Files` — объекты `File` для всех файлов, находящихся внутри определенного каталога.

Наконец, из сценария можно читать информацию из текстовых файлов и записывать в них данные. Методы для этого предоставляет объект `TextStream`.

В табл. 5.1 кратко описано, какие именно объекты, свойства и методы могут понадобиться для выполнения наиболее часто используемых файловых операций.

Таблица 5.1. *Выполнение основных файловых операций*

Операция	Используемые объекты, свойства и методы
Получение сведений об определенном диске (тип файловой	Свойства объекта `Drive`. Сам объект `Drive` создается с помощью метода

системы, метка тома, общий объем и количество свободного места и т.д.)	`GetDrive` объекта `FileSystemObject`
Получение сведений о заданном каталоге или файле (дата создания или последнего доступа, размер, атрибуты и т.д.)	Свойства объектов `Folder` и `File`. Сами эти объекты создаются с помощью методов `GetFolder` и `GetFile` объекта `FileSystemObject`
Проверка существования определенного диска, каталога или файла	Методы `DriveExists`, `FolderExists` и `FileExists` объекта `FileSystemObject`
Копирование файлов и каталогов	Методы `CopyFile` и `CopyFolder` объекта `FileSystemObject`, а также методы `File.Copy` и `Folder.Copy`
Перемещение файлов и каталогов	Методы `MoveFile` и `MoveFolder` объекта `FileSystemObject` или методы `File.Move` и `Folder.Move`
Удаление файлов и каталогов	Методы `DeleteFile` и `DeleteFolder` объекта `FileSystemObject` или методы `File.Delete` и `Folder.Delete`
Создание каталога	Методы `FileSystemObject.CreateFolder` или `Folders.Add`
Создание текстового файла	Методы `FileSystemObject.CreateTextFile` или `Folder.CreateTextFile`
Получение списка всех доступных дисков	Коллекция `Drives`, содержащаяся в свойстве `FileSystemObject.Drives`
Получение списка всех подкаталогов заданного каталога	Коллекция `Folders`, содержащаяся в свойстве `Folder.SubFolders`
Получение списка всех файлов заданного каталога	Коллекция `Files`, содержащаяся в свойстве `Folder.Files`
Открытие текстового файла для чтения, записи или добавления	Методы `FileSystemObject.CreateTextFile` или `File.OpenAsTextStream`
Чтение информации из заданного текстового файла или запись ее в него	Методы объекта `TextStream`

Перейдем теперь к подробному рассмотрению объектов, используемых при работе с файловой системой.

Объект *FileSystemObject*

Объект `FileSystemObject` является основным объектом, обеспечивающим доступ к файловой системе компьютера; его методы используются для создания остальных объектов (`Drives`, `Drive`, `Folders`, `Folder`, `Files`, `File` и `TextStream`).

Для создания внутри сценария экземпляра объекта `FileSystemObject` можно воспользоваться методом `CreateObject` объекта `WScript`:

```
var FSO = WScript.CreateObject("Scripting.FileSystemObject");
```

Также можно использовать объект `ActiveXObject` языка JScript (с помощью этого объекта можно работать с файловой системой из сценариев, находящихся внутри HTML-страниц):

```
var FSO = new ActiveXObject("Scripting.FileSystemObject");
```

Объект `FileSystemObject` имеет единственное свойство `Drives`, в котором хранится коллекция, содержащая объекты `Drive` для всех доступных дисков компьютера. Примеры, иллюстрирующие использование свойства `Drives` приведены ниже в *разд. "Коллекция Drives"*.

Методы объекта `FileSystemObject` представлены в табл. 5.2.

Таблица 5.2. *Методы объекта FileSystemObject*

Метод	Описание
`BuildPath(`*`path, name`*`)`	Добавляет к заданному пути (параметр *`path`*) новое имя (параметр *`name`*)
`CopyFile(`*`source, destination [, overwrite]`*`)`	Копирует один или несколько файлов из одного места (параметр *`source`*) в другое (параметр *`destination`*)
`CopyFolder(`*`source, destination`* `[, `*`overwrite`*`])`	Копирует каталог со всеми подкаталогами из одного места (параметр *`source`*) в другое (параметр *`destination`*)
`CreateFolder(`*`foldername`*`)`	Создает новый каталог с именем *`foldername`*. Если каталог *`foldername`* уже существует, то произойдет ошибка
`CreateTextFile(`*`filename`*	Создает новый текстовый файл с именем *`filename`* и возвращает указывающий на

[/overwrite[, unicode]])	этот файл объект TextStream
DeleteFile(*filespec [, force]*)	Удаляет файл, путь к которому задан параметром *filespec*
DeleteFolder(*folderspec [, force]*)	Удаляет каталог, путь к которому задан параметром *folderspec*, вместе со всем его содержимым
DriveExists(*drivespec*)	Возвращает True, если заданное параметром *drivespec* устройство существует и False в противном случае
FileExists(*filespec*)	Возвращает True, если заданный параметром *filespec* файл существует и False в противном случае
FolderExists(*folderspec*)	Возвращает True, если заданный параметром *folderspec* каталог существует и False в противном случае
GetAbsolutePathName(*pathspec*)	Возвращает полный путь для заданного относительного пути *pathspec* (из текущего каталога)
GetBaseName(*path*)	Возвращает базовое имя (без расширения) для последнего компонента в пути *path*
GetDrive(*drivespec*)	Возвращает объект Drive, соответствующий диску, заданному параметром *drivespec*
GetDriveName(*path*)	Возвращает строку, содержащую имя диска в заданном пути. Если из параметра *path* нельзя выделить имя диска, то метод возвращает пустую строку (" ")
GetExtensionName(*path*)	Возвращает строку, содержащую расширение для последнего компонента в пути *path*. Если из параметра *path* нельзя выделить компоненты пути, то GetExtensionName возвращает пустую строку (""). Для сетевых дисков корневой каталог (\) рассматривается как компонент пути

`GetFile(`*`filespec`*`)`	Возвращает объект `File`, соответствующий файлу, заданному параметром*`filespec`*. Если файл, путь к которому задан параметром*`filespec`*, не существует, то при выполнении метода `GetFile` возникнет ошибка
`GetFileName(`*`pathspec`*`)`	Возвращает имя файла, заданного полным путем к нему. Если из параметра*`pathspec`* нельзя выделить имя файла, метод `GetFileName` возвращает пустую строку ("")
`GetFolder(`*`folderpec`*`)`	Возвращает объект `Folder`, соответствующий каталогу, заданному параметром*`folderspec`*. Если каталог, путь к которому задан параметром *`folderspec`*, не существует, при выполнении метода `GetFolder` возникнет ошибка
`GetParentFolderName(`*`path`*`)`	Возвращает строку, содержащую имя родительского каталога для последнего компонента в заданном пути. Если для последнего компонента в пути, заданном параметром*`path`*, нельзя определить родительский каталог, то метод возвращает пустую строку ("")
`GetSpecialFolder(`*`folderpec`*`)`	Возвращает объект `Folder` для некоторых специальных папок Windows, заданных числовым параметром*`folderspec`*
`GetTempName()`	Возвращает случайным образом сгенерированное имя файла или каталога, которое может быть использовано для операций, требующих наличия временного файла или каталога
`MoveFile(`*`source, destination`*`)`	Перемещает один или несколько файлов из одного места (параметр *`source`*) в другое (параметр *`destination`*)
	Перемещает один или несколько

`MoveFolder(`*`source, destination`*`)`	каталогов из одного места (параметр*`source`*) в другое (параметр*`destination`*)
`OpenTextFile(`*`filename[, iomode[, create[, format]]]`*`)`	Открывает заданный текстовый файл и возвращает объект `TextStream` для работы с этим файлом

Сами названия методов объекта `FileSystemObject` довольно прозрачно указывают на выполняемые ими действия. Приведем необходимые пояснения и примеры для перечисленных методов.

Методы *CopyFile* и *CopyFolder*

Для копирования нескольких файлов или каталогов в последнем компоненте параметра *`source`* можно указывать групповые символы "?" и "*"; в параметре *`destination`* групповые символы недопустимы. Например, следующий пример является корректным кодом:

```
    FSO = WScript.CreateObject("Scripting.FileSystemObject");
    FSO.CopyFile("c:\\mydocuments\\letters\\*.doc",
"c:\\tempfolder\\");
```

А так писать нельзя:

```
    FSO = WScript.CreateObject("Scripting.FileSystemObject");
    FSO.CopyFile("c:\\mydocuments\\*\\R1???97.xls",
"c:\\tempfolder");
```

Необязательный параметр *`overwrite`* является логической переменной, определяющей, следует ли заменять уже существующий файл/каталог с именем *`destination`*(*`overwrite`*=*`true`*) или нет (*`overwrite`*=`false`).

При использовании методов `CopyFile` и `CopyFolder` процесс копирования прерывается после первой возникшей ошибки (как и в команде `COPY` операционной системы).

Метод *CreateTextFile*

Параметр *`overwrite`*, используемый в методе, имеет значение в том случае, когда создаваемый файл уже существует. Если *`overwrite`* равно `true`, то такой файл перепишется (старое содержимое будет утеряно), если же в качестве *`overwrite`* указано `false`, то файл переписываться не будет. Если этот параметр вообще не указан, то существующий файл также не будет переписан.

Параметр *unicode* является логическим значением, указывающим, в каком формате (ASCII или Unicode) следует создавать файл. Если *unicode* равно `true`, то файл создается в формате Unicode, если же *unicode* равно `false` или этот параметр вообще не указан, то файл создается в режиме ASCII.

Для дальнейшей работы с созданным файлом, т.е. для записи или чтения информации, нужно использовать методы объекта `TextStream`. Соответствующий пример сценария приведен в листинге 5.1.

Листинг 5.1. Создание текстового файла и запись в него строки

```
    /*****************************************************************
    /*                                                          Имя:
CreateFile.js                                                  */
    /*                                                         Язык:
JScript                                                       */
    /*  Описание:  Создание  текстового  файла  и  запись  в  него
строки      */
    /*****************************************************************
    var FSO,f; //Объявляем переменные
    //Создаем объект FileSystemObject
    FSO = WScript.CreateObject("Scripting.FileSystemObject");
    //Создаем на диске C: текстовый файл TestFile.txt
    f = FSO.CreateTextFile("C:\\TestFile.txt", true);
    //Записываем строку в файл
    f.WriteLine("Привет!");
    //Закрываем файл
    f.Close();
    /*************                                            Конец
*******************************************/
```

Методы *DeleteFile* и *DeleteFolder*

Параметры *filespec* или *folderspec*, используемые в методах, могут содержать групповые символы "?" и "*" в последнем компоненте пути для удаления сразу нескольких файлов/каталогов.

Если параметр *force* равен `false` или не указан вовсе, то с помощью методов `DeleteFile` или `DeleteFolder` будет нельзя удалить файл/каталог с атрибутом "только для чтения" (read-only). Установка для *force* значения `true` позволит сразу удалять такие файлы/каталоги.

Замечание

При использовании метода `DeleteFolder` неважно, является ли удаляемый каталог пустым или нет — он удалится в любом случае

Если заданный для удаления файл/каталог не будет найден, то возникнет ошибка.

Метод *DriveExists*

Для дисководов со съемными носителями метод `DriveExists` вернет `true` даже в том случае, если носитель физически отсутствует. Для того чтобы определить готовность дисковода, нужно использовать свойство `IsReady` соответствующего объекта `Drive`.

В качестве примера использования метода `DriveExists` приведем функцию `ReportDriveStatus`, которая возвращает информацию о наличии диска, передаваемого в эту функцию в качестве параметра (листинг 5.2).

Листинг 5.2. Функция `ReportDriveStatus`

```
function ReportDriveStatus(drv) {
 var FSO, s ="" //Объявляем переменные
 //Создаем объект FileSystemObject
 FSO = WScript.CreateObject("Scripting.FileSystemObject");
 //Проверяем наличие диска drv
 if (FSO.DriveExists(drv)) s += "Диск " + drv + " существует.";
 else s += "Диск " + drv + " не существует.";
 return(s);
}
```

Функция `ReportDriveStatus` будет возвращать информацию о наличии диска, передаваемого в эту функцию в качестве параметра.

Метод *GetAbsolutePathName*

Для иллюстрации работы этого метода предположим, что текущим каталогом является C:\MyDocuments\Reports. В табл. 5.3 приведены значения, возвращаемые методом `GetAbsolutePathName`, при различных

значениях параметра *pathspec*.

Таблица 5.3. *Варианты работы метода* `GetAbsolutePathName`

Параметр*pathspec*	**Возвращаемое значение**
"C:"	"C:\MyDocuments\Reports"
"C:.."	"C:\MyDocuments"
"C:\\"	"C:\"
"Region1"	"C:\MyDocuments\Reports\Region1"
"C:\\..\\..\\MyDocuments"	"C:\MyDocuments"

Метод *GetBaseName*

Работу этого метода иллюстрирует сценарий BaseName.js, приведенный в листинге 5.3. В результате выполнения этого сценария на экран выводится диалоговое окно, в котором отражены полный путь к файлу и базовое имя, выделенное из этого пути (рис. 5.1).

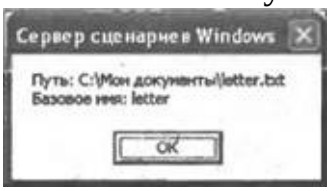


Рис. 5.1. Полный путь к файлу и базовое имя для этого пути

Листинг 5.3. Выделение базового имени файла

```
    /*****************************************************************
    /*                                                          Имя:
BaseName.js                                                  */
    /*                                                         Язык:
JScript                                                     */
    /*  Описание:  Создание  текстового  файла  и  запись  в  него
строки      */
    /*****************************************************************
    var FSO, BaseName, SPath,s;  //Объявляем переменные
    //Создаем объект FileSystemObject
    FSO = WScript.CreateObject("Scripting.FileSystemObject");
    //Задаем путь к файлу
    SPath="C:\\Мои документы\\letter.txt";
    //Выделяем базовое имя файла (без расширения)
```

```
    BaseName = FSO.GetBaseName(SPath);
    //Выводим на экран путь и базовое имя
    s="Путь: "+SPath+"\n";
    s+="Базовое имя: "+BaseName;
    WScript.Echo(s);
    /*************                                          Конец
*******************************************/
```

Метод *GetDrive*

Параметр *drivespec* в данном методе может задаваться одной буквой (например, "C"), буквой с двоеточием ("C:"), буквой с двоеточием и символом разделителя пути ("C:\\"). Для сетевого диска drivespec можно указывать в формате UNC (например, "Server1\\Programs").

Если параметр *drivespec* указывает на несуществующий диск или задан в неверном формате, то при выполнении метода GetDrive возникнет ошибка.

Если вам требуется преобразовать строку, содержащую обычный путь к файлу или каталогу, в вид, пригодный для GetDrive, необходимо применять методы GetAbsolutePathName и GetDriveName:

```
    DriveSpec = GetDriveName(GetAbsolutePathName(Path))
```

Метод *GetParentFolderName*

Для иллюстрации работы этого метода запустим сценарий ParentFolder.js (листинг 5.4). В результате будет выведено диалоговое окно с полным путем к файлу и путь к родительскому каталогу этого файла (рис. 5.2).

Рис. 5.2. Полный путь к файлу и родительский каталог этого файла

Листинг 5.4. Определение родительского каталога для файла

```
    /*****************************************************************
```

```
    /*                                                          Имя:
ParentFolder.js                                                  */
    /*                                                          Язык:
JScript                                                         */
    /*   Описание:   Определение   родительского   каталога   для
файла          */
    /*************************************************************
    var FSO,ParentFolder,Spath,s;  //Объявляем переменные
    //Создаем объект FileSystemObject
    FSO = WScript.CreateObject("Scripting.FileSystemObject");
    //Задаем путь к файлу
    SPath="C:\\Programs\\letter.txt";
    //Определяем родительский каталог для файла letter.txt
    ParentFolder = FSO.GetParentFolderName(SPath);
    s="Путь: "+SPath+"\n";
    s+="Родительский каталог: "+ParentFolder;
    //Выводим на экран полный путь к файлу letter.txt
    //и родительский каталог для этого файла
    WScript.Echo(s);
    /*************                                             Конец
*****************************************/
```

Метод *GetSpecialFolder*

Параметр *folderspec* в этом методе является числом и может принимать значения, описанные в табл. 5.4.

Таблица 5.4. *Значения параметра folderspec*

Константа	Значение	Описание
WindowsFolder	0	Каталог Windows (например, "C:\Windows")
SystemFolder	1	Системный каталог, содержащий файлы библиотек, шрифтов и драйверы устройств
TemporaryFolder	2	Каталог для временных файлов, путь к которому хранится в переменной среды TMP

Метод *GetTempName*

Метод GetTempName только возвращает имя файла, но не создает его. Для создания файла можно воспользоваться методом CreateTextFile, подставив в качестве параметра этого метода сгенерированное случайное имя (листинг 5.5).

Листинг 5.5. Создание временного файла со случайным именем

```
    /*****************************************************************
    /*                                                          Имя:
TempFile.js                                                  */
    /*                                                         Язык:
JScript                                                    */
    /*   Описание:   Создание   временного   файла   со   случайным
именем         */
    /*****************************************************************
    var FSO,FileName,f,s;  //Объявляем переменные
    //Создаем объект FileSystemObject
    FSO = WScript.CreateObject("Scripting.FileSystemObject");
    //Генерируем случайное имя файла
    FileName=FSO.GetTempName();
    //Создаем файл и именем FileName
    f = FSO.CreateTextFile(FileName, true);
    //Закрываем файл
    f.Close();
    //Сообщаем о создании файла
    WScript.Echo("Был создан файл",FileName);
    /*************                                            Конец
*********************************************/
```

Методы *MoveFile* и *MoveFolder*

Как и при использовании методов `CopyFile` и `CopyFolder`, для перемещения нескольких файлов или каталогов в последнем компоненте параметра *`source`* можно указывать групповые символы (? и *); в параметре *`destination`* групповые символы недопустимы.

При использовании методов `MoveFile` и `MoveFolder` процесс перемещения прерывается после первой возникшей ошибки (как и в команде move операционной системы). Перемещать файлы и каталоги с одного диска на другой нельзя.

Метод *OpenTextFile*

Числовой параметр *`iomode`* задает режим ввода/вывода для открываемого файла и может принимать следующие значения (табл. 5.5).

Таблица 5.5. *Параметр `iomode`*

Константа	Значение	Описание
`ForReading`	1	Файл открывается только для чтения, записывать информацию в него нельзя
`ForWriting`	2	Файл открывается для записи. Если файл с таким именем уже существовал, то при новой записи его содержимое теряется
`ForAppending`	8	Файл открывается для добавления. Если файл уже существовал, то информация будет дописываться в конец этого файла

Параметр *`create`* имеет значение в том случае, когда открываемый файл физически не существует. Если *`create`* равно `true`, то этот файл создастся, если же в качестве значения create указано `false` или параметр *`create`* опущен, то файл создаваться не будет.

Числовой параметр *`format`* определяет формат открываемого файла (табл. 5.6).

Таблица 5.6. *Параметр `format`*

Константа	Значение	Описание
`TristateUseDefault`	-2	Файл открывается в формате, используемом системой по умолчанию
`TristateTrue`	-1	Файл открывается в формате Unicode
`TristateFalse`	0	Файл открывается в формате ASCII

Для дальнейшей работы с открытым файлом, т.е. для записи или чтения информации, нужно использовать методы объекта `TextStream`.

В следующем примере с помощью метода `OpenTextFile` текстовый файл открывается в режиме добавления информации (листинг 5.6).

Листинг 5.6. Добавление информации в текстовый файл

```
    /*****************************************************************
    /*                                                          Имя:
AppendFile.js                                                  */
    /*                                                         Язык:
JScript                                                       */
    /*     Описание:     Добавление     строки     в     текстовый
```

```
файл                    */
    /*************************************************************
    //Объявляем переменные и инициализируем константы
    var FSO,f,ForAppending = 8;
    //Создаем объект FileSystemObject
    FSO = WScript.CreateObject("Scripting.FileSystemObject");
    //Открываем файл
    f = FSO.OpenTextFile("C:\\TestFile.txt", ForAppending, true);
    //Добавление в файл строку
    f.WriteLine("Привет!");
    //Закрываем файл
    f.Close();
    /*************                                          Конец
*****************************************/
```

Замечание

Мнемонические константы, используемые в качестве параметров *iomode* и *create*, можно не определять явным образом **в** сценарии, как это сделано в вышеприведенном примере, а брать из самого объекта FileSystemObject (точнее говоря, из библиотеки типов этого объекта). Для этого в сценариях нужно применять разметку XML (см. листинг 3.9).

Объект *Drive*

С помощью объекта Drive можно получить доступ к свойствам заданного локального или сетевого диска. Создается объект Drive с помощью метода GetDrive объекта FileSystemObject следующим образом:

```
var FSO, D;
FSO = WScript.CreateObject("Scripting.FileSystemObject");
D = FSO.GetDrive("C:");
```

Также объекты Drive могут быть получены как элементы коллекции Drives.

Свойства объекта Drive представлены в табл. 5.7; методов у этого объекта нет.

Таблица 5.7. *Свойства объекта Drive*

Свойство	Описание
	Содержит количество доступного для пользователя места

`AvailableSpace`	(в байтах) на диске
`DriveLetter`	Содержит букву, ассоциированную с локальным устройством или сетевым ресурсом. Это свойство доступно только для чтения
`DriveType`	Содержит числовое значение, определяющее тип устройства: 0 — неизвестное устройство; 1 — устройство со сменным носителем; 2 — жесткий диск; 3 — сетевой диск; 4 — CD-ROM; 5 — RAM-диск
`FileSystem`	Содержит тип файловой системы, использующейся на диске (FAT, NTFS или CDFS)
`FreeSpace`	Содержит количество свободного места (в байтах) на локальном диске или сетевом ресурсе. Доступно только для чтения
`IsReady`	Содержит `true`, если устройство готово, и `false` в противном случае. Для устройств со сменными носителями и приводов CD-ROM `IsReady` возвращает `true` только в том случае, когда в дисковод вставлен соответствующий носитель и устройство готово предоставить доступ к этому носителю
`Path`	Содержит путь к диску (например, "`C:`", но не "`C:\`")
`RootFolder`	Содержит объект `Folder`, соответствующий корневому каталогу на диске. Доступно только для чтения
`SerialNumber`	Содержат десятичный серийный номер тома заданного диска
`ShareName`	Содержит сетевое имя для диска. Если объект не является сетевым диском, то в свойстве `ShareName` содержится пустая строка ("")
`TotalSize`	Содержит общий объем в байтах локального диска или сетевого ресурса
`VolumeName`	Содержит метку тома для диска. Доступно для чтения и записи

В листинге 5.7 приведен сценарий DriveInfo.js, в котором объект `Drive` используется для доступа к некоторым свойствам диска C: (рис. 5.3).

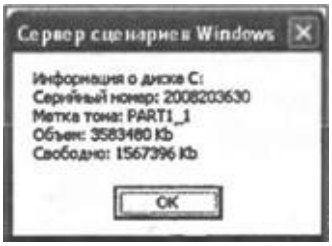


Рис. 5.3. Свойства диска C:

Листинг 5.7. Получение свойств диска C

```
    /*****************************************************************
    /*                                                          Имя:
DriveInfo.js                                                  */
    /*                                                         Язык:
JScript                                                     */
    /*     Описание:     Вывод     на     экран     свойств     диска
C:                        */
    /*****************************************************************
    //Объявляем переменные
    var FSO,D,TotalSize,FreeSpace,s;
    //Создаем объект FileSystemObject
    FSO = WScript.CreateObject("Scripting.FileSystemObject");
    //Создаем объект Drive для диска C:
    D = FSO.GetDrive("C:");
    s="Информация о диске C:\n";
    //Получаем серийный номер диска
    s+="Серийный номер: "+D.SerialNumber+"\n";
    //Получаем метку тома диска
    s+="Метка тома: "+D.VolumeName+"\n";
    //Вычисляем общий объем диска в килобайтах
    TotalSize=D.TotalSize/1024;
    s+="Объем: "+TotalSize+" Kb\n";
    //Вычисляем объем свободного пространства диска в килобайтах
    FreeSpace=D.FreeSpace/1024;
    s+="Свободно: "+FreeSpace+" Kb\n";
    //Выводим свойства диска на экран
    WScript.Echo(s);
    /*************                                             Конец
*********************************************/
```

Коллекция *Drives*

Доступная только для чтения коллекция `Drives` содержит объекты `Drive` для всех доступных дисков компьютера, в том числе для сетевых дисков и дисководов со сменными носителями.

В свойстве `Count` коллекции `Drives` хранится число ее элементов, т.е. число доступных дисков.

С помощью метода `Item(`*`drivespec`*`)` можно получить доступ к объекту `Drive` для диска, заданного параметром*`drivespec`*. Например:

```
var FSO, DriveCol, D; //Создаем объект FileSystemObject
FSO = WScript.CreateObject("Scripting.FileSystemObject");
//Создаем коллекцию имеющихся в системе дисков
DriveCol = FSO.Drives;
// Извлечение элемента коллекции (диск C:)
D = DriveCol.Item("C:");
//Вывод на экран метки тома диска C:
WScript.Echo("Диск C: имеет метку", D.VolumeName);
```

Для перебора всех элементов коллекции `Drives` нужно, как обычно, использовать объект `Enumerator`.

В листинге 5.8 приведен файл ListDrives.js, в котором с помощью объекта `Enumerator` на экран выводятся сведения обо всех доступных дисках (рис. 5.4).

Рис. 5.4. Список всех дисков, имеющихся в системе

Листинг 5.8. Построение списка всех имеющихся дисков

```
/*****************************************************************
/*                                                          Имя:
ListDrives.js                                              */
/*                                                         Язык:
JScript                                                   */
/*     Описание:     Получение     списка     всех     имеющихся
дисков                 */
/*****************************************************************
//Объявляем переменные
```

```
    var FSO,s,ss,Drives,D;
    //Создаем объект FileSystemObject
    FSO = WScript.CreateObject("Scripting.FileSystemObject");
    //Создаем коллекцию дисков, имеющихся в системе
    Drives = new Enumerator(FSO.Drives);
    s="";
    //Цикл по всем дискам в коллекции
    for (;!Drives.atEnd();Drives.moveNext()) {
     //Извлекаем текущий элемента коллекции
     D=Drives.item();
     //Получаем букву диска
     s+=D.DriveLetter;
     s+=" - ";
     if (D.DriveType == 3) //Проверяем, не является ли диск сетевым
      //Получаем имя сетевого ресурса
      ss=D.ShareName;
     else
      //Диск является локальным
      if (D.IsReady)  //Проверяем готовность диска
       //Если диск готов, то получаем метку тома для диска
       ss=D.VolumeName;
      else ss="Устройство не готово";
     s+=ss+"\n";
    }
    //Выводим полученные строки на экран
    WScript.Echo(s);
    /*************                                                Конец
*****************************************/
```

Объект *Folder*

Объект `Folder` обеспечивает доступ к свойствам каталога. Создать этот объект можно с помощью свойства `RootFolder` объекта `Drive` или методов `GetFolder`, `GetParentFolder` и `GetSpecialFolder` объекта `FileSystemObject` следующим образом:

```
    var FSO, Folder;
    FSO = WScript.CreateObject("Scripting.FileSystemObject");
    Folder = FSO.GetFolder("C:\\Мои документы");
```

Также объекты `Folder` могут быть получены как элементы коллекции `Folders`.

Свойства объекта `Folder` представлены в табл. 5.8.

Таблица 5.8. *Свойства объекта* `Folder`

Свойство	Описание

`Attributes`	Позволяет просмотреть или установить атрибуты каталога
`DateCreated`	Содержит дату и время создания каталога. Доступно только для чтения
`DateLastAccessed`	Содержит дату и время последнего доступа к каталогу. Доступно только для чтения
`DateLastModified`	Содержит дату и время последней модификации каталога. Доступно только для чтения
`Drive`	Содержит букву диска для устройства, на котором находится каталог. Доступно только для чтения
`Files`	Содержит коллекцию `Files`, состоящую из объектов `File` для всех файлов в каталоге (включая скрытые и системные)
`IsRootFolder`	Содержит `true`, если каталог является корневым, и `false` в противном случае
`Name`	Позволяет просмотреть и изменить имя каталога. Доступно для чтения и записи
`ParentFolder`	Содержит объект `Folder` для родительского каталога. Доступно только для чтения
`Path`	Содержит путь к каталогу
`ShortName`	Содержит короткое имя каталога (в формате 8.3)
`ShortPath`	Содержит путь к каталогу, состоящий из коротких имен каталогов (в формате 8.3)
`Size`	Содержит размер всех файлов и подкаталогов, входящих в данный каталог, в байтах
`SubFolders`	Содержит коллекцию `Folders`, состоящую из всех подкаталогов каталога (включая подкаталоги с атрибутами "Скрытый" и "Системный")
`Type`	Содержит информацию о типе каталога

Следующий пример показывает, как объект `Folder` используется для получения даты создания каталога (листинг 5.9).

Листинг 5.9. Вывод даты создания текущего каталога

```
    /*****************************************************************
    /*                                                              Имя:
DateFolder.js                                                  */
    /*                                                             Язык:
JScript                                                       */
    /*   Описание:   Вывод   на   экран   даты   создания   текущего
каталога        */
    /*****************************************************************
    var FSO,WshShell,s;   //Объявляем переменные
    //Создаем объект FileSystemObject
    FSO = WScript.CreateObject("Scripting.FileSystemObject");
    //Создаем объект WshShell
    WshShell=WScript.CreateObject("WScript.Shell");
    //Определяем каталог, из которого был запущен сценарий
    //(текущий каталог)
    Folder = FSO.GetFolder(WshShell.CurrentDirectory);
    //Получаем имя текущего каталога
    s="Текущий каталог: "+Folder.Name+"\n";
    //Получаем дату создания текущего каталога
    s+="Дата создания: "+Folder.DateCreated+"\n";
    //Выводим информацию на экран
    WScript.Echo(s);
    /*************                                              Конец
*********************************************/
```

Методы объекта `Folder` описаны в табл. 5.9.

Таблица 5.9. *Методы объекта* `Folder`

Метод	**Описание**
`Copy(`*`destination [, overwrite]`*`)`	Копирует каталог в другое место
`CreateTextFile(`*`filename[, overwrite [, unicode]]`*`)`	Создает новый текстовый файл с именем `filename` и возвращает указывающий на этот файл объект `TextStream` (этот метод аналогичен рассмотренному выше методу `CreateTextFile` объекта `FileSystemObject`)
`Delete(`*`[force]`*`)`	Удаляет каталог
Move(*destination*)	Перемещает каталог в другое место

Приведем необходимые замечания для методов из табл. 5.9.

Метод *Copy*

Обязательный параметр *destination* определяет каталог, в который будет производиться копирование; групповые символы в имени каталога недопустимы.

Параметр *overwrite* является логической переменной, определяющей, следует ли заменять уже существующий каталог с именем *destination* (*overwrite*=true) или нет (*overwrite=false*).

Замечание

Вместо метода Copy можно использовать метод CopyFolder объекта FileSystemObject.

Метод *Delete*

Если параметр *force* равен false или не указан, то с помощью метода Delete будет нельзя удалить каталог с атрибутом "только для чтения" (read-only). Установка для *force* значения true позволит сразу удалять такие каталоги.

При использовании метода Delete неважно, является ли заданный каталог пустым или нет — он удалится в любом случае.

Замечание

Вместо метода Delete можно использовать метод DeleteFolder объекта FileSystemObject.

Метод *Move*

Обязательный параметр *destination* определяет каталог, в который будет производиться перемещение; групповые символы в имени каталога недопустимы.

Замечание

Вместо метода Move можно использовать метод MoveFolder

объекта `FileSystemObject`.

Коллекция *Folders*

Коллекция `Folders` содержит объекты `Folder` для всех подкаталогов определенного каталога. Создается эта коллекция с помощью свойства `SubFolders` соответствующего объекта `Folder`. Например, в следующем примере переменная `SubFolders` является коллекцией, содержащей объекты `Folder` для всех подкаталогов каталога C:\Program Files:

```
var FSO, F, SubFolders;
//Создаем объект FileSystemObject
FSO=WScript.CreateObject("Scripting.FileSystemObject");
//Создаем объект Folder для каталога C:\Program Files
F=FSO.GetFolder("C:\\Program Files");
//Создаем коллекцию подкаталогов каталога C:\Program Files
SubFolders=F.SubFolders;
```

Коллекция `Folders` (как и `Drives`) имеет свойство `Count` и метод `Item`. Кроме этого, у `Folders` есть метод `Add(`*`folderName`*`)`, позволяющий создавать новые подкаталоги. В листинге 5.10 приведен сценарий MakeSubFold.js, который создает в каталоге "C:\Мои документы" подкаталог "Новая папка".

Листинг 5.10. Создание нового каталога

```
/*****************************************************************
/*                                                          Имя:
MakeSubFold.js                                              */
/*                                                         Язык:
JScript                                                    */
/*            Описание:            Создание            нового
каталога                          */
/*****************************************************************
//Объявляем переменные
var FSO, F, SubFolders;
//Создаем объект FileSystemObject
FSO=WScript.CreateObject("Scripting.FileSystemObject");
//Создаем объект Folder для каталога C:\Program Files
F=FSO.GetFolder("C:\\Program Files");
//Создаем коллекцию подкаталогов каталога C:\Program Files
SubFolders=F.SubFolders;
// Создаем каталог C:\Program Files\Новая папка
SubFolders.Add("Новая папка");
```

```
    /*************                                                    Конец
*******************************************/
```

Замечание

Напомним, что новый каталог также можно создать с помощью метода `CreateFolder` объекта `FileSystemObject`.

Для доступа ко всем элементам коллекции нужно использовать, как обычно, объект `Enumerator`. Например, в листинге 5.11 приведен сценарий ListSubFold.js, в котором на экран выводятся названия всех подкаталогов каталога C:\Program Files (рис. 5.5).

Рис. 5.5. Список всех подкаталогов каталога C:\Program Files

Листинг 5.11. Построение списка подкаталогов

```
    /*****************************************************************
    /*                                                              Имя:
ListSubFold.js                                                    */
    /*                                                              Язык:
```

```
JScript                                                    */
    /*  Описание:  Получение  списка  всех  подкаталогов  заданного
каталога */
    /************************************************************
    //Объявляем переменные
    var FSO,F,SFold,SubFolders,s;
    //Создаем объект FileSystemObject
    FSO=WScript.CreateObject("Scripting.FileSystemObject");
    //Путь к каталогу
    SFold="C:\\Program Files";
    s="Каталог "+SFold+"\n";
    s+="Подкаталоги:\n";
    //Создаем объект Folder для каталога C:\Program Files
    F=FSO.GetFolder(SFold);
    //Создаем коллекцию подкаталогов каталога C:\Program Files
    SubFolders= new Enumerator(F.SubFolders);
    //Цикл по всем подкаталогам
    for (; !SubFolders.atEnd(); SubFolders.moveNext()) {
      s+=SubFolders.item()+"\n";    //Добавляем  строку  с  именем
подкаталога
    }
    //Выводим полученные строки на экран
    WScript.Echo(s);
    /*************                                          Конец
*****************************************/
```

Объект *File*

Объект `File` обеспечивает доступ ко всем свойствам файла. Создать этот объект можно с помощью метода `GetFile` объекта `FileSystemObject` следующим образом:

```
var FSO, F;
//Создаем объект FileSystemObject
FSO=WScript.CreateObject("Scripting.FileSystemObject");
//Создаем объект File
F=FSO.GetFile("C:\\Мои документах\letter.txt");
```

Также объекты `File` могут быть получены как элементы коллекции `Files`. Свойства объекта `File` описаны в табл. 5.10.

Таблица 5.10. *Свойства объекта File*

Свойство	Описание
`Attributes`	Позволяет просмотреть или установить атрибуты файлов

`DateCreated`	Содержит дату и время создания файла. Доступно только для чтения
`DateLastAccessed`	Содержит дату и время последнего доступа к файлу. Доступно только для чтения
`DateLastModified`	Содержит дату и время последней модификации файла. Доступно только для чтения
`Drive`	Содержит букву диска для устройства, на котором находится файл. Доступно только для чтения
`Name`	Позволяет просмотреть и изменить имя файла. Доступно для чтения и записи
`ParentFolder`	Содержит объект `Folder` для родительского каталога файла. Доступно только для чтения
`Path`	Содержит путь к файлу
`ShortName`	Содержит короткое имя файла (в формате 8.3)
`ShortPath`	Содержит путь к файлу, состоящий из коротких имен каталогов (в формате 8.3)
`Size`	Содержит размер заданного файла в байтах
`Type`	Возвращает информацию о типе файла. Например, для файла с расширением txt возвратится строка "`Text Document`"

Методы объекта `File` представлены в табл. 5.11.

Таблица 5.11. *Методы объекта* `File`

Метод	**Описание**
`Copy(`*`destination [, overwrite]`*`)`	Копирует файл в другое место
`Delete(`*`[force]`*`)`	Удаляет файл
`Move(`*`destination`*`)`	Перемещает файл в другое место
`OpenAsTextStream(`*`[iomode, [format]]`*`)`	Открывает заданный файл и возвращает объект `TextStream`, который может быть использован для чтения, записи или добавления данных в текстовый файл

Приведем необходимые замечания для методов из табл. 5.11.

Метод *Copy*

Обязательный параметр *destination* определяет файл, в который будет производиться копирование; групповые символы в имени файла недопустимы.

Параметр *overwrite* является логической переменной, определяющей, следует ли заменять уже существующий файл с именем *destination* (*overwrite=true*) или нет (*overwrite=false*).

В листинге 5.12 приведен сценарий CopyFile.js, иллюстрирующий использование метода Copy. В этом сценарии на диске C: создается файл TestFile.txt, который затем копируется на рабочий стол.

Листинг 5.12. Создание текстового файла и копирование его в другой каталог

```
    /*****************************************************************
    /*                                                          Имя:
CopyFile.js                                                  */
    /*                                                         Язык:
JScript                                                     */
    /*        Описание:        Создание        и        копирование
файла                          */
    /*****************************************************************
    //Объявляем переменные
    var FSO,F,WshShell,WshFldrs,PathCopy;
    //Создаем объект FileSystemObject
    FSO=WScript.CreateObject("Scripting.FileSystemObject");
    //Создаем файл
    F=FSO.CreateTextFile("C:\\TestFile.txt", true);
    //Записываем в файл строку
    F.WriteLine("Тестовый файл");
    //Закрываем файл
    F.Close();
    //Создаем объект WshShell
    WshShell=WScript.CreateObject("Wscript.Shell");
    //Создаем объект WshSpecialFolders
    WshFldrs=WshShell.SpecialFolders;
    //Определяем путь к рабочему столу
    PathCopy=WshFldrs.item("Desktop")+"\\";
    //Создаем объект File для файла C:\TestFile.txt
    F=FSO.GetFile("C:\\TestFile.txt");
    //Копируем файл на рабочий стол
```

```
    F.Copy(PathCopy);
    /*************                                          Конец
*******************************************/
```

Замечание

Вместо метода `Copy` можно использовать метод `CopyFile` объекта `FileSystemObject`.

Метод *Delete*

Если параметр *`force`* равен `false` или не указан, то с помощью метода `Delete` будет нельзя удалить файл с атрибутом "только для чтения" (read-only). Установка для *`force`* значения `true` позволит сразу удалять такие файлы.

Замечание

Вместо метода `Delete` можно использовать метод `DeleteFile` объекта `FileSystemObject`.

Метод *Move*

Обязательный параметр *`destination`* определяет файл, в который будет производиться перемещение; групповые символы в имени файла недопустимы.

Замечание

Вместо метода `Move` можно использовать метод `MoveFile` объекта `FileSystemObject`.

Метод *OpenAsTextStream*

Числовой параметр *`iomode`* задает режим ввода/вывода для

открываемого файла и может принимать те же значения, что и одноименный параметр в методе `OpenTextFile` объекта `FileSystemObject` (табл. 5.5).

Числовой параметр *`format`* определяет формат открываемого файла (ASCII или Unicode). Этот параметр также может принимать те же значения, что и *`format`* в методе `OpenTextFile` объекта `FileSystemObject` (табл. 5.6).

Замечание

Открыть текстовый файл для чтения можно также с помощью метода `OpenTextFile` объекта `FileSystemObject`.

В листинге 5.13 приведен сценарий WriteTextFile.js, иллюстрирующий использование метода **OpenAsTextStream** для записи строки в файл и чтения из него.

Листинг 5.13. Запись информации в текстовый файл и чтение ее из него

```
    /*****************************************************************
    /*                                                          Имя:
WriteTextFile.js                                              */
    /*                                                         Язык:
JScript                                                     */
    /*  Описание:  Запись  строк  в  текстовый  файл  и  чтение  из
него        */
    /*****************************************************************
    var FSO,F,TextStream,s;  //Объявляем переменные
    //Инициализируем константы
    var ForReading = 1, ForWriting = 2, TristateUseDefault = -2;
    //Создаем объект FileSystemObject
    FSO=WScript.CreateObject("Scripting.FileSystemObject");
    //Создаем в текущем каталоге файл test1.txt
    FSO.CreateTextFile("test1.txt");
    //Создаем объект File для файла test1.txt
    F=FSO.GetFile("test1.txt");
    //Создаем объект TextStream (файл открывается для записи)
    TextStream=F.OpenAsTextStream(ForWriting, TristateUseDefault);
    //Записываем в файл строку
```

```
    TextStream.WriteLine("Это первая строка");
    //Закрываем файл
    TextStream.Close();
    //Открываем файл для чтения
    TextStream=F.OpenAsTextStream(ForReading, TristateUseDefault);
    //Считываем строку из файла
    s=TextStream.ReadLine();
    //Закрываем файл
    TextStream.Close();
    //Отображаем строку на экране
    WScript.Echo("Первая строка из файла test1.txt:\n\n",s);
    /*************                                                Конец
*****************************************/
```

Коллекция *Files*

Коллекция `Files` содержит объекты `File` для всех файлов, находящихся внутри определенного каталога. Создается эта коллекция с помощью свойства `Files` соответствующего объекта `Folder`. Например, в следующем примере переменная `Files` является коллекцией, содержащей объекты `File` для всех файлов в каталоге C:\Мои документы:

```
var FSO, F, Files;
FSO=WScript.CreateObject("Scripting.FileSystemObject");
F=FSO.GetFolder("C:\\Мои документы");
Files=F.Files;
```

Как и рассмотренные выше коллекции `Drives` и `Folders`, коллекция `Files` имеет свойство `Count` и метод `Item`.

Для доступа в цикле ко всем элементам коллекции `Files` применяется объект `Enumerator`. В качестве примера использования этого объекта в листинге 5.14 приведен сценарий ListFiles.js, выводящий на экран названия всех файлов, которые содержатся в специальной папке "Мои документы" (рис. 5.6).

Рис. 5.6. Список всех файлов в специальной папке "Мои документы"

Листинг 5.14. Построение списка файлов

```
    /*****************************************************************
    /*                                                              Имя:
ListFiles.js                                                    */
    /*                                                             Язык:
JScript                                                        */
    /*   Описание:   Получение   списка   всех   файлов   заданного
каталога       */
    /*****************************************************************
    //Объявляем переменные
    var FSO,F,Files,WshShell,PathList,s;
    //Создаем объект FileSystemObject
    FSO=WScript.CreateObject("Scripting.FileSystemObject");
    //Создаем объект WshShell
    WshShell=WScript.CreateObject("Wscript.Shell");
    //Создаем объект WshSpecialFolders
    WshFldrs=WshShell.SpecialFolders;
    //Определяем путь к папке "Мои документы"
    PathList=WshFldrs.item("MyDocuments")+"\\";
    //Создаем объект Folder для папки "Мои документы"
    F=FSO.GetFolder(PathList);
    //Создаем коллекцию файлов каталога "Мои документы"
    Files=new Enumerator(F.Files);
    s = "Файлы из каталога "+PathList+"\n";
    //Цикл по всем файлам
    for (; !Files.atEnd(); Files.moveNext())
     //Добавляем строку с именем файла
     s+=Files.item().Name+"\n";
    //Выводим полученные строки на экран
    WScript.Echo(s);
    /*************                                                Конец
*******************************************/
```

Объект *TextStream*

Объект `TextStream` обеспечивает последовательный (строка за строкой) доступ к текстовому файлу. Методы этого объекта позволяют читать информацию из файла и записывать ее в него.

Создать объект `TextStream` можно с помощью следующих методов:

- `CreateTextFile` объектов `FileSystemObject` и `Folder`;
- `OpenTextFile` объекта `FileSystemObject`;
- `OpenAsTextStream` объекта `File`.

В следующем примере переменная `F` является объектом TextStream и используется для записи строки текста в файл C:\TestFile.txt:

```
    //Создаем объект FileSystemObject
```

```
var FSOWScript.CreateObject("Scripting. FileSystemObject");
//Создаем текстовый файл
var F=FSO.CreateTextFile("C:\\TestFile.txt", true);
//Записываем строку в файл
F.WriteLine("Строка текста");
//Закрываем файл
F.Close();
```

Свойству объекта TextStream описаны в табл. 5.12.

Таблица 5.12. *Свойства объекта* `TextStream`

Свойство	Описание
`AtEndOfLine`	Содержит `true`, если указатель достиг конца строки в файле, и `false` в противном случае. Доступно только для чтения
`AtEndOfStream`	Содержит `true`, если указатель достиг конца файла, и `false` в противном случае. Доступно только для чтения
`Column`	Содержит номер колонки текущего символа в текстовом файле. Доступно только для чтения
`Line`	Содержит номер текущей строки в текстовом файле. Доступно только для чтения

Методы объекта `TextStream` представлены в табл. 5.13.

Таблица 5.13. *Методы объекта* `TextStream`

Метод	Описание
`Close()`	Закрывает открытый файл
`Read(`*`n`*`)`	Считывает из файла *n* символов и возвращает полученную строку
`ReadAll()`	Считывает полностью весь файл и возвращает полученную строку
`ReadLine()`	Возвращает полностью считанную из файла строку
`Skip(`*`n`*`)`	Пропускает при чтении n символов
`SkipLine()`	Пропускает целую строку при чтении
`Write(`*`string`*`)`	Записывает в файл строку *string* (без символа конца строки)
`WriteBlankLines(`*`n`*`)`	Записывает в файл *n* пустых строк (символы перевода строки и возврата каретки)

`WriteLine(`*`[string]`*`)`	Записывает в файл строку *`string`* (вместе с символом конца строки). Если параметр `string` опущен, то в файл записывается пустая строка

В листинге 5.15 приведен сценарий TextFile.js, иллюстрирующий использование методов объекта `TextStream`. В этом сценарии на диске C: создается файл TestFile.txt и в него записываются три строки, вторая из которых является пустой. После этого файл открывается для чтения и из него считывается третья строка, которая выводится на экран (рис. 5.7).

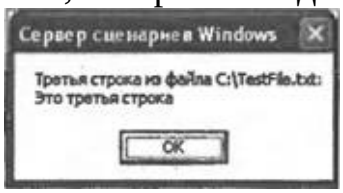


Рис. 5.7. Результат работы сценария TextFile.js

Листинг 5.15. Работа с текстовым файлом с помощью методов объекта TextStream

```
    /*****************************************************************
    /*                                                          Имя:
TextFile.js                                                  */
    /*                                                         Язык:
JScript                                                     */
    /* Описание: Работа с текстовым файлом (запись и чтение
информации)*/
    /*****************************************************************
    var FSO,F,s;  //Объявляем переменные
    var ForReading = 1; //Инициализируем константы
    //Создаем объект FileSystemObject
    FSO=WScript.CreateObject("Scripting.FileSystemObject");
    //Создаем на диске C: текстовый файл TestFile.txt
    F=FSO.CreateTextFile("C:\\TestFile.txt", true);
    //Записываем в файл первую строку
    F.Write("Это ");
    F.WriteLine("первая строка");
    //Записываем в файл пустую строку
    F.WriteBlankLines(1);
    //Записываем в файл третью строку
    F.WriteLine("Это третья строка");
    //Закрываем файл
```

```
    F.Close();
    //Открываем файл для чтения
    F=FSO.OpenTextFile("C:\\TestFile.txt", ForReading);
    //Пропускаем в файле две первые строки
    F.SkipLine();
    F.SkipLine();
    s="Третья строка из файла C:\\TestFile.txt:\n";
    //Считываем из файла третью строку
    s+=F.ReadLine();
    //Выводим информацию на экран
    WScript.Echo(s);
    /*************                                                    Конец
*********************************************/
```

Примеры сценариев

Ниже подробно рассмотрены несколько завершенных сценариев, иллюстрирующих работу с файловой системой компьютера.

Отчет об использовании дискового пространства

Напишем сценарий DrivesReport.js, который будет создавать таблицу использования дискового пространства для дисков всех типов (съемных, жестких и сетевых), имеющихся на компьютере, в следующем формате:

Диск: *буква_диска*

Метка тома: *метка* **Общий объем,** Mb: *n1*

Используется, Mb: *n2* **Свободно, Mb:** *n3*

Для этого в сценарии вывод информации производится в текстовый файл rep.txt (переменная `RepFile`), который затем открывается с помощью Блокнота (рис. 5.8):

```
    //Создаем объект WshShell
    WshShell=WScript.CreateObject("WScript.Shell");
    //Запускаем Блокнот (notepad.exe) и открываем в нем файл rep.txt
    WshShell.Run("notepad.exe rep.txt");
```

Данные об одном диске формируются в функции `WriteDriveInfo(`*`drive`*`)`, где в качестве параметра *`drive`* подставляется объект Drive для нужного диска. Соответствующие объекты `Drive` для всех дисков, имеющихся в системе, создаются в функции `LoopDrives()`:

```
    // Функция для перебора в цикле всех устройств (дисков)
    function LoopDrives() {
     var Drives;
     //Создаем коллекцию дисков
     Drives = new Enumerator( FSO.Drives );
     //Цикл по всем дискам
            for(;        !Drives.atEnd();        Drives.moveNext())
WriteDriveInfo(Drives.item());
    }
```

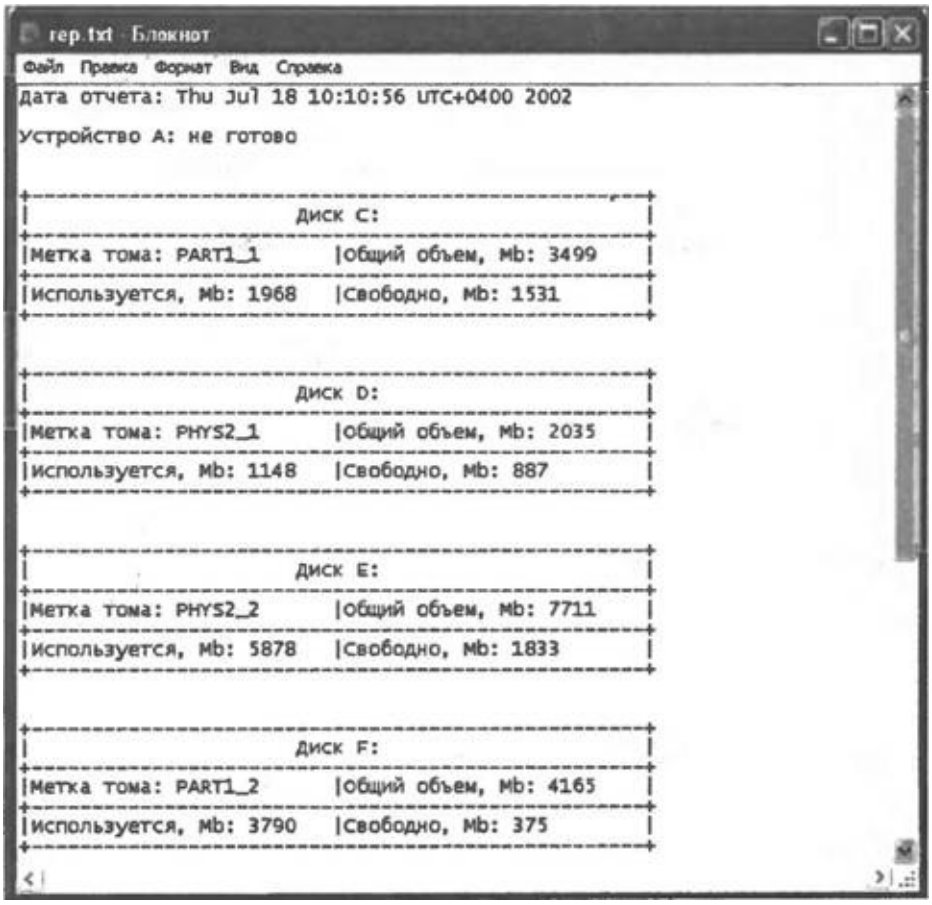


Рис. 5.8. Сформированный отчет об использовании дискового пространства

В функции `WriteDriveInfo(`*`drive`*`)` сначала проверяется готовность устройства `drive` — если свойство `IsReady` объекта `Drive` равно `true`, то для этого устройства определяются общий объем (свойство `TotalSize`), объем свободного пространства (свойство `FreeSpace`), буква диска (свойство `DriveLetter`) и метка тома (свойство `FreeSpace`):

```
//Вычисляем общий объем диска в мегабайтах
Total = Math.round(drive.TotalSize/1048576);
//Вычисляем объем свободного пространства в мегабайтах
Free = Math.round(drive.FreeSpace/1048576);
//Вычисляем объем использованного пространства в мегабайтах
Used = Total - Free;
//Определяем букву диска
DriveLetter=drive.DriveLetter+":";
//Определяем метку тома
VolumeName=drive.VolumeName;
```

Строки с полученными значениями затем приводятся к нужному виду с помощью вспомогательных функций `LFillStr` (выравнивание строки

влево), FillStr (выравнивание строки по центру) и записываются в выходной файл RepFile:

```
    RepFile.WriteLine("+----------------------------------------
--------+");
    //Записываем информацию о букве диска
    s="|"+FillStr(51,"Диск "+DriveLetter)+"|";
    RepFile.WriteLine(s);
    RepFile.WriteLine("+----------------------------------------
--------+");
    //Записываем информацию о метке тома
    s="|"+LFillStr(25,"Метка тома: "+VolumeName)+"|";
    //Записываем информацию об общем объеме диска
    s+=LFillStr(25,"Общий объем, Mb: "+Total)+"|";
    RepFile.WriteLine(s);
    RepFile.WriteLine("+----------------------------------------
--------+");
    //Записываем информацию об использованном пространстве
    s="|"+LFillStr(25,"Используется, Mb: "+Used.toString())+"|";
    //Записываем информацию о свободном пространстве
    s+=LFillStr(25,"Свободно, Mb: "+Free.toString())+"|";
    RepFile.WriteLine(s);
    RepFile.WriteLine("+----------------------------------------
--------+");
```

Если же устройство *drive* не готово (свойство IsReady равно false), то с помощью свойства DriveType проверяется, не является ли ли диск сетевым (в этом случае DriveType=3), после чего в файл выводится соответствующее сообщение:

```
    if (drive.DriveType == 3) { //Проверяем, является ли диск
сетевым
     //Записываем информацию о букве диска
      RepFile.WriteLine( "Диск " + drive.DriveLetter + " является
сетевым" );
     //Записываем пустые строки
     RepFile.WriteLine();
     RepFile.WriteLine();
    } else {
     //Устройство не готово
      RepFile.WriteLine( "Устройство " + drive.DriveLetter + ": не
готово" );
     //Записываем пустые строки
     RepFile.WriteLine();
     RepFile.WriteLine();
    }
```

Полный текст сценария DrivesReport.js приведен в листинге 5.16.

Листинг 5.16. Отчет об использовании дискового пространства для всех устройств на компьютере

```
    /*****************************************************************
    /*                                                              Имя:
DrivesReport.js                                                 */
    /*                                                             Язык:
JScript                                                        */
    /*    Описание:    Формирование    отчета    об    использовании
дискового        */
    /*                        пространства  для  всех  устройств  на
компьютере         */
    /*****************************************************************
    //Объявляем переменные
    var FSO, RepFile, MDate,WshShell, ForWriting = 2;
    // Функция для перебора в цикле всех устройств (дисков)
    function LoopDrives() {
     var Drives;
     //Создаем коллекцию дисков
     Drives = new Enumerator( FSO.Drives );
     //Цикл по всем дискам
            for(;        !Drives.atEnd();        Drives.moveNext())
WriteDriveInfo(Drives.item());
    }
    // Функция для вывода информации об одном устройстве (диске)
    function WriteDriveInfo(drive) {
     var s,Total,Free,Used,DriveLetter,VolumeName;
     if (drive.IsReady) { //Проверяем готовность устройства
      //Вычисляем общий объем диска в мегабайтах
      Total = Math.round(drive.TotalSize/1048576);
      //Вычисляем объем свободного пространства в мегабайтах
      Free = Math.round(drive.FreeSpace/1048576);
      //Вычисляем объем использованного пространства в мегабайтах
      Used = Total - Free;
      //Определяем букву диска
      DriveLetter=drive.DriveLetter+":";
      //Определяем метку тома
      VolumeName=drive.VolumeName;
       RepFile.WriteLine("+-----------------------------------------
----------+");
      //Записываем информацию о букве диска
      s="|"+FillStr(51,"Диск "+DriveLetter)+"|";
      RepFile.WriteLine(s);
       RepFile.WriteLine("+-----------------------------------------
----------+");
      //Записываем информацию о метке тома
```

```
      s="|"+LFillStr(25,"Метка тома: "+VolumeName)+"|";
      //Записываем информацию об общем объеме диска
      s+=LFillStr(25,"Общий объем, Mb: "+Total)+"|";
      RepFile.WriteLine(s);
       RepFile.WriteLine("+----------------------------------------
----------+");
      //Записываем информацию об использованном пространстве
      s="|"+LFillStr(25,"Используется, Mb: "+Used.toString())+"|";
      //Записываем информацию о свободном пространстве
      s+=LFillStr(25,"Свободно, Mb: "+Free.toString())+"|";
      RepFile.WriteLine(s);
       RepFile.WriteLine("+----------------------------------------
----------+");
      //Записываем пустые строки
      RepFile.WriteLine();
      RepFile.WriteLine();
     } else if (drive.DriveType == 3) { //Проверяем, является ли
диск сетевым
      //Записываем информацию о букве диска
       RepFile.WriteLine( "Диск " + drive.DriveLetter + " является
сетевым" );
      //Записываем пустые строки
      RepFile.WriteLine();
      RepFile.WriteLine();
     } else {
      //Устройство не готово
       RepFile.WriteLine( "Устройство " + drive.DriveLetter + ": не
готово" );
      //Записываем пустые строки
      RepFile.WriteLine();
      RepFile.WriteLine();
     }
    }
    /*******************                                    Начало
*********************************/
    //Создаем объект FileSystemObject
    FSO = WScript.CreateObject("Scripting.FileSystemObject");
    //Открываем файл rep.txt
    RepFile = FSO.OpenTextFile("rep.txt", ForWriting, true);
    //Определяем текущую дату и время
    MDate = new Date();
    //Записываем дату и время в отчет
    RepFile.WriteLine("Дата отчета: " + MDate);
    RepFile.WriteLine();
    //Вызываем функцию LoopDrives
    LoopDrives();
    //Закрываем файл rep.txt
```

```
    RepFile.Close();
    //Создаем объект WshShell
    WshShell=WScript.CreateObject("WScript.Shell");
    //Запускаем Блокнот (notepad.exe) и открываем в нем файл
rep.txt
    WshShell.Run("notepad.exe rep.txt");
    /*************                                          Конец
*****************************************/
    // Вспомогательные функции
    //Выравнивание строки s вправо в поле длиной l символов
    function RFillStr(l,s) {
     var ss,i,ll;
     ll=l-s.length;
     if (s.length>=l) {
      return(s);
     } else {
      ss=s;
      for (i=1;i<=ll;i++) {
       ss=" "+ss;
      }
      return(ss);
     }
    }
    //Выравнивание строки s влево в поле длиной l символов
    function LFillStr(l,s) {
     var ss,i,ll;
     ll=l-s.length;
     if (s.length>=l) {
      return(s);
     } else {
      ss=s;
      for (i=1;i<=ll;i++) {
       ss=ss+" ";
      }
      return(ss);
     }
    }
    //Выравнивание строки s по центру в поле длиной l символов
    function FillStr(l,s) {
     var ss,i,ll,s1,l2;
     ll=l-s.length;
     if (s.length>=l) {
      return(s);
     } else {
      ss=s;
      l2=Math.round((l-s.length)/2);
      ss=LFillStr(s.length+l2,s);
```

```
  ss=RFillStr(l,ss);
  return(ss);
 }
}
```

Удаление ненужных временных файлов с жесткого диска

В результате некорректного завершения приложений на жестком диске часто остаются "бесхозные" временные файлы (с расширением tmp), которые затем приходится удалять вручную. Весьма удобно было бы удалять все такие файлы сразу со всего жесткого диска, не заботясь о том, в каких каталогах они находятся. Используя стандартные средства операционной системы, можно сначала с помощью пункта **Найти** (Find) меню **Пуск** (Start) составить список всех временных файлов (рис. 5.9), затем выделить все файлы в этом списке и нажать комбинацию клавиш <Shift>+<Delete> (удаление файлов без помещения в Корзину).

Однако если хотя бы один из временных файлов будет занят каким-либо приложением, то в доступе к нему будет отказано и процесс удаления прервется (рис. 5.10).

Поэтому для удаления с жесткого диска всех временных файлов мы напишем сценарий DelTmp.js (основная идея, которая используется в данном сценарии, позаимствована из статьи [6]).

Алгоритм работы сценария состоит в следующем:

□ в Блокноте (notepad.exe) создается новый файл для отображения в нем результатов работы;

□ на диске D: просматриваются подкаталоги всех уровней вложенности;

□ в каждом подкаталоге ищутся все временные файлы с расширением tmp;

□ для каждого найденного временного файла производится попытка удаления. В случае успеха в Блокноте печатается путь к файлу и слово "OK", если же удаляемый файл занят, то печатается путь к файлу и слово "Busy" ("Занят");

□ после просмотра всех каталогов в Блокноте печатается общее количество найденных временных файлов (рис. 5.11).

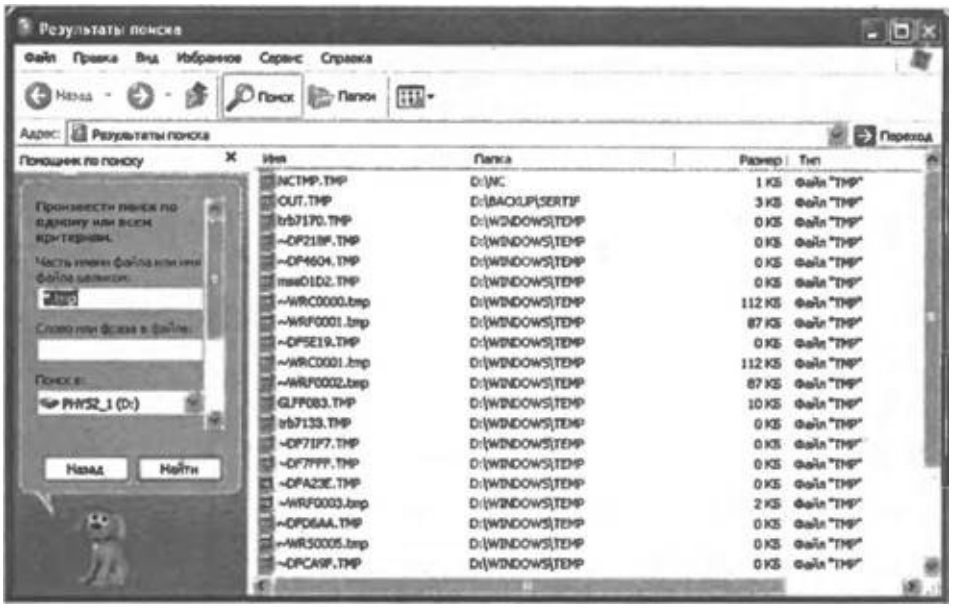


Рис. 5.9. Список всех временных файлов на диске D:

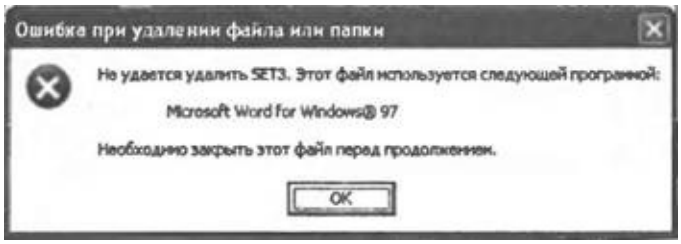


Рис. 5.10. Ошибка, возникающая при попытке удалить занятый файл

Итак, наиболее важными в сценарии являются: функция `DelFilesInFolder(`*`Fold, SExt`*`)`, в которой происходит попытка удаления в каталоге Fold всех файлов, имеющих расширение *`SExt`*, и *рекурсивная* (т.е. содержащая обращение к самой себе) функция `LoopSubFolders(`*`Fold`*`)`, в которой происходит вызов функции `DelFilesInFolder` для подкаталогов каталога Fold.

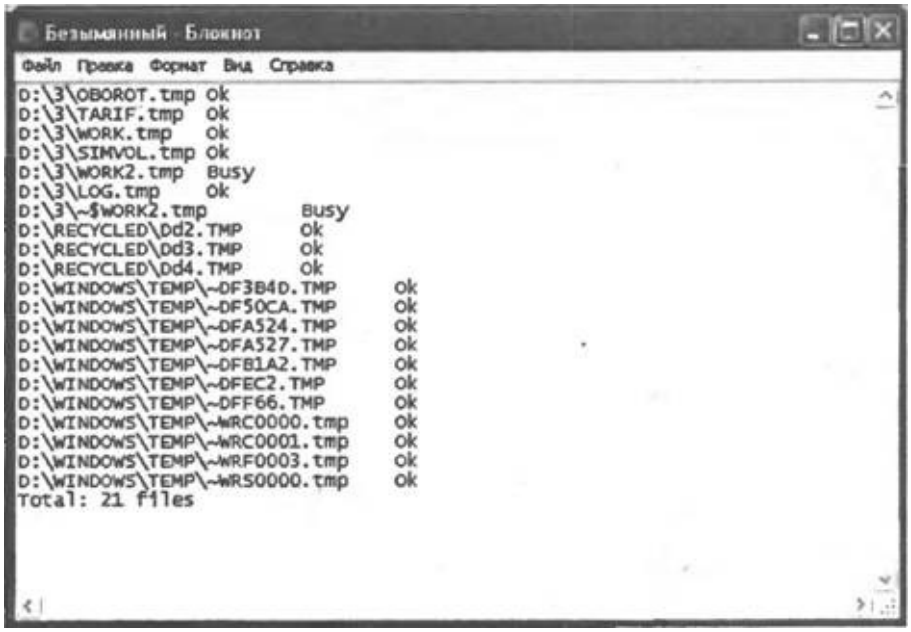


Рис. 5.11. Результат работы сценария DelTmp.js

Рассмотрим сначала функцию `DelFilesInFolder(`*`Fold, SExt`*`)`. Здесь сначала создается объект `Enumerator` с именем `Files` для доступа к коллекции всех файлов в каталоге Fold:

```
var Files, s, SPath, FileExt, Status;
Files=new Enumerator(Fold.Files);
```

после чего все элементы коллекции просматриваются в цикле while:

```
s="";
//Цикл по всем файлам
while (!Files.atEnd()) {
 …
 Files.moveNext(); //Переходим к следующему файлу
}
```

Для текущего файла в коллекции выделяется его расширение, которое преобразуется к верхнему регистру:

```
//Определяем путь к файлу
SPath=Files.item().Path;
//Выделяем расширение файла
FileExt=FSO.GetExtensionName(SPath).toUpperCase();
```

В случае совпадения расширения `FileExt` с расширением `SExt` в блоке `try…catch` производится попытка удаления текущего файла:

```
ColTmp++; //Увеличиваем счетчик файлов для удаления
try {
 Status="Ok";
 //Пытаемся удалить файл
 Files.item().Delete();
} catch (e) {
```

```
 //Обрабатываем возможные ошибки
 if (e != 0) {
  //Произошла ошибка при удалении файла
  Status="Busy";
 }
}
```

Таким образом, в переменной `Status` будет записан результат удаления файла: "OK" в случае успеха, и "Busy" в том случае, если удаляемый файл занят другим приложением. После этого полный путь к удаляемому файлу и значение переменной `Status` печатаются в окне Блокнота с помощью метода `SendKeys` объекта `WshShell`. Здесь только нужно учесть, что в имени файла могут встретиться символы, имеющие специальный смысл для метода `SendKeys`, поэтому предварительно нужно соответствующим образом заменить такие символы в имени файла:

```
//Заменяем специальные символы в названии файла
SPath=SPath.replace("~", "{-}");
SPath=SPath.replace("+", "{+}");
SPath=SPath.replace("^", "{^}");
SPath=SPath.replace("%", "{%}");
//Посылаем название файла и результат его удаления в Блокнот
WshShell.SendKeys(SPath+"{TAB}"+Status+"~");
//Приостанавливаем сценарий на 0,5 сек
WScript.Sleep(500);
```

Перейдем теперь к описанию функции `LoopSubFolders(`*`Fold`*`)`. Сначала в этой функции удаляются временные файлы в каталоге Fold:

```
//Удаляем временные файлы из каталога Fold
DelFilesInFolder(Fold, ExtForDel);
```

Затем происходит удаление файлов во всех подкаталогах каталога Fold, причем делается это с помощью обращения для каждого подкаталога к той же функции `LoopSubFolders`:

```
//Создаем коллекцию подкаталогов
SubFolders = new Enumerator(Fold.SubFolders);
//Цикл по всем подкаталогам
while (!SubFolders.atEnd()) {
 //Вызываем рекурсивно функцию LoopSubFolders
 LoopSubFolders(SubFolders.item());
 //Переходим к следующему подкаталогу
 SubFolders.moveNext();
}
```

Полный текст сценария DelTmp.js приведен в листинге 5.17.

Листинг 5.17. Удаление всех временных файлов на диске D:

```
    /*****************************************************************
    /*                                                           Имя:
DelTmp.js                                                    */
    /*                                                          Язык:
JScript                                                    */
    /*    Описание:    Удаление    временных    файлов    во    всех
подкаталогах        */
    /*                                                       текущего
каталога                                          */
    /*****************************************************************
    //Объявляем глобальные переменные
    var WshShell,FSO,Folder,
     ColTmp=0,        //Счетчик файлов для удаления
     ExtForDel="tmp"; //Расширение файлов, подлежащих удалению

    //Рекурсивная функция для удаления временных файлов в каталоге
Fold
    function LoopSubFolders(Fold) {
     var SubFolders;
     //Удаляем временные файлы из каталога Fold
     DelFilesInFolder(Fold,ExtForDel);
     //Создаем коллекцию подкаталогов
     SubFolders = new Enumerator(Fold.SubFolders);
     //Цикл по всем подкаталогам
     while (!SubFolders.atEnd()) {
      //Вызываем рекурсивно функцию LoopSubFolders
      LoopSubFolders(SubFolders.item());
      //Переходим к следующему подкаталогу
      SubFolders.moveNext();
     }
    }
    //Функция для удаления файлов с расширением SExt в каталоге
Fold
    function DelFilesInFolder(Fold,SExt) {
     //Объявляем переменные
     var Files,s,SPath,FileExt,Status;
     //Создаем коллекцию всех файлов в каталоге Fold
     Files=new Enumerator(Fold.Files);
     s="";
     //Цикл по всем файлам
     while (!Files.atEnd()) {
      //Определяем путь к файлу
      SPath=Files.item().Path;
      //Выделяем расширение файла
      FileExt=FSO.GetExtensionName(SPath).toUpperCase();
      //Сравниваем расширение файла с расширением SExt
      if (FileExt==SExt.toUpperCase()) {
```

```
       ColTmp++; //Увеличиваем счетчик файлов для удаления
       try {
        Status="Ok";
        //Пытаемся удалить файл
        //Files.item().Delete();
       } catch (e) { //Обрабатываем возможные ошибки
        if (e != 0) {
         //Произошла ошибка при удалении файла
         Status="Busy";
        }
       }
       //Заменяем специальные символы в названии файла
       SPath=SPath.replace("~","{~}");
       SPath=SPath.replace("+","{+}");
       SPath=SPath.replace("^","{^}");
       SPath=SPath.replace("%","{%}");
       //Посылаем название файла и результат его удаления
       //в Блокнот
       WshShell.SendKeys(SPath+"{TAB}"+Status+"~");
       //Приостанавливаем сценарий на 0,5 сек
       WScript.Sleep(500);
      }
      Files.moveNext();  //Переходим к следующему файлу
     }
    }
    /*******************                                    Начало
*********************************/
    //Создаем объект WshShell
    WshShell=WScript.CreateObject("WScript.Shell");
    //Запускаем Блокнот
    theNotepad = WshShell.Exec("Notepad");
    //Приостанавливаем сценарий на 0,5 сек
    WScript.Sleep(500);
    //Активизируем окно Блокнота
    WshShell.AppActivate(theNotepad.ProcessID);
    //Создаем объект FileSystemObject
    FSO=WScript.CreateObject("Scripting.FileSystemObject");
    //Определяем каталог, с которого будет начато удаление файлов
    Folder = FSO.GetFolder("D:\\");
    //Вызываем функцию LoopSubFolder для каталога Folder
    LoopSubFolders(Folder);
    //Формируем информацию о количестве обработанных файлов
    SItog="Total: "+ColTmp+ " file(s)";
    //Посылаем в Блокнот итоговую информацию
    WshShell.SendKeys(SItog);
    /*************                                          Конец
*****************************************/
```

Поиск файлов с использованием регулярных выражений

Всем хорошо известно, что для поиска файлов и папок с помощью стандартных средств Windows в именах можно использовать подстановочные символы "?" (обозначает любой один символ) и "*" (обозначает любое число любых символов). Например, на рис. 5.12 представлен результат поиска файлов *.sys (все файлы с расширением sys) на диске C:.

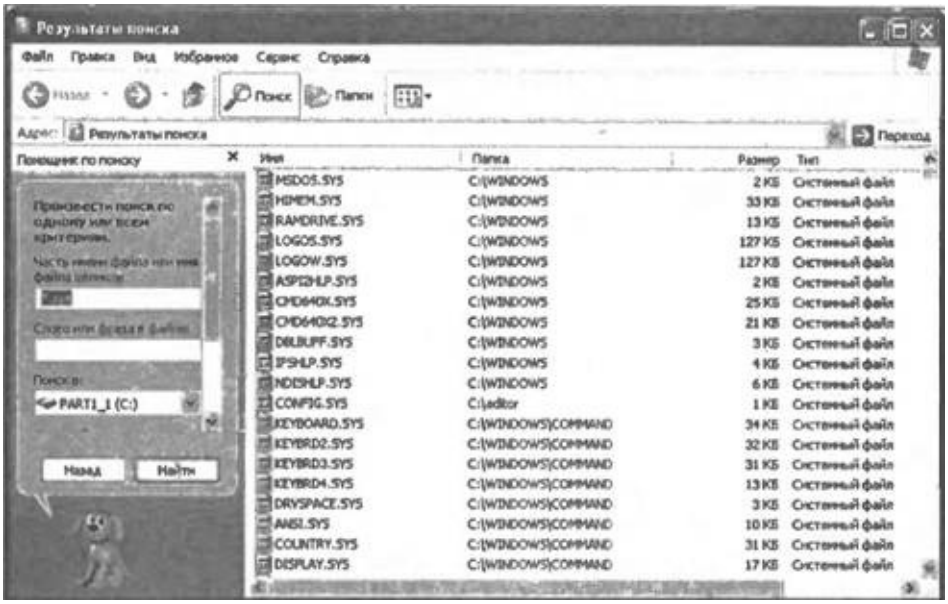


Рис. 5.12. Использование подстановочных символов при поиске файлов

В сценариях WSH можно производить поиск файлов (как и любого другого текста) с помощью гораздо более сложных правил для определения соответствий. Для этого используются *регулярные выражения*, которые определяют образец текста для поиска. Для задания этого образца используются *литералы* и *метасимволы.* Каждый символ, который не имеет специального значения в регулярных выражениях, рассматривается как литерал и должен точно совпасть при поиске. Метасимволы — это символы со специальным значением в регулярных выражениях. Описание наиболее часто используемых метасимволов приведено в табл. 5.14.

Таблица 5.14. *Некоторые метасимволы, использующиеся в регулярных выражениях*

Символы Описание

Следующий символ будет являться специальным символом или,

`\`	наоборот, литералом. Например, `n` означает символ "n", а "`\n`" означает символ новой строки. Последовательности `\\` соответствует символ "\", а `\(` — символ "("
`^`	Начало строки
`$`	Конец строки
`*`	Предыдущий символ повторяется любое число раз (в том числе ни разу). Например, выражению `zo*` соответствуют как "z", так и "zoo"
`+`	Предыдущий символ повторяется не менее одного раза. Например, выражению `zo+` соответствует "zoo", но не "z"
`?`	Предыдущий символ повторяется не более одного раза
`.` (точка)	Любой символ, кроме перевода строки
`x|y`	Либо символ "x", либо символ "y". Например, выражению `z|food` соответствуют "z" или "food"
`[xyz]`	Множество символов. Означает любой один символ из набора символов, заключенных в квадратные скобки. Например, выражению `[abc]` соответствует символ "a" в слове "plain"
`[a-z]`	Диапазон символов. Означает любой один символ из заданного диапазона. Например, выражению `[a-z]` соответствует любая буква английского алфавита в нижнем регистре
`[^m-z]`	Означает любой символ, не входящий в заданный диапазон. Например, выражению `[^m-z]` соответствует любой символ, не попадающий в диапазон символов от "m" до "z"
`\b`	Граница слова, т.е. позиция между словом и пробелом. Например, выражению `er\b` соответствует символ "er" в слове "never", но не в слове "verb"
`\B`	Позиция внутри слова (не на границе). Например, выражению `ea*r\B` соответствует подстрока "ear" в "never early"
`\d`	Символ, обозначающий цифру. Эквивалентно `[0-9]`
`\D`	Любой символ, кроме цифры. Эквивалентно `[^0-9]`

Метасимволы можно употреблять совместно, например, комбинация "`.*`" означает любое число любых символов.

Замечание

Более подробную информацию о регулярных выражениях можно найти, например, в документации Microsoft по языку VBScript.

В качестве примера использования регулярных выражений в листинге 5.18 приведен сценарий FindRegExp.js, в котором производится поиск в подкаталоге ForFind текущего каталога всех файлов, имена которых начинаются с символов "П", "А" или "И" и имеют расширение txt.

Для получения доступа к каталогу ForFind в сценарии используется метод `GetFolder` объекта `FileSystemObject`:

```
//Создаем объект WshShell
WshShell=WScript.CreateObject("WScript.Shell");
//Создаем объект FileSystemObject
FSO=WScript.CreateObject("Scripting.FileSystemObject");
//Создаем объект Folder для доступа к подкаталогу ForFind
//текущего каталога
Folder = FSO.GetFolder(WshShell.CurrentDirectory+"\\ForFind");
```

Поиск нужных файлов будет выполняться с помощью следующего регулярного выражения:

```
//Создаем регулярное выражение (объект RegExp)
RegEx=new RegExp("^[ПАИ].*\.txt$", "i");
```

Сам поиск и вывод имен найденный файлов производятся в функции `FindFilesInFolder(`*`Fold, RegEx`*`)`. Здесь сначала инициализируются счетчик найденных файлов и переменная, в которой будут сохраняться имена найденных файлов, а также создается объект `Enumerator` (переменная `Files`) для доступа к файлам каталога *`Fold`*:

```
ColFind=0; //Счетчик найденных файлов
SFileNames=""; //Строка с именами файлов
//Создаем коллекцию файлов в каталоге Fold
Files=new Enumerator(Fold.Files);
```

Элементы коллекции просматриваются в цикле `while`:

```
//Цикл по всем файлам в коллекции
while (!Files.atEnd()) {
 Files.moveNext(); //Переходим к следующему файлу
}
```

Для текущего файла в коллекции выделяется его имя, которое затем с помощью метода `test` объекта `RegExp` проверяется на соответствие заданному регулярному выражению:

```
//Выделяем имя файла
SName=Files.item().Name;
//Проверяем, соответствует ли имя файла регулярному выражению
if (RegEx.test(SName)) {
 ColFind++; //Увеличиваем счетчик найденных файлов
```

```
 //Добавляем имя файла к переменной SFileNames
 SFileNames+=SName+ " \n";
}
```

В конце функции `FindFilesInFolder(`*`Fold, RegEx`*`)` на экран выводятся имена найденных файлов и их общее количество:

```
SItog="Найдено файлов: "+ColFind;
//Выводим на экран имена и количество найденных файлов
WScript.Echo(SFileNames+SItog);
```

Листинг 5.18. Поиск файлов, имена которых соответствуют регулярному выражению

```
/*****************************************************************
/*                                                          Имя:
FindRegExp.js                                              */
/*                                                         Язык:
JScript                                                   */
/*  Описание:  Поиск  файлов,  имена  которых  соответствуют
заданному   */
/*                                                   регулярному
выражению                                  */
/*****************************************************************
//Объявляем переменные
var WshShell,FSO,Folder,ColFind,RegExp,SFileNames;
//Функция для поиска файлов в заданном каталоге
function FindFilesInFolder(Fold,RegEx) {
 var Files,SName; //Объявляем переменные
 ColFind=0; //Счетчик найденных файлов
 SFileNames=""; //Строка с именами файлов
 //Создаем коллекцию файлов в каталоге Fold
 Files=new Enumerator(Fold.Files);
 //Цикл по всем файлам в коллекции
 while (!Files.atEnd()) {
  //Выделяем имя файла
  SName=Files.item().Name;
  //Проверяем, соответствует ли имя файла регулярному
  //выражению
  if (RegEx.test(SName)) {
   ColFind++; //Увеличиваем счетчик найденных файлов
   //Добавляем имя файла к переменной SFileNames
   SFileNames+=SName+"\n";
  }
  Files.moveNext(); //Переходим к следующему файлу
 }
```

```
     SItog="Найдено файлов: "+ColFind;
     //Выводим на экран имена и количество найденных файлов
     WScript.Echo(SFileNames+SItog);
    }

    /*******************                                          Начало
*********************************/
    //Создаем объект WshShell
    WshShell=WScript.CreateObject("WScript.Shell");
    //Создаем объект FileSystemObject
    FSO=WScript.CreateObject("Scripting.FileSystemObject");
    //Создаем объект Folder для доступа к подкаталогу ForFind
    //текущего каталога
    Folder = FSO.GetFolder(WshShell.CurrentDirectory+"\\ForFind");
    //Создаем регулярное выражение (объект RegExp)
    RegExp=new RegExp("^[ПАИ].*\.txt$","i");
    //Ищем файлы, имена которых соответствуют регулярному
    //выражению RegExp в каталоге Folder
    FindFilesInFolder(Folder,RegExp);
    /*************                                               Конец
*******************************************/
```

Перемещение файлов с ведением журнала действий

Поставим перед собой следующую задачу. Пусть в заданный каталог на жестком диске (например, C:\In) поступают из почтовой программы или по локальной сети файлы с различными расширениями. Требуется выделить из них все файлы с заданным расширением (например, 003) и перенести их в другой каталог (например, C:\Out). При этом необходимо вести журнал операций (log-файл), в который для каждого переносимого файла записывать следующую информацию: имя файла, дату и время его создания, дату и время перемещения файла. Структура log-файла (в нашем случае это файл C:\In\log.txt) должна быть следующей:

Имя файла (Дата и время создания) Дата и время перемещения

Например:

```
34556.003    (19/10/2002 10:45) 19/10/2002 11:02
43432_KL.003 (19/10/2002 10:45) 19/10/2002 11:02
45.003       (19/10/2002 10:45) 19/10/2002 11:02
              ...
```

Кроме этого, во время перемещения файлов на экран должна выводиться информация о том, какой именно файл обрабатывается, а после завершения работы сценария нужно напечатать общее количество перемещенных файлов (рис. 5.13).

Рис. 5.13. Информация о перемещении файлов

Поставленную задачу выполняет рассматриваемый ниже сценарий MoveLog.js; запускать этот сценарий следует в консольном режиме с помощью cscript.exe.

Пути к каталогу-источнику, в котором первоначально находятся файлы, и к целевому каталогу, в который будут перемещены эти файлы, хранятся соответственно в переменных `Source` и `Dest`:

```
var
  Source="C:\\In\\", //Путь к каталогу-источнику файлов для перемещения
 Dest="C:\\Out\\"; //Путь к целевому каталогу
```

В переменных `Mask` и `PathLog` записаны соответственно расширение файлов для перемещения и путь к log-файлу:

```
var
 Mask="003", //Расширение файлов для перемещения
 PathLog="C:\\In\\log.txt"; //Путь к log-файлу
```

Сначала в сценарии с помощью метода `FolderExists` объекта `FileSystemObject` проверяется наличие на диске каталога-источника; в случае отсутствия этого каталога выводится соответствующее сообщение и выполнение сценария прерывается:

```
//Создаем объект FileSystemObject
FSO=WScript.CreateObject("Scripting.FileSystemObject");
//Проверяем наличие каталога-источника
if (!FSO.FolderExists(Source)) {
 //Выводим сообщение об отсутствии каталога-источника
 WScript.Echo("Каталог-источник", Source, "не существует.");
 WScript.Echo("Нажмите Enter...");
 WScript.StdIn.ReadLine();
 //Выходим из сценария
 WScript.Quit();
```

```
}
```

Аналогичным образом проверяется наличие целевого каталога:

```
//Проверяем наличие целевого каталога
if (!FSO.FolderExists(Dest)) {
 //Выводим сообщение об отсутствии целевого каталога
 WScript.Echo("Целевой каталог", Dest, "не существует.");
 WScript.StdIn.ReadLine();
 WScript.StdIn.ReadLine();
 //Выходим из сценария
 WScript.Quit();
}
```

После этого создается объект `Enumerator` (переменная `Files`) для доступа к коллекции всех файлов в каталоге-источнике:

```
//Создаем объект Folder для каталога-источника
Fold=FSO.GetFolder(Source);
//Создаем объект Enumerator для доступа к коллекции файлов
//в каталоге-источнике
Files=new Enumerator(Fold.Files);
```

Операции записи информации в log-файл и непосредственно переноса файлов из одного каталога в другой реализованы соответственно в функциях `WriteLog()` и `MoveFiles()`.

В функции `WriteLog()` после объявления переменных открывается log-файл в режиме добавления строк:

```
var s, ss, s1, d, File, FLog;
WScript.Echo(" ");
WScript.Echo("Записываем информацию в log-файл...");
//Открываем log-файл для добавления
FLog=FSO.OpenTextFile(PathLog, ForAppending, true);
```

Затем в цикле `while` происходит просмотр коллекции файлов в каталоге-источнике:

```
//Переходим к первому элементу коллекции файлов
Files.moveFirst();
//Цикл по всем файлам в коллекции
while (!Files.atEnd()) {
 //Извлекаем текущий файл из коллекции
 File=Files.item();
 //Переходим к следующему файлу в коллекции
 Files.moveNext();
}
```

Если файл подлежит перемещению (расширение этого файла совпадает с расширением файлов для перемещения), то определяется его имя (свойство Name), дата создания (свойство DateCreated) и текущая дата (объект Date), и в log-файл записывается соответствующая информация:

```
//Выделяем расширение файла
```

```
s=FSO.GetExtensionName(File.Name);
//Проверяем, совпадает ли расширение текущего файла
//с расширением файлов для перемещения
if (s==Mask) {
 //Выводим имя файла на экран
 WScript.Echo("  "+File.Name);
 //Определяем дату создания файла
 d=new Date(File.DateCreated);
 //Формируем строку ss для записи в log-файл
 ss=LFillStr(13,File.Name)
 s1="("+DateToStr(d)+" ";
 s1+=TimeToStr(d)+")";
 ss+=LFillStr(20,s1);
 //Определяем текущую дату
 d=new Date();
 ss+=DateToStr(d);
 ss+=" "+TimeToStr(d);
 //Записываем сформированную строку в log-файл
 FLog.WriteLine(ss);
}
```

Записываемая в log-файл строка формируется в нужном виде с помощью вспомогательных функций `LFillStr` (выравнивание строки влево в поле заданной длины), `DateToStr` (формирование из объекта Date строки формата ДД/ММ/ГГГГ) и `TimeTostr` (формирование из объекта Date строки формата ЧЧ:ММ).

В функции `MoveFiles()`, как и в `WriteLog()`, производится перебор в цикле `while` файлов каталога-источника (элементов коллекции Files). Перемещение файлов осуществляется с помощью последовательного применения методов `Copy` и `Delete`:

```
Files.moveFirst();
//Цикл по всем файлам в коллекции
while (!Files.atEnd()) {
 //Извлекаем текущий файл из коллекции
 File=Files.item();
 //Выделяем расширение файла
 s=FSO.GetExtensionName(File.Name);
 //Проверяем, совпадает ли расширение текущего файла
 //с расширением файлов для перемещения
 if (s==Mask) {
  //Выводим имя файла на экран
  WScript.Echo("  "+File.name);
  //Копируем файл в целевой каталог
  File.Copy(Dest);
  //Удаляем файл
  File.Delete();
```

```
  //Увеличиваем счетчик количества перемещенных файлов
  Col++;
 }
 //Переходим к следующему файлу в коллекции
 Files.moveNext();
}
```

После перемещения всех файлов на экран выводится информация об их количестве:

```
WScript.Echo("Перемещено файлов:", Col);
WScript.Echo("Нажмите Enter...");
WScript.StdIn.ReadLine();
```

Полный текст сценария MoveLog.js приведен в листинге 5.19.

Листинг 5.19. Поиск файлов с ведением log-файла

```
/*****************************************************************
/*                                                          Имя:
MoveLog.js                                                  */
/*                                                         Язык:
JScript                                                    */
/*   Описание:   Перемещение   файлов   из   каталога-источника
в            */
/*                          целевой   каталог   с   ведением   log-
файла                  */
/*****************************************************************
//Объявляем переменные
var
  Source="C:\\In\\",  //Путь  к  каталогу-источнику  файлов  для
перемещения
 Dest="C:\\Out\\",  //Путь к целевому каталогу
 Mask="003",        //Расширение файлов для перемещения
 PathLog="C:\\In\\log.txt",   //Путь к log-файлу
 ForAppending=8;    //Константа для работы с файлами
//Объявляем переменные
var FSO,Fold,Files;
//Функция для записи информации в log-файл
function WriteLog() {
//Объявляем переменные
var s,ss,s1,d,File,FLog;
 WScript.Echo("");
 WScript.Echo("Записываем информацию в log-файл...");
 //Открываем log-файл для добавления
 FLog=FSO.OpenTextFile(PathLog,ForAppending,true);
 //Переходим к первому элементу коллекции файлов
```

```
 Files.moveFirst();
 //Цикл по всем файлам в коллекции
 while (!Files.atEnd()) {
  //Извлекаем текущий файл из коллекции
  File=Files.item();
  //Выделяем расширение файла
  s=FSO.GetExtensionName(File.Name);
  //Проверяем, совпадает ли расширение текущего файла
  //с расширением файлов для перемещения
  if (s==Mask) {
   //Выводим имя файла на экран
   WScript.Echo("  "+File.Name);
   //Определяем дату создания файла
   d=new Date(File.DateCreated);
   //Формируем строку ss для записи в log-файл
   ss=LFillStr(13,File.Name)
   s1="("+DateToStr(d)+" ";
   s1+=TimeToStr(d)+")";
   ss+=LFillStr(20,s1);
   //Определяем текущую дату
   d=new Date();
   ss+=DateToStr(d);
   ss+=" "+TimeToStr(d);
   //Записываем сформированную строку в log-файл
   FLog.WriteLine(ss);
  }
  //Переходим к следующему файлу в коллекции
  Files.moveNext();
 }
}
//Функция для перемещения файлов
function MoveFiles() {
 //Объявляем переменные
 var s,ss,Col,File;

 Col=0; //Счетчик количества перемещенных файлов
 WScript.Echo("");
 WScript.Echo("Перемещаем файлы ...");
 //Переходим к первому элементу коллекции файлов
 Files.moveFirst();
 //Цикл по всем файлам в коллекции
 while (!Files.atEnd()) {
  //Извлекаем текущий файл из коллекции
  File=Files.item();
  //Выделяем расширение файла
  s=FSO.GetExtensionName(File.Name);
  //Проверяем, совпадает ли расширение текущего файла
```

```
      //с расширением файлов для перемещения
      if (s==Mask) {
       //Выводим имя файла на экран
       WScript.Echo("  "+File.name);
       //Копируем файл в целевой каталог
       File.Copy(Dest);
       //Удаляем файл
       File.Delete();
       //Увеличиваем счетчик количества перемещенных файлов
       Col++;
      }
      //Переходим к следующему файлу в коллекции
      Files.moveNext();
     }
     //Выводим информацию о количестве перемещенных файлов
     WScript.Echo("");
     WScript.Echo("Перемещено файлов:",Col);
     WScript.Echo("Нажмите Enter...");
     WScript.StdIn.ReadLine();
    }
    /*******************                                    Начало
*********************************/
    //Создаем объект FileSystemObject
    FSO=WScript.CreateObject("Scripting.FileSystemObject");
    //Проверяем наличие каталога-источника
    if (!FSO.FolderExists(Source)) {
     //Выводим сообщение об отсутствии каталога-источника
     WScript.Echo("Каталог-источник",Source,"не существует.");
     WScript.Echo("Нажмите Enter...");
     WScript.StdIn.ReadLine();
     //Выходим из сценария
     WScript.Quit();
    }
    //Проверяем наличие целевого каталога
    if (!FSO.FolderExists(Dest)) {
     //Выводим сообщение об отсутствии целевого каталога
     WScript.Echo("Целевой каталог",Dest,"не существует.");
     WScript.StdIn.ReadLine();
     WScript.StdIn.ReadLine();
     //Выходим из сценария
     WScript.Quit();
    }
    //Создаем объект Folder для каталога-источника
    Fold=FSO.GetFolder(Source);
    //Создаем объект Enumerator для доступа к коллекцию файлов
    //в каталоге-источнике
    Files=new Enumerator(Fold.Files);
```

```
    //Записываем информацию в log-файл
    WriteLog();
    //Перемещаем файлы в целевой каталог
    MoveFiles();
    /*************                                                    Конец
*****************************************/
    // Вспомогательные функции
    //Дополнение строки ss ведущими нулями до длины ll
    function LeadZero(ll,ss) {
     var i,s,l1;
     s=ss.toString();
     l1=s.length;
     if (l1<=ll) {
      for (i=1;i<=ll-l1;i++) s="0"+s;
     }
     return(s);
    }
    //Формирование из объекта Date строки формата ДД/ММ/ГГГГ
    function DateToStr(dd) {
     var s;
     s=LeadZero(2,dd.getDate())+"/";
     s+=LeadZero(2,dd.getMonth()+1)+"/";
     s+=dd.getYear();
     return(s);
    }
    //Формирование из объекта Date строки формата ЧЧ:ММ
    function TimeToStr(dd) {
     var s;
     s=LeadZero(2,dd.getHours())+":"+LeadZero(2,dd.getMinutes());
     return(s);
    }
    //Выравнивание строки s влево в поле длиной l символов
    function LFillStr(l,s) {
     var ss,i,ll;
     ll=l-s.length;
     if (s.length>=l) {
      return(s);
     } else {
      ss=s;
      for (i=1;i<=ll;i++) {
       ss=ss+" ";
      }
      return(ss);
     }
    }
```

Разработка записной книжки в формате текстового файла

Последний пример, который мы рассмотрим в этой главе, посвящен обработке данных, хранящихся в текстовом файле с разделителями (это может быть, например, выборка из электронной таблицы, базы данных и т.п.).

Предположим, что имеется файл book.txt, содержащий информацию из записной книжки в следующем формате:

```
Фамилия|Имя|Телефон|Улица|Дом|Кв.|Примечание
```

В качестве примера мы будем рассматривать файл book.txt, приведенный в листинге 5.20.

Листинг 5.20. Содержимое файла book.txt

```
Потапов|Сергей|55-55-55|Моховая|3|10|Без примечаний
Попов|Андрей|56-56-56|Ленина|3|5|Без примечаний
Иванов|Иван|17-17-17|Садовая|4|6|Очень хороший человек
Казаков|Сергей|24-19-68|Полежаева|101|22|Тоже очень хороший человек
```

Ниже приведен сценарий SortName.js, который реализует одну из функций для работы с записной книжкой — в результате работы этого сценария все записи из book.txt сортируются по фамилии и отображаются в Блокноте (рис. 5.14).

Опишем кратко алгоритм работы сценария SortName.js.

1. Информация из файла book.txt считывается в массив `PersonArr`. Каждый элемент массива является объектом типа `Person`, в котором хранятся все данные для одного человека.

2. Массив `PersonArr` сортируется в нужном порядке (в нашем случае — по возрастанию фамилий).

3. Содержимое всех записей из массива `PersonArr` выводится в текстовый файл out.txt.

4. Файл out.txt открывается в Блокноте.

Как мы видим, взаимодействие с текстовым файлом данных происходит только при загрузке его содержимого в массив. Поэтому остальная часть сценария не зависит от формата файла с данными и может в дальнейшем применяться без изменений для обработки файлов в XML-формате.

Рис. 5.14. Содержимое записной книжки (сортировка по фамилии)

Запускной функцией в SortName.js является функция `Main()`:

```
function Main() {
 //Создаем объект WshShell
 WshShell = WScript.CreateObject("WScript.Shell");
 //Определяем пути к файлам
 InitPath();
 //Создаем объект FileSystemObject
 FSO=WScript.CreateObject("Scripting.FileSystemObject");
 //Открываем выходной файл для записи
 FOut=FSO.OpenTextFile(PathOut,ForWriting,true);
 //Печатаем заголовок отчета
 TopReport("Сортировка по фамилии");
 //Выводим содержимого файла с данными
 ListFile();
 //Печатаем итоговую информацию
 BottomReport();
 //Открываем выходной файл в Блокноте
 MakeOut();
}
```

Итак, основная работа по формированию выходного файла, который

затем будет открыт в Блокноте, производится в функции ListFile(). В свою очередь, в этой функции вызываются функции FileToArray() и ListPersonArray().

```
function ListFile() {
 //Считывание данных из файла в массив
 FileToArray();
 //Запись информации из массива в выходной файл
 ListPersonArray();
}
```

Опишем сначала функцию FileToArray(). Здесь производится создание массива PersonArr, чтение в цикле while строк из файла book.txt (с помощью метода ReadLine) и формирование массива ss (с помощью метода split), элементы которого содержат данные для текущей записи. После этого вызывается функция PersonToArray() для создания нового элемента массива PersonArr:

```
function FileToArray() {
var s,ss;
 //Открываем файл с данными для чтения
 FBook=FSO.OpenTextFile(PathBook,ForReading,true);
 //Создаем массив PersonArr
 PersonArr=new Array();
 //Читаем содержимое файла FBook
 while (!FBook.AtEndOfStream) {
  //Считываем строку
  s=FBook.ReadLine();
  //Записываем содержимое строки s в массив ss
  ss = s.split("|");
  //Добавляем новый элемент в массив объектов Person
  PersonToArray(ss);
 }
 //Закрываем файл
 FBook.Close();
}
```

В функции PersonToArray(*sArr*) из элементов массива sArr формируется экземпляр PersonRec объекта Person. После этого объект PersonRec добавляется в массив PersonArr:

```
function PersonToArray(sArr) {
 //Создаем новый объект Person
   PersonRec=new  Person(sArr[0],  sArr[1],  sArr[2],  sArr[3],
sArr[4], sArr[5], sArr[6]);
 //Сохраняем объект Person в массиве
 PersonArr[PersonArr.length]=PersonRec;
}
```

Конструктор объекта Person имеет следующий вид:

```
function Person(LastName,Name,Phone,Street,House,App,Note) {
 this.LastName=LastName; //Фамилия
 this.Name=Name;         //Имя
 this.Phone=Phone;       //Телефон
 this.Street=Street;     //Улица
 this.House=House;       //Дом
 this.App=App;           //Квартира
 this.Note=Note;         //Примечание
}
```

Перейдем теперь к рассмотрению функции `ListPersonArray()`:

```
function ListPersonArray() {
var i;
 //Сортировка массива по фамилии
 PersonArr.sort(SortLastName);
 //Цикл по всем элементам массива PersonArr
 for (i=0;i<=PersonArr.length-1;i++) {
  //Запись информации в выходной файл
  PrintPerson(PersonArr[i]);
 }
}
```

Сначала здесь производится сортировка массива `PersonArr` в возрастающем порядке по фамилиям, для чего в качестве параметра метода `sort` указывается имя `SortLastName` функции, которая производит сравнение двух элементов массива:

```
function SortLastName(Pers1,Pers2) {
 if (Pers1.LastName<Pers2.LastName) return -1;
 else if (Pers1.LastName==Pers2.LastName) return 0;
 else return 1;
}
```

Замечание

Сортировать массивы подобным образом можно только в сценариях JScript, где массивы рассматриваются как объекты. В сценариях VBScript отсутствуют встроенные средства для сортировки массивов

После сортировки массива содержимое его элементов в цикле выводится в файл out.txt с помощью функции `PrintPerson(PersRec)`:

```
function PrintPerson(PersRec) {
 FOut.WriteLine("Фамилия : "+PersRec.LastName);
 FOut.WriteLine("Имя : "+PersRec.Name);
 FOut.WriteLine("Телефон: "+PersRec.Phone);
```

```
 FOut.WriteLine("Улица: "+PersRec.Street);
 FOut.WriteLine("Дом: "+PersRec.House);
 FOut.WriteLine("Кв.: "+PersRec.App);
 FOut.WriteLine("Заметки: "+PersRec.Note);
 FOut.WriteLine("*********************************");
 //Увеличиваем счетчик числа записей
 NomRec++;
}
```

В листинге 5.21 приводится полный текст сценария SortName.js.

Листинг 5.21. Вывод содержимого записной книжки с сортировкой по фамилии

```
/*****************************************************************
/*                                                          Имя:
SortName.js                                                 */
/*                                                         Язык:
JScript                                                    */
/*  Описание:  Записная  книжка  (данные  в  текстовом  файле
book.txt).  */
/*                    Вывод  всех  записей  с  сортировкой  по
фамилии          */
/*****************************************************************
//Объявляем переменные
var
 WshShell,FSO,
 BasePath,  //Путь к текущему каталогу
 PathBook,  //Путь к файлу с данными
 PathOut,   //Путь к выходному файлу
 FBook,     //Файл с данными
 FOut,      //Выходной файл
 NomRec=0,  //Счетчик количества записей
 PersonRec, //Объект для хранения данных об одном человеке
 PersonArr; //Массив для хранения объектов PersonRec
//Инициализируем константы для работы с файлами
var
 ForReading=1,
 ForWriting=2;

//Конструктор объекта Person
function Person(LastName,Name,Phone,Street,House,App,Note) {
 this.LastName=LastName; //Фамилия
 this.Name=Name;         //Имя
 this.Phone=Phone;       //Телефон
```

```
     this.Street=Street;     //Улица
     this.House=House;       //Дом
     this.App=App;           //Квартира
     this.Note=Note;         //Примечание
    }
    //Заполнение нового элемента массива
    function PersonToArray(sArr) {
     //Создаем новый объект Person
      PersonRec=new  Person(sArr[0],  sArr[1],  sArr[2],  sArr[3],
sArr[4], sArr[5], sArr[6]);
     //Сохраняем объект Person в массиве
     PersonArr[PersonArr.length]=PersonRec;
    }
    //Создание массива объектов Person
    function FileToArray() {
    var s,ss;
     //Открываем файл с данными для чтения
     FBook=FSO.OpenTextFile(PathBook,ForReading,true);
     //Создаем массив PersonArr
     PersonArr=new Array();
     //Читаем содержимое файла FBook
     while (!FBook.AtEndOfStream) {
      //Считываем строку
      s=FBook.ReadLine();
      //Записываем содержимое строки s в массив ss
      ss = s.split("|");
      //Добавляем новый элемент в массив объектов Person
      PersonToArray(ss);
     }
     //Закрываем файл
     FBook.Close();
    }
    //Запись в выходной файл итоговой информации
    function TopReport(Mess) {
     FOut.WriteLine(Mess);
     FOut.WriteLine("--------------------");
     FOut.WriteLine("");
    }
    //Запись в выходной файл итоговой информации
    function BottomReport() {
     FOut.WriteLine("Всего записей: "+NomRec);
    }
    //Запись данных из объекта Person в выходной файл
    function PrintPerson(PersRec) {
     FOut.WriteLine("Фамилия : "+PersRec.LastName);
     FOut.WriteLine("Имя : "+PersRec.Name);
     FOut.WriteLine("Телефон: "+PersRec.Phone);
```

```
 FOut.WriteLine("Улица: "+PersRec.Street);
 FOut.WriteLine("Дом: "+PersRec.House);
 FOut.WriteLine("Кв.: "+PersRec.App);
 FOut.WriteLine("Заметки: "+PersRec.Note);
 FOut.WriteLine("**********************************");
 //Увеличиваем счетчик числа записей
 NomRec++;
}
//Сортировка массива и вывод его содержимого в выходной файл
function ListPersonArray() {
var i;
 //Сортировка массива по фамилии
 PersonArr.sort(SortLastName);
 //Цикл по всем элементам массива PersonArr
 for (i=0;i<=PersonArr.length-1;i++) {
  //Запись информации в выходной файл
  PrintPerson(PersonArr[i]);
 }
}
//Функция для сортировки массива по фамилии
function SortLastName(Pers1,Pers2) {
 if (Pers1.LastName<Pers2.LastName) return -1;
 else if (Pers1.LastName==Pers2.LastName) return 0;
 else return 1;
}

//Вывод содержимого файла с
данными
function ListFile() {
 //Считывание данных из файла в массив
 FileToArray();
 //Запись информации из массива в выходной файл
 ListPersonArray();
}
//Просмотр содержимого выходного файла в Блокноте
function MakeOut() {
 //Закрываем выходной файл
 FOut.Close();
 //Открываем выходной файл в Блокноте
 WshShell.Run("notepad "+PathOut,1);
}
//Построение путей к файлам
function InitPath() {
 BasePath=WshShell.CurrentDirectory+"\\";
 //Путь к файлу с данными
 PathBook=BasePath+"book.txt",
 //Путь к выходному файлу
```

```
 PathOut=BasePath+"out.txt";
}
//Основная запускная функция
function Main() {
 //Создаем объект WshShell
 WshShell = WScript.CreateObject("WScript.Shell");
 //Определяем пути к файлам
 InitPath();
 //Создаем объект FileSystemObject
 FSO=WScript.CreateObject("Scripting.FileSystemObject");
 //Открываем выходной файл для записи
 FOut=FSO.OpenTextFile(PathOut,ForWriting,true);
 //Печатаем заголовок отчета
 TopReport("Сортировка по фамилии");
 //Выводим содержимого файла с данными
 ListFile();
 //Печатаем итоговую информацию
 BottomReport();
 //Открываем выходной файл в Блокноте
 MakeOut();
}
/*******************                                    Начало
********************************/
Main();
/*************                                           Конец
*******************************************/
```

Глава 6
Практическая работа с данными в XML-файлах

В *главе 3* мы уже кратко описывали основные принципы языка XML, которые необходимы для понимания объектной модели сценариев WS XML. В настоящее время применение XML становится все более широким, поэтому настоящая глава посвящена рассмотрению практических примеров сценариев WSH, которые позволяют анализировать и изменять содержимое файлов в формате XML (естественно, описанные методы анализа и модификации XML-файлов могут применяться и в сценариях, которые встроены в HTML-страницы).

Как известно, основной целью разработки XML являлось создание простого текстового формата для хранения и передачи структурированной информации (иерархичность и объектность описываемых данных — ключевые свойства XML). Основные задачи, решаемые при помощи этой технологии в бизнес-приложениях, таковы:

- межплатформенный обмен данными между системами разных разработчиков;
- сбор данных из подразделений организации;
- обмен коммерческими документами между предприятиями;
- сбор отчетности государственными органами.

Сейчас библиотеки для работы с XML созданы практически для всех популярных систем разработки приложений и систем управления базами данных. При использовании сценариев WSH также нет необходимости писать собственные программы для разбора XML-формата (такие программы называются *парсерами*), т.к. встроенный в Windows браузер Internet Explorer версии 4.01 и выше имеет в своем составе в качестве COM-объекта парсер MSXML — Microsoft XML library. В настоящей главе для простоты и краткости изложения мы будем пользоваться лишь двумя объектными моделями, которые предоставляет MSXML, не затрагивая рассмотрение таких специфических для XML-файлов понятий, как определения DTD — Documents Type Definitions, используемые для описания и проверки структуры XML-документа, или стилевые таблицы XSL — Extensible Stylesheet Language, предназначенные для формирования на основе данных из XML-источника страницы HTML.

Записная книжка в формате XML

В предыдущей главе мы рассматривали сценарий для работы с записной книжкой, которая хранится в простом текстовом файле book.txt с разделителями. Каждая строка этого файла содержала одну запись в формате `Фамилия|Имя|Телефон|Улица|Дом|Кв.|Примечание`:

```
Потапов|Сергей|55-55-55|Моховая|3|10|Без примечаний
Попов|Андрей|56-56-56|Ленина|3|5|Без примечаний
Иванов|Иван|17-17-17|Садовая|4|6|Очень хороший человек
Казаков|Сергей|24-19-68|Полежаева|101|22|Тоже очень хороший человек
```

Для преобразования файла book.txt к формату XML мы введем теги, описанные в табл. 6.1.

Таблица 6.1. *Описание тегов для записной книжки в XML-формате*

Тег	Значение
`<PhoneList>`	Корневой тег, обозначает начало записной книжки
`<Person>`	Обозначает начало новой записи в книжке
`<LastName>`	Фамилия человека
`<Name>`	Имя
`<Phone>`	Телефон
`<Street>`	Улица
`<House>`	Дом
`<App>`	Квартира
`<Note>`	Замечания

Иерархия элементов из таблицы 6.1 показана в листинге 6.1.

Листинг 6.1. Иерархия XML-элементов для записной книжки

```
<?xml version="1.0" standalone="yes"?>
<PhoneList>
 <Person>
 <LastName> Фамилия </LastName>
 <Name> Имя </Name>
 <Phone> Телефон </Phone>
```

```
 <Street> Улица </Street>
 <House> Дом </House>
 <App> Квартира </App>
 <Note> Примечание </Note>
 </Person>
 Другие записи
</PhoneList>
```

Файл book.xml для записной книжки формируется в соответствии с листингом 6.1 (листинг 6.2)

Листинг 6.2. Содержимое файла book.xml

```
<?xml version="1.0" encoding="windows-1251"?>
<PhoneList>
 <!-- корневой тэг, список людей -->
 <Person>
  <LastName>Потапов</LastName>
  <Name>Сергей</Name>
  <Phone>55-55-55</Phone>
  <Street>Моховая</Street>
  <House>3</House>
  <App>10</App>
  <Note>Без примечаний</Note>
 </Person>
 <Person>
  <LastName>Попов</LastName>
  <Name>Андрей</Name>
  <Phone>56-56-56</Phone>
  <Street>Ленина</Street>
  <House>3</House>
  <App>5</App>
  <Note>Без примечаний</Note>
 </Person>
 <Person>
  <LastName>Иванов</LastName>
  <Name>Иван</Name>
  <Phone>17-17-17</Phone>
  <Street>Садовая</Street>
  <House>4</House>
  <App>6</App>
  <Note>Очень хороший человек</Note>
 </Person>
 <Person>
  <LastName>Казаков</LastName>
  <Name>Сергей</Name>
```

```
  <Phone>24-19-68</Phone>
  <Street>Полежаева</Street>
  <House>101</House>
  <App>22</App>
  <Note>Тоже очень хороший человек</Note>
 </Person>
</PhoneList>
```

Просмотр XML-файла с помощью объектной модели Internet Explorer 4.0

Если требуется только просматривать и анализировать XML-файл, не модифицируя его, то проще всего воспользоваться объектной моделью MSXML, реализованной в Internet Explorer 4.01.

Замечание

Как отмечено в документации MSDN, эта объектная модель является устаревшей и должна быть заменена моделью XML DOM (*XML Document Object Model*), которая является стандартом корпорации W3C. Однако последняя на момент написания книги версия Internet Explorer 6.0 поддерживает обе эти модели для разбора XML-файлов.

Описание объектной модели

При рассмотрении объектной модели MSXML данные, которые хранятся в XML-файле, удобно представлять в виде иерархического дерева, имеющего один корневой элемент и множество дочерних элементов различного уровня вложенности.

Для анализа содержимого XML-файла используются три объекта: `XML Document` (объект для работы с XML-документом в целом), `XML Element` (отвечает за работу с каждым из элементов XML-файла) и `Element Collection` (коллекция XML-элементов, доступ к которым при помощи метода `item()` возможен по имени или порядковому номеру).

Полный набор свойств и методов этих трех объектов мы рассматривать не будем; в табл. 6.2 и 6.3 приведено описание нескольких основных свойств объектов `XML Document` и `XML Element`, некоторые из них понадобятся нам в дальнейшем при составлении сценария на языке JScript для просмотра записной книжки.

Таблица 6.2. *Свойства объекта* `XML Document`

Свойство	Описание

`URL`	Задает или возвращает путь к обрабатываемому документу
`root`	Содержит корневой элемент XML-документа, Свойство доступно только для чтения
`charset`	Возвращает или устанавливает название текущей кодировочной таблицы
`version`	Содержит номер версии XML. Свойство доступно только для чтения

Таблица 6.3. *Свойства объекта* `XML Element`

Свойство	**Описание**
`children`	Содержит коллекцию дочерних элементов
`tagName`	Содержит имя тега. Свойство доступно для чтения и записи
`text`	Возвращает текстовое содержимое элементов и комментариев
`parent`	Возвращает указатель на родительский элемент. Ссылки на родительский элемент имеют все элементы, за исключением корневого
`type`	Возвращает тип элемента: 0 — элемент, 1 — текст, 2 — комментарий, 3 — Document, 4 — DTD

Пример сценария

С помощью приведенного ниже сценария SortNameMSXML.js все записи из book.xml сортируются по фамилии и отображаются в Блокноте. Напомним, что аналогичную задачу для текстового файла с разделителями book.txt реализует сценарий SortName.js, приведенный в листинге 5.21. Алгоритм работы сценария SortNameMSXML.js, как и SortName.js, сводится к следующим основным шагам.

1. Информация из файла book.xml считывается в массив `PersonArr`. Каждый элемент массива является экземпляром объекта `Person`, в котором хранятся все данные для одного человека.
2. Массив `PersonArr` сортируется по возрастанию фамилий.
3. Содержимое всех записей из массива `PersonArr` выводится в текстовый файл out.txt.
4. Файл out.txt открывается в Блокноте.

Таким образом, специфика работы с XML-файлом проявляется лишь при считывании данных из файла book.xml в массив `PersonArr`. Для этого используется функция `FileToArray()`. Сначала в этой функции создается

пустой массив PersonArr и экземпляр XML объекта XML Document:

```
PersonArr=new Array();
XML=WScript.CreateObject("MSXML");
```

В свойство url объекта XML записывается путь к файлу book.xml, который хранится в переменной PathBook:

```
XML.url=PathBook;
```

Далее в функции FileToArray о определяется количество элементов <Person>, т.е. количество записей в книжке (переменная NomRec):

```
NamRec=XML.root.children.item("Person").length;
```

В цикле for происходит перебор всех элементов <Person>, которые являются элементами соответствующей коллекции:

```
//Перебираем коллекцию XML-элементов Person
for (i=0; i<NomRec; i++) {
 //Выделяем в коллекции XML-элементов i-й элемент Person
 XItem=XML.root.children.item("Person", i);
 //Добавляем новый элемент, в массив объектов Person
 PersonToArray(XItem);
}
```

Как мы видим, каждый элемент <Person> передается в качестве аргумента в функцию PersonToArray(*XItem*), в которой создается новый экземпляр PersonRec объекта Person, заполняются поля этого объекта и происходит добавление PersonRec в массив PersonArr:

```
function PersonToArray(XItem) {
 //Создаем новый экземпляр PersonRec объекта Person
 PersonRec=new Person();
 //Заполняем поля объекта PersonRec
 PersonRec.LastName=GetTagVal(XItem,"LastName");
 PersonRec.Name=GetTagVal(XItem,"Name");
 PersonRec.Phone=GetTagVal(XItem,"Phone");
 PersonRec.Street=GetTagVal(XItem,"Street");
 PersonRec.House=GetTagVal(XItem,"House");
 PersonRec.App=GetTagVal(XItem,"App");
 PersonRec.Note=GetTagVal(XItem,"Note");
 //Сохраняем объект PersonRec в массиве
 PersonArr[PersonArr.length]=PersonRec;
}
```

Поля объекта PersonRec заполняются с помощью функции GetTagVal(*obj, tgName*), которая возвращает значение дочернего для элемента *obj* элемента с именем *tgName*:

```
function GetTagVal(obj, tgName) {
 //Возвращаем значение тега tgName
 return obj.Children.Item(tgName,0).Text;
}
```

В листинге 6.3 приводится полный текст сценария

SortNameMSXMLjs.

Листинг 6.3. Чтение данных из XML-файла с помощью объектной модели Internet Explorer 4.0

```
    /*****************************************************************
    /*                                                            Имя:
SortNameMSXML.js                                                 */
    /*                                                           Язык:
JScript                                                        */
    /*   Описание:   Записная   книжка   (данные   в   XML-файле
book.xml).        */
    /*                Вывод всех записей с сортировкой по фамилии
с         */
    /*                 помощью объектной модели Internet Explorer
4.0        */
    /*****************************************************************
    //Объявляем переменные
    var
     WshShell,FSO,
     BasePath,     //Путь к текущему каталогу
     PathBook,     //Путь к файлу с данными
     PathOut,      //Путь к выходному файлу
     FBook,        //Файл с данными
     FOut,         //Выходной файл
     NomRec=0,     //Счетчик количества записей
     PersonRec,    //Объект для хранения данных об одном человеке
     PersonArr;    //Массив для хранения объектов PersonRec
     ForWriting=2; //Константа для создания выходного файла
    //Конструктор объекта Person
    function Person(LastName,Name,Phone,Street,House,App,Note) {
     this.LastName=LastName; //Фамилия
     this.Name=Name;         //Имя
     this.Phone=Phone;       //Телефон
     this.Street=Street;     //Улица
     this.House=House;       //Дом
     this.App=App;           //Квартира
     this.Note=Note;         //Примечание
    }
    //Определение значения тега tgName XML-элемента obj
    function GetTagVal(obj, tgName) {
     //Возвращаем значение тега tgName
     return obj.Children.Item(tgName,0).Text;
    }
    //Заполнение нового элемента массива
```

```
function PersonToArray(XItem) {
 //Создаем новый экземпляр PersonRec объекта Person
 PersonRec=new Person();
 //Заполняем поля объекта PersonRec
 PersonRec.LastName=GetTagVal(XItem,"LastName");
 PersonRec.Name=GetTagVal(XItem,"Name");
 PersonRec.Phone=GetTagVal(XItem,"Phone");
 PersonRec.Street=GetTagVal(XItem,"Street");
 PersonRec.House=GetTagVal(XItem,"House");
 PersonRec.App=GetTagVal(XItem,"App");
 PersonRec.Note=GetTagVal(XItem,"Note");
 //Сохраняем объект PersonRec в массиве
 PersonArr[PersonArr.length]=PersonRec;
}
//Создание массива объектов Person
function FileToArray() {
var
 XML,NomRec,XItem,ex;
 //Создаем массив PersonArr
 PersonArr=new Array();
 //Создаем объект MSXML
 XML=WScript.CreateObject("MSXML");
 //Задаем путь к файлу с данными
 XML.url=PathBook;
 //Инициализируем счетчик числа элементов Person
 //в XML-файле
 NomRec=0;
 try {
  //Определяем число элементов Person в XML-файле
  NomRec=XML.root.children.item("Person").length;
  if (typeof(NomRec)=="undefined") NomRec=1;
 } catch (ex) {
  NomRec=0;
 }
 //Перебираем коллекцию XML-элементов Person
 for (i=0;i<NomRec;i++) {
  //Выделяем в коллекции XML-элементов i-й элемент Person
  XItem=XML.root.children.item("Person",i);
  //Добавляем новый элемент в массив объектов Person
  PersonToArray(XItem);
 }
}
//Запись в выходной файл заголовка отчета
function TopReport(Mess) {
 FOut.WriteLine(Mess);
 FOut.WriteLine("--------------------");
 FOut.WriteLine("");
```

```
}
//Запись в выходной файл итоговой информации
function BottomReport() {
 FOut.WriteLine("Всего записей: "+NomRec);
}
//Запись данных из объекта Person в выходной файл
function PrintPerson(PersRec) {
 FOut.WriteLine("Фамилия: "+PersRec.LastName);
 FOut.WriteLine("Имя: "+PersRec.Name);
 FOut.WriteLine("Телефон: "+PersRec.Phone);
 FOut.WriteLine("Улица: "+PersRec.Street);
 FOut.WriteLine("Дом: "+PersRec.House);
 FOut.WriteLine("Кв.: "+PersRec.App);
 FOut.WriteLine("Заметки: "+PersRec.Note);
 FOut.WriteLine("**********************************");
 NomRec++;
}
//Сортировка массива и вывод его содержимого в выходной файл
function ListPersonArray() {
var i,a;
 //Сортировка массива по фамилии
 PersonArr.sort(SortLastName);
 //Цикл по всем элементам массива PersonArr
 for (i=0;i<=PersonArr.length-1;i++) {
  //Запись информации в выходной файл
  PrintPerson(PersonArr[i]);
 }
}
//Функция для сортировки массива по фамилии
function SortLastName(Pers1,Pers2) {
 if (Pers1.LastName<Pers2.LastName) return -1;
 else if (Pers1.LastName==Pers2.LastName) return 0;
 else return 1;
}
//Вывод содержимого файла с данными
function ListFile() {
 //Считывание данных из файла в массив
 FileToArray();
 //Запись информации из массива в выходной файл
 ListPersonArray();
}
//Просмотр содержимого выходного файла в Блокноте
function MakeOut() {
 //Закрываем выходной файл
 FOut.Close();
 //Открываем выходной файл в Блокноте
 WshShell.Run("notepad "+PathOut,1);
```

```
    }
    //Построение путей к файлам
    function InitPath() {
     BasePath=WshShell.CurrentDirectory+"\\";
     //Путь к файлу с данными
     PathBook=BasePath+"book.xml",
     //Путь к выходному файлу
     PathOut=BasePath+"out.txt";
    }
    //Основная запускная функция
    function Main() {
     //Создаем объект WshShell
     WshShell = WScript.CreateObject("WScript.Shell");
     //Определяем пути к файлам
     InitPath();
     //Создаем объект FileSystemObject
     FSO=WScript.CreateObject("Scripting.FileSystemObject");
     //Открываем выходной файл для записи
     FOut=FSO.OpenTextFile(PathOut,ForWriting,true);
     //Печатаем заголовок отчета
     TopReport("Сортировка по фамилии");
     //Выводим содержимого файла с данными
     ListFile();
     //Печатаем итоговую информацию
     BottomReport("Всего записей: "+PersonArr.length);
     //Открываем выходной файл в Блокноте
     MakeOut();
    }
    /*******************                                    Начало
*********************************/
    Main();
    /*************                                          Конец
*********************************************/
```

Использование XML DOM для просмотра и изменения XML-файла

Объектная модель XML DOM (XML Document Object Model, *объектная модель документа XML*) является рекомендованным корпорацией W3C стандартом, который определяет интерфейсы, с помощью которых приложения могут загружать XML-файл, просматривать его содержимое, производить поиск, добавление, изменение и удаление данных, сохранять сделанные изменения в файле. Отметим, что в модели XML DOM документ в формате XML рассматривается как иерархическое дерево, которое состоит из элементов, называемых *узлами (nodes)*, и имеет один корневой элемент (узел).

Замечание

В дальнейшем в этой главе терминами "элемент" и "узел" мы будем пользоваться как синонимами.

Описание модели XML DOM

Парсер MSXML поддерживает много объектов, определяемых в модели XML DOM, с помощью которых можно решать связанные с XML задачи различного уровня сложности. Нам в дальнейшем для написания сценариев, которые осуществляют просмотр записной книжки в XML-формате, а также поиск, добавление и удаление записей из этой книжки, понадобятся только три основных объекта: `DOMDocument` (представляет XML-документ в целом), `XMLDOMNode` (представляет одиночный XML-элемент, т. е. один узел в дереве) и `XMLDOMNodeList` (коллекция элементов, являющихся дочерними по отношению к определенному узлу в дереве, доступ к которым возможен по порядковому номеру при помощи метода `item()`).

В свою очередь, объекты `DOMDocument` и `XMLDOMNode` имеют множество свойств и методов, некоторые из них (включая все свойства и методы, которые используются при написании сценариев для работы с записной книжкой) описаны в табл. 6.4–6.6.

Таблица 6.4. *Свойства и методы объекта* `DOMDocument`

Название	Тип	Описание
`childNodes`	Свойство	Содержит коллекцию всех узлов документа. Свойство доступно только для чтения
`documentElement`	Свойство	Содержит ссылку на корневой элемент документа. Свойство доступно как для чтения, так и для записи
`getElementsByTagName(`*`tagName`*`)`	Метод	Возвращает коллекцию всех элементов в документе, имеющих имя, которое задается параметром *`tagName`*
`hasChildNodes()`	Метод	Возвращает `true`, если в документе есть элементы. В противном случае возвращает `false`
`load(url)`	Метод	Загружает XML-документ из файла, путь к которому задан параметром `url`
`loadXML(`*`xmlString`*`)`	Метод	Загружает XML-документ, содержимое которого содержится в строке *`xmlString`*
`url`	Свойство	Содержит путь к загруженному XML-документу. Для того чтобы изменить это свойство, нужно заново загрузить документ с помощью метода `load`

Таблица 6.5. *Свойства объекта* `XMLDOMNode`

Название	Описание
`attributes`	Содержит список атрибутов узла. Свойство доступно только для чтения
`childNodes`	Содержит коллекцию всех узлов, которые являются дочерними по отношению к данному узлу. Свойство доступно только для

	чтения
`firstChild`	Содержит ссылку на первый дочерний узел. Свойство доступно только для чтения
`lastChild`	Содержит ссылку на последний дочерний узел. Свойство доступно только для чтения
`nodeName`	Содержит имя узла. Свойство доступно только для чтения
`parentNode`	Содержит ссылку на родительский узел (для тех узлов, которые имеют родительский элемент). Свойство доступно только для чтения
`text`	Возвращает или устанавливает текстовое содержимое узла

Таблица 6.6. *Методы объекта* `XMLDOMNode`

Название	**Описание**
`appendChild(`*`NewElem`*`)`	Добавляет новый элемент *`NewElem`* в качестве последнего дочернего элемента. В качестве результата возвращает ссылку на добавленный узел
`cloneNode(`*`deep`*`)`	Создает новый узел, который является точной копией текущего узла. Параметр *`deep`* — это логическая константа, которая указывает, нужно ли при создании нового узла копировать дочерние узлы текущего элемента (*`deep`*`=true`), либо этого делать не следует (*`deep`*`=true`)
`hasChildNodes()`	Возвращает `true`, если у узла есть дочерние элементы. В противном случае возвращает `false`
`removeChild(`*`OldElem`*`)`	Удаляет дочерний элемент, ссылка на который содержится в параметре *`OldElem`*
`replaceChild(`*`OldElem, NewElem`*`)`	Заменяет элемент, ссылка на который содержится в параметре *`OldElem`*, на элемент, ссылка на который содержится в параметре *`NewElem`*
	Производит поиск дочерних элементов,

`selectNodes(`*`patternString`*`)`	содержимое которых удовлетворяет шаблону поиска *`patternString`*. В результате возвращает объект `XMLDOMNodeList`, содержащий коллекцию всех найденных узлов
`selectSingleNode(`*`patternString`*`)`	Производит поиск первого дочернего элемента, содержимое которого удовлетворяет шаблону поиска *`patternString`*. В случае удачного поиска возвращает ссылку на найденный элемент, в противном случае возвращает `Null`

Просмотр содержимого записной книжки

Для того чтобы использовать схему XML DOM в сценарии SortNameMSXML.js, осуществляющем вывод информации из XML-файла book.xml в Блокнот, нужно внести изменения в три функции: `GetTagVal(`*`obj, tgName`*`)`, `PersonToArray(`*`XNode`*`)` и `FileToArray()`. Сценарий, который получится в результате этих изменений, назовем SortNameXMLDOM.js.

В функции `FileToArray()` сначала создается пустой массив `PersonArr` и экземпляр `XML` объекта `DOMDocument`:

```
PersonArr=new Array();
XML = WScript.CreateObject("Msxml.DOMDocument");
```

Для загрузки содержимого файла book.xml (путь к этому файлу хранится в переменной `PathBook`) в объект xml, используется метод `load`:

```
XML.load(PathBook);
```

Указатель на корневой элемент записывается в переменную `Root` с помощью свойства `documentElement` объекта `XML`:

```
Root=XML.documentElement;
```

После этого нам остается в цикле перебрать все элементы `Person` (для корневого элемента они являются дочерними элементами первого уровня вложенности) и для каждого из них вызвать функцию `PersonToArray()`:

```
for (i=1; i<=Root.childNodes.length-1; i++) {
 //Выделяем в коллекции XML-элементов i-й элемент
 //первого уровня вложенности
 CurrNode=Root.childNodes.item(i);
 //Добавляем новый элемент в массив объектов Person
 PersonToArray(CurrNode);
```

```
}
```

Функция `PersonToArray(`*`XNode`*`)` в SortNameXMLDOM.js имеет тот же вид, что и в сценарии SortNameMSXML.js:

```
function PersonToArray(XNode) {
 //Создаем новый экземпляр PersonRec объекта Person
 PersonRec=new Person();
 //Заполняем поля объекта PersonRec
 PersonRec.LastName=GetTagVal(XNode,"LastName");
 PersonRec.Name=GetTagVal(XNode,"Name");
 PersonRec.Phone=GetTagVal(XNode,"Phone");
 PersonRec.Street=GetTagVal(XNode,"Street");
 PersonRec.House=GetTagVal(XNode,"House");
 PersonRec.App=GetTagVal(XNode,"App");
 PersonRec.Note=GetTagVal(XNode,"Note");
 //Сохраняем объект PersonRec в массиве
 PersonArr[PersonArr.length]=PersonRec;
}
```

Здесь для построения функции `GetTagVal(`*`obj, tgName`*`)`, которая возвращает значение дочернего для элемента *`obj`* элемента с именем *`tgName`*, используется метод `getElementsByTagName`, возвращающий коллекцию дочерних элементов с заданным именем:

```
function GetTagVal(obj, tgName) {
 var ElemList;
 //Создаем коллекцию дочерних для obj элементов, которые
 //задаются тегом tgName
 ElemList=obj.getElementsByTagName(tgName);
 //Проверяем, есть ли в коллекции ElemList элементы
 if (ElemList.length>0)
  //Возвращаем значение первого встретившегося элемента tgName
  return ElemList.item(0).text
 else return "";
}
```

В листинге 6.4 приводится полный текст сценария SortNameXMLDOM.js.

Листинг 6.4. Чтение данных из XML-файла с помощью XML DOM

```
/*****************************************************************
/*                                                          Имя:
SortNameXMLDOM.js                                          */
/*                                                         Язык:
JScript                                                   */
/*   Описание:   Записная   книжка   (данные   в   XML-файле
```

```
book.xml).        */
    /*                Вывод всех записей с сортировкой по фамилии
с         */
    /*                              помощью объектной модели XML
DOM                      */
    /*****************************************************************
    //Объявляем переменные
    var
     WshShell,FSO,
     BasePath,     //Путь к текущему каталогу
     PathBook,     //Путь к файлу с данными
     PathOut,      //Путь к выходному файлу
     FBook,        //Файл с данными
     FOut,         //Выходной файл
     NomRec=0,     //Счетчик количества записей
     PersonRec,    //Объект для хранения данных об одном человеке
     PersonArr;    //Массив для хранения объектов PersonRec
     ForWriting=2; //Константа для создания выходного файла
    //Конструктор объекта Person
    function Person(LastName,Name,Phone,Street,House,App,Note) {
     this.LastName=LastName; //Фамилия
     this.Name=Name;         //Имя
     this.Phone=Phone;       //Телефон
     this.Street=Street;     //Улица
     this.House=House;       //Дом
     this.App=App;           //Квартира
     this.Note=Note;         //Примечание
    }
    //Определение значения тега tgName XML-элемента obj
    function GetTagVal(obj, tgName) {
     var ElemList;
     //Создаем коллекцию дочерних для obj элементов, которые
     //задаются тегом tgName
     ElemList=obj.getElementsByTagName(tgName);
     //Проверяем, есть ли в коллекции ElemList элементы
     if (ElemList.length>0)
      //Возвращаем значение первого встретившегося элемента tgName
      return ElemList.item(0).text
     else return "";
    }
    //Заполнение нового элемента массива
    function PersonToArray(XNode) {
     //Создаем новый экземпляр PersonRec объекта Person
     PersonRec=new Person();
     //Заполняем поля объекта PersonRec
     PersonRec.LastName=GetTagVal(XNode,"LastName");
     PersonRec.Name=GetTagVal(XNode,"Name");
```

```
      PersonRec.Phone=GetTagVal(XNode,"Phone");
      PersonRec.Street=GetTagVal(XNode,"Street");
      PersonRec.House=GetTagVal(XNode,"House");
      PersonRec.App=GetTagVal(XNode,"App");
      PersonRec.Note=GetTagVal(XNode,"Note");
      //Сохраняем объект PersonRec в массиве
      PersonArr[PersonArr.length]=PersonRec;
     }
     //Создание массива объектов Person
     function FileToArray() {
      var XML,Root,NomRec,CurrNode,ex,i;
      //Создаем массив PersonArr
      PersonArr=new Array();
      //Создаем объект XML DOM
      XML = WScript.CreateObject("Msxml.DOMDocument");
      //Загружаем XML-документ из файла
      XML.load(PathBook);
       //Сохраняем в переменной Root ссылку на корневой элемент
документа
      Root=XML.documentElement;
      //Перебираем все дочерние элементы первого уровня вложенности
      //для корневого элемента
      for (i=1; i<=Root.childNodes.length-1;i++) {
       //Выделяем в коллекции XML-элементов i-й элемент
       CurrNode=Root.childNodes.item(i);
       //Добавляем новый элемент в массив объектов Person
       PersonToArray(CurrNode);
      }
     }
     //Запись в выходной файл заголовка отчета
     function TopReport(Mess) {
      FOut.WriteLine(Mess);
      FOut.WriteLine("--------------------");
      FOut.WriteLine("");
     }
     //Запись в выходной файл итоговой информации
     function BottomReport(Mess) {
      FOut.WriteLine(Mess);
     }
     //Запись данных из объекта Person в выходной файл
     function PrintPerson(PersRec) {
      FOut.WriteLine("Фамилия: "+PersRec.LastName);
      FOut.WriteLine("Имя: "+PersRec.Name);
      FOut.WriteLine("Телефон: "+PersRec.Phone);
      FOut.WriteLine("Улица: "+PersRec.Street);
      FOut.WriteLine("Дом: "+PersRec.House);
      FOut.WriteLine("Кв.: "+PersRec.App);
```

```
 FOut.WriteLine("Заметки: "+PersRec.Note);
 FOut.WriteLine("**********************************");
 NomRec++;
}
//Сортировка массива и вывод его содержимого в выходной файл
function ListPersonArray() {
 var i;
 //Сортировка массива по фамилии
 PersonArr.sort(SortLastName);
 //Цикл по всем элементам массива PersonArr
 for (i=0;i<=PersonArr.length-1;i++) {
  //Запись информации в выходной файл
  PrintPerson(PersonArr[i]);
 }
}
//Функция для сортировки массива по фамилии
function SortLastName(Pers1,Pers2) {
 if (Pers1.LastName<Pers2.LastName) return -1;
 else if (Pers1.LastName==Pers2.LastName) return 0;
 else return 1;
}
//Вывод содержимого файла с данными
function ListFile() {
 //Считывание данных из файла в массив
 FileToArray();
 //Запись информации из массива в выходной файл
 ListPersonArray();
}
//Просмотр содержимого выходного файла в Блокноте
function MakeOut() {
 //Закрываем выходной файл
 FOut.Close();
 //Открываем выходной файл в Блокноте
 WshShell.Run("notepad "+PathOut,1);
}
//Построение путей к файлам
function InitPath() {
 BasePath=WshShell.CurrentDirectory+"\\";
 //Путь к файлу с данными
 PathBook=BasePath+"book.xml",
 //Путь к выходному файлу
 PathOut=BasePath+"out.txt";
}
//Основная запускная функция
function Main() {
 //Создаем объект WshShell
 WshShell = WScript.CreateObject("WScript.Shell");
```

```
    //Определяем пути к файлам
    InitPath();
    //Создаем объект FileSystemObject
    FSO=WScript.CreateObject("Scripting.FileSystemObject");
    //Открываем выходной файл для записи
    FOut=FSO.OpenTextFile(PathOut,ForWriting,true);
    //Печатаем заголовок отчета
    TopReport("Сортировка по фамилии");
    //Выводим содержимого файла с данными
    ListFile();
    //Печатаем итоговую информацию
    BottomReport("Всего записей: "+PersonArr.length);
    //Открываем выходной файл в Блокноте
    MakeOut();
   }
   /*******************                                    Начало
*********************************/
   Main();
   /*************                                          Конец
*********************************************/
```

Добавление информации в записную книжку

В принципе можно добавлять информацию в записную книжку, просто записывая строки с соответствующими тегами в текстовый файл book.xml. Однако лучше для этой цели воспользоваться специальными методами XML DOM (в этом случае не нужно, например, заботиться о закрывающих тегах).

Для иллюстрации методов XML DOM, позволяющих записывать данные в XML-файл, рассмотрим сценарий AddRecord.js, в котором производится добавление в book.xml следующей записи:

```
<Person>
 <LastName>Сидоров</LastName>
 <Name>Антон</Name>
 <Phone>18-18-18</Phone>
 <Street>Саранская</Street>
 <House>12</House>
 <App>4</App>
 <Note>Запись добавлена из сценария</Note>
</Person>
```

Процесс добавления записи в книжку осуществляется в функции `AddRecord()`. Здесь сначала заполняются нужными значениями поля объекта `PersonRec` (функция `MakePersonRec()`), а затем данные из

PersonRec добавляются в файл book.xml (функция RecordToFile(PersonRec)):

```
function AddRecord() {
 //Заполняем поля объекта PersonRec
 MakePersonRec();
 //Сохраняем данные из объекта PersonRec в XML-файле
 RecordToFile(PersonRec);
}
```

Итак, наиболее важной в сценарии является функция RecordToFile(*PersonRec*). В этой функции сначала создается экземпляр XMLDoc объекта DOMDocument и с помощью метода load загружается файл book.xml:

```
XMLDoc = WScript.CreateObject("Msxml.DOMDocument");
XMLDoc.load(PathBook);
```

Указатель на корневой элемент сохраняется в переменной Root:

```
Root=XMLDoc.documentElement;
```

После этого с помощью метода createElement создается новый элемент Person, который затем добавляется в book.xml (метод appendChild):

```
//Создаем XML-элемент Person
NewElem=XMLDoc.createElement("Person");
//Добавляем новый элемент в XML-файл
Root.appendChild(NewElem);
```

Другие добавляемые элементы (LastName, Name, Phone, Street, House, App и Note) должны быть дочерними относительно элемента Person, поэтому в переменной Root мы сохраним ссылку на последний добавленный элемент Person:

```
Root=Root.lastChild;
```

Все элементы добавляются с помощью вызовов методов createElement и appendChild, например:

```
//Создаем элемент LastName
NewElem=XMLDoc.createElement("LastName");
//Добавляем новый элемент в XML-файл (внутри элемента Person)
Root.appendChild(NewElem);
```

Содержимое добавляемых элементов (свойство text) берется из соответствующих полей объекта PersRec, например:

```
//Подставляем в качестве содержимого элемента LastName
//значение поля LastName объекта PersRec
Root.lastChild.text=PersRec.LastName;
```

После того как все нужные элементы добавлены, измененный файл book.xml с помощью метода save сохраняется на жестком диске:

```
XMLDoc.save(PathBook);
```

Полный текст сценария AddRecord.js приводится в листинге 6.5.

Листинг 6.5. Добавление данных в XML-файл с помощью XML DOM

```
    /*****************************************************************
    /*                                                          Имя:
AddRecord.js                                                  */
    /*                                                         Язык:
JScript                                                     */
    /*   Описание:   Записная   книжка   (данные   в   XML-файле
book.xml).        */
    /*                          Вставка  новых  элементов  в  XML-
файл                    */
    /*****************************************************************
    //Объявляем переменные
    var
     WshShell,
     BasePath,  //Путь к текущему каталогу
     PathBook,  //Путь к файлу с данными
     XMLDoc,    //XML-файл с данными
     NomRec=0,  //Счетчик количества записей
     PersonRec, //Объект для хранения данных об одном человеке
     PersonArr; //Массив для хранения объектов PersonRec
    //Инициализируем константы для диалоговых окон
    var
vbYesNo=4,vbQuestion=32,vbInformation=64,vbYes=6,vbOkOnly=0;

    //Конструктор объекта Person
    function Person(LastName,Name,Phone,Street,House,App,Note) {
     this.LastName=LastName; //Фамилия
     this.Name=Name;         //Имя
     this.Phone=Phone;       //Телефон
     this.Street=Street;     //Улица
     this.House=House;       //Дом
     this.App=App;           //Квартира
     this.Note=Note;         //Примечание
    }
    //Заполнение полей объекта PersonRec
    function MakePersonRec() {
     //Создаем экземпляр PersonRec объекта Person
     PersonRec=new Person();
     //Заполняем поля объекта PersonRec
     PersonRec.LastName="Сидоров";
     PersonRec.Name="Антон";
     PersonRec.Phone="18-18-18";
```

```
 PersonRec.Street="Саранская";
 PersonRec.House="12";
 PersonRec.App="4";
 PersonRec.Note="Запись добавлена из сценария";
}

//Сохранение данных из объекта PersonRec в XML-файле
function RecordToFile(PersRec) {
 //Объявляем переменные
 var Root,NewElem,s;
 //Создаем объект DOMDocument
 XMLDoc = WScript.CreateObject("Msxml.DOMDocument");
 //Загружаем XML-файл
 XMLDoc.load(PathBook);
 //Сохраняем в переменной Root ссылку на корневой элемент
 //документа
 Root=XMLDoc.documentElement;
 //Создаем XML-элемент Person
 NewElem=XMLDoc.createElement("Person");
 //Добавляем новый элемент в XML-файл
 Root.appendChild(NewElem);
 //Сохраняем в переменной Root ссылку на последний добавленный
 //элемент Person
 Root=Root.lastChild;
 //Создаем элемент LastName
 NewElem=XMLDoc.createElement("LastName");
 //Добавляем новый элемент в XML-файл (внутри элемента Person)
 Root.appendChild(NewElem);
 //Подставляем в качестве содержимого элемента LastName
 //значение поля LastName объекта PersRec
 Root.lastChild.text=PersRec.LastName;
 //Создаем элемент Name
 NewElem=XMLDoc.createElement("Name");
 //Добавляем новый элемент в XML-файл (внутри элемента Person)
 Root.appendChild(NewElem);
 //Подставляем в качестве содержимого элемента Name
 //значение поля Name объекта PersRec
 Root.lastChild.text=PersRec.Name;
 //Создаем элемент Phone
 NewElem=XMLDoc.createElement("Phone");
 //Добавляем новый элемент в XML-файл (внутри элемента Person)
 Root.appendChild(NewElem);
 //Подставляем в качестве содержимого элемента Phone
 //значение поля Phone объекта PersRec
 Root.lastChild.text=PersRec.Phone;
 //Создаем элемент Street
 NewElem=XMLDoc.createElement("Street");
```

```
 //Добавляем новый элемент в XML-файл (внутри элемента Person)
 Root.appendChild(NewElem);
 //Подставляем в качестве содержимого элемента Street
 //значение поля Street объекта PersRec
 Root.lastChild.text=PersRec.Street;
 //Создаем элемент House
 NewElem=XMLDoc.createElement("House");
 //Добавляем новый элемент в XML-файл (внутри элемента Person)
 Root.appendChild(NewElem);
 //Подставляем в качестве содержимого элемента House
 //значение поля House объекта PersRec
 Root.lastChild.text=PersRec.House;
 //Создаем элемент App
 NewElem=XMLDoc.createElement("App");
 //Добавляем новый элемент в XML-файл (внутри элемента Person)
 Root.appendChild(NewElem);
 //Подставляем в качестве содержимого элемента App
 //значение поля House объекта PersRec
 Root.lastChild.text=PersRec.App;
 //Создаем элемент Note
 NewElem=XMLDoc.createElement("Note");
 //Добавляем новый элемент в XML-файл (внутри элемента Person)
 Root.appendChild(NewElem);
 //Подставляем в качестве содержимого элемента App
 //значение поля House объекта PersRec
 Root.lastChild.text=PersRec.Note;
 //Сохраняем содержимое XML-файла на диске
 XMLDoc.save(PathBook);
}
//Добавление новой записи в книжку
function AddRecord() {
 //Заполняем поля объекта PersonRec
 MakePersonRec();
 //Сохраняем данные из объекта PersonRec в XML-файл
 RecordToFile(PersonRec);
}
//Построение путей к файлам
function InitPath() {
 BasePath=WshShell.CurrentDirectory+"\\";
 //Путь к файлу с данными
 PathBook=BasePath+"book.xml";
}

//Основная запускная функция
function Main() {
 var Res;
 //Создаем объект WshShell
```

```
      WshShell = WScript.CreateObject("WScript.Shell");
      //Определяем пути к файлам
      InitPath();
      //Запрос на создание нового ключа
      Res=WshShell.Popup("Добавить запись в \n"+PathBook+"?", 0,
       "Работа с XML-файлом", vbQuestion+vbYesNo);
      if (Res==vbYes) { //Нажата кнопка Да
       //Добавляем новую запись в книжку
       AddRecord();
       //Выводим информацию на экран
       WshShell.Popup("Новая запись\n\n"+PersonRec.LastName+" "+
        PersonRec.Name+"\n"+PersonRec.Phone+"\n"+
                       PersonRec.Street+",      "+PersonRec.House+"-
"+PersonRec.App+"\n\n"+
        "добавлена в файл "+PathBook, 0,
        "Работа с XML-файлом", vbInformation+vbOkOnly);
      }
    }
    /*******************                                    Начало
*********************************/
    Main();
    /*************                                          Конец
*******************************************/
```

Поиск и удаление записи из книжки

Рассмотрим сценарий FindAndDelRecord.wsf, с помощью которого можно будет полностью удалить из записной книжки данные о человеке, фамилия которого введена в диалоговом окне (рис. 6.1).

Рис. 6.1. Ввод фамилии для удаления

Сценарий FindAndDelRecord.wsf реализован в виде WS-файла для того, чтобы можно было внутри JScript-кода воспользоваться функцией `InputName` на языке VBScript, которая реализует диалоговое окно с полем ввода, показанное на рис. 6.1:

```
    Function InputName
     'Вводим фамилию в диалоговом окне
```

```
      InputName = InputBox("Введите фамилию для удаления:",
"Записная книжка")
    End Function
```

Фамилия, которую следует найти и удалить в записной книжке, сохраняется в глобальной переменной `LastName`:

```
    LastName=InputName();
```

Непосредственно поиск и удаление данных производятся в функции `FindAndDelRecord()`. Здесь, как и во всех рассмотренных ранее примерах, сначала создается экземпляр `XMLDoc` объекта `DOMDocument`, с помощью метода `load` загружается файл book.xml и указатель на корневой элемент сохраняется в переменной `Root`:

```
    XMLDoc = WScript.CreateObject("Msxml.DOMDocument");
    XMLDoc.load(PathBook);
    Root=XMLDoc.documentElement;
```

Для выделения в записной книжке всех фамилий, которые требуется удалить, используется метод `selectNodes()`. В качестве аргумента этого метода подставляется строка `sSelect`, которая указывает, что нужно искать расположенные внутри элементов `Person` элементы с именем `LastName` и значением, которое совпадает со значением переменной `LastName`. Все найденные элементы помещаются в коллекцию `NodeList`:

```
    //Формируем строку для поиска фамилии
    sSelect="Person/LastName[text()='"+LastName+"']";
    //Создаем коллекцию NodeList всех элементов LastName,
    //значение которых совпадает со значением переменной LastName
    NodeList=XMLDoc.documentElement.selectNodes(sSelect);
```

Если найден хотя бы один подходящий элемент `LastName`, т.е. коллекция `NodeList` не является пустой, то для каждого такого элемента в цикле `for` определяется родительский элемент (в нашем случае это элемент `Person`) и этот элемент вместе со всеми своими дочерними элементами удаляется с помощью метода `removeChild()`:

```
    for (i=0;i<=NodeList.length-1;i++) {
     //Определяем родительский элемент (Person) для найденного
     //элемента LastName
     Parent=NodeList.item(i).parentNode;
      //Удаляем элемент Person вместе со всеми его дочерними
элементами
     Root.removeChild(Parent);
     //Выводим диалоговое окно с сообщением
     WshShell.Popup("Запись удалена!",0,
      "Работа с XML-файлом",vbInformation+vbOkOnly);
    }
```

После удаления всех записей содержимое XML-файла book.xml сохраняется на диске с помощью метода `save()`:

```
XMLDoc.save(PathBook);
```

Полный текст сценария FindAndDelRecord.wsf приводится в листинге 6.6.

Листинг 6.6. Поиск и удаление данных в XML-файле с помощью XML DOM

```
<job id="PhoneBook">
<runtime>
<description>
Имя: FindAndDelRecord.wsf
Описание: Записная книжка (данные в XML-файле book.xml).
          Поиск и удаление элементов из XML-файла
</description>
</runtime>
<script language="VBScript">
 'Функция возвращает фамилию для удаления
 Function InputName
 'Вводим фамилию в диалоговом окне
   InputName  =  InputBox("Введите  фамилию  для  удаления:",
"Записная книжка")
 End Function
</script>
<script language="JScript">
//Объявляем переменные
var
 WshShell,
 LastName,  //Удаляемая фамилия
 BasePath,  //Путь к текущему каталогу
 PathBook,  //Путь к файлу с данными
 XMLDoc,    //XML-файл с данными
 NomRec=0,  //Счетчик количества записей
 PersonRec, //Объект для хранения данных об одном человеке
 PersonArr; //Массив для хранения объектов PersonRec
//Инициализируем константы для диалоговых окон
var
vbYesNo=4,vbQuestion=32,vbInformation=64,vbYes=6,vbOkOnly=0;
//Поиск фамилии в записной книжке и удаление всех
//реквизитов, относящихся к этой фамилии
function FindAndDelRecord() {
 var Root,sSelect,i,Parent,NodeList;
 //Создаем объект DOMDocument
 XMLDoc = WScript.CreateObject("Msxml.DOMDocument");
 //Загружаем XML-файл
```

```
      XMLDoc.load(PathBook);
      //Сохраняем в переменной Root ссылку на корневой элемент
      //документа
      Root=XMLDoc.documentElement;
      //Формируем строку для поиска фамилии
      sSelect="Person/LastName[text()='"+LastName+"']";
      //Создаем коллекцию NodeList всех элементов LastName,
      //значение которых совпадает со значением переменной LastName
      NodeList=XMLDoc.documentElement.selectNodes(sSelect);
      if (NodeList.length==0) //Коллекция NodeList пуста
       //Выводим диалоговое окно с сообщением
       WshShell.Popup("Фамилия "+LastName+
        " не найдена в записной книжке!",0,
        "Работа с XML-файлом",vbInformation+vbOkOnly);
      else { //Требуемая фамилия найдена
       //Цикл по всем найденным элементам LastName
       for (i=0;i<=NodeList.length-1;i++) {
        //Определяем родительский элемент (Person) для найденного
        //элемента LastName
        Parent=NodeList.item(i).parentNode;
          //Удаляем элемент Person вместе со всеми его дочерними
элементами
        Root.removeChild(Parent);
        //Выводим диалоговое окно с сообщением
        WshShell.Popup("Запись удалена!",0,
         "Работа с XML-файлом",vbInformation+vbOkOnly);
       }
       //Сохраняем содержимое XML-файла на диске
       XMLDoc.save(PathBook);
      }
     }

     //Построение путей к файлам
     function InitPath() {
      BasePath=WshShell.CurrentDirectory+"\\";
      //Путь к файлу с данными
      PathBook=BasePath+"book.xml";
     }
     //Основная запускная функция
     function Main() {
      var Res;
      //Создаем объект WshShell
      WshShell = WScript.CreateObject("WScript.Shell");
      //Определяем пути к файлам
      InitPath();
      LastName=InputName();
      //Запрос на удаление записи
```

```
      Res=WshShell.Popup("Удалить фамилию "+LastName+
       " из \n"+PathBook+"?",0,
       "Работа с XML-файлом",vbQuestion+vbYesNo);
      if (Res==vbYes) { //Нажата кнопка Да
       //Ищем в книжке нужную фамилию и удаляем относящуюся к
       //ней запись
       FindAndDelRecord();
      }
     }
     /*******************                                    Начало
*********************************/
     Main();
     /*************                                          Конец
*********************************************/
     </script>
     </job>
```

Глава 7
Способы организации диалогового режима работы сценариев

В этой главе на примере работы с записной книжкой будут рассмотрены три способа организации из сценария диалога с пользователем.

Во-первых, мы еще раз подробно остановимся на том, как можно использовать параметры командной строки для запуска различных заданий из многозадачного WS-файла, который в нашем случае будет реализовывать несколько функций для работы с записной книжкой.

Во-вторых, для того же WS-файла мы создадим кнопочное (командное) меню, в котором пользователь сможет выбрать нужное ему действие с помощью ввода определенных символов в диалоговом окне.

Наконец, будет показано, как можно организовать в сценарии полноценный пользовательский интерфейс с помощью HTML-форм и браузера Internet Explorer.

Многозадачный сценарий для работы с записной книжкой

В качестве примера рассмотрим сценарий для работы с записной книжкой в XML-формате (структура файла book.xml, в котором хранится записная книжка, описана в предыдущей главе), в котором будут реализованы следующие функции:

□ просмотр всех записей в алфавитном порядке;

□ поиск записей по фамилии;

□ добавление и удаление записей.

Каждая из этих задач реализуется в виде отдельного задания в файле PhoneBook.wsf (листинг 7.1).

Листинг 7.1. Схема многозадачного файла PhoneBook.wsf

```
    <package>
        <!--    ******************    Просмотр    всех    записей
******************* -->
     <job id="SortName">
      …
     </job>
     <!-- *************** поиск записей по фамилии ************** -
->
     <job id="FindName">
      …
     </job>
       <!--    ***************    удаление    записи    по    фамилии
*************** -->
     <job id="DelRec">
      …
     </job>
     <!-- *************** добавление записи *************** -->
     <job id="AddRec">
      …
     </job>
    </package>
```

Некоторые функции (например, настройка пути к XML-файлу) должны присутствовать во всех заданиях, поэтому такие общие функции мы вынесем в отдельный файл Usage.js, который будет подключаться в

каждом из заданий следующим образом:

```
<script language="JScript" src="Usage.js"/>
```

В файл Usage.js помещены следующие функции:

□ конструктор объекта `Person` (одна запись из книжки);

□ функция `GetTagVal(`*`obj, tgName`*`)`, которая возвращает значения тега *`tgName`* XML-элемента *`obj`*;

□ функция `PersonToArray(`*`XNode`*`)`, которая заполняет поля экземпляра `PersonRec` объекта `Person` данными из соответствующих XML-элементов и добавляет сформированную запись в массив `PersonArr`;

□ функции `TopReport(`*`Mess`*`)` и `BottomReport(`*`Mess`*`)`, с помощью которых в выходной файл печатается заголовок отчета и итоговая информация соответственно;

□ функция `PrintPerson(`*`PersRec`*`)`, в которой происходит вывод данных из полей объекта `PersonRec` в выходной файл;

□ функция `MakeOut()`, которая обеспечивает запуск Блокнота и открывает в нем выходной файл;

□ функция InitPath(), в которой строятся пути к XML-файлу, содержащему данные, и выходному файлу.

Usage.js объявляются глобальные переменные и константы. Полностью содержимое файла Usage.js приведено в листинге 7.2.

Листинг 7.2. Общие функции для работы с записной книжкой

```
    /*****************************************************************
    /*                                                              Имя:
Usage.js                                                        */
    /*                                                             Язык:
JScript                                                        */
    /*      Описание:      Общие      функции      для      записной
книжки                      */
    /*****************************************************************
    //Объявляем переменные
    var
     WshShell,FSO,
     BasePath,     //Путь к текущему каталогу
     PathBook,     //Путь к файлу с данными
     PathOut,      //Путь к выходному файлу
     FBook,        //Файл с данными
     FOut,         //Выходной файл
     NomRec=0,     //Счетчик количества записей
     PersonRec,    //Объект для хранения данных об одном человеке
```

```
 PersonArr;    //Массив для хранения объектов PersonRec
 ForWriting=2; //Константа для создания выходного файла
//Инициализируем константы для диалоговых окон
var
vbYesNo=4,vbQuestion=32,vbInformation=64,vbYes=6,vbOkOnly=0;

//Конструктор объекта Person
function Person(LastName,Name,Phone,Street,House,App,Note) {
 this.LastName=LastName; //Фамилия
 this.Name=Name;         //Имя
 this.Phone=Phone;       //Телефон
 this.Street=Street;     //Улица
 this.House=House;       //Дом
 this.App=App;           //Квартира
 this.Note=Note;         //Примечание
}
//Определение значения тега tgName XML-элемента obj
function GetTagVal(obj, tgName) {
 var ElemList;
 //Создаем коллекцию дочерних для obj элементов, которые
 //задаются тегом tgName
 ElemList=obj.getElementsByTagName(tgName);
 //Проверяем, есть ли в коллекции ElemList элементы
 if (ElemList.length>0)
  //Возвращаем значение тега tgName
  return ElemList.item(0).text
 else return "";
}
//Заполнение нового элемента массива
function PersonToArray(XNode) {
 //Создаем новый экземпляр PersonRec объекта Person
 PersonRec=new Person();
 //Заполняем поля объекта PersonRec
 PersonRec.LastName=GetTagVal(XNode,"LastName");
 PersonRec.Name=GetTagVal(XNode,"Name");
 PersonRec.Phone=GetTagVal(XNode,"Phone");
 PersonRec.Street=GetTagVal(XNode,"Street");
 PersonRec.House=GetTagVal(XNode,"House");
 PersonRec.App=GetTagVal(XNode,"App");
 PersonRec.Note=GetTagVal(XNode,"Note");
 //Сохраняем объект PersonRec в массиве
 PersonArr[PersonArr.length]=PersonRec;
}
//Запись в выходной файл заголовка отчета
function TopReport(Mess) {
 FOut.WriteLine(Mess);
 FOut.WriteLine("--------------------");
```

```
 FOut.WriteLine("");
}
//Запись в выходной файл итоговой информации
function BottomReport(Mess) {
 FOut.WriteLine(Mess);
}
//Запись данных из объекта Person в выходной файл
function PrintPerson(PersRec) {
 FOut.WriteLine("Фамилия: "+PersRec.LastName);
 FOut.WriteLine("Имя: "+PersRec.Name);
 FOut.WriteLine("Телефон: "+PersRec.Phone);
 FOut.WriteLine("Улица: "+PersRec.Street);
 FOut.WriteLine("Дом: "+PersRec.House);
 FOut.WriteLine("Кв.: "+PersRec.App);
 FOut.WriteLine("Заметки: "+PersRec.Note);
 FOut.WriteLine("********************************");
 NomRec++;
}
//Просмотр содержимого выходного файла в Блокноте
function MakeOut() {
 //Закрываем выходной файл
 FOut.Close();
 //Открываем выходной файл в Блокноте
 WshShell.Run("notepad "+PathOut,1);
}
//Построение путей к файлам
function InitPath() {
 BasePath=WshShell.CurrentDirectory+"\\";
 //Путь к файлу с данными
 PathBook=BasePath+"book.xml",
 //Путь к выходному файлу
 PathOut=BasePath+"out.txt";
}
/*************                                                    Конец
*****************************************/
```

Также в отдельный файл WSHInputBox.vbs мы поместим функцию `WSHinputBox(`*`Message, Title`*`)` на языке VBScript, с помощью которой из JScript-сценариев будет выводиться диалоговое окно со строкой ввода (напомним, что ни язык JScript, ни объектная модель WSH такой функции не предоставляют):

```
Function WSHInputBox(Message,Title)
 'Выводим диалоговое окно со строкой ввода
 WSHInputBox = InputBox(Message,Title)
End Function
```

Сами задания из файла PhoneBook.wsf составлены (с некоторыми

изменениями, на которых мы подробно останавливаться не будем) из одиночных сценариев, которые были рассмотрены в предыдущей главе: см. листинг 6.3 (просмотр всех записей в алфавитном порядке), листинг 6.4 (добавление записей) и листинг 6.5 (поиск записей по фамилии и удаление записей).

В листинге 7.3 приводится полный текст сценария PhoneBook.wsf.

Листинг 7.3. Многозадачный сценарий PhoneBook.wsf для работы с записной книжкой

```
    <package>
    <!--    ******************        Просмотр    всех    записей
****************** -->
    <job id="SortName">
    <script language="JScript" src="usage.js"/>
    <script language="JScript">
    //Создание массива объектов Person
    function FileToArray() {
     var XML,Root,NomRec,CurrNode,ex,i;
     //Создаем массив PersonArr
     PersonArr=new Array();
     //Создаем объект XML DOM
     XML = WScript.CreateObject("Msxml.DOMDocument");
     //Загружаем XML-документ из файла
     XML.load(PathBook);
     //Сохраняем в переменной Root ссылку на корневой элемент
     //документа
     Root=XML.documentElement;
     //Перебираем все дочерние элементы первого уровня вложенности
     //для корневого элемента
     for (i=1; i<=Root.childNodes.length-1;i++) {
      //Выделяем в коллекции XML-элементов i-й элемент
      CurrNode=Root.childNodes.item(i);
      //Добавляем новый элемент в массив объектов Person
      PersonToArray(CurrNode);
     }
    }
    //Сортировка массива и вывод его содержимого в выходной файл
    function ListPersonArray() {
     var i;
     //Сортировка массива по фамилии
     PersonArr.sort(SortLastName);
     //Цикл по всем элементам массива PersonArr
     for (i=0;i<=PersonArr.length-1;i++) {
```

```
      //Запись информации в выходной файл
      PrintPerson(PersonArr[i]);
     }
    }
    //Функция для сортировки массива по фамилии
    function SortLastName(Pers1,Pers2) {
     if (Pers1.LastName<Pers2.LastName) return -1;
     else if (Pers1.LastName==Pers2.LastName) return 0;
     else return 1;
    }

    //Основная запускная функция
    function Main() {
     //Создаем объект WshShell
     WshShell = WScript.CreateObject("WScript.Shell");
     //Определяем пути к файлам
     InitPath();
     //Создаем объект FileSystemObject
     FSO=WScript.CreateObject("Scripting.FileSystemObject");
     //Открываем выходной файл для записи
     FOut=FSO.OpenTextFile(PathOut,ForWriting,true);
     //Печатаем заголовок отчета
     TopReport("Список всех записей, сортировка по фамилии");
     //Считываем данные из файла в массив
     FileToArray();
     //Записываем информацию из массива в выходной файл
     ListPersonArray();
     //Печатаем итоговую информацию
     BottomReport("Всего записей: "+PersonArr.length);
     //Открываем выходной файл в Блокноте
     MakeOut();
    }
    //Запускаем основную функцию
    Main();
    </script>
    </job>

    <!-- ***************  Поиск записей по фамилии ************** -
->
    <job id="FindName">
    <script language="VBScript" src="WSHInputBox.vbs"/>
    <script language="JScript" src="usage.js"/>
    <script language="JScript">
    //Поиск в XML-файле нужных записей и сохранение их в
    //массиве PersonArr
    function RecordsToArray(LastName) {
```

```
 var XMLDoc,Root,sSelect,i,Parent,NodeList;
 //Создаем массив PersonArr
 PersonArr=new Array();
 //Создаем объект DOMDocument
 XMLDoc = WScript.CreateObject("Msxml.DOMDocument");
 //Загружаем XML-файл
 XMLDoc.load(PathBook);
 //Сохраняем в переменной Root ссылку на корневой элемент
 //документа
 Root=XMLDoc.documentElement;
 //Формируем строку для поиска фамилии
 sSelect="Person/LastName[text()='"+LastName+"']";
 //Создаем коллекцию NodeList всех элементов LastName,
 //значение которых совпадает со значением переменной LastName
 NodeList=XMLDoc.documentElement.selectNodes(sSelect);
 if (NodeList.length==0) {//Коллекция NodeList пуста
  //Выводим диалоговое окно с сообщением
  WshShell.Popup("Фамилия "+ LastName+ " не найдена!", 0,
   "Записная книжка",vbInformation+vbOkOnly);
  //Завершаем выполнение задания
  WScript.Quit();
 } else  {  //Требуемая фамилия найдена
  //Цикл по всем найденным элементам LastName
  for (i=0;i<=NodeList.length-1;i++) {
   //Определяем родительский элемент (Person) для найденного
   //элемента LastName
   Parent=NodeList.item(i).parentNode;
   //Добавляем новый элемент в массив объектов Person
   PersonToArray(Parent);
  }
 }
}
//Вывод в выходной файл информации о найденных записях
function PrintAllFind() {
 var i;
 for (i=0;i<PersonArr.length;i++) {
  PrintPerson(PersonArr[i]);
 }
}
//Основная запускная функция
function Main() {
var LastName;
 //Создаем объект WshShell
 WshShell = WScript.CreateObject("WScript.Shell");
 //Определяем пути к файлам
 InitPath();
 //Создаем объект FileSystemObject
```

```
     FSO=WScript.CreateObject("Scripting.FileSystemObject");
     //Открываем выходной файл для записи
     FOut=FSO.OpenTextFile(PathOut,ForWriting,true);
     //Печатаем заголовок отчета
     TopReport("Поиск записей");
     //Вводим фамилию для поиска
      LastName=WSHInputBox("Введите фамилию для поиска:","Записная
книжка")
      //Ищем в XML-файле нужные записи и сохраняем их в массиве
PersonArr
     RecordsToArray(LastName);
     //Выводим все найденные записи из массива PersonArr в выходной
файл
     PrintAllFind(LastName);
     //Печатаем итоговую информацию
     BottomReport("Всего найдено: "+PersonArr.length);
     //Открываем выходной файл в Блокноте
     MakeOut();
    }
    //Запускаем основную функцию
    Main();
    </script>
    </job>
    <!--    ***************       Удаление    записи    по    фамилии
*************** -->
    <job id="DelRec">
    <script language="VBScript" src="WSHInputBox.vbs"/>
    <script language="JScript" src="usage.js"/>
    <script language="JScript">
    //Поиск фамилии в записной книжке и удаление всех
    //реквизитов, относящихся к этой фамилии
    function FindAndDelRecord(LastName) {
     var Root,sSelect,i,Parent,NodeList;
     //Создаем объект DOMDocument
     XMLDoc = WScript.CreateObject("Msxml.DOMDocument");
     //Загружаем XML-файл
     XMLDoc.load(PathBook);
     //Сохраняем в переменной Root ссылку на корневой элемент
     //документа
     Root=XMLDoc.documentElement;
     //Формируем строку для поиска фамилии
     sSelect="Person/LastName[text()='"+LastName+"']";
     //Создаем коллекцию NodeList всех элементов LastName,
     //значение которых совпадает со значением переменной LastName
     NodeList=XMLDoc.documentElement.selectNodes(sSelect);
     if (NodeList.length==0) //Коллекция NodeList пуста
      //Выводим диалоговое окно с сообщением
```

```
      WshShell.Popup("Фамилия "+LastName+ " не найдена!", 0,
       "Записная книжка",vbInformation+vbOkOnly);
     else { //Требуемая фамилия найдена
      //Цикл по всем найденным элементам LastName
      for (i=0;i<=NodeList.length-1;i++) {
       //Определяем родительский элемент (Person) для найденного
       //элемента LastName
       Parent=NodeList.item(i).parentNode;
         //Удаляем элемент Person вместе со всеми его дочерними
элементами
       Root.removeChild(Parent);
       //Выводим диалоговое окно с сообщением
       WshShell.Popup("Запись удалена!",0,
        "Записная книжка",vbInformation+vbOkOnly);
      }
      //Сохраняем содержимое XML-файла на диске
      XMLDoc.save(PathBook);
     }
    }
    //Основная запускная функция
    function Main() {
     var LastName,Res;
     //Создаем объект WshShell
     WshShell = WScript.CreateObject("WScript.Shell");
     //Определяем пути к файлам
     InitPath();
     LastName=WSHInputBox("Введите фамилию для удаления:","Записная
книжка")
     //Запрос на удаление записи
       Res=WshShell.Popup("Удалить   фамилию   "+LastName+   "   из
\n"+PathBook+"?",0,
      "Записная книжка",vbQuestion+vbYesNo);
     if (Res==vbYes) { //Нажата кнопка Да
      //Ищем в книжке нужную фамилию и удаляем относящуюся к
      //ней запись
      FindAndDelRecord(LastName);
     }
    }
    //Запускаем основную функцию
    Main();
    </script>
    </job>

    <!-- ***************  Добавление записи *************** -->
    <job id="AddRec">
    <script language="VBScript" src="WSHInputBox.vbs"/>
    <script language="JScript" src="Usage.js"/>
```

```
    <script language="JScript">
    //Ввод значений полей объекта PersonRec
    function MakePersonRec() {
     //Создаем новый экземпляр PersonRec объекта Person
     PersonRec=new Person();
      //Вводим значения полей добавляемой записи с помощью
диалогового
     //окна со строкой ввода
      PersonRec.LastName=WSHInputBox("Введите  фамилию","Добавление
записи");
     PersonRec.Name=WSHInputBox("Введите имя","Добавление записи");
        PersonRec.Phone=WSHInputBox("Введите   телефон","Добавление
записи");
         PersonRec.Street=WSHInputBox("Введите    улицу","Добавление
записи");
          PersonRec.House=WSHInputBox("Введите     дом","Добавление
записи");
         PersonRec.App=WSHInputBox("Введите    квартиру","Добавление
записи");
       PersonRec.Note=WSHInputBox("Введите   примечание","Добавление
записи");
    }
    //Сохранение данных из объекта PersonRec в XML-файле
    function RecordToFile(PersRec) {
     var Root,NewElem,s;
     //Создаем объект DOMDocument
     XMLDoc = WScript.CreateObject("Msxml.DOMDocument");
     //Загружаем XML-файл
     XMLDoc.load(PathBook);
     //Сохраняем в переменной Root ссылку на корневой элемент
     //документа
     Root=XMLDoc.documentElement;
     //Создаем XML-элемент Person
     NewElem=XMLDoc.createElement("Person");
     //Добавляем новый элемент в XML-файл
     Root.appendChild(NewElem);
     //Сохраняем в переменной Root ссылку на последний добавленный
     //элемент Person
     Root=Root.lastChild;
     //Создаем элемент LastName
     NewElem=XMLDoc.createElement("LastName");
     //Добавляем новый элемент в XML-файл (внутри элемента Person)
     Root.appendChild(NewElem);
     //Подставляем в качестве содержимого элемента LastName
     //значение поля LastName объекта PersRec
     Root.lastChild.text=PersRec.LastName;
     //Создаем элемент Name
```

```
 NewElem=XMLDoc.createElement("Name");
 //Добавляем новый элемент в XML-файл (внутри элемента Person)
 Root.appendChild(NewElem);
 //Подставляем в качестве содержимого элемента Name
 //значение поля Name объекта PersRec
 Root.lastChild.text=PersRec.Name;
 //Создаем элемент Phone
 NewElem=XMLDoc.createElement("Phone");
 //Добавляем новый элемент в XML-файл (внутри элемента Person)
 Root.appendChild(NewElem);
 //Подставляем в качестве содержимого элемента Phone
 //значение поля Phone объекта PersRec
 Root.lastChild.text=PersRec.Phone;
 //Создаем элемент Street
 NewElem=XMLDoc.createElement("Street");
 //Добавляем новый элемент в XML-файл (внутри элемента Person)
 Root.appendChild(NewElem);
 //Подставляем в качестве содержимого элемента Street
 //значение поля Street объекта PersRec
 Root.lastChild.text=PersRec.Street;
 //Создаем элемент House
 NewElem=XMLDoc.createElement("House");
 //Добавляем новый элемент в XML-файл (внутри элемента Person)
 Root.appendChild(NewElem);
 //Подставляем в качестве содержимого элемента House
 //значение поля House объекта PersRec
 Root.lastChild.text=PersRec.House;
 //Создаем элемент App
 NewElem=XMLDoc.createElement("App");
 //Добавляем новый элемент в XML-файл (внутри элемента Person)
 Root.appendChild(NewElem);
 //Подставляем в качестве содержимого элемента App
 //значение поля House объекта PersRec
 Root.lastChild.text=PersRec.App;
 //Создаем элемент Note
 NewElem=XMLDoc.createElement("Note");
 //Добавляем новый элемент в XML-файл (внутри элемента Person)
 Root.appendChild(NewElem);
 //Подставляем в качестве содержимого элемента App
 //значение поля House объекта PersRec
 Root.lastChild.text=PersRec.Note;
 //Сохраняем содержимое XML-файла на диске
 XMLDoc.save(PathBook);
}
//Добавление новой записи в книжку
function AddRecord() {
 //Заполняем поля объекта PersonRec
```

```
     MakePersonRec();
     //Сохраняем данные из объекта PersonRec в XML-файл
     RecordToFile(PersonRec);
    }
    //Построение путей к файлам
    function InitPath() {
     BasePath=WshShell.CurrentDirectory+"\\";
     //Путь к файлу с данными
     PathBook=BasePath+"book.xml";
    }
    //Основная запускная функция
    function Main() {
     var Res;
     //Создаем объект WshShell
     WshShell = WScript.CreateObject("WScript.Shell");
     //Определяем пути к файлам
     InitPath();
     //Запрос на создание нового ключа
     Res=WshShell.Popup("Добавить запись в \n"+PathBook+"?",0,
      "Записная книжка",vbQuestion+vbYesNo);
     if (Res==vbYes) { //Нажата кнопка Да
      //Добавляем новую запись в книжку
      AddRecord();
      //Выводим информацию на экран
      WshShell.Popup("Новая запись\n\n"+PersonRec.LastName+" "+
       PersonRec.Name+"\n"+PersonRec.Phone+"\n"+
                      PersonRec.Street+",      "+PersonRec.House+"-
"+PersonRec.App+"\n\n"+
       "добавлена!",0, "Записная книжка",vbInformation+vbOkOnly);
     }
    }
    //Запускаем основную функцию
    Main();
    </script>
    </job>
    </package>
```

Итак, у нас теперь имеется многозадачный WS-файл PhoneBook.wsf, обеспечивающий необходимую функциональность для работы с записной книжкой, и следующая задача состоит в организации более или менее удобного диалога с пользователем для запуска заданий из этого файла.

Обработка параметров командной строки

Самый простой вариант организовать диалог с пользователем состоит в использовании параметров командной строки. Напомним, что объектная

модель WSH предоставляет несколько методов, которые позволяют производить анализ именных и безымянных параметров *(см. разд. "Работа с параметрами командной строки сценария" главы 2)*, с которыми был запущен сценарий, а в схеме WS XML есть несколько специальных элементов (<runtime>, <named>, <unnamed>, <description> и <example>), предназначенных для быстрого создания встроенной справки, описывающей синтаксис сценария и смысл каждого из параметров.

Для нашего примера мы создадим сценарий ArgMenu.wsf, в котором будем анализировать аргументы командной строки и в зависимости от них запускать то или иное задание из файла PhoneBook.wsf. Названия и назначения именных параметров, которые мы будем использовать, приведены в табл. 7.1.

Таблица 7.1. *Параметры командной строки сценария для работы с записной книжкой*

Название параметра	**Назначение**
/L	Просмотр всех записей книжки (сортировка по фамилии)
/F	Поиск записей по фамилии, которая вводится в диалоговом окне
/A	Добавление записи по фамилии (данные вводятся в диалоговом окне)
/D	Удаление записи (фамилия для удаления вводится в диалоговом окне)

Если запустить сценарий ArgMenu.wsf вообще без параметров, либо с параметрами, не указанными в табл. 7.1, то на экран будет выведена встроенная справка (рис. 7.1).

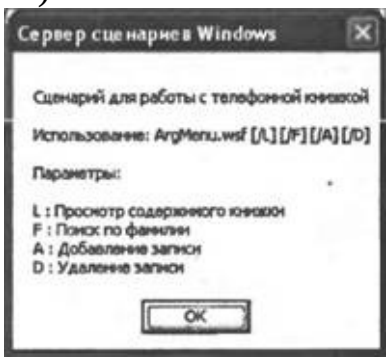


Рис. 7.1. Встроенная справка для сценария ArgMenu.wsf

В листинге 7.4 приводится полный текст сценария ArgMenu.wsf.

Листинг 7.4. Обработка параметров командной строки сценария для работы с записной книжкой

```
    <job id="ArgMenu">
     <runtime>
      <description>
      Сценарий для работы с телефонной книжкой
      </description>
        <named name="L" helpstring="Просмотр содержимого книжки"
type="simple" required="false"/>
       <named name="F" helpstring="Поиск по фамилии" type="simple"
required="false"/>
       <named name="A" helpstring="Добавление записи" type="simple"
required="false"/>
       <named name="D" helpstring="Удаление записи" type="simple"
required="false"/>
     </runtime>
     <script language="JScript">
      var WshShell;
      WshShell=WScript.CreateObject("WScript.Shell");

      if ((WScript.Arguments.Named.Exists("L")) ||
       (WScript.Arguments.Named.Exists("l"))) {
       WshShell.Run("wscript PhoneBook.wsf //Job:SortName");
       WScript.Quit();
      }
      if ((WScript.Arguments.Named.Exists("F")) ||
       (WScript.Arguments.Named.Exists("f"))) {
       WshShell.Run("wscript PhoneBook.wsf //Job:FindName");
       WScript.Quit();
      }
      if ((WScript.Arguments.Named.Exists("A")) ||
       (WScript.Arguments.Named.Exists("a"))) {
       WshShell.Run("wscript PhoneBook.wsf //Job:AddRec");
       WScript.Quit();
      }
      if ((WScript.Arguments.Named.Exists("D")) ||
       (WScript.Arguments.Named.Exists("d"))) {
       WshShell.Run("wscript PhoneBook.wsf //Job:DelRec");
       WScript.Quit();
```

```
  }
  //Ни один из нужных аргументов не был указан, выводим
  //описание параметров
  WScript.Arguments.ShowUsage();
 </script>
</job>
```

Теперь, если понадобится ввести дополнительную функцию при работе с записной книжкой (например, поиск по номеру телефона), нужно будет в файл PhoneBook.wsf добавить задание с новым идентификатором, а в файл ArgMenu.wsf — обработку нового параметра командной строки.

Организация диалога с помощью кнопочного меню

Вторым вариантом организации диалога, который мы рассмотрим, является кнопочное (командное) меню. Принцип его работы в нашем примере остается практически тем же, что и при описанной выше обработке аргументов командной строки — пользователь должен в диалоговом окне ввести символ, соответствующий одной из описанных в этом окне команд (рис. 7.2). Этот символ анализируется в сценарии, и в зависимости от его значения вызывается то или иное задание из файла PhoneBook.wsf.

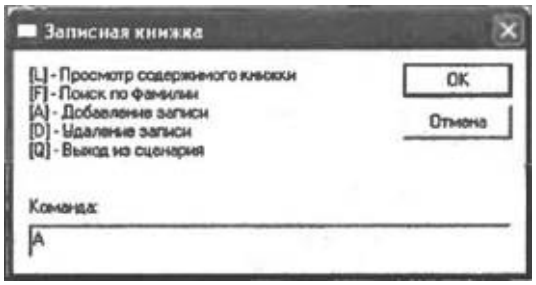


Рис. 7.2. Кнопочное меню для работы с записной книжкой

Диалоговое окно, показанное на рис. 7.2, выводится в цикле `while`, в котором с помощью оператора `switch` анализируется введенный пользователем символ. Выход из цикла совершается, если введенный символ совпадает с "q" или "Q".

Текст сценария ComMenu.wsf, реализующего кнопочное меню для работы с записной книжкой, приводится в листинге 7.5.

Листинг 7.5. Командное меню для работы с записной книжкой

```
<job id="ComMenu">
<script language="VBScript" src="WSHInputBox.vbs"/>
<script language="JScript">
 var WshShell,SMenu,Res;
 WshShell=WScript.CreateObject("WScript.Shell");
 SMenu="[L] - Просмотр содержимого книжки\n";
 SMenu+="[F] - Поиск по фамилии\n";
 SMenu+="[A] - Добавление записи\n";
 SMenu+="[D] - Удаление записи\n";
 SMenu+="[Q] - Выход из сценария\n";
 SMenu+="\n\nКоманда:";
 Res="";
 while ((Res!="q") && (Res!="Q")) {
  Res=WSHInputBox(SMenu,"Записная книжка");
  switch (Res) {
  case "L": {
   WshShell.Run("wscript PhoneBook.wsf //Job:SortName",1,true);
   break;
  }
  case "l": {
   WshShell.Run("wscript PhoneBook.wsf //Job:SortName",1,true);
   break;
  }
  case "F": {
   WshShell.Run("wscript PhoneBook.wsf //Job:FindName",1,true);
   break;
  }
  case "f": {
   WshShell.Run("wscript PhoneBook.wsf //Job:FindName",1,true);
   break;
  }
  case "A": {
   WshShell.Run("wscript PhoneBook.wsf //Job:AddRec",1,true);
   break;
  }
  case "a": {
   WshShell.Run("wscript PhoneBook.wsf //Job:AddRec",1,true);
   break;
  }
  case "D": {
   WshShell.Run("wscript PhoneBook.wsf //Job:DelRec",1,true);
   break;
  }
  case "d": {
   WshShell.Run("wscript PhoneBook.wsf //Job:DelRec",1,true);
   break;
  }
```

```
 }
}
</script>
</job>
```

Однозадачный сценарий для работы с записной книжкой

Как мы видим из вышеприведенных примеров, ни объектная модель WSH, ни языки JScript и VBScript не предоставляют средств для создания полноценного графического интерфейса пользователя.

Тем не менее, такой интерфейс в сценариях WSH создать можно. Мы продемонстрируем это на примере еще одного сценария (состоящего из однозадачного JScript-файла) для работы с записной книжкой, в котором для диалога с пользователем будет организована пользовательская форма с несколькими кнопками и текстовыми полями ввода. Для создания этой формы и работы с ней будут использоваться HTML-файл и браузер Internet Explorer.

Использование Internet Explorer для создания диалоговых окон

Процесс создания сценария WSH, использующего Internet Explorer в качестве интерфейса, можно условно разделить на несколько этапов:

□ создание HTML-формы в отдельном файле;

□ написание функции для сценария WSH, в которой будет производиться вывод на экран построенной формы;

□ написание части сценария, в которой будет реализована необходимая функциональность (например, обмен информацией между формой и внешним файлом с данными, корректное отображение данных в форме и т.д.);

□ добавление в сценарий функций-обработчиков событий, связанных с поведением браузера Internet Explorer;

□ добавление в сценарий функций-обработчиков событий, которые генерируются элементами управления в форме.

Ниже мы рассмотрим каждый из этих этапов на примере создания сценария IEPhoneBook.js для работы с записной книжкой, которая хранится, как и прежде, в XML-файле book.xml.

Разработка HTML-формы для диалогового окна

В качестве интерфейса записной книжки мы создадим диалоговое окно (пользовательскую форму), которое показано на рис. 7.3.

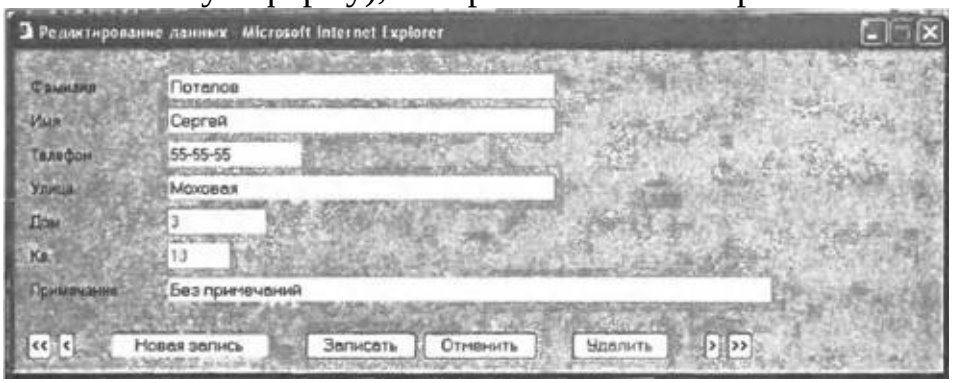


Рис. 7.3. Диалоговое окно для работы с записной книжкой

Эта форма реализуется с помощью HTML-файл Phone.htm, который полностью приведен в листинге 7.6.

В самом начале файла Phone.htm ставится тег `<html>`, указывающий на то, что содержимым файла является текст в формате HTML, а также теги `<head>` и `</head>`, внутри которых задаются используемая кодировка (`charset=windows-1251`) и заголовок формы (теги `<title>` и `</title>`):

```
    <head>
    <meta       http-equiv="Content-Type"       content="text/html;
charset=windows-1251">
    <title>Форма для записной книжки</title>
    </head>
```

Для того чтобы задать цвет формы, в теге `<body>` используется атрибут `bgcolor` со значением `"silver"`:

```
    <body bgcolor="silver" scroll="no">
```

Атрибут `scroll="no"` указывает на то, что в диалоговом окне не должно быть полос прокрутки.

Наша форма состоит из семи текстовых полей ввода (табл. 7.2) и восьми кнопок (табл. 7.3).

Таблица 7.2. *Поля ввода диалогового окна для работы с записной книжкой*

Имя поля	Размер поля (символов)	Назначение
`txtLastName`	50	Поле для ввода фамилии
`txtName`	50	Поле для ввода имени
`txtPhone`	15	Поле для ввода номера телефона
`txtStreet`	50	Поле для ввода названия улицы

txtHouse	10	Поле для ввода номера дома
txtApp	5	Поле для ввода номера квартиры
txtNote	80	Поле для ввода примечания

Таблица 7.3. *Кнопки диалогового окна для работы с записной книжкой*

Текст кнопки	Имя кнопки	Назначение
<<	btnFirst	Переход к первой записи
<	btnPrevious	Переход к предыдущей записи
Новая запись	btnNew	Добавление новой пустой записи
Записать	btnSave	Сохранение сделанных изменений **в** XML-файле
Отменить	btnCancel	Отмена сделанных в форме изменений
Удалить	btnDelete	Удаление текущей записи
>	btnNext	Переход к следующей записи
>>	btnFinal	Переход к последней записи

Команды, создающие форму, находятся внутри тегов `<form>` и `</form>`. Сами текстовые поля ввода и кнопки создаются в HTML-файле с помощью одного и того же тега `<input>`. Внутри этого тега нужно указать несколько атрибутов:

- `type` — определяет тип элемента управления (для поля ввода `type="text"`, для кнопки `type="button"`);
- `name` — задает имя элемента управления;
- `size` — определяет длину строки поля ввода в символах;
- `value` — задает надпись на кнопке.

Для того чтобы поля ввода располагались точно друг под другом, мы поместим их в таблицу с невидимыми границами, состоящую из двух столбцов: в первом находится описание (метка) для поля, во втором — сам элемент управления. Таблица в HTML-файле создается с помощью парных тегов `<table>` и `</table>`, внутри которых приводятся теги `<tr>` и `</tr>`, задающие начало и конец одной строки таблицы соответственно:

```
    <table border="0" width="100%" style="font-family:Arial; font-
size:10pt">
     <tr>
     </tr>
```

```
</table>
```

Здесь аргумент `border` задает ширину границ таблицы (в нашем случае границы невидимы), а в аргументе `style` указываются название и размер шрифта, которым будет выводиться содержимое таблицы.

В свою очередь, внутри тегов `<tr>` и `</tr>` находятся теги `<td>` и `</td>`, определяющие одну ячейку таблицы, например:

```
<tr>
 <td width="15%">Фамилия</td>
   <td width="85%"><input type="text" name="txtLastName"
size="50"></td>
</tr>
```

Для тегов `<td>` указывается аргумент `width`, задающий ширину строки в процентах от общей ширины строки.

Кнопки в форме выводятся друг за другом, нужное расстояние между ними достигается с помощью неразрывных пробелов (escape-последовательность ` `), например:

```
<input type="button" value="<" name="btnPrevious">
     
<input type="button" value="Новая запись" name="btnNew">
     
```

Листинг 7.6. Описание формы в HTML-файле (Phone.htm)

```
<html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=windows-1251">
<title>Форма для записной книжки</title>
</head>
<body bgcolor="silver" scroll="no">
<form name="MainForm">
<table border="0" width="100%" style="font-family:Arial; font-
size:10pt">
 <tr>
  <td width="15%">Фамилия</td>
     <td width="85%"><input type="text" name="txtLastName"
size="50"></td>
 </tr>
 <tr>
  <td>Имя</td>
  <td><input type="text" name="txtName" size="50"></td>
 </tr>
```

```
     <tr>
      <td>Телефон</td>
      <td><input type="text" name="txtPhone" size="15"></td>
     </tr>
     <tr>
      <td>Улица</td>
      <td><input type="text" name="txtStreet" size="50"></td>
     </tr>
     <tr>
      <td>Дом</td>
      <td><input type="text" name="txtHouse" size="10"></td>
     </tr>
     <tr>
      <td>Кв.</td>
      <td><input type="text" name="txtApp" size="5"></td>
     </tr>
     <tr>
      <td>Примечание</td>
      <td><input type="text" name="txtNote" size="80"></td>
     </tr>
    </table>
    <br>
    <input type="button" value="<<" name="btnFirst">
    <input                  type="button"                 value="<"
name="btnPrevious">     
    <input        type="button"        value="Новая        запись"
name="btnNew">     
    <input type="button" value="Записать" name="btnSave">
    <input              type="button"              value="Отменить"
name="btnCancel">     
    <input               type="button"               value="Удалить"
name="btnDelete">     
    <input type="button" value=">" name="btnNext">
    <input type="button" value=">>" name="btnFinal">
    </form>
    </body>
    </html>
```

Создание объекта для обмена данными между XML-файлом и формой

В отличие от рассмотренного выше сценария PhoneBook.wsf, в сценарии IEPhoneBook.js функции для работы с записной книжкой не будут разделены по разным заданиям, поэтому для более четкой организации сценария мы воспользуемся объектно-ориентированным подходом и

создадим два объекта Person и ListPersons, методы которых и будут осуществлять обработку данных и связь между XML-файлом и пользовательской формой.

Как и раньше, в свойствах объекта Person будет храниться запись об одном человеке. Кроме этого, мы добавим в объект Person метод LoadDialog, который будет заполнять поля ввода в форме данными из соответствующих свойств объекта Person:

```
//Конструктор объекта Person
function Person() {
 //Инициализируем свойства объекта
 this.LastName="";
 this.Name="";
 this.Phone="";
 this.Street="";
 this.House="";
 this.App="";
 this.Note="";
 //Устанавливаем для метода LoadDialog указатель на
 //функцию Person_LoadDialog
 this.LoadDialog=Person_LoadDialog;
}
//Заполнение полей в форме для текущей записи
function Person_LoadDialog() {
 //Заполняем поля ввода в форме значениями соответствующих
 //свойств объекта Person
 doc.all.txtLastName.value = this.LastName;
 doc.all.txtName.value=this.Name;
 doc.all.txtPhone.value=this.Phone;
 doc.all.txtStreet.value=this.Street;
 doc.all.txtHouse.value=this.House;
 doc.all.txtApp.value = this.App;
 doc.all.txtNote.value = this.Note;
}
```

Принцип доступа к полям ввода формы по их именам, который используется в методе LoadDialog(), объясняется ниже *(см. разд. "Обработка событий, генерируемых элементами управления формы").*

Основным объектом, который обеспечивает обмен данными между XML-файлом записной книжки и разработанной нами формой, является объект ListPersons. Этот объект будет содержать три свойства и десять методов.

Первым свойством объекта ListPersons мы сделаем массив PersonArr объектов Person; этот массив будет служить промежуточным буфером при чтении данных из XML-файла для отображения в форме и при записи

измененных данных из формы в файл. В остальных двух свойствах `CurRecord` и `IsChanged` объекта `ListPersons` будут соответственно храниться номер текущей записи и логическое значение (`true` или `false`), являющееся признаком того, были ли изменены пользователем данные в форме.

Назначение методов объекта `ListPersons` ясно из комментариев, которые приведены в конструкторе этого объекта (листинг 7.7).

Листинг 7.7. Конструктор объекта `ListPersons`

```
function ListPersons() {
 //          Свойства объекта
 //Создаем массив PersonArr экземпляров объекта Person
 this.PersonArr = new Array();
 //Инициализируем номер текущей записи
 this.CurRecord = 0;
 //Сбрасываем признак изменения данных в форме
 this.IsChanged = false;
 //          Методы объекта
  //Устанавливаем для методов указатели на соответствующие функции
 this.FileToArray=ListPersons_FileToArray;
 this.SaveData=ListPersons_SaveData;
 this.LoadDialog=ListPersons_LoadDialog;
 this.RefreshDialog=ListPersons_RefreshDialog;
 this.NextRecord=ListPersons_NextRecord;
 this.PreviousRecord=ListPersons_PreviousRecord;
 this.FirstRecord=ListPersons_FirstRecord;
 this.FinalRecord=ListPersons_FinalRecord;
 this.NewRecord=ListPersons_NewRecord;
 this.DelRecord = ListPersons_DelRecord;
}
```

Текст всех методов объекта `ListPersons` с подробными комментариями приведен в листинге 7.8.

Листинг 7.8. Методы объекта `ListPersons`

```
//Считывание данных из XML-файла в массив объектов Person
function ListPersons_FileToArray() {
 var Root,CurrNode,i;
 //Создаем объект XML DOM
```

```
 XML = WScript.CreateObject("Msxml.DOMDocument");
 //Загружаем XML-документ из файла
 XML.load(PathBook);
 //Сохраняем в переменной Root ссылку на корневой элемент
 //документа
 Root=XML.documentElement;
 //Обнуляем массив PersonArr
 this.PersonArr.length=0;
 //Перебираем все дочерние элементы первого уровня вложенности
 //для корневого элемента
 for (i=1; i<=Root.childNodes.length-1;i++) {
  //Выделяем в коллекции XML-элементов i-й элемент
  CurrNode=Root.childNodes.item(i);
  //Создаем новый экземпляр PersonRec объекта Person
  PersonRec=new Person();
  //Заполняем поля объекта PersonRec
  PersonRec.LastName=GetTagVal(CurrNode,"LastName");
  PersonRec.Name=GetTagVal(CurrNode,"Name");
  PersonRec.Phone=GetTagVal(CurrNode,"Phone");
  PersonRec.Street=GetTagVal(CurrNode,"Street");
  PersonRec.House=GetTagVal(CurrNode,"House");
  PersonRec.App=GetTagVal(CurrNode,"App");
  PersonRec.Note=GetTagVal(CurrNode,"Note");
  //Сохраняем объект PersonRec в массиве PersonArr
  this.PersonArr[this.PersonArr.length]=PersonRec;
 }
}
//Запись данных из формы в XML-файл
function ListPersons_SaveData() {
 var Root,CurrNode,ElemList;
 //Сохраняем в переменной Root ссылку на корневой элемент
 //документа
 Root=XML.documentElement;
 //Сохраняем в переменной CurrNode ссылку на (CurRecord+1)-й
 //элемент Person
 CurrNode=Root.childNodes.item(this.CurRecord+1);
 //Записываем данные из полей ввода формы в соответствующие
 //XML-элементы, которые являются дочерними узлами
 //относительно  CurrNode
 SetTagVal(CurrNode,"LastName",doc.all.txtLastName.value);
 SetTagVal(CurrNode,"Name",doc.all.txtName.value);
 SetTagVal(CurrNode,"Phone",doc.all.txtPhone.value);
 SetTagVal(CurrNode,"Street",doc.all.txtStreet.value);
 SetTagVal(CurrNode,"House",doc.all.txtHouse.value);
 SetTagVal(CurrNode,"App",doc.all.txtApp.value);
 SetTagVal(CurrNode,"Note",doc.all.txtNote.value);
 //Сохраняем XML-файл на диске
```

```
 XML.save(PathBook);
}
//Загрузка данных для текущей записи в форму
function ListPersons_LoadDialog() {
 //Вызываем метод LoadDialog для объекта Person,
 //который является CurRecord-м элементом массива PersonArr
 this.PersonArr[this.CurRecord].LoadDialog();
}
//Обновление данных в форме
function ListPersons_RefreshDialog(IsGoTop) {
 //Обнуляем массив PersonArr
 this.PersonArr.length=0;
 //Заново загружаем данные из XML-файла в массив PersonArr
 this.FileToArray();
 if (IsGoTop)
  //Переходим к первой записи в массиве
  this.FirstRecord()
 else
  //Переходим к последней записи в массиве
  this.FinalRecord();
 //Загружаем в форму данные для текущей записи
 this.LoadDialog();
}
//Переход к следующей записи
function ListPersons_NextRecord() {
 if (this.CurRecord<this.PersonArr.length - 1)
  //Если текущая запись не является последней, увеличиваем
  //номер текущей записи
  this.CurRecord++;
}
//Переход к предыдущей записи
function ListPersons_PreviousRecord() {
 if (this.CurRecord > 0)
  //Если текущая запись не является первой, уменьшаем
  //номер текущей записи
  this.CurRecord--;
}
//Переход к первой записи
function ListPersons_FirstRecord() {
 this.CurRecord = 0;
}
//Переход к последней записи
function ListPersons_FinalRecord() {
 this.CurRecord = this.PersonArr.length - 1;
}
//Добавление новой записи
function ListPersons_NewRecord() {
```

```
     var Root,NewElem;
     //Сохраняем в переменной Root ссылку на корневой элемент
     //документа
     Root=XML.documentElement;
     //Создаем новый элемент Person
     NewElem=XML.createElement("Person");
     //Добавляем новый элемент в XML-файл
     Root.appendChild(NewElem);
     //Сохраняем XML-файл на диске
     XML.save(PathBook);
     //Обновлем в форме данные для последней добавленной записи
     this.RefreshDialog(false);
    }
    //Удаление текущей записи
    function ListPersons_DelRecord() {
     var Root,DelNom;
     //Сохраняем в переменной Root ссылку на корневой элемент
     //документа
     Root=XML.documentElement;
      //В переменной DelNom сохраняем номер удаляемого элемента
Person
     DelNom=this.CurRecord+1;
     //Удаляем DelNom-й элемент Person из XML-файла
     Root.removeChild(Root.childNodes.item(DelNom))
     //Сохраняем XML-файл на диске
     XML.save(PathBook);
     //Выводим сообщение о том, что запись удалена
     WshShell.Popup("Запись N "+DelNom+" удалена",0,"Информация",
      vbInformation+vbOkOnly);
     //Обновлем в форме данные для первой записи
     this.RefreshDialog(true);
    }
```

Вывод формы из сценария WSH

Для того чтобы вывести из сценария WSH разработанную HTML-форму на экран, нужно вначале получить ссылку на объект `Application`, который определяется в объектной модели Internet Explorer. Делается это следующим образом:

```
    var ie = WScript.CreateObject("InternetExplorer.Application",
"ie_");
```

При этом в память загружается новый экземпляр Internet Explorer, а ссылка на этот объект присваивается переменной `ie` (само окно браузера по умолчанию невидимо, для его отображения на экране необходимо

установить свойство Visible объекта Application в 1). В качестве второго параметра метода CreateObject указан префикс "ie_", посредством которого мы сможем написать функции-обработчики событий Internet Explorer.

Внешний вид браузера Internet Explorer настраивается с помощью нескольких свойств объекта Application:

```
//Устанавливаем свойства объекта ie для отображения формы
ie.AddressBar = false; //Адресная строка не выводится
ie.Fullscreen = false; //Полноэкранный режим запрещен
ie.MenuBar = false; //Главное меню браузера не выводится
ie.Resizable = false; //Изменять размеры окна нельзя
ie.StatusBar = false; //Строка статуса не выводится
ie.ToolBar = false; //Инструментальная панель не выводится
//Устанавливаем размеры окна
ie.Height = 300; //Высота
ie.Width = 780; //Длина
```

Для того чтобы загрузить в браузер нужный нам файл Phone.htm с описанием формы, используется метод Navigate объекта Application:

```
ie.Navigate(PathHTML);
```

В качестве параметра метода Navigate указывается путь к файлу Phone.htm, который заранее устанавливается в функции InitPath():

```
function InitPath() {
 BasePath=WshShell.CurrentDirectory+"\\";
 //Путь к файлу с данными
 PathBook=BasePath+"book.xml";
 //Путь к файлу с HTML-формой
 PathHTML=BasePath+"Phone.htm";
}
```

Далее следует учесть, что сценарий WSH и окно браузера, в котором загружена форма, — это два независимых процесса. Поэтому в сценарии после загрузки формы в окно браузера необходимо дождаться, пока пользователь не закроет это окно. Для этого мы присвоим глобальной переменной IsQuit значение false и заставим сценарий выполняться до тех пор, пока значение этой переменной не станет равным true:

```
while (!IsQuit)
 //Приостанавливаем сценарий на 0,1 сек
 WScript.Sleep(100);
```

При закрытии формы будет генерироваться событие OnQuit объекта Application, поэтому мы напишем функцию-обработчик ie_OnQuit() этого события, в которой будем устанавливать isQuit в true и сохранять в XML-файле данные, которые были изменены в форме (листинг 7.9).

Листинг 7.9. Функция-обработчик закрытия окна браузера

```
function ie_OnQuit() {
 IsQuit=true;
 //Сохраняем данные из формы в XML-файле
 objListPersons.SaveData();
}
```

В нашем сценарии загрузка в браузер HTML-файла с формой будет производиться в основной запускной функции `Main()` (листинг 7.10).

Листинг 7.10. Функция Main()

```
function Main() {
 //Создаем объект WshShell
 WshShell = WScript.CreateObject("WScript.Shell");
 //Определяем пути к файлам
 InitPath();
 //Создаем объект InternetExplorer.Application с возможностью
 //обработки событий этого объекта
  ie    =  WScript.CreateObject("InternetExplorer.Application",
"ie_");
 //Устанавливаем свойства объекта ie для отображения формы3
 ie.AddressBar = false;
 ie.FullScreen = false;
 ie.MenuBar    = false;
 ie.Resizable  = false;
 ie.StatusBar  = false;
 ie.ToolBar    = false;
 //Устанавливаем размеры окна
 ie.Height = 300;  //Высота
 ie.Width = 780;   //Длина
 IsQuit=false;
 //Загружаем HTML-файл с формой
 ie.Navigate(PathHTML);
 while (!IsQuit)
  //Приостанавливаем сценарий на 0,1 сек
  WScript.Sleep(100);
}
```

После окончания загрузки в браузер HTML-файла с формой нужно считать информацию из XML-файла с данными и отобразить в форме данные для первой записи. Мы будем это делать в функции-обработчике `ie_DocumentComplete()` события `DocumentComplete` объекта `Application`,

которое генерируется как раз после окончания загрузки документа в браузер (листинг 7.11).

Листинг 7.11. Функция-обработчик окончания загрузки документа в браузер

```
function ie_DocumentComplete() {
 //Создаем экземпляр objListPersons объекта ListPersons
 objListPersons = new ListPersons();
 //Загружаем данные из XML-файла в массив PersonArr
 objListPersons.FileToArray();
 //Получаем ссылку на объект Document
 doc = ie.Document;
 //Устанавливаем заголовок окна
 doc.title = "Редактирование данных";
 //Указываем функции-обработчики нажатий на кнопки формы
 doc.all.btnSave.onclick=btnSave_OnClick;
 doc.all.btnCancel.onclick=btnCancel_OnClick;
 doc.all.btnFirst.onclick=btnFirst_OnClick;
 doc.all.btnPrevious.onclick=btnPrevious_OnClick;
 doc.all.btnNew.onclick=btnNew_OnClick;
 doc.all.btnDelete.onclick=btnDelete_OnClick;
 doc.all.btnNext.onclick=btnNext_OnClick;
 doc.all.btnFinal.onclick=btnFinal_OnClick;

 //Указываем функции-обработчики изменения текста в полях ввода
 doc.all.txtLastName.onchange  = txtBoxOnChange;
 doc.all.txtName.onchange  = txtBoxOnChange;
 doc.all.txtPhone.onchange = txtBoxOnChange;
 doc.all.txtStreet.onchange  = txtBoxOnChange;
 doc.all.txtHouse.onchange = txtBoxOnChange;
 doc.all.txtApp.onchange  = txtBoxOnChange;
 doc.all.txtNote.onchange  = txtBoxOnChange;
 if (objListPersons.PersonArr.length < 1)
  //Если в XML-файле нет данных, добавляем пустую запись
  objListPersons.AddRecord();
 //В качестве текущей устанавливаем первую запись
 objListPersons.CurRecord = 0;
 //Загружаем в форму данные из массива PersonArr
 //для первой записи
 objListPersons.LoadDialog();
 // Делаем окно Internet Explorer'а видимым
 ie.Visible = true;
}
```

В функции `ie_DocumentComplete()`, кроме прочего, задаются функции-обработчики событий, генерируемых в форме текстовыми полями ввода и кнопками. К описанию процесса обработки таких событий мы и перейдем.

Обработка событий, генерируемых элементами управления формы

В нашем сценарии мы будем обрабатывать события, связанные с нажатием на кнопки в форме и с изменением текста в полях ввода. Для этого нужно, во-первых, получить ссылку на соответствующий элемент управления в форме, зная его имя, которое задается атрибутом name в HTML-файле, например:

```
<input type="button" value="<<" name="btnFirst">
```

Для доступа к элементу управления используется объект Document, который соответствует загруженному в браузер HTML-документу. Ссылка на объект Document Хранится в свойстве Document объекта Application:

```
//Получаем ссылку на объект Document
doc = ie.Document;
```

Обработчики событий для элементов управления формы указываются тогда следующим образом:

```
doc.all.ControlName.EventName=FunctionName;
```

Здесь *`ControlName`* — имя элемента управления, *`EventName`* — имя обрабатываемого события, *`FunctionName`* — имя функции-обработчика, которая будет вызываться при наступлении события *`EventName`*. Событие, возникающее при нажатии на кнопку в форме, называется `onclick`, а событие, происходящее при изменении текста в поле ввода, — `onchange`:

```
//Указываем функции-обработчики нажатий на кнопки формы
doc.all.btnSave.onclick=btnSave_OnClick;
doc.all.btnCancel.onclick=btnCancel_OnClick;
doc.all.btnFirst.onclick=btnFirst_OnClick;
doc.all.btnPrevious.onclick=btnPrevious_OnClick;
doc.all.btnNew.onclick=btnNew_OnClick;
doc.all.btnDelete.onclick=btnDelete_OnClick;
doc.all.btnNext.onclick=btnNext_OnClick;
doc.all.btnFinal.onclick=btnFinal_OnClick;
//Указываем функции-обработчики изменения текста в полях ввода
doc.all.txtLastName.onchange = txtBoxOnChange;
doc.all.txtName.onchange = txtBoxOnChange;
doc.all.txtPhone.onchange = txtBoxOnChange;
doc.all.txtStreet.onchange = txtBoxOnChange;
doc.all.txtHouse.onchange = txtBoxOnChange;
doc.all.txtApp.onchange = txtBoxOnChange;
```

```
doc.all.txtNote.onchange = txtBoxOnChange;
```

Сами функции-обработчики нажатий на различные кнопки и изменения текста в полях ввода приведены с подробными комментариями в листинге 7.12.

Листинг 7.12. Функции-обработчики нажатия кнопок и изменения поло ввода

```
//Функция-обработчик нажатия на кнопку "Сохранить"
function btnSave_OnClick() {
 //Сохраняем данные из формы в XML-файле
 objListPersons.SaveData();
}
//Функция-обработчик нажатия на кнопку "Отменить"
function btnCancel_OnClick() {
 //Заново загружаем данные из текущего элемента массива
 //в форму
 objListPersons.LoadDialog();
 //Выводим сообщение в заголовке окна
 doc.title = "Данные восстановлены";
}
//Функция-обработчик нажатия на кнопку "<<"
function btnFirst_OnClick() {
 //Проверяем, были ли сделаны изменения в форме
 if (objListPersons.IsChanged) { //Изменения были сделаны
  //Сохраняем данные в XML-файле
  objListPersons.SaveData();
  //Заново загружаем данные из XML-файла в массив PersonArr
  objListPersons.FileToArray();
 }
 //Переходим к первой записи в массиве
 objListPersons.FirstRecord();
 //Загружаем в форму данные из массива PersonArr
 //для первой записи
 objListPersons.LoadDialog();
 //Выводим сообщение в заголовке окна
 doc.title = "Запись N " + (objListPersons.CurRecord + 1);
}
//Функция-обработчик нажатия на кнопку "<"
function btnPrevious_OnClick() {
 //Проверяем, были ли сделаны изменения в форме
 if (objListPersons.IsChanged) {  //Изменения были сделаны
  //Сохраняем данные в XML-файле
  objListPersons.SaveData();
```

```
  //Заново загружаем данные из XML-файла в массив PersonArr
  objListPersons.FileToArray();
 }
 //Переходим к предыдущей записи в массиве
 objListPersons.PreviousRecord();
 //Загружаем в форму данные из массива PersonArr
 //для текущей записи
 objListPersons.LoadDialog();
 //Выводим сообщение в заголовке окна
 doc.title = "Запись N " + (objListPersons.CurRecord + 1);
}
//Функция-обработчик нажатия на кнопку "Новая запись"
function btnNew_OnClick() {
 //Проверяем, были ли сделаны изменения в форме
 if (objListPersons.IsChanged) { //Изменения были сделаны
  //Сохраняем данные в XML-файле
  objListPersons.SaveData();
  //Заново загружаем данные из XML-файла в массив PersonArr
  objListPersons.FileToArray();
 }
 //Добавляем новую запись в XML-файл
 objListPersons.NewRecord();
 //Загружаем в форму данные из массива PersonArr
 //для добавленной записи
 objListPersons.LoadDialog();
 //Выводим сообщение в заголовке окна
 doc.title = "Добавлена новая запись";
}
//Функция-обработчик нажатия на кнопку "Удалить"
function btnDelete_OnClick() {
 //Удаляем текущую запись из XML-файла
 objListPersons.DelRecord();
 //Загружаем в форму данные из массива PersonArr
 //для первой записи
 objListPersons.LoadDialog();
 //Выводим сообщение в заголовке окна
 doc.title = "Запись удалена";
}
//Функция-обработчик нажатия на кнопку ">"
function btnNext_OnClick() {
 //Проверяем, были ли сделаны изменения в форме
 if (objListPersons.IsChanged) { //Изменения были сделаны
  //Сохраняем данные в XML-файле
  objListPersons.SaveData();
  //Заново загружаем данные из XML-файла в массив PersonArr
  objListPersons.FileToArray();
 }
```

```
 //Переходим к следующей записи в массиве
 objListPersons.NextRecord();
 //Загружаем в форму данные из массива PersonArr
 //для текущей записи
 objListPersons.LoadDialog();
 //Выводим сообщение в заголовке окна
 doc.title = "Запись N " + (objListPersons.CurRecord + 1);
}
//Функция-обработчик нажатия на кнопку ">>"
function btnFinal_OnClick() {
 //Проверяем, были ли сделаны изменения в форме
 if (objListPersons.IsChanged) { //Изменения были сделаны
  //Сохраняем данные в XML-файле
  objListPersons.SaveData();
  //Заново загружаем данные из XML-файла в массив PersonArr
  objListPersons.FileToArray();
 }
 //Переходим к последней записи в массиве
 objListPersons.FinalRecord();
 //Загружаем в форму данные из массива PersonArr
 //для текущей записи
 objListPersons.LoadDialog();
 //Выводим сообщение в заголовке окна
 doc.title = "Запись N " + (objListPersons.CurRecord + 1);
}
//Функция-обработчик изменения текста в полях ввода
function txtBoxOnChange() {
 //Устанавливаем признак изменения данных в форме
 objListPersons.IsChanged = true;
 //Выводим сообщение в заголовке окна
 doc.title = "Редактирование данных";
}
```

Окончательная доработка сценария IEPhoneBook.js

Выше были описаны все основные функции, которые используются для работы с записной книжкой в диалоговом режиме. Осталось лишь собрать эти функции в один JScript-сценарий IEPhoneBook.js, определить глобальные переменные и добавить вспомогательные функции `GetTagVal(`*`obj, tgName`*`)` и `SetTagVal(`*`obj, tgName, sVal`*`)` для доступа к значениям XML-элементов (листинг 7.13).

Листинг 7.13. Функция для доступа к значениям XML-элементов

```
//Определение значения тега tgName XML-элемента obj
function GetTagVal(obj, tgName) {
 var ElemList;
 //Создаем коллекцию дочерних для obj элементов, которые
 //задаются тегом tgName
 ElemList=obj.getElementsByTagName(tgName);
 //Проверяем, есть ли в коллекции ElemList элементы
 if (ElemList.length>0)
  //Возвращаем значение тега tgName
  return ElemList.item(0).text
 else return "";
}
//Изменение значения тега tgName XML-элемента obj
function SetTagVal(obj, tgName, sVal) {
 var ElemList,New;
 //Создаем коллекцию дочерних для obj элементов, которые
 //задаются тегом tgName
 ElemList=obj.getElementsByTagName(tgName);
 //Проверяем, есть ли в коллекции ElemList элементы
 if (ElemList.length>0)
  //Устанавливаем значение элемента, задаваемого
  //тегом tgName
  ElemList.item(0).text=sVal;
 else {
  //Создаем новый элемент с именем tgName
  NewElem=XML.createElement(tgName);
  //Добавляем новый элемент в качестве дочернего для
  //элемента obj
  obj.appendChild(NewElem);
  //Устанавливаем значение добавленного элемента
  obj.lastChild.text=sVal;
 }
}
```

Полный текст сценария IEPhoneBook.js приведен в листинге 7.14.

Листинг 7.14. Сценарий IEPhoneBook.js

```
/*****************************************************************
/*                                                          Имя:
IEPhoneBook.js                                            */
/*                                                         Язык:
JScript                                                  */
/*    Описание:    Сценарий    для    работы    с    записной
книжкой                 */
/*                    (графический  интерфейс  пользователя  на
```

```
основе         */
    /*                                                          HTML-
формы).                                          */
    /*****************************************************************
    //Объявляем глобальные переменные
    var
     WshShell,
     PathBook,       //Путь к файлу с данными
     PathHTML,       //Путь к HTML-файлу с формой
     XML,            //Экземпляр объекта XML DOM
       ie,                                     //Экземпляр   объекта
InternetExplorer.Application
     doc,            //Экземпляр объекта Document
     IsQuit,         //Признак выхода из сценария
     objListPersons; //Экземпляр объекта ListPersons
    //Инициализируем константы для диалоговых окон
    var vbInformation=64,vbOkOnly=0;
    //Построение путей к файлам
    function InitPath() {
     BasePath=WshShell.CurrentDirectory+"\\";
     //Путь к файлу с данными
     PathBook=BasePath+"book.xml";
     //Путь к файлу с HTML-формой
     PathHTML=BasePath+"Phone.htm";
    }
    //Определение значения тега tgName XML-элемента obj
    function GetTagVal(obj, tgName) {
     var ElemList;
     //Создаем коллекцию дочерних для obj элементов, которые
     //задаются тегом tgName
     ElemList=obj.getElementsByTagName(tgName);
     //Проверяем, есть ли в коллекции ElemList элементы
     if (ElemList.length>0)
      //Возвращаем значение тега tgName
      return ElemList.item(0).text
     else return "";
    }
    //Изменение значения тега tgName XML-элемента obj
    function SetTagVal(obj, tgName, sVal) {
     var ElemList,New;
     //Создаем коллекцию дочерних для obj элементов, которые
     //задаются тегом tgName
     ElemList=obj.getElementsByTagName(tgName);
     //Проверяем, есть ли в коллекции ElemList элементы
     if (ElemList.length>0)
      //Устанавливаем значениеэлемента, задаваемого
      //тегом tgName
```

```
  ElemList.item(0).text=sVal;
 else {
  //Создаем новый элемент с именем tgName
  NewElem=XML.createElement(tgName);
  //Добавляем новый элемент в качестве дочернего для
  //элемента obj
  obj.appendChild(NewElem);
  //Устанавливаем значение добавленного элемента
  obj.lastChild.text=sVal;
 }
}
//Конструктор объекта Person
function Person() {
 //Инициализируем свойства объекта
 this.LastName="";
 this.Name="";
 this.Phone="";
 this.Street="";
 this.House="";
 this.App="";
 this.Note="";
 //Устанавливаем для метода LoadDialog указатель на
 //функцию Person_LoadDialog
 this.LoadDialog=Person_LoadDialog;
}
//Заполнение полей в форме для текущей записи
function Person_LoadDialog() {
 //Заполняем поля ввода в форме значениями соответствующих
 //свойств объекта Person
 doc.all.txtLastName.value = this.LastName;
 doc.all.txtName.value=this.Name;
 doc.all.txtPhone.value=this.Phone;
 doc.all.txtStreet.value=this.Street;
 doc.all.txtHouse.value=this.House;
 doc.all.txtApp.value = this.App;
 doc.all.txtNote.value = this.Note;
}
//Конструктор объекта ListPersons
function ListPersons() {
 //          Свойства объекта
 //Создаем массив PersonArr экземпляров объекта Person
 this.PersonArr = new Array();
 //Инициализируем номер текущей записи
 this.CurRecord = 0;
 //Сбрасываем признак изменения данных в форме
 this.IsChanged = false;
 //          Методы объекта
```

```
      //Устанавливаем для методов указатели на соответствующие
функции
     this.FileToArray=ListPersons_FileToArray;
     this.SaveData=ListPersons_SaveData;
     this.LoadDialog=ListPersons_LoadDialog;
     this.RefreshDialog=ListPersons_RefreshDialog;
     this.NextRecord=ListPersons_NextRecord;
     this.PreviousRecord=ListPersons_PreviousRecord;
     this.FirstRecord=ListPersons_FirstRecord;
     this.FinalRecord=ListPersons_FinalRecord;
     this.NewRecord=ListPersons_NewRecord;
     this.DelRecord = ListPersons_DelRecord;
    }
    //Считывание данных из XML-файла в массив объектов Person
    function ListPersons_FileToArray() {
     var Root,CurrNode,i;
     //Создаем объект XML DOM
     XML = WScript.CreateObject("Msxml.DOMDocument");
     //Загружаем XML-документ из файла
     XML.load(PathBook);
     //Сохраняем в переменной Root ссылку на корневой элемент
     //документа
     Root=XML.documentElement;
     //Обнуляем массив PersonArr
     this.PersonArr.length=0;
     //Перебираем все дочерние элементы первого уровня вложенности
     //для корневого элемента
     for (i=1; i<=Root.childNodes.length-1;i++) {
      //Выделяем в коллекции XML-элементов i-й элемент
      CurrNode=Root.childNodes.item(i);
      //Создаем новый экземпляр PersonRec объекта Person
      PersonRec=new Person();
      //Заполняем поля объекта PersonRec
      PersonRec.LastName=GetTagVal(CurrNode,"LastName");
      PersonRec.Name=GetTagVal(CurrNode,"Name");
      PersonRec.Phone=GetTagVal(CurrNode,"Phone");
      PersonRec.Street=GetTagVal(CurrNode,"Street");
      PersonRec.House=GetTagVal(CurrNode,"House");
      PersonRec.App=GetTagVal(CurrNode,"App");
      PersonRec.Note=GetTagVal(CurrNode,"Note");
      //Сохраняем объект PersonRec в массиве PersonArr
      this.PersonArr[this.PersonArr.length]=PersonRec;
     }
    }
    //Запись данных из формы в XML-файл
    function ListPersons_SaveData() {
     var Root,CurrNode,ElemList;
```

```
 //Сохраняем в переменной Root ссылку на корневой элемент
 //документа
 Root=XML.documentElement;
 //Сохраняем в переменной CurrNode ссылку на (CurRecord+1)-й
 //элемент Person
 CurrNode=Root.childNodes.item(this.CurRecord+1);
 //Записываем данные из полей ввода формы в соответствующие
 //XML-элементы, которые являются дочерними узлами
 //относительно  CurrNode
 SetTagVal(CurrNode,"LastName",doc.all.txtLastName.value);
 SetTagVal(CurrNode,"Name",doc.all.txtName.value);
 SetTagVal(CurrNode,"Phone",doc.all.txtPhone.value);
 SetTagVal(CurrNode,"Street",doc.all.txtStreet.value);
 SetTagVal(CurrNode,"House",doc.all.txtHouse.value);
 SetTagVal(CurrNode,"App",doc.all.txtApp.value);
 SetTagVal(CurrNode,"Note",doc.all.txtNote.value);
 //Сохраняем XML-файл на диске
 XML.save(PathBook);
}
//Загрузка данных для текущей записи в форму
function ListPersons_LoadDialog() {
 //Вызываем метод LoadDialog для объекта Person,
 //который является CurRecord-м элементом массива PersonArr
 this.PersonArr[this.CurRecord].LoadDialog();
}
//Обновление данных в форме
function ListPersons_RefreshDialog(IsGoTop) {
 //Обнуляем массив PersonArr
 this.PersonArr.length=0;
 //Заново загружаем данные из XML-файла в массив PersonArr
 this.FileToArray();
 if (IsGoTop)
  //Переходим к первой записи в массиве
  this.FirstRecord()
 else
  //Переходим к последней записи в массиве
  this.FinalRecord();
 //Загружаем в форму данные для текущей записи
 this.LoadDialog();
}
//Переход к следующей записи
function ListPersons_NextRecord() {
 if (this.CurRecord<this.PersonArr.length - 1)
  //Если текущая запись не является последней, увеличиваем
  //номер текущей записи
  this.CurRecord++;
}
```

```
    //Переход к предыдущей записи
    function ListPersons_PreviousRecord() {
     if (this.CurRecord > 0)
      //Если текущая запись не является первой, уменьшаем
      //номер текущей записи
      this.CurRecord--;
    }
    //Переход к первой записи
    function ListPersons_FirstRecord() {
     this.CurRecord = 0;
    }
    //Переход к последней записи
    function ListPersons_FinalRecord() {
     this.CurRecord = this.PersonArr.length - 1;
    }
    //Добавление новой записи
    function ListPersons_NewRecord() {
     var Root,NewElem;
     //Сохраняем в переменной Root ссылку на корневой элемент
     //документа
     Root=XML.documentElement;
     //Создаем новый элемент Person
     NewElem=XML.createElement("Person");
     //Добавляем новый элемент в XML-файл
     Root.appendChild(NewElem);
     //Сохраняем XML-файл на диске
     XML.save(PathBook);
     //Обновлем в форме данные для последней добавленной записи
     this.RefreshDialog(false);
    }
    //Удаление текущей записи
    function ListPersons_DelRecord() {
     var Root,DelNom;
     //Сохраняем в переменной Root ссылку на корневой элемент
     //документа
     Root=XML.documentElement;
      //В переменной DelNom сохраняем номер удаляемого элемента
Person
     DelNom=this.CurRecord+1;
     //Удаляем DelNom-й элемент Person из XML-файла
     Root.removeChild(Root.childNodes.item(DelNom))
     //Сохраняем XML-файл на диске
     XML.save(PathBook);
     //Выводим сообщение о том, что запись удалена
     WshShell.Popup("Запись N "+DelNom+" удалена",0,"Информация",
      vbInformation+vbOkOnly);
     //Обновлем в форме данные для первой записи
```

```
     this.RefreshDialog(true);
    }
    //             Обработчики событий Internet Explorer'a
    //Функция-обработчик окончания загрузки документа в Internet
Explorer
    function ie_DocumentComplete() {
     //Создаем экземпляр objListPersons объекта ListPersons
     objListPersons = new ListPersons();
     //Загружаем данные из XML-файла в массив PersonArr
     objListPersons.FileToArray();
     //Получаем ссылку на объект Document
     doc = ie.Document;
     //Устанавливаем заголовок окна
     doc.title = "Редактирование данных";
     //Указываем функции-обработчики нажатий на кнопки формы
     doc.all.btnSave.onclick=btnSave_OnClick;
     doc.all.btnCancel.onclick=btnCancel_OnClick;
     doc.all.btnFirst.onclick=btnFirst_OnClick;
     doc.all.btnPrevious.onclick=btnPrevious_OnClick;
     doc.all.btnNew.onclick=btnNew_OnClick;
     doc.all.btnDelete.onclick=btnDelete_OnClick;
     doc.all.btnNext.onclick=btnNext_OnClick;
     doc.all.btnFinal.onclick=btnFinal_OnClick;

     //Указываем функции-обработчики изменения текста в полях ввода
     doc.all.txtLastName.onchange  = txtBoxOnChange;
     doc.all.txtName.onchange  = txtBoxOnChange;
     doc.all.txtPhone.onchange = txtBoxOnChange;
     doc.all.txtStreet.onchange  = txtBoxOnChange;
     doc.all.txtHouse.onchange = txtBoxOnChange;
     doc.all.txtApp.onchange  = txtBoxOnChange;
     doc.all.txtNote.onchange  = txtBoxOnChange;
     if (objListPersons.PersonArr.length < 1)
      //Если в XML-файле нет данных, добавляем пустую запись
      objListPersons.AddRecord();
     //В качестве текущей устанавливаем первую запись
     objListPersons.CurRecord = 0;
     //Загружаем в форму данные из массива PersonArr
     //для первой записи
     objListPersons.LoadDialog();
     // Делаем окно Internet Explorer'a видимым
     ie.Visible = true;
    }
    //Функция-обработчик закрытия окна Internet Explorer'a
    function ie_OnQuit() {
     IsQuit=true;
     //Сохраняем данные из формы в XML-файле
```

```
 objListPersons.SaveData();
}
//              Обработчики нажатий на кнопки в форме
//Функция-обработчик нажатия на кнопку "Сохранить"
function btnSave_OnClick() {
 //Сохраняем данные из формы в XML-файле
 objListPersons.SaveData();
}
//Функция-обработчик нажатия на кнопку "Отменить"
function btnCancel_OnClick() {
 //Заново загружаем данные из текущего элемента массива
 //в форму
 objListPersons.LoadDialog();
 //Выводим сообщение в заголовке окна
 doc.title = "Данные восстановлены";
}
//Функция-обработчик нажатия на кнопку "<<"
function btnFirst_OnClick() {
 //Проверяем, были ли сделаны изменения в форме
 if (objListPersons.IsChanged) { //Изменения были сделаны
  //Сохраняем данные в XML-файле
  objListPersons.SaveData();
  //Заново загружаем данные из XML-файла в массив PersonArr
  objListPersons.FileToArray();
 }
 //Переходим к первой записи в массиве
 objListPersons.FirstRecord();
 //Загружаем в форму данные из массива PersonArr
 //для первой записи
 objListPersons.LoadDialog();
 //Выводим сообщение в заголовке окна
 doc.title = "Запись N " + (objListPersons.CurRecord + 1);
}
//Функция-обработчик нажатия на кнопку "<"
function btnPrevious_OnClick() {
 //Проверяем, были ли сделаны изменения в форме
 if (objListPersons.IsChanged) {  //Изменения были сделаны
  //Сохраняем данные в XML-файле
  objListPersons.SaveData();
  //Заново загружаем данные из XML-файла в массив PersonArr
  objListPersons.FileToArray();
 }
 //Переходим к предыдущей записи в массиве
 objListPersons.PreviousRecord();
 //Загружаем в форму данные из массива PersonArr
 //для текущей записи
 objListPersons.LoadDialog();
```

```
 //Выводим сообщение в заголовке окна
 doc.title = "Запись N " + (objListPersons.CurRecord + 1);
}
//Функция-обработчик нажатия на кнопку "Новая запись"
function btnNew_OnClick() {
 //Проверяем, были ли сделаны изменения в форме
 if (objListPersons.IsChanged) { //Изменения были сделаны
  //Сохраняем данные в XML-файле
  objListPersons.SaveData();
  //Заново загружаем данные из XML-файла в массив PersonArr
  objListPersons.FileToArray();
 }
 //Добавляем новую запись в XML-файл
 objListPersons.NewRecord();
 //Загружаем в форму данные из массива PersonArr
 //для добавленной записи
 objListPersons.LoadDialog();
 //Выводим сообщение в заголовке окна
 doc.title = "Добавлена новая запись";
}
//Функция-обработчик нажатия на кнопку "Удалить"
function btnDelete_OnClick() {
 //Удаляем текущую запись из XML-файла
 objListPersons.DelRecord();
 //Загружаем в форму данные из массива PersonArr
 //для первой записи
 objListPersons.LoadDialog();
 //Выводим сообщение в заголовке окна
 doc.title = "Запись удалена";
}
//Функция-обработчик нажатия на кнопку ">"
function btnNext_OnClick() {
 //Проверяем, были ли сделаны изменения в форме
 if (objListPersons.IsChanged) { //Изменения были сделаны
  //Сохраняем данные в XML-файле
  objListPersons.SaveData();
  //Заново загружаем данные из XML-файла в массив PersonArr
  objListPersons.FileToArray();
 }
 //Переходим к следующей записи в массиве
 objListPersons.NextRecord();
 //Загружаем в форму данные из массива PersonArr
 //для текущей записи
 objListPersons.LoadDialog();
 //Выводим сообщение в заголовке окна
 doc.title = "Запись N " + (objListPersons.CurRecord + 1);
}
```

```
    //Функция-обработчик нажатия на кнопку ">>"
    function btnFinal_OnClick() {
     //Проверяем, были ли сделаны изменения в форме
     if (objListPersons.IsChanged) { //Изменения были сделаны
      //Сохраняем данные в XML-файле
      objListPersons.SaveData();
      //Заново загружаем данные из XML-файла в массив PersonArr
      objListPersons.FileToArray();
     }
     //Переходим к последней записи в массиве
     objListPersons.FinalRecord();
     //Загружаем в форму данные из массива PersonArr
     //для текущей записи
     objListPersons.LoadDialog();
     //Выводим сообщение в заголовке окна
     doc.title = "Запись N " + (objListPersons.CurRecord + 1);
    }
    //Функция-обработчик изменения текста в полях ввода
    function txtBoxOnChange() {
     //Устанавливаем признак изменения данных в форме
     objListPersons.IsChanged = true;
     //Выводим сообщение в заголовке окна
     doc.title = "Редактирование данных";
    }
    //Основная запускная функция
    function Main() {
     //Создаем объект WshShell
     WshShell = WScript.CreateObject("WScript.Shell");
     //Определяем пути к файлам
     InitPath();
     //Создаем объект InternetExplorer.Application с возможностью
     //обработки событий этого объекта
      ie    =  WScript.CreateObject("InternetExplorer.Application",
"ie_");
     //Устанавливаем свойства объекта ie для отображения формы3
     ie.AddressBar = false;
     ie.FullScreen = false;
     ie.MenuBar    = false;
     ie.Resizable  = false;
     ie.StatusBar  = false;
     ie.ToolBar    = false;
     //Устанавливаем размеры окна
     ie.Height = 300;  //Высота
     ie.Width = 780;   //Длина
     IsQuit=false;
     //Загружаем HTML-файл с формой
     ie.Navigate(PathHTML);
```

```
     while (!IsQuit)
      //Приостанавливаем сценарий на 0,1 сек
      WScript.Sleep(100);
    }
    /*******************                                    Начало
*********************************/
    Main();
    /*************                                          Конец
*********************************************/
```

Глава 8
Взаимодействие сценариев с Microsoft Office

Не будет большим преувеличением сказать, что почти на всех компьютерах с операционной системой Windows установлены программы пакета Microsoft Office. Эти приложения являются серверами автоматизации, т.е. их действиями можно управлять из внешних программ, в том числе и из сценариев WSH. В этой главе мы рассмотрим на примерах, каким образом можно выводить из сценариев WSH информацию в две наиболее распространенные офисные программы — Microsoft Word и Microsoft Excel.

Объектные модели Microsoft Word и Excel

Для того чтобы использовать из сценариев WSH те возможности, которые поддерживают программы Word и Excel, необходимо знать, какие именно объекты предоставляются для внешнего использования этими серверами автоматизации и как объекты соотносятся друг с другом. Хотя объектные модели различных приложений Microsoft Office довольно сложны (например, Word содержит порядка 200 взаимосвязанных друг с другом объектов), они очень похожи друг на друга, причем для практических целей достаточно понять принцип работы с несколькими ключевыми объектами. Здесь мы не будем останавливаться на подробном рассмотрении свойств и методов объектов Word и Excel (для желающих глубже познакомиться с этой темой можно порекомендовать, например, замечательную книгу [17]), а лишь кратко упомянем, какие именно объекты будут использоваться в рассмотренных ниже примерах сценариев.

Увидеть структуру объектной модели Word можно воспользовавшись встроенной в Word справкой по Visual Basic (рис. 8.1).

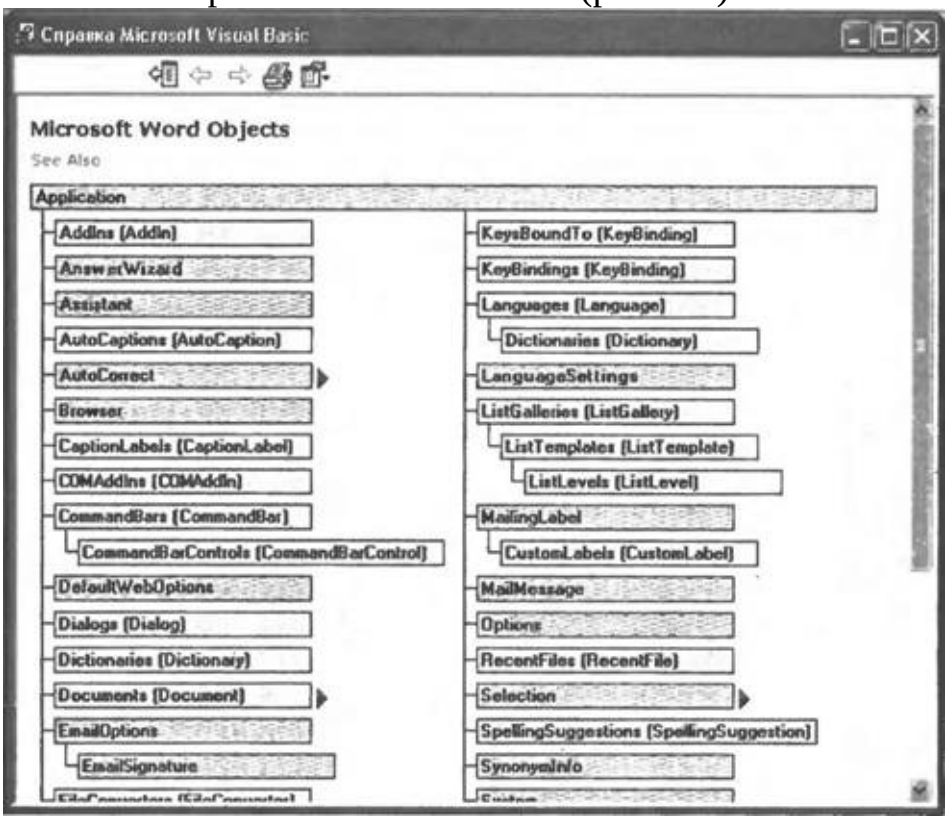


Рис. 8.1. Объектная модель Microsoft Word

Итак, на самом верхнем уровне объектной модели Word находится объект `Application`, который представляет непосредственно само приложение Word и содержит (в качестве свойств) все остальные объекты. Таким образом, объект `Application` используется для получения доступа к любому другому объекту Word.

Семейство `Documents` является свойством объекта `Application` и содержит набор объектов `Document`, каждый из которых соответствует открытому в Word документу. Класс `Documents` понадобится нам в сценариях для создания новых документов. Объект `Document` содержит в качестве своих свойств семейства различных объектов документа; символов (`Characters`), слов (`Words`), предложений (`Sentences`), параграфов (`Paragraphs`) и т.д. В одном из рассмотренных ниже сценариев, например, нам понадобится работать с семейством закладок в документе (`Bookmarks`).

Объект `Selection` позволяет работать с выделенным фрагментом текста (этот фрагмент может быть и пустым). Таким образом, можно сказать, что объект `Selection` открывает путь в документ, т.к. он предоставляет доступ к выделенному фрагменту документа. В частности, у объекта `Selection` имеется метод `TypeText(`*`Text`*`)`, с помощью которого можно вставлять текст в документ. Используя свойства этого объекта (которые, в свою очередь, могут являться объектами со своими свойствами), можно управлять параметрами выделенного фрагмента, например, устанавливать нужный размер и гарнитуру шрифта, выравнивать параграфы по центру и т.п.

Объектная модель Excel построена по тому же принципу, что и объектная модель Word. Основным объектом, содержащим все остальные, является `Application` (рис. 8.2).

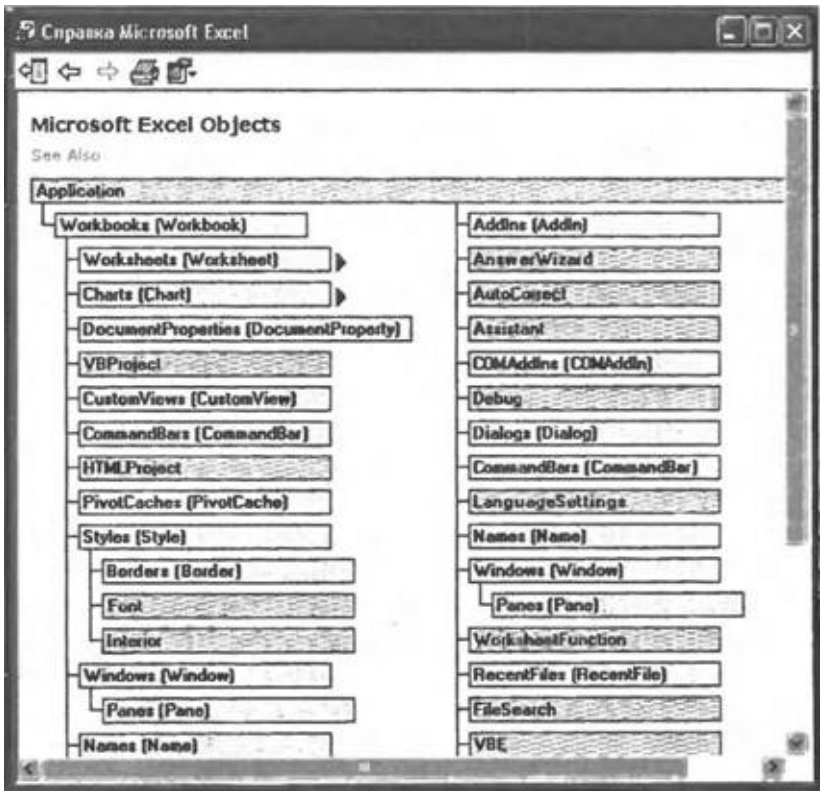


Рис. 8.2. Объектная модель Microsoft Excel

Напомним, что отдельные файлы в Excel называются *рабочими книгами*. Семейство `Workbooks` в Excel является аналогом семейства `Documents` в Word и содержит набор объектов `Workbook` (аналог объекта `Document` в Word), каждый из которых соответствует открытой в Word рабочей книге. Новая рабочая книга создается с помощью метода `Add()` объекта `Workbooks`.

Для доступа к ячейкам активного рабочего листа Excel используется свойство `Cells` объекта `Application`. Для получения или изменения значения отдельной ячейки применяется конструкция `Cells(`*`row, column`*`).Value`, где *`row`* и *`column`* являются соответственно номерами строки и столбца, на пересечении которых находится данная ячейка.

В Excel, как и в Word, имеется объект `Selection`, позволяющий работать с выделенным фрагментом электронной таблицы. Самым простым способом выделить диапазон ячеек активного рабочего листа является использование метода `Select()` объекта `Range`. Например, выражение `Range("A1:C1").Select()` позволяет выделить три смежные ячейки: `"A1"`, `"B1"` и `"C1"`.

Для того чтобы понять, какой именно объект Word или Excel нужно использовать для решения той или иной задачи, часто проще всего бывает

проделать в соответствующем приложении необходимые манипуляции вручную, включив предварительно режим записи макроса. В результате мы получим текст макроса на языке VBA (Visual Basic for Applications), из которого будет ясно, какие методы и с какими параметрами нужно вызывать и какие значения нужно присваивать свойствам объектов. В качестве простой иллюстрации проделаем следующие действия. Запустим Word, запустим Macro Recorder (**Сервис|Макрос|Начать запись** (Tools|Macros|Record)), назовем новый макрос "Andrey" и нажмем на кнопку **OK** (рис. 8.3).

Рис. 8.3. Создание нового макроса в Macro Recorder

После этого напишем в документе слово "Андрей" и прекратим запись макроса. Теперь можно посмотреть содержимое записанного макроса. Для этого нужно выбрать пункт **Макросы** (Macroses) в меню **Сервис|Макрос** (Tools|Macros), выделить макрос "Andrey" в списке всех доступных макросов и нажать кнопку **Изменить** (Edit). В открывшемся окне редактора Visual Basic появится текст макроса:

```
Sub Андрей()
'
' Андрей Макрос
' Макрос записан 01.08.02 Андрей Владимирович Попов
'
 Selection.TypeText Text:="Андрей"
End Sub
```

Как мы видим, для печати слова в документе был использован метод `TypeText` объекта `Selection`.

Макросы в Excel записываются и редактируются аналогичным образом.

Вывод данных из записной книжки в документ Microsoft Word

В качестве примера взаимодействия WSH с Microsoft Word мы рассмотрим два сценария, которые будут создавать документы Word и выводить туда информацию из записной книжки в XML-формате, которая хранится в файле book.xml.

Рис. 8.4. Вывод данных из XML-файла в документ Word в виде обычного текста

В первом из этих сценариев поля каждой записи будут располагаться друг под другом, т.е. информация печатается в виде обычного текста (рис. 8.4).

Второй сценарий будет заполнять данными из XML-файла строки таблицы в документе Word (рис. 8.5).

Рис. 8.5. Вывод данных из XML-файла в таблицу Word

Вывод записей в виде обычного текста

Для печати данных из book.xml в документ Word в режиме обычного текста мы создадим JScript-сценарий ListWord.js. За основу этого сценария взят рассмотренный в *главе 6* сценарий SortNameXMLDOM.js, в котором вывод информации производился в текстовый файл, открываемый затем в Блокноте.

Основная функция `Main()` сценария ListWord.js начинается с создания объекта `WshShell` и вызова функции `InitPath()`, в которой определяются пути к файлам book.xml (переменная `PathBook`) и out.doc (переменная `PathOut`):

```
//Создаем объект WshShell
WshShell = WScript.CreateObject("WScript.Shell");
//Определяем пути к файлам
InitPath();
```

Для запуска новой копии Microsoft Word мы создаем экземпляр объекта Word.Application:

```
//Создаем объект Application
WA=WScript.CreateObject("Word.Application");
```

После запуска Word в нем создается новый пустой документ с помощью метода `Add()` семейства `Documents`; ссылка на получающийся в результате экземпляр объекта `Document` сохраняется в переменной `WD`:

```
//Создаем новый документ
wd=WA.Documents.Add();
```

Так как процесс печати данных из сценария может быть довольно длительным, окно Word мы сделаем видимым и максимизируем его:

```
//Делаем окно Winword видимым
WA.Visible=true;
//Максимизируем окно Winword
WA.WindowState=wdWindowStateMaximize;
```

Для того чтобы начать печатать в окне Word, нужно получить ссылку на объект `Selection` (глобальная переменная `Sel`), который позволяет работать с выделенным текстом:

```
//Получаем ссылку на объект Selection
Sel=WA.Selection;
```

С помощью свойства `Font` объекта `Selection` мы устанавливаем размер шрифта, которым далее будет печататься текст:

```
//Устанавливаем размер шрифта 12 пт
Sel.Font.Size=12;
```

Теперь мы полностью готовы начать печать текста. Сначала в функции `TopReport()` печатается заголовок отчета:

```
//Печатаем заголовок отчета
TopReport("Общий список");
```

Функция `TopReport(`*`Mess`*`)` имеет следующий вид:

```
//Вывод заголовка отчета
function TopReport(Mess) {
 //Устанавливаем выравнивание по центру
 Sel.ParagraphFormat.Alignment=wdAlignParagraphCenter;
 //Устанавливаем полужирный шрифт
 Sel.Font.Bold=true;
 //Выводим сообщение с переводом строки
 Sel.TypeText(Mess+"\n");
 Sel.TypeText("\n");
 //Устанавливаем выравнивание слева
 Sel.ParagraphFormat.Alignment=wdAlignParagraphLeft;
 //Отменяем полужирный шрифт
 Sel.Font.Bold=false;
}
```

Как мы видим, текст печатается с помощью метода `TypeText()`, а форматируется путем изменения соответствующих свойств объекта `Selection` (которые, в свою очередь, являются объектами того или иного типа). Отметим, что именные константы, которые используются для форматирования текста, были предварительно проинициализированы в самом начале сценария:

```
//Инициализируем константы Winword'a
var wdAlignParagraphLeft=0, wdAlignParagraphCenter=1,
wdWindowStateMaximize=1;
```

После выполнения функции `TopReport()` в документе Word будет полужирным шрифтом с выравниванием по центру напечатана строка

"Общий список", а курсор установится на две строки ниже (рис. 8.6).

Рис. 8.6. Заголовок отчета, напечатанный в сценарии ListWord.js

Далее в сценарии данные из XML-файла book.xml считываются в массив `PersonArr` с использованием объектной модели XML DOM (этот процесс был подробно описан в *главе* 7). Печать информации из элемента массива `PersonArr` (экземпляра объекта `Person`) производится в функции `PrintPerson(`*`PersRec`*`)`:

```
//Печать содержимого полей объекта Person
function PrintPerson(PersRec) {
 //Печатаем поля текущей записи
 TypeString("Фамилия",PersRec.LastName);
 TypeString("Имя",PersRec.Name);
 TypeString("Телефон",PersRec.Phone);
 TypeString("Улица",PersRec.Street);
 TypeString("Дом",PersRec.House);
 TypeString("Кв.",PersRec.App);
 TypeString("Заметки",PersRec.Note);
 //Печатаем разделитель с переводом строки
 Sel.TypeText("---------------------------------------\n");
 //Увеличиваем номер текущей записи
 NomRec++;
}
```

Здесь используется функция `TypeString(`*`Title, Content`*`)`, в которой происходит печать наклонным шрифтом названия поля (параметр *`Title`*) и прямым шрифтом значения этого поля (параметр *`Content`*):

```
//Вывод одного поля из записи
function TypeString(Title, Content) {
 //Устанавливаем наклонный шрифт
 Sel.Font.Italic=true;
```

```
 //Печатаем название поля
 Sel.TypeText(Title+":\t");
 //Отменяем наклонный шрифт
 Sel.Font.Italic=false;
 //Печатаем содержимое поля
 Sel.TypeText(Content+"\n");
}
```

В качестве итоговой информации в функции `BottomReport(`*`Mess`*`)` печатается общее количество записей в книжке:

```
//Вывод итоговой информации
function BottomReport(Mess) {
 //Устанавливаем полужирный шрифт
 Sel.Font.Bold=true;
 //Выводим сообщение с переводом строки
 Sel.TypeText(Mess+"\n");
 //Отменяем полужирный шрифт
 Sel.Font.Bold=false;
}
```

После того как вся нужная информация напечатана в документе, он сохраняется на диске с помощью метода `SaveAs()` объекта `Document`:

```
//Сохраняем созданный документ под именем out.doc
WD.SaveAs(PathOut);
```

В листинге 8.1 приводится полный текст сценария ListWord.js.

Листинг 8.1. Вывод данных из XML-файла в Microsoft Word (обычный текст)

```
/*****************************************************************
/*                                                          Имя:
ListWord.js                                                 */
/*                                                         Язык:
JScript                                                     */
/*    Описание:    Печать    данных    из    записной    книжки
в                    */
/*                                                 файл    Microsoft
Word                                        */
/*****************************************************************
//Объявляем переменные
var
 WshShell,  //Экземпляр объекта WshShell
 BasePath,  //Путь к текущему каталогу
 PathBook,  //Путь к файлу с данными
 PathOut,   //Путь к выходному файлу Winword
 WA,        //Экземпляр объекта Application
```

```
      WD,        //Экземпляр объекта Document
      Sel,       //Экземпляр объекта Selection
      NomRec=0,  //Счетчик количества записей
      PersonRec, //Объект для хранения данных об одном человеке
      PersonArr; //Массив для хранения объектов PersonRec
     //Инициализируем константы Winword'a
     var
wdAlignParagraphLeft=0,wdAlignParagraphCenter=1,wdWindowStateMaximiz

     //Построение путей к файлам
     function InitPath() {
     var BasePath;
      BasePath=WshShell.CurrentDirectory+"\\";
      //Путь к файлу с данными
      PathBook=BasePath+"book.xml",
      //Путь к выходному файлу
      PathOut=BasePath+"out.doc";
     }
     //Конструктор объекта Person
     function Person(LastName,Name,Phone,Street,House,App,Note) {
      this.LastName=LastName; //Фамилия
      this.Name=Name;         //Имя
      this.Phone=Phone;       //Телефон
      this.Street=Street;     //Улица
      this.House=House;       //Дом
      this.App=App;           //Квартира
      this.Note=Note;         //Примечание
     }
     //Определение значения тега tgName XML-элемента obj
     function GetTagVal(obj, tgName) {
      var ElemList;
      //Создаем коллекцию дочерних для obj элементов, которые
      //задаются тегом tgName
      ElemList=obj.getElementsByTagName(tgName);
      //Проверяем, есть ли в коллекции ElemList элементы
      if (ElemList.length>0)
       //Возвращаем значение тега tgName
        return ElemList.item(0).text
      else return "";
     }
     //Заполнение нового элемента массива
     function PersonToArray(XNode) {
      //Создаем новый экземпляр PersonRec объекта Person
      PersonRec=new Person();
      //Заполняем поля объекта PersonRec
      PersonRec.LastName=GetTagVal(XNode,"LastName");
      PersonRec.Name=GetTagVal(XNode,"Name");
```

```
      PersonRec.Phone=GetTagVal(XNode,"Phone");
      PersonRec.Street=GetTagVal(XNode,"Street");
      PersonRec.House=GetTagVal(XNode,"House");
      PersonRec.App=GetTagVal(XNode,"App");
      PersonRec.Note=GetTagVal(XNode,"Note");
      //Сохраняем объект PersonRec в массиве
      PersonArr[PersonArr.length]=PersonRec;
     }
     //Создание массива объектов Person
     function FileToArray() {
      var XML,Root,NomRec,CurrNode,i;
      //Создаем массив PersonArr
      PersonArr=new Array();
      //Создаем объект XML DOM
      XML = WScript.CreateObject("Msxml.DOMDocument");
      //Загружаем XML-документ из файла
      XML.load(PathBook);
       //Сохраняем в переменной Root ссылку на корневой элемент
документа
      Root=XML.documentElement;
      //Перебираем все дочерние элементы первого уровня вложенности
      //для корневого элемента
      for (i=1; i<=Root.childNodes.length-1;i++) {
       //Выделяем в коллекции XML-элементов i-й элемент
       CurrNode=Root.childNodes.item(i);
       //Добавляем новый элемент в массив объектов Person
       PersonToArray(CurrNode);
      }
     }
     //Вывод заголовка отчета
     function TopReport(Mess) {
      //Устанавливаем выравнивание по центру
      Sel.ParagraphFormat.Alignment=wdAlignParagraphCenter;
      //Устанавливаем полужирный шрифт
      Sel.Font.Bold=true;
      //Выводим сообщение с переводом строки
      Sel.TypeText(Mess+"\n");
      Sel.TypeText("\n");
      //Устанавливаем выравнивание слева
      Sel.ParagraphFormat.Alignment=wdAlignParagraphLeft;
      //Отменяем полужирный шрифт
      Sel.Font.Bold=false;
     }
     //Вывод итоговой информации
     function BottomReport(Mess) {
      //Устанавливаем полужирный шрифт
      Sel.Font.Bold=true;
```

```
 //Выводим сообщение с переводом строки
 Sel.TypeText(Mess+"\n");
 //Отменяем полужирный шрифт
 Sel.Font.Bold=false;
}
//Вывод одного поля из записи
function TypeString(Title, Content) {
 //Устанавливаем наклонный шрифт
 Sel.Font.Italic=true;
 //Печатаем название поля
 Sel.TypeText(Title+":\t");
 //Отменяем наклонный шрифт
 Sel.Font.Italic=false;
 //Печатаем содержимое поля
 Sel.TypeText(Content+"\n");
}
//Печать содержимого полей объекта Person
function PrintPerson(PersRec) {
 //Печатаем поля текущей записи
 TypeString("Фамилия",PersRec.LastName);
 TypeString("Имя",PersRec.Name);
 TypeString("Телефон",PersRec.Phone);
 TypeString("Улица",PersRec.Street);
 TypeString("Дом",PersRec.House);
 TypeString("Кв.",PersRec.App);
 TypeString("Заметки",PersRec.Note);
 //Печатаем разделитель с переводом строки
 Sel.TypeText("------------------------------------\n");
 //Увеличиваем номер текущей записи
 NomRec++;
}
//Сортировка массива и печать его содержимого
function ListPersonArray() {
 var i;
 //Сортировка массива по фамилии
 PersonArr.sort(SortLastName);
 //Цикл по всем элементам массива PersonArr
 for (i=0;i<=PersonArr.length-1;i++) {
  //Печать информации для текущей записи
  PrintPerson(PersonArr[i]);
 }
}
//Функция для сортировки массива по фамилии
function SortLastName(Pers1,Pers2) {
 if (Pers1.LastName<Pers2.LastName) return -1;
 else if (Pers1.LastName==Pers2.LastName) return 0;
 else return 1;
```

```
    }
    //Печать содержимого файла с данными
    function ListFile() {
     //Считываем данные из файла в массив
     FileToArray();
     //Печатаем информацию из массива
     ListPersonArray();
    }
    //Основная запускная функция
    function Main() {
     //Создаем объект WshShell
     WshShell = WScript.CreateObject("WScript.Shell");
     //Определяем пути к файлам
     InitPath();
     //Создаем объект Application
     WA=WScript.CreateObject("Word.Application");
     //Создаем новый документ
     WD=WA.Documents.Add();
     //Делаем окно Winword видимым
     WA.Visible=true;
     //Максимизируем окно Winword
     WA.WindowState=wdWindowStateMaximize;
     //Получаем ссылку на объект Selection
     Sel=WA.Selection;
     //Устанавливаем размер шрифта 12 пт
     Sel.Font.Size=12;
     //Печатаем заголовок отчета
     TopReport("Общий список");
     //Печатаем содержимое XML-файла с данными
     ListFile();
     //Печатаем итоговую информацию
     BottomReport("Всего записей: "+PersonArr.length);
     //Сохраняем созданный документ под именем out.doc
     WD.SaveAs(PathOut);
    }
    /*******************                                    Начало
********************************/
    Main();
    /*************                                           Конец
*********************************************/
```

Вывод записей в таблицу

Для того чтобы выводить записи из файла с данными в таблицу Word, мы поступим следующим образом.

Создадим вначале документ-шаблон table.dot, в котором будет нарисована таблица для вывода информации из записной книжки, а также будут написаны заголовок отчета и итоговая информация (рис. 8.7). Задача сценария заключается в создании нового документа по этому шаблону и заполнении строк таблицы нужными данными.

Рис. 8.7. Документ-шаблон table.dot

Напомним, как создается новый шаблон в Word. Запустив Word, нужно выбрать в меню **Файл** (File) пункт **Создать** (New) и установить переключатель **Создать** (New) в положение **шаблон** (template) (рис. 8.8)

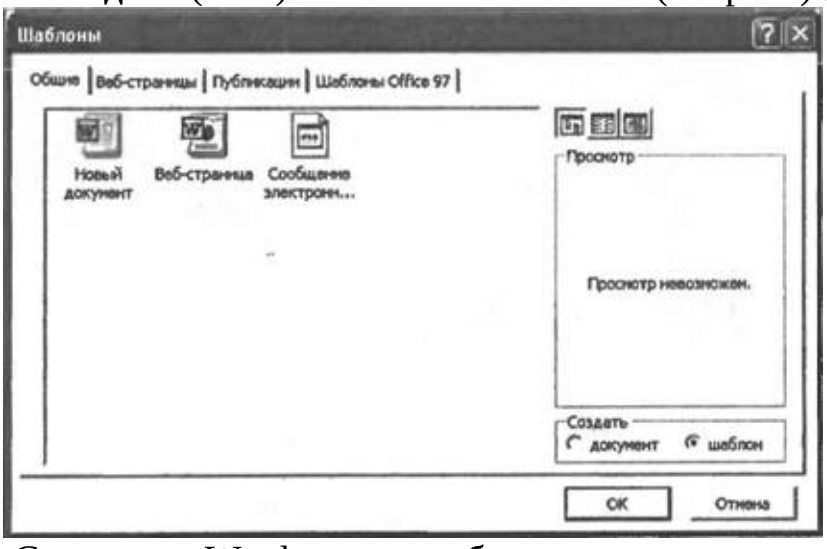


Рис. 8.8. Создание в Word нового шаблона

Для обозначения в документе места, откуда будет начинаться вывод текста, в шаблон мы добавим две *закладки* (bookmarks). Вставляется

закладка в текст следующим образом: курсор перемещается в нужную позицию, в меню **Вставка** (Insert) выбирается пункт **Закладка** (Bookmark), в диалоговом окне **Закладка** (Bookmark) пишется имя закладки и нажимается кнопка **Добавить** (Add) (рис. 8.9)

Рис. 8.9. Добавление новой закладки в документ Word

Первую закладку с именем "TableStart" нужно поместить в первую ячейку таблицы, т.е. в то место, откуда начнется печататься фамилия для самой первой записи. Вторая закладка с именем "NomRec" ставится после слов "Всего записей:" — здесь будет напечатано число записей (строк в таблице).

Перейдем теперь к рассмотрению сценария ListWordTable.js, который создает на основе шаблона table.dot файл out.doc и заполняет таблицу в этом файле данными из записной книжки book.xml (рис. 8.5).

Основной функцией в этом сценарии является, как обычно, функция `Main()`. Здесь сначала вызывается функция `InitPath()` для определения путей к файлам book.xml (переменная `PathBook`), out.doc (переменная `PathOut`) и table.dot (переменная `PathTempl`), после чего создается экземпляр объекта `Word.Application`:

```
//Создаем объект Application
WA=WScript.CreateObject("Word.Application");
```

Для создания нового документа на основе шаблона Table.dot мы указываем путь к этому шаблону в качестве аргумента метода `Add()` семейства `Documents`:

```
//Создаем новый документ
WD=WA.Documents.Add(PathTempl, false);
```

Окно Word делается видимым и максимизируется:

```
//Делаем окно Winword видимым
WA.Visible=true;
```

```
//Максимизируем окно Winword
WA.WindowState=wdWindowStateMaximize;
```

В переменной `Sel` сохраняется ссылка на объект `Selection`:

```
//Получаем ссылку на объект Selection
Sel=WA.Selection;
```

Как и в сценарии ListWord.js, данные из файла book.xml считываются в массив `PersonArr` с использованием объектной модели XML DOM. Вывод информации из этого массива в строки таблицу происходит в функции `ListPersonArray()`:

```
//Сортировка массива и печать его содержимого
function ListPersonArray() {
 var i;
 //Сортировка массива по фамилии
 PersonArr.sort(SortLastName);
 //Переходим к закладке TableStart
 WD.Bookmarks("TableStart").Select();
 //Цикл по всем элементам массива PersonArr
 for (i=0;i<=PersonArr.length-1;i++) {
  //Печать информации для текущей записи
  PrintPerson(PersonArr[i]);
 }
}
```

Как мы видим, сначала в этой функции в семействе `Bookmarks` находится закладка с именем "TableStart" и с помощью метода `Select()` происходит выделение этой закладки в документе. Затем в цикле `for` вызывается функция `PrintPerson(`*`PersRec`*`)` для каждого элемента массива `PersonArr`; в этой функции содержимое полей объекта `PersRec` последовательно печатается в ячейки таблицы:

```
//Печать содержимого полей объекта Person
function PrintPerson(PersRec) {
 //Печатаем поля текущей записи
 WA.Selection.Text=PersRec.LastName;
 //Переходим к следующей ячейке таблицы
 WA.Selection.MoveRight(wdCell);
 WA.Selection.Text=PersRec.Phone;
 WA.Selection.MoveRight(wdCell);
 WA.Selection.Text=PersRec.Note;
 if (NomRec<PersonArr.length-1)
  //Если напечатаны еще не все записи, то нужно
  //добавить в таблицу новую строку
  WA.Selection.MoveRight(wdCell);
 //Увеличиваем номер текущей записи
 NomRec++;
}
```

Итак, печать в таблице происходит следующим образом: после вывода текста в текущую ячейку мы перемещаемся в соседнюю ячейку справа (константа `wdCell` проинициализирована в самом начале сценария, `wdCell=12`):

```
WA.Selection.MoveRight(wdCell);
```

Если при этом текущая ячейка находилась в третьем столбце, то после такого перемещения в таблицу автоматически будет добавлена новая строка.

После того как все строки в таблице напечатаны, в файл выводится итоговая информация. Для этого мы выделяем закладку с именем "NomRec" и печатаем туда количество элементов в массиве `PersonArr`:

```
//Выделяем закладку "NomRec"
WD.Bookmarks("NomRec").Select();
//Печатаем итоговую информацию
WA.Selection.Text=PersonArr.length;
```

Окончательно сформированный файл сохраняется на диске под именем out.doc:

```
//Сохраняем созданный документ под именем out.doc
WD.SaveAs(PathOut);
```

Полностью текст сценария ListWordTable.js приведен в листинге 8.2.

Листинг 8.2. Вывод данных из XML-файла в таблицу Microsoft Word

```
/*****************************************************************
/*                                                          Имя:
ListWordTable.js                                          */
/*                                                         Язык:
JScript                                                  */
/*   Описание:   Печать   данных   из   записной   книжки   в
таблицу            */
/*                                                      Microsoft
Word                                          */
/*****************************************************************
//Объявляем переменные
var
 WshShell,  //Экземпляр объекта WshShell
 BasePath,  //Путь к текущему каталогу
 PathBook,  //Путь к файлу с данными
 PathOut,   //Путь к выходному файлу Winword
 PathTempl, //Путь к документу-шаблону
 WA,        //Экземпляр объекта Application
 WD,        //Экземпляр объекта Document
```

```
     Sel,       //Экземпляр объекта Selection
     NomRec=0,  //Счетчик количества записей
     PersonRec, //Объект для хранения данных об одном человеке
     PersonArr; //Массив для хранения объектов PersonRec
    //Инициализируем константы Winword'a
    var              wdCell=12,             wdAlignParagraphLeft=0,
wdAlignParagraphCenter=1, wdWindowStateMaximize=1;

    //Построение путей к файлам
    function InitPath() {
     var BasePath;
     BasePath=WshShell.CurrentDirectory+"\\";
     //Путь к файлу с данными
     PathBook=BasePath+"book.xml",
     //Путь к выходному файлу
     PathOut=BasePath+"out.doc";
     //Путь к документу-шаблону
     PathTempl=BasePath+"table.dot";
    }
    //Конструктор объекта Person
    function Person(LastName,Name,Phone,Street,House,App,Note) {
     this.LastName=LastName; //Фамилия
     this.Name=Name;         //Имя
     this.Phone=Phone;       //Телефон
     this.Street=Street;     //Улица
     this.House=House;       //Дом
     this.App=App;           //Квартира
     this.Note=Note;         //Примечание
    }
    //Определение значения тега tgName XML-элемента obj
    function GetTagVal(obj, tgName) {
     var ElemList;
     //Создаем коллекцию дочерних для obj элементов, которые
     //задаются тегом tgName
     ElemList=obj.getElementsByTagName(tgName);
     //Проверяем, есть ли в коллекции ElemList элементы
     if (ElemList.length>0)
      //Возвращаем значение тега tgName
      return ElemList.item(0).text
     else return "";
    }
    //Заполнение нового элемента массива
    function PersonToArray(XNode) {
     //Создаем новый экземпляр PersonRec объекта Person
     PersonRec=new Person();
     //Заполняем поля объекта PersonRec
     PersonRec.LastName=GetTagVal(XNode,"LastName");
```

```
      PersonRec.Name=GetTagVal(XNode,"Name");
      PersonRec.Phone=GetTagVal(XNode,"Phone");
      PersonRec.Street=GetTagVal(XNode,"Street");
      PersonRec.House=GetTagVal(XNode,"House");
      PersonRec.App=GetTagVal(XNode,"App");
      PersonRec.Note=GetTagVal(XNode,"Note");
      //Сохраняем объект PersonRec в массиве
      PersonArr[PersonArr.length]=PersonRec;
     }
     //Создание массива объектов Person
     function FileToArray() {
      var XML,Root,NomRec,CurrNode,i;
      //Создаем массив PersonArr
      PersonArr=new Array();
      //Создаем объект XML DOM
      XML = WScript.CreateObject("Msxml.DOMDocument");
      //Загружаем XML-документ из файла
      XML.load(PathBook);
       //Сохраняем в переменной Root ссылку на корневой элемент
документа
      Root=XML.documentElement;
      //Перебираем все дочерние элементы первого уровня вложенности
      //для корневого элемента
      for (i=1; i<=Root.childNodes.length-1;i++) {
       //Выделяем в коллекции XML-элементов i-й элемент
       CurrNode=Root.childNodes.item(i);
       //Добавляем новый элемент в массив объектов Person
       PersonToArray(CurrNode);
      }
     }
     //Печать содержимого полей объекта Person
     function PrintPerson(PersRec) {
      //Печатаем поля текущей записи
      WA.Selection.Text=PersRec.LastName;
      //Переходим к следующей ячейке таблицы
      WA.Selection.MoveRight(wdCell);
      WA.Selection.Text=PersRec.Phone;
      WA.Selection.MoveRight(wdCell);
      WA.Selection.Text=PersRec.Note;
      if (NomRec<PersonArr.length-1)
       //Если напечатаны еще не все записи, то нужно
       //добавить в таблицу новую строку
       WA.Selection.MoveRight(wdCell);
      //Увеличиваем номер текущей записи
      NomRec++;
     }
     //Сортировка массива и печать его содержимого
```

```
function ListPersonArray() {
 var i;
 //Сортировка массива по фамилии
 PersonArr.sort(SortLastName);
 //Переходим к закладке TableStart
 WD.Bookmarks("TableStart").Select();
 //Цикл по всем элементам массива PersonArr
 for (i=0;i<=PersonArr.length-1;i++) {
  //Печать информации для текущей записи
  PrintPerson(PersonArr[i]);
 }
}
//Функция для сортировки массива по фамилии
function SortLastName(Pers1,Pers2) {
 if (Pers1.LastName<Pers2.LastName) return -1;
 else if (Pers1.LastName==Pers2.LastName) return 0;
 else return 1;
}
//Печать содержимого файла с данными
function ListFile() {
 //Считываем данные из файла в массив
 FileToArray();
 //Печатаем информацию из массива
 ListPersonArray();
}
//Основная запускная функция
function Main() {
 //Создаем объект WshShell
 WshShell = WScript.CreateObject("WScript.Shell");
 //Определяем пути к файлам
 InitPath();
 //Создаем объект Application
 WA=WScript.CreateObject("Word.Application");
 //Создаем новый документ
 WD=WA.Documents.Add(PathTempl,false);
 //Делаем окно Winword видимым
 WA.Visible=true;
 //Максимизируем окно Winword
 WA.WindowState=wdWindowStateMaximize;
 //Получаем ссылку на объект Selection
 Sel=WA.Selection;
 //Выводим в таблицу содержимое файла с данными
 ListFile();
 //Выделяем закладку "NomRec"
 WD.Bookmarks("NomRec").Select();
 //Печатаем итоговую информацию
 WA.Selection.Text=PersonArr.length;
```

```
     //Сохраняем созданный документ под именем out.doc
     WD.SaveAs(PathOut);
    }
    /*******************                                    Начало
********************************/
    Main();
    /*************                                           Конец
*******************************************/
```

Вывод данных из записной книжки в таблицу Microsoft Excel

Напишем сценарий, который будет создавать файл (рабочую книгу) Microsoft Excel и заносить туда данные из записной книжки (рис. 8.10).

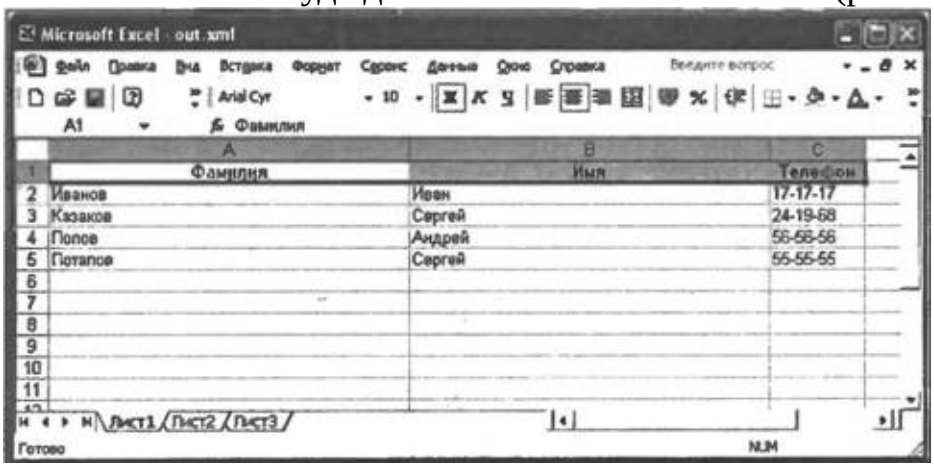


Рис. 8.10. Рабочая книга Microsoft Excel с данными из файла book.xml

Для того чтобы использовать определенные в Excel именные константы без их предварительной инициализации (как мы это делали в сценариях, работающих с Word), наш сценарий будет представлять собой WS-файл ListXLS.wsf, в котором мы определим с помощью тега `<reference>` ссылку на объект `Excel.Sheet`:

```
<reference object="Excel.Sheet"/>
```

Основная функция сценария `Main()` как всегда начинается с создания объекта `WshShell` и вызова функции `InitPath()`, в которой определяется путь к файлу с данными book.xls и выходному файлу out.xls:

```
//Создаем объект WshShell
WshShell = WScript.CreateObject("WScript.Shell");
//Определяем пути к файлам
InitPath();
```

Для того чтобы запустить Excel и получить доступ к его интерфейсам, нужно создать экземпляр объекта `Excel.Application` (переменная `XL`):

```
//Создаем объект Application
XL=WScript.CreateObject("Excel.Application");
```

Чтобы визуально контролировать процесс вывода информации, окно Excel мы сделаем видимым:

```
//Делаем окно Microsoft Excel видимым
XL.Visible=true;
```

Для создания в Excel нового файла (рабочей книги) используется метод Add() семейства Workbooks:

```
//Открываем новую рабочую книгу
XL.WorkBooks.Add();
```

Затем в сценарии выставляется необходимая ширина трех первых колонок (здесь мы будем печатать имя, фамилию и номер телефона). Для этого следует присвоить нужные значения свойству ColumnWidth соответствующих элементов коллекции Columns:

```
//Устанавливаем нужную ширину колонок
XL.Columns(1).ColumnWidth = 40;
XL.Columns(2).ColumnWidth = 40;
XL.Columns(3).ColumnWidth = 10;
```

Заголовок отчета (названия столбцов) печатается в функции TopReport():

```
//Вывод заголовка отчета
function TopReport() {
  //Печатаем в ячейки текст
 XL.Cells(1,1).Value="Фамилия";
 XL.Cells(1,2).Value="Имя";
 XL.Cells(1,3).Value="Телефон";
 //Выделяем три ячейки
 XL.Range("A1:C1").Select();
 //Устанавливаем полужирный текст для выделенного диапазона
 XL.Selection.Font.Bold = true;
  //Устанавливаем выравнивание по центру для выделенного диапазона
 XL.Selection.HorizontalAlignment=xlCenter;
}
```

Как мы видим, доступ к ячейке можно получить с помощью свойства Cells объекта Application, указав в круглых скобках индексы строки и столбца, на пересечении которых находится данная ячейка (нумерация строк и столбцов начинается с единицы). Свойство Value соответствует содержимому ячейки.

Так как названия столбцов должны быть выделены полужирным шрифтом и выровнены по центру, мы в функции TopReport() с помощью метода Select() объекта Range сначала выделяем сразу три ячейки ("A1", "B1" и "C1"):

```
//Выделяем три ячейки XL.Range("A1:C1").Select();
```

а затем устанавливаем необходимые свойства у объекта Selection, который соответствует выделенному диапазону:

```
//Устанавливаем полужирный текст для выделенного диапазона
XL.Selection.Font.Bold = true;
//Устанавливаем выравнивание по центру для выделенного
```

```
диапазона
    XL.Selection.HorizontalAlignment=xlCenter;
```

Как и во всех предыдущих сценариях этой главы, данные из файла book.xml посредством функции `FileToArray()` заносятся в массив `PersonArr`. Содержимое этого массива сортируется по фамилии и выводится в рабочую книгу Excel в функции `ListPersonArray()` (этот шаг также является одинаковым во всех сценариях):

```
//Сортировка массива и печать его содержимого
function ListPersonArray() {
 var i;
 //Сортировка массива по фамилии
 PersonArr.sort(SortLastName);
 for (i=0;i<=PersonArr.length-1;i++) {
  PrintPerson(PersonArr[i]);
 }
}
```

В функции `PrintPerson(`*`PersRec`*`)` происходит печать фамилии, имени и номера телефона для одной записи*`PersRec`* (напомним, что эта запись является экземпляром объекта `Person`). Для этого нужно определить номер строки, в ячейки которой будут записаны данные, что делается с помощью увеличения значения счетчика количества записей `NomRec`:

```
//Печать содержимого полей объекта Person
function PrintPerson(PersRec) {
 //Увеличиваем счетчик количества записей
 NomRec++;
 //В первом столбце печатаем фамилию
 XL.Cells(NomRec+1,1).Value=PersRec.LastName;
 //Во втором столбце печатаем имя
 XL.Cells(NomRec+1,2).Value=PersRec.Name;
 //В третьем столбце печатаем телефон
 XL.Cells(NomRec+1,3).Value=PersRec.Phone;
}
```

Полностью текст сценария ListXLS.wsf приведен в листинге 8.3.

Листинг 8.3. Вывод данных из XML-файла в таблицу Microsoft Excel

```
<package>
<job id="list_xl">
<runtime>
<description>
Имя: ListXLS.wsf
Описание: Печать данных из записной книжки в Microsoft Excel
</description>
```

```
</runtime>
<reference object="Excel.Sheet"/>
<script language="JScript">
var
 WshShell,  //Экземпляр объекта WshShell
 BasePath,  //Путь к текущему каталогу
 PathBook,  //Путь к файлу с данными
 PathOut,   //Путь к выходному файлу Winword
 XL,        //Экземпляр объекта Application
 NomRec=0,  //Счетчик количества записей
 PersonRec, //Объект для хранения данных об одном человеке
 PersonArr; //Массив для хранения объектов PersonRec

//Построение путей к файлам
function InitPath() {
 var BasePath;
 BasePath=WshShell.CurrentDirectory+"\\";
 //Путь к файлу с данными
 PathBook=BasePath+"book.xml",
 //Путь к выходному файлу
 PathOut=BasePath+"out.xml";
}

//Конструктор объекта Person
function Person(LastName,Name,Phone,Street,House,App,Note) {
 this.LastName=LastName; //Фамилия
 this.Name=Name;         //Имя
 this.Phone=Phone;       //Телефон
 this.Street=Street;     //Улица
 this.House=House;       //Дом
 this.App=App;           //Квартира
 this.Note=Note;         //Примечание
}
//Определение значения тега tgName XML-элемента obj
function GetTagVal(obj, tgName) {
 var ElemList;
 //Создаем коллекцию дочерних для obj элементов, которые
 //задаются тегом tgName
 ElemList=obj.getElementsByTagName(tgName);
 //Проверяем, есть ли в коллекции ElemList элементы
 if (ElemList.length>0)
  //Возвращаем значение тега tgName
  return ElemList.item(0).text
 else return "";
}
//Заполнение нового элемента массива
```

```
    function PersonToArray(XNode) {
     //Создаем новый экземпляр PersonRec объекта Person
     PersonRec=new Person();
     //Заполняем поля объекта PersonRec
     PersonRec.LastName=GetTagVal(XNode,"LastName");
     PersonRec.Name=GetTagVal(XNode,"Name");
     PersonRec.Phone=GetTagVal(XNode,"Phone");
     PersonRec.Street=GetTagVal(XNode,"Street");
     PersonRec.House=GetTagVal(XNode,"House");
     PersonRec.App=GetTagVal(XNode,"App");
     PersonRec.Note=GetTagVal(XNode,"Note");
     //Сохраняем объект PersonRec в массиве
     PersonArr[PersonArr.length]=PersonRec;
    }

    //Создание массива объектов Person
    function FileToArray() {
     var XML,Root,NomRec,CurrNode,i;
     //Создаем массив PersonArr
     PersonArr=new Array();
     //Создаем объект XML DOM
     XML = WScript.CreateObject("Msxml.DOMDocument");
     //Загружаем XML-документ из файла
     XML.load(PathBook);
      //Сохраняем в переменной Root ссылку на корневой элемент
документа
     Root=XML.documentElement;
     //Перебираем все дочерние элементы первого уровня вложенности
     //для корневого элемента
     for (i=1; i<=Root.childNodes.length-1;i++) {
      //Выделяем в коллекции XML-элементов i-й элемент
      CurrNode=Root.childNodes.item(i);
      //Добавляем новый элемент в массив объектов Person
      PersonToArray(CurrNode);
     }
    }
    //Вывод заголовка отчета
    function TopReport() {
      //Печатаем в ячейки текст
     XL.Cells(1,1).Value="Фамилия";
     XL.Cells(1,2).Value="Имя";
     XL.Cells(1,3).Value="Телефон";
     //Выделяем три ячейки
     XL.Range("A1:C1").Select();
     //Устанавливаем полужирный текст для выделенного диапазона
     XL.Selection.Font.Bold = true;
      //Устанавливаем выравнивание по центру для выделенного
```

```
диапазона
     XL.Selection.HorizontalAlignment=xlCenter;
    }
    //Печать содержимого полей объекта Person
    function PrintPerson(PersRec) {
     //Увеличиваем счетчик количества записей
     NomRec++;
     //В первом столбце печатаем фамилию
     XL.Cells(NomRec+1,1).Value=PersRec.LastName;
     //Во втором столбце печатаем имя
     XL.Cells(NomRec+1,2).Value=PersRec.Name;
     //В третьем столбце печатаем телефон
     XL.Cells(NomRec+1,3).Value=PersRec.Phone;
    }
    //Сортировка массива и печать его содержимого
    function ListPersonArray() {
     var i;
     //Сортировка массива по фамилии
     PersonArr.sort(SortLastName);
     for (i=0;i<=PersonArr.length-1;i++) {
      PrintPerson(PersonArr[i]);
     }
    }
    //Функция для сортировки массива по фамилии
    function SortLastName(Pers1,Pers2) {
     if (Pers1.LastName<Pers2.LastName) return -1;
     else if (Pers1.LastName==Pers2.LastName) return 0;
     else return 1;
    }
    //Печать содержимого файла с данными
    function ListFile() {
     //Считываем данные из файла в массив
     FileToArray();
     //Печатаем информацию из массива
     ListPersonArray();
    }
    //Основная запускная функция
    function Main() {
     //Создаем объект WshShell
     WshShell = WScript.CreateObject("WScript.Shell");
     //Определяем пути к файлам
     InitPath();
     //Создаем объект Application
     XL=WScript.CreateObject("Excel.Application");
     //Делаем окно Microsoft Excel видимым
     XL.Visible=true;
     //Открываем новую рабочую книгу
```

```
 XL.WorkBooks.Add();
 //Устанавливаем нужную ширину колонок
 XL.Columns(1).ColumnWidth = 40;
 XL.Columns(2).ColumnWidth = 40;
 XL.Columns(3).ColumnWidth = 10;
 //Печатаем заголовок таблицы
 TopReport();
 //Печатаем содержимое файла с данными
 ListFile();
 //Сохраняем созданный документ под именем out.xls
 XL.ActiveWorkbook.SaveAs(PathOut);
}
//Запускаем основную функцию
Main();
</script>
</job>
</package>
```

Глава 9
Использование в сценариях баз данных

На практике довольно часто возникают задачи, для решения которых необходимо из сценариев получать доступ к данным, хранящимся во внешних базах самого различного формата (структурированные текстовые файлы, таблицы DBF и Paradox, базы данных Microsoft Access и Microsoft SQL Server и т.д.). Это довольно просто можно сделать с помощью технологии Microsoft ADO (ActiveX Data Objects). Объекты ADO являются частью *компонентов доступа к данным Microsoft* (MDAC, Microsoft Data Access Components), которые поставляются, например, с браузером Microsoft Internet Explorer версии 5.0 и выше (таким образом, в Windows XP эти компоненты присутствуют изначально) или могут быть свободно получены с сервера Microsoft (http://msdn.microsoft.com/data/download.htm).

Мы не будем здесь останавливаться на объектной модели и принципах работы ADO (желающие подробнее разобраться с этими вопросами могут обратиться к документации MSDN), а здесь лишь разберем несколько конкретных примеров работы с таблицей самой простой структуры: DBF-формата (до появления XML формат DBF широко применялся для обмена данными между разными автоматизированными системами, да и сейчас продолжает поддерживаться многими производителями программных продуктов). Как и в предыдущих главах, здесь будут использоваться данные записной книжки (ниже мы создадим сценарий, который будет копировать в таблицу информацию из XML-файла book.xml, с которым мы работали ранее).

Создать DBF-таблицу можно с помощью систем управления базами данных (СУБД) FoxPro, Microsoft Access или программы Microsoft Excel. Мы опишем процесс создания таблицы в Microsoft Access.

Создание таблицы Phone.dbf в Microsoft Access

Первом шагом при создании таблицы в любой базе данных является определение структуры этой таблицы — нужно определить имена, типы и размеры всех полей. Мы назовем нашу таблицу Phone.dbf; структура ее описана в табл. 9.1.

Таблица 9.1. *Структура таблицы Phone.dbf*

Имя поля	Тип	Размер (символов)
LastName	Символьный	50
Name	Символьный	50
Phone	Символьный	12
Street	Символьный	50
House	Символьный	4
App	Символьный	4
Notes	Символьный	100

Определившись со структурой таблицы, запустим Microsoft Access и создадим новую базу данных (рис. 9.1).

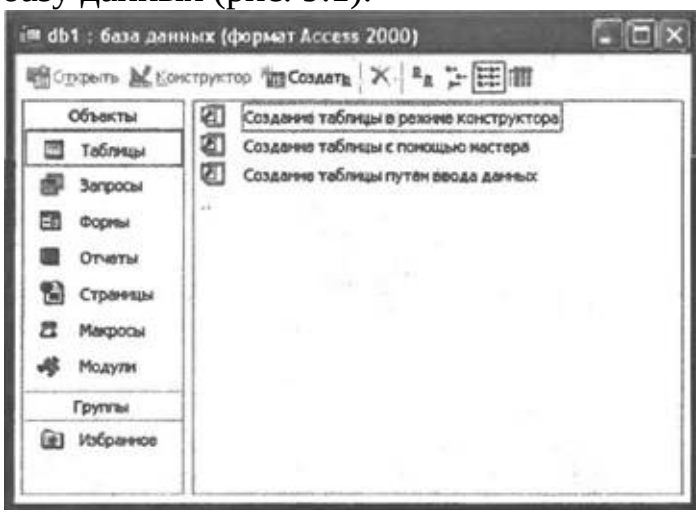


Рис. 9.1. Новая база данных в Microsoft Access

Выберем пункт **Создание таблиц в режиме конструктора** (Construct tables), последовательно введем названия полей, выбирая для каждого поля

символьный тип данных и устанавливая нужную длину (рис. 9.2).

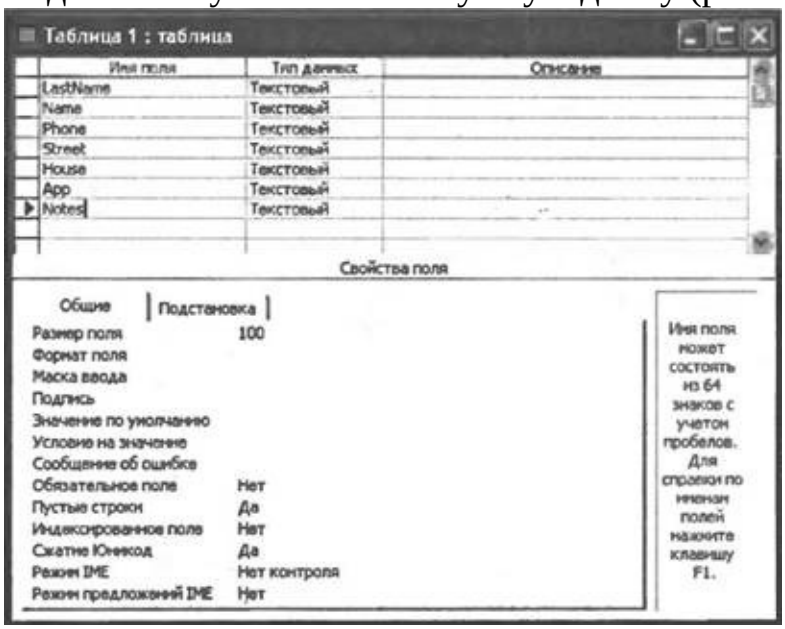


Рис. 9.2. Создание структуры новой таблицы

Затем закроем окно конструктора и введем название "Phone" в диалоговое окно сохранения таблицы (рис. 9.3).

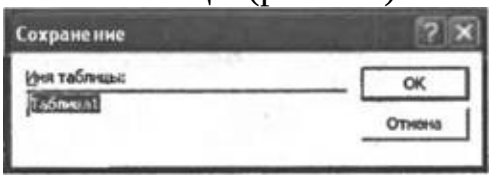


Рис. 9.3. Сохранение новой таблицы

Так как мы не будем определять ключевые поля в нашей таблице, то нужно ответить отрицательно на соответствующий вопрос (рис. 9.4).

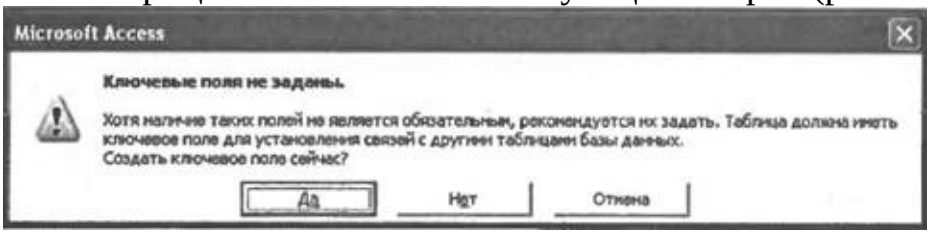


Рис. 9.4. Запрос на создание ключевых полей

После этого выделим таблицу Phone и выберем пункт **Экспорт** (Export) в меню **Файл** (File). В диалоговом окне **Экспорт объекта** (Object Export) выберем нужный каталог для сохранения таблицы Phone (далее таким каталогом мы будем считать C:\Tabl), укажем в качестве типа файла **dBase IV** и нажмем кнопку **Сохранить** (Save) (рис. 9.5).

Рис. 9.5. Экспорт таблицы Phone в формате DBF

Теперь пустая DBF-таблица нужной нам структуры сохранена на диске и программу Microsoft Access можно закрыть.

Настройка источника данных ODBC

Для получения доступа к созданной DBF-таблице из сценария WSH мы воспользуемся технологией ODBC (Open DataBase Connectivity). ODBC — это стандартное средство Microsoft для работы с реляционными базами данных различных форматов и производителей, способное обрабатывать запросы к базам на языке SQL (Structured Query Language, язык структурированных запросов).

Замечание

> Для начального ознакомления с языком SQL можно порекомендовать книгу [4].
>
> Для более глубокого изучения языка SQL рекомендуется книга Дж. Грофф, П. Вайнберг SQL: Полное руководство: Пер. с англ. — 2-е изд., перераб. и доп. — Киев: Издательская группа BHV, 2001. — 816 с.

Вначале нам понадобится завести в системе ODBC-запись для связи с нашей базой, т.е. создать новый DSN (Data Source Name, имя источника данных). В Windows XP это делается следующим образом.

Загрузим **Панель управления** (Control Panel) Windows (меню **Пуск** (Start)) и переключимся к классическому виду (рис. 9.6).

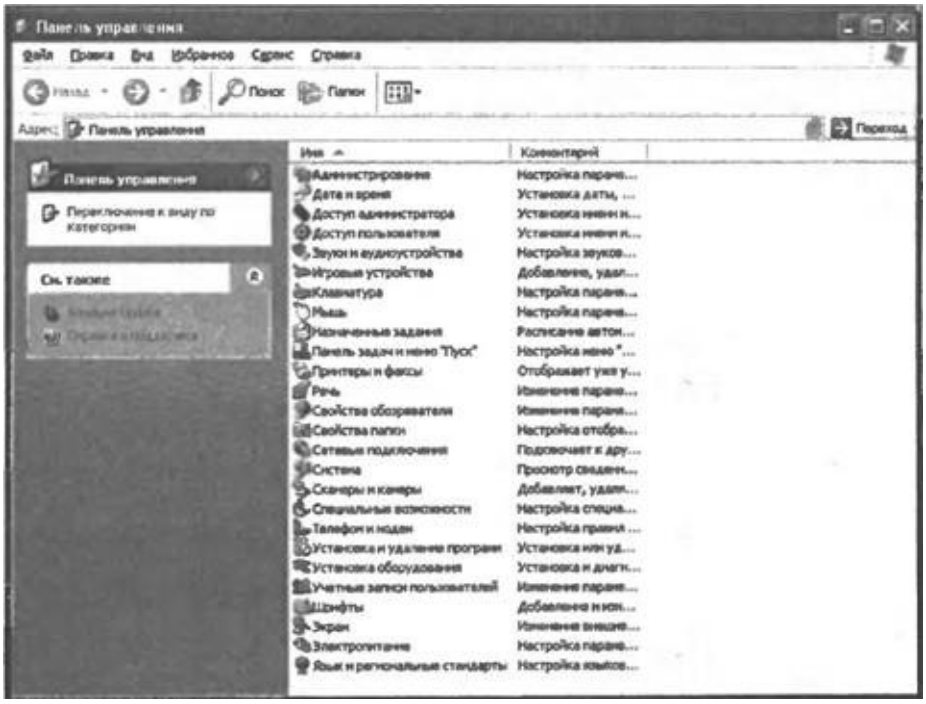


Рис. 9.6. Классический вид панели управления Windows XP

Последовательно выберем пункты **Администрирование** (Administrative tools) и **Источники данных (ODBC)** (Data sources (ODBC)). В появившемся диалоговом окне выберем вкладку **Системный DSN** (System DSN), что позволит создать DSN, доступный всем пользователям компьютера (рис. 9.7).

Рис. 9.7. Администратор источников данных ODBC в Windows XP

Нажмем кнопку **Добавить** (Add) и в появившемся окне выберем

драйвер Microsoft dBase Driver (*.dbf) (рис. 9.8).

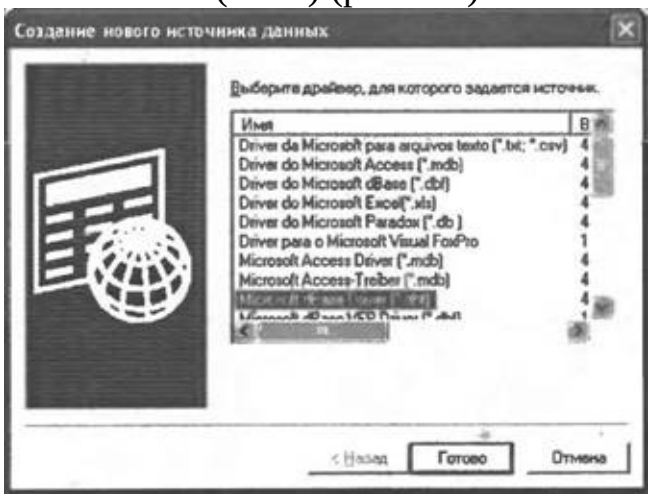


Рис. 9.8. Список драйверов ODBC

После нажатия кнопки **Готово** (Done) появится новое окно, описывающее параметры подключения к нашей базе. Здесь в поле **Имя источника данных** (Data Source Name) напишем имя "PhoneDS", через которое будет осуществляться доступ к нашей базе. Для выбора пути к базе данных снимем флажок **Использовать текущий каталог** (Use Current Directory) и нажмем на кнопку Выбор каталога (Select Directory). В открывшемся окне укажем путь C:\Tabl и нажмем **OK** (рис. 9.9).

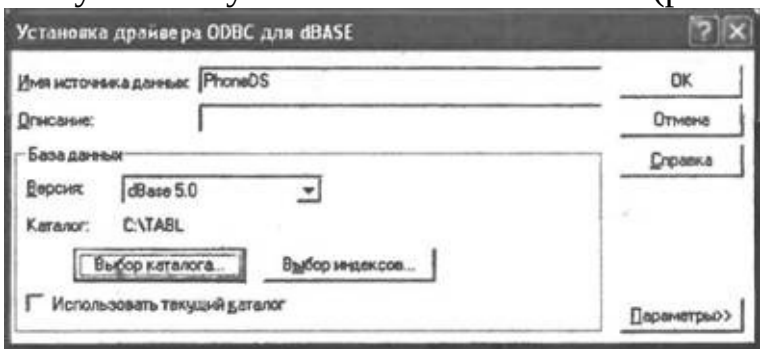


Рис. 9.9. Параметры источника данных ODBC

Для завершения настройки DSN нажмем кнопку OK и закроем окно ODBC.

Примеры сценариев

Начнем мы с того, что скопируем данные записной книжки из XML-файла book.xml в DBF-таблицу Phone.dbf (сценарий InsertRecords.js). Все остальные сценарии, которые рассматриваются в этой главе, будут работать уже с этой таблицей.

Копирование данных из XML-файла в таблицу БД

Напишем сценарий InsertRecords.js, который будет извлекать данные из XML-файла book.xml и добавлять записи с этими данными в таблицу Phone.dbf, для доступа к которой мы предварительно создали DSN (рис. 9.9).

Сценарий InsertRecords.js будет состоять из нескольких функций, главной из которых является `Main()`. В этой функции сначала создается объект `WshShell` и определяется путь к XML-файлу, который должен находиться в текущем каталоге:

```
//Создаем объект WshShell
WshShell = WScript.CreateObject("WScript.Shell");
//Путь к XML-файлу с данными
PathBook=WshShell.CurrentDirectory+"\\book.xml";
```

Для доступа к таблице Phone мы создаем объект `Connection`, который позволяет с помощью метода `Open()` устанавливать связь с заданной базой данных. Для этого необходимо в качестве параметра `Open()` указать строку с именем источника данных, к которому происходит обращение (в нашем случае эта строка имеет вид "DSN=PhoneDS"):

```
//Создаем объект Connection
Connect=WScript.CreateObject("ADODB.Connection");
//Формируем строку с параметрами соединения с БД
//(указываем нужный DSN)
SConnect="DSN=PhoneDS";
//Устанавливаем связь с БД
Connect.Open(SConnect);
```

После этого происходит вызов функции `XMLToBase()`, в которой происходит разбор XML-файла с помощью объектной модели XML DOM (применение XML DOM было подробно описано в *главе* 7).

```
//Копирование данных из XML-файла в таблицу Phone
function XMLToBase() {
 var XML,Root,NomRec,CurrNode,i;
```

```
      //Создаем объект XML DOM
      XML = WScript.CreateObject("Msxml.DOMDocument");
      //Загружаем XML-документ из файла
      XML.load(PathBook);
       //Сохраняем в переменной Root ссылку на корневой элемент документа
      Root=XML.documentElement;
      //Перебираем все дочерние элементы первого уровня вложенности
      //для корневого элемента
      for (i=1; i<=Root.childNodes.length-1;i++) {
       //Выделяем в коллекции XML-элементов i-й элемент
       CurrNode=Root.childNodes.item(i);
       //Вставляем новую запись в таблицу Phone
       PersonToTable(CurrNode);
      }
     }
```

Как мы видим, в функции `XMLToBase()` определяется цикл `for`, в котором для каждого XML-элемента, содержащего данные об одном человеке, происходит вызов функции `PersonToTable(`*`XNode`*`)`. В функции `PersonToTable(`*`XNode`*`)` формируется SQL-запрос `INSERT INTO Phone…`, который позволяет вставить в таблицу Phone новую запись с заданными значениями полей, например:

```
INSERT INTO Phone (LastName, Name, Phone, Street, House, App, Notes)
VALUES ('Иванов', 'Иван', '17-17-17', 'Садовая', '4', '6', 'Очень хороший человек')
```

Строится строка с SQL-запросом (переменная SSQL) следующим образом:

```
//Строим список значений полей добавляемой записи
SSQL+="'"+GetTagVal(XNode, "LastName")+"',";
SSQL+="'"+GetTagVal(XNode, "Name")+"',";
SSQL+="'"+GetTagVal(XNode, "Phone")+"',";
SSQL+="'"+GetTagVal(XNode, "Street")+"',";
SSQL+="'"+GetTagVal(XNode, "House")+"',";
SSQL+="'"+GetTagVal(XNode, "App")+"',";
SSQL+="'"+GetTagVal(XNode, "Note")+"'";
//Формируем текст SQL-запроса для вставки записи
SSQL="INSERT INTO Phone (LastName, Name, Phone, Street, House, App, Notes) VALUES ("+SSQL+")";
```

После формирования переменной SSQL происходит вызов SQL-запроса с помощью метода `Execute()` объекта `Connection`:

```
//Выполняем подготовленный SQL-запрос (добавляем запись в таблицу)
Connect.Execute(SSQL);
```

После окончания копирования данных в функции `Main()` выводится соответствующее сообщение:

```
//Выводим сообщение об окончании переноса данных
WshShell.Popup("Данные из XML-файла в таблицу перенесены!", 0, "Работа с базой данных", vbInformation+vbOkOnly);
```

Полностью текст сценария InsertRecords.js приведен в листинге 9.1.

Листинг 9.1. Копирование данных из XML-файла в таблицу БД

```
/*****************************************************************
/*                                                          Имя: InsertRecords.js                                          */
/*                                                         Язык: JScript                                                  */
/*   Описание:   Копирование   данных   из   XML-файла   таблицу базы          */
/*
данных                                                  */
/*****************************************************************
//Объявляем переменные
var
 WshShell, //Экземпляр объекта WshShell
 BasePath, //Путь к текущему каталогу
 PathBook, //Путь к файлу с данными
 NomRec=0, //Счетчик количества записей
 SConnect, //Строка с параметрами соединения с БД
 Connect;  //Экземпляр объекта Connection
//Инициализируем константы для диалоговых окон
var vbInformation=64,vbOkOnly=0;

//Определение значения тега tgName XML-элемента obj
function GetTagVal(obj, tgName) {
 var ElemList;
 //Создаем коллекцию дочерних для obj элементов, которые
 //задаются тегом tgName
 ElemList=obj.getElementsByTagName(tgName);
 //Проверяем, есть ли в коллекции ElemList элементы
 if (ElemList.length>0)
  //Возвращаем значение тега tgName
  return ElemList.item(0).text
 else return "";
}
//Вставка в таблицу одной записи
function PersonToTable(XNode) {
```

```
     var SSQL="";  //Переменная для формирования текста SQL-запроса
     //Строим список значений полей добавляемой записи
     SSQL+="'"+GetTagVal(XNode,"LastName")+"',";
     SSQL+="'"+GetTagVal(XNode,"Name")+"',";
     SSQL+="'"+GetTagVal(XNode,"Phone")+"',";
     SSQL+="'"+GetTagVal(XNode,"Street")+"',";
     SSQL+="'"+GetTagVal(XNode,"House")+"',";
     SSQL+="'"+GetTagVal(XNode,"App")+"',";
     SSQL+="'"+GetTagVal(XNode,"Note")+"'";
     //Формируем текст SQL-запроса для вставки записи
                  SSQL="INSERT              INTO              Phone
(LastName,Name,Phone,Street,House,App,Notes) VALUES ("+SSQL+")";
      //Выполняем подготовленный SQL-запрос (добавляем запись в
таблицу)
     Connect.Execute(SSQL);
    }
    //Копирование данных из XML-файла в таблицу Phone
    function XMLToBase() {
     var XML,Root,NomRec,CurrNode,i;
     //Создаем объект XML DOM
     XML = WScript.CreateObject("Msxml.DOMDocument");
     //Загружаем XML-документ из файла
     XML.load(PathBook);
      //Сохраняем в переменной Root ссылку на корневой элемент
документа
     Root=XML.documentElement;
     //Перебираем все дочерние элементы первого уровня вложенности
     //для корневого элемента
     for (i=1; i<=Root.childNodes.length-1;i++) {
      //Выделяем в коллекции XML-элементов i-й элемент
      CurrNode=Root.childNodes.item(i);
      //Вставляем новую запись в таблицу Phone
      PersonToTable(CurrNode);
     }
    }


    //Основная запускная функция
    function Main() {
     //Создаем объект WshShell
     WshShell = WScript.CreateObject("WScript.Shell");
     //Путь к XML-файлу с данными
     PathBook=WshShell.CurrentDirectory+"\\book.xml";
     //Создаем объект Connection
     Connect=WScript.CreateObject("ADODB.Connection");
     //Формируем строку с параметрами соединения с БД
     //(указываем нужный DSN)
     SConnect="DSN=PhoneDS";
```

```
 //Устанавливаем связь с БД
 Connect.Open(SConnect);
 //Копируем данные из XML-файла в таблицу БД
 XMLToBase();
 //Выводим сообщение об окончании переноса данных
 WshShell.Popup("Данные из XML-файла в таблицу перенесены!", 0,
  "Работа с базой данных",vbInformation+vbOkOnly);
}
/*******************                                    Начало
*********************************/
Main();
/*************                                          Конец
*********************************************/
```

Просмотр записей в таблице

После того как мы произвели копирование записной книжки в таблицу Phone.dbf, возникает естественное желание просмотреть из сценария введенные записи. Напишем для этой цели сценарий ListRecords1.js, результат работы которого представлен на рис. 9.10.

Рис. 9.10. Записи из таблицы Phone

Основным объектом, позволяющим получить доступ к записям таблицы, является `Recordset`, который представляет собой набор записей, полученных, например, в результате выполнения SQL-запроса. Создается объект `Recordset` следующим образом (экземпляром объекта будет переменная `RS`):

```
//Создаем объект Recordset
RS=WScript.CreateObject("ADODB.Recordset");
```

Все строки из таблицы Phone выбираются с помощью следующего SQL- запроса:

```
SELECT * from Phone
```

Этот запрос записывается в переменную `SSource`:

```
//Формируем SQL-запрос к таблице Phone
SSource = "SELECT * FROM Phone";
```

Для занесения в RS требуемого набора записей используется метод

Open(), в качестве параметров которого указываются две строки. В первой из них должен содержаться SQL-запрос (переменная SSource), а во второй — параметры соединения с базой данных (в нашем случае это переменная SConnect, в которой записан нужный DSN):

```
//Формируем строку с параметрами соединения с БД
//(указываем нужный DSN)
SConnect = "DSN=PhoneDS";
//Открываем набор записей - результат запроса
RS.Open(SSource, SConnect);
```

Текст, который будет в конце сценария выведен на экран, будет храниться в переменной SOut. Первоначально в эту переменную записываем заголовок:

```
SOut="ВСЕ ЗАПИСИ ИЗ ТАБЛИЦЫ phone.dbf:\n";
```

Для перемещения по записям набора RS используется так называемый *курсор*, который после выполнения метода Open() автоматически устанавливается на первую запись. Перебор всех записей производится в цикле while, условием выхода из которого является перемещение курсора за последнюю запись таблицы (при этом свойство EOF объекта Recordset станет равным True).

```
//Перебираем все записи набора данных RS
while (!RS.EOF) {
 …
}
```

Для доступа к полям текущей записи используется коллекция Fields, содержащая значения всех полей; чтобы получить значение конкретного поля, нужно указать в круглых скобках имя этого поля в кавычках, скажем, RS.Fields("Name"). Метод MoveNext() выполняет переход к следующей записи таблицы:

```
while (!RS.EOF) {
 //Формируем строку со значениями трех полей, которые разделены
 //символами табуляции
            s=RS.Fields("LastName")+"\t"+RS.Fields("Name")+"\t"+
RS.Fields("Phone");
 //В конце строки ставим символ перевода строки
 s+="\n";
 //Добавляем сформированную строку к переменной SOut
 SOut+=s;
 //Переходим к следующей записи
 RS.MoveNext();
}
```

Закрывается объект Recordset с помощью метода Close():

```
//Закрываем объект Recordset
RS.Close();
```

После этого сформированный в переменной sout текст выводится на экран:

```
//Выводим на экран строку SOut
WScript.Echo(SOut);
```

Полностью сценарий ListRecords1.js приведен в листинге 9.2.

Листинг 9.2. Просмотр записей таблицы

```
/*****************************************************************
/*                                                          Имя:
ListRecords1.js                                            */
/*                                                         Язык:
JScript                                                   */
/*    Описание:    Просмотр    записей    из    таблицы    базы
данных                */
/*****************************************************************
//Объявляем переменные
var
 RS,       //Экземпляр объекта Recordset
 SSource,  //Строка с текстом SQL-запроса к БД
 SConnect, //Строка с параметрами соединения с БД
 SOut,     //Строка, в которой сохраняется выходная информация
 s;
//Формируем SQL-запрос к таблице Phone
SSource = "SELECT * FROM Phone";
//Формируем строку с параметрами соединения с БД
//(указываем нужный DSN)
SConnect = "DSN=PhoneDS";
//Создаем объект Recordset
RS=WScript.CreateObject("ADODB.Recordset");
//Открываем набор записей-результат запроса
RS.Open(SSource,SConnect);
SOut="ВСЕ ЗАПИСИ ИЗ ТАБЛИЦЫ phone.dbf:\n";
//Перебираем все записи набора данных RS
while (!RS.EOF) {
 //Формируем строку со значениями трех полей, которые разделены
 //символами табуляции
 s=RS.Fields("LastName")+"\t"+RS.Fields("Name")+"\t"+RS.Fields("
 //В конце строки ставим символ перевода строки
 s+="\n";
 //Добавляем сформированную строку к переменной SOut
 SOut+=s;
 //Переходим к следующей записи
 RS.MoveNext();
```

```
    }
    //Закрываем объект Recordset
    RS.Close();
    //Выводим на экран строку SOut
    WScript.Echo(SOut);
    /*************                                                Конец
*******************************************/
```

Для получения набора записей `Recordset` можно не создавать его непосредственно, как это было сделано в ListRecords1.js, а воспользоваться методом `Execute()` объекта `Connection`. В качестве иллюстрации такого подхода рассмотрим сценарий ListRecords2.js (листинг 9.3).

Сначала в этом сценарии создается объект `Connection` и устанавливается связь с нашей базой данных (`DSN=PhoneDs`):

```
    //Формируем строку с параметрами соединения с БД
    //(указываем нужный DSN)
    SConnect = "DSN=PhoneDS";
    //Создаем объект Connection
    Connect=WScript.CreateObject("ADODB.Connection");
    //Устанавливаем связь с БД
    Connect.Open(SConnect);
```

Для получения всех записей из таблицы Phone используется тот же запрос, что и в сценарии ListRecords1.js:

```
    //Формируем SQL-запрос к таблице Phone
    SSource = "SELECT * FROM Phone";
    //Выполняем SQL-запрос, в результате создается объект Recordset
    RS=Connect.Execute(SSource);
```

В результате переменная `RS` становится нужным нам экземпляром объекта `Recordset`. После этого записи из набора `RS` обрабатываются точно так же, как в сценарии ListRecords1.js.

Листинг 9.3. Просмотр записей таблицы с использованием объекта Connection

```
    /*************************************************************
    /*                                                          Имя:
ListRecords2.js                                            */
    /*                                                         Язык:
JScript                                                   */
    /*    Описание:    Просмотр    записей    из    таблицы    базы
данных                 */
    /*                                    с    использованием    объекта
```

```
Connection                    */
    /*************************************************************
    //Объявляем переменные
    var
     RS,       //Экземпляр объекта Recordset
     Connect,  //Экземпляр объекта Connection
     SSource,  //Строка с текстом SQL-запроса к БД
     SConnect, //Строка с параметрами соединения с БД
     SOut,     //Строка, в которой сохраняется выходная информация
     s;
    //Формируем SQL-запрос к таблице Phone
    SSource = "SELECT * FROM Phone";
    //Формируем строку с параметрами соединения с БД
    //(указываем нужный DSN)
    SConnect = "DSN=PhoneDS";
    //Создаем объект Connection
    Connect=WScript.CreateObject("ADODB.Connection");
    //Устанавливаем связь с БД
    Connect.Open(SConnect);
    //Выполняем SQL-запрос, в результате создается объект Recordset
    RS=Connect.Execute(SSource);
    SOut="ВСЕ ЗАПИСИ ИЗ ТАБЛИЦЫ phone.dbf:\n";
    //Перебираем все записи набора данных RS
    while (!RS.EOF) {
     //Формируем строку со значениями трех полей, которые разделены
     //символами табуляции
     s=RS.Fields("LastName")+"\t"+RS.Fields("Name")+"\t"+RS.Fields('
     //В конце строки ставим символ перевода строки
     s+="\n";
     //Добавляем сформированную строку к переменной SOut
     SOut+=s;
     //Переходим к следующей записи
     RS.MoveNext();
    }
    //Закрываем объект Recordset
    RS.Close();
    //Выводим на экран строку SOut
    WScript.Echo(SOut);
    /*************                                          Конец
*****************************************/
```

Получение информации о полях и записях таблицы

Иногда бывает необходимо определить число и названия полей или подсчитать общее число записей в таблице. Приведенный в листинге 9.4

сценарий TableInfo.js выводит на экран диалоговое окно, где приведены названия всех полей и общее количество записей таблицы Phone.dbf (рис. 9.11).

Рис. 9.11. Информация о полях и записях таблицы Phone

В TableInfo.js для доступа к таблице Phone создается объект `Recordset`:

```
//Формируем SQL-запрос к таблице Phone
SSource = "SELECT * FROM Phone";
//Формируем строку с параметрами соединения с БД
//(указываем нужный DSN)
SConnect = "DSN=PhoneDS";
//Создаем объект Recordset
RS=WScript.CreateObject("ADODB.Recordset");
```

Перед открытием набора записей с помощью метода `Open()` нужно установить свойство `CursorType` объекта `Recordset` равным 3 (при этом создается статическая копия таблицы, что позволяет получить информацию о полях и записях):

```
//Задаем статический курсор
RS.CursorType = 3;
//Открываем набор записей - результат запроса
RS.Open(SSource, SConnect);
```

После этого в коллекции `Fields` будут содержаться все поля набора записей `Recordset`, а в свойстве `RecordCount` — число записей в этом наборе:

```
//Определяем число полей в наборе данных
RS_NomFields = RS.Fields.Count;
//Определяем число записей в наборе данных
RS_NomRecs = RS.RecordCount;
```

В переменной `SOut` будет формироваться текст для вывода на экран:

```
SOut="ТАБЛИЦА Phone СОДЕРЖИТ "+NomFields+" ПОЛЯ(ЕЙ):\n\n";
```

Для получения списка полей мы перебираем в цикле `for` все элементы коллекции `Fields`:

```
//Перебираем все поля набора данных RS
for (i=0; i<RS.Fields.Count; i++) {
```

```
 //Формируем строку с номером и именем поля таблицы
 SOut+"Поле N "+i+": "+RS.Fields(i).Name+"\n";
}
```

После завершения цикла к переменной `SOut` добавляется информация о количестве записей, набор записей `RS` закрывается и значение переменной `SOut` выводится на экран:

```
SOut+="\nИ "+NomRecs+" ЗАПИСЬ(ЕЙ)";
//Закрываем объект Recordset
RS.Close();
//Выводим на экран строку SOut
WScript.Echo(SOut);
```

Листинг 9.4. Получение информации о полях таблицы

```
/*****************************************************************
/*                                                          Имя:
TableInfo.js                                                 */
/*                                                         Язык:
JScript                                                     */
/*      Описание:      Получение      информации      о      полях
таблицы                  */
/*****************************************************************
//Объявляем переменные
var
 RS,       //Экземпляр объекта Recordset
 SSource,  //Строка с текстом SQL-запроса к БД
 SConnect, //Строка с параметрами соединения с БД
 SOut,     //Строка, в которой сохраняется выходная информация
 NomField, //Количество полей в таблице
 NomRecs,  //Количество записей в таблице
 i;
//Формируем SQL-запрос к таблице Phone
SSource = "SELECT * FROM Phone";
//Формируем строку с параметрами соединения с БД
//(указываем нужный DSN)
SConnect = "DSN=PhoneDS";
//Создаем объект Recordset
RS=WScript.CreateObject("ADODB.Recordset");
//Задаем статический курсор
RS.CursorType = 3;
//Открываем набор записей-результат запроса
RS.Open(SSource,SConnect);
//Определяем число полей в наборе данных RS
NomFields = RS.Fields.Count;
```

```
//Определяем число записей в наборе данных RS
NomRecs = RS.RecordCount;
SOut="ТАБЛИЦА Phone СОДЕРЖИТ "+NomFields+" ПОЛЯ(ЕЙ):\n\n";
//Перебираем все поля набора данных RS
for (i=0; i<RS.Fields.Count; i++) {
 //Формируем строку с номером и именем поля таблицы
 SOut+="Поле N "+i+": "+RS.Fields(i).Name+"\n";
}
SOut+="\nИ "+NomRecs+" ЗАПИСЬ(ЕЙ)";
//Закрываем объект Recordset
RS.Close();
//Выводим на экран строку SOut
WScript.Echo(SOut);
/*************                                          Конец
*******************************************/
```

Сортировка записей в таблице

Используя предложение `ORDER BY` в операторе `SELECT`, можно получить из источника данных записи, отсортированные по какому-либо полю. Например, следующий SQL-запрос

```
SELECT * FROM Phone ORDER BY LastName
```

вернет упорядоченный по полю `LastName` набор, содержащий все записи из таблицы Phone (рис. 9.12).

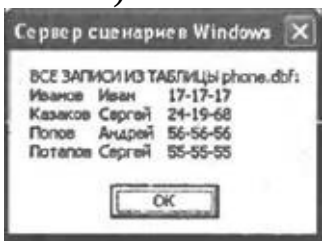


Рис. 9.12. Сортировка записей таблицы Phone по полю LastName

Мы напишем сценарий SortRecords.wsf, с помощью которого можно будет выводить записи из таблицы Phone, отсортированные либо по фамилии (поле `LastName`), либо по телефону (поле `Phone`). Выбор осуществляется с помощью ввода соответствующего значения в диалоговом окне (рис. 9.13).

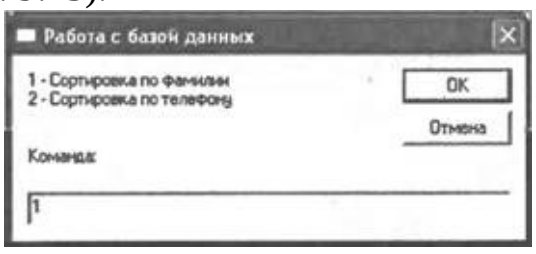


Рис. 9.13. Выбор поля для сортировки

Само диалоговое окно реализуется с помощью VBScript-функции `WSHInputBox(`*`Message, Title`*`)`:

```
Function WSHInputBox(Message, Title)
 'Выводим диалоговое окно со строкой ввода
 WSHInputBox = InputBox(Message,Title)
End Function
```

Эта функция вызывается из написанной на JScript части сценария следующим образом:

```
//Формируем текст сообщения в диалоговом окне ввода
SMenu="1 - Сортировка по фамилии\n";
SMenu+="2 - Сортировка по телефону\n";
SMenu+="\n\nКоманда:";
//Выводим диалоговое окно для ввода режима сортировки
Res=WSHInputBox(SMenu, "Работа с базой данных");
```

Введенное в диалоговом окне значение записывается в переменную `Res` и анализируется в операторе `switch`:

```
//Анализируем введенное значение
switch (Res) {
case "1": {
 SSort="LastName";
 break;
}
case "2": {
 SSort="Phone";
 break;
}
default: {
 WScript.Echo("Вы ввели неправильное значение!");
 WScript.Quit();
}
}
```

После того как в переменную `SSort` занесено имя поля, по которому нужно производить сортировку, мы можем полностью сформировать нужный SQL-запрос к таблице Phone:

```
//Формируем SQL-запрос к таблице Phone SSource = "SELECT * FROM Phone ORDER BY "+SSort;
```

Оставшаяся часть сценария взята из сценария ListRecords1.js и комментариев не требует.

Полностью текст сценария SortRecords.wsf приведен в листинге 9.5.

Листинг 9.5. Сортировка записей в таблице по заданному полю

```
    <job id="SortRecs">
    <runtime>
    <description>
    Имя: SortRecords.wsf
    Описание: Сортировка записей таблицы БД по заданному полю
    </description>
    </runtime>
    <script language="VBScript">
    'Функция для реализации диалогового окна со строкой ввода
    'в сценариях JScript
    Function WSHInputBox(Message,Title)
    'Выводим диалоговое окно со строкой ввода
    WSHInputBox = InputBox(Message,Title)
    End Function
    </script>
    <script language="JScript">
    //Объявляем переменные
    var
     RS,       //Экземпляр объекта Recordset
     SSource,  //Текст SQL-запроса к БД
     SConnect, //Строка с параметрами соединения с БД
     SOut,     //Строка, в которой сохраняется выходная информация
     Res,      //Результат ввода в диалоговом окне
     SSort,     //Имя поля таблицы, по которому будет производиться
сортировка
     SMenu,    //Текст сообщения в диалоговом окне ввода
     s;
    //Формируем текст сообщения в диалоговом окне ввода
    SMenu="1 - Сортировка по фамилии\n";
    SMenu+="2 - Сортировка по телефону\n";
    SMenu+="\n\nКоманда:";
    //Выводим диалоговое окно для ввода режима сортировки
    Res=WSHInputBox(SMenu,"Работа с базой данных");
    //Анализируем введенное значение
     switch (Res) {
     case "1": {
      SSort="LastName";
      break;
     }
     case "2": {
      SSort="Phone";
      break;
     }
     default: {
      WScript.Echo("Вы ввели неправильное значение!");
      WScript.Quit();
     }
```

```
}
//Формируем SQL-запрос к таблице Phone
SSource = "SELECT * FROM Phone ORDER BY "+SSort;
//Формируем строку с параметрами соединения с БД
//(указываем нужный DSN)
SConnect = "DSN=PhoneDS";
//Создаем объект Recordset
RS=WScript.CreateObject("ADODB.Recordset");
//Открываем набор записей-результат запроса
RS.Open(SSource,SConnect);
SOut="ВСЕ ЗАПИСИ ИЗ ТАБЛИЦЫ phone.dbf:\n";
//Перебираем все записи набора данных RS
while (!RS.EOF) {
 //Формируем строку со значениями трех полей, которые разделены
 //символами табуляции
 s=RS.Fields("LastName")+"\t"+RS.Fields("Name")+"\t"+RS.Fields('
 //В конце строки ставим символ перевода строки
 s+="\n";
 //Добавляем сформированную строку к переменной SOut
 SOut+=s;
 //Переходим к следующей записи
 RS.MoveNext();
}
//Закрываем объект Recordset
RS.Close();
//Выводим на экран строку SOut
WScript.Echo(SOut);
</script>
</job>
```

Фильтрация записей в таблице

Оператор `SELECT` позволяет выбирать из источника данных не все записи, а лишь те, которые удовлетворяют определенному критерию. Для этой цели в операторе `SELECT` применяется оператор `WHERE`. Например, следующий SQL-запрос

```
SELECT * FROM Phone WHERE (LastName LIKE 'П%')
```

вернет только те записи таблицы Phone, у которых значение поля `LastName` начинается на букву 'П' (шаблон '%' означает любое число любых символов).

Мы напишем сценарий FilterRecords.wsf, в котором можно ввести в диалоговом окне один или несколько символов (рис. 9.14) и получить список людей, фамилии которых начинаются с этих символов (рис. 9.15).

Рис. 9.14. Ввод первых символов фамилии для фильтрации записей

Рис. 9.15. Отфильтрованные в сценарии FilterRecords.wsf записи

Как и в сценарии SortRecords.wsf, символы в диалоговом окне вводятся с помощью VBScript-функции `WSHInputBox()`:

```
    //Выводим диалоговое окно для ввода первой буквы фамилии
    Res=WSHInputBox("Введите первые буквы фамилии", "Работа с базой данных");
```

После этого формируется строка с нужным SQL-запросом (переменная `SSource`) и строка с параметрами соединения с базой данных (переменная `SConnect`):

```
    //Формируем шаблон, по которому будет производиться фильтрация
    SFilt="'"+Res+"%'";
    //Формируем SQL-запрос к таблице Phone
    SSource = "SELECT * FROM Phone WHERE (LastName LIKE "+SFilt+")";
    //Формируем строку с параметрами соединения с БД
    //(указываем нужный DSN)
    SConnect = "DSN=PhoneDS";
```

Создав объект `Recordset` (переменная `RS`), мы присваиваем значение 3 свойству `CursorType` (это позволит узнать количество записей в наборе `RS` после выполнения SQL-запроса):

```
    //Создаем объект Recordset
    RS=WScript.CreateObject("ADODB.Recordset");
    //Задаем статический курсор
    RS.CursorType = 3;
    //Открываем набор записей - результат запроса
    RS.Open(SSource, SConnect);
    //Определяем число записей в наборе
    RS_NomRecs = RS.RecordCount;
```

Если в наборе RS не окажется ни одной записи (нет фамилий,

начинающихся на нужные символы), то будет выведено соответствующее сообщение и произойдет выход из сценария:

```
if (NomRecs==0) {
   WScript.Echo("В таблице Phone нет ни одной фамилии,
начинающейся на '"+Res+"'");
 WScript.Quit();
}
```

Если же подходящие записи найдены, то они, как обычно, обрабатываются в цикле `for`. В результате формируется строка `SOut` со значениями полей `LastName`, `Name` и `Phone` для этих записей:

```
SOut="ВСЕГО "+NomRecs+" ЗАПИСЕЙ, НАЧИНАЮЩИХСЯ НА '"+Res+"':\n";
//Перебираем все записи набора данных RS
while (!RS.EOF) {
 //Формируем строку со значениями трех полей, которые разделены
 //символами табуляции
           s=RS.Fields("LastName")+"\t"+RS.Fields("Name")+"\t"+
RS.Fields("Phone");
 //В конце строки ставим символ перевода строки
 s+="\n";
 //Добавляем сформированную строку к переменной SOut
 SOut+=s;
 //Переходим к следующей записи
 RS.MoveNext();
}
```

В конце сценария объект `Recordset` закрывается, а строка `SOut` выводится на экран:

```
//Закрываем объект Recordset
RS.Close();
//Выводим на экран строку SOut
WScript.Echo(SOut);
```

Полностью текст сценария FilterRecords.wsf приведен в листинге 9.6.

Листинг 9.6. Фильтрация записей в таблице

```
<job id="FiltRecs">
<runtime>
<description>
Имя: FilterRecords.wsf
Описание: Фильтрация записей таблицы Phone по первому символу
          фамилии
</description>
</runtime>
<script language="VBScript">
```

```
    'Функция для реализации диалогового окна со строкой ввода
    'в сценариях JScript
    Function WSHInputBox(Message,Title)
    'Выводим диалоговое окно со строкой ввода
    WSHInputBox = InputBox(Message,Title)
    End Function
    </script>
    <script language="JScript">
    //Объявляем переменные
    var
     RS,       //Экземпляр объекта Recordset
     SSource,  //Текст SQL-запроса к БД
     SConnect, //Строка с параметрами соединения с БД
     SOut,     //Строка, в которой сохраняется выходная информация
     Res,      //Результат ввода в диалоговом окне
     SFilt,    //Шаблон, по которому будет производиться фильтрация
     NomRecs,  //Количество записей в отфильтрованном наборе
     s;
    //Выводим диалоговое окно для ввода первой буквы фамилии
    Res=WSHInputBox("Введите первые буквы фамилии","Работа с базой
данных");
    //Формируем шаблон, по которому будет производиться фильтрация
    SFilt="'"+Res+"%'";
    //Формируем SQL-запрос к таблице Phone
    SSource  =  "SELECT  *  FROM  Phone  WHERE  (LastName  LIKE
"+SFilt+")";
    //Формируем строку с параметрами соединения с БД
    //(указываем нужный DSN)
    SConnect = "DSN=PhoneDS";
    //Создаем объект Recordset
    RS=WScript.CreateObject("ADODB.Recordset");
    //Задаем статический курсор
    RS.CursorType = 3;
    //Открываем набор записей-результат запроса
    RS.Open(SSource,SConnect);
    //Определяем число записей в наборе RS
    NomRecs = RS.RecordCount;
    if (NomRecs==0) {
       WScript.Echo("В  таблице  Phone  нет  ни  одной  фамилии,
начинающейся на '" + Res+"'");
     WScript.Quit();
    }
    SOut="ВСЕГО "+NomRecs+" ЗАПИСЕЙ, НАЧИНАЮЩИХСЯ НА '"+Res+"':\n";
    //Перебираем все записи набора данных RS
    while (!RS.EOF) {
     //Формируем строку со значениями трех полей, которые разделены
     //символами табуляции
```

```
 s=RS.Fields("LastName")+"\t"+RS.Fields("Name")+"\t"+RS.Fields('
 //В конце строки ставим символ перевода строки
 s+="\n";
 //Добавляем сформированную строку к переменной SOut
 SOut+=s;
 //Переходим к следующей записи
 RS.MoveNext();
}
//Закрываем объект Recordset
RS.Close();
//Выводим на экран строку SOut
WScript.Echo(SOut);
</script>
</job>
```

Перемещение в наборе записей

Во всех рассмотренных выше сценариях мы перемещались в наборе `Recordset` сверху вниз, от первой записи к последней. Существует, однако, возможность перемещаться по записям не только вперед, но и назад. Это осуществляется с помощью метода `MovePrevious()`, для использования которого нужно установить тип курсора (свойство `CursorType`) в объекте `Recordset` равным 1, 2 или 3.

Рассмотрим сценарий MoveInTable.js, в котором записи таблицы Phone выводятся в порядке, обратном физическому (рис. 9.16).

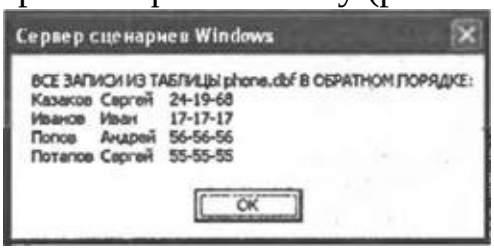


Рис. 9.16. Записи таблицы Phone в обратном порядке

Набор записей Recordset в этом сценарии открывается в режиме статической копии таблицы (свойство `CursorType` равно 3):

```
//Формируем SQL-запрос к таблице Phone
SSource = "SELECT * FROM Phone";
//Формируем строку с параметрами соединения с БД
//(указываем нужный DSN)
SConnect = "DSN=PhoneDS";
//Создаем объект Recordset
RS=WScript.CreateObject("ADODB.Recordset");
//Задаем статический курсор
```

```
RS.CursorType = 3;
```

После открытия набора записей мы переходим к последней записи с помощью метода `MoveLast()`:

```
//Открываем набор записей - результат запроса
RS.Open(SSource, SConnect);
//Переходим на последнюю запись
RS.MoveLast();
```

После этого записи перебираются в цикле `while`:

```
//Перебираем все записи набора данных RS
while (!RS.BOF) {
 //Формируем строку со значениями трех полей, которые разделены
 //символами табуляции
 s=RS.Fields("LastName")+"\t"+RS.Fields("Name")+"\t"+RS.Fields('
 //В конце строки ставим символ перевода строки
 s+="\n";
 //Добавляем сформированную строку к переменной SOut
 SOut+=s;
 //Переходим к предыдущей записи
 RS.MovePrevious();
}
```

Свойство `BOF`, используемое в цикле `while`, становится равным `true`, когда курсор будет находиться перед первой записью таблицы `RS`.

После выхода из цикла объект `Recordset` закрывается и сформированная строка `SOut` выводится на экран:

```
//Закрываем объект Recordset
RS.Close();
//Выводим на экран строку SOut
WScript.Echo(SOut);
```

Полностью текст сценария MoveInTable.js приведен в листинге 9.7.

Листинг 9.7. Перемещение в наборе записей снизу вверх

```
/*****************************************************************
/*                                                          Имя:
MoveInTable.js                                            */
/*                                                         Язык:
JScript                                                  */
/*  Описание:  Перемещение  по  набору  записей  в  обратном
порядке      */
/*****************************************************************
//Объявляем переменные
var
 RS,       //Экземпляр объекта Recordset
```

```
     SSource,  //Строка с текстом SQL-запроса к БД
     SConnect, //Строка с параметрами соединения с БД
     SOut,     //Строка, в которой сохраняется выходная информация
     s;
    //Формируем SQL-запрос к таблице Phone
    SSource = "SELECT * FROM Phone";
    //Формируем строку с параметрами соединения с БД
    //(указываем нужный DSN)
    SConnect = "DSN=PhoneDS";
    //Создаем объект Recordset
    RS=WScript.CreateObject("ADODB.Recordset");
    //Задаем статический курсор
    RS.CursorType = 3;
    //Открываем набор записей-результат запроса
    RS.Open(SSource,SConnect);
    //Переходим на последнюю запись
    RS.MoveLast();
    SOut="ВСЕ ЗАПИСИ ИЗ ТАБЛИЦЫ phone.dbf В ОБРАТНОМ ПОРЯДКЕ:\n";
    //Перебираем все записи набора данных RS
    while (!RS.BOF) {
     //Формируем строку со значениями трех полей, которые разделены
     //символами табуляции
     s=RS.Fields("LastName")+"\t"+RS.Fields("Name")+"\t"+RS.Fields('
     //В конце строки ставим символ перевода строки
     s+="\n";
     //Добавляем сформированную строку к переменной SOut
     SOut+=s;
     //Переходим к предыдущей записи
     RS.MovePrevious();
    }
    //Закрываем объект Recordset
    RS.Close();
    //Выводим на экран строку SOut
    WScript.Echo(SOut);
    /*************                                          Конец
*******************************************/
```

Доступ к БД без создания DSN

Напомним, что для получения доступа к базе данных мы предварительно создали DSN с именем PhoneDS (рис. 9.9). Однако связываться с базами данных можно и без описания DSN (DSN, Less Database Access). Для этого необходимо в качестве второго параметра метода `Open` объекта `Recordset` задать строку, в которой явно будут записаны название нужного драйвера и параметры соединения с базой. Для

используемой нами базы данных DBF-формата достаточно указать каталог, в котором находятся нужные таблицы (этот параметр называется `DefaultDir`). Например, если таблицы расположены в каталоге Tabl на диске C:, то строка `SConn` с параметрами соединения имеет следующий вид:

```
SConn="Driver={Microsoft dBase Driver (*.dbf)};DefaultDir=C:\\Tabl";
```

К положительным моментам доступа к данным без предварительного создания DSN можно отнести то, что строка соединения с базой формируется во время выполнения сценария (имя базы данных можно передавать в качестве параметра) — это позволяет писать более гибкие сценарии. Кроме этого, сценарий, не требующий предварительной настройки ODBC, легче переносить на другие машины. Недостаток этого подхода состоит в невозможности установить контроль над соединением с базой в той мере, в какой это позволяет сделать ODBC (это становится важным при работе с базой данных, находящейся в сети).

В листинге 9.8 приведен сценарий DSN_Less.js, в котором доступ к таблице Phone осуществляется без использования DSN (предполагается, что файл Phone.dbf находится в текущем каталоге).

Листинг 9.8. Доступ к базе Phone без использования DSN

```
/*****************************************************************
/*                                                          Имя: DSN_Less.js                                                  */
/*                                                          Язык: JScript                                                     */
/*  Описание:  Просмотр  записей  таблицы  без  использования DSN        */
/*****************************************************************
//Объявляем переменные
var
 WshShell,    //Экземпляр объекта WshShell
 RS,          //Экземпляр объекта Recordset
 SSource,     //Строка с текстом SQL-запроса к БД
 SConnect,    //Строка с параметрами соединения с БД
  SOut,             //Строка, в которой сохраняется выходная информация
  SDefaultDir, //Путь к каталогу, в котором находится таблица Phone
 s;
//Создаем объект WshShell
WshShell = WScript.CreateObject("WScript.Shell");
```

```
    //Определяем путь к текущему каталогу, в котором хранится
таблица Phone
    SDefaultDir=WshShell.CurrentDirectory;
    //Формируем SQL-запрос к таблице Phone
    SSource = "SELECT * FROM Phone";
    //Формируем строку с параметрами соединения с БД
    //(указываем нужный DSN)
    SConnect="Driver={Microsoft            dBase            Driver
(*.dbf)};DefaultDir="+SDefaultDir;
    //Создаем объект Recordset
    RS=WScript.CreateObject("ADODB.Recordset");
    //Открываем набор записей-результат запроса
    RS.Open(SSource,SConnect);
    SOut="ВСЕ ЗАПИСИ ИЗ ТАБЛИЦЫ phone.dbf:\n";
    //Перебираем все записи набора данных RS
    while (!RS.EOF) {
     //Формируем строку со значениями трех полей, которые разделены
     //символами табуляции
     s=RS.Fields("LastName")+"\t"+RS.Fields("Name")+"\t"+RS.Fields('
     //В конце строки ставим символ перевода строки
     s+="\n";
     //Добавляем сформированную строку к переменной SOut
     SOut+=s;
     //Переходим к следующей записи
     RS.MoveNext();
    }
    //Закрываем объект Recordset
    RS.Close();
    //Выводим на экран строку SOut
    WScript.Echo(SOut);
    /*************                                          Конец
*****************************************/
```

Глава 10
Разработка COM-объектов с помощью языков сценариев

Во всех сценариях WSH, которые мы рассматривали ранее, создавались экземпляры внешних COM-объектов, являющихся серверами автоматизации, после чего мы не задумываясь использовали свойства и методы этих объектов. В основном COM-объекты создаются в виде откомпилированных модулей с помощью универсальных языков типа Visual C++ или Visual Basic, однако с помощью специальной технологии Windows Script Components можно зарегистрировать в системе COM-объект, написанный на языке JScript или VBScript, причем для этого не нужно проводить никакой компиляции исходного кода сценария! Таким образом, любой сценарий WSH можно "упаковать" в COM-объект (мы будем называть такие объекты *компонентами-сценариями* или *объектами-сценариями*) и использовать его в приложениях, написанных на любом языке, который позволяет подключать внешние объекты. Преимуществами компонентов-сценариев над обычными откомпилированными компонентами (как и вообще любой интерпретируемой программы над компилируемой) является простота разработки, внесения изменений и распространения объектов-сценариев, недостатками — более медленная скорость работы и возможность внесения несанкционированных изменений в исходный код компонентов.

В этой главе мы кратко опишем механизм работы объектов-сценариев и внутреннюю структуру файлов с описанием таких объектов. Также будет подробно разобран пример создания, регистрации и использования объекта-сценария, предназначенного для создания архивных копий файлов.

Технология Windows Script Components

Сразу оговоримся, что мы будем рассматривать только объекты-сценарии, которые являются серверами автоматизации (т.е. предоставляют свои свойства и методы другим приложениям), не затрагивая вопросы разработки специальных объектов для использования внутри HTML- или ASP-страниц (Active Server Pages).

Механизм работы объектов-сценариев базируется на технологии ActiveX Scripting, основную роль здесь играет динамическая библиотека Scrobj.dll, которая является оболочкой компонентов-сценариев и отвечает за функционирование файла-сценария в качестве COM-объекта. С точки зрения технологии COM эта библиотека для объектов-сценариев действует как внутренний сервер (inprocess server). Другими словами, оболочка компонентов-сценариев Scrobj.dll отвечает за то, чтобы при вызове из внешнего приложения метода объекта-сценария или обращении к его свойству запускалась соответствующая функция, описанная в этом сценарии. Для этого на уровне операционной системы незаметно для самого объекта-сценария и для приложения, в котором создается экземпляр этого объекта, производятся следующие действия.

□ При регистрации объекта-сценария в разделе `HKEY_CLASSES_ROOT\CLSID` системного реестра создается новый подраздел, название которого совпадает с глобальным кодом (CLSID) регистрируемого объекта. В этом новом разделе создается подраздел `InprocServer32`**,** значением по умолчанию которого является полный путь к библиотеке Scrobj.dll. Кроме `InprocServer32`, создаются подразделы `ProgID` (программный идентификатор объекта) и `ScripletURL` (полный путь к файлу объекта-сценария).

□ Если создание экземпляра объекта из внешнего приложения происходит с помощью программного идентификатора (ProgID) объекта, то сначала определяется глобальный код (CLSID) этого объекта. Для этого в разделе реестра `HKEY_LOCAL_MACHINE\SOFTWARE\Classes` ищется подраздел с именем, которое совпадает с программным идентификатором объекта (этот подраздел создается при регистрации объекта-сценария), и в качестве глобального кода берется значение параметра `CLSID` из этого подраздела.

□ По известному глобальному коду объекта происходит поиск раздела с нужным названием в `HKEY_CLASSES_ROOT\CLSID`, после чего определяется значение подраздела `InprocServer32` (путь к библиотеке Scrobj.dll) и

загружается оболочка компонентов-сценариев Scrobj.dll.

□ Библиотека Scrobj.dll загружает указанный в подразделе `ScripletURL` файл со сценарием и перенаправляет вызовы методов объекта в этот сценарий.

Таким образом, нам остается лишь научиться правильным образом создавать файлы с исходным кодом компонентов-сценариев и регистрировать эти файлы в системе в качестве COM-объектов.

Компоненты-сценарии, реализованные в технологии Windows Script Components, представляют из себя файлы с расширениями wsc (WSC-файлы), которые содержат специальную XML-разметку (объектная модель WSC XML), к описанию которой мы и перейдем.

Схема WSC XML

Ранее в *главе 3* отмечалось, что объектная модель, которая используется при создании многозадачных сценариев (WS-файлов), была в основном позаимствована из схемы WSC XML, поэтому многие элементы WSC-файлов окажутся вам знакомыми.

В листинге 10.1 приводится несколько упрощенная (некоторые необязательные атрибуты у XML-элементов опущены) схема, поясняющая иерархию и порядок появления элементов в WSC-файле.

Листинг 10.1. Упрощенная схема WSC XML

```
    <?xml version="1.0" encoding="windows-1251"?>
    <package>
     <component [id="ComponentID"]>
            <registration    progid="ProgID"    class    id="GUID"
[description="Description"] [version="Version"]>
       <script>
    Сценарии подключения и отключения
       </script>
      </registration>
      <public>
       <property name="PropertyName">
        <get [internalName="getFunctionName"]/>
        <put [internalName="getFunctionName"]/>
       </property>
       <method name= "MethodName" [internalName="FunctionName"]>
        <parameter name="ParameterID"/>
       </method>
       <event name="Name" [dispid="DispID"]/>
      </public>
      <resource id="ResourceID"> Строка или число </resource>
      <object id="ObjID" [classld="clsid:GUID"|progid="ProgID"]/>
                <reference     [object="ProgiD"|guid="typelibGUID"]
[version="version"]/>
      <script language="language">
       <![CDATA[
        Код сценария
       ]]>
      </script>
     </component>
```

```
 Другие компоненты
</package>
```

Несколько замечаний относительно количества вхождений различных XML-элементов из листинга 10.1 в WSC-файл:

□ элемент `<package>` может содержать один или несколько элементов `<component>`;

□ элемент `<component>` должен содержать один элемент `<registration>` и один элемент `<public>`;

□ элемент `<public>` может содержать один или несколько элементов `<property>`, `<method>` или `<event>`.

Обязательными для создания компонента-сценария являются элементы `<component>`, `<registration>`, `<public>` и `<script>`.

Опишем теперь элементы XML, использующиеся в WSC-файлах, более подробно.

Элементы WSC-файла

В WSC-файлы можно вставлять комментарии двумя разными способами: с помощью элемента `<!-- -->` или элемента `<comment>`. Например:

```
<!-- Первый комментарий -->
```

или

```
<comment>
Второй комментарий
</comment>
```

Элементы <*?xml?*> и <*![CDATA[]]*>

Напомним, что эти элементы являются стандартными для разметки W3C XML 1.0 *(см. главу 3)*. Для того чтобы использовать символы кириллицы в файлах компонентов-сценариев, нужно обязательно в элементе `<?xml?>` указать атрибут `encoding` со значением, соответствующим используемой кодировке, например:

```
<?xml version="1.0" encoding="windows-1251"?>
```

Элемент <*package*>

Этот элемент необходим в тех WSC-файлах, в которых с помощью элементов `<component>` определено более одного компонента. В этом случае `<package>` является контейнером для элементов `<component>`.

Если же в WSC-файле описан только один компонент, то элемент `<package>` можно не использовать.

Элемент <*component*>

Внутри элемента `<component>` описывается один компонент-сценарий (COM-объект). Необязательный атрибут `id` определяет идентификатор объекта (это может понадобиться в том случае, когда в одном WSC-файле находится несколько COM-объектов).

Элемент <*registration*>

В элементе <registration> приводится информация, которая необходима для регистрации в системе компонента-сценария в качестве COM-объекта.

Атрибуты progid и classid задают соответственно программный идентификатор и глобальный код, с помощью которых компонент-сценарий может быть использован в других приложениях (например, progid="MyClass.MyObject" и classid="{424ac2bc-5732-4dea-be17-0211af99cd79}"). Из этих двух атрибутов обязательно должен быть указан хотя бы один (можно указать и оба). Если в элементе <registration> приведен только атрибут progid, то глобальный код (GUID) для описываемого объекта будет сгенерирован автоматически при регистрации объекта в системе. Рекомендуется, однако, явно указывать глобальный код объекта, т.к. в противном случае этот код может оказаться различным при регистрации объекта на разных машинах.

Замечание

Глобальный код объекта может быть сгенерирован с помощью описанной ниже программы Windows Script Component Wizard.

С помощью атрибута description можно задать краткое описание объекта, которое будет занесено в системный реестр при регистрации объекта.

Атрибут version позволяет указать номер версии описываемого объекта. Этот номер позволяет запрашивать из приложения определенную версию COM-объекта (он должен быть указан через точку после программного идентификатора объекта, например "Myclass.MyObject.1").

С помощью элемента <script> внутри контейнера <registration> можно указать две функции, одна из которых будет вызываться при регистрации объекта в системе (эта функция должна иметь имя Register()), а другая — при удалении объекта из системы (эта функция должна иметь имя Unregister()).

Элемент <*public*>

В элементе <public> описываются те свойства, методы и события объекта, которые после его регистрации будут доступны извне другим

приложениям (клиентам автоматизации). Другими словами, этот элемент является контейнером для элементов <property>, <method> и <event>.

Элемент <*property*>

Элемент <property> объявляет свойство COM-объекта, которое будет доступно для клиентов автоматизации.

Атрибут name определяет имя этого свойства (в дальнейшем внутри элемента <script> должна быть объявлена глобальная переменная с тем же именем, с помощью которой можно будет изменять значение свойства). Объявляемое свойство может быть доступно либо только для чтения (внутри контейнера <property> указан только элемент <get>), либо только для записи (внутри <property> указан только элемент <put>), либо и для чтения и для записи (внутри <property> указаны как элемент <get>, так и элемент <put>).

Атрибут internalName в элементах <get> и <put> задает имена функций, которые будут использоваться для чтения и записи свойства соответственно (эти функции описываются внутри контейнера <script>). Если же атрибут internalName не указан, то чтение (запись) свойства производится в функции с именем get_*PropertyName*(put_*PropertyName*), где *PropertyName* — имя свойства (атрибут <name>).

Элемент <*method*>

Элемент <method> объявляет метод COM-объекта, который будет доступен для внешних клиентов автоматизации.

Атрибут name определяет имя этого метода. В дальнейшем, если не указан атрибут internalName, внутри контейнера <script> должна быть описана функция или процедура с таким же именем

Задание атрибута internalName позволяет внутри контейнера <script> использовать для реализации метода функцию или процедуру с именем, отличным от значения аргумента name.

Список параметров метода (если они имеются) задается внутри элемента <method> с помощью элементов <parameter>, каждый из которых должен содержать аргумент name, определяющий имя параметра.

Элемент <*event*>

Элемент <event> объявляет событие, которое может возникать в COM-объекте и обрабатываться клиентами автоматизации.

Аргумент name, как обычно, определяет имя этого события. С помощью аргумента dispid можно указать числовой идентификатор интерфейса события. Этот идентификатор компилируется в библиотеку типов объекта и используется в клиентском приложении для обработки нужного события.

Для того чтобы вызвать наступление события, внутри элемента <script> используется функция fireEvent() с именем нужного события в качестве параметра.

Элементы <*resource*>, <*object*> и <*reference*>

Элементы <resource>, <object> и <reference> имеют тот же смысл, что и в модели WS XML *(см. описание этих элементов в главе 3).*

Элемент <*script*>

В элементе <script> приводится сценарий на языках JScript или VBScript, который определяет поведение создаваемого COM-объекта — здесь нужно задать глобальные переменные, соответствующие объявленным в элементах <property> свойствам объекта, описать функции или процедуры для объявленных с помощью элементов <method> методов объекта и т.д.

Перейдем теперь к рассмотрению конкретного примера, для которого мы подробно опишем создание компонента-сценария, регистрацию его в системе в качестве COM-объекта и использование этого объекта в JScript-сценарии.

Пример: COM-объект для архивирования файлов

Представим себе следующую ситуацию. Имеется несколько каталогов на жестком диске, в которых хранятся рабочие файлы разных пользователей. Необходимо каждый день с помощью программы-архиватора arj.exe делать архивы этих файлов в заданный каталог, при этом имя архивного файла должно соответствовать шаблону ппММДД.arj", где пп — уникальный для каждого пользователя префикс, ММ — текущий месяц, ДД — текущий день.

Мы создадим компонент-сценарий `DateArc.WSC`, с помощью свойств и методов которого можно будет выполнить поставленную задачу.

Начнем мы с того, что определимся, какие именно свойства и методы будет содержать создаваемый объект (табл. 10.1).

Таблица 10.1. *Свойства и методы объекта DateArc.WSC*

Название	Свойство/ метод	Режим доступа	Описание
`SFrom`	Свойство	Чтение/ запись	Исходный каталог для архивирования
`SArch`	Свойство	Чтение/ запись	Каталог, в котором хранятся архивные файлы
`SPref`	Свойство	Чтение/ запись	Префикс файла-архива
`SMask`	Свойство	Чтение/ запись	Маска, по которой отбираются файлы для архивирования
`SErrMess`	Свойство	Чтение	Текст сообщения об ошибке
`FilesToArchiv()`	Метод		Метод, осуществляющий архивирование файлов

Имея зарегистрированный в системе COM-объект с такими свойствами и методами, несложно написать сценарий (назовем его RunArj.js), в котором создавался бы экземпляр объекта `DateArc.WSC` и производилось с помощью метода `FilesToArchiv()` архивирование файлов из исходного в целевой каталог (листинг 10.2).

Листинг 10.2. Архивирование файлов с помощью COM-объекта DateArc.WSC

```
    /*****************************************************************
    /*                                                              Имя:
RunArj.js                                                        */
    /*                                                             Язык:
JScript                                                         */
    /*  Описание:  Архивирование  файлов  с  помощью  COM-объекта
DateArc.WSC */
    /*****************************************************************
    //Объявляем переменные
    var
     DateArc,  //Экземпляр объекта DateArc.WSC
     Result;   //Результат выполнения метода FilesToArchiv()
    //Инициализируем константы для диалоговых окон
    var vbCritical=16,vbInformation=64;
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    //Создаем объект DateArc.WSC
    DateArc=WScript.CreateObject("DateArc.WSC");
    DateArc.SFrom="D:\\1";  //Исходный каталог
    DateArc.SArch="D:\\2";       //Каталог,   в   который   будут
архивироваться
                            //файлы
    DateArc.SPref="aa";  //Префикс для файл-архива
    DateArc.SMask="*.*"; //Маска, по которой будут отбираться файлы
                         //для архивирования
    //Запускаем метод FilesToArchiv()
    Result=DateArc.FilesToArchiv();
    if (!Result)
     //Если возникла ошибка, выводим соответствующее сообщение
      WshShell.Popup(DateArc.SErrMess,  0,  "Архивирование  файлов",
vbCritical);
    else  WshShell.Popup("Архивирование завершено успешно!", 0,
     "Архивирование файлов", vbInformation);
    /*************                                                Конец
*******************************************/
```

Создание макета файла DateArc.wsc с помощью Windows Script Component Wizard (JScript)

Из листинга 10.1 можно понять, что создание компонента-сценария связано с написанием большого количества вспомогательного кода (нужно

заполнить элементы <registration>, <property>, <method> и <events>, написать функции для чтения и записи каждого из свойств объекта и т.д). Поэтому мы воспользуемся специальным мастером для создания компонентов-сценариев Windows Script Component Wizard (эту программу можно свободно получить с сайта Microsoft **http://msdn.microsoft.com/scripting**).

Первым шагом после запуска мастера является заполнение полей диалогового окна, представленного на рис. 10.1. Здесь нужно ввести имя создаваемого компонента ("DateArc"), имя файла с его описанием ("DateArc"), программный идентификатор ("DateArc.WSC"), версию компонента ("1.00") и каталог, в котором будет создан WSC-файл ("C:\WSC").

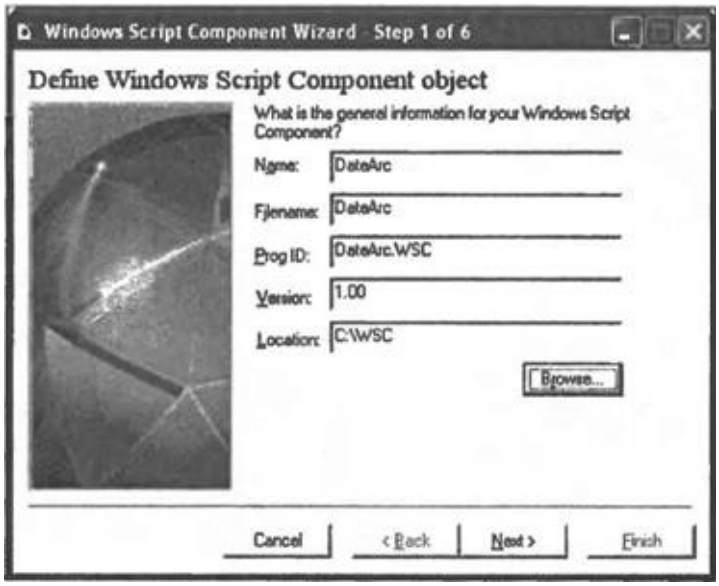


Рис. 10.1. Общая информация о создаваемом компоненте-сценарии

На втором шаге работы мастера мы выберем, какой язык будет использоваться при написании сценария ("JScript"), и укажем, что при регистрации и работе объекта нужно выполнять проверку ошибок ("Error checking") (рис. 10.2).

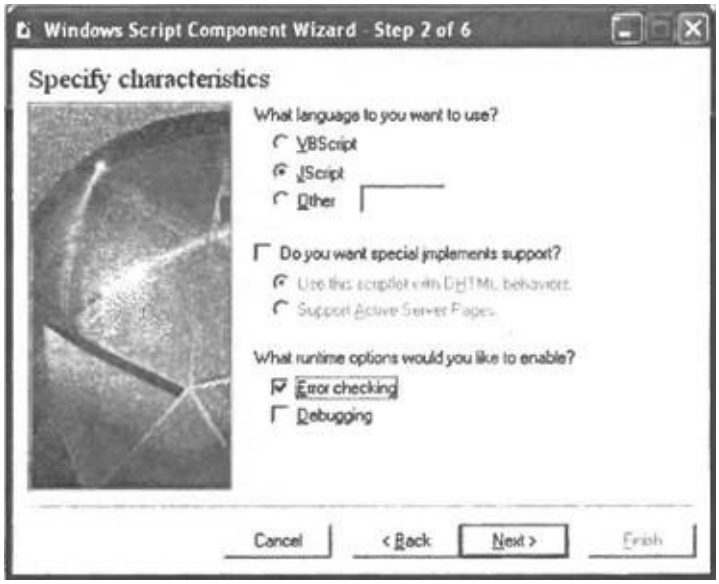


Рис. 10.2. Определение характеристик компонента-сценария

Третий шаг работы мастера позволяет описать свойства создаваемого объекта. Здесь для каждого свойства мы указываем его имя (колонка "Name"), тип ("Read" — только чтение, "Write" — только запись, "Read/Write" — чтение и запись) и значение по умолчанию (колонка "Default") (рис. 10.3).

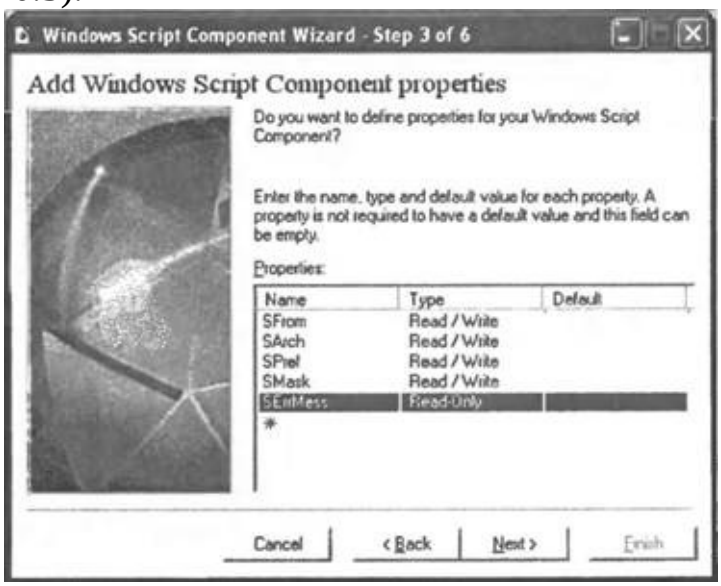


Рис. 10.3. Задание свойств компонента-сценария

На четвертом шаге описываются методы объекта. В нашем случае объект `DateArc.WSC` имеет единственный метод `FilesToArchiv()`, вызываемый без параметров (рис. 10.4).

Рис. 10.4. Задание методов компонента-сценария

На пятом шаге нам предлагается указать, какие события могут возникать в объекте. Для нашего компонента-сценария мы не будем задавать никаких событий (рис. 10.5).

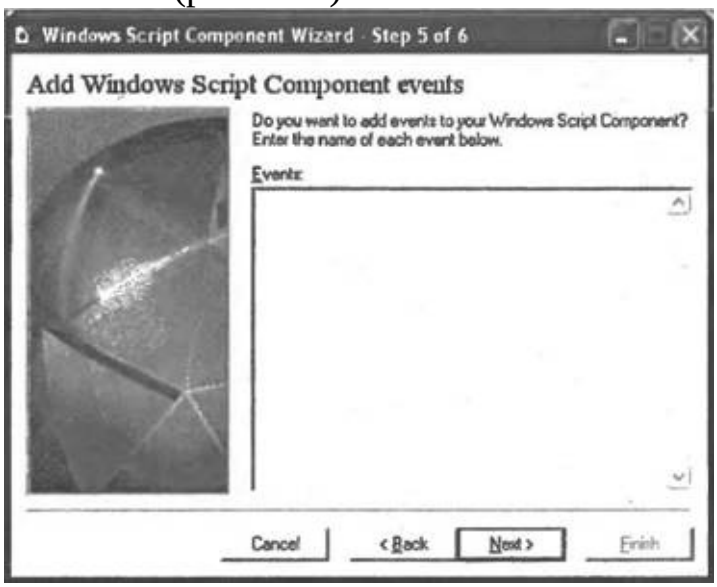


Рис. 10.5. Задание событий компонента-сценария

Шестой шаг является заключительным в работе мастера. Здесь нам выдается вся информация о создаваемом объекте (рис. 10.6). После нажатия кнопки **Finish** в каталоге C:\WSC будет создан файл DateArc.wsc, приведенный в листинге 10.3.

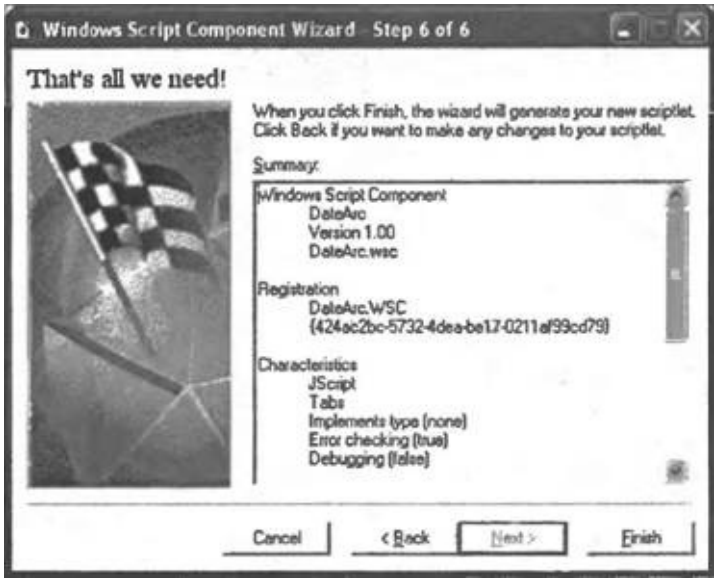


Рис. 10.6. Итоговая информация о создаваемом компоненте-сценарии

Листинг 10.3. Файл DateArc.wsc, сгенерированный программой Windows Script Component Wizard (JScript)

```
    <?xml version="l.0"?>
    <component>
     <?component error="true" debug="false"?>
       <registration   description="DateArc"   progid="DateArc.WSC"
version="1.00" classid="{424ac2bc-5732-4dea-bel7-0211af99cd79}">
     </registration>
     <public>
      <property name="SFrom">
       <get/>
       <put/>
      </property>
      <property name="SArch">
       <get/>
       <put/>
      </property>
      <property name="SPref">
       <get/>
       <put/>
      </property>
      <property name="SMask">
       <get/>
       <put/>
      </property>
```

```
  <property name="SErrMess">
   <get/>
  </property>
  <method name="FilesToArchiv">
  </method> </public>
  <script language="JScript">
   <![CDATA[
var description = new DateArc;
function DateArc() {
 this.get_SFrom = get_SFrom;
 this.put_SFrom = put_SFrom;
 this.get_SArch = get_SArch;
 this.put_SArch = put_SArch;
 this.get_SPref = get_SPref;
 this.put_SPref = put_SPref;
 this.get_SMask = get_SMask;
 this.put_SMask = put_SMask;
 this.get_SErrMess = get_SErrMess;
 this.FilesToArchiv = FilesToArchiv;
}

var SFrom;
var SArch;
var SPref;
var SMask;
var SErrMess;

function get_SFrom() {
 return SFrom;
}

function put_SFrom(newValue) {
 SFrom = newValue;
}

function get_SArch() {
 return SArch;
}

function put_SArch(newValue) {
 SArch = newValue;
}

function get_SPref() {
 return SPref;
}
```

```
function put_SPref(newValue) {
 SPref = newValue;
}

function get_SMask() {
 return SMask;
}

function put_SMask(newValue) {
 SMask = newValue;
}

function get_SErrMess(){
 return SErrMess;
}

function FilesToArchiv() {
 return "Temporary Value";
}
  ]]>
 </script>
</component>
```

Как мы видим из листинга 10.3, при использовании в компоненте-сценарии языка JScript в результате работы мастера внутрь контейнера `<script>` помещаются:

- глобальные переменные, которые соответствуют объявленным в элементах `<property>` свойствам;
- заготовки функций с префиксами `get_` и `put_`, которые осуществляют чтение и запись свойств объекта;
- заготовки функций, которые соответствуют объявленным в элементах `<method>` методам.

Кроме этого, создается экземпляр внутреннего объекта, содержащего те же свойства и методы, что были описаны внутри элемента `<public>` (переменная `description`). Имя этого внутреннего объекта совпадает с именем класса описываемого COM-объекта (в нашем случае это "`DateArc`").

Замечание

Создаваемый мастером внутренний объект нужен только в

иллюстративных целях, т.к. здесь наглядно видно, какие именно свойства и методы будет предоставлять клиентам автоматизации компонент-сценарий. Если убрать из контейнера `<script>` описание внутреннего объекта, это никак не скажется на функционировании объекта-сценария.

Доработка объекта-сценария DateArc.wsc (JScript)

Для получения нужного нам COM-объекта из сформированного с помощью Windows Script Component Wizard файла DateArc.wsc нужно выполнить несколько шагов.

Во-первых, для того, чтобы использовать внутри описания COM-объекта символы кириллицы, необходимо добавить в директиву `<?xml?>` аргумент `encoding="windows-1251"` (без этого в сценарии не удастся даже написать по-русски комментарии):

<?xml version="1.0" encoding="windows-1251"?>

Во-вторых, в контейнер `<registration>` мы вставим элемент `<script>` с двумя функциями `Register()` и `Unregister()`, которые будут выводить на экран диалоговые окна с соответствующей информацией при регистрации компонента-сценария и его отключении соответственно:

```
<script language="JScript">
<![CDATA[
var WshShell;
//Инициализируем константы для диалоговых окон
var vbInformation=64;
function Register() {
 //Создаем объект WshShell
 WshShell = new ActiveXObject("WScript.Shell");
 WshShell.Popup("Компонент зарегистрирован в системе",0,
  "Компонент для архивирования файлов",vbInformation);
}
function Unregister()   {
 //Создаем объект WshShell
 WshShell = new ActiveXObject("WScript.Shell");
 WshShell.Popup("Компонент удален из системы", 0,
  "Компонент для архивирования файлов",vbInformation);
}
]]>
</script>
```

Затем нам понадобятся две дополнительные функции. Первая из них `NowIs()` формирует строку формата ММДД, где ММ — текущий месяц, ДД — текущий день:

```
    //Вспомогательная функция для символьного представления даты
    function NowIs() {
     var d, s="", s1="";
     //Создаем объект Date (текущая дата)
     d=new Date();
     //Выделяем номер месяца
     s+=(d.getMonth()+1);
      //Если месяц представляется одним символом, добавляем слева
"0"
     if (s.length==1) s="0"+s;
     //Выделяем в дате день
     s1+=d.getDate();
     //Если день представляется одним символом, добавляем слева "0"
     if (s1.length==1) s1="0"+s1;
     s+=s1;
     //Возвращаем сформированную строку
     return s;
    }
```

Вторая функция `CheckPath()` будет проверять наличие исходного каталога и каталога для хранения архивных файлов. Для этого используется метод `FolderExists()` объекта `FileSystemObject`. Заметим, что сам объект `FileSystemObject` нужно создавать не путем вызова метода `CreateObject` объекта `WScript`, а с помощью конструкции `new ActiveXObject()`:

```
    FSO=new ActiveXObject("Scripting.FileSystemObject");
```

т.к. в отличие от обычного сценария WSH в компоненте-сценарии нет встроенного объекта `WScript`. Если хотя бы один из каталогов не существует, функция `CheckPath()` запишет соответствующее сообщение в свойство `SErrMess` и вернет значение `false`, в противном случае функция `CheckPath()` возвращает значение `true`.

```
    //Проверка доступности каталогов
    function CheckPath() {
     var FSO;
     //Создаем объект FileSystemObject
     FSO=new ActiveXObject("Scripting.FileSystemObject");
     //Проверяем доступность исходного каталога
       if  (!FSO.FolderExists(SFrom))  {  //Исходный  каталог  не
существует
      //Формируем строку с информацией об ошибке
      SErrMess="Не найден исходный каталог "+SFrom;
      return false;
     }
     //Проверяем доступность каталога для архивирования
     if (!FSO.FolderExists(SArch)) {
      //Каталог для архивирования не существует
      //Формируем строку с информацией об ошибке
```

```
  SErrMess="Не найден каталог для хранения архивов "+SArch;
  return false;
 }
 //Если оба каталога существуют, возвращаем true
 return true;
}
```

Основной функцией, осуществляющей архивирование файлов, является `FilesToArchiv()`. В самом начале этой функции с помощью вызова `CheckPath()` проверяется наличие рабочих каталогов. Если хотя бы один из каталогов не существует, то выполнение `FilesToArchiv()` прерывается и возвращается значение `false`:

```
if (!CheckPath()) return false;
```

Затем создаются экземпляры объектов `FileSystemObject` и `WshShell`:

```
//Создаем объект FileSystemObject
FSO = new ActiveXObject("Scripting. FileSystemObject");
//Создаем объект WshShell
WshShell = new ActiveXObject("WScript.Shell");
```

Имя файла-архива формируется следующим образом:

```
//Формируем имя файла-архива
SFName=SPref+NowIs()+".arj";
```

Архиватор arj.exe мы будем запускать в отдельном командном окне, которое закроется после выполнения этой программы. Для этого мы сформируем командную строку `SComLine`, с помощью которой запускается вторичная копия командного процессора (путь к командному процессору хранится в переменной среды `%COMSPEC%`, вторичная копия вызывается с помощью ключа `/c`):

```
SComLine="%COMSPEC% /c ";
```

Синтаксис запуска arj.exe для создания архивного файла имеет следующий вид:

```
arj.exe a Archiv Files
```

где*`Archiv`* — путь к файлу-архиву,*`Files`* — имена добавляемых в архив файлов (здесь можно использовать шаблонные символы ? и *). Поэтому мы добавим нужные компоненты к командной строке `SComLine`:

```
SComLine+=" arj.exe a "+FSO.BuildPath(SArch, SFName) + " ";
SComLine+= FSO.BuildPath(SFrom, SMask);
```

Команды, записанные в `SComLine`, запускаются с помощью метода `Run` объекта `WshShell`; код возврата сохраняется в переменной `RetCode`:

```
RetCode = WshShell.Run(SComLine, 1, true);
```

Равенство нулю переменной `RetCode` означает, что архивирование выполнено без ошибок — в этом случае функция `FilesToArchiv()` возвращает `true`. Если же при выполнении arj.exe возникла ошибка (переменная `RetCode` не равна нулю), то ее код вносится в сообщение об

ошибке (свойство SErrMess):

```
//Анализируем код возврата для arj.exe
if (0==RetCode)
 //Выполнение arj.exe завершилось без ошибок
 return true;
else {
 //Формируем строку с информацией об ошибке
 SErrMess="Ошибка ARJ.EXE! Код "+RetCode;
 return false;
}
```

Полностью содержимое файла DateArc.wsc приведено в листинге 10.4.

Листинг 10.4. COM-объект DateArc.WSC (JScript)

```
<?xml version="1.0" encoding="windows-1251"?>
<component>
 <registration description="DateArc" progid="DateArc.WSC"
  version="1.00"
  classid="{424ac2bc-5732-4dea-be17-0211af99cd79}">
 <script language="JScript">
  <![CDATA[
var WshShell;
//Инициализируем константы для диалоговых окон
var vbInformation=64;
function Register() {
 //Создаем объект WshShell
 WshShell = new ActiveXObject("WScript.Shell");
 WshShell.Popup("Компонент зарегистрирован в системе",0,
  "Компонент для архивирования файлов",vbInformation);
}
function Unregister() {
 //Создаем объект WshShell
 WshShell = new ActiveXObject("WScript.Shell");
 WshShell.Popup("Компонент удален из системы",0,
  "Компонент для архивирования файлов",vbInformation);
}
   ]]>
  </script>
 </registration>
 <public>
  <property name="SFrom">
   <get/>
   <put/>
  </property>
```

```
  <property name="SArch">
   <get/>
   <put/>
  </property>
  <property name="SPref">
   <get/>
   <put/>
  </property>
  <property name="SMask">
   <get/>
   <put/>
  </property>
  <property name="SErrMess">
   <get/>
  </property>
  <method name="FilesToArchiv">
  </method>
 </public>
 <script language="JScript">
  <![CDATA[
var description = new DateArc;
//Конструктор объекта DateArc
function DateArc()   {
 //Объявляем свойства объекта DateArc
 this.get_SFrom = get_SFrom;
 this.put_SFrom = put_SFrom;
 this.get_SArch = get_SArch;
 this.put_SArch = put_SArch;
 this.get_SPref = get_SPref;
 this.put_SPref = put_SPref;
 this.get_SMask = get_SMask;
 this.put_SMask = put_SMask;
 this.get_SErrMess = get_SErrMess;
 //Объявляем метод FilesToArchiv
 this.FilesToArchiv = FilesToArchiv;
}
var SFrom,    //Исходный каталог для архивирования
    SArch,    //Каталог, в котором будет создаваться архив
    SPref,    //Префикс файла
    SMask,    //Маска, по которой отбираются файлы для
              //архивирования
    SErrMess; //Текст сообщения об ошибке
//Чтение и запись свойства SFrom
function get_SFrom() {
 return SFrom;
}
function put_SFrom(newValue) {
```

```
 SFrom = newValue;
}
//Чтение и запись свойства SArch
function get_SArch() {
 return SArch;
}
function put_SArch(newValue) {
 SArch = newValue;
}
//Чтение и запись свойства SPref
function get_SPref() {
 return SPref;
}
function put_SPref(newValue) {
 SPref = newValue;
}
//Чтение и запись свойства SMask
function get_SMask() {
 return SMask;
}
function put_SMask(newValue) {
 SMask = newValue;
}
//Чтение свойства SErrMess
function get_SErrMess() {
 return SErrMess;
}
//Вспомогательная функция для символьного представления даты
function NowIs() {
 var d, s="", s1="";
 //Создаем объект Date (текущая дата)
 d=new Date();
 //Выделяем номер месяца
 s+=(d.getMonth()+1);
  //Если месяц представляется одним символом, добавляем слева
"0"
 if (s.length==1) s="0"+s;
 //Выделяем в дате день
 s1+=d.getDate();
 //Если день представляется одним символом, добавляем слева "0"
 if (s1.length==1) s1="0"+s1;
 s+=s1;
 //Возвращаем сформированную строку
 return s;
}
//Проверка доступности каталогов
function CheckPath() {
```

```
      var FSO;
      //Создаем объект FileSystemObject
      FSO=new ActiveXObject("Scripting.FileSystemObject");
      //Проверяем доступность исходного каталога
       if  (!FSO.FolderExists(SFrom))  {  //Исходный  каталог  не
существует
       //Формируем строку с информацией об ошибке
       SErrMess="Не найден исходный каталог "+SFrom;
       return false;
      }
      //Проверяем доступность каталога для архивирования
      if (!FSO.FolderExists(SArch)) {
       //Каталог для архивирования не существует
       //Формируем строку с информацией об ошибке
       SErrMess="Не найден каталог для хранения архивов "+SArch;
       return false;
      }
      //Если оба каталога существуют, возвращаем true
      return true;
     }
     //Архивирование файлов из исходного каталога
     function FilesToArchiv()  {
      var WshShell,SComLine,RetCode,SFName,FSO;
       //Если  хотя  бы  один  из  каталогов  не  существует,  возвращаем
false
      if (!CheckPath()) return false;
      //Создаем объект FileSystemObject
      FSO=new ActiveXObject("Scripting.FileSystemObject");
      //Создаем объект WshShell
      WshShell = new ActiveXObject("WScript.Shell");
      //Формируем имя файла-архива
      SFName=SPref+NowIs()+".arj";
      //Формируем командную строку для запуска архиватора arj.exe
      SComLine="%COMSPEC% /c arj.exe a ";
      SComLine+=FSO.BuildPath(SArch,SFName)+" ";
      SComLine+=FSO.BuildPath(SFrom,SMask);
      //Запускаем архиватор arj.exe
      RetCode = WshShell.Run(SComLine, 1, true);
      //Анализируем код возврата для arj.exe
      if (0==RetCode)
       //Выполнение arj.exe завершилось без ошибок
       return true;
      else {
       //Формируем строку с информацией об ошибке
       SErrMess="Ошибка ARJ.EXE! Код "+RetCode;
       return false;
      }
```

```
}
  ]]>
 </script>
</component>
```

Регистрация файла DateArc.wsc в качестве COM-объекта

После написания текста объекта-сценария нужно внести информацию о нем в системный реестр, т.е, зарегистрировать объект. Это можно сделать несколькими способами, самый простой из которых заключается в следующем. Нужно выделить в Проводнике Windows необходимый WSC-файл с компонентом-сценарием (в нашем случае это файл DateArc.wsc в каталоге C:\WSC), щелкнуть правой кнопкой мыши и в появившемся контекстном меню выбрать пункт **Подключить** (Register) (рис. 10.7).

Рис. 10.7. Контекстное меню, сопоставленное расширению WSC

После этого необходимая информация запишется в реестр и выполнится функция `Register()`, которая описана в файле DateArc.wsc внутри элемента `<registration>`, в результате чего на экран будет выведено диалоговое окно, показанное на рис. 10.8.

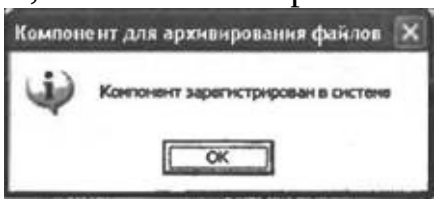


Рис. 10.8. Информация о регистрации компонента-сценария DateArc.WSC

В системном реестре данные о регистрируемом объекте `DateArc.WSC` заносятся в две ветви: `HKEY_LOCAL_MACHINE` и `HKEY_CLASSES_ROOT`.

В разделе HKEY_LOCAL_MACHINE\SOFTWARE\Classes создается новый подраздел DateArc.WSC со значением по умолчанию "DateArc". В подразделе DateArc.WSC в параметр CLSID записывается глобальный код объекта DateArc.WSC — "{424AC2BC-5732-4DEA-BE17-0211AF99CD79}" (рис. 10.9).

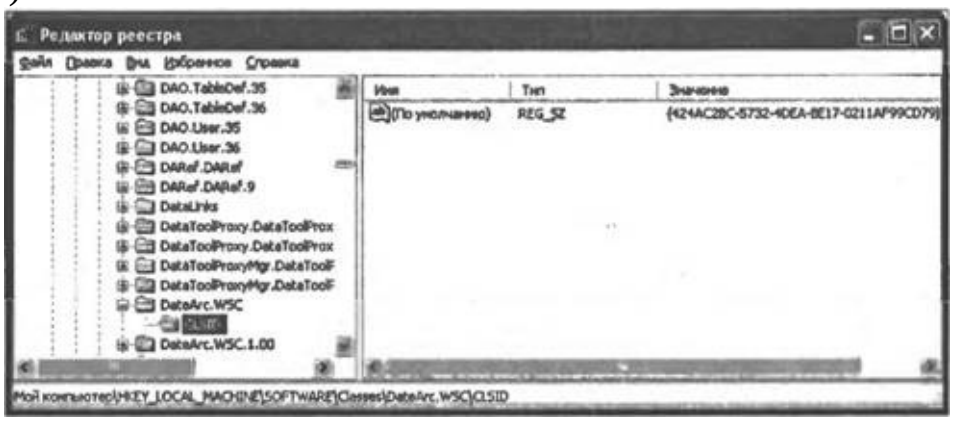


Рис. 10.9. Информация о зарегистрированном объекте DateArc.WSC в ветви HKEY_LOCAL_MACHINE

В разделе HKEY_CLASSES_ROOT\CLSID создается новый подраздел, название которого совпадает с глобальным кодом объекта DateArc.WSC — "{424AC2BC-5732-4DEA-BE17-0211AF99CD79}". Значением по умолчанию для нового подраздела является "DateArc" (рис. 10.10).

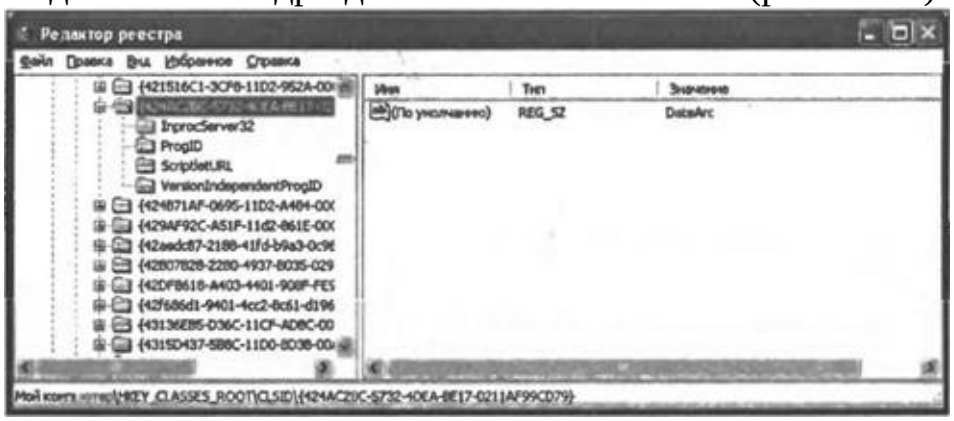


Рис. 10.10. Информация о зарегистрированном объекте DateArc.WSC в ветви HKEY_CLASSES_ROOT

В новом разделе создаются, в свою очередь, еще несколько подразделов, значения которых очень важны для функционирования компонента-сценария в качестве COM-объекта *(см. разд. "Технология Windows Script Components").* Эти разделы для нашего примера описаны в табл. 10.2.

Таблица 10.2. *Подразделы, создаваемые в разделе HKCR\CLSID \ {424AC2BC-5732-4DEA-BE17-0211AF99CD79} при регистрации объекта* `DateArc.WSC`

Название	Значение по умолчанию	Описание
`InprocServer32`	"F:\WINDOWS\System32\scrobj.dll"	Полный пут оболочке компонентов сценариев scrobj.dll
`ProgID`	"DateArc.WSC.1.00"	Программны идентифика объекта, включающи номер верси
`ScriptletURL`	"file://C:\WSC\DateArc.wsc"	Полный пут WSC-файлу
`VersionIndependentProgID`	"DateArc.WSC"	Программны идентифика объекта без номера верс

Отключается компонент-сценарий так же просто, как и регистрируется. Снова нужно выделить в Проводнике Windows WSC-файл, щелкнуть правой кнопкой мыши и в появившемся контекстном меню выбрать пункт **Отключить** (Unregister). При этом из системного реестра записи об этом объекте будут удалены, после чего выполнится функция `Unregister()` (рис. 10.11).

Рис. 10.11. Информация об отключении компонента-сценария DateArc.WSC

Реализация объекта DateArc.wsc на VBScript

Различие между компонентами-сценариями, написанными на языках JScript и VBScript, проявляется только в секции <script> WSC-файлов. Во-первых, естественным образом меняется синтаксис описанных внутри контейнера <script> функций. Во-вторых, в WSC-файле, написанном с помощью VBScript, отсутствует описание внутреннего объекта, который генерируется программой Windows Script Component Wizard и имеет поля и методы, совпадающие с объявленными внутри элемента <public> (см. листинг 10.3).

Объяснение этому очень простое — в VBScript нельзя создавать свои внутренние объекты.

Полностью содержимое файла DateArcVB.wsc, который реализует COM-объект DateArc.WSC с помощью VBScript, приведено в листинге 10.5.

Листинг 10.5. COM-объект DateArc.WSC (VBScript)

```
<?xml version="1.0" encoding="windows-1251"?>
<component>
 <registration description="DateArc" progid="DateArc.WSC"
           version="1.00"     classid="{424ac2bc-5732-4dea-be17-
0211af99cd79}">
  <script language="VBScript">
   <![CDATA[
Dim WshShell
Function Register()
 'Создаем объект WshShell
 Set WshShell = CreateObject("WScript.Shell")
 WshShell.Popup "Компонент зарегистрирован в системе",0,_
  "Компонент для архивирования файлов",vbInformation
End Function
Function Unregister()
 'Создаем объект WshShell
 Set WshShell = CreateObject("WScript.Shell")
 WshShell.Popup "Компонент удален из системы",0,_
  "Компонент для архивирования файлов",vbInformation
End Function
  ]]>
  </script>
 </registration>
 <public>
  <property name="SFrom">
   <get/>
   <put/>
  </property>
```

```
  <property name="SArch">
   <get/>
   <put/>
  </property>
  <property name="SPref">
   <get/>
   <put/>
  </property>
  <property name="SMask">
   <get/>
   <put/>
  </property>
  <property name="SErrMess">
   <get/>
  </property>
  <method name="FilesToArchiv">
  </method>
 </public>
 <script language="VBScript">
  <![CDATA[
Dim SFrom    'Исходный каталог для архивирования
Dim SArch    'Каталог, в котором будет создаваться архив
Dim SPref    'Префикс файла
Dim SMask    'Маска, по которой отбираются файлы для
             'архивирования
Dim SErrMess 'Текст сообщения об ошибке
'Чтение и запись свойства SFrom
Function get_SFrom()
 get_SFrom = SFrom
End Function
Function put_SFrom(newValue)
 SFrom = newValue
End Function
'Чтение и запись свойства SArch
Function get_SArch()
 get_SArch = SArch
End Function
Function put_SArch(newValue)
 SArch = newValue
End Function
'Чтение и запись свойства SPref
Function get_SPref()
 get_SPref = SPref
End Function
Function put_SPref(newValue)
 SPref = newValue
End Function
```

```
'Чтение и запись свойства SMask
Function get_SMask()
 get_SMask = SMask
End Function
Function put_SMask(newValue)
 SMask = newValue
End Function
'Чтение свойства SErrMess
Function get_SErrMess()
 get_SErrMess = SErrMess
End Function
'Вспомогательная функция для символьного представления даты
Function NowIs()
 Dim d,s,s1
 s=""
 s1=""
 'Определяем текущую дату
 d=Date()
 'Выделяем номер месяца
 s=s & Month(d)
 'Если месяц представляется одним символом, добавляем слева "0"
 If Len(s)=1 Then
  s="0" & s
 End If
 'Выделяем в дате день
 s1=s1 & Day(d)
 'Если день представляется одним символом, добавляем слева "0"
 If Len(s1)=1 Then
  s1="0" & s1
 End If
 s=s & s1
 'Возвращаем сформированную строку
 NowIs=s
End Function

'Проверка доступности каталогов
Function CheckPath()
 Dim FSO
 'Создаем объект FileSystemObject
 Set FSO=CreateObject("Scripting.FileSystemObject")
 'Проверяем доступность исходного каталога
 If Not FSO.FolderExists(SFrom) Then
  'Исходный каталог не существует
  'Формируем строку с информацией об ошибке
  SErrMess="Не найден исходный каталог " & SFrom
  CheckPath=false
 End If
```

```
     'Проверяем доступность каталога для архивирования
     If Not FSO.FolderExists(SArch) Then
      'Каталог для архивирования не существует
      'Формируем строку с информацией об ошибке
      SErrMess="Не найден каталог для хранения архивов " & SArch
      CheckPath=false
     End If
     'Если оба каталога существуют, возвращаем true
     CheckPath=true
    End Function
    'Архивирование файлов из исходного каталога
    Function FilesToArchiv()
     Dim WshShell,SComLine,RetCode,SFName,FSO
      'Если хотя бы один из каталогов не существует, возвращаем
false
     If Not CheckPath() Then
      FilesToArchiv=false
     End If
     'Создаем объект FileSystemObject
     Set FSO=CreateObject("Scripting.FileSystemObject")
     'Создаем объект WshShell
     Set WshShell=CreateObject("WScript.Shell")
     'Формируем имя файла-архива
     SFName=SPref & NowIs() & ".arj"
     'Формируем командную строку для запуска архиватора arj.exe
     SComLine="%COMSPEC% /c arj.exe a "
     SComLine=SComLine & FSO.BuildPath(SArch,SFName)+" "
     SComLine=SComLine & FSO.BuildPath(SFrom,SMask)
     'Запускаем архиватор arj.exe
     RetCode = WshShell.Run(SComLine, 1, true)
     'Анализируем код возврата для arj.exe
     If 0=RetCode Then
      'Выполнение arj.exe завершилось без ошибок
      FilesToArchiv=true
     Else
      'Формируем строку с информацией об ошибке
      SErrMess="Ошибка ARJ.EXE! Код " & RetCode
      FilesToArchiv=false
     End If
    End Function
      ]]>
     </script>
    </component>
```

Глава 11
Применение сценариев WSH для администрирования Windows XP

Одним из основных назначений сценариев WSH является, в конечном счете, автоматизация работы администраторов компьютерных систем, построенных на базе Windows. В данной главе мы рассмотрим примеры сценариев, которые могут быть полезны администраторам в их повседневной работе, например, при создании сценариев регистрации для пользователей.

Особое внимание мы уделим вопросам применения в сценариях WSH таких мощных современных технологий Microsoft, как ADSI — Active Directory Service Interface и WMI — Windows Management Instrumentation, которые позволяют автоматизировать процесс администрирования как отдельной рабочей станции, так и крупной корпоративной информационной системы в целом. Отметим, что в данной книге не ставится задача более или менее полного раскрытия этих технологий, а лишь кратко описываются их основные возможности и приводятся примеры сценариев для их реализации.

Использование службы каталогов Active Directory Service Interface (ADSI)

Обсудим сначала термины "каталог" и "служба каталога", которые будут использоваться в этом разделе. Под *каталогом* в общем смысле этого слова подразумевается источник информации, в котором хранятся данные о некоторых объектах. Например, в телефонном каталоге хранятся сведения об абонентах телефонной сети, в библиотечном каталоге — данные о книгах, в каталоге файловой системы — информация о находящихся в нем файлах.

Что касается компьютерных сетей (локальных или глобальных), здесь также уместно говорить о каталогах, содержащих объекты разных типов: зарегистрированные пользователи, доступные сетевые принтеры и очереди печати и т.д. Для пользователей сети важно уметь находить и использовать такие объекты (а их в крупной сети может быть огромное количество), администраторы же сети должны поддерживать эти объекты в работоспособном состоянии. Под *службой каталога* (directory service) понимается та часть распределенной компьютерной системы (компьютерной сети), которая предоставляет средства для поиска и использования имеющихся сетевых ресурсов. Другими словами, служба каталога — это единое образование, объединяющее данные об объектах сети и совокупность служб, осуществляющих манипуляцию этими данными.

В гетерогенной (неоднородной) компьютерной сети могут одновременно функционировать несколько различных служб каталогов, например, NetWare Bindery для Novell Netware 3.x, NDS для Novell NetWare 4.x/5.x, Windows Directory Service для Windows NT 4.0 или Active Directory для Windows 2000. Естественно, для прямого доступа к разным службам каталогов приходится использовать разные инструментальные средства, что усложняет процесс администрирования сети в целом. Для решения этой проблемы можно применить технологию *ADSI — Active Directory Service Interface* фирмы Microsoft, которая предоставляет набор объектов ActiveX, обеспечивающих единообразный, не зависящий от конкретного сетевого протокола, доступ к функциям различных каталогов.

Замечание

Объекты ADSI включены в операционные системы Windows XP/2000, а также могут быть установлены в более ранних версиях, для чего их нужно скачать с сервера Microsoft (http://www.microsoft.com/NTWorkstation/downloads/Other/ADSI25.a

Для того чтобы находить объекты в каталоге по их именам, необходимо определить для этого каталога *пространство имен* (namespace). Скажем, файлы на жестком диске находятся в пространстве имен файловой системы. Уникальное имя файла определяется расположением этого файла в пространстве имен, например:

```
C:\Windows\Command\command.com
```

Пространство имен службы каталогов также предназначено для нахождения объекта по его уникальному имени, которое обычно определяется расположением этого объекта в каталоге, где он ищется. Разные службы каталогов используют различные виды имен для объектов, которые они содержат. ADSI определяет соглашение для имен, с помощью которых можно однозначно идентифицировать любой объект в гетерогенной сетевой среде. Такие имена называются *строками связывания* (binding string) или *строками ADsPath* и состоят из двух частей. Первая часть имени определяет, к какой именно службе каталогов (или, другими словами, к какому именно *провайдеру ADSI*) мы обращаемся, например:

□ "LDAP://" — для службы каталогов, созданной на основе протокола LDAP (Lightweight Directory Access Protocol), в том числе для Active Directory в Windows 2000;

□ "WinNT://" — для службы каталогов в сети Windows NT 4.0 или на локальной рабочей станции Windows XP/2000;

□ "NDS://" — для службы каталогов NetWare NDS (Novell Directory Service);

□ "NWCOMPAT://" — для службы каталогов NetWare Bindery.

Вторая часть строки ADsPath определяет расположение объекта в конкретном каталоге. Приведем несколько примеров полных строк ADsPath:

```
"LDAP://ldapsrv1/CN=Kazakov,DC=DEV,DO=MSFT, DC-COM"
"WinNT://Domain1/Server1,Computer"
"WinNT://Domain1/Kazakov"
"NDS://TreeNW/0=SB/CN=Kazakov"
"NWCOMPAT://NWServer/MyNw3xPrinter"
```

В этом разделе мы подробно рассмотрим несколько простых сценариев, использующих объекты ADSI для автоматизации некоторых распространенных задач администрирования на отдельной рабочей

станции с операционной системой Windows XP; поняв принцип их работы, вы без труда сможете написать аналогичные сценарии для локальной сети, которая функционирует под управлением Active Directory или контроллера домена с Windows NT 4.0 (множество подобных примеров приведено в [18]).

Напомним, что на выделенном компьютере с Windows XP имеется база данных, содержащая информацию обо всех локальных пользователях этого компьютера. Пользователи компьютера определяются своими атрибутами (имя регистрации, полное имя, пароль и т.п.) и могут объединяться в группы. Ниже мы приведем примеры сценариев WSH, с помощью которых можно:

□ получить список имеющихся в локальной сети доменов;

□ получить список всех групп, определенных на компьютере;

□ добавить и удалить пользователя компьютера;

□ определить всех пользователей заданной группы или все группы, в которые входит определенный пользователь;

□ просмотреть атрибуты пользователя и изменить его пароль.

Для получения более полной информации по технологии ADSI следует обратиться к документации Microsoft или специальной литературе *(см. введение)*.

Связывание с нужным объектом каталога

Первым шагом для доступа к пространству имен любого каталога в целях получения информации о его объектах или изменения свойств этих объектов является *связывание* (binding) с нужным объектом ADSI.

Рассмотрим вначале, каким образом формируется строка связывания для доступа к объектам отдельной рабочей станции с операционной системой Windows XP. В общем виде эта строка имеет следующий формат:

`"WinNT:[//ComputerName[/ObjectName[, className]]]"`

Здесь параметр *`ComputerName`* задает имя компьютера; *`ObjectName`* — имя объекта (это может быть имя группы, пользователя, принтера, сервиса и т. п.); *`className`* — класс объекта. Возможными значениями параметра *`className`* являются, например, `group` (группа пользователей), `user` (пользователь), `printer` (принтер) или `service` (сервис Windows XP).

Указав в качестве строки ADsPath просто `"WinNT:"`, можно выполнить связывание с корневым объектом-контейнером, содержащим все остальные объекты службы каталога.

Приведем несколько примеров строк связывания для доступа к

различным объектам компьютера Windows XP (табл. 11.1).

Таблица 11.1. *Варианты строк связывания на компьютере Windows XP*

Строка ADsPath	Описание
`"WinNT:"`	Строка для связывания с корневым объектом пространства имен
`"WinNT://404_Popov"`	Строка для связывания с компьютером `404_Popov`
`"WinNT://404_Popov/Popov,user"`	Строка для связывания с пользователем `Popov` компьютера `404_Popov`
`"WinNT://404_Popov/BankUsers, group"`	Строка для связывания с группой `BankUsers` на компьютере `404_Popov`

Для того чтобы из сценария WSH использовать объект ADSI, соответствующий сформированной строке связывания, необходимо применить функцию `GetObject` языка JScript, которая возвращает ссылку на объект ActiveX, находящийся во внешнем каталоге. Например:

```
var NameSpaceObj = GetObject("WinNT:");
var ComputerObj = GetObject("WinNT://404_Popov");
var UserObj = GetObject("WinNT://404_Popov/Popov,user");
var GroupObj = GetObject("WinNT://404_Popov/BankUsers, group");
```

Замечание

Во всех рассмотренных ранее сценариях для создания объектов ActiveX мы пользовались методами `CreateObject` и `GetObject` объекта `WScript` или объектом `ActiveXObject` языка JScript. Для связывания же с объектом ADSI нужно использовать именно функцию `GetObject` языка JScript (или VBScript)!

Перейдем теперь к рассмотрению конкретных примеров сценариев, использующих объекты ADSI.

Список всех доступных доменов в локальной сети

В листинге 11.1 приведен JScript-сценарий ListDomains.js, в котором

создается список всех доменов, доступных в сети (рис. 11.1)

Рис. 11.1. Список всех имеющихся в сети доменов

В рассматриваемом сценарии производятся следующие действия. Сначала создается корневой объект `NameSpaceObj` класса `Namespace` для провайдера Windows NT, который содержит все остальные объекты службы каталога:

```
//Связываемся с корневым объектом Namespace
NameSpaceObj = GetObject("WinNT:");
```

Затем с помощью свойства `Filter` из коллекции `NameSpaceObj` выделяются все содержащиеся в ней объекты класса `Domain` и создается экземпляр объекта `Enumerator` (переменная `e`) для доступа к элементам коллекции `NameSpaceObj`:

```
//Устанавливаем фильтр для выделения объектов-доменов
NameSpaceObj.Filter = Array("domain");
//Создаем объект Enumerator для доступа к коллекции NameSpaceObj
E=new Enumerator(NameSpaceObj);
```

Список доменов будет храниться в переменной `List`, которая инициализируется следующим образом:

```
List="Все доступные домены в сети:\n\n";
```

В цикле `while` выполняется перебор всех элементов коллекции, которые являются объектами класса `Domain`; название домена, хранящееся в свойстве Name, добавляется (вместе с символом разрыва строки) в переменную `List`:

```
while (!E.atEnd()) {
 //Извлекаем текущий элемент коллекции (объект класса Domain)
 DomObj=E.item();
 //Формируем строку с именами доменов
 List+=DomObj.Name+"\n";
 //Переходим к следующему элементу коллекции
 E.moveNext();
}
```

Сформированная таким образом переменная `List` выводится на экран

с помощью метода `Echo()` объекта `WScript`:

```
    WScript.Echo(List);
```

Листинг 11.1. Вывод на экран списка всех доменов локальной сети

```
    /*****************************************************************
    /*                                                           Имя:
ListDomains.js                                                  */
    /*                                                          Язык:
JScript                                                        */
    /*  Описание:  Вывод  на  экран  списка  всех  доменов  локальной
сети      */
    /*****************************************************************
    //Объявляем переменные
    var
     NameSpaceObj, //Корневой объект Namespace
     DomObj,       //Экземпляр объекта Domain
     E,            //Объект Enumerator
     SList;        //Строка для вывода на экран

    //Связываемся с корневым объектом Namespace
    NameSpaceObj = GetObject("WinNT:");
    //Устанавливаем фильтр для выделения объектов-доменов
    NameSpaceObj.Filter = Array("domain");
    //Создаем   объект   Enumerator   для   доступа   к   коллекции
NameSpaceObj
    E=new Enumerator(NameSpaceObj);
    List="Все доступные домены в сети:\n\n";
    //Цикл по всем элементам коллекции доменов
    while (!E.atEnd()) {
     //Извлекаем текущий элемент коллекции (объект класса Domain)
     DomObj=E.item();
     //Формируем строку с именами доменов
     List+=DomObj.Name+"\n";
     //Переходим к следующему элементу коллекции
     E.moveNext();
    }
    //Вывод информацию на экран
    WScript.Echo(List);
    /*************                                              Конец
*******************************************/
```

Создание пользователя и группы на рабочей станции

В сценарии AddUser.js, который приведен в листинге 11.2, для создания нового пользователя на рабочей станции выполняются следующие шаги. Во-первых, производится связывание с нужным компьютером (в нашем примере это рабочая станция с именем 404_Popov), т.е. создается экземпляр `ComputerObj` объекта `Computer`:

```
ComputerObj = GetObject("WinNT://404_Popov");
```

Во-вторых, создается экземпляр `UserObj` объекта `User` для нового пользователя. Для этого используется метод `Create()` объекта `Computer`; в качестве параметров этого метода указывается имя класса "`user`" и имя создаваемого пользователя (в нашем примере это имя хранится в переменной `UserStr`):

```
UserObj=ComputerObj.Create("user", UserStr);
```

Замечание

Для создания пользователя или группы у вас в системе должны быть назначены права, которыми обладает администратор.

Для созданного пользователя в свойство `Description` мы помещаем текст описания:

```
UserObj.Description="Этот пользователь создан из сценария WSH";
```

Для сохранения информации о новом пользователе в базе данных пользователей вызывается метод `SetInfo()` объекта `User`:

```
UserObj.SetInfo();
```

Листинг 11.2. Создание нового локального пользователя на рабочей станции

```
/*****************************************************************
/*                                                          Имя:
AddUser.js                                                  */
/*                                                         Язык:
JScript                                                    */
/*      Описание:      Создание      нового      пользователя
компьютера                 */
/*****************************************************************
//Объявляем переменные
var
```

```
     ComputerObj,        //Экземпляр объекта Computer
     UserObj,            //Экземпляр объекта User
     UserStr = "XUser";  //Имя создаваемого пользователя

    //Связываемся с компьютером 404_Popov
    ComputerObj = GetObject("WinNT://404_Popov");
    //Создаем объект класса User
    UserObj=ComputerObj.Create("user",UserStr);
    //Добавляем описание созданного пользователя
    UserObj.Description="Этот пользователь создан из сценария WSH";
    //Сохраняем информацию на компьютере
    UserObj.SetInfo();
    /*************                                          Конец
*******************************************/
```

Группа на рабочей станции создается аналогичным образом (листинг 11.3).

Листинг 11.3. Создание новой локальной группы на рабочей станции

```
    /*****************************************************************
    /*                                                          Имя:
AddGroup.js                                                  */
    /*                                                         Язык:
JScript                                                     */
    /*      Описание:      Создание      новой      группы      на
компьютере                    */
    /*****************************************************************
    //Объявляем переменные
    var
     ComputerObj,         //Экземпляр объекта Computer
     GroupObj,            //Экземпляр объекта Group
     GroupStr = "XGroup"; //Имя создаваемой группы

    //Связываемся с компьютером 404_Popov
    ComputerObj = GetObject("WinNT://404_Popov");
    //Создаем объект класса Group
    GroupObj=DomainObj.Create("group", GroupStr);
    //Сохраняем информацию на компьютере
    GroupObj.SetInfo();
    /*************                                          Конец
*******************************************/
```

Вывод информации о пользователе и смена его пароля

В листинге 11.4 приведен сценарий UserInfo.js, в котором выводится на экран информация о созданном в сценарии AddUser.js пользователе XUser (рис. 11.2).

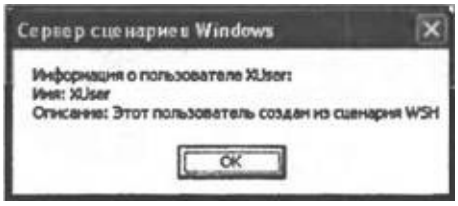


Рис. 11.2. Информация о локальном пользователе XUser

Для получения этой информации мы производим связывание с нужным пользователем, т.е. создаем экземпляр `UserObj` объекта `User` и читаем данные из полей этого объекта:

```
//Связываемся с пользователем XUser компьютера 404_Popov
UserObj=GetObject("WinNT://404_Popov/XUser, user");
//Формируем строку с информацией о пользователе
SInfo="Информация о пользователе XUser:\n";
SInfo+="Имя: "+UserObj.Name+"\n";
SInfo+="Описание: "+UserObj.Description+"\n";
//Выводим сформированную строку на экран
WScript.Echo(SInfo);
```

После этого в сценарии выдается запрос на изменение пароля пользователя XUser. Для этого мы используем метод `Popup()` объекта `WshShell`:

```
//Создаем объект WshShell
WshShell=WScript.CreateObject("WScript.Shell");
//Запрос на изменение пароля
Res=WshShell.Popup("Изменить пароль у XUser?",
0, "Администрирование пользователей", vbQuestion+vbYesNo);
```

В случае утвердительного ответа пароль изменяется с помощью метода `SetPassword()` объекта `User`, после чего все произведенные изменения сохраняются на рабочей станции с помощью метода `SetInfo()`:

```
if (Res==vbYes) {
 //Нажата кнопка "Да"
 //Устанавливаем новый пароль
 UserObj.SetPassword("NewPassword");
 //Сохраняем сделанные изменения
 UserObj.SetInfo();
 WScript.Echo("Пароль был изменен");
}
```

Листинг 11.4. Вывод информации о пользователе компьютера и смена

его пароля

```
    /*****************************************************************
    /*                                                           Имя:
UserInfo.js                                                    */
    /*                                                          Язык:
JScript                                                      */
    /*  Описание:  Вывод  информации  о  пользователе  компьютера  и
смена     */
    /*                                                            его
пароля                                                  */
    /*****************************************************************
    var
     UserObj,   //Экземпляр объекта User
     Res,       //Результат нажатия кнопки в диалоговом окне
     SPassword, //Строка с новым паролем
     SInfo;     //Строка для вывода на экран
    //Инициализируем константы для диалогового окна
    var vbYesNo=4,vbQuestion=32,vbYes=6;
    //Связываемся с пользователем XUser компьютера 404_Popov
    UserObj=GetObject("WinNT://404_Popov/XUser,user");
    //Формируем строку с информацией о пользователе
    SInfo="Информация о пользователе XUser:\n";
    SInfo+="Имя: "+UserObj.Name+"\n";
    SInfo+="Описание: "+UserObj.Description+"\n";
    //Выводим сформированную строку на экран
    WScript.Echo(SInfo);
    //Создаем объект WshShell
    WshShell=WScript.CreateObject("WScript.Shell");
    //Запрос на изменение пароля
    Res=WshShell.Popup("Изменить пароль у XUser?", 0,
     "Администрирование пользователей", vbQuestion+vbYesNo);
    if (Res==vbYes) { //Нажата кнопка Да
     //Устанавливаем новый пароль
     UserObj.SetPassword("NewPassword");
     //Сохраняем сделанные изменения
     UserObj.SetInfo();
     WScript.Echo("Пароль был изменен");
    } else WScript.Echo("Вы отказались от изменения пароля");
    /*************                                             Конец
*******************************************/
```

Удаление пользователя и группы на рабочей станции

Для удаления созданных с помощью сценариев AddUser.js и

AddGroup.js пользователя XUser и группы XGroup мы создадим сценарий DelUserAndGroup.js, который представлен в листинге 11.5.

Замечание

Для удаления пользователя или группы у вас в системе должны быть назначены права, которыми обладает администратор.

В принципе, удалить пользователя и группу так же просто, как и создать — нужно связаться с объектом `Computer`:

```
ComputerObj = GetObject("WinNT://404_Popov");
```

и вызвать метод `Delete()`, указав в качестве первого параметра класс объекта, который мы хотим удалить, и в качестве второго параметра — имя этого объекта:

```
//Удаляем пользователя
ComputerObj.Delete("user", UserStr);
```

Однако здесь могут возникнуть ошибки (например, мы не запускали предварительно сценарий `AddUser.js` и у нас на компьютере не зарегистрирован пользователь, которого мы хотим удалить). Поэтому в сценарии `DelUserAndGroup.js` предусмотрена обработка исключительных ситуаций с помощью конструкции `try…catch`:

```
IsError=false;
try {
 //Удаляем пользователя
 ComputerObj.Delete("user", UserStr);
} catch (e) { //Обрабатываем возможные ошибки
 if (e != 0) {
  //Выводим сообщение об ошибке
 IsError=true;
    Mess="Ошибка при удалении пользователя "+UserStr+"\nКод ошибки: " + e.number+"\nОписание: "+e.description;
  WshShell.Popup(Mess, 0, "Удаление пользователя", vbCritical);
 }
}
```

Как мы видим, если при вызове метода `Delete()` произойдет какая-либо ошибка, значение переменной `IsError` станет равным `true`, а на экран с помощью метода `Popup()` объекта `WshShell` выведется соответствующее сообщение (рис. 11.3).

Рис. 11.3. Сообщение, формируемое при попытке удаления несуществующего пользователя

Если же удаление прошло успешно (значение переменной IsError равно false), то на экран также выведется соответствующее диалоговое окно (рис. 11.4):

```
if (!IsError) { //Все в порядке
 Mess="Пользователь."+UserStr+" удален";
    WshShell.Popup(Mess,    0,    "Удаление    пользователя",
vbInformation);
}
```

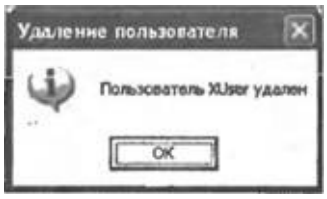


Рис. 11.4. Сообщение об удачном удалении пользователя

Листинг 11.5. Удаление пользователя и группы на рабочей станции

```
/*****************************************************************
/*                                                          Имя:
DelUserAndGroup.js                                          */
/*                                                         Язык:
JScript                                                    */
/*     Описание:     Удаление     пользователя     и     группы
компьютера              */
/*****************************************************************
//Объявляем переменные
var
 ComputerObj,          //Экземпляр объекта Computer
 UserStr = "XUser",    //Имя удаляемого пользователя
 GroupStr = "XGroup", //Имя удаляемой группы
 WshShell;             //Объект WshShell
//Инициализируем константы для диалоговых окон
var vbCritical=16,vbInformation=64;
//Связываемся с компьютером 404_Popov
```

```
    ComputerObj = GetObject("WinNT://404_Popov");
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    /*************  Удаление пользователя  ***********************/
    IsError=false;
    try {
     //Удаляем пользователя
     ComputerObj.Delete("user", UserStr);
    } catch (e) {  //Обрабатываем возможные ошибки
     if (e != 0) {
      //Выводим сообщение об ошибке
      IsError=true;
        Mess="Ошибка  при  удалении  пользователя  "+UserStr+"\nКод
ошибки: " + e.number + "\nОписание: " + e.description;
      WshShell.Popup(Mess,0,"Удаление пользователя",vbCritical);
     }
    }
    if (!IsError)  {
     //Все в порядке
     Mess="Пользователь "+UserStr+" удален";
     WshShell.Popup(Mess,0,"Удаление пользователя",vbInformation);
    }
    /*************  Удаление группы  ***********************/
    IsError=false;
    try  {
     //Удаляем группу
     ComputerObj.Delete("group", GroupStr);
    } catch (e) {  //Обрабатываем возможные ошибки
     if (e != 0) {
      //Выводим сообщение об ошибке
      IsError=true;
       Mess="Ошибка при удалении группы "+GroupStr+"\nКод ошибки: "
+ e.number+"\nОписание: " + e.description;
      WshShell.Popup(Mess,0,"Удаление группы",vbCritical);
     }
    }
    if (!IsError)  {
     //Все в порядке
     Mess="Группа "+GroupStr+" удалена";
     WshShell.Popup(Mess,0,"Удаление группы",vbInformation);
    }
    /*************                                          Конец
*****************************************/
```

Список всех групп на рабочей станции

Принцип формирования списка всех групп рабочей станции остается тем же, что и для рассмотренного выше списка всех доступных доменов локальной сети, однако первоначальное связывание нужно производить не с корневым объектом класса `Namespace`, а с нужным объектом класса `Computer`.

В приведенном в листинге 11.6 сценарии ListAllGroups.js для связывания с компьютером 404_Popov мы создаем объект-контейнер `ComputerObj`, в котором содержатся все объекты рабочей станции 404_Popov:

```
//Связываемся с компьютером 404_Popov
ComputerObj = GetObject("WinNT://404_Popov");
```

Затем в сценарии создается объект `Enumerator` для доступа к элементам коллекции `ComputerObj` и инициализируется переменная `SList`, в которой будет храниться список всех локальных групп рабочей станции:

```
//Создание объекта Enumerator для доступа к коллекции ComputerObj
E=new Enumerator(ComputerObj);
SList="На компьютере 404_Popov созданы группы:\n";
```

После этого в цикле `while` из коллекции `ComputerObj` выбираются лишь объекты класса `Group`, т.е. те объекты, у которых в поле `Class` записана строка "`Group`"; в `SList` заносятся названия групп из поля `Name`:

```
while (!E.atEnd()) {
 //Извлекаем текущий элемент коллекции
 GroupObj=E.item();
 //Выделение объектов класса Group
 if (GroupObj.Class == "Group")
  //Формируем строку с именами групп
  SList+=GroupObj.Name+"\n";
 //Переход к следующему элементу коллекции
 E.moveNext();
}
```

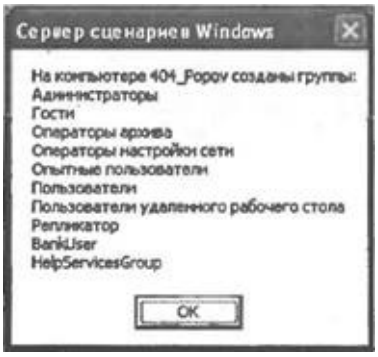


Рис. 11.5. Список всех локальных групп, определенных на рабочей станции

После окончания цикла сформированная строка выводится на экран (см. рис. 11.5):

```
//Выводим информацию на экран
WScript.Echo(SList);
```

Листинг 11.6. Вывод на экран имен всех локальных групп заданной рабочей станции

```
/*****************************************************************
/*                                                          Имя:
ListAllGroups.js                                              */
/*                                                         Язык:
JScript                                                   */
/*  Описание:  Вывод  на  экран  имен  всех  групп  заданного
компьютера    */
/*****************************************************************
//Объявляем переменные
var
 ComputerObj, //Экземпляр объекта Computer
 E,           //Объект Enumerator
 SList;       //Строка для вывода на экран

//Связываемся с компьютером 404_Popov
ComputerObj = GetObject("WinNT://404_Popov");
//Создание  объекта  Enumerator  для  доступа  к  коллекции
ComputerObj
E=new Enumerator(ComputerObj);
SList="На компьютере 404_Popov созданы группы:\n";
//Цикл по всем элементам коллекции объектов компьютера
while (!E.atEnd()) {
 //Извлекаем текущий элемент коллекции
 GroupObj=E.item();
 //Выделение объекты класса Group
 if (GroupObj.Class == "Group")
  //Формируем строку с именами групп
  SList+=GroupObj.Name+"\n";
 //Переход к следующему элементу коллекции
 E.moveNext();
}
//Выводим информацию на экран
WScript.Echo(SList);
/*************                                            Конец
*******************************************/
```

Список всех пользователей в группе

В листинге 11.7 приведен сценарий ListUsers.js, в котором формируется список всех пользователей, входящих в группу "Пользователи" на компьютере 404_Popov.

Для связывания с группой "Пользователи" рабочей станции 404_Popov создается объект `GroupObj`; коллекция пользователей этой группы формируется с помощью метода `Members()` объекта `Group`:

```
//Связываемся с группой Пользователи компьютера 404_Popov
GroupObj=GetObject("WinNT://404_Popov/Пользователи,group");
//Создание объекта Enumerator для доступа к коллекции пользователей
E=new Enumerator(GroupObj.Members());
```

После инициализации переменной `SList` мы обрабатываем в цикле `while` все элементы полученной коллекции; на каждом шаге цикла к переменной `SList` добавляется имя текущего пользователя (поле `Name` в объекте `user` — текущем элементе коллекции):

```
SList="Все пользователи группы Пользователи на компьютере 404_Popov:\n";
//Цикл по всем элементам коллекции пользователей
while (!E.atEnd()) {
 //Извлечение элемента коллекции класса User
 UserObj=E.item();
 //Формируем строку с именами пользователей
 SList+=UserObj.Name+"\n";
 //Переходим к следующему элементу коллекции
 E.moveNext();
}
```

После выхода из цикла сформированная строка `SList` выводится на экран (рис. 11.6):

```
//Выводим информацию на экран
WScript.Echo(SList);
```

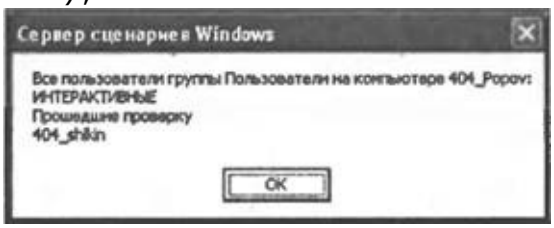


Рис. 11.6. Список всех пользователей заданной группы

Листинг 11.7. Вывод на экран имен всех пользователей заданной группы

```
    /*****************************************************************
    /*                                                           Имя:
ListUsers.js                                                   */
    /*                                                          Язык:
JScript                                                      */
    /* Описание: Вывод на экран имен всех пользователей заданной
группы */
    /*****************************************************************
    //Объявляем переменные
    var
     GroupObj, //Экземпляр объекта Group
     SList,    //Строка для вывода на экран
     E,        //Объект Enumerator
     UserObj;  //Экземпляр объекта User
    //Связываемся с группой Пользователи компьютера 404_Popov
    GroupObj=GetObject("WinNT://404_Popov/Пользователи,group");
    //Создание объекта Enumerator для доступа к коллекции
пользователей
    E=new Enumerator(GroupObj.Members());
    SList="Все пользователи группы Пользователи на компьютере
404_Popov:\n";
    //Цикл по всем элементам коллекции пользователей
    while (!E.atEnd()) {
     //Извлечение элемента коллекции класса User
     UserObj=E.item();
     //Формируем строку с именами пользователей
     SList+=UserObj.Name+"\n";
     //Переходим к следующему элементу коллекции
     E.moveNext();
    }
    //Вывод информации на экран
    WScript.Echo(SList);
    /*************                                            Конец
*******************************************/
```

Список всех групп, в которые входит пользователь

В сценарии ListGroups.js, который представлен в листинге 11.8, на экран выводятся названия всех локальных групп, в которые входит пользователь Popov на рабочей станции 404_Popov (рис. 11.7).

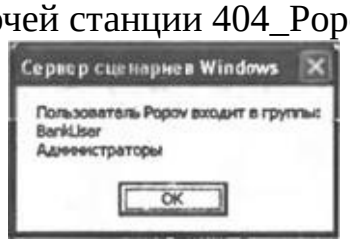


Рис. 11.7. Список всех групп, членом которых является заданный пользователь

Для создания коллекции групп, членом которых является пользователь, нужно выполнить связывание с нужным пользователем, т.е. создать экземпляр объекта `User` и воспользоваться методом `Groups()` этого объекта:

```
//Связывание с пользователем Popov компьютера
404_Popov UserObj = GetObject("WinNT://404_Popov/Popov");
//Создание объекта Enumerator для доступа к коллекции групп пользователя
E=new Enumerator(UserObj.Groups());
```

Как и в предыдущих примерах, после инициализации переменной `SList` в цикле `while` происходит перебор всех элементов полученной коллекции:

```
Slist="Пользователь Popov входит в группы: \n";
//Цикл по всем элементам коллекции групп
while (!E.atEnd()) {
 //Извлекаем элемент коллекции класса Group
 GroupObj=E.item();
 //Формируем строку с названиями групп
 SList+=GroupObj.Name+"\n";
 //Переходим к следующему элементу коллекции
 E.moveNext();
}
```

Как мы видим, название групп хранится в свойстве `Name` объекта `Group`.

Сформированная строка `SList` выводится на экран, как обычно, с помощью метода `Echo()` объекта `WScript`:

```
//Вывод информации на экран
WScript.Echo(SList);
```

Листинг 11.8. Вывод на экран названия всех групп, членом которых является заданный пользователь

```
/*****************************************************************
/*                                                          Имя: ListGroups.js                                                  */
/*                                                          Язык: JScript                                                       */
/* Описание: Вывод на экран названия всех групп, членом которых     */
/*                                          является заданный
```

```
пользователь                                */
    /*************************************************************
    //Объявляем переменные
    var
     UserObj,  //Экземпляр объекта User
     E,        //Объект Enumerator
     GroupObj, //Экземпляр объекта Group
     SList;    //Строка для вывода на экран

    //Связывание с пользователем Popov компьютера 404_Popov
    UserObj = GetObject("WinNT://404_Popov/Popov");
    //Создание объекта Enumerator для доступа к коллекции групп
пользователя
    E=new Enumerator(UserObj.Groups());
    SList="Пользователь Popov входит в группы:\n";
    //Цикл по всем элементам коллекции групп
    while (!E.atEnd()) {
     //Извлекаем элемент коллекции класса Group
     GroupObj=E.item();
     //Формируем строку с названиями групп
     SList+=GroupObj.Name+"\n";
     //Переходим к следующему элементу коллекции
     E.moveNext();
    }
    //Вывод информации на экран
    WScript.Echo(SList);
    /*************                                          Конец
*****************************************/
```

Создание сценариев включения/выключения и входа/выхода

Напомним, что в Windows XP/2000/NT для настройки среды пользователя используются *профили* (локальные и серверные), в состав которых входят все настраиваемые пользователем параметры: язык и региональные настройки, настройка мыши и звуковых сигналов, подключаемые сетевые диски и принтеры и т.д. Профили, сохраняемые на сервере, обеспечивают пользователям одну и ту же рабочую среду вне зависимости от того, с какого компьютера (под управлением Windows) зарегистрировался пользователь. Создание и поддержание профилей пользователей описываются практически в любой книге по администрированию Windows и здесь рассматриваться не будут.

Начиная с Windows NT, для настройки среды пользователей, кроме профилей, применяются *сценарии входа (сценарии регистрации)* — сценарии WSH, командные или исполняемые файлы, которые запускаются на машине пользователя каждый раз при его регистрации в сети или на локальной рабочей станции. Это позволяет администратору задавать только некоторые параметры среды пользователя, не вмешиваясь в остальные настройки; кроме этого, сценарии входа легче создавать и поддерживать, чем профили.

В Windows XP/2000 для объектов групповой политики можно дополнительно задавать сценарии следующих типов.

□ *Сценарии включения*, которые автоматически выполняются при запуске операционной системы, причем до регистрации пользователей.

□ *Сценарии входа групповой политики*, которые автоматически выполняются при регистрации пользователя, причем до запуска упомянутого выше обычного сценария регистрации для этого пользователя.

□ *Сценарии выхода*, которые автоматически выполняются после окончания сеанса работы пользователя.

□ *Сценарии выключения*, которые автоматически выполняются при завершении работы Windows.

Для простоты проверки примеров мы далее будем рассматривать сценарии включения/выключения и входа/выхода, которые хранятся на локальной рабочей станции, работающей под управлением Windows XP. Ниже будет подробно описано, в каких специальных папках нужно сохранять сценарии того или иного вида и каким образом происходит

подключение этих сценариев. Для использования сценариев включения/выключения и входа/выхода в сети со службой каталогов Active Directory нужно просто перенести сценарии в соответствующие папки на контроллере домена и воспользоваться оснасткой **Active Directory — пользователи и компьютеры** (Active Directory — users and computers) консоли управления MMC для назначения этих сценариев соответствующим объектам групповой политики.

Сценарии, выполняемые при загрузке операционной системы

Сценарии включения/выключения, как и сценарии входа/выхода групповой политики, подключаются с помощью оснастки **Групповая политика** (Group Policy) в MMC. Процесс добавления оснастки **Групповая политика** (Group Policy) для локальной рабочей станции был подробно описан в *разд. "Блокировка локальных и удаленных сценариев WSH. Пример административного шаблона" главы 4* (рис. 11.8).

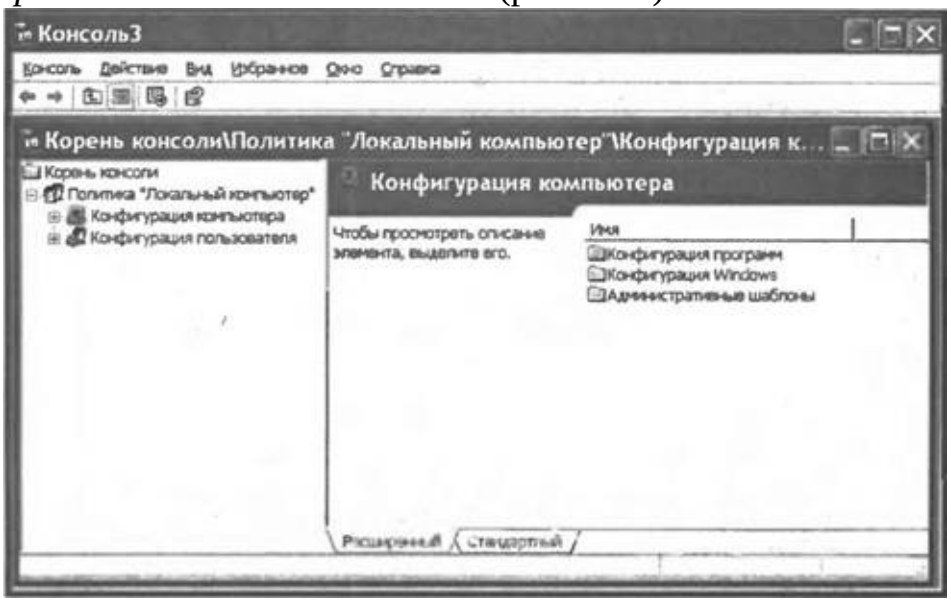


Рис. 11.8. Мастер групповой политики

Для того чтобы подключить определенный сценарий включения, нужно выделить раздел Конфигурация **компьютера|Конфигурация Windows|Сценарии (запуск/завершение)** (Computer Configuration | Windows Configuration|Scripts (Startup/Shutdown)) и выбрать свойство **Автозагрузка** (Startup), после чего будет выведено диалоговое окно **Свойства: Автозагрузка** (Properties: Startup) (рис. 11.9).

Рис. 11.9. Список установленных сценариев включения

Для добавления нового сценария нужно нажать кнопку **Добавить** (Add) и в диалоговом окне **Добавление сценария** (Adding script) указать имя нужного файла (для этого можно воспользоваться кнопкой **Обзор** (Browse)) и, в случае необходимости, параметры сценария (рис. 11.10).

Отметим, что по умолчанию сценарии включения хранятся в каталоге %SystemRoot%\System32\GroupPolicy\Machine\Scripts\Startup.

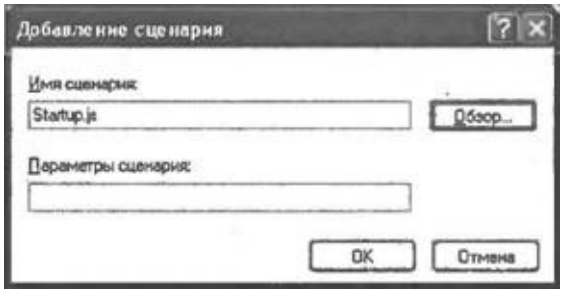


Рис. 11.10. Имя и параметры сценария включения

Сценарии, выполняемые при завершении работы операционной системы

Для подключения сценариев выключения нужно выбрать свойство **Завершение работы** (Shutdown) в разделе **Сценарии (запуск/завершение)**

(Scripts (Startup/Shutdown)), после чего будет выведено диалоговое окно **Свойства: Завершение работы** (Properties: Shutdown) (рис. 11.11).

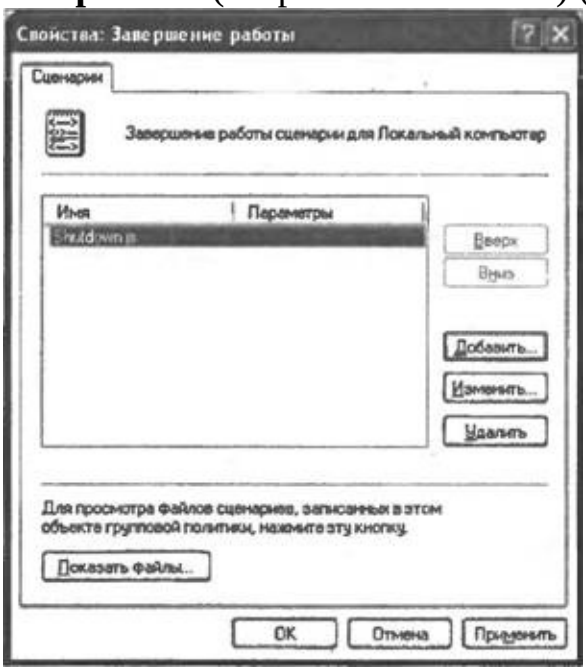


Рис. 11.11. Список установленных сценариев выключения

Как и в предыдущем случае, для добавления нового сценария нужно нажать кнопку **Добавить** (Add) и в диалоговом окне **Добавление сценария** (Adding script) указать имя нужного файла (по умолчанию сценарии выключения хранятся в каталоге %SystemRoot%\System32\GroupPolicy\Machine\Scripts\Shutdown) и параметры сценария.

Сценарии входа для всех локальных пользователей

Сценарии входа групповой политики подключаются в разделе **Конфигурация пользователя|Конфигурация Windows|Сценарии (вход/выход из системы)** (User Configuration|Windows Configuration|Scripts (Logon/Logoff)). В этом разделе нужно выбрать свойство **Вход в систему** (Logon), после чего будет выведено диалоговое окно **Свойства: Вход в систему** (Properties: Logon) (рис. 11.12).

Для добавления нового сценария входа нужно нажать кнопку **Добавить** (Add) и в диалоговом окне **Добавление сценария** (Adding script) указать имя нужного файла (по умолчанию сценарии выключения хранятся в каталоге %SystemRoot%\System32\GroupPolicy\User\Scripts\Logon) и параметры сценария.

Рис. 11.12. Список установленных сценариев входа

Сценарий выхода для всех локальных пользователей

Для подключения сценариев выхода нужно выбрать свойство **Выход из системы** (Logoff) в разделе **Сценарии (вход/выход из системы)** (Scripts (Logon/Logoff)), после чего будет выведено диалоговое окно **Свойства: Выход из системы** (Properties: Logoff) (рис. 11.13).

Для добавления нового сценария нужно нажать кнопку **Добавить** (Add) и в диалоговом окне **Добавление** сценария (Adding script) указать имя нужного файла (по умолчанию сценарии выхода хранятся в каталоге %SystemRoot%\System32\GroupPolicy\User\Scripts\Logoff) и параметры сценария.

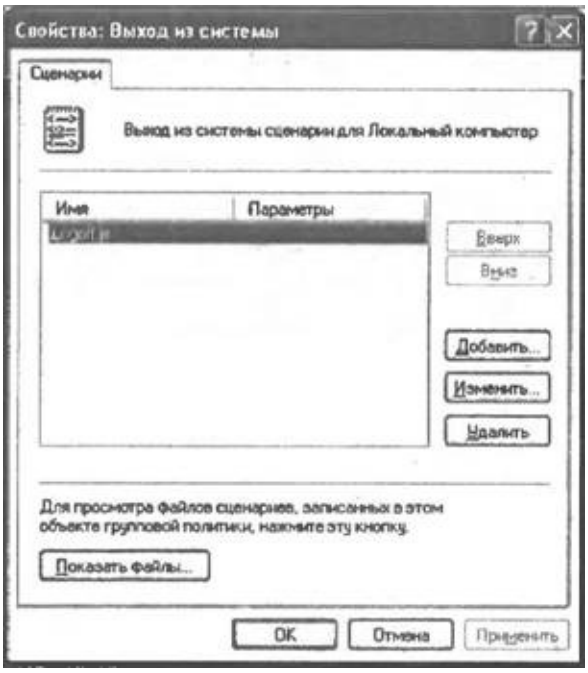


Рис. 11.13. Список установленных сценариев выхода

Сценарий входа для одного пользователя

Сценарии входа для отдельных пользователей назначаются с помощью оснастки **Локальные пользователи и группы** (Local users and groups).

Замечание

В Windows NT для этого использовался Диспетчер пользователей (User Manager for Domain).

Для добавления этой оснастки в консоли MMC выберем пункт **Добавить или удалить оснастку** (Add/Remove Snap-in) в меню **Консоль** (Console) и нажмем кнопку **Добавить** (Add). В появившемся списке всех имеющихся оснасток нужно выбрать пункт **Локальные пользователи и группы** (Local users and groups) и нажать кнопку **Добавить** (Add). После этого появится диалоговое окно, в котором нужно указать, что выбранная оснастка будет управлять локальным компьютером, и нажать кнопку **Готово** (Finish) (рис. 11.14).

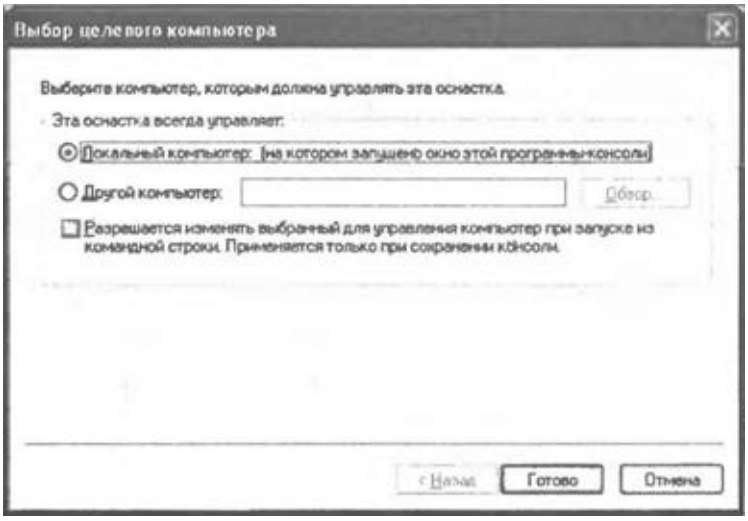


Рис. 11.14. Выбор компьютера, которым будет управлять оснастка **Локальные пользователи и группы**

Никаких других оснасток в окно консоли мы добавлять не будем, поэтому нажимаем кнопку **Закрыть** (Close) в списке оснасток и кнопку **OK** в окне добавления/удаления оснасток. После этого мы можем в окне консоли просматривать список локальных пользователей компьютера и изменять их свойства (рис. 11.15).

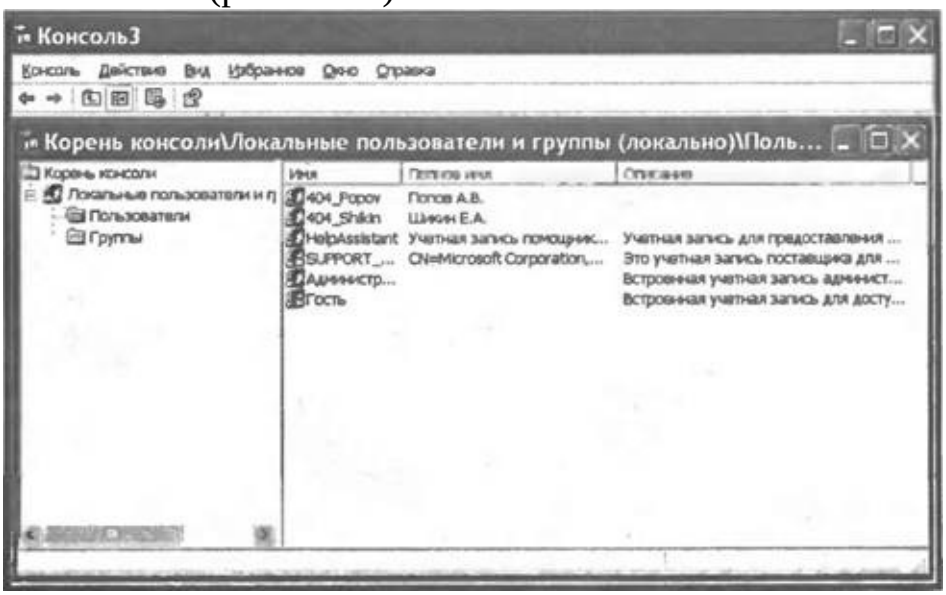


Рис. 11.15. Список пользователей локального компьютера

Для назначения пользователю сценария входа нужно выбрать этого пользователя (например, Popov) в списке и перейти на вкладку **Профиль** (Profile) в диалоговом окне со свойствами пользователя. Имя сценария входа вводится в поле **Сценарий входа** (Logon Script) этого окна (рис.

11.16).

Рис. 11.16. Настройки профиля пользователя

Путь к сценарию входа нужно указывать относительно каталога %SystemRoot%\System32\Repl\Import\Scripts. Если, скажем, сценарий scr99.bat для пользователя Popov находится в каталоге с полным именем F:\Windows\System32\Repl\Import\Scripts\Script99, то в качестве пути к сценарию входа нужно указать \Script99\scr99.bat.

Примеры сценариев входа/выхода

Ниже рассмотрены несколько сценариев (два из которых являются обычными командными файлами), которые можно использовать в качестве сценариев входа или выхода.

Подключение сетевых дисков и синхронизация времени при регистрации пользователей

Часто сценарии входа используются для подключения дисков и портов принтера к сетевым ресурсам, а также для синхронизации системного времени пользовательских компьютеров с системным временем определенного сервера (это необходимо, например, для файл-серверных банковских систем, работающих в реальном времени). Конечно, для этих

целей можно написать сценарий WSH, однако в подобных случаях проще ограничиться обычным командным (пакетным) файлом. Отметим, что в пакетных файлах можно использовать различные утилиты командной строки из пакетов Windows NT/2000/XP Resource Kit, с помощью которых многие задачи можно решить наиболее быстрым и простым способом. В качестве примера упомянем лишь одну полезную команду `IFMEMBER`, которая позволяет, не прибегая к помощи ADSI, проверить принадлежность пользователя, выполняющего регистрацию, к определенной группе.

Замечание

Желающим больше узнать о возможностях пакетных файлов в Windows и командах, которые в них используются, можно порекомендовать мою предыдущую книгу [8].

Предположим, что при регистрации определенного пользователя нам нужно произвести следующие действия:

1. Синхронизировать системное время клиентской рабочей станции с системным временем на сервере Server1.

2. Подключить диск M: к сетевому ресурсу \\Server1\Letters.

3. Предоставить каталог C:\TEXT на клиентском компьютере в общее пользование с именем BOOKS.

Для этого пользователю в качестве сценария регистрации можно назначить командный файл Logon.bat, который состоит (вместе с комментариями) всего из шести строк (листинг 11.9).

Листинг 11.9. Пример командного файла-сценария входа

```
@ECHO OFF
REM Имя: Logon.bat
REM Описание: Использование командного файла в качестве сценария входа
NET TIME \\Server1 /SET
NET USE M: \\Server1\Letters /PERSISTENT:NO
NET SHARE MyTxt=C:\TEXT
```

В первой строке файла Logon.bat мы отключаем режим дублирования команд на экране:

```
@ECHO OFF
```

Синхронизация времени с сервером \\Server1 производится с помощью ключа `/SET` в команде `NET TIME`:

```
NET TIME \\Server1 /SET
```

Сетевой диск подключается командой `NET USE`:

```
NET USE M: \\Server1\Letters /PERSISTENT:NO
```

Ключ `/PERSISTENT:NO` в команде `NET USE` нужен для создания временных подключений (не сохраняющихся после завершения сеанса пользователя). Если бы подключения были постоянными (`/PERSISTENT:YES`), то при следующем входе пользователя в систему возникла бы ошибка (повторное использование уже имеющегося подключения).

Наконец, папка C:\TEXT предоставляется в общее пользование командой `NET SHARE`:

```
NET SHARE MyTxt=C:\TEXT
```

Интерактивный выбор программ автозагрузки

Как известно, в подменю **Программы** (Programs) стартового меню Windows имеется пункт **Автозагрузка** (Startup), в который можно поместить ярлыки тех программ, которые должны быть автоматически запущены при регистрации пользователя в системе. Однако в процессе загрузки своего профиля пользователь не имеет возможности запустить только какие-либо определенные программы из папки автозагрузки — можно либо запустить все программы, либо не запускать ни одной (для этого необходимо в процессе регистрации в системе удерживать нажатой клавишу <Shift>).

Мы напишем сценарий Logon.js, с помощью которого пользователь при входе сможет выбрать запускаемые программы; назначив этот сценарий в качестве сценария входа групповой политики, мы сделаем процесс автозагрузки приложений интерактивным.

Начнем мы с того, что создадим в каталоге %SystemDrive%\Documents and Settings\All Users\Главное меню, в котором хранятся ярлыки программ из стартового меню для всех пользователей, папку **Выборочная автозагрузка** и поместим туда ярлыки для нужных приложений (рис. 11.17).

После этого ярлыки из обычной папки **Автозагрузка** нужно убрать. Рассмотрим теперь алгоритм работы сценария входа Logon.js.

Вначале нужно определить путь к папке выборочной автозагрузки (переменная `PathStartup`). Для этого мы воспользуемся объектом `WshSpecialFolders`:

```
//Создаем объект WshShell
```

```
    WshShell=WScript.CreateObject("Wscript.Shell");
    //Создаем объект WshSpecialFolders
    WshFldrs=WshShell.SpecialFolders;
    //Определяем путь к папке выборочной автозагрузки
    PathStartup=WshFldrs.item("AllUsersStartMenu")+"\\Выборочная
автозагрузка\\";
```

Рис. 11.17. Новая папка **Выборочная автозагрузка**

Зная путь к нужной папке, мы формируем коллекцию всех файлов, которые находятся в ней (переменная `Files`):

```
    //Создаем объект FileSystemObject
    FSO=WScript.CreateObject("Scripting.FileSystemObject");
    //Создаем объект Folder для папки выборочной автозагрузки
    Folder=FSO.GetFolder(PathStartup);
    //Создаем коллекцию файлов каталога выборочной автозагрузки
    Files=new Enumerator(Folder.Files);
```

После этого мы выводим на экран диалоговое окно с вопросом, нужно ли запустить программы из папки автозагрузки в пакетном режиме, т.е. все сразу (рис. 11.18).

Рис. 11.18. Выбор режима автозагрузки программ (пакетный или интерактивный)

В зависимости от нажатой в диалоговом окне кнопки мы присваиваем логическое значение переменной IsRunAll, определяющей режим автозагрузки программ (если IsRunAll равно false, то для каждой программы будет выдаваться запрос на ее запуск, в противном случае все программы запускаются без предупреждения):

```
//Выводим запрос на запуск сразу всех программ
Res=WshShell.Popup("Запустить сразу все программы?", 0,
 "Выборочная автозагрузка", vbQuestion+vbYesNo);
//Определяем, нажата ли кнопка "Да"
IsRunAll=(Res==vbYes);
```

Далее в цикле while производится перебор всех файлов из коллекции Files; переменная File соответствует текущему файлу в коллекции:

```
//Цикл по всем файлам каталога выборочной автозагрузки
while (!Files.atEnd()) {
 //Создаем объект File для текущего элемента коллекции
 File=Files.item();
 //Переходим к следующему файлу в коллекции
 Files.moveNext();
}
```

Если ранее был выбран интерактивный режим запуска программ (переменная IsRunAll равна false), то мы выводим запрос на запуск текущего файла (рис. 11.19):

```
//Обнуляем переменную Res Res=0;
if (!IsRunAll) //Программы нужно запускать по одной
 //Выводим запрос на запуск одной программы
  Res=WshShell.Popup("Запустить "+File.Name+"?", 0, "Выборочная автозагрузка", vbQuestion+vbYesNo);
```

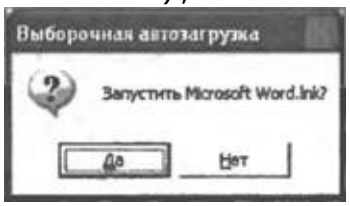


Рис. 11.19. Запрос на запуск одной программы из папки автозагрузки

Если пользователь решил запустить программу (переменная Res равна vbYes) или программы запускаются в пакетном режиме, то мы запускаем текущую программу в минимизированном окне с помощью метода Run объекта WshShell:

```
if ((IsRunAll) || (Res=vbYes))
 //Запускаем текущую программу в минимизированном окне
 WshShell.Run("\""+File.Path+" \"", vbMinimizedFocus);
```

Так как в полном имени запускаемой программы могут быть пробелы, это имя нужно заключить в двойные кавычки с помощью escape-

последовательности \".

Замечание

Другим вариантом запуска с помощью метода `Run` программ, имена которых содержат пробелы, можно использовать короткие имена папок и файлов посредством свойства `ShortPath` объекта `File`: WshShell.Run(File.ShortPath, vbMinimizedFocus);

Полностью текст сценария Logon.js приведен в листинге 11.10.

Листинг 11.10. Сценарий входа, позволяющий выбирать программы для автозагрузки

```
    //*************************************************************
    /*                                                       Имя:
Logon.js                                                     */
    /*                                                       Язык:
JScript                                                     */
    /* Описание: Сценарий входа, позволяющий выбирать программы
для    */
    /*
автозагрузки                                                */
    /*************************************************************
    //Объявляем переменные
    var
     FSO,         //Экземпляр объекта FileSystemObject
     WshShell,    //Экземпляр объекта WshShell
     WshFldrs,    //Экземпляр объекта WshSpecialFolders
     PathStartup, //Путь к папке выборочной автозагрузки
     Folder,      //Экземпляр объекта Folder для папки
                  //выборочной автозагрузки
      Files,              //Коллекция файлов в папке выборочной
автозагрузки
     File,        //Экземпляр объекта File для ярлыка в папке
                  //выборочной автозагрузки
     Res,         //Результат нажатия кнопок в диалоговых окнах
     IsRunAll;    //Логический флаг, указывающий, надо ли запустить
                  //сразу все программы из автозагрузки
    //Инициализируем константы для диалоговых окон
    var
vbYesNo=4,vbQuestion=32,vbYes=6,vbOkOnly=0,vbMinimizedFocus=2;
```

```
    //Создаем объект WshShell
    WshShell=WScript.CreateObject("Wscript.Shell");
    //Создаем объект WshSpecialFolders
    WshFldrs=WshShell.SpecialFolders;
    //Определяем путь к папке выборочной автозагрузки
    PathStartup=WshFldrs.item("AllUsersStartMenu")+"\\Выборочная
автозагрузка\\";
    //Создаем объект FileSystemObject
    FSO=WScript.CreateObject("Scripting.FileSystemObject");
    //Создаем объект Folder для папки выборочной автозагрузки
    Folder=FSO.GetFolder(PathStartup);
    //Создаем коллекцию файлов каталога выборочной автозагрузки
    Files=new Enumerator(Folder.Files);
    //Выводим запрос на запуск сразу всех программ
    Res=WshShell.Popup("Запустить сразу все программы?",0,
      "Выборочная автозагрузка",vbQuestion+vbYesNo);
    //Определяем, нажата ли кнопка Да
    IsRunAll=(Res==vbYes);
    //Цикл по всем файлам каталога выборочной автозагрузки
    while (!Files.atEnd()) {
     //Создаем объект File для текущего элемента коллекции
     File=Files.item();
     //Обнуляем переменную Res
     Res=0;
     if (!IsRunAll) //Программы нужно запускать по одной
      //Выводим запрос на запуск одной программы
      Res=WshShell.Popup("Запустить "+File.Name+"?",0,
       "Выборочная автозагрузка",vbQuestion+vbYesNo);
     if ((IsRunAll) || (Res==vbYes))
      //Запускаем текущую программу в минимизированном окне
      WshShell.Run("\""+File.Path+"\"",vbMinimizedFocus);
     //Переходим к следующему файлу в коллекции
     Files.moveNext();
    }
    /*************                                          Конец
*******************************************/
```

Резервное копирование документов пользователя при окончании сеанса работы

Для каждого пользователя Windows XP в каталоге Documents and Settings автоматически создается личная папка, имя которой совпадает с именем этого пользователя. В подкаталоге "Мои документы" (My Documents) этой папки по умолчанию сохраняются все созданные

пользователем документы. Для того чтобы всегда иметь резервную копию документов пользователей, можно написать универсальный сценарий выхода, в котором будет происходить копирование всех файлов и подкаталогов из пользовательского каталога "Мои документы" в другой каталог с именем пользователя. В нашем примере резервные копии документов будут сохраняться в каталоге D:\Backup, т.е. при выходе пользователя Popov все его документы скопируются в каталог D:\Backup\Popov, а при выходе пользователя Kazakov — в каталог D:\Backup\Kazakov.

Командный файл

Самым быстрым решением поставленной задачи является создание командного файла Logoff.bat (листинг 11.11) и назначение его в качестве сценария выхода для всех пользователей. Результат работы этого пакетного файла будет виден в командном окне (рис. 11.20).

Листинг 11.11. Командный файл-сценарий выхода, позволяющий производить : резервное копирование документов пользователя

```
    @ECHO OFF
    REM Имя: Logoff.bat
    REM Описание: BAT-файл, выполняющий резервное копирование
    REM           документов пользователя
    ECHO Окончание сеанса пользователя %UserName%.
    ECHO.
    ECHO Начинаем копирование документов в каталог
D:\Backup\%UserName%...
    XCOPY /C /D /E /I /Y "%HomeDrive%%HomePath%\Мои документы"
D:\Backup\%UserName%
    ECHO.
    ECHO Копирование документов завершено.
    PAUSE
```

Как мы видим, вся работа файла Logoff.bat заключается в вызове команды XCOPY для нужных каталогов:

```
    XCOPY /C /D /E /I /Y "%HomeDrive%%HomePath%\Мои документы"
"D:\Backup\%UserName%"
```

Рис. 11.20. Результат работы сценария выхода Logoff.bat для пользователя Popov

Здесь для XCOPY указаны несколько ключей, которые позволяют:

□ не прерывать копирование при возникновении ошибки (ключ /C);

□ копировать только те файлы, которые были изменены (ключ /D);

□ копировать все подкаталоги, включая пустые (ключ /E);

□ создавать, при необходимости, каталог, в который производится копирование (ключ /I);

□ перезаписывать файлы без подтверждения пользователя (ключ /Y).

Замечание

Подробнее о ключах команды XCOPY можно узнать из встроенной справки для этой команды. Для вывода этой справки на экран необходимо в командном окне запустить XCOPY с ключом /?; для вывода справки в текстовый файл нужно воспользоваться символом перенаправления вывода '>', например: `XCOPY /? > spr.txt`.

Пути к каталогу, где хранятся документы пользователя, и к каталогу, в который будет производиться копирование, формируются с помощью переменных среды %HomeDir%, %HomePath% и %UserName%. Описание этих и некоторых других переменных среды, которые определены в Windows, приведено в табл. 11.2.

Таблица 11.2. *Переменные среды, полезные для использования в*

сценариях входа/выхода

Переменная	Описание
%COMSPEC%	Путь к командному интерпретатору
%HOMEDIR%	Буква переопределенного диска на компьютере пользователя, которая ссылается на сетевой ресурс, содержащий личный каталог пользователя
%HOMEDRIVE%	Локальный, либо перенаправленный диск, на котором расположен личный каталог
%HOMEPATH%	Путь к личному каталогу
%HOMESHARE%	Имя каталога общего доступа, включающее личный каталог и локальный, либо переопределенный диск
%OS%	Операционная система, управляющая рабочей станцией
%PROCESSOR_ARCHITECTURE%	Архитектура процессора (например, x86) рабочей станции пользователя
%SYSTEMDRIVE%	Диск, на котором находится системный каталог Windows
%SYSTEMROOT%	Путь к системному каталогу Windows
%PROCESSOR_LEVEL%	Тип процессора рабочей станции пользователя
%TEMP%	Путь к каталогу для хранения временных файлов
%USERDOMAIN%	Домен, в котором зарегистрирован пользователь
%USERNAME%	Имя, под которым регистрировался при входе в сеть пользователь

Так как имена каталогов, присутствующих в команде XCOPY, могут содержать пробелы, эти имена взяты в кавычки.

Сценарий WSH

Для создания нужных нам резервных копий можно также написать сценарий WSH (назовем этот сценарий Logoff.js), который, конечно, будет

намного больше по объему, чем командный файл, но будет выводить сообщения в красивые графические диалоговые окна (рис. 11.21–11.23).

Сначала в сценарии Logoff.js создаются экземпляры объектов `WshShell`, `FileSystemObject` и `WshSpecialFolders`, после чего в переменную `SHomePath` заносится путь к каталогу с документами текущего пользователя (специальная папка с именем My Documents):

```
//Создаем объект WshShell
WshShell = WScript.CreateObject("WScript.Shell");
//Создаем объект FileSystemObject
FSO = WScript.CreateObject("Scripting.FileSystemObject");
//Создаем объект WshSpecialFolders
WshFldrs=WshShell.SpecialFolders;
//Определяем путь к папке выборочной автозагрузки
SHomePath=WshFldrs.item("MyDocuments");
```

Путь к каталогу, в который будет производиться копирование документов, формируется с помощью переменной среды `%UserName%`; значение такой переменной извлекается с помощью метода `ExpandEnvironmentStrings()` объекта `WshShell`:

```
//Определяем имя пользователя
SUserName=WshShell.ExpandEnvironmentStrings("%UserName%");
//Формируем полный путь к каталогу с резервными копиями документов
//пользователя
SBackupPath+=SUserName;
```

Копирование документов мы будем производить только после утвердительного ответа пользователя на соответствующий вопрос (см. рис. 11.21):

```
//Запрос на создание резервной копии
Res=WshShell.Popup("Выполнить резервное копирование документов в\n" + SBackupPath + " ?", 0, "Выход пользователя " + SUserName, vbQuestion+vbYesNo);
```

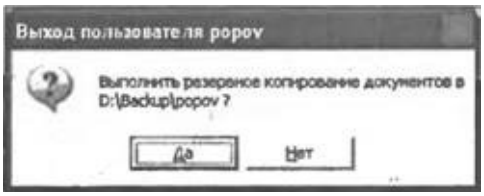


Рис. 11.21. Диалоговое окно с запросом о необходимости копирования

Если пользователь согласен, мы копируем нужный каталог с помощью метода `CopyFolder()`, причем делаем это внутри блока `try` конструкции `try…catch`.

```
IsError=false;
try {
```

```
 //Производим копирование каталога
 FSO.CopyFolder(SHomePath,SBackupPath);
}
```

В случае возникновения ошибки переменной IsError в блоке catch присваивается значение true, а на экран выводится диалоговое окно с соответствующим сообщением (см. рис. 11.22):

```
catch (e) { //Обрабатываем возможные ошибки
 if (e != 0) {
  //Выводим сообщение об ошибке
  IsError=true;
    Mess="Ошибка при копировании каталога "+SHomePath+"\nКод ошибки: " + e.number + "\nОписание: " + e.description;
   WshShell.Popup(Mess, 0, "Выход пользователя " + SUserName, vbCritical);
 }
}
```

Рис. 11.22. Диалоговое окно с сообщением о возникшей ошибке

Если же в процессе копирования ошибок не возникло (переменная `IsError` равна `false`), то пользователю также выдается сообщение об этом (см. рис. 11.23):

```
if (!IsError) {
 //Производим копирование каталога
 FSO.CopyFolder(SHomePath, SBackupPath);
 //Все в порядке
 Mess = "Копирование документов произведено";
  WshShell.Popup(Mess, 0, "Выход пользователя " + SUserName, vbInformation);
}
```

Рис. 11.23. Диалоговое окно с сообщением о возникшей ошибке

Полностью текст сценария Logoff.js приведен в листинге 11.12.

Листинг 11.12. JScript-сценарий выхода, позволяющий производить резервное копирование документов пользователя

```
    /*****************************************************************
    /*                                                          Имя:
Logoff.js                                                    */
    /*                                                         Язык:
JScript                                                    */
    /*   Описание:   Сценарий   выхода,   позволяющий   производить
резервное     */
    /*                                    копирование   документов
пользователя                    */
    /*****************************************************************
    //Объявляем переменные
    var
     WshShell,                   //Экземпляр объекта WshShell
      WshFldrs,                                //Экземпляр  объекта
WshSpecialFolders
      FSO,                                     //Экземпляр  объекта
FileSystemObject
      SUserDocPath,                    //Путь  к  папке  с  документами
пользователя
     SUserName,                  //Имя пользователя
      SBackupPath="D:\\Backup\\",  //Каталог  для  резервных  копий
документов
     Res,IsError;
    //Инициализируем константы для диалоговых окон
    var
vbYesNo=4,vbQuestion=32,vbInformation=64,vbYes=6,vbOkOnly=0,
     vbCritical=16;
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    //Создаем объект FileSystemObject
    FSO = WScript.CreateObject("Scripting.FileSystemObject");
    //Создаем объект WshSpecialFolders
    WshFldrs=WshShell.SpecialFolders;
    //Определяем путь к папке выборочной автозагрузки
    SHomePath=WshFldrs.item("MyDocuments");
    //Определяем имя пользователя
    SUserName=WshShell.ExpandEnvironmentStrings("%UserName%");
    //Формируем  полный  путь  к  каталогу  с  резервными  копиями
документов
    //пользователя
    SBackupPath+=SUserName;
    //Запрос на создание резервной копии
    Res=WshShell.Popup("Выполнить  резервное  копирование  документов
```

```
в\n"+
       SBackupPath+"  ?",  0,  "Выход  пользователя  "+SUserName,
vbQuestion+vbYesNo);
    if (Res==vbYes) { //Нажата кнопка Да
     IsError=false;
     try {
      //Производим копирование каталога
      FSO.CopyFolder(SHomePath,SBackupPath);
     } catch (e) {  //Обрабатываем возможные ошибки
      if (e != 0) {
       //Выводим сообщение об ошибке
       IsError=true;
         Mess="Ошибка  при  копировании  каталога  "+SHomePath+"\nКод
ошибки: "+
        e.number+"\nОписание: "+e.description;
                    WshShell.Popup(Mess,0,"Выход      пользователя
"+SUserName,vbCritical);
      }
     }
     if (!IsError)  {
      //Все в порядке
      Mess="Копирование документов произведено";
                  WshShell.Popup(Mess,0,"Выход        пользователя
"+SUserName,vbInformation);
     }
    }
    /*************                                                Конец
*****************************************/
```

Вызов системных функций и стандартных диалоговых окон оболочки Windows

Из сценариев WSH можно выводить на экран стандартные диалоговые окна Windows (например, **Выполнить** (Run)) и модули панели управления (например, **Установка даты и времени** (Date/Time)). Для этого используются системные функции Windows (API-функции) и объект `Shell.Application`, который позволяет получить доступ к оболочке Windows.

Конкретные примеры применения системных функций и методов объекта-оболочки Windows приведены ниже.

Вызов модулей панели управления

Напомним, что в Windows XP модули панели управления хранятся в каталоге %SystemRoot%\System32 в нескольких файлах с расширением cpl. Эти модули можно вывести на экран с помощью утилиты Control.exe, запустив ее из командной строки или из меню **Выполнить** (Run) с тремя параметрами (два из них необязательны):

`Control.exe` *`File.cpl`*`,[`*`Name`*`],[`*`Page`*`]`

Здесь *`File.cpl`* — название cpl-файла; *`Name`* — имя модуля; *`Page`* — номер страницы в диалоговом окне, которая будет выведена на передний план.

Например, команда

```
Control.exe Main.cpl, @0
```

вызовет диалоговое окно для настройки мыши (рис. 11.24).

Рис. 11.24. Модуль панели управления для настройки мыши

Если же выполнить команду

```
Control.exe Main.cpl, @1
```

то на экран будет выведено диалоговое окно для настройки клавиатуры (рис. 11.25).

Рис. 11.25. Модуль панели управления для настройки клавиатуры

Описание модулей панели управления для Windows XP приведено в табл. 11.3 (в других версиях операционной системы количество имен и страниц может быть другим).

Таблица 11.3. *Модули панели управления в Windows XP*

Модуль панели управления	Имя	Индекс	Описание
appwiz.cpl	—	0…3	Установка и удаление программ
desk.cpl	—	0…4	Свойства экрана
hdwwiz.cpl	—	—	Мастер установки оборудования
inetcpl.cpl	—	0…6	Параметры браузера Internet Explorer
intl.cpl	—	0…2	Языки и региональные стандарты
joy.cpl	—	—	Установленные игровые устройства и их свойства
main.cpl	@0, @1	0…4	Параметры мыши и клавиатуры
mmsys.cpl	—	0…4	Свойства аудиоустройств
ncpa.cpl	—	—	Сетевые подключения
nusrmgr.cpl	—	—	Учетные записи пользователей
odbccp32.cpl	—	—	Администратор источников данных ODBC
powercfg.cpl	—	—	Настройки управления электропитанием
sysdm.cpl	@0, @1	0…6	Свойства системы
telephon.cpl	—	—	Телефонные подключения
timedate.cpl	—	0…1	Установка даты и времени
access.cpl	—	0…5	Настройка специальных возможностей
AccessSetup.cpl	—	—	Установка пользователя по умолчанию

Из сценариев WSH модули панели управления можно вызывать несколькими способами, два из которых мы рассмотрим ниже.

Запуск с помощью оболочки Windows

Для доступа к стандартным диалоговым окнам Windows и модулям

панели управления нужно сначала создать экземпляр объекта-оболочки Windows:

```
//Создаем объект Shell.Application
Shell=WScript.CreateObject("Shell.Application");
```

Модули панели управления вызываются с помощью метода `ControlPanelItem()`, в качестве параметра которого указывается имя соответствующего cpl-файла, например:

```
Shell.ControlPanelItem("Appwiz.cpl");
```

Если запустить `ControlPanelItem()` без параметра, то откроется вся панель управления.

В листинге 11.13 приведен сценарий RunCPL.js, в котором происходит вызов некоторых модулей панели управления.

Листинг 11.13. Вызов модулей панели управления с помощью оболочки Windows

```
/*****************************************************************
/*                                                          Имя:
RunCPL.js                                                  */
/*                                                         Язык:
JScript                                                   */
/*    Описание:    Вызов    модулей    панели    управления    с
помощью              */
/*                                                        объекта
Shell.Application                                 */
/*****************************************************************
//Объявляем переменные
var
 WshShell, //Экземпляр объекта WshShell
 Shell,    //Экземпляр объекта Shell.Application
 Res;      //Результат нажатия кнопок в диалоговом окне
//Инициализируем константы для диалоговых окон
var vbYesNo=4,vbQuestion=32,vbYes=6;
//Создаем объект WshShell
WshShell = WScript.CreateObject("WScript.Shell");
//Создаем объект Shell.Application
Shell=WScript.CreateObject("Shell.Application");
//Выводим запрос
Res=WshShell.Popup("Открыть панель управления?",0,
 "Вызов модулей панели управления",vbQuestion+vbYesNo);
if (Res==vbYes) //Нажата кнопка Да
 //Выводим панель управления
 Shell.ControlPanelItem("");
```

```
    //Выводим запрос
    Res=WshShell.Popup("Открыть окно установки и удаления
программ?",0,
     "Вызов модулей панели управления",vbQuestion+vbYesNo);
    if (Res==vbYes) //Нажата кнопка Да
     //Выводим окно установки и удаления программ
     Shell.ControlPanelItem("Appwiz.cpl");
    //Выводим запрос
    Res=WshShell.Popup("Открыть окно установки даты и времени?",0,
     "Вызов модулей панели управления",vbQuestion+vbYesNo);
    if (Res==vbYes) //Нажата кнопка Да
     //Выводим окно установки даты и времени
     Shell.ControlPanelItem("TimeDate.cpl");
    /*************                                          Конец
*****************************************/
```

Запуск с помощью системных функций Windows

Другим вариантом запуска модулей панели управления является использование специальных функций, находящихся в библиотечном файле shell32.dll. Хотя из сценариев нельзя напрямую вызывать системные функции Windows, для этой цели можно воспользоваться стандартной утилитой RunDll32.exe, которая позволяет запускать функции, хранящиеся в библиотечных dll-файлах. В свою очередь RunDll32.exe запускается в сценарии с помощью метода `Run()` объекта `WshShell`. В качестве параметров программы RunDll32.exe нужно через запятую указать имя dll-файла и имя вызываемой функции, например:

```
    //Выводим окно установки Windows
    WshShell.Run("Rundll32.exe           shell32.dll,Control_RunDLL
appwiz.cpl,,2");
```

Здесь мы вызываем функцию `Control_RunDLL()` из файла shell32.dll. В качестве же параметров функции `Control_RunDLL()` указываются через запятую название нужного cpl-файла, имя и индекс страницы модуля, которая будет выведена на передний план (в вышеприведенной команде вызывается страница с индексом 2 ("Установка Windows") из модуля appwiz.cpl ("Установка и удаление программ")).

В листинге 11.14 приведен сценарий RunCPL2.js, в котором вызовы модулей панели управления осуществляются с помощью запуска системных функций Windows.

Листинг 11.14. Вызов модулей панели управления с помощью вызовов системных функций

```
    /*****************************************************************
    /*                                                              Имя:
RunCPL2.js                                                      */
    /*                                                             Язык:
JScript                                                        */
    /*    Описание:    Вызов    модулей    панели    управления    с
помощью              */
    /*                                              вызовов    системных
функций                              */
    /*****************************************************************
    //Объявляем переменные
    var
     WshShell, //Экземпляр объекта WshShell
     Res;      //Результат нажатия кнопок в диалоговом окне
    //Инициализируем константы для диалоговых окон
    var vbYesNo=4,vbQuestion=32,vbYes=6;
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    //Выводим запрос
    Res=WshShell.Popup("Открыть панель управления?",0,
     "Вызов модулей панели управления",vbQuestion+vbYesNo);
    if (Res==vbYes) //Нажата кнопка Да
     //Выводим панель управления
     WshShell.Run("Rundll32.exe shell32.dll,Control_RunDLL");

    //Выводим запрос
    Res=WshShell.Popup("Открыть окно установки Windows?",0,
     "Вызов модулей панели управления",vbQuestion+vbYesNo);
    if (Res==vbYes) //Нажата кнопка Да
     //Выводим окно установки Windows
          WshShell.Run("Rundll32.exe      shell32.dll,Control_RunDLL
appwiz.cpl,,2");

    //Выводим запрос
    Res=WshShell.Popup("Открыть окно установки даты и времени?",0,
     "Вызов модулей панели управления",vbQuestion+vbYesNo);
    if (Res==vbYes) //Нажата кнопка Да
     //Выводим окно установки даты и времени
          WshShell.Run("Rundll32.exe      shell32.dll,Control_RunDLL
timedate.cpl");
    /*************                                                 Конец
*********************************************/
```

Открытие папки в Проводнике Windows

С помощью объекта `Shell.Application` можно запустить Проводник Windows и открыть в нем определенную папку. Для этого используется метод `Explore()`, в качестве параметра которого указывается путь к открываемой папке; соответствующий пример приведен в листинге 11.15.

Листинг 11.15. Открытие заданной папки в Проводнике Windows

```
    /*****************************************************************
    /*                                                           Имя:
Explore.js                                                    */
    /*                                                          Язык:
JScript                                                     */
    /*   Описание:   Открытие   заданной   папки   в   Проводнике
Windows          */
    /*****************************************************************
    //Объявляем переменные
    var
     WshShell,     //Экземпляр объекта WshShell
     Shell,        //Экземпляр объекта Shell.Application
     SPath="C:\\", //Путь к открываемой папке
     Res;          //Результат нажатия кнопок в диалоговом окне
    //Инициализируем константы для диалоговых окон
    var vbYesNo=4,vbQuestion=32,vbYes=6;
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    //Создаем объект Shell.Application
    Shell=WScript.CreateObject("Shell.Application");
    //Выводим запрос
    Res=WshShell.Popup("Открыть папку "+SPath+"?",0,
     "Вызов стандартных диалогов Windows",vbQuestion+vbYesNo);
    if (Res==vbYes) //Нажата кнопка Да
     //Открываем папку в Проводнике
     Shell.Explore(SPath);
    /*************                                             Конец
*********************************************/
```

Вызов окна форматирования диска

Диалоговое окно, позволяющее форматировать диск с заданными параметрами (рис. 11.26), вызывается с помощью системной функции

`SHFormatDrive()` из библиотечного файла shell32.dll.

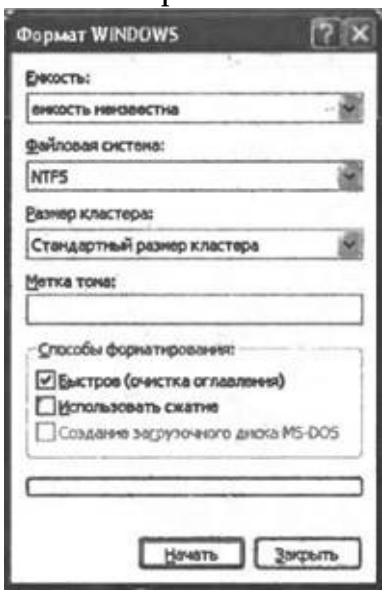


Рис. 11.26. Диалоговое окно форматирования диска

Соответствующий пример приведен в листинге 11.16.

Листинг 11.16. Вызов окна форматирования диска

```
    /*****************************************************************
    /*                                                          Имя:
FormatDisk.js                                                  */
    /*                                                         Язык:
JScript                                                       */
    /*        Описание:        Вызов        окна        форматирования
диска                          */
    /*****************************************************************
    //Объявляем переменные
    var
     WshShell, //Экземпляр объекта WshShell
     Res;      //Результат нажатия кнопок в диалоговом окне
    //Инициализируем константы для диалоговых окон
    var vbYesNo=4,vbQuestion=32,vbYes=6;
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    //Выводим запрос
    Res=WshShell.Popup("Открыть окно форматирования?",0,
     "Вызов стандартных диалогов Windows",vbQuestion+vbYesNo);
    if (Res==vbYes) //Нажата кнопка Да
```

```
     //Вызываем окно форматирования
     WshShell.Run("Rundll32.exe shell32.dll,SHFormatDrive");
    /*************                                          Конец
*****************************************/
```

Вызов окна запуска программ

Окно запуска программ открывается с помощью метода `FileRun()` объекта `Shell.Application`. Соответствующий пример приведен в листинге 11.17.

Листинг 11.17. Вызов окна запуска программ

```
    /*****************************************************************
    /*                                                          Имя:
FileRun.js                                                    */
    /*                                                             Язык:
JScript                                                      */
    /*         Описание:         Вызов         окна          запуска
программ                                  */
    /*****************************************************************
    //Объявляем переменные
    var
     WshShell, //Экземпляр объекта WshShell
     Shell,    //Экземпляр объекта Shell.Application
     Res;      //Результат нажатия кнопок в диалоговом окне
    //Инициализируем константы для диалоговых окон
    var vbYesNo=4,vbQuestion=32,vbYes=6;
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    //Создаем объект Shell.Application
    Shell=WScript.CreateObject("Shell.Application");
    //Выводим запрос
    Res=WshShell.Popup("Открыть окно запуска программ?",0,
     "Вызов стандартных диалогов Windows",vbQuestion+vbYesNo);
    if (Res==vbYes) //Нажата кнопка Да
     //Вызываем окно запуска программ
     Shell.FileRun();
    /*************                                          Конец
*****************************************/
```

Блокировка рабочей станции

Заблокировать рабочую станцию Windows XP можно с помощью вызова функции `LockWorkStation()` из библиотечного файла user32.dll. В листинге 11.18 приведен сценарий Lock.js, в котором происходит блокировка компьютера с помощью этой функции.

Листинг 11.18. Блокировка рабочей станции

```
    /*****************************************************************
    /*                                                          Имя:
Lock.js                                                     */
    /*                                                         Язык:
JScript                                                    */
    /*            Описание:            Блокировка            рабочей
станции                            */
    /*****************************************************************
    //Объявляем переменные
    var
     WshShell, //Экземпляр объекта WshShell
     Res;      //Результат нажатия кнопок в диалоговом окне
    //Инициализируем константы для диалоговых окон
    var vbYesNo=4,vbQuestion=32,vbYes=6;
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    //Выводим запрос на блокировку рабочей станции
    Res=WshShell.Popup("Заблокировать рабочую станцию?",0,
     "",vbQuestion+vbYesNo);
    if (Res==vbYes) //Нажата кнопка Да
     //Блокируем рабочую станцию
     WshShell.Run("Rundll32.exe user32.dll,LockWorkStation");
    /*************                                            Конец
*******************************************/
```

Вызов окна выключения компьютера

Из сценария WSH можно вызвать диалоговое окно, в котором производится выбор действия при завершении работы Windows (рис. 11.27).

Рис. 11.27. Диалоговое окно выключения компьютера

Для этого необходимо вызвать метод `ShutdownWindows()` объекта `Shell.Application`. Соответствующий пример приведен в листинге 11.19.

Листинг 11.19. Вызов окна выключения компьютера

```
    /*****************************************************************
    /*                                                          Имя:
ShutdownWindow.js                                            */
    /*                                                         Язык:
JScript                                                     */
    /*        Описание:        Вызов        окна        выключения
компьютера                      */
    /*****************************************************************
    //Объявляем переменные
    var
     WshShell, //Экземпляр объекта WshShell
     Shell,    //Экземпляр объекта Shell.Application
     Res;      //Результат нажатия кнопок в диалоговом окне
    //Инициализируем константы для диалоговых окон
    var vbYesNo=4,vbQuestion=32,vbYes=6;
    //Создаем объект WshShell
    WshShell = WScript.CreateObject("WScript.Shell");
    //Создаем объект Shell.Application
    Shell=WScript.CreateObject("Shell.Application");
    //Выводим запрос
    Res=WshShell.Popup("Открыть окно выключения компьютера?",0,
     "Вызов стандартных диалогов Windows",vbQuestion+vbYesNo);
    if (Res==vbYes) //Нажата кнопка Да
     //Вызываем окно выключения компьютера
     Shell.ShutdownWindows();
    /*************                                            Конец
*******************************************/
```

Использование технологии Windows Management Instrumentation (WMI)

В Windows XP/2000 ядром системы управления является технология WMI — Windows Management Instrumentation. WMI — это глобальная концепция настройки, управления и слежения за работой различных частей корпоративной компьютерной сети. В частности, используя WMI, можно из сценария WSH контролировать и изменять параметры самых разнородных физических и логических элементов компьютерной системы, в качестве которых могут выступать, например, файл на жестком диске, запущенный экземпляр приложения, системное событие, сетевой пакет или установленный в компьютере процессор. Очень важно, что при этом для доступа ко всем элементам используется единый интерфейс с помощью CIMOM — Common Information Model Object Manager — базы данных объектов, представляющих эти элементы. Это позволяет, в частности, быстро получать информацию разнообразного типа об объектах с помощью запросов на языке SQL. Другой важной особенностью WMI является то, что этот же интерфейс можно использовать для дистанционного управления компьютерами в сети (естественно, если на локальной и удаленной машине установлен WMI, а у пользователя, который выполняет удаленное администрирование, имеются соответствующие права).

Технология WMI — это созданная фирмой Microsoft реализация *модели управления предприятием на базе Web* (WBEM, Web-Based Enterprise Management), которая была разработана и принята рабочей группой по *управлению распределенными системами* (DMTF, Distributed Management Task Force), при участии таких компаний, как BMC Software, Cisco Systems, Compaq Computer, Intel и Microsoft. Задачей WBEM была разработка стандартного набора интерфейсов для управления информационной средой предприятия.

В WBEM информация интерпретируется в рамках модели Common Information Model (CIM). CIM представляет собой стандартную схему именования для физических и логических компонентов компьютера. К любому элементу CIM можно обратиться с помощью объектно-ориентированной терминологии:

□ класс CIM — это шаблон управляемых элементов, имеющий свойства и методы;

□ объект CIM — это экземпляр класса, представляющий базовый

компонент системы;

□ схема (schema) — это совокупность классов, описывающая систему в целом.

В Windows используются две схемы: CIM (соответствует спецификации CIM 2.0) и Win32 (расширяет спецификацию CIM 2.0).

Замечание

Объекты WMI также могут использоваться и в Windows 9x/ME/NT, для этого нужно скачать с сервера Microsoft(http://www.microsoft.com/downloads/release.asp?ReleaseID=18490).

Здесь мы не будем рассматривать классы, свойства и методы, которые поддерживает WMI, т.к. даже поверхностное ознакомление с ними потребовало бы отдельной книги, а лишь приведем несколько простых примеров сценариев, из которых станет ясно, каким образом происходит соединение с WMI, запрос нужной информации и использование объектов WMI.

Доступ к свойствам файла

Первый пример, который мы рассмотрим, будет посвящен работе с файловой системой. Мы напишем сценарий FileInfoWMI.js, в котором с помощью WMI будет формироваться диалоговое окно с информацией о файле C:\boot.ini (рис. 11.28).

Рис. 11.28. Свойства файла C:\boot.ini

Замечание

Напомним, что из сценария к файловой системе можно получить доступ с помощью стандартного объекта `FileSystemObject`, однако использование WMI дает возможность собрать более полную информацию.

Для доступа к файлу на диске нужно создать для этого файла объект класса `DataFile` схемы CIM. Как и при использовании объектов ADSI, это делается с помощью JScript-функции `GetObject()`, в качестве параметра которой указывается строка вида "`winMgmts:`*`Prefix_class.Propert`*`y=`*`Value`*", где параметр *`Prefix`* соответствует используемой схеме (`CIM` или `Win32`), *`Class`* задает имя требуемого класса, *`Property`* соответствует имени свойства класса, а*`Value`* определяет конкретное значение этого свойства. В нашем случае нужный объект (переменная `File`) создается следующим образом:

```
//Создаем объект класса CIM_DataFile для файла C:\boot.ini
File=GetObject("winMgmts:CIM_DataFile.Name='C:\\boot.ini'")
```

После этого свойства файла извлекаются обычным образом из переменной `File`:

```
//Инициализируем символьную переменную SInfo
SInfo="Информация о файле "+File.Name+"\n\n";
//Извлекаем свойства файла
SInfo+="Имя:\t\t"+File.Name+"\n";
…
//Определяем, доступен ли файл для чтения и записи
SInfo+="\n";
if (File.Readable) SInfo+="Файл доступен для чтения\n"
else SInfo+="Файл не доступен для чтения\n";
if (File.Writeable) SInfo+="Файл доступен для записи\n"
else SInfo+="Файл не доступен для записи\n";
```

Сформированная символьная переменная `SInfo` выводится на экран с помощью метода `Echo()` объекта `WScript`:

```
WScript.Echo(SInfo);
```

Полностью текст сценария FileInfoWMI.js приведен в листинге 11.20.

Листинг 11.20. Доступ к свойствам файла с помощью WMI

```
    /*****************************************************************
    /*                                                              Имя:
FileInfoWMI.js                                                  */
    /*                                                             Язык:
JScript                                                        */
    /*   Описание:   Доступ   к   свойствам   файла   с   помощью
WMI                 */
    /*****************************************************************
    //Объявляем переменные
    var
     File,  //Объект класса CIM_DataFile
     SInfo; //Строка для вывода на экран
    //Функция для форматирования символьного представления даты
    function StrDate(d) {
     var s;
     s=d.substr(6,2)+"."+d.substr(4,2)+"."+d.substr(0,4)
     return s;
    }
    /*************                                              Начало
*********************************************/
    //Создаем объект класса CIM_DataFile для файла C:\boot.ini
    File=GetObject("winMgmts:CIM_DataFile.Name='C:\\boot.ini'")
    //Инициализируем символьную переменную SInfo
    SInfo="Информация о файле "+File.Name+"\n\n";
    //Извлекаем свойства файла
    SInfo+="Имя:\t\t"+File.Name+"\n";
    SInfo+="Путь:\t\t"+File.Path+"\n";
    SInfo+="Диск:\t\t"+File.Drive+"\n";
    SInfo+="Размер:\t\t"+File.FileSize+"\n";
    SInfo+="Создан:\t\t"+StrDate(File.CreationDate)+"\n";
    SInfo+="Изменен:\t\t"+StrDate(File.LastModified)+"\n";
    SInfo+="Открыт:\t\t"+StrDate(File.LastAccessed)+"\n";
    SInfo+="Короткое имя:\t"+File.EightDotThreeFileName+"\n";
    SInfo+="Расширение:\t"+File.Extension+"\n";
    SInfo+="Тип:\t"+File.FileType+"\n";
    //Определяем атрибуты файла
    SInfo+="\n";
    SInfo+="Атрибуты:\n";
    if (File.Archive) SInfo+="\tАрхивный\n";
    if (File.Hidden) SInfo+="\tСкрытый\n";
    if (File.System) SInfo+="\tСистемный\n";
    if      (File.Compressed)      SInfo+="\tСжат      с      помощью
"+File.CompressionMethod+"\n";
    if     (File.Encrypted)     SInfo+="\tЗашифрован     с     помощью
"+File.EncryptionMethod+"\n";
```

```
    //Определяем, доступен ли файл для чтения и записи
    SInfo+="\n";
    if (File.Readable) SInfo+="Файл доступен для чтения\n"
    else SInfo+="Файл не доступен для чтения\n";
    if (File.Writeable) SInfo+="Файл доступен для записи\n"
    else SInfo+="Файл не доступен для записи\n";
    //Выводим сформированную строку на экран
    WScript.Echo(SInfo);
    /*************                                                Конец
*******************************************/
```

Список всех запущенных процессов

В следующих двух примерах мы будем работать с запущенными в операционной системе процессами.

Создадим сценарий ListProcesses.js, который будет выводить на экран имена всех запущенных процессов (рис. 11.29).

Рис. 11.29. Список всех запущенных в системе процессов

Первое, что необходимо сделать в сценарии — подключиться к службе Windows Management service, т.е. создать корневой элемент WMI, который содержит в себе все остальные.

Для этого в качестве параметра функции `GetObject()` указывается "`winMgmts:`"; в нашем примере мы соединяемся с WMI внутри блока `try`, что позволяет обработать возможные исключительные ситуации:

```
    try {
     //Соединяемся с WMI
     WMI=GetObject("winMgmts:");
```

```
} catch (e) {
 //Обрабатываем возможные ошибки
 if (e != 0) {
  //Выводим сообщение об ошибке
  Mess="Ошибка при соединении с WMI";
  WshShell.Popup(Mess, 0, "Запущенные процессы", vbCritical);
  //Выходим из сценария
  WScript.Quit();
 }
}
```

Запущенным процессам соответствует класс `Process` схемы `Win32`. Коллекция всех процессов создается с помощью выполнения следующего SQL-запроса:

```
SELECT * FROM Win32 Process
```

Таким образом, можно сказать, что класс `Win32_Process` является аналогом таблицы базы данных; сам запрос выполняется с помощью метода `ExecQuery()`:

```
Processes=new Enumerator(WMI.ExecQuery("SELECT * FROM Win32_Process"));
```

После создания коллекции мы просматриваем в цикле `while` все ее элементы, каждый из которых соответствует одному процессу, и добавляем имя процесса, хранящееся в свойстве `Name`, к переменной `SList`:

```
//Инициализируем строку SList
SList="Запущенные процессы\n\n";
//Цикл по всем элементам коллекции
while (!Processes.atEnd()) {
 //Извлекаем текущий элемент коллекции (запущенный процесс)
 Process=Processes.item();
 //Формируем строку с именами процессов
 SList+=Process.Name+"\n";
 //Переходим к следующему элементу коллекции
 Processes.moveNext();
}
```

После выхода из цикла переменная `SInfo` выводится на экран с помощью метода `Echo()` объекта `WScript`:

```
WScript.Echo(SInfo);
```

Полностью текст сценария ListProcesses.js приведен в листинге 11.21.

Листинг 11.21. Вывод на экран списка всех запущенных процессов

```
/*****************************************************************
/*                                                          Имя: ListProcesses.js                                                  */
```

```
    /*                                                          Язык:
JScript                                                     */
    /*  Описание:  Вывод  на  экран  списка  всех  запущенных  на
локальной     */
    /*                                          рабочей    станции
процессов                                 */
    /*****************************************************************
    var
     WMI,       //Экземпляр WMI
     Processes, //Коллекция процессов
     Process,   //Экземпляр коллекции
     SList;     //Строка для вывода на экран
    //Инициализируем константы для диалоговых окон
    var vbCritical=16;
    try  {
     //Соединяемся с WMI
     WMI=GetObject("winMgmts:");
    } catch (e) {  //Обрабатываем возможные ошибки
     if (e != 0) {
      //Выводим сообщение об ошибке
      Mess="Ошибка при соединении с WMI";
      WshShell.Popup(Mess,0,"Запущенные процессы",vbCritical);
      //Выходим из сценария
      WScript.Quit();
     }
    }
    //Создаем коллекцию всех запущенных процессов
    Processes=new    Enumerator(WMI.ExecQuery("SELECT    *    FROM
Win32_Process"));
    //Инициализируем строку SList
    SList="Запущенные процессы\n\n";
    //Цикл по всем элементам коллекции
    while (!Processes.atEnd()) {
     //Извлекаем текущий элемент коллекции (запущенный процесс)
     Process=Processes.item();
     //Формируем строку с именами процессов
     SList+=Process.Name+"\n";
     //Переходим к следующему элементу коллекции
     Processes.moveNext();
    }
    //Выводим информацию на экран
    WScript.Echo(SList);
    /*************                                            Конец
*******************************************/
```

Закрытие всех экземпляров запущенного приложения

WMI позволит нам закрывать сразу все экземпляры какого-либо запущенного приложения.

В сценарии KillNotepads.js мы будем закрывать все копии Блокнота (Notepad.exe). Как и в предыдущем примере, сначала мы соединяемся с WMI внутри блока `try` конструкции `try…catch`:

```
try {
 //Соединяемся с WMI
 WMI=GetObject("winMgmts:");
} catch (e) {
 //Обрабатываем возможные ошибки
 if (e != 0) {
  //Выводим сообщение об ошибке Mess="Ошибка при соединении с WMI";
      WshShell.Popup(Mess,   0,   "Закрытие   всех   Блокнотов", vbCritical);
  //Выходим из сценария
  WScript.Quit();
 }
}
```

Далее нам нужно получить коллекцию всех процессов с именем "Notepad.exe". Для этого мы выполняем соответствующий SQL-запрос, текст которого предварительно заносится в переменную `SQuery`:

```
//Формируем текст запроса
SQuery="SELECT * FROM Wln32_Process WHERE Name='Notepad.exe'"
//Создаем коллекцию-результат запроса
Processes=new Enumerator(WMI.ExecQuery(SQuery));
```

Теперь, имея коллекцию нужных нам процессов, мы в цикле `while` перебираем все ее элементы, вызывая для каждого элемента (запущенного экземпляра Блокнота) метод `Terminate()`, который завершает этот процесс:

```
//Цикл по всем элементам коллекции
while (!Processes.atEnd()) {
  //Извлекаем текущий элемент коллекции (процесс с именем Notepad.exe)
 Process=Processes.item();
 try {
  //Завершаем процесс
  Process.Terminate();
 } catch (e) {
  //Обрабатываем возможные ошибки if (e != 0) {
  //Выводим сообщение об ошибке
  Mess="Ошибка при закрытии текущего экземпляра";
      WshShell.Popup(Mess,   0,   "Закрытие   всех   Блокнотов", vbCritical);
 }
}
```

```
//Переходим к следующему элементу коллекции
Processes.moveNext();
```

Полностью текст сценария KillNotepads.js приведен в листинге 11.22.

Листинг 11.22. Закрытие всех запущенных экземпляров Блокнота

```
/*****************************************************************
/*                                                          Имя:
KillNotepads.js                                            */
/*                                                         Язык:
JScript                                                   */
/*    Описание:    Закрытие    всех    запущенных    экземпляров
Блокнота          */
/*****************************************************************
var
 WMI,       //Экземпляр WMI
 SQuery,    //Текст запроса
 Processes, //Коллекция процессов
 Process,   //Экземпляр коллекции
 WshShell;  //Объект WshShell
//Инициализируем константы для диалоговых окон
var vbCritical=16;
//Создаем объект WshShell
WshShell = WScript.CreateObject("WScript.Shell");

try {
 //Соединяемся с WMI
 WMI=GetObject("winMgmts:");
} catch (e) {  //Обрабатываем возможные ошибки
 if (e != 0) {
  //Выводим сообщение об ошибке
  Mess="Ошибка при соединении с WMI";
  WshShell.Popup(Mess,0,"Закрытие всех Блокнотов",vbCritical);
  //Выходим из сценария
  WScript.Quit();
 }
}

//Формируем текст запроса
SQuery="SELECT * FROM Win32_Process WHERE Name='Notepad.exe'"
//Создаем коллекцию-результат запроса
Processes=new Enumerator(WMI.ExecQuery(SQuery));
//Цикл по всем элементам коллекции
while (!Processes.atEnd()) {
```

```
      //Извлекаем текущий элемент коллекции (процесс с именем
Notepad.exe)
     Process=Processes.item();
     try {
      //Завершаем процесс
      Process.Terminate();
     } catch (e) {  //Обрабатываем возможные ошибки
      if (e != 0) {
       //Выводим сообщение об ошибке
       Mess="Ошибка при закрытии текущего экземпляра";
       WshShell.Popup(Mess,0,"Закрытие всех Блокнотов",vbCritical);
      }
     }
     //Переходим к следующему элементу коллекции
     Processes.moveNext();
    }
    /*************                                                Конец
*****************************************/
```

Заключение

Для того чтобы автоматизировать и существенно облегчить повседневную работу пользователей и администраторов компьютерных систем, базирующихся на Windows, следует использовать мощные возможности, которые предоставляет сервер сценариев Windows Script Host (WSH), позволяющий создавать полноценные сценарии, работающие непосредственно в операционной системе и использующие внешние объекты ActiveX.

Конечно, написание сценариев WSH по сравнению, скажем, с созданием командных файлов поначалу может показаться довольно непростой задачей — кроме знания специальных языков сценариев (например, VBScript или JScript) и представления об объектах ActiveX, нужно знать, по крайней мере, собственную объектную модель WSH и структуру объекта FileSystemObject. Однако с помощью этих средств уже можно писать сценарии, которые имеют полный доступ к файловой системе компьютера, системному реестру и ресурсам локальной сети.

Основным же назначением WSH является интеграция с помощью сценариев различных современных технологий компании Microsoft, предназначенных, например, для обеспечения доступа к настройкам операционной системы и установленного оборудования (Windows Management Instrumentation, WMI), базам данных (ADO, ActiveX Data Objects), службам каталогов (ADSI, Active Directory Service Interface), или для управления приложениями семейства Microsoft Office. В книге мы постарались дать общее представление об этих технологиях, привести практические примеры их использования из сценариев, а также осветить вопросы безопасности при работе со сценариями.

Итак, для составления грамотных и профессиональных сценариев WSH необходимо разобраться в нескольких смежных технологиях, однако затраченные усилия наверняка будут вознаграждены — с помощью ActiveX-сценариев можно быстро решать возникающие перед администратором операционной системы задачи практически любой сложности!

Приложение 1
Справочник по языку JScript

Язык JScript — это разработанный Microsoft интерпретируемый объектно-ориентированный язык сценариев, который первоначально предназначался для создания динамических HTML-страниц. Отметим, что JScript не является урезанной версией какого-либо другого языка программирования, хотя по синтаксису он похож на языки Java и C. В этом приложении мы кратко рассмотрим те возможности и свойства JScript, которые могут потребоваться при составлении сценариев, выполняемых с помощью WSH, не затрагивая при этом вопросы, связанные с составлением сценариев для HTML-страниц.

Строки кода и комментарии

В конце каждого оператора JScript нужно ставить точку с запятой. Например:

```
var theSum =0, i;
sum[0] = 0;
```

При этом один оператор может располагаться на нескольких строках, например:

```
var
 theSum = 0,
 i;
sum[0] = 0;
```

Комментарии в JScript могут быть двух видов. Для комментария, который занимает не более одной строки, можно использовать символы `//`. Например:

```
//Этот комментарий занимает всю строку
theSum=1; //А этот часть строки
```

Если же в текст сценария необходимо внести комментарий, расположенный на нескольких строках, то текст такого комментария нужно помещать внутри блока `/*…*/`.

Например:

```
/* Это комментарий
 на нескольких
 строках */
```

Переменные

В сценариях JScript, как и в любом другом языке программирования, можно использовать переменные, обращаясь к ним по имени. При этом переменные могут быть как глобальными (доступными из любого места сценария), так и локальными (область действия ограничивается функцией, в которой они определены).

Хорошим тоном считается предварительное объявление используемых переменных с помощью ключевого слова `var`, хотя это является обязательным условием только для локальных переменных, определенных в функциях. Пример объявления переменной имеет вид:

```
var MyVariable;
```

При объявлении тип переменной явным образом не указывается (как это делается, например, в языках C или Pascal). Определенный тип переменной присваивается только тогда, когда в нее записывается какое-либо значение.

Язык JScript является регистро-зависимым, т.е. имена `MyVariable` и `myvariable` представляют разные переменные. Кроме этого, при выборе имен переменных следует придерживаться следующих правил:

□ имя переменной должно начинаться с буквы или с символов "_", "$" и может состоять только из букв, цифр, а также символов "_", "$";

□ имя переменной не должно совпадать с зарезервированными ключевыми словами языка JScript.

Список ключевых слов JScript приведен в табл. П1.1.

Таблица П1.1. *Зарезервированные ключевые слова JScript*

```
break    default false    new    true
case     delete  finally  null   try
catch    do      for      return typeof
class    else    function super  var
const    enum    if       switch void
continue export  import   this   while
debugger extends in       throw  with
```

Значения переменным в JScript присваиваются с помощью оператора присваивания "=". Например:

```
var MyVariable;
MyVariable = "Привет!";
```

Здесь мы объявили переменную `MyVariable` и записали в нее текстовую строку. Однако далее в любом месте сценария мы можем

присвоить переменной `MyVariable` числовое значение (при этом тип переменной изменится), например:

```
MyVariable = 10;
```

Кроме этого, переменной можно присвоить специальное значение `null`:

```
MyVariable = null;
```

В этом случае переменной `MyVariable` не назначается никакого определенного типа (пустой тип). Такое присваивание применяется в тех случаях, когда необходимо объявить переменную и проинициализировать ее, не присваивая этой переменной никакого определенного типа и значения.

Типы данных

В JScript поддерживаются шесть типов данных, главными из которых являются числа, строки, объекты и логические данные. Оставшиеся два типа — это null (пустой тип) и undefined (неопределенный тип).

Числа

В сценариях JScript числа могут использоваться в различных форматах.

□ Целые числа в диапазоне от -999 999 999 999 999 до 999 999 999 999 999. Кроме обычного десятичного, целые числа могут записываться в восьмеричном (префикс "0" и цифры 0–7) или в шестнадцатиричном (префикс "0x", цифры 0–9, символы "A", "B", "C", "D", "E" и "F") виде. Например, восьмеричное число 0377 и шестнадцатеричное 0xFF равны десятичному 255.

□ Вещественные числа, которые могут быть записаны как с плавающей точкой (например, -10.567), так и в научной нотации (например, 10567E-3, что равно 10.567 ("число 10 567 умножить на 10 в степени -3")). Значения вещественных переменных и констант должны лежать в диапазоне от -Number.MAX_VALUE до Number.MAX_VALUE, где параметр Number.MAX_VALUE является специальным числовым значением, которое равно наибольшему вещественному числу, с которым может работать JScript (Number.MAX_VALUE приблизительно равно 1.79E +308).

Кроме Number.MAX_VALUE в JScript имеются еще несколько специальных числовых значений.

□ NaN (Not a Number) — так называемое "нечисло", которое не соответствует никакому числу (это значение генерируется в тех случаях, когда результат выполнения операции не может быть представлен в виде числа, например, при преобразовании строки "1s2" к числовому типу).

□ Положительная бесконечность Number.POSITIVE_INFINITY (число, которое больше, чем Number.MAX_VALUE).

□ Отрицательная бесконечность Number.NEGATIVE_INFINITY (число, которое меньше, чем -Number.MAX_VALUE).

□ Самое близкое число к нулю Number.MIN_VALUE (примерно равно 2.22E-308). Все числа, большие -Number.MIN_VALUE, но меньшие

Number.MIN_VALUE, считаются равными нулю.

Текстовые строки

Текстовые строки — это последовательность символов, заключенных в одинарные или двойные кавычки, например:

```
"Привет!"
'Большой привет!'
'И тогда он крикнул "Берегись!"'
```

Строка может иметь нулевую длину (пустая строка):

```
MyVariable = "";
```

В JScript можно также использовать специальные комбинации символов, с помощью которых в строки включаются некоторые неотображаемые символы или символы, имеющие специальное значение. Каждая из этих комбинаций (escape-последовательностей) начинается с символа обратной косой черты "\" (табл. П1.2).

Таблица П1.2. *Специальные комбинации символов*

Escape-последовательность	Описание
\b	Backspace <←>
\f	Перевод формата
\n	Перевод строки
\r	Возврат каретки
\t	Горизонтальная табуляция (<Ctrl>+<I>)
\'	Одинарная кавычка
\"	Двойная кавычка
\\	Обратная косая черта

Объекты

В JScript под объектом понимается совокупность *свойств* и *методов.* Метод — это внутренняя функция объекта, свойство — это одно значение какого-либо типа или несколько таких значений (в виде массива или объекта), хранящихся внутри объекта. Поддерживаются три вида объектов:

- встроенные (внутренние) объекты;
- объекты, создаваемые программистом в сценарии;
- внешние объекты (например, объекты WSH). Более подробно

объекты будут рассмотрены ниже.

Логические данные

Логические данные предназначены для выполнения операций сравнения, а также для использования в условных операторах. При этом логические данные имеют только два значения: `true` (истина) и `false` (ложь). Отметим, что в JScript эти значения никак не соотносятся с числами 1 и 0.

Null (пустой тип) и undefined (неопределенный тип)

Если переменная была объявлена с помощью ключевого слова `var`, но ей еще ни разу не присваивалось значение, она имеет неопределенный тип `(undefined)`:

```
var MyVariable;
```

После выполнения этой строки переменная `MyVariable` имеет тип `undefined`. Как уже отмечалось выше, если теперь присвоить переменной значение `null`, то эта переменная будет типа `null` (пустой тип):

```
MyVariable = null;
```

Преобразование типов данных

Одной из особенностей языка JScript является то, что если в выражениях встречаются переменные разных типов, то автоматически происходит преобразование всех числовых данных в строковое представление. Например, следующие логические выражения будут равны `true`:

```
"100" == 100
false == 0
```

(здесь "==" означает оператор сравнения). Для преобразования строк в числа нужно применять две специальные функции: `parseInt` (преобразование к целому числу) и `parseFloat` (преобразование к числу с плавающей запятой). Например, после выполнения следующих строк:

```
var s="";
s=(parseInt("3")-2)+"3";
```

значением переменной `s` будет строка "`13`".

Операторы

В JScript поддерживаются операторы различных типов, которые похожи на операторы языка C.

Унарные операторы

Унарными называются операторы, которые применяются к одному операнду (табл. П1.3).

Таблица П1.3. *Унарные операторы*

Оператор	Описание
-	Изменение знака на противоположный
!	Дополнение. Используется для изменения значения логической переменной на противоположное
++	Увеличение значения числовой переменной на единицу (инкремент). Может применяться как префикс переменной или как ее суффикс
--	Уменьшение значения числовой переменной на единицу (декремент). Может применяться как префикс переменной или как ее суффикс

Бинарные операторы

Бинарными называются операторы, которые соединяют два операнда (табл. П1.4).

Таблица П1.4. *Бинарные операторы*

Оператор	Описание	Оператор	Описание
-	Вычитание	/	Деление
+	Сложение	%	Вычисление остатка от деления
*	Умножение		

Операторы побитовых логических операций и сдвига

Эти операторы позволяют производить над числовыми переменными побитовые операции, описанные в табл. П1.5.

Таблица П1.5. *Операторы побитовых логических операций и сдвига*

Оператор	Описание
`&`	Логическое И
`|`	Логическое ИЛИ
`^`	Логическое ИСКЛЮЧАЮЩЕЕ ИЛИ
`~`	Логическое НЕ
`>>`	Сдвиг вправо
`<<`	Сдвиг влево
`>>>`	Сдвиг вправо с заполнением освобождаемых разрядов нулями

Замечание

Перед использованием операторов из табл. П1.5. значения переменных преобразуются в 32-разрядные целые числа.

Операторы присваивания

В JScript, как и в языке C, для изменения содержимого переменных можно комбинировать оператор присваивания "=" с другими операторами (табл. П1.6).

Таблица П1.6. *Комбинации оператора присваивания и других операторов*

Оператор	Описание
`=`	Простое присваивание
`+=`	Увеличение численного значения или конкатенация (склеивание) строк
`-=`	Уменьшение численного значения
`*=`	Умножение
`/=`	Деление
`%=`	Вычисление остатка от деления
`>>=`	Сдвиг вправо

`>>>=`	Сдвиг вправо с заполнением освобождаемых разрядов нулями
`<<=`	Сдвиг влево
`|=`	Логическое ИЛИ
`&=`	Логическое И
`^=`	Логическое ИСКЛЮЧАЮЩЕЕ ИЛИ

Операторы отношения

Операторы отношения используются для сравнения значений нескольких переменных. Эти операторы, описанные в табл. П1.7, могут возвращать только логические значения `true` или `false`.

Таблица П1.7. *Операторы отношения*

Оператор	Условие, при котором возвращается `true`
`>`	Левый операнд больше правого
`>=`	Левый операнд больше или равен правому
`<`	Левый операнд меньше правого
`<=`	Левый операнд меньше или равен правому
`==`	Левый операнд равен правому
`!=`	Левый операнд не равен правому

Также в условных операторах применяются логические операторы (табл. П1.8).

Таблица П1.8. *Логические операторы*

Оператор	Описание
`||`	Оператор отношения "ИЛИ". Возвращает `true`, если один из операндов равен `true`. В противном случае возвращает `false`
`&&`	Оператор отношения "И". Возвращает `true`, если оба операнда равны `true`. В противном случае возвращает `false`

Условные операторы

В JScript поддерживается условный оператор `if…else`. Общий вид этого оператора показан ниже:

```
if (условие) выражение_1
```

[else *выражение_2*]

При выполнении оператора if…else оценивается логическое условие, заданное в круглых скобках после ключевого слова if. Если в результате оценки условия получилось значение true, то выполняется первое выражение. В противном случае выполняется второе выражение (если оно присутствует).

Оператор if…else может быть вложенным. Заметим, что если в первом или втором выражении нужно расположить несколько операторов, то их следует выделить фигурными скобками:

```
if (x == 5) {
 if (y == 6) z = 17;
} else z = 20;
```

В JScript также существует специальный тип условного оператора, который называется оператором "?:". В общем виде он записывается так:

условие ? *выражение_1* : *выражение_2*

При вычислении оператора "?:" вначале оценивается условие, расположенное в левой части. Если оно равно true, то выполняется первое выражение, в противном случае — второе. Например:

```
hours += (theHour >=12) ? " PM" : " AM";
```

Операторы циклов

Microsoft JScript поддерживает несколько типов циклов: цикл for, цикл for…in, цикл while, цикл do…while. Рассмотрим каждый из них подробнее.

Цикл *for*

В общем случае оператор цикла **for** имеет три раздела (инициализация, условие и итерация) и записывается следующим образом:

```
for ([инициализация;] [условие;] [итерация]){
 тело цикла
}
```

В разделе инициализации обычно выполняется присваивание начальных значений переменным цикла. Здесь можно объявлять новые переменные с помощью ключевого слова var.

Во втором разделе задается условие выхода из цикла. Это условие оценивается каждый раз при прохождении цикла. Если в результате такой оценки получается логическое значение true, то начинают выполняться

строки из тела цикла; в противном случае происходит выход из цикла. В том случае, когда условие было ложным с самого начала (при первой проверке), цикл не будет выполнен ни разу.

Раздел итерации применяется для изменения значений переменных цикла (например, увеличения или уменьшения значения счетчика цикла).

Пример использования цикла `for` приведен в листинге П1.1.

Листинг П1.1. Пример использования цикла `for`

```
    var howFar = 11; // Верхний предел для счетчика цикла
    var sum = new Array(howFar); //Массив из 11 элементов, индексы
от 0 до 10
    var theSum = 0;
    sum[0] = 0;
    //Цикл выполнится 10 раз
    for(var icount = 1; icount < howFar; icount++) {
     theSum += icount;
     sum [icount] = theSum;
    }
    var newSum = 0;
    //Цикл не выполнится ни разу
    for(var icount = 1; icount > howFar; icount++) {
     newSum += icount;
    }
    var sum = 0;
    //Бесконечный цикл
    for(var icount = 1; icount > 0; icount++) {
     sum += icount;
    }
```

Цикл *for…in*

Оператор цикла `for…in` предназначен для просмотра всех свойств объекта. Для каждого свойства указанный цикл выполняет операторы, содержащиеся в теле цикла:

```
    for (переменная in объект) {
     тело цикла
    }
```

Цикл `for…in` можно использовать для вывода на экран всех свойств объекта в одном цикле:

```
    function objectDisplay(obj) {
     var displayLine;
```

```
 for (var prop in obj) {
  displayLine=obj.name+"."+prop+"="+obj[prop];
  WScript.Echo(displayLine)
 }
 WScript.Echo("--------------");
}
```

Цикл *while*

Цикл `while` похож на цикл `for`. В нем также условие выхода из цикла проверяется перед выполнением итерации, однако в цикле `while`, в отличие от `for`, нет встроенного счетчика и выражения, его изменяющего.

Оператор `while` записывается в следующем виде:

```
while (условие) {
 тело цикла
}
```

Пример использования цикла `while` приведен в листинге П1.2.

Листинг П1.2. Пример использования цикла `while`

```
var theMoments = "";
var theCount = 42; // Начальное значение счетчика цикла
while (theCount >= 1) {
 if (theCount > 1) {
  theMoments = "До взрыва осталось " + theCount + " сек!";
 } else {
  theMoments = "Осталась секунда!";
 }
 theCount--; // Уменьшаем значение счетчика
}
theMoments = "ВЗРЫВ!";
```

Цикл *do…while*

Этот цикл является примером цикла с пост-условием и записывается в следующем виде:

```
do {
 тело цикла
} while (условие);
```

В этом случае цикл выполняется до тех пор, пока проверяемое после ключевого слова `while` условие не станет ложным (`false`). Так как условие

проверяется уже после прохождения тела цикла, то операторы внутри цикла do…while выполнятся по крайней мере один раз.

Пример использования цикла do…while приведен в листинге П1.3.

Листинг П1.3. Пример использования цикла do…while

```
    var howFar = 11; // Верхний предел для счетчика цикла
    var sum = new Array(howFar); //Массив из 11 элементов, индексы
от 0 до 10
    var theSum = 0;
    sum[0] = 0;
    var icount - 1;
    //Цикл выполнится 10 раз
    do {
    theSum += icount; sum [icount] = theSum;
    icount++; }
    while (icount < howFar);
```

Внутри цикла любого вида можно применять два специальных оператора: break и continue.

Оператор *break*

С помощью оператора break можно прервать выполнение цикла в любом месте; управление при этом передастся на оператор, следующий сразу за циклом.

```
    var i = 0;
    while (i < 100) {
     if (i == 50) break;
     i++;
    }
    i++; // Значение i станет равным 51
```

Оператор *continue*

Оператор continue прерывает текущую итерацию цикла и начинает новую. В различных видах циклов этот оператор производит следующие действия:

□ в циклах while и do…while проверяется условие цикла и если оно равно true, то вновь выполняется тело цикла;

□ в цикле for изменяется значение счетчика в разделе итерации,

проверяется условие цикла и если оно равно `true`, то тело цикла выполняется вновь;

□ в цикле `for…in` переменная цикла переходит к следующему полю объекта, и тело цикла выполняется вновь.

Пример использования оператора `continue`:

```
var s = "", i=0;
while (i < 10) {
 i++;
 // Пропускаем число 5
 if (i==5) {
  continue;
 }
}
s += i;
```

Прочие операторы

Рассмотрим еще несколько часто применяемых операторов (см. табл. П1.9).

Таблица П1.9. *Прочие операторы*

Оператор	Описание
`.`	Точка. Применяется для доступа к свойству объекта или для вызова его метода
`[]`	Квадратные скобки. Применяются для индексирования массива
`()`	Скобки. Применяются либо для изменения порядка вычисления выражений, либо для передачи параметров функциям
`,`	Запятая. Применяется для многократных вычислений

С помощью оператора "," можно, например, в разделе итерации цикла `for` изменять значение сразу нескольких переменных:

```
var i, j;
j = 10;
for (i = 0; i<=10; i++, j--) {
 …
}
```

Обработка исключительных ситуаций

Во время выполнения сценария могут возникать различные

исключительные ситуации (например, деление на ноль или попытка открыть несуществующий файл), которые приводят к ошибкам времени выполнения — при этом на экран выводится диалоговое окно с сообщением об ошибке и выполнение сценария прекращается.

Существует возможность написать код сценария таким образом, чтобы исключительные ситуации не приводили к завершению работы, а обрабатывались бы внутри сценария. Для осуществления подобной обработки исключительных ситуаций в JScript необходимо использовать конструкцию `try…catch`. Синтаксис следующий:

```
try
 Защищенный блок
catch (except)
 Блок обработки исключительных ситуаций
```

После ключевого слова `try` здесь записываются те операторы, при выполнении которых потенциально может возникнуть ошибка. В качестве параметра *except* надо использовать любое имя переменной (предварительно объявлять эту переменную не нужно). Если в защищенном блоке не возникло ошибки, то значение этой переменной будет равно нулю; в случае же возникновения исключительной ситуации управление сразу передается в блок `catch`, при этом переменная *except* как экземпляр объекта `Error` будет содержать два свойства: `error` — числовой код возникшей ошибки и `description` — краткое описание ошибки.

В качестве примера приведем часть сценария, в которой происходит обработка исключительных ситуаций при подключении сетевого диска:

```
try {
 //Подключаем сетевой диск
 WshNetwork.MapNetworkDrive(Drive, NetPath);
} catch (e) {
 //Обрабатываем возможные ошибки
 if (e != 0) {
  //Выводим сообщение об ошибке
    Mess="Ошибка при подключении диска " + Drive + " к " +
NetPath+
   "\nКод ошибки: "+e.number+"\nОписание: "+e.description;
     WshShell.Popup(Mess,  0,  "Подключение сетевого диска",
vbCritical);
 }
}
```

Порядок выполнения операторов

В табл. П1.10 операторы языка JScript расположены по старшинству,

т.е. в составных операторах первыми будут выполняться те из них, которые стоят в этой таблице выше. Если операторы расположены в одной строке таблицы, то они выполняются слева направо.

Таблица П1.10. *Порядок выполнения операторов*

Оператор	Описание
`. [] ()`	Доступ к полю объекта, индексирование в массиве, вызов функции
`++ -- - ~ !`	Унарные операторы
`* / %`	Умножение, деление, вычисление остатка от деления
`+ - +`	Сложение, вычитание, конкатенация строк
`<< >> >>>`	Битовые сдвиги
`< <= > >=`	Меньше, меньше или равно, больше, больше или равно
`== !=`	Равенство, неравенство
`&`	Логическое И
`^`	Логическое ИСКЛЮЧАЮЩЕЕ ИЛИ
`|`	Логическое ИЛИ
`&&`	Оператор отношения И
`||`	Оператор отношения ИЛИ
`?:`	Условный оператор
`= += -= *= /= %= >>= >>>= <<= |= &= ^=`	Присваивание
`,`	Многократное вычисление

Функции

Функции в JScript, как и в других алгоритмических языках, позволяют объединить несколько операций под одним именем. В случае необходимости функция может быть вызвана из любого места сценария.

В сценариях JScript поддерживаются два вида функций: встроенные функции и функции пользователя, которые вы пишете сами.

Встроенные функции

В табл. П1.11 описаны встроенные функции языка JScript, которые в сценарии можно вызывать в любом месте без предварительного описания.

Таблица П1.11. *Встроенные функции*

Функция	Описание
`escape(`*`charstring`*`)`	Кодирование строки *`charstring`* с применением URL-кодировки. При этом все специальные неотображаемые символы преобразуются к виду %xx, где xx — шестнадцатеричный код символа
`eval(codes`*`tring`*`)`	Интерпретация и выполнение кода JScript, содержащегося в строке *`codestring`*. Эта функция позволяет динамически создавать текст сценария
`isFinite(`*`number`*`)`	Возвращает `true`, если параметр *`number`* является корректным числом. В противном случае возвращает `false`
`isNaN(`*`numvalue`*`)`	Возвращает `true`, если параметр *`numvalue`* имеет специальное значение NaN (см. *описание числового типа*). В противном случае возвращает `false`. Эту функцию можно применять для оценки значений; возвращаемых функциями преобразования типов `parseInt` и `parseFloat`
`parseFloat(`*`numstring`*`)`	Преобразовывает строку *`numstring`* в число с плавающей точкой
	Преобразовывает строку *`numstring`* в целое число. Целочисленный параметр *`radix`* может принимать значения от 2 до 36, которые указывают основание

parseInt(*numstring, [radix]*)	счисления для числа, содержащегося в numstring. Если *radix* не указан, то строки с префиксом '0x' задают шестнадцатеричные числа, а строки с префиксом '0' — восьмеричные. Все остальные строки определяют десятичные числа
unescape(*charstring*)	Выполняет действие, противоположное действию функции escape, т.е. перекодирует строку *charstring* из URL-кодировки в текстовую строку

Функции пользователя

Функции, определяемые пользователем, могут находиться в любом месте сценария и иметь произвольное число параметров (аргументов). Общий вид определения функции имеет вид:

```
function Имя_функции([параметр1] [, параметр2] [... , параметрN]) {
 …
 Тело функции
 …
 [return значение;]
}
```

Ключевое слово return позволяет функции вернуть значение любого допустимого типа. Например, приведенная ниже функция MyFunction возвращает true, если оба ее аргумента меньше 10:

```
function MyFunction(x,y) {
 if ((x<10) && (y<10)) return true
 else return false;
}
```

Встроенные объекты (классы)

Как уже отмечалось в самом начале раздела, JScript является объектно-ориентированным языком, поэтому математические вычисления, работа со строками, датой и временем, а также такими структурами, как массивы и коллекции, осуществляются с помощью соответствующих встроенных объектов. В табл. П1.12 описаны некоторые объекты, которые могут быть полезны при создании сценариев с помощью WSH.

Таблица П1.12. *Некоторые встроенные объекты JScript*

Объект	Описание
`Array`	Создание и работа с массивами данных произвольного типа
`Date`	Работа с данными, содержащими дату или время
`Enumerator`	Работа с коллекциями данных произвольного типа
`Math`	Выполнение математических вычислений
`String`	Работа с текстовыми строками

Для того чтобы в сценарии использовать встроенный объект, необходимо создать переменную, с помощью которой можно будет получить доступ к свойствам и методам этого объекта. Для создания большинства переменных такого вида применяется оператор `new` и специальная функция — конструктор нужного объекта. Название конструктора всегда совпадает с названием соответствующего встроенного объекта. Приведем пример создания объектов `Date` и `Array`!

```
var d;
d = new Date();
var a;
a = new Array(10);
```

Отметим, что объекты string можно создавать, просто записывая в кавычках значение строки:

```
var s;
s = "Привет!";
```

Опишем объекты, приведенные в табл. П1.12, более подробно.

Объект *Array*

Новый объект встроенного класса `Array` можно создать с помощью оператора `new` следующими способами:

□ `new Array()` — создание массива нулевой длины;

□ `new Array(N)` — создание массива длины *N*;

□ `new Array(a0, a1, ..., aN)` — создание массива длины *N+1* с элементами *a0*, *a1*, ..., *aN*.

Например:

```
var A1, A2, A3;
A1 = new Array();
A2 = new Array(3);
A3 = new Array(0, "Строка", 2.5);
```

Нумерация элементов в массивах всегда начинается с нуля. После того как массив создан и проинициализирован, обращаться к его элементам можно с помощью обычного оператора индексации [], например:

```
A3[1] = A3[0] + A3[2];
```

Длину массива, т.е. число содержащихся в нем элементов, можно узнать с помощью свойства `length` объекта `Array`. Для того чтобы динамически изменить длину массива (уменьшить или увеличить), достаточно просто записать соответствующее значение в свойство length:

```
var A;
A = new Array(1,2,3,4,5); // Длина массива A равна 5
A.length = 3; // Теперь длина массива A равна 3
```

Некоторые наиболее часто используемые методы встроенного объекта `Array` описаны в табл. П1.13.

Таблица П1.13. *Методы объекта* `Array`

Метод	Описание
a1.concat(a2)	Возвращает новый массив, являющийся результатом объединения (склеивания) двух массивов: *a1* (его элементы идут первыми) и *a2* (его элементы идут после элементов массива *a1*)
`join(`*separator*`)`	Возвращает строку, содержащую все идущие друг за другом элементы массива, разделенные символом, указанным в параметре *separator*
`reverse()`	Располагает элементы массива в обратном порядке (первый меняется местами с последним, второй — с предпоследним и т.д.). Новый массив при этом не создается
	Возвращает часть массива, начиная с элемента с индексом *start*, заканчивая элементом с индексом *end*. Если в качестве *end* указано отрицательное

`slice(start, [end])`	число, то оно задает смещение от конца массива. Если параметр *end* не указан, то берутся все элементы массива, начиная с элемента с индексом *start*
`sort([sortfunction])`	Возвращает массив с отсортированными элементами. Параметр *sortfunction* определяет имя функции, используемой для сортировки; если этот параметр опущен, то сортировка производится в порядке увеличения ASCII-кодов элементов массива

Пример использования методов объекта Array приведен в листинге П1.4.

Листинг П1.4. Пример использования методов объекта `Array`

```
var A1, A2, A3;
A1 = new Array(2);
A2 = new Array(2,3,4,5);
A1[0] = 0;
A1[1] = 1;
A3 = A2.concat(A1); // A3=(2,3,4,5,0,1)
A3.sort(); // A3=(0,1,2,3,4,5)
```

Объект *Date*

Для создания нового объекта встроенного класса `Date` используется один из трех конструкторов.

Конструктор первого вида позволяет создать объект, в котором хранится информация о текущих дате и времени:

```
var d;
d = new Date();
```

Здесь время задается по Гринвичу, т.е. с использованием времени (UCT — Universal Coordinated Time).

Конструктор второго вида имеет единственный параметр:

```
var d;
d = new Date(nMilliseconds);
```

Параметр `nMilliseconds` задает дату в миллисекундах, считая от 1 января 1970 года.

Конструктор третьего вида предназначен для раздельного задания компонентов даты и имеет следующий вид:

```
var d;
d = new Date(year, month, date [, hours [, min [, sec [, ms]]]]);
```

Значения параметров последнего конструктора приведены в табл. П1.14.

Таблица П1.14. *Параметры конструктора* `Date`

Параметр	**Описание**
`year`	Год в четырехзначном формате, например 1998 (но не 98)
`month`	Номер месяца от 0 (январь) до 11 (декабрь)
`date`	Календарная дата в диапазоне от 1 до 31
`hours`	Час дня в диапазоне от 0 до 23
`min`	Минуты в диапазоне от 0 до 59
`sec`	Секунды в диапазоне от 0 до 59
`ms`	Миллисекунды в диапазоне от 0 до 999

Наиболее часто используемые методы объекта `Date` описаны в табл. П1.15.

Таблица П1.15. *Некоторые методы объекта* `Date`

Метод	**Описание**
`getDate()`	Возвращает календарную дату в диапазоне от 1 до 31
`getDay()`	Возвращает номер дня недели (0 для воскресенья, 1 — для понедельника и т.д.)
`getFullYear()`	Возвращает четырехзначный номер года
`getHours()`	Возвращает число часов (отсчет идет с полуночи)
`getMilliseconds()`	Возвращает число миллисекунд
`getMinutes()`	Возвращает число минут (отсчет идет с начала часа)
`getMonth()`	Возвращает число месяцев (отсчет идет с января)
`getSeconds()`	Возвращает число секунд (отсчет идет с начала минуты)
`getTime()`	Определение времени для объекта Date. Возвращает количество миллисекунд, прошедших с 1 января 1970 года

`getTimezoneOffset()`	Возвращает смещение локального времени относительно времени по Гринвичу (в миллисекундах)
`parse(`*`dateVal`*`)`	Возвращает число миллисекунд, прошедших с полуночи 1 января 1970 года по время, заданное параметром *`dateVal`*. Для вызова метода `parse` необязательно создавать объект класса `Date`, достаточно просто сослаться на имя этого класса: `n = Date.parse("10 May 2001 13:00:00");` Параметр *`dateVal`* может задаваться в нескольких форматах *(подробнее см документацию по языку JScript)*
`setDate(`*`date`*`)`	Устанавливает календарную дату. Параметр `date` может принимать любые положительные или отрицательные значения. Если значение *`date`* больше, чем количество дней в месяце, который хранится в объекте `Date`, или *`date`* является отрицательным числом, то календарная дата устанавливается в число, равное разности параметра *`date`* и числа дней в этом месяце
`setFullYear(`*`year`*`)`	Устанавливает номер года, заданный параметром *`year`*
`setHours(`*`hours`*`)`	Устанавливает количество часов, заданное параметром *`hours`*. Параметр *`hours`* может принимать любые положительные или отрицательные значения (при необходимости происходит соответствующее изменение даты, записанной в объекте класса `Date`)
`setMilliseconds(`*`ms`*`)`	Устанавливает количество миллисекунд, заданное параметром *`ms`*. Параметр *`ms`* может принимать любые положительные или отрицательные значения (при необходимости происходит соответствующее изменение даты, записанной в объекте класса `Date`)
`setMinutes(`*`min`*`)`	Устанавливает количество минут, заданное параметром *`min`*. Параметр *`min`* может принимать любые положительные или отрицательные значения (при необходимости происходит соответствующее изменение даты, записанной в объекте класса `Date`)
	Устанавливает номер месяца, прошедшего с начала

`setMonth(`*`mon`*`)`	года. Параметр *`mon`* может принимать любые положительные или отрицательные значения (при необходимости происходит соответствующее изменение даты, записанной в объекте класса `Date`)
`setSeconds(`*`sec`*`)`	Устанавливает количество секунд, заданное параметром *`sec`*. Параметр *`sec`* может принимать любые положительные или отрицательные значения (при необходимости происходит соответствующее изменение даты, записанной в объекте класса `Date`)
`setTime(`*`ms`*`)`	Устанавливает дату, соответствующую количеству миллисекунд (параметр *`ms`*), прошедших с 1 января 1970 года
`toGMTString()`	Преобразует дату в строку и возвращает результат в стандартном формате времени по Гринвичу (Greenwich Mean Time, GMT)
`ToLocaleString()`	Преобразует дату в строку и возвращает результат в формате локального времени
`ToUTCString()`	Преобразует дату в строку и возвращает результат в формате UTC
`UTC(`*`year, month, date[, hours[, min[, see [,ms]]]]`*`)`	Преобразует дату, заданную параметрами метода, в количество миллисекунд, прошедшее с полуночи 1 января 1970 года. При использовании этого метода, как и метода `parse`, объект класса `Date` создавать необязательно: `n = Date.UTC(year, month, date);`

Пример использования методов объекта `Date` приведен в листинге П1.5.

Листинг П1.5. Пример использования методов объекта `Date`

```
    var d;
    var s = "";
    d = new Date();
    s = "Дата: " + d.getDate() + "." + d.getMonth() + "." +
d.getYear(); s += "\n";
    s += "Время: " + d.getHours() + ":" + d.getMinutes() + ":" +
d.getSeconds();
```

После выполнения этих строк в переменной `s` будут записаны текущие

дата и время.

Объект *Enumerator*

С помощью объекта `Enumerator` можно получить доступ к любому элементу *коллекции* (в VBScript для этого служит цикл `For…Each`). Коллекцией в языке JScript называется множество элементов, которое отличается от массива тем, что к элементам коллекции нельзя получить прямой доступ с помощью индексов — можно только перемешать указатель текущего элемента на самый первый или следующий относительно текущего элемент.

Для создания нового объекта встроенного класса `Enumerator` используется конструктор следующего вида:

```
var e;
e = new Enumerator(collection);
```

Здесь параметр *`collection`* указывает на коллекцию, для доступа к элементам которой и создается объект класса `Enumerator`. Сами коллекции обычно являются свойствами других объектов.

Методы объекта `Enumerator` представлены в табл. П1.16 (свойств у этого объекта нет).

Таблица П1.16. *Методы объекта* `Enumerator`

Метод	Описание
`atEnd()`	Возвращает `true`, если указатель текущего элемента находится на элементе, следующем за последним экземпляром коллекции, либо коллекция пуста, либо текущий элемент не определен. В противном случае возвращается `false`
`item()`	Возвращает значение текущего элемента коллекции. Если коллекция пуста или текущий элемент не определен, возвращается неопределенное значение `undefined`
`moveFirst()`	Перемещает указатель на первый элемент коллекции. Если в коллекции нет элементов, текущий элемент принимает неопределенное значение `undefined`
`moveNext()`	Перемещает указатель на следующий элемент коллекции. Если перед применением этого метода указатель находился на последнем элементе коллекции, либо коллекция пуста, текущий элемент принимает неопределенное значение

```
undefined
```

Пример, поясняющий схему применения объекта `Enumerator`, приведен в листинге П1.6.

Листинг П1.6. Пример использования `Enumerator`

```
//Объявляем переменные
var FSO, Folder, Files, s;
//Создаем объект FileSystemObject
FSOWScript.CreateObject("Scripting.FileSystemObject");
//Создаем объект Folder для корневого каталога диска C:
Folder=FSO.GetFolder("C:\\")?
//Создаем коллекцию файлов каталога "Мои документы"
Files=new Enumerator(Folder.Files);
//Цикл по всем файлам
for (; !Files.atEnd(); Files.moveNext())
 //Добавляем строку с именем файла
 s+=Files.item().Name+"\n";
//Выводим полученные строки на экран
WScript.Echo(s);
```

Объект *Math*

Встроенный класс `Math` применяется для математических вычислений и содержит основные математические константы и функции.

Замечание

Объект `Math` создается сервером сценариев автоматически и не может быть создан при помощи оператора `new`, как другие встроенные объекты. Все методы и свойства этого объекта доступны из сценария без какого-либо предварительного объявления.

Свойства объекта `Math` (все они являются математическими константами) описаны в табл. П1.17.

Таблица П1.17. *Свойства объекта `Math`*

Свойство	Описание
`E`	Константа e. Приблизительное ее значение равно 2,718
`LN2`	Натуральный логарифм числа 2 (приблизительно 0,693)
`LN10`	Натуральный логарифм числа 10 (приблизительно 2,302)
`LOG2E`	Логарифм числа e по основанию 2 (примерно 1,442)
`LOG10E`	Логарифм числа e по основанию 10 (примерно 0,434)
`PI`	Число π, т.е. константа с приблизительным значением, равным 3,142
`SQRT1_2`	Корень квадратный из ½ (примерно 0,707)
`SQRT2`	Корень квадратный из 2 (примерно 1,414)

Методы объекта `Math` (они являются стандартными математическими функциями) приведены в табл. П1.18.

Таблица П1.18. *Методы объекта* `Math`

Метод	Описание
`abs(`*x*`)`	Возвращает абсолютное значение числа *x*
`acos(`*x*`)`	Возвращает арккосинус числа *x*
`asin(`*x*`)`	Возвращает арксинус числа *x*
`atan(`*x*`)`	Возвращает арктангенс числа *x*
`atan2(`*y*`,`*x*`)`	Вычисляет угол в радианах от оси `0x` до точки (*y*, *x*). Возвращаемое значение лежит в диапазоне от -π до π
`ceil(`*x*`)`	Возвращает наименьшее целое значение, большее или равное аргументу *x*
`cos(`*x*`)`	Возвращает косинус числа *x*
`exp(`*x*`)`	Экспоненциальная функция, возвращает число e, возведенное в степень *x*
`floor(`*x*`)`	Возвращает наибольшее целое значение, меньшее или равное аргументу *x*
`log(`*x*`)`	Возвращает натуральный логарифм числа *x*
`max(`*x1*`, `*x2*`)`	Возвращает наибольшее из двух значений *x1* и *x2*
`min(`*x2*`, `*x2*`)`	Возвращает наименьшее из двух значений *x1* и *x2*
`pow(`*y*`,`*x*`)`	Возводит число *y* в степень *x* и возвращает полученный

	результат
`random()`	Возвращает случайное число в интервале от 0 до 1
`round(`*x*`)`	Выполняет округление значения аргумента *x* до ближайшего целого. Если десятичная часть числа равна 0,5 или больше этого значения, то округление выполняется в бóльшую сторону, иначе — в меньшую
`sin(`*x*`)`	Возвращает синус числа *x*
`sqrt(`*x*`)`	Вычисляет квадратный корень из числа *x* и возвращает полученное значение
`tan(`*x*`)`	Возвращает тангенс числа *x*

Приведем пример использования свойств и методов встроенного объекта `Math`:

```
var x,y,z;
x = 12;
y = Math.sqrt(x);
z = 2*Math.PI * Math.pow(y, x);
```

Объект *String*

Встроенный объект `String` предназначен для выполнения различных операций над текстовыми строками. Обычно объекты класса `String` создаются просто с помощью записи в переменную текстового литерала:

```
var s1, s2;
s1 = "Это строка";
s2 = "Это тоже строка";
```

Также можно создавать такие объекты с помощью оператора `new`:

```
var s1, s2;
s1 = new String("Это строка");
s2 = new String("Это тоже строка");
```

Объект `String` имеет свойство `length`, в котором хранится длина строки.

Некоторые методы объекта `String`, не связанные с тегами HTML, приведены в табл. П1.19.

Таблица П1.19. *Некоторые методы объекта* `String`

Метод	Описание
	Возвращает символ с индексом *`index`* из строки. Нумерация символов в строке начинается с

`charAt(`*`index`*`)`	нуля, поэтому допустимыми значениями параметра *`index`* являются числа из диапазона от нуля до числа, на единицу меньшего длины строки
`charCodeAt(`*`index`*`)`	Выбирает символ с индексом *`index`* из строки и возвращает этот символ в кодировке Unicode
`s1.concat(`*`s2`*`)`	Возвращает строку, являющуюся результатом конкатенации (склеивания) строк `s1` и `s2` (то же самое, что `s1+s2`)
`fromCharCode(`*`c1`*`,...,`*`cN`*`)`	Возвращает строку, состоящую из Unicode-символов с кодами *`c1`*,..., *`cN`*
`s.indexOf(`*`substr`*`, `*`startindex`*`)`	Возвращает индекс символа, с которого начинается первое вхождение подстроки *`substr`* в строку `s`. Если подстрока не найдена, возвращается -1. Параметр *`startindex`* задает номер символа, с которого следует начинать поиск. Если этот параметр не задан, то поиск производится с начала строки. Поиск производится слева направо
`s.lastIndexOf(`*`substr`*`, `*`startindex`*`)`	То же самое, что метод `indexOf`, но поиск производится справа налево, т.е. возвращается номер последнего символа, с которого начинается вхождение подстроки *`substr`* в строку `s`
`s.match(`*`rgExp`*`)`	Возвращает в виде массива результат поиска в строке `s` подстроки, задаваемой регулярным выражением *`rgExp`* *(поиск с использованием регулярных выражений описан в документации по JScript)*
`replace(`*`rgExp`*`, `*`replaceText`*`)`	Возвращает копию строки, в которой произведены необходимые замены текста. Шаблон для поиска задается регулярным выражением *`rgExp`*, строка для замены — параметром *`replaceText`*. Первоначальная строка методом `replace` не изменяется
`search(`*`rgExp`*`)`	Возвращает номер первого символа в строке, с которого начинается подстрока,

	удовлетворяющая регулярному выражению *rgExp*
slice(*start, [end]*)	Возвращает часть строки, начиная с позиции *start* и заканчивая символом, стоящим в позиции с номером *end* (или последним символом в строке, если параметр *end* опущен). Если в качестве *end* указано отрицательное число, то этот параметр задает смещение от конца массива
s.split(*str*)	Возвращает массив строк, полученных в результате разбиения строки s на подстроки. Параметр *str* задает строку или объект Regular Expression, которые определяют символ, являющийся признаком начала разбиения
substr(*start [, length]*)	Возвращает подстроку, начинающуюся с позиции *start* и имеющую длину *length*. Если в качестве *length* указано отрицательное число или ноль, то возвращается пустая строка
substring(*start, end*)	Возвращается подстрока, состоящая из символов, начинающихся с позиции *start* и заканчивающихся позицией *end*. В качестве *start* и *end* могут быть указаны строки, которые в этом случае автоматически преобразуются в числа
toLowerCase()	Возвращает строку, в которой все алфавитные символы преобразованы к нижнему регистру
toUpperCase()	Возвращает строку, в которой все алфавитные символы преобразованы к верхнему регистру

Приведем пример использования методов объекта String:

```
var s1, s2, s3;
s1 = "Первая строка";
s2 = "Вторая строка";
s1 = s1.toUpperCase();
s2 = s2.substr(0, 6);
s1 = s1.slice(7);
s3 = s2 + " " + s1;
```

После выполнения этих строк значением переменной s3 будет строка "Вторая строка".

Приложение 2
Справочник по языку VBScript

Язык VBScript (Visual Basic Script Edition) является урезанной версией языка Microsoft Visual Basic, поэтому для тех, кто программировал на Visual Basic или VBA, язык VBScript окажется очень знакомым.

Строки кода и комментарии

В отличие от JScript, для сценариев VBScript в конце строки не нужно ставить точку с запятой. В случае необходимости написания одного оператора на нескольких строках в конце этих строк нужно ставить символ подчеркивания "_":

```
s = "Символьная " & _
 "строка"
```

Комментарием в VBScript считаются все символы в строке, идущие после символа апострофа ' или ключевого слова Rem. Например:

```
Rem Этот комментарий занимает всю строку
theSum=1 'А этот — часть строки
```

Переменные

Переменные в VBScript могут быть глобальными (доступными из любого места сценария) и локальными (область действия ограничивается блоком кода, в котором они определены). Все переменные VBScript имеют стандартный тип `Variant`. Объявляются переменные обычно с помощью ключевого слова `Dim`, например:

```
Dim MyVariable
```

По умолчанию переменные в VBScript можно предварительно не объявлять; для включения режима обязательного объявления переменных нужно вставить в самую первую строку сценария выражение `Option Explicit`.

Язык VBScript является регистро-независимым, т.е. имена `MyVariable` и `myvariable` представляют одинаковые переменные. При выборе имен переменных следует придерживаться следующих правил:

□ имя переменной должно начинаться с буквы и не должно содержать символа ",".

□ имя переменной не должно превышать 255 символов.

Значения переменным в VBScript присваиваются с помощью оператора "=". Например:

```
Dim MyVariable MyVariable = "Привет!"
```

Здесь мы объявили переменную `MyVariable` и записали в нее текстовую строку. Отметим, что далее в любом месте сценария мы можем присвоить переменной `MyVariable`, скажем, числовое значение, например:

```
MyVariable = 10
```

Подтипы данных

Хотя в VBScript определен только один тип `Variant`, внутри этого типа имеется разделение на *подтипы,* описание которых приведено в табл. П2.1. Для преобразования переменных к определенному подтипу нужно использовать соответствующую функцию преобразования; такие функции также представлены в табл. П2.1.

Таблица П2.1. *Подтипы данных*

Подтип	Функция преобразования	Описание
`Empty`	—	Автоматически присваивается новым переменным, когда для них еще не определено явное значение
`Null`	—	Указывает на то, что переменная не содержит допустимых значений
`Bool`	`CBool(x)`	Используется для работы с логическими переменными, принимающим два допустимых значения: `true` или `false`
`Byte`	`CByte(x)`	Содержит целые числа в диапазоне от 0 до 255
`Integer`	`CInt(X)`	Содержит целые числа в диапазоне от –32768 до 32768
`Currency`	`CCur(x)`	Специальный числовой формат для денежных величин
`Long`	`CLng(x)`	Содержит целые числа в диапазоне от -2147483648 до 2147483647
`Single`	`CSngl(x)`	Тип чисел с плавающей точкой одинарной точности
`Double`	`CDbl(x)`	Тип чисел с плавающей точкой двойной точности
`Date/Time`	`CDate(x)`	Содержит числа, соответствующие датам и времени от 1 января 100 года до 31 декабря 9999 года
`String`	`CStr(x)`	Символьный подтип данных. Текстовые строки в VBScript — это последовательность

		символов, заключенных в двойные кавычки
`Object`	—	Ссылка на объект
`Error`	—	Тип данных, предназначенный для хранения номеров ошибок

В переменную, которая была ранее объявлена с использованием ключевого слова `Dim`, можно записать ссылку на какой-либо объект. Делается это с помощью оператора `Set`, например:

```
Dim FSO
Set FSO=CreateObject("Scripting.FileSystemObject")
```

Здесь функция `CreateObject()` возвращает экземпляр объекта `FileSystemObject`, ссылка на который заносится в переменную `FSO`.

После того как ссылка на объект станет ненужной, переменную можно освободить с помощью ключевого слова `Nothing`:

```
Set FSO=Nothing
```

Константы

Пользовательские константы в VBScript объявляются с помощью ключевого слова `Const`, например:

```
Const MyConst="Это моя константа"
```

Кроме этого, VBScript поддерживает много встроенных именованных констант (их не нужно дополнительно объявлять в сценарии), применение которых упрощает использование различных внутренних функций (например, `MsgBox()` или `InputBox()`). Имена, значения и описания внутренних констант приведены в табл. П2.2–П2.9.

Таблица П2.2. *Константы для обозначения цветов*

Имя	Значение	Описание
`vbBlack`	&h00	Черный цвет
`vbRed`	&hFF	Красный цвет
`vbGreen`	&hFF00	Зеленый цвет
`vbYellow`	&hFFF	Желтый цвет
`vbBlue`	&hFF0000	Синий цвет
`vbMagenta`	&hFF00FF	Фиолетовый цвет
`vbCyan`	&hFFFF00	Бирюзовый цвет
`vbWhite`	&hFFFFFF	Белый цвет

Таблица П2.3. *Константы для нумерации дней недели*

Имя	Значение	Описание
`vbSunday`	1	Воскресенье
`vbMonday`	2	Понедельник
`vbTuesday`	3	Вторник
`vbWednesday`	4	Среда
`vbThursday`	5	Четверг
`vbFriday`	6	Пятница
`vbSaturday`	7	Суббота

Таблица П2.4. *Константы для определения первого дня в неделе и первой недели в году*

Имя	Значение	Описание
vbUseSystemDayOfWeek	0	Использовать для определения первого дня недели региональные настройки системы
vbFirstJan1	1	Первой неделей в году считается та, в которой было 1 января
vbFirstFourDays	2	Первой неделей в году считается та, в которой было по крайней мере четыре дня нового года
vbFirstFullWeek	3	Первой неделей в году считается первая полная неделя

Таблица П2.5. *Константы для работы с датой и временем*

Имя	Значение	Описание
vbGeneralDate	0	Дата и время выводятся в формате, определяемом региональными настройками системы
vbLongDate	1	Выводить дату, используя полный формат
vbShortDate	2	Выводить дату, используя краткий формат
vbLongTime	3	Выводить время, используя полный формат
vbShortTime	4	Выводить время, используя краткий формат

Таблица П2.6. *Константы для диалоговых окон*

Имя	Значение	Описание
vbOkOnly	0	Выводится кнопка **OK**
vbOkCancel	1	Выводятся кнопки **OK** и **Отмена** (Cancel)
vbAbortRetryIgnore	2	Выводятся кнопки **Стоп** (Abort), **Повтор** (Retry) и **Пропустить** (Ignore)
vbYesNoCancel	3	Выводятся кнопки **Да** (Yes), **Нет** (No) и **Отмена** (Cancel)
vbYesNo	4	Выводятся кнопки **Да** (Yes) и **Нет** (No)
vbRetryCancel	5	Выводятся кнопки **Повтор** (Retry) и **Отмена** (Cancel)
vbCritical	16	Выводится значок **Stop Mark**
vbQuestion	32	Выводится значок **Question Mark**
vbExclamation	48	Выводится значок **Exclamation Mark**

`vbInformation`	64	Выводится значок `Information Mark`
`vbDefaultButton1`	0	По умолчанию в окне выбирается первая кнопка
`vbDefaultButton2`	256	По умолчанию в окне выбирается вторая кнопка
`vbDefaultButton3`	512	По умолчанию в окне выбирается третья кнопка
`vbDefaultButton4`	768	По умолчанию в окне выбирается четвертая кнопка
`vbApplicationModal`	0	Диалоговое окно выводится в модальном режиме
`vbSystemModal`	4096	Диалоговое окно выводится в модальном режиме и располагается сверху всех запускаемых приложений

Таблица П2.7. *Результаты нажатия кнопок в диалоговых окнах*

Имя	Значение	Описание
`vbOk`	1	Нажата кнопка **OK**
`vbCancel`	2	Нажата кнопка **Отмена** (Cancel)
`vbAbort`	3	Нажата кнопка **Стоп** (Abort)
`vbRetry`	4	Нажата кнопка **Повтор** (Retry)
`vbIgnore`	5	Нажата кнопка **Пропустить** (Ignore)
`vbYes`	6	Нажата кнопка **Да** (Yes)
`vbNo`	7	Нажата кнопка **Нет** (No)

Таблица П2.8. *Константы для обозначения подтипов данных*

Имя	Значение	Описание
`vbEmpty`	0	Переменная не инициализирована
`vbNull`	1	Переменная не содержит корректных данных
`vbInteger`	2	Переменная имеет подтип `Integer`
`vbLong`	3	Переменная имеет подтип `Long`
`vbSingle`	4	Переменная имеет подтип `Single`
`vbDouble`	5	Переменная имеет подтип `Double`
`vbCurrency`	6	Переменная имеет подтип `Currency`

vbDate	7	Переменная имеет подтип Date
vbString	8	Переменная имеет подтип String
vbObject	9	Переменная имеет подтип Object
vbError	10	Переменная имеет подтип Error
vbBoolean	11	Переменная имеет подтип Boolean
vbVariant	12	Переменная имеет подтип Variant (только для массивов переменных типа Variant)
vbDataObject	13	Объект доступа к данным
vbDecimal	14	Переменная имеет подтип Decimal
vbByte	17	Переменная имеет подтип Byte
vbArray	8192	Переменная является массивом

Таблица П2.9. *Прочие константы*

Имя	Значение	Описание
vbCr	Chr(13)	Возврат каретки
vbCrLf	Chr(13) & Chr(10)	Возврат каретки и перевод строки
vbFormFeed	Chr(12)	Перевод страницы
vbLf	Chr(10)	Перевод строки
vbNullChar	Chr(0)	Символ с нулевым кодом
vbNullString	Нулевая строка	Нулевая строка
vbTab	Chr(9)	Символ табуляции
vbVerticalTab	Chr(11)	Символ вертикальной табуляции
vbUseDefault	-2	Использовать значения по умолчанию из региональных настроек системы
vbTrue	-1	Логическое значение "истина"
vbFalse	0	Логическое значение "ложь"
vbObjectError	-2147221504	Определяет минимальное значение для номеров ошибок, задаваемых пользователем

Массивы

Массивы в VBScript могут быть двух видов: статические (фиксированной длины) и динамические (переменной длины). Объявляются массивы, как и обычные переменные, с помощью ключевого слова `Dim`.

Для объявления статического массива нужно после его названия указать в круглых скобках наибольшее значение, которое может принимать индекс элемента в этом массиве, например:

```
Dim MyArr(10)
```

В языке VBScript (в отличие, например, от VBA) нумерация в массивах всегда начинается с нуля, поэтому объявленный выше массив `MyArr` будет содержать 11 элементов, обращаться к которым нужно следующим образом:

```
MyArr(0)="Это первый элемент"
MyArr(1)="Это второй элемент"
MyVar=MyArr(0)
```

Можно объявить двумерный массив, указав максимальные значения индексов для строк и столбцов соответственно, например:

```
Dim MyArr(5, 10) 'Массив из 6 строк и одиннадцати столбцов
```

При объявлении динамического массива его размеры в круглых скобках не указываются:

```
Dim MyArr()
```

Для использования динамического массива в сценарии применяется оператор `ReDim`, который определяет конкретную длину массива, например:

```
ReDim MyArray(10)
```

После этого к элементам динамического массива можно обращаться так же, как и к элементам обычного:

```
MyArr(0)="Это первый элемент"
MyArr(1)="Это второй элемент"
```

Отметим, что размеры динамического массива можно менять неоднократно (с помощью того же `ReDim`). Для сохранения при этом содержимого массива следует в операторе `ReDim` использовать ключевое слово `Preserve`, например:

```
ReDim Preserve MyArray(20)
```

Операторы

В VBScript поддерживаются операторы нескольких типов, которые описаны ниже.

Арифметические операторы

Арифметические операторы языка VBScript представлены в табл. П2.10.

Таблица П2.10. *Арифметические операторы*

Оператор	Описание
- (унарный оператор)	Изменение знака аргумента на противоположный
- (бинарный оператор)	Вычитание двух чисел
`+`	Сложение двух чисел
`*`	Умножение двух чисел
`/`	Деление двух чисел
`\`	Целочисленное деление двух чисел
`Mod`	Вычисление остатка от деления двух чисел
`^`	Оператор возведения в степень

Операторы отношения и логические операторы

Операторы отношения используются для сравнения значений двух переменных. Эти операторы, описанные в табл. П2.11, могут возвращать только логические значения `true` или `false`.

Таблица П2.11. *Операторы отношения*

Оператор	Условие, при котором возвращается `true`
`>`	Левый операнд больше правого
`>=`	Левый операнд больше или равен правому
`<`	Левый операнд меньше правого
`<=`	Левый операнд меньше или равен правому
`=`	Левый операнд равен правому

<>	Левый операнд не равен правому

Также внутри условных операторов могут применяться логические операторы (табл. П2.12).

Таблица П2.12. *Логические операторы*

Оператор	Описание
Not	Оператор отрицания. Возвращает true, если операнд равен false. В противном случае возвращает false
Or	Оператор отношения "ИЛИ". Возвращает true, если один из операндов равен true. В противном случае возвращает false
Xor	Оператор отношения "ИСКЛЮЧАЮЩЕЕ ИЛИ". Возвращает true, если один из операндов равен true, а другой равен false. В противном случае возвращает false
And	Оператор отношения "И". Возвращает true, если оба операнда равны true. В противном случае возвращает false

Условные операторы

В VBScript поддерживается условный оператор If…Then…Else. Общий вид этого оператора:

```
If условие_1 Then
 выражение_1
[ElseIf условие_2 Then
 выражение_2]
…
[Else
 выражение_3]
End If
```

При выполнении оператора If…Then…Else оценивается логическое условие (*условие_1*), стоящее после ключевого слова If. Если в результате оценки условия получилось значение true, то выполняется*выражение_1* и происходит выход из оператора. В противном случае начинают по очереди проверяться условия, стоящие после ключевых слов ElseIf; если одно из этих условий истинно, то выполняется соответствующее выражение, после чего управление передается следующему после End If оператору. Если ни одно из проверяемых условий не является истинным, выполняется выражение, стоящее после ключевого слова Else. Пример:

```
If (theCount > 1) Then
```

```
 theMoments = "До взрыва осталось " & theCount & " сек!"
Else
 theMoments = "Осталась секунда!"
End If
```

Другим оператором, позволяющим производить выбор из нескольких вариантов, является `Select Case`. Синтаксис этого оператора:

```
Select Case выражение
Case значение_1
 выражение
[Case значение_2
 выражение_2]
…
[Case Else
 выражение_3]
End Select
```

Здесь сначала вычисляется значение*выражения*, которое затем по очереди сравнивается со значениями, стоящими после ключевых слов `Case`. В случае совпадения выполняются операторы в соответствующем блоке `Case`. Если же ни одно из имеющихся значений не совпадает со значением*выражения*, то выполняются операторы, стоящие после слова `Case Else`. Пример использования оператора `Select Case`:

```
Select Case MyVar
Case vbRed
 Color = "Красный"
Case vbGreen
 Color = "Зеленый"
Case vbBlue
 Color = "Синий"
Case Else
 Color = "Цвет непонятен"
End Select
```

Операторы циклов

В VBScript поддерживаются несколько типов циклов: цикл `For…Next`, цикл `Do…Loop`, цикл `While…Wend`, цикл `For Each…Next`. Рассмотрим каждый из них подробнее.

Цикл *For…Next*

В общем случае оператор цикла `For…Next` записывается следующим образом:

```
For counter=start To end [Step step]
 тело цикла
 [Exit For]
 тело цикла
Next
```

Параметр *counter* здесь является счетчиком цикла; *start* — начальное значение этого счетчика; *end* — конечное значение; *step* — шаг приращения счетчика. Если ключевое слово Step не указано, то шаг приращения берется равным единице. Выход из цикла For…Next происходит, когда значение счетчика *counter* становится больше, чем значение параметра *end*. Выражение Exit For используется для безусловного выхода из цикла.

Пример использования цикла for приведен в листинге П2.1.

Листинг П2.1. Пример использования цикла For…Next

```
Dim howFar 'Верхний предел для счетчика цикла
Dim sum(10) 'Массив из 11 элементов, индексы от 0 до 10
Dim icount, theSum
howFar = 10
theSum = 0
sum(0) = 0
'Цикл выполнится 11 раз
For icount = 0 To howFar
 theSum=theSum+icount
 sum (icount) = theSum
Next
```

Цикл *For Each…Next*

Оператор цикла For Each…Next предназначен для перебора всех элементов массива или коллекции:

```
For Each element In group
тело цикла
 [Exit For]
 тело цикла
Next [element]
```

Здесь параметр *element* является переменной, в которую будет записываться значение текущего элемента массива или коллекции при итерациях; *group* — имя массива или коллекции объектов.

Замечание

Напомним, что в JScript для перебора всех элементов коллекции необходимо использовать вспомогательный объект `Enumerator`.

С помощью оператора `Exit For` можно осуществить немедленный выход из цикла.

Пример использования цикла `For Each…Next` приведен в листинге П2.2.

Листинг П2.2. Пример использования цикла `For Each…Next`

```
    'Объявляем переменные
    Dim FSO, Folder, Files, File, s
    s = "Список файлов" & vbCrLf
    'Создаем объект FileSystemObject
    Set FSO = CreateObject("Scripting.FileSystemObject")
    ' Создаем объект Folder для корневого каталога диска C:
    Set Folder = FSO.GetFolder("C:\")
    'Создаем коллекцию Files всех файлов в корневом каталоге диска
C:
    Set Files = Folder.Files
    'Перебираем все элементы коллекции Files
    For Each File In Files
     'Выделяем имя файла для текущего элемента File коллекции
     s = s & File.Name & vbCrLf
    Next
    'Выводим сформированную строку на экран
    WScript.Echo s
```

Цикл *While…Wend*

Цикл `While…Wend` записывается в следующем виде:

```
While условие
 тело цикла
Wend
```

Таким образом, в цикле `While…Wend` условие выполнения тела цикла проверяется перед началом очередной итерации. Если *условие* равно `true`, то тело цикла выполняется, в противном случае цикл завершается.

Пример использования цикла `While…Wend` приведен в листинге П2.3.

Листинг П2.3. Пример использования цикла `While…Wend`

```
Dim theMoments, theCount
theMoments = ""
theCount = 42 'Начальное значение счетчика цикла
While (theCount >= 1)
 If (theCount > 1) Then
  theMoments = "До взрыва осталось " & theCount & " сек!"
 Else
  theMoments = "Осталась секунда!"
 End If
 theCount = theCount - 1 'Уменьшаем значение счетчика
Wend
theMoments = "ВЗРЫВ!"
```

Цикл *Do…Loop*

Этот цикл может применяться в двух видах (с предусловием, которое проверяется до начала очередной итерации, и с пост-условием, которое проверяется после окончания итерации):

```
Do [While | Until] условие
 тело цикла
 [Exit Do]
 тело цикла
Loop
```

или

```
Do
 тело цикла
 [Exit Do]
 тело цикла
Loop [While | Until] условие
```

Если в цикле используется ключевое слово `While`, то итерации продолжаются до тех пор, пока*условие* равно `true`; если же применяется `Until`, то как только значением условия станет `true`, произойдет выход из цикла.

Оператор `Exit Do` позволяет выйти из цикла до завершения его итераций.

Пример использования цикла `Do…Loop` приведен в листинге П2.4.

Листинг П2.4. Пример использования цикла Do…Loop

```
Dim howFar 'Верхний предел для счетчика цикла
Dim sum(10) 'Массив из 11 элементов, индексы от 0 до 10
Dim icount, theSum
howFar = 10
theSum = 0
sum(0) = 0
'Цикл выполнится 11 раз
Do
 theSum = theSum+icount
 sum(icount) = theSum
 icount=icount+l
Loop While (icount < howFar)
```

Прочие операторы

Рассмотрим еще несколько часто применяемых операторов (табл. П2.13).

Таблица П2.13. *Прочие операторы*

Оператор	Описание
.	Точка. Применяется для доступа к свойству объекта или для вызова его метода
()	Скобки. Применяются либо для изменения порядка вычисления выражений, либо для передачи параметров функциям, либо для индексирования массива
&	Оператор конкатенации (склеивание между собой) символьных строк
With…End With	Позволяет обращаться к свойствам объекта без написания имени этого объекта

Обработка исключительных ситуаций

Режим обработки исключительных ситуаций в VBScript включается с помощью оператора `On Error Resume Next`. Если после этого при исполнении какою-либо оператора в сценарии произойдет ошибка времени выполнения, то управление передастся к следующему оператору в тексте.

Для анализа ошибок используется специальный объект `Err`, который

содержит два свойства: Number — числовой код возникшей ошибки и Description — краткое описание этой ошибки.

В качестве примера приведем часть сценария, в которой происходит обработка исключительных ситуаций при подключении сетевого диска:

```
    On Error Resume Next ' Включаем обработку ошибок времени выполнения
    ' Подключаем сетевой диск
    WshNetwork.MapNetworkDrive Drive, NetPath
    If Err.Numbero<>0 Then
     Mess="Ошибка при подключении диска " & Drive & " к " & NetPath &_
      "Код ошибки: " & e.number & "Описание: " & e.description
       WshShell.Popup Mess, 0, "Подключение сетевого диска", vbCritical
    Else
     ' Все в порядке
     Mess = "Диск " & Drive & " успешно подключен к " & NetPath
      WshShell.Popup Mess, 0, "Подключение сетевого диска", vbInformation
    End If
```

Для отмены режима обработки исключительных ситуаций нужно выполнить оператор On Error Goto 0.

Процедуры и функции

VBScript поддерживаются два вида подпрограмм: встроенные функции и функции или процедуры пользователя.

Математические функции

Имеющиеся в VBScript функции, предназначенные для математических вычислений, описаны в табл. П2.14.

Таблица П2.14. *Математические функции*

Функция	Описание
`Abs(x)`	Возвращает абсолютное значение числа *x*
`Atn(x)`	Возвращает арктангенс числа *x*
`Cos(x)`	Возвращает косинус числа *x*
`Exp(x)`	Экспоненциальная функция, возвращает число e, возведенное в степень *x*
`Int(x)`	Возвращает целую часть числа *x*
`Log(x)`	Возвращает натуральный логарифм числа *x*
`Rnd[(x)]`	Возвращает случайное число от 0 до 1
`Round(x[, nvmdecimal])`	Возвращает результат округления числа *x* с точностью до *numdecimal* знаков после запятой
`Sgn(x)`	Знаковая функция числа *x*
`Sin(x)`	Возвращает синус числа *x*
`Sqr(x)`	Вычисляет квадратный корень из числа *x* и возвращает полученное значение
`Tan(x)`	Возвращает тангенс числа *x*

Символьные функции

Наиболее часто используемые функции, с помощью которых можно производить различные операции над символьными строками, описаны в табл. П2.15.

Таблица П2.15. *Символьные функции*

Функция	**Описание**
`Asc(str)`	Возвращает ASCII-код первого символа в строке *str*
`Chr(code)`	Возвращает символ с ASCII-кодом *code*
`InStr([start,] str1, str2[, compare])`	Возвращает индекс символа, с которого начинается первое вхождение подстроки *str2* в строку *str1*. Параметр *start* задает номер символа, с которого следует начинать поиск. Если этот параметр не задан, то поиск производится с начала строки. Поиск производится слева направо. Параметр *compare* задает режим сравнения при обработке строк (0 — двоичное, сравнение, 1 — текстовое сравнение)
`InStrRev(str1, str2[, start[, compare]])`	То же самое, что функция *InStr*, но поиск производится справа налево, т.е. возвращается номер последнего символа, с которого начинается вхождение подстроки *str2* в строку *str1*
`Join(list[,delim])`	Возвращает строку, полученную в результате конкатенации подстрок, содержащихся в массиве *list*. Параметр *delim* задает символ, разделяющий подстроки (по умолчанию таким символом является пробел)
`LCase(str)`	Возвращает строку, в которой все алфавитные символы преобразованы к нижнему регистру
`Left(str, len)`	Возвращает *len* символов с начала строки *str*
`Len(str)`	Возвращает число символов в строке *str*
`LTrim(str), RTrim(str), Trim(str)`	Удаляет из строки `str` начальные, конечные или и те и другие пробелы соответственно
`Mid(str, start[, len])`	Возвращает из строки `str` подстроку, которая начинается с позиции *start* и имеет длину *len*. Если параметр *len* не указан, то возвращаются все символы, начиная с позиции *start* до конца строки `str`
`Replace(expr, find, replacewith[,`	Возвращает строку, которая получается из строки *expr* путем замен входящих в нее подстрок *find* на подстроки *replacewith*. Параметр *count* определяет число подстрок, которые будут обработаны таким

start[, count[, compare]]])	образом (по умолчанию производятся все возможные замены). Параметр *compare* задает режим сравнения при работе со строками (0 — двоичное сравнение, 1 — текстовое сравнение)
Right(*str, len*)	Возвращает *len* символов с конца строки *str*
Space(*x*)	Возвращает строку, состоящую из *x* пробелов
Split(*Expr[, delim[, count[, compare]]]*)	Возвращает массив строк, полученных в результате разбиения строки *Expr* на подстроки. Параметр *delim* задает символ, разделяющий подстроки (по умолчанию таким символом является пробел). Параметр *count* определяет число подстрок, которые будут обработаны таким образом (по умолчанию в массив записываются все подстроки). Параметр *compare* задает режим сравнения при работе со строками (0 — двоичное сравнение, 1 — текстовое сравнение)
StrComp(*str1, str2[, compare]*)	Возвращает число — результат сравнения строк *str1* и *str2*. Если *str1<str2*, то возвращается -1; если *str1=str2*, то возвращается 0; если *str1>str2*, то возвращается 1. Параметр *compare* задает режим сравнения при работе со строками (0 — двоичное сравнение, 1 — текстовое сравнение)
String(*number, char*)	Возвращает строку, состоящую из *number* символов *char*
UCase(*str*)	Возвращает строку, в которой все алфавитные символы преобразованы к верхнему регистру

Для работы с датой и временем в VBScript имеется большой набор функций, основные из которых приведены в табл. П2.16.

Таблица П2.16. *Функции для работы с датой и временем*

Функция	**Описание**
Date	Возвращает текущую системную дату
DateAdd(*interval, number, date*)	Возвращает дату, отстоящую от даты *date* на *number* интервалов, заданных параметром *interval*, который может принимать следующие значения: "yyyy" — год, "q" — квартал, "m" —

	месяц, "y" — день года, "d" — день, "w" — неделя, "ww" — неделя года, "h"— час, "m" — минута, "s" — секунда
DateDiff(*interval, date1, date2[, firstdayofweek [, firstweekofyear]]*)	Возвращает разницу в интервалах *interval* (возможные значения этого параметра те же, что и в функции DateAdd) между датами *date1* и *date2*. Параметр *firstdayofweek* — это константа, показывающая, какой из дней недели следует считать первым (см. табл. П2.3, П2.4). Параметр *firstweekofyear* — это константа, показывающая, какую неделю следует считать первой в году (см. табл. П2.4)
DatePart(*interval, date [, firstdayofweek [, firstweekofyear]]*)	Возвращает ту часть даты *date*, которая соответствует параметру *interval*. Значения параметров *interval, firstdayofweek* и *firstweekofyear* здесь те же, что и в функции DateDiff
DateSerial(*year, month, day*)	Возвращает переменную подтипа Date, которая соответствует указанным году (параметр *year*), месяцу (параметр *month*) и дню (параметр *day*)
DateValue(*date*)	Возвращает переменную Variant подтипа Date, которая соответствует дате, заданной символьным параметром *date*
Hour(*time*)	Выделяет номер часа из даты или момента времени, заданных параметром time. Возвращает целое число от 0 до 23
IsDate(*expr*)	Возвращает true, если параметр *expr* задает корректную дату, и false в противном случае
Minute(*time*)	Выделяет количество минут из даты или момента времени, заданных параметром *time*. Возвращает целое число от 0 до 59
Month(*date*)	Выделяет номер месяца из даты, заданной параметром *date*. Возвращает целое число от 1 до 12
MonthName(*month[, abbr]*)	Возвращает наименование для месяца с номером *month*. Если логический параметр *abbr* равен true, то наименование месяца представляется в виде

	аббревиатуры, в противном случае — в полном виде
`Now`	Возвращает текущие дату и время в виде, соответствующем региональным настройкам Windows
`Time`	Возвращает текущее системное время
`Timer`	Возвращает количество секунд, прошедших с полуночи
`TimeSerial(`*`hour, minute, second`*`)`	Возвращает переменную подтипа `Date`, которая соответствует указанным часу (параметр *`hour`*), минуте (параметр *`minute`*) и секунде (параметр *`second`*)
`TimeValue(`*`time`*`)`	Возвращает переменную подтипа `Date`, которая соответствует времени, заданному символьным параметром *`time`*
`Weekday(date`*`[, firstdayofweek]`*`)`	Возвращает целое число — день недели для даты, заданной параметром `date`. Параметр *`firstdayofweek`* — это константа, показывающая, какой из дней недели следует считать первым
`WeekdayName(`*`weekday[, abbr[, firstdayofweek]]`*`)`	Возвращает наименование для дня недели с порядковым номером *`weekday`*. Если логический параметр *`abbr`* равен `true`, то наименование дня недели представляется в виде аббревиатуры, в противном случае — в полном виде. Значение параметра *`firstdayofweek`* здесь то же, что и в функции `Weekday`
`Year(`*`date`*`)`	Выделяет год из даты, заданной параметром *`date`*, и возвращает это целое число

Функции для работы с массивами

В табл. П2.17 приведены функции, с помощью которых можно создавать новые массивы и получать сведения об уже имеющихся.

Таблица П2.17. *Функции для работы с массивами*

Функция	Описание
	Возвращает значение типа `Variant`, которое является

`Array(arglist)`	массивом, составленным из элементов списка *`arglist`*. Отдельные элементы в *`arglist`* должны быть отделены друг от друга запятой
`IsArray(varname)`	Возвращает `true`, если переменная *`varname`* является массивом, и `false` в противном случае
`LBound(arrayname[, dimension])`	Возвращает наименьшее значение, которое может принимать индекс в массиве *`arrayname`*. Параметр *`dimension`* определяет, для какой именно размерности массива мы ищем это наименьшее значение (1 для первой размерности, 2 для второй размерности и т.д.). По умолчанию *`dimension`* равно 1
`UBound(arrayname[, dimension])`	Возвращает наибольшее значение, которое может принимать индекс в массиве *`arrayname`*. Параметр *`dimension`* определяет, для какой именно размерности массива мы ищем это наибольшее значение (1 для первой размерности, 2 для второй размерности и т.д.). По умолчанию *`dimension`* равно 1

Функции для работы с подтипами данных

При рассмотрении подтипов данных мы уже описывали функции конвертации, которые применяются для преобразования переменной к тому или иному подтипу (см. табл. П2.9).

В табл. П2.18 приведены функции, с помощью которых можно узнать, к какому подтипу принадлежит заданная переменная.

Таблица П2.18. *Функции для работы с подтипами данных*

Функция	Описание
`IsArray(expr)`	Возвращает `true`, если параметр *`expr`* является массивом, и `false` в противном случае
`IsDate(expr)`	Возвращает `true`, если параметр *`expr`* задает корректную дату (т. е. переменная expr является переменной подтипа `Date`), и `false` в противном случае
`IsEmpty(expr)`	Возвращает `true`, если переменная *`expr`* объявлена, но не инициализирована
`IsNull(expr)`	Возвращает `true`, если переменная *`expr`* не содержит никаких корректных данных

`IsNumeric(`*`expr`*`)`	Возвращает `true`, если выражение *`expr`* может быть рассмотрено в качестве числа, и `false` в противном случае
`IsObject(`*`expr`*`)`	Возвращает `true`, если переменная *`expr`* является указателем на внешний объект, и `false` в противном случае
`VarType(`*`varname`*`)`	Возвращает числовое значение, соответствующее подтипу переменной *`varname`* (см. табл. П2.8)

Прочие функции

Опишем еще несколько часто используемых функций (табл. П2.19).

Таблица П2.19. *Некоторые прочие функции*

Функция	Описание
`CreateObject(`*`servername.typename[, location]`*`)`	Создает экземпляр объекта-сервера автоматизации и возвращает ссылку на него. Здесь *`servername`* — имя приложения, являющегося сервером; *`typename`* — тип или класс создаваемого объекта; *`location`* — сетевое имя компьютера, на котором будет создан объект
`GetObject(`*`[pathname][, classname]`*`)`	Возвращает ссылку на объект класса *`classname`*, который хранится в отдельном файле, путь к которому задается параметром *`pathname`*
`Hex(`*`number`*`)`	Возвращает шестнадцатеричное представление (в символьном виде) числа *`number`*
	Выводит на экран диалоговое окно со строкой ввода и кнопками **OK**, **Отмена** и возвращает введенную в этом окне символьную строку. Параметр *`prompt`* задает сообщение, которое печатается перед строкой

	ввода; *title* определяет заголовок диалогового окна; *default* — значение, которое выводится по умолчанию в строку ввода. Параметры *xpos* и *ypos* определяют координаты левого верхнего угла окна. В случае необходимости элементам диалогового окна можно сопоставить контекстно-зависимую помощь. Параметр *helpfile* задает путь к файлу помощи, а число *context* — идентификатор содержания помощи
`InputBox(`*`prompt[, title] [, default] [, xpos][,ypos] [, helpfile, context]`*`)`	
`MsgBox(`*`prompt[, buttons] [, title] [, helpfile, context]`*`)`	Выводит на экран диалоговое окно с сообщением и различными кнопками и возвращает результат нажатия на одну из кнопок (возможные варианты возвращаемых функцией значений приведены в табл. П2.6). Параметр *prompt* задает сообщение, *title* определяет заголовок диалогового окна. Числовой параметр *buttons* определяет, какие именно кнопки должны быть представлены в окне (возможные значения этого параметра приведены в табл. П2.5). Параметры *helpfile* и *context* имеют то же значение, что и в функции `InputBox`
`Oct(`*`number`*`)`	Возвращает восьмеричное представление (в символьном виде) числа *number*

Функции и процедуры пользователя

Для определения процедуры, т.е. подпрограммы, которая не возвращает никакого значения, в VBScript используется конструкция `Sub`…

`End Sub`. После названия процедуры в круглых скобках указывается список ее параметров, например:

```
Sub MyProcedure(Param1, Param2)
 Dim Sum
 Sum = Param1+Param2
End Sub
```

Если процедура не имеет параметров, то в скобках после имени ничего указывать не нужно:

```
Sub MyProcedure()
 …
End Sub
```

Вызывать процедуру из сценария можно двумя способами. Во-первых, можно просто написать имя нужной процедуры и указать через пробел список передаваемых параметров, например:

```
MyProcedure 3,10
```

Во-вторых, можно использовать специальный оператор `Call`, при этом список параметров обязательно должен быть заключен в круглые скобки:

```
Call MyProcedure(3, 10)
```

Для определения функции, т.е. подпрограммы, которая возвращает определенное значение, применяется конструкция `Function…End Function`. Как и при описании процедур, после названия функции в круглых скобках указывается список ее параметров, например:

```
Function MyFunction(Param1, Param2)
 …
End Function
```

Для того чтобы возвратить из функции какое-либо значение, нужно внутри функции присвоить это значение переменной, название которой совпадает с именем функции:

```
Function MyFunction(Param1, Param2)
Dim Sum
 Sum = Param1 + Param2
MyFunction = Sum
End Function
```

Если возвращаемое функцией значение не нужно присваивать никакой переменной, то функция вызывается так же, как и процедура — пишется имя этой функции и через пробел указывается список ее аргументов:

```
MyFunction 3, 5
```

Если же необходимо записать значение функции в какую-либо переменную, то аргументы функции заключаются в круглые скобки:

```
Dim a
a = MyFunction(3, 5)
```

Приложение 3
Средства разработки и отладки сценариев

В принципе, можно создавать сценарии в Блокноте Windows или в текстовых редакторах файловых оболочек типа Far Manager, а отлаживать их с помощью вывода в нужных местах сценария значений переменных на экран (метод `Echo` объекта `WScript`), однако при разработке больших сложных сценариев намного удобнее работать со специализированными редакторами и отладчиками.

Создание и редактирование сценариев

Одним из наиболее мощных и удобных редакторов сценариев для Windows является Primalscript, который разработан компанией SAPIEN Technologies, Inc. Последней на момент написания книги являлась версия Primalscript 2.2, ознакомительная 40-дневная версия которой может быть получена с сайта SAPIEN (http://www.sapien.com).

Редактор Primalscript

Редактор Primalscript поддерживает среду разработки практически для всех распространенных языков сценариев: JScript, VBScript, Perl, Python, Rexx, TCL, WinBatch, LotusScript и т.д. Для нас самым важным является то, что в Primalscript 2.2 полностью реализована поддержка Windows Script Host 5.6.

Ниже будут описаны основные операции, которые позволяет выполнить Primalscript при работе со сценариями.

Создание нового одиночного сценария

Для того чтобы создать новый одиночный сценарий (например, на языке JScript или VBScript), нужно выбрать пункт меню **File|New**, после чего на экран будет выведено диалоговое окно **New**, на вкладке **Files** которого представлен список поддерживаемых типов сценариев (рис. П3.1).

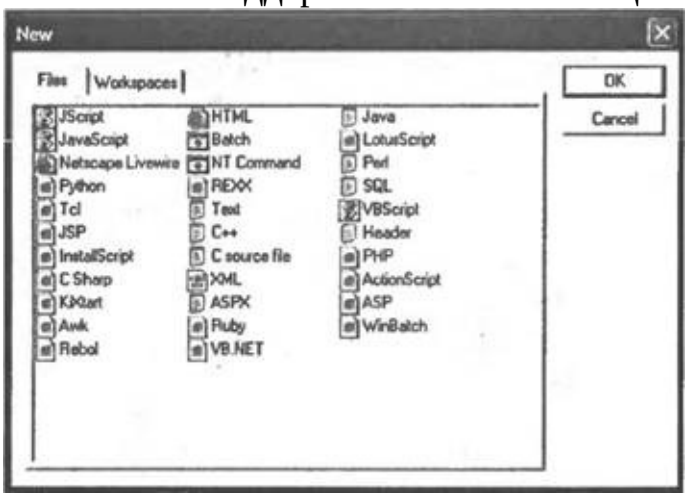


Рис. П3.1. Создание нового сценария — список типов файлов,

поддерживаемых Primalscript

Замечание

Так как локализованных русскоязычных версий программ Primalscript и MS Script Debugger, которые описываются в этом приложении, пока нет (и вряд ли будут), автор специально не указывал перевод команд и пунктов меню на русский язык.

Указав в этом списке нужный тип (например, JScript) и нажав кнопку OK, мы получим в окне редактирования заготовку сценария с заполненным заголовком (рис. П3.2).

Рис. П3.2. Заготовка нового сценария на языке JScript

Открытие существующего одиночного сценария

Открыть уже существующий сценарий можно с помощью пункта меню **File|Open**. Диалоговое окно открытия файлов позволяет запретить изменения выбранного файла (режим "Только чтение"), а также отобразить этот файл в текстовом (Text) или шестнадцатеричном (Binary) виде (рис. П3.3).

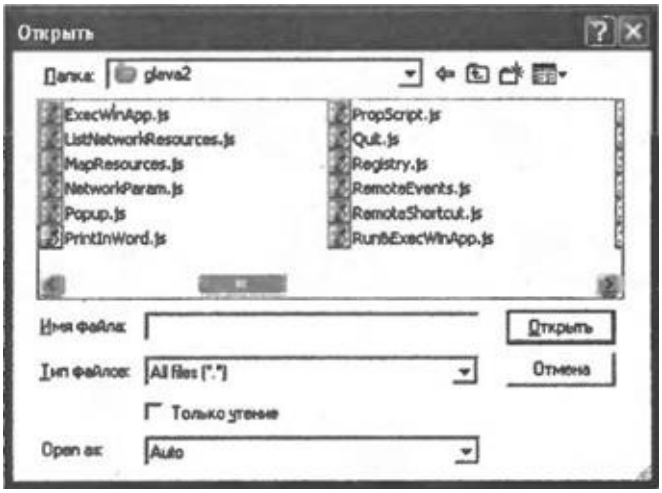


Рис. П3.3. Открытие существующего сценария

Выбранный файл будет отображен в окне редактирования, при этом различные элементы сценария выделены цветом (рис. П3.4).

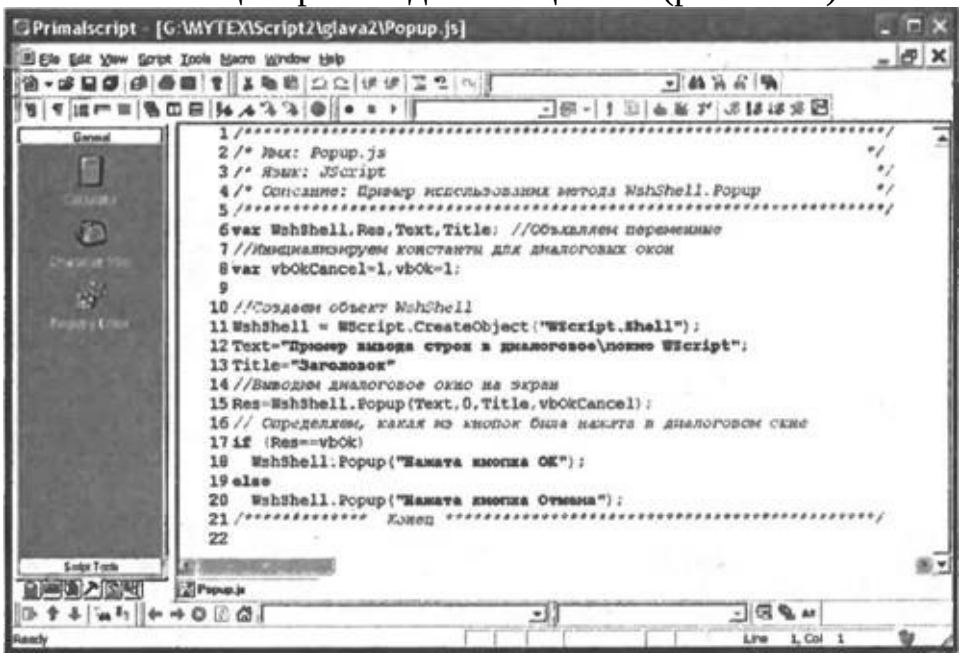


Рис. П3.4. JScript-сценарий в режиме редактирования

Создание нового WS-файла

Как и в случае обычного одиночного сценария, новый WS-файл создается с помощью пункта меню **File|New**, однако в диалоговом окне **New** нужно выбрать вкладку **Workspaces**, где представлены несколько мастеров для создания файлов различных типов (рис. П3.5).

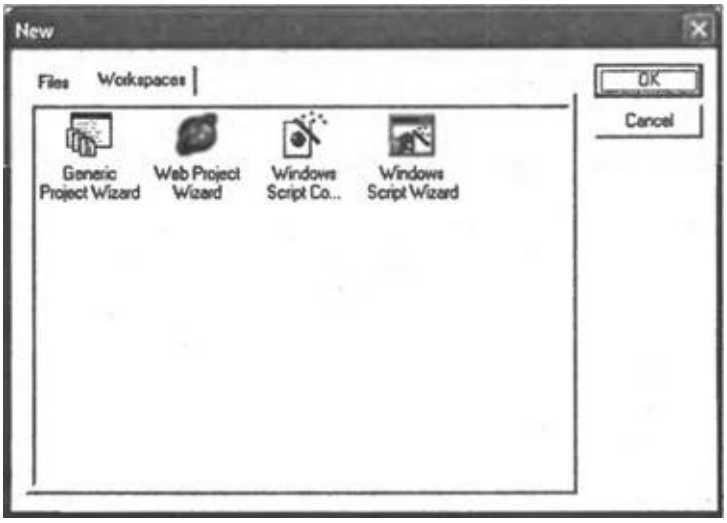


Рис. П3.5. Список мастеров для создания файлов различных типов

В этом списке нам нужно выбрать **Windows Script Wizard**, после чего, на экран будет выведено диалоговое окно, в котором указывается имя создаваемого сценария (**Script Name**), каталог, в котором он будет храниться (**Location**), название задания (**Job Name**) и выбирается используемый язык (**Language**) (рис. П3.6).

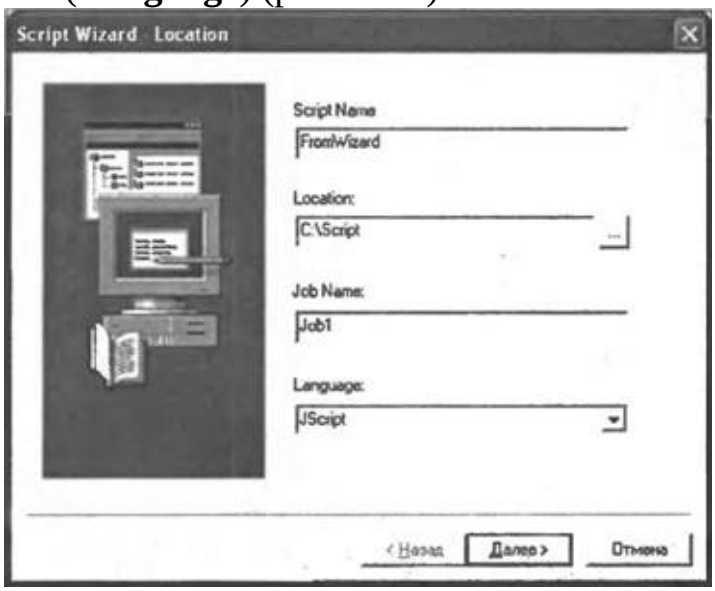


Рис. П3.6. Общая информация о создаваемом сценарии

На втором шаге работы мастера мы указываем, какие внешние объекты будут использоваться в создаваемом сценарии (элемент `<object>`). По умолчанию предлагаются объекты `Dictionary`, `FileSystem`, `Network` и `Shell` (рис. П3.7).

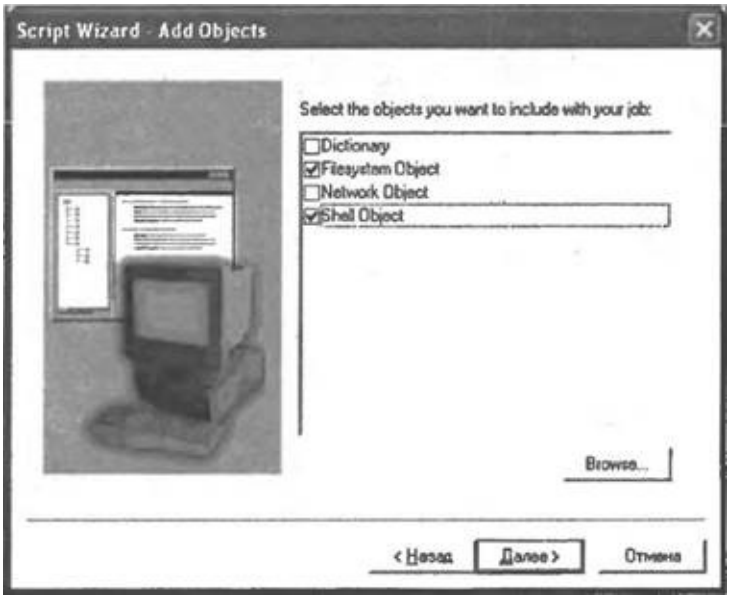


Рис. П3.7. Выбор внешних объектов, которые будут использоваться в создаваемом сценарии

Кнопка **Browse** открывает диалоговое окно **Select Object**, с помощью которого можно выбрать любой зарегистрированный в системе объект (рис. П3.8).

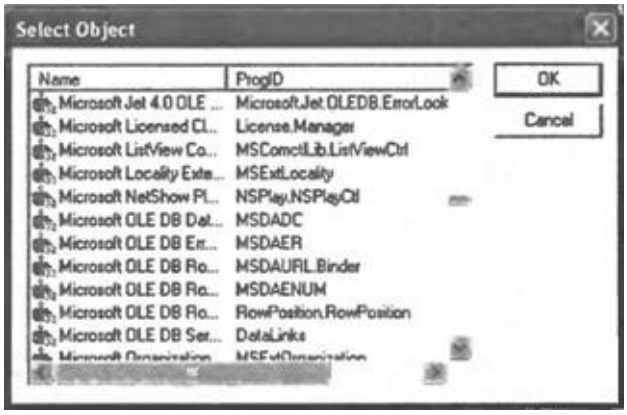


Рис. П3.8. Список зарегистрированных в системе объектов

Третий шаг работы мастера позволяет добавить в создаваемый WS-файл ссылки на нужные библиотеки типов зарегистрированных объектов (элемент `<reference>`) (рис. П3.9).

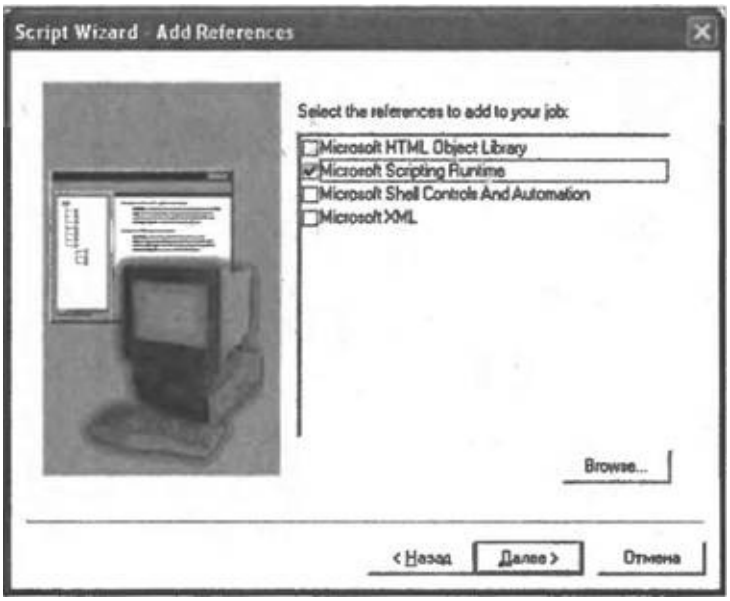


Рис. П3.9. Выбор библиотек типов, которые будут использоваться в создаваемом сценарии

Как и на предыдущем шаге, кнопка **Browse** используется для выбора библиотеки типов, которая не представлена в списке по умолчанию (рис. П3.10).

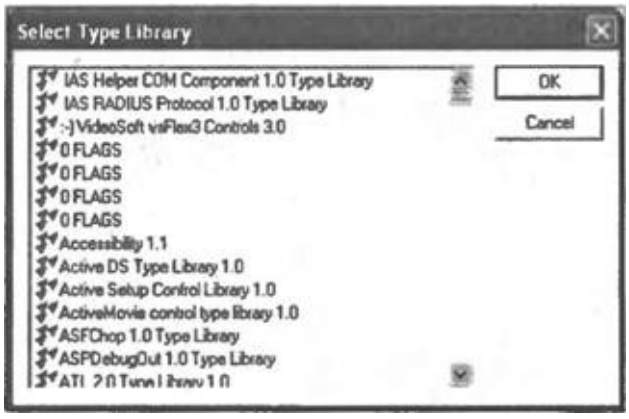


Рис. П3.10. Список всех библиотек типов

На четвертом шаге требуется указать, какие внешние файлы со сценариями должны быть включены в создаваемое задание (элемент `<script>` с атрибутом `src`) (рис. П3.11).

Установив флажок **Copy files to script folder**, можно скопировать выбранные файлы в тот каталог, где будет находиться создаваемый WS-файл.

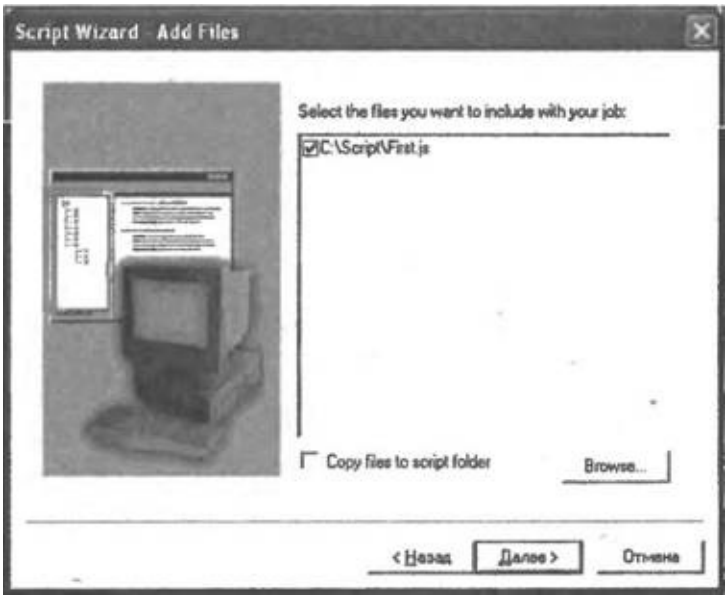


Рис. П3.11. Подключаемые внешние файлы со сценариями

Пятый шаг является заключительным в работе мастера. Здесь нам выдается вся информация о создаваемом сценарии (рис. П3.12).

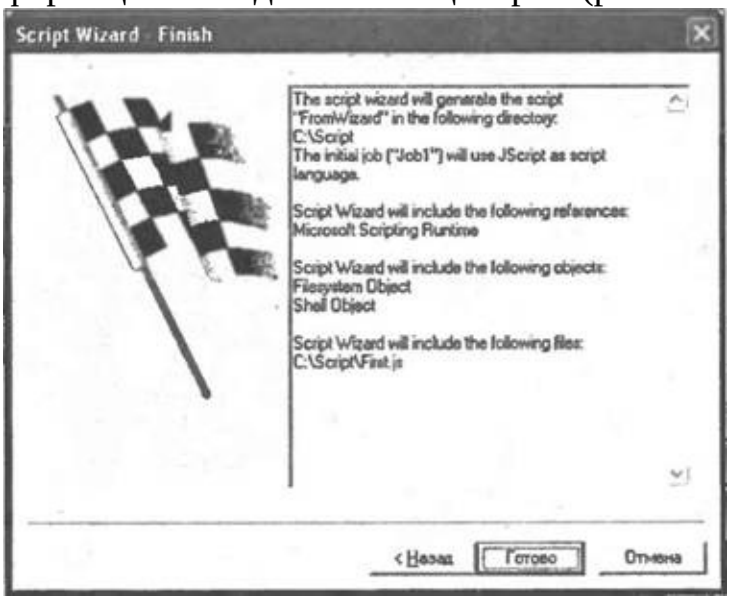


Рис. П3.12. Итоговая информация о создаваемом сценарии

Созданный с помощью мастера сценарий отображается в двух окнах (режим workspace) (рис. П3.13).

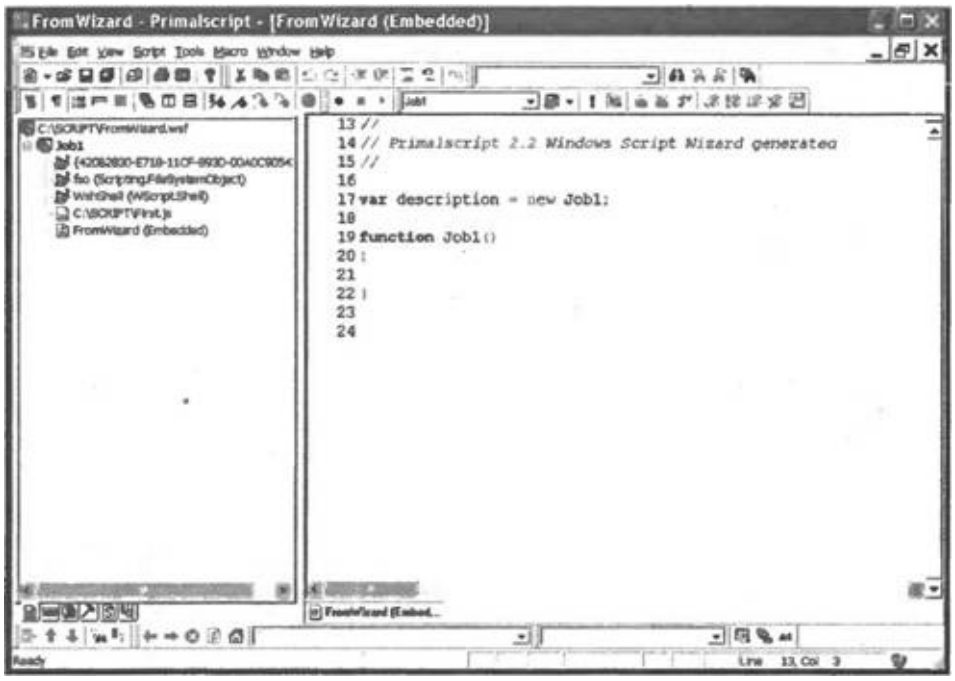


Рис. П3.13. Просмотр и редактирование WS-файла в режиме workspace

Слева, на панели **Workspace Nexus** окна **Nexus**, в графическом виде представлена структура созданного WS-файла. Мы видим, что пока этот файл содержит единственное задание Job1, внутри которого показаны используемые объекты, ссылки на библиотеки типов и подключаемые внешние файлы со сценариями, которые мы задавали при описании WS-файла в мастере, а также раздел с внутренним (Embedded) сценарием.

Справа расположено окно редактирования, в котором показано содержимое внутреннего сценария. По умолчанию здесь создается единственная функция `Job1()`.

От подобного графического представления WS-файла можно всегда перейти к обычному текстовому представлению. Для этого нужно выделить имя WS- файла (самая верхняя строка на панели **Workspace Nexus**), нажать правую кнопку мыши и выбрать в контекстном меню пункт **Open as textfile**. После этого в окне редактирования будет полностью показан WS-файл в привычном для нас текстовом виде с цветовым выделением различных элементов (рис. П3.14).

Рис. П3.14. Просмотр и редактирование WS-файла в текстовом виде

Открытие существующего WS-файла

Имеющийся на диске WS-файл открывается так же, как и обычный одиночный сценарий — с помощью пункта меню **File|Open**. Для примера откроем созданный в *главе 7* файл PhoneBook.wsf, который содержит четыре задания. По умолчанию этот файл открывается в режиме workspace (рис. П3.15).

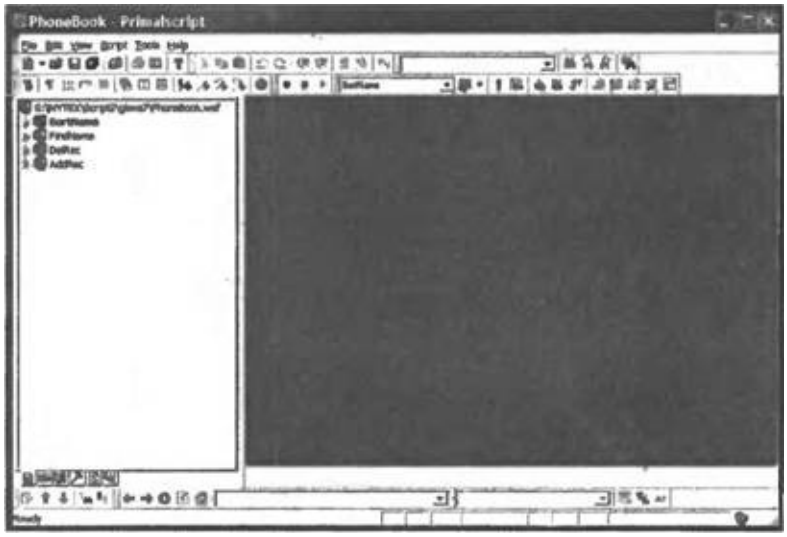


Рис. П3.15. Просмотр многозадачного файла PhoneBook.wsf в режиме workspace

Редактирование WS-файла на панели *Workspace Nexus*

При работе с WS-файлом в режиме workspace на панели **Workspace Nexus** можно производить следующие операции.

□ Добавлять новое задание (элемент `<job>`). Для этого нужно выделить имя WS-файла, нажать правую кнопку мыши и выбрать в контекстном меню пункт Add **new job to workspace**.

□ Удалять имеющееся задание. Для этого требуется выделить нужное задание, нажать правую кнопку мыши и выбрать в контекстном меню пункт **Remove job**.

□ Определять свойства задания. Для этого нужно выделить задание, нажать правую кнопку мыши, выбрать в контекстном меню пункт **Properties**. После этого на экран будет выведено диалоговое окно **Job Properties**, в котором нужно заполнить вкладку **Properties** (рис. П3.16).

□ Описывать элементы `<description>`, `<arguments>`, `<usage>` и `<example>`. Для этого нужно выделить задание, нажать правую кнопку мыши, выбрать в контекстном меню пункт **Properties**. После этого на экран будет выведено диалоговое окно **Job Properties**, в котором нужно выбрать вкладку **Description**, **Arguments**, **Usage** или **Example** соответственно (см. рис. П3.16). Например, на рис. П3.17 представлено содержимое вкладки Arguments для файла ArgMenu.wsf, который мы создали в *главе* 7.

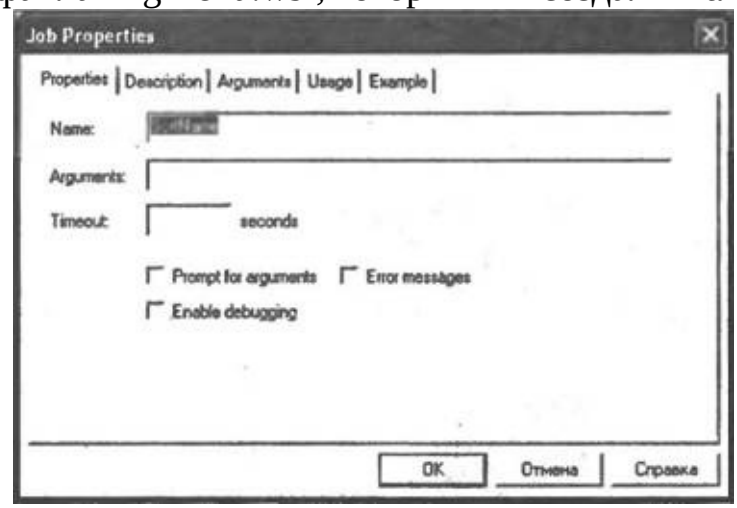


Рис. П3.16. Определение свойств текущего задания

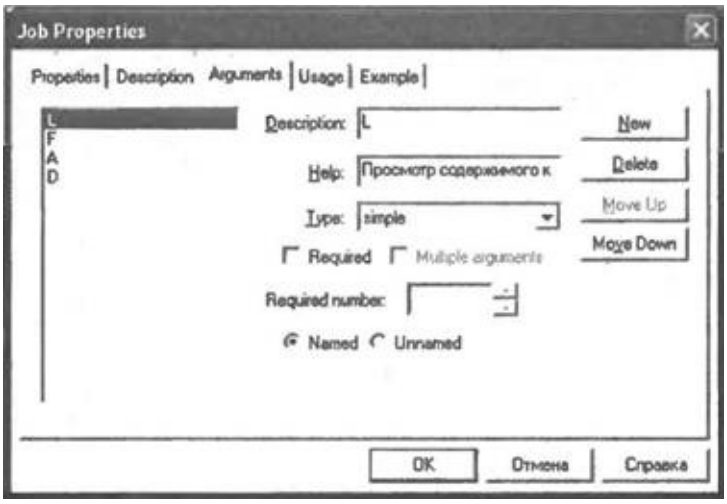


Рис. П3.17. Диалоговое окно, представляющее содержимое элемента `<arguments>` для файла ArgMenu.wsf

□ Добавлять в задание внешний файл со сценарием (элемент `<script>` с атрибутом `src`), внутренний сценарий (элемент `<script>` без атрибута `src`), внешний объект (элемент `<object>`), ссылку на библиотеку типов (элемент `<reference>`), символьную или числовую константу (элемент `<resource>`). Для этого нужно выделить задание, нажать правую кнопку мыши и выбрать в контекстном меню пункт **Add files to job**, **Add script to job**, **Add object to job**, **Add reference to job** или **Add resource to job** соответственно.

□ Изменять свойства у находящихся внутри задания элементов (`<script>`, `<object>`, `<reference>` или `<resource>`). Для этого нужно выделить соответствующий элемент, нажать правую кнопку мыши и выбрать в контекстном меню пункт **Properties**. После этого на экран будет выведено диалоговое окно (**Script Properties**, **Object Properties**, **Reference Properties** или **Resource Properties**), в котором можно поменять свойства соответствующего элемента.

□ Удалять находящиеся внутри задания элементы (`<script>`, `<object>`, `<reference>` или `<resource>`). Для этого нужно выделить соответствующий элемент, нажать правую кнопку мыши и выбрать в контекстном меню пункт Remove.

Запуск одиночного сценария

Запуск одиночного сценария, который открыт в активном окне редактирования, производится с помощью пункта **Run Script** меню **Script**.

При этом сценарий может запускаться как с помощью cscript.exe (устанавливается по умолчанию), так и с помощью wscript.exe, а выводимая сценарием информация может перенаправляться в специальное окно **Output** редактора.

Нужный режим запуска сценария того или иного типа задается на вкладке **Languages** диалогового окна **Options** (пункт меню **Tools|Options**). Например, на рис. П3.18 приведены настройки, позволяющие запускать JScript-сценарии в графическом режиме без перенаправления вывода в окно **Output**.

Рис. П3.18. Настройка режима запуска JScript-сценариев

Запуск задания из WS-файла

Для запуска активного задания из WS-файла нужно выбрать пункт **Run active Job** меню **Script**. Режим запуска заданий настраивается на вкладке **WSH** диалогового окна **Options** (пункт меню **Tools|Options**) (рис. П3.19).

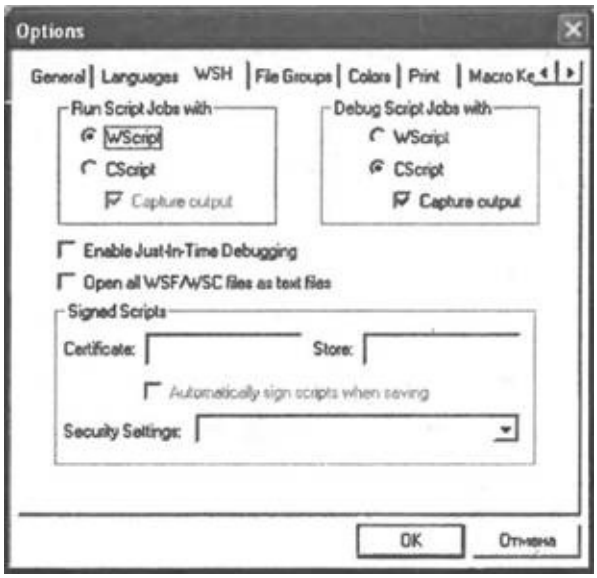


Рис. П3.19. Настройки параметров WS-файлов

Как мы видим, на этой вкладке можно также указать сертификат для цифровой подписи сценария и выбрать один из трех режимов безопасности для выполнения сценариев *(см. разд. "Три режима выполнения сценариев WSH" главы 3).*

Подписывание сценариев

Для того чтобы подписать сценарий в Primalscript, нужно сначала выбрать нужный цифровой сертификат. Имя этого сертификата (**Certificate**) и, в случае необходимости, хранилище (**Store**), в котором он находится, указываются на вкладке **WSH** диалогового окна **Options** (пункт меню **Tools|Options**) (см. рис. П3.19). После этого подписывание открытого сценария производится с помощью пункта **Sign Script** меню **Script**.

Возможности пользовательского интерфейса

В окне редактирования можно отображать номера строк в колонке слева от текста сценария. За включение/выключение этого режима отвечает переключатель **View Line Numbers** в меню **View**. Для перехода к строке с заданным номером нужно нажать <Ctrl>+<L> или выбрать пункт **Jump to line** меню **View**.

В любое место сценария можно вставить *закладку* (Bookmark) с помощью нажатия клавиш <Ctrl>+<F2> или выбора пункта **Toggle**

Breakpoint меню **View**. После этого можно перемещаться между закладками путем нажатия клавиш <F2> (переход к следующей закладке, пункт меню **View|Next Bookmark**) или <Shift>+<F2> (переход к предыдущей закладке, пункт меню **View|Previous Bookmark**).

Различные элементы сценария (ключевые слова, функции, объекты и т.д.) выделяются в окне редактирования разным цветом. Цветовые схемы выделения для поддерживаемых типов сценариев настраиваются на вкладке **Colors** диалогового окна **Options** (пункт меню **Tools|Options**) (рис. П3.20).

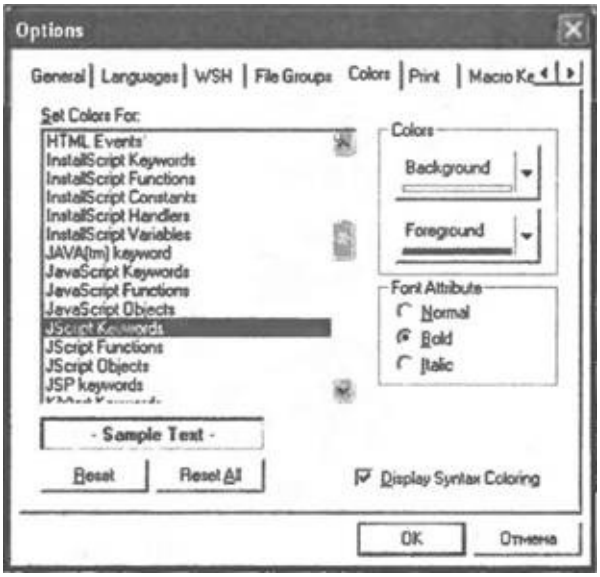


Рис. П3.20. Настройка цветового выделения элементов сценариев различных типов

Для увеличения скорости написания кода сценария можно вставлять в текст шаблоны конструкций определенного языка (например, `switch…case` в JScript или `For…To…Step` в VBScript). Для этого в окне **Nexus** нужно выбрать панель **Snippets Nexus** (рис. П3.21).

Рис. П3.21. Панель Snippets Nexus окна Nexus

Затем на этой панели следует раскрыть папку, соответствующую нужному языку, и выбрать требуемую конструкцию (рис. П3.22).

Рис. П3.22. Результат вставки шаблона конструкции `Select…Case` языка VBScript

Самое, пожалуй, полезное свойство редактора Primalscript заключается в возможности автоматического завершения вводимых выражений (эта функция имеется практически во всех современных средах разработки типа Microsoft Visual Basic for Applications или Borland Delphi). Например, после

ввода имени объекта автоматически появляется список всех свойств и методов этого объекта (рис. П3.23).

Рис. П3.23. Автозавершение ввода — список всех свойств и методов объекта

При выборе из этого списка какого-либо метода на экране появляется подсказка о параметрах данного метода (рис. П3.24).

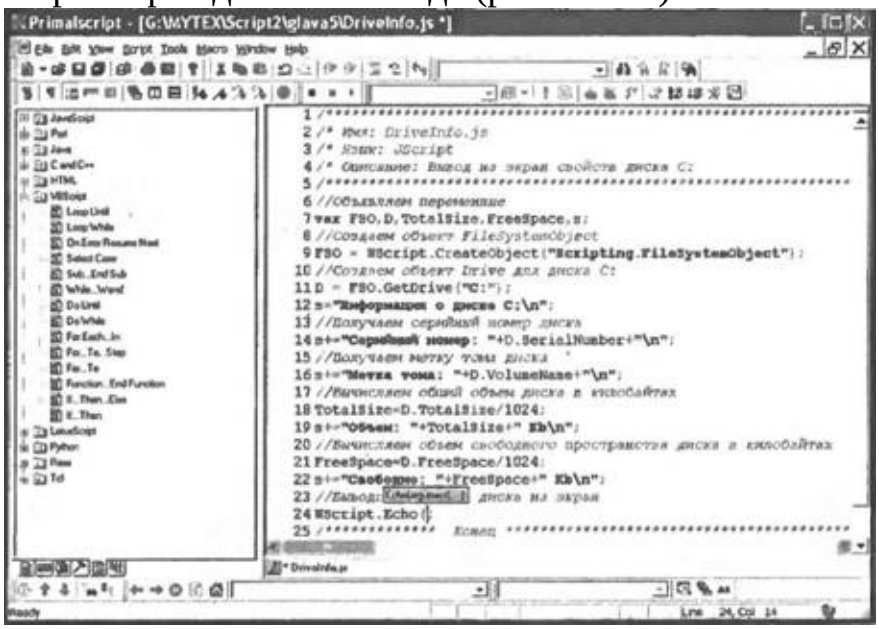


Рис. П3.24. Автозавершение ввода — список параметров введенного метода

Можно также вывести список свойств и методов для тех объектов,

которые были созданы в сценарии ранее. Для этого нужно выделить в тексте имя этого объекта и выбрать пункт **List Members** меню **Edit** (рис. П3.25).

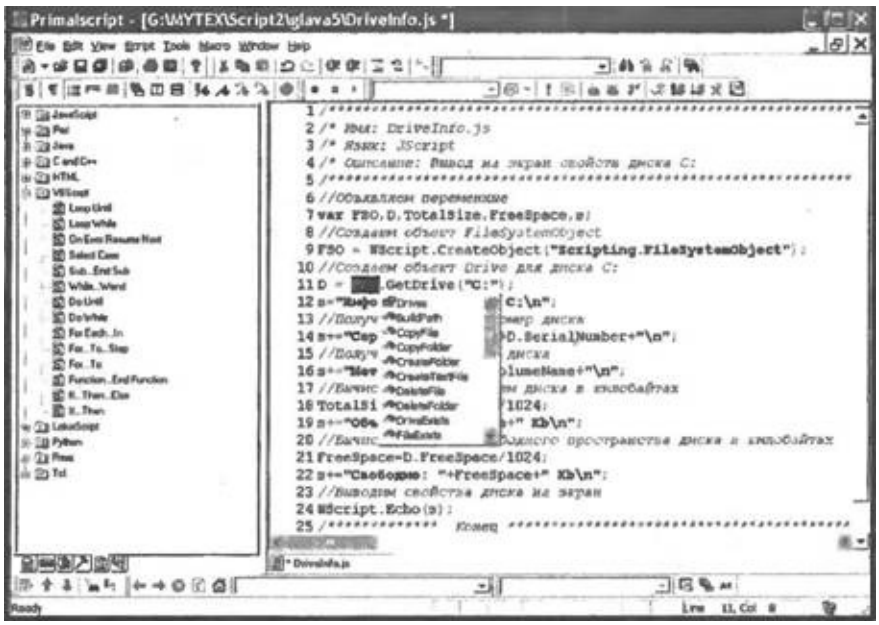


Рис. П3.25. Список всех свойств и методов объекта `FileSystemObject`

Для получения списка параметров введенной ранее функции (метода объекта) нужно выделить в тексте название этой функции (метода) и выбрать пункт **Parameter Info** меню **Edit** (рис. П3.26).

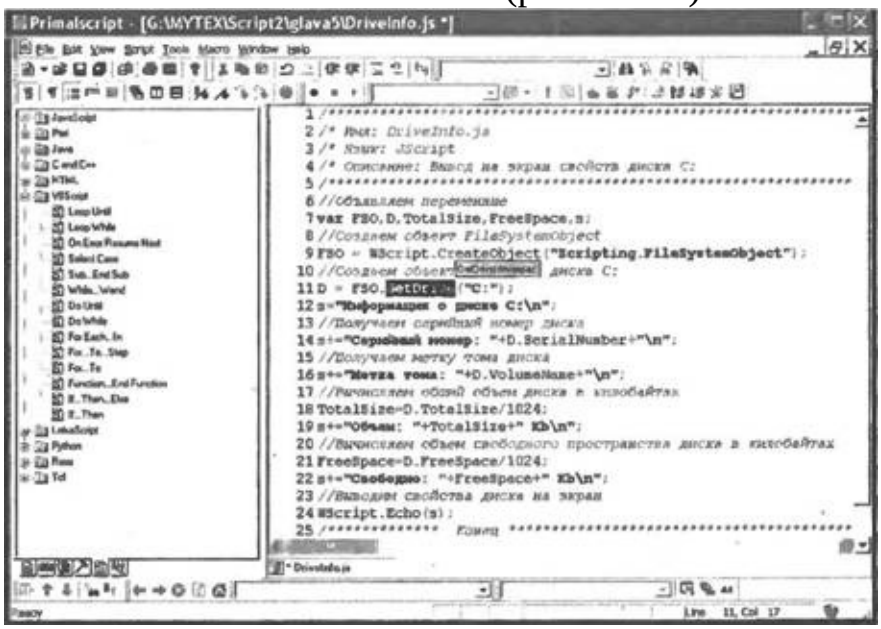


Рис. П3.26. Параметры метода `GetDrive()`

Другие редакторы

Упомянем еще несколько из множества редакторов, которые могут использоваться для создания и изменения сценариев WSH.

Script Editor — программа, входящая в состав Microsoft Office 2000/XP. В ней имеется функция выделения цветом элементов JScript-, VBScript- и WS-сценариев, а также поддерживается автоматическое завершение ввода ключевых слов.

Aditor — условно-бесплатный редактор файлов текстового формата, который позволяет запускать изменяемые сценарии, не выходя из режима редактирования, а также поддерживает выделение цветом ключевых слов языков JScript и VBScript. Aditor можно загрузить с http://aditor.swrus.com/.

UltraEdit-32 — условно-бесплатный редактор текстовых файлов, обладающий, в целом, теми же возможностями, что и Aditor. Может быть загружен с http://www.ultraedit.com/.

Отладка сценариев в Microsoft Script Debugger

Если при выполнении сценариев возникают ошибки или получаются непредвиденные результаты, можно воспользоваться специальным отладчиком для трассировки сценария и проверки значений переменных. Мы в качестве такого отладчика рассмотрим программу Microsoft Script Debugger, версии которой для различных операционных систем можно бесплатно получить с сайта Microsoft (http://msdn.microsoft.com/scripting). Этот отладчик позволяет работать со сценариями, которые встроены в HTML- или ASP-файлы, а также со сценариями Windows Script Host.

Активизация отладчика

Активизировать отладчик при работе со сценариями можно несколькими способами.

Во-первых, внутри JScript- или VBScript-сценариев можно вставить специальные операторы (`debugger` для JScript и `Stop` для VBScript) и выполнить сценарий с параметром //D. Для примера рассмотрим сценарий ForDebug.js, содержимое которого приведено в листинге П3.1.

Листинг П3.1. JScript-сценарий для отладки в Microsoft Script Debugger

```
    /*****************************************************************
    /*                                                          Имя:
ForDebug.js                                                  */
    /*                                                         Язык:
JScript                                                    */
    /*  Описание:  Сценарий  для  отладки  в  Microsoft  Script
Debugger      */
    /*****************************************************************
    var s;
    function MyFunc() {
     WScript.Echo("Функция MyFunc()");
    }
    WScript.Echo("Это сообщение выведется до запуска отладчика");
    debugger;
    s="А это сообщение появится ";
    s+="уже в отладчике";
    WScript.Echo(s);
```

```
    MyFunc();
    /*************                                              Конец
*******************************************/
```

Запустим этот сценарий из командной строки следующим образом:

```
wscript.exe //D ForDebug.js
```

Тогда сначала нам будет выведено диалоговое окно со строкой "Это сообщение выведется до запуска отладчика", а после нажатия в этом окне кнопки OK запустится отладчик, и управление передастся ему (рис. П3.27). Далее выполнения сценария может производиться в отладчике *(см. следующий раздел).*

Замечание

Файл со сценарием в отладчике доступен только для чтения.

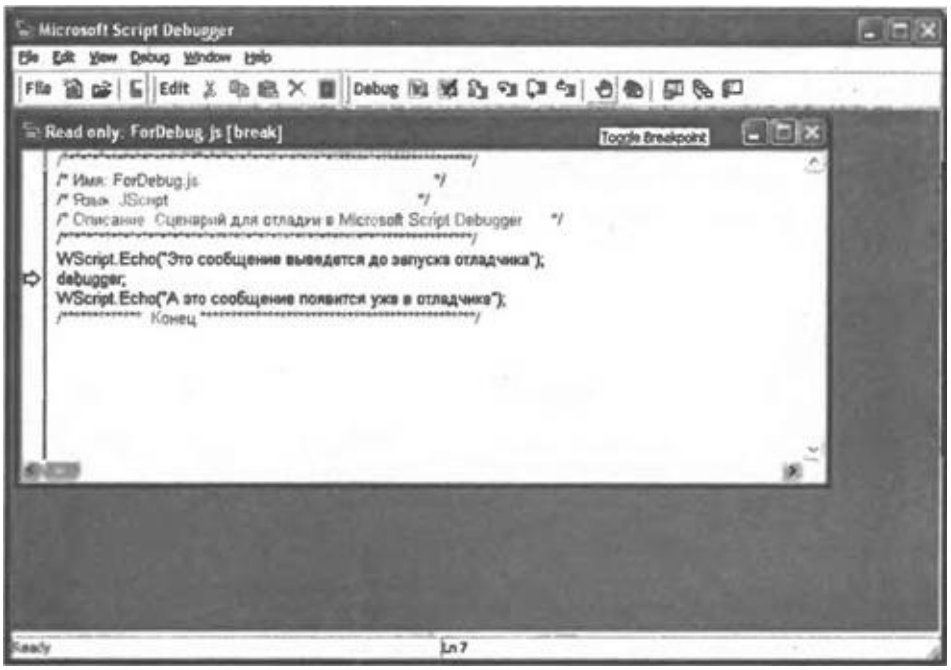


Рис. П3.27. Активизация отладчика с помощью параметра `//D` и специальных операторов

Второй путь активизации отладчика состоит в использовании параметра `//X` при выполнении сценария:

```
wscript.exe //X ForDebug.js
```

При этом отладчик запускается сразу, с первой строки сценария (рис. П3.28).

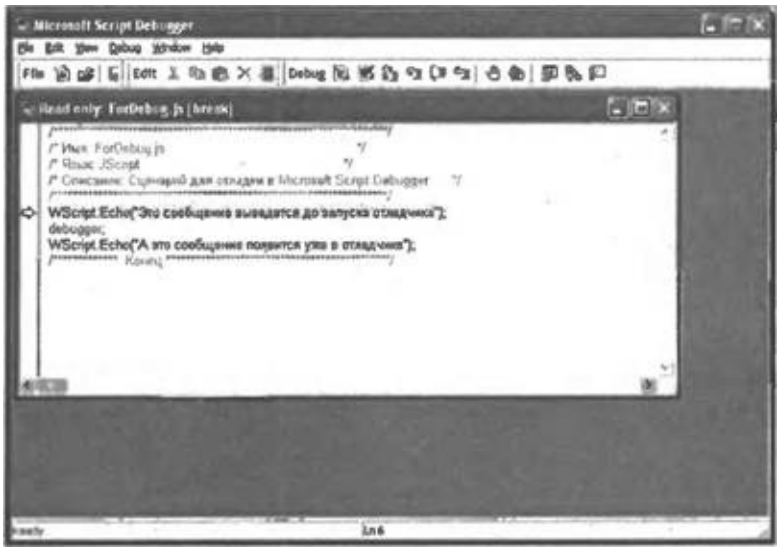


Рис. П3.28. Активизация отладчика с помощью параметра `//X`

Режим отладки WS-файлов зависит от значения атрибута `debug` в инструкции `<?job?>` *(см. главу 3)*.Для примера рассмотрим WS-файл ForDebug.wsf, содержимое которого приведено в листинге П3.2.

Листинг П3.2. WS-файл для отладки в Microsoft Script Debugger

```
<job id="Encoded">
 <?job debug="true"?>
 <runtime>
  <description>
  Имя: ForDebug.wsf
  Описание: WS-файл для отладки в Microsoft Script Debugger
  </description>
 </runtime>
 <script language="JScript">
 WScript.Echo("Это сообщение выведется до запуска отладчика");
 debugger;
 WScript.Echo("А это сообщение появится уже в отладчике");
 </script>
</job>
```

Так как значение атрибута `debug` равно `true`, то оператор `debugger` передаст управление отладчику, причем, в отличие от одиночного сценария, WS-файл можно запускать как с ключами `//D` или `//X`, так и без них.

Если же изменить в сценарии ForDebug.wsf инструкцию `<?job?>` следующим образом:

```
<?job debug="false"?>
```

то отладка будет отключена, причем независимо от использования ключей `//D` и `//X` при запуске сценария (это может понадобиться при эксплуатировании в рабочем режиме WS-файла, содержащего операторы `debug` или `Stop`).

Команды отладчика

Команды, имеющиеся в Microsoft Script Debugger, позволяют выполнять трассировку сценариев, просматривать список вызванных процедур или функций, анализировать и изменять значения переменных.

Установка и удаление точек прерывания

Для того чтобы установить точку прерывания в определенной строке сценария, нужно поместить курсор в эту строку и нажать <F9> или выбрать пункт меню **Debug|Toggle Breakpoint**. Строки с точками прерывания будут отмечены красными точками около левой границы окна отладчика. При достижении точки прерывания отладчик останавливает исполнение сценария.

Для удаления одной точки прерывания необходимо поместить курсор в нужную строку и вновь нажать <F9> или выбрать пункт меню **Debug|Toggle Breakpoint**. Если нужно убрать все точки прерывания, то можно воспользоваться командой меню **Debug|Clear All Breakpoints**.

Выполнение сценария

Открытый в отладчике сценарий может выполняться в разных режимах с помощью соответствующих команд меню **Debug**.

С помощью команды **Debug|Run** (или нажатия клавиши <F5) можно выполнить все операторы сценария до первой точки прерывания.

Для того чтобы выполнить только один оператор (режим пошагового выполнения), нужно выбрать команду **Debug|Step Into** или нажать <F8>. Следующий исполняемый оператор при этом будет помечен стрелкой около левой границы окна отладчика.

Если в режиме пошагового выполнения в сценарии встречается вызов определенной пользователем процедуры/функции, то возможны два варианта. Продолжая выполнять команду **Debug|Step Into**, мы будем останавливаться на каждом операторе внутри процедуры/функции. Для

того чтобы, находясь внутри процедуры/функции, не проходить оставшиеся операторы по отдельности, можно выполнить команду **Debug|Step Out** (или нажать <Ctrl>+<Shift>+<F8>). После этого управление передастся оператору, который стоит в сценарии первым после вызова этой процедуры/функции.

Если нет необходимости проверять внутреннюю работу процедуры/функции пользователя, то нужно выбрать команду **Debug|Step Over** или нажать комбинацию клавиш <Shift>+<F8>. При этом данная процедура/функция выполнится без остановок и исполнение сценария остановится на следующем после нее операторе.

Команда **Debug|Stop Debugging** прерывает исполнение сценария и завершает процесс отладки.

Просмотр стека вызовов

В отладчике можно вывести окно Call Stack со списком всех активных процедур и функций сценария. Для этого нужно выполнить команду **View|Call Stack**. Например, если вызвать это окно, находясь внутри функции `MyFunc()` в сценарии ForDebug.js, то в списке мы увидим название функции `MyFunc()` (рис. П3.29).

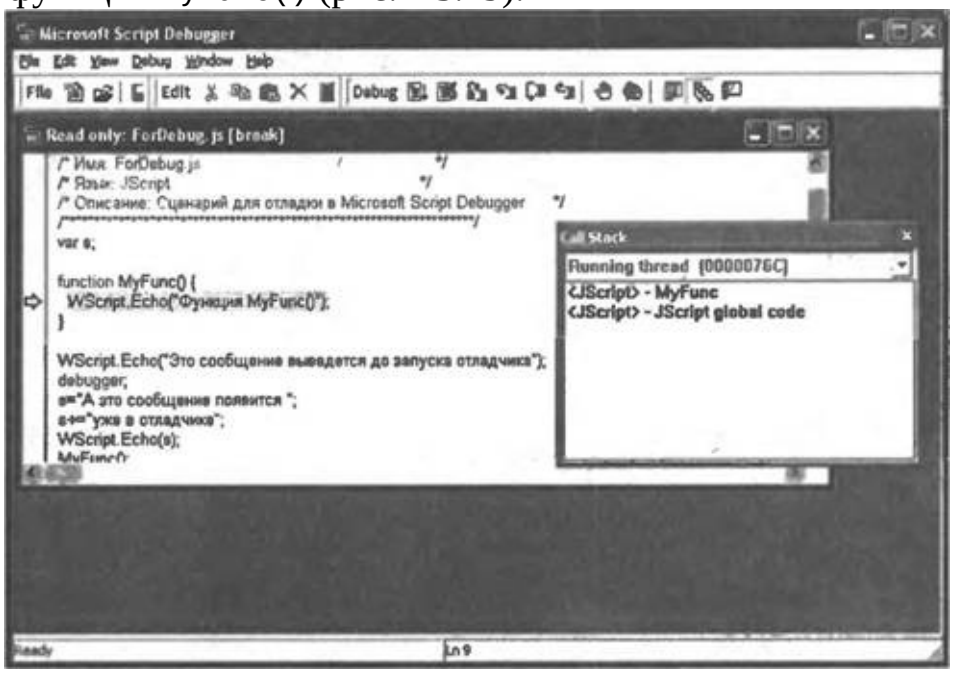


Рис. П3.29. Окно `Call Stack`

Просмотр и изменение значений переменных

Получать и изменять текущие значения переменных или свойств объектов во время остановки исполнения сценария позволяет окно **Command**, которое вызывается командой **View|Command Window**. При этом анализ переменных в JScript- и VBScript-сценариях производится следующим образом.

Для просмотра значения переменной в JScript-сценарии нужно в окне **Command** набрать имя этой переменной и нажать <Enter>. Например, для того, чтобы увидеть значение переменной s в сценарии ForDebug.js, мы в окне Command вводим s и нажимаем <Enter> (рис. П3.30).

Чтобы вывести значение переменной в VBScript-сценарии, нужно в окне Command ввести имя этой переменной и поставить перед ним знак ?. Например,

```
? s
```

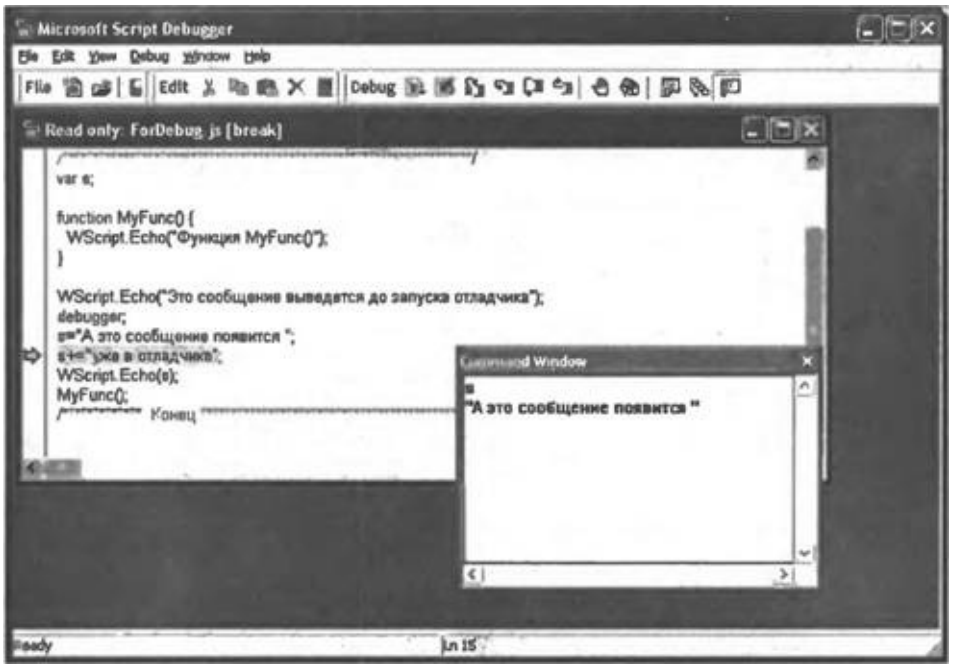


Рис. П3.30. Просмотр значений переменных в окне

Для изменения значения переменной нужно просто присвоить новое значение этой переменной в окне **Command** (это относится и к JScript- и к VBScript-сценариям). Например,

```
s="Новое значение переменной s"
```

Приложение 4
Ошибки выполнения сценариев в WSH

Ошибки, которые могут возникнуть при выполнении сценариев WSH, вместе с описанием возможных причин их появления, приведены в табл. П4.1.

Таблица П4.1. *Ошибки WSH 5.6*

Сообщение об ошибке	Причина
A duplicate name for a named or unnamed element was encountered: xxx	Попытка повторного использования имени аргумента
Argument list too long	Связано с запуском сценария при помощи технологии Drag-and-Drop: на файл сценария "опущено" слишком много параметров — имен файлов
Cannot write to wsh.log	При вызове метода `LogEvent` в Windows 9x или Windows ME файл %windir%\wsh.log оказался заблокированным для записи
Can't save settings	Ошибка при сохранении файла с настройками сценария (*.wsh)
Environment variable <name> could not be removed	Вызов метода `Environment.Remove` для несуществующей переменной среды
Invalid attempt to call Exec without a command	Вызов метода `WshShell.Exec()` без указания аргумента (команды для выполнения)
Invalid shortcut path name	Попытка создать ярлык с неправильным расширением файла (расширение должно быть lnk или url)
Printer <name> not found	Неправильно указано имя принтера при вызове метода `SetDefaultPrinter`
Protocol handler for <name> could not be found	Попытка установить ярлык на сетевой ресурс, использующий некорректно зарегистрированный обработчик протокола

Registry key <name> contains invalid root	Вызов метода `RegRead` или `RegWrite` для некорректного ключа реестра
Registry key <name> could not be opened	Вызов метода `RegRead` для несуществующего ключа реестра
Registry key <name> could not be removed	Вызов метода `RegDelete` для несуществующего ключа реестра
Remote script object can only be executed once	Попытка повторно запустить объект — удаленный сценарий
Shortcut <name> contains invalid syntax	Сохранение ярлыка на сетевой ресурс, имеющий некорректный URL
Shortcut <name> could not be saved	Попытка сохранить новый ярлык в файле, который уже существует и имеет атрибут "Только для чтения"
Shortcut <name> failed to execute protocol handler	Попытка установить ярлык на сетевой ресурс, использующий несуществующий обработчик протокола
Unable to execute remote script	Невозможно создать процесс — удаленный сценарий
Unable to find job <job identifier>	В WS-файле нет задания с идентификатором `<job identifier>`
Unable to wait for process	С помощью метода `Run` дано указание ожидать завершение процесса, которое из сценария определить нельзя

Приложение 5
Описание прилагаемой дискеты

Большинство примеров сценариев, которые приведены в книге, содержатся на прилагаемой дискете. Примеры находятся в папках, названных в соответствии с нумерацией глав, к которым они относятся: \Chapter01, \Chapter02, …, \Chapter11 (табл. П5.1).

Таблица П5.1. *Структура дискеты*

Папки	Содержание
\Chapter01	JScript- и VBScript-сценарии, которые иллюстрируют использование стандартных объектов WSH 5.6
\Chapter02	JScript- и VBScript-сценарии, которые иллюстрируют использование стандартных объектов WSH 5.6
\Chapter03	WS-файлы, в которых используются возможности XML-разметки
\Chapter04	Обычные (js, vbs и wsf) и зашифрованные (jse, vbe) сценарии, а также сценарии с цифровой подписью. Кроме этого, приведен пример административного шаблона wsh.adm, позволяющий запрещать/разрешать выполнение локальных или удаленных сценариев
\Chapter05	JScript-сценарии для работы с файловой системой и телефонной записной книжкой в текстовом файле book.txt
\Chapter06	JScript-сценарии для работы с телефонной записной книжкой в XML-файле book.xml
\Chapter07	JScript- и WS-файлы, с помощью которых организуются различные типы пользовательского интерфейса для работы с записной книжкой в XML-файле book.xml
\Chapter08	JScript- и WS-файлы, с помощью которых данные из записной книжки book.xml выводятся в файлы Microsoft Word (в том числе с использованием шаблона Table.dot) и Excel
\Chapter09	JScript- и WS-файлы, которые позволяют работать с записной книжкой в виде DBF-таблицы Phone.dbf, данные в которую копируются из XML-файла book.xml

\Chapter10	Примеры COM-объектов, написанных на языках JScript и VBScript (wsc-файлы), и JScript-сценарии RunArj.exe, использующий один из этих объектов
\Chapter11	JScript-сценарии, которые выводят различные системные диалоговые окна, а также иллюстрируют применение технологий ADSI и WMI для решения задач администрирования. Также приведены bat-файлы, которые можно использовать в качестве сценариев входа/выхода

Для использования примеров нужно скопировать соответствующий каталог с дискеты на жесткий диск, после чего можно просматривать, редактировать и запускать нужные сценарии.

Следует учесть, что перед запуском сценариев из папки \Chapter09 требуется предварительно настроить источник данных ODBC с именем PhoneDS *(см. главу 9),* а компоненты-сценарии из папки \Chapter10 следует зарегистрировать в системе *(см. главу 10).* Кроме этого, для корректной работы сценария \Chapter10\RunArj.exe необходимо наличие архиватора arj.exe.

Источники информации

Список литературы

1. Андерсон К. Сценарии Windows и управление системой // Windows 2000 Magazine/RE. 2002. № 5 (http://www2.osp.ru/win2000/2002/05/053.htm).

2. Андерсон К. Сценарии WMI для начинающих // Windows 2000 Magazine/RE. 2001. № 5 (http://www2.osp.ru/win2000/2001/05/070.htm).

3. Борн Г. Руководство разработчика на Microsoft Windows Script Host 2.0. Мастер-класс: Пер. с англ. — СПб.: Питер; М.: Издательско-торговый дом "Русская редакция", 2001. — 480 с.

4. Грабер М. Введение в SQL: Пер. с англ. — М.: Лори, 1996. — 379 с.

5. Кокорева О. И. Реестр Windows XP. — СПб.: БХВ-Петербург, 2002. — 560 с.

6. Корнелл Г. Сценарии Windows для работы с файлами // PC Magazine. 1998. № 9 (http://www.pcmag.ru/archive/9805/099824.asp).

7. Мар-Элиа Д. Дополнительные рычаги управления Windows 2000 // Windows 2000 Magazine/RE. 2000. №5 (http://www2.osp.ru/win2000/2000/05/009.htm).

8. Попов А. В. Командные файлы и сценарии Windows Script Host. — СПб.: БХВ-Петербург, 2002. — 320 с.

9. Рубенкинг Н. Дж. Сценарии Windows Scripting Host // PC Magazine/RE. 2001. № 6 (http://pcmag.ru/?ID=35954).

10. Уэллс Б. Extensible Markup Language. Роль языка XML в Windows Scripting Host 2.057 // Windows 2000 Magazine/RE. 2000. № 3 (http://www2.osp.ru/win2000/2000/03/056.htm).

11. Уэллс Б. Инструменты управления Windows: помощник системного администратора // Windows 2000 Magazine/RE. 2000. № 6 (http://www2.osp.ru/win2000/2000/06/073.htm).

12. Уэллс Б. Основы WSH // Windows 2000 Magazine/RE. 1999. № 2 (http://www2.osp.ru/win2000/1999/02/14.htm).

13. Уэллс Б. Регистрационные сценарии WSH // Windows 2000 Magazine/RE. 1999. № 1 (http://www2.osp.ru/win2000/1999/01/55.htm).

14. Уэллс Б. Сценарии для Active Directory. Часть 1 // Windows 2000 Magazine/RE. 2001. №6 (http://www2.osp.ru/win2000/2001/06/060.htm).

15. Уэллс Б. Сценарии для Active Directory. Часть 2 // Windows 2000 Magazine/RE. 2001. № 7 (http://www2.osp.ru/win2000/2001/07/064.htm).

16. Уэллс Б. Файлы Windows Script в действии // Windows 2000 Magazine/RE. 2000. № 4 (http://www2.osp.ru/win2000/2000/04/063.htm).

17. Харт-Девис Г. Word 2000. Руководство разработчика: Пер. с англ. — Киев: Издательская группа BHV, 2000. — 944 с.

18. Экк Т. Сценарии ADSI для системного администрирования Windows NT/2000: Пер. с англ. — М. — СПб. — Киев: Издательский дом "Вильямс", 2000. - 576 с.

19. Aitken P. G. Windows Script Host. — Prentice Hall PTR, 2001. — 384 p.

20. Borge S. Managing Enterprise Systems with the Windows Script Host. — Apress, 2001. — 950 p.

21. Born G. Advanced Development with Microsoft Windows Script Host. — Microsoft Press, 2001 — 450 p.

22. Esposito D. Windows Script Host Programmer's Reference. — Wrox Press, 1999. — 373 p.

23. Fredel T. SAMS Teach Yourself Windows Scripting Host in 21 days. — SAMS, 1999 — 624 p.

24. Hill T. Windows Script Host. — New Riders Publishing, 1999. — 430 p.

25. Meggitt A., Lavy M.M. Windows Management Instrumentation (WMI). — New Riders Publishing, 2001. — 432 p.

26. Policht M. WMI Essentials for Automating Windows Management. — SAMS, 2001. — 624 p.

27. Stanek W. R. Windows 2000 Scripting Bible. — Hungry Minds, 2000. — 667 p.

28. Weltner T. Windows Scripting Secrets. — Hungry Minds, 2000. — 751 p.

29. Windows Script Host для входа в систему // Lan/Журнал сетевых решений. 1999. № 11 (http://www.osp.ru/lan/1999/11/005.htm).

Ссылки на ресурсы Internet

Журналы и статьи

Адрес	Описание
http://msdn.microsoft.com/msdnmag/default.asp	Журнал "MSDN Magazine". Публикуются статьи по различным технологиям Microsoft
http://msdn.microsoft.com/library/defaultasp?url=/library/en-us/dnclinic/html/vbsvjs.asp	"MSDN Online Voices". Здесь находятся статьи разработчика Microsoft Эндрю Клиника (Andrew Clinick), посвященные сценариям WSH
http://www.ddj.com/topics/altlang/	Журнал "Dr. Dobb's Journal", раздел "Scripting and Alternative Languages Discussion Forum"
http://www.win32scripting.com/	Журнал "Windows Scripting Solutions". Освещаются различные аспекты использования сценариев и командных файлов в Windows

Сайты компании Microsoft

Адрес	Описание
http://msdn.microsoft.com/scripting/	Сайт Microsoft Windows Script Technologies, посвященный ActiveX-сценариям. Отсюда можно скачать последнюю версию WSH, документацию по

	WSH, WSC, JScript, VBScript, отладчики сценариев
http://msdn.microsoft.com/developer/default.htm	Электронная библиотека MSDN (Microsoft Developer Network) содержит подробную информацию об объектах автоматизации, которые может использовать WSH, в том числе об объектах ADO, ADSI и WMI
microsoft.public.scripting.wsh	Телеконференция Microsoft, посвященная WSH

Зарубежные сайты

Адрес	Описание
http://communities.msn.com/windowsscript/	Документация, статьи, примеры сценариев, ответы на часто задаваемые вопросы (Frequently Asked Questions, FAQ)
http://scripting.winguides.com/	Документация и примеры сценариев JScript и VBScript
http://www.winscripter.com/	Статьи, примеры сценариев, ссылки на сайты схожей тематики
http://www.borncity.de/WSHBazaar/WSHBazaar.htm	Сайт Г. Борна, автора нескольких книг по WSH
http://www.netspace.net.au/~torrboy/code/jscriptfaq/	Часто задаваемые вопросы по языку JScript
	Модули для WSH, поддерживающие языки

http://www.activestate.com	Active Perl, Active Python, Active XSLT

Российские сайты

Адрес	Описание
http://www.scripting.vlink.ru	Сайт посвящен использованию языков сценариев VBScript и JScript, их расширенному применению с использованием ActiveX-элементов. Здесь помещена книга А.В. Неверова "Windows Scripting Host 2.0 (с использованием MS Visual Basic Script и MS JScript)"
http://script.net.ru	Начальные сведения о WSH
http://www.webhowto.ru/reg/reg_t1.shtml	Теория WSH, примеры сценариев для работы с файловой системой и системным реестром
http://scripting.narod.ru	Часто задаваемые вопросы по WSH, избранные статьи из Microsoft Knowledge Base, статьи о сценариях, ссылки на сайты, посвященные сценариям