

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
Федеральное государственное бюджетное
образовательное учреждение
высшего профессионального образования
ОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
им. Ф.М. ДОСТОЕВСКОГО

Д.М. Бречка

**ОПЕРАЦИОННЫЕ СИСТЕМЫ.
ЧАСТЬ 2.
WINDOWS SCRIPT HOST**

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

ОМСК-2012

УДК 004.451
ББК 32.973.2
Б877

*Рекомендовано к изданию учебно-методической комиссией и ученым
советом факультета компьютерных наук ОмГУ*

Рецензенты

канд. тех. наук, доц. Д.Н. Лавров
канд. тех. наук Ю.С. Ракицкий

Бречка, Д.М.

Б877 Операционные системы. Часть 2. Windows Script Host: учебно-методическое пособие / Д.М. Бречка. – Омск: Изд-во Ом. гос. ун-та, 2012 – 108с.

ISBN

Рассматриваются вопросы управления компьютером с помощью инструмента Windows Script Host. Приводятся примеры выполнения базовых действий в операционных системах Windows.

Соответствует государственным образовательным стандартам высшего профессионального образования по специальностям 230101.65 «Вычислительные машины, комплексы системы и сети», 090102.65 «Компьютерная безопасность» и направлениям подготовки бакалавриата 090900.62 «Информационная безопасность», 230100 «Информатика и вычислительная техника».

Для студентов вышеперечисленных специальностей.

**УДК 004.451
ББК 32.973.2**

© Бречка Д.М., 2012
© ФГБОУ ВПО «ОмГУ им. Ф.М. Достоевского», 2012

1. Введение в Windows Script Host. Основы JScript

1.1. Теоретический блок

1.1.1. Windows Script Host

С помощью 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 Windows Script Host (WSH), 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.

Возможности технологии ActiveX

Windows с самого начала для обеспечения обмена данными между приложениями была разработана технология связывания и внедрения объектов (Object Linking and Embedding, OLE). 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 Component Object Model (COM). 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 термин OLE используется только по историческим причинам. Вместо него Microsoft с 1996 года использует новый термин — ActiveX, первоначально обозначавший WWW (World Wide Web) компоненты (объекты), созданные на базе технологии COM.

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 [2].

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.

1.1.2. Создание и запуск простейших сценариев WSH

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".

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!". Для этого 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 этих двух 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 виде комментариев, может выглядеть следующим образом:

```
/* **** */
/* Имя: First.js */
/* Язык: JScript */
/* Описание: Вывод на экран приветствия */
/* **** */
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 [1].

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. 1.1).



Рис. 1.1. 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. 1.2).

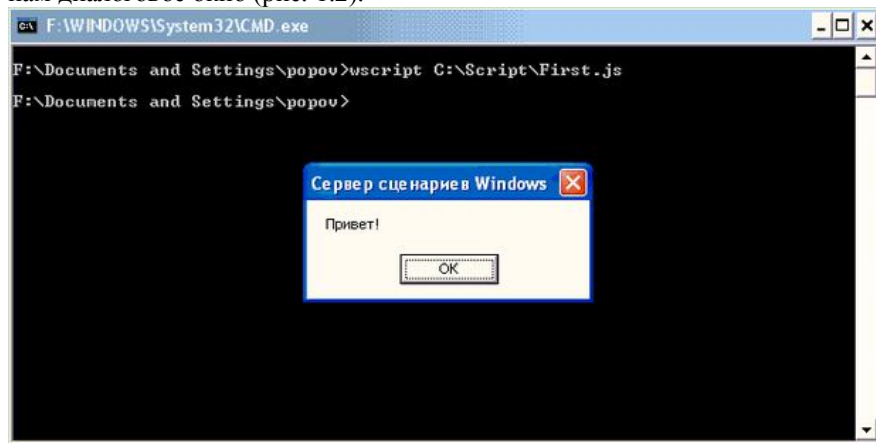


Рис. 1.2. Результат выполнения `First.js` в графическом режиме (`wscript.exe`)

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.

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)* и нажать кнопку *Ok*. 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 (аналогично любому другому исполняемому файлу). 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.

Если сценарий запускается в консольном режиме, то его исполнение контролируется с помощью параметров командной строки для *cscript.exe* (см. табл. 1.1), которые включают или отключают различные опции WSH (все эти параметры начинаются с символов *"/"*).

Например, команда

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

запустит сценарий *First.js* без информации о версии WSH.

Сценарий можно запускать с параметрами командной строки, которые указываются после имени этого сценария. 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*.

Если сценарий запускается в графическом режиме (с помощью *wscript.exe*), то свойства сценария можно устанавливать с помощью вкладки *Сценарий (Script)* диалогового окна, задающего свойства файла в Windows (рис. 1.3).

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.

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.

Таблица 1.1. Параметры командной строки для 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	Задаёт 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 ниже)
//U	Позволяет использовать при перенаправлении ввода-вывода с консоли кодировку Unicode

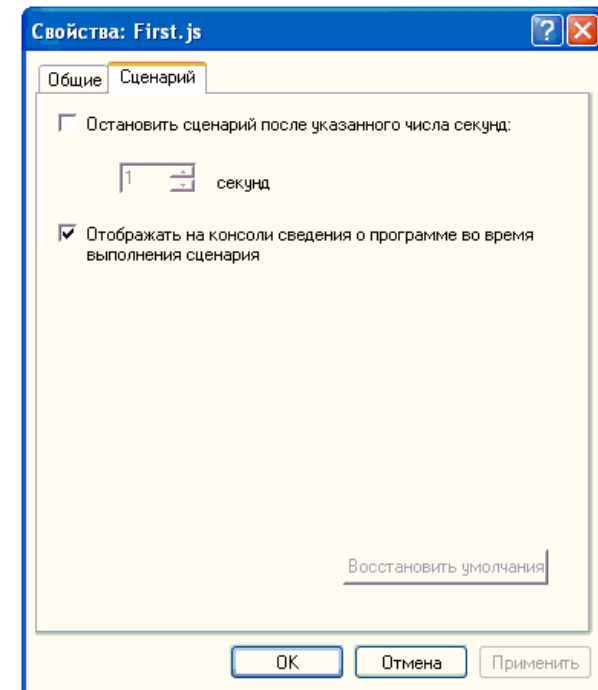


Рис. 1.3. Установка свойств сценария First.js

1.1.3. Язык JScript

Подобно многим другим языкам программирования, код на Microsoft *JScript* пишется в текстовом формате, и организован в инструкции, блоки, состоящие из связанных наборов инструкций, и комментариев. В пределах инструкции Вы можете использовать переменные и данные, такие как строки, числа и выражения [1,2].

Инструкции

Код JScript-инструкции состоит из одного или более символов в строке. Новая строка начинает новую инструкцию, но хорошим стилем является объявление конца инструкции явно. В JScript для этого используется точка с запятой (;).

```
aBird = "Robin";  
var today = new Date();
```

Группа JScript-инструкций, заключенная в фигурные скобки ({}), называется блоком. Блоки инструкций используются как функции и условные выражения. В следующем примере, первая инструкция

определяет функцию, которая состоит из блока пяти инструкций. Последние три инструкции, которые не окружены фигурными скобками - блоком не являются.

```
function convert(inches) {  
    feet = inches / 12; // Эти пять инструкций - блок.  
    miles = feet / 5280;  
    nauticalMiles = feet / 6080;  
    cm = inches * 2.54;  
    meters = inches / 39.37;  
}  
km = meters / 1000; // Эти инструкции блоком не являются.  
kradius = km;  
mradius = miles;
```

Комментарии

Комментарием в JavaScript является текст, расположенный после двойного слэша (//) до конца строки. Многострочный комментарий начинается слэшем со знаком умножения (*), и кончается их обратной комбинацией (*).

```
aGoodIdea = "Comment your code thoroughly."; // Однострочный  
комментарий.  
/* Это многострочный комментарий. */
```

Присваивание и равенство

Знак равенства (=) используется в JavaScript как присваивание. Следующий код

```
anInteger = 3;
```

подразумевает "Присвоить значение 3 переменной anInteger," или "anInteger принимает значение 3." При сравнении двух значений на равенство применяется двойной знак равенства (==).

Выражения

JavaScript выражения можно разделить на Логические или Числовые. Выражения содержат некоторые особенности, к примеру, символ "+" означает "добавить к...". Любая допустимая комбинация значений, переменных, операторов, и других выражений является выражением.

```
var anExpression = "3 * (4 / 5)";  
var aSecondExpression = "Math.PI * radius * 2";  
var aThirdExpression = aSecondExpression + "%" + anExpression;  
var aFourthExpression = "(" + aSecondExpression + ")" % (" +  
anExpression + ")";
```

Переменные

Переменные используются в Microsoft JScript для присваивания значений в сценариях. Для различия переменных, им присваивают имена.

Объявление переменных

Необязательно, но считается хорошим стилем программирования, объявление переменной перед использованием. Это делается с помощью инструкции **var**. Вы обязаны применять инструкцию **var** при объявлении локальной (local) переменной внутри функции. В остальных случаях объявление инструкции **var** перед применением в сценариях рекомендуется.

Примеры объявления переменных:

```
var mim = "A man, a plan, a canal, Panama!"; // Строковый тип
var ror = 3; // Целый числовой тип.
var nen = true; // Boolean или логический тип.
var fif = 2.718281828 // Числовой тип.
```

Имя переменной

JScript различает регистр в имени переменной: *myCounter* отличается от *MYCounter*. На практике присваивания имен требуется соблюдать следующие правила:

- Первым символом может быть буква любого регистра, или символ подчеркивания (), или знак доллара (\$).
- Следующими символами могут быть буквы, символы подчеркивания, цифры и знаки доллара.
- Именами переменных не могут служить зарезервированные слова.

Несколько примеров правильных имен:

- *_pagecount*
- *Part9*
- *Number_Items*

Некоторые неправильные имена:

- *99Balloons* // Первый символ - цифра.
- *Smith&Wesson* // Амперсанд(&) не разрешено применять в именах.

Если необходимо объявить и инициализировать переменную, но без присваивания определенного значения, можно применить значение **null**.

```
var zaz = null;
var notalot = 3 * zaz; // В данном случае notalot равен 0.
```

Если переменная объявлена, а значение не присвоено, она существует, но значение не определено - *undefined*.

```
var godot;  
var waitingFor = 1 * godot; // waitingFor имеет значение NaN, так как  
значение переменной godot не определено.
```

Разрешается объявление переменной неявно - без инструкции **var**. Однако, в выражениях применять необъявленные переменные не допускается.

```
let = ""; // Переменная let объявлена неявно.  
var aMess = yuv + zez; // Ошибка, так как yuv and zez не существуют.
```

Преобразование типов

Так как JavaScript - язык с нестрогим контролем типов, переменные в JavaScript не имеют строго фиксированного типа. Переменные имеют тип, эквивалентный типу значения, которое они содержат. Однако, в некоторых случаях, необходимо принудительное преобразование переменной в определенный тип. Числа могут быть объявлены как строки, а строки необходимо преобразовать в числовой тип. Для этого применяют конверсионные функции **parseInt()** и **parseFloat()**.

```
var theFrom = 1;  
var theTo = 10;  
var doWhat = "Count from ";  
doWhat += theFrom + " to " + theTo + " ";
```

После выполнения кода, переменная *doWhat* принимает значение "Count from 1 to 10." Числовой тип преобразовывается в строковый тип.

```
var nowWhat = 0;  
nowWhat += 1 + "10"; // В этом случае, "10" является строкой,  
// "+" - оператор конкатенации.
```

После исполнения кода, переменная *nowWhat* принимает значение "0110". Нижеследующее объясняет выполнение кода:

1. Посмотрите на типы 1 и "10". "10" - строковый, а 1 - числовой, поэтому число было преобразовано в строку.

2. Оператор + над строками, является оператором конкатенации. Результатом является "110".

3. Посмотрите на типы по обе стороны оператора +=. *nowWhat* включает число, и "110" - строку, и поэтому число преобразуется в строку.

4. В данный момент по обе стороны оператора += находятся строки, происходит конкатенация строк. Результатом является "0110".

5. Результат хранится в переменной *nowWhat*.

```
var nowThen = 0;
```

nowThen += 1 + parseInt("10"); // В данном случае, "+=" является оператором сложения

После выполнения кода, переменная *nowThen* принимает значение 11.

Операторы JScript

В JScript применяются множество операторов: арифметические, логические, разрядные, присваивания и прочие. Операторы JScript представлены в таблице 1.2 и 1.3.

Таблица 1.2. Операторы JScript

Вычислительные		Логические		Разрядные	
Название	Сим вол	Название	Сим вол	Название	Сим вол
Унарный минус	-	Логическое НЕ	!	Разрядное НЕ	~
Инкремент	++	Меньше	<	Поразрядный левый сдвиг	<<
Декремент	--	Больше	>	Поразрядный правый сдвиг	>>
Умножение	*	Меньше или равно	<=	Беззнаковый поразрядный правый сдвиг	>>>
Деление	/	Больше или равно	>=	Разрядное И	&
Деление по остатку	%	Равно	==	Разрядное ИСКЛЮЧАЮЩЕЕ ИЛИ	^
Сложение	+	Не равно	!=	Разрядное ИЛИ	

Таблица 1.2. Операторы JScript (продолжение)

Вычислительные		Логические		Разрядные	
Название	Сим вол	Название	Сим вол	Название	Сим вол
Вычитание	-	Логическое И	&&		
		Логическое ИЛИ			
		Условное выражение (Тринар)	?:		
		Запятая	,		
		Тождественно	===		
		Нетожественно	!==		

Таблица 1.3. Операторы JScript

Присваивания		Прочие	
Название	Символ	Название	Символ
Присваивание	=	Удаление	delete
Составное присваивание	OP=	Тип	typeof
		Пусто	void

Приоритет операторов

В JScript операторы выполняются в определенном порядке, называемом приоритет операций. Следующий список отражает приоритет операторов от высшего к низшему (таблица 1.4). Операторы, указанные в одной строке, выполняются слева направо.

Таблица 1.4. Приоритет операций в JScript

Приоритет	Оператор	Описание
1	. [] ()	Точка, индексы массивов, вызов функции
2	++ -- ~ ! typeof new void delete	Унарные операции, вывод типов данных, создание объектов, неопределенные значения
3	* / %	Умножение, деление, деление по остатку
4	+ - +	Сложение, вычитание, конкатенация строк
5	<< >> >>>	Поразрядные сдвиги
6	< <= > >=	Меньше, меньше или равно, больше, больше или равно
7	== != === !==	Равно, неравно, тождественно, нетождественно
8	&	Разрядное И
9	^	Разрядное ИСКЛЮЧАЮЩЕЕ ИЛИ
10		Разрядное ИЛИ
11	&&	Логическое И
12		Логическое ИЛИ
13	?:	Условное выражение
14	= OP=	Операторы присваивания
15	,	Запятая

Круглые скобки используются, чтобы изменить порядок выполнения операторов. Выражение в круглых скобках полностью вычисляется прежде, и его значение используется как остаточный член инструкции.

Оператор с более высоким приоритетом выполняется ранее оператора с низким приоритетом. Например:

$z = 78 * (96 + 3 + 45)$

В данном выражении пять операторов: =, *, (), +, и +. Приоритет операторов - следующий: (), *, +, +, =.

1. Первым вычисляется значение выражения в круглых скобках: К сумме операции 96 и 3 прибавляется 45, общая сумма равна 144.
2. Далее выполняется умножение: произведение 78 и 144 дает результат 11232.
3. Переменной z присваивается значение 11232 .

Управление ходом программы

Существует несколько видов проверки условий. Все условные выражения в Microsoft JScript - логические, поэтому результат их проверки равен либо **true**, либо **false**. Вы можете проверять значения логического, числового строкового типов данных.

В JScript простейшими структурами управления являются условные выражения.

Использование условных выражений

В JScript поддерживаются условные выражения **if** и **if...else**. В выражении **if** проверяется условие, при соответствии этому условию, выполняется написанный разработчиками JScript-код. (В выражении **if...else** выполняется при несоответствии условию другой код.) Простейшая форма оператора **if** может быть написана одной строкой, Но обычно операторы **if** и **if...else** записываются в несколько строк.

Следующий пример демонстрирует синтаксис выражений **if** и **if...else**. В первом примере - простейшая логическая проверка. Если выражение в круглых скобках равно **true**, выражение или блок выражений после **if** выполняется.

```
// Функция smash() определена в другом месте кода.  
if (newShip)  
    smash(champagneBottle,bow); // Логическая проверка newShip на  
    равенство true.
```

```
// В данном примере, условие выполняется, если оба подусловия  
равны true.  
if (rind.color == "ярко-желтый " && rind.texture == "большие и  
малые пятна")  
{  
    theResponse = ("Это тыква? <br> ");  
}
```

```
// В следующем примере, условие выполнится, если хотя бы одно из  
подусловий (или оба) равны true.  
var theReaction = "";  
if ((lbsWeight > 15) || (lbsWeight > 45))
```



```

{
  theReaction = ("Ненлохо! <br>");
}
else
  theReaction = ("He очень! <br>");

```

Условный оператор

В JScript также поддерживается упрощенная форма условия. Это использование вопросительного знака после условия для проверки (подобно **if** до условия), и двух альтернативных выражений, одно применяется при успешной проверке, другое - при не соответствии условию. Альтернативные выражения разделяются двоеточием.

```
var hours = "";
```

```
// Код для проверки времени
// theHour, or theHour - 12.
```

```
hours += (theHour >= 12) ? " PM" : " AM";
```

Циклы

Существует несколько вариантов выполнения инструкции или блоков инструкций неоднократно. Повторное выполнение называется циклом. Цикл обычно управляется исходя из значения некоторых условий и(ли) переменных, значения которых меняется в конце каждого цикла. В Microsoft JScript применяется несколько типов циклов: **for**, **for...in**, **while**, **do...while** и **switch**.

Применение цикла for

В выражении **for** определены переменная-счетчик, условие проверки и действие, изменяющее значение счетчика. После каждого выполнения цикла (это называется итерацией цикла), проверяется условие, производится обработка выполнения действий на переменной-счетчиком, и если условия проверки выполнены, выполняется новая итерация цикла.

Если условие для выполнения цикла никогда не будет выполнено, цикл никогда не завершится. Если проверяемое условие всегда выполняется, цикл бесконечен. Поэтому разработчикам следует позаботиться об этом.

```
/*Изменение выражения (" icount ++ " в приведенных примерах)
выполняется в конце цикла, после блока инструкций,
прежде, чем условие проверено.*/
```

```

var howFar = 11; // Ограничение выполнения цикла - 11 итераций.
var sum = new Array(howFar); // Создание массива размер - 11,
                               //индексы от 0 по 10.

var theSum = 0;
sum[0] = 0;
for(var icount = 1; icount < howFar; icount++) { // Счетчик от 1 до
                                                // 10 в данном случае.

    theSum += icount;
    sum[icount] = theSum;
}
var newSum = 0;
for(var icount = 1; icount > howFar; icount++) { // Цикл не будет
                                                // выполнен.

    newSum += icount;
}
var sum = 0;
for(var icount = 1; icount > 0; icount++) { // Бесконечный цикл.
    sum += icount;
}

```

Применение цикла for...in

В JScript используется специальный цикл для пошагового обхода свойств объекта. Счетчик цикла **for...in** проходит все индексы массива. Это строковый тип, а не числовой.

```

for (j in tagliatelleVerde) // tagliatelleVerde является объектом с
//несколькими свойствами
{
    // Код JScript.
}

```

Применение цикла while

Цикл **while** напоминает цикл **for**. Различие в том, что в цикле **while** отсутствует встроенная переменная-счетчик, а, следовательно, и условия ее изменения. Выражения над условием для выполнения цикла **while** находятся внутри цикла в блоке инструкций.

```

var theMoments = "";
var theCount = 42; // Инициализация переменной-счетчика.
while (theCount >= 1) {
    if (theCount > 1) {
        theMoments = "Осталось " + theCount + " секунд!";
    }
    else {

```

```

        theMoments = "Осталась одна секунда!";
    }
    theCount--;    // изменение счетчика.
}
theMoments = "Время истекло!";

```

Применение выражений **break** и **continue** в циклах

В Microsoft JScript существует инструкция остановки выполнения цикла. Оператор завершения **break** может использоваться, чтобы остановить цикл, при выполнении какого-либо условия. Инструкция **continue** используется, чтобы немедленно перейти к выполнению следующей итерации, пропуская остальную часть выполнения кода текущей итерации, но обновляя переменную-счетчик как в обычных циклах **for** или **for...in**.

```

var theComment = "";
var theRemainder = 0;
var theEscape = 3;
var checkMe = 27;
for (kcount = 1; kcount <= 10; kcount++)
{
    theRemainder = checkMe % kcount;
    if (theRemainder == theEscape)
    {
        break; // выход при выполнении условия (theRemainder ==
//theEscape).
    }
    theComment = checkMe + " divided by " + kcount + " leaves a remainder
of " + theRemainder;
}
for (kcount = 1; kcount <= 10; kcount++)
{
    theRemainder = checkMe % kcount;
    if (theRemainder != theEscape)
    {
        continue; // При неравенстве theRemainder и theEscape переходим
//к следующей итерации.
    }
}
// JScript код.
}

```

Функции в JScript

Функции в Microsoft JScript выполняют определенные действия. Они могут возвращать некоторый результат, например результат вычислений или сравнения

Функции исполняют при вызове определенный блок инструкций. Это позволяет однажды определить функцию, а в дальнейшем вызывать ее когда потребуется.

Вы передаете данные функции, включая их в круглые скобки после имени функции. Данные в круглых скобках называются параметрами. В некоторых функциях нет параметров вообще; в некоторых - один параметр; иногда параметров несколько.

В JScript имеется два вида функций: встроенные и определяемые.

Специальные встроенные функции

Язык JScript включает несколько встроенных функций. Некоторые из них позволяют обрабатывать выражения и специальные символы, и преобразовывать строки в числовые значения.

Например, **escape()** и **unescape()** используются для перевода кода HTML и спецсимволов, то есть символов, которые нельзя непосредственно размещать в тексте. К примеру, символы "<" и ">", применяются для обозначения HTML-тэгов.

Функция **escape** принимает в параметре спецсимволы, а возвращает ESC-код символа. Каждый код ESC-код начинается со знака процента (%) и последующим двухзначным числом. Функция **unescape** является инверсионной. В качестве параметра вводится ESC-код, а возвращает символ.

Еще одна встроенная функция - **eval()**, которая выполняет любое правильное математическое выражение, представленное в виде строки. Функция **eval()** имеет один аргумент - выражение для выполнения.

```
var anExpression = "6 * 9 % 7";  
var total = eval(anExpression);    // Вычисляет выражение, равно 5.  
var yetAnotherExpression = "6 * (9 % 7)";  
total = eval(yetAnotherExpression) // Вычисляет выражение, равно 12.  
var totality = eval("Текст.");     // Возвращает ошибку.
```

Создание собственных функций

Вы можете создавать собственные функции и вызывать их, когда потребуется. Определение функции состоит из объявления параметров и блока инструкций JScript.

Функция *checkTriplet* в следующем примере принимает в качестве параметров длины трех сторон треугольника, и определяет, является ли треугольник прямоугольным, проверяя согласно Пифагорову правилу

(триплету). (Квадрат длины гипотенузы прямоугольного треугольника равен сумме квадратов длин катетов.) Функция *checkTriplet* в ходе проверки вызывает одну из двух других функций.

При проверке значений при входных значениях чисел с плавающей точкой используется "машинное epsilon" ("epsilon") - очень маленькое число. Из-за погрешности округления в вычислениях с плавающей точкой, без применения этого числа, функция может возвращать неверный результат. "Машинное epsilon" является максимальной допустимой погрешностью.

```
var epsilon = 0.0000000000000001; // "Машинное epsilon".
var triplet = false;
function integerCheck(a, b, c) { // Функция проверки при целых числах.
  if ( (a*a) == ((b*b) + (c*c)) ) { // Код проверки.
    triplet = true;
  }
} // Конец функции проверки при целых числах.
function floatCheck(a, b, c) { // Функция проверки при числах с
//плавающей точкой.
var theCheck = ((a*a) - ((b*b) + (c*c))) // Контрольное число.
  if (theCheck < 0) { // Вычисление модуля контрольного числа.
    theCheck *= -1;
  }
  if (epsilon > theCheck) { // Сравнение отклонения с допустимой
//погрешностью!
    triplet = true;
  }
} // Конец функции проверки при числах с плавающей точкой.
function checkTriplet(a, b, c) { // Проверка триплета. Первым делом,
// присвоим "a" наибольшее значение из трех входных чисел.
var d = 0; // Временная переменная.
  if (c > b) { // При c > b, меняем местами.
    d = c;
    c = b;
    b = d;
  } // Иначе - не меняем.
  if (b > a) { // При b > a, меняем местами.
    d = b;
    b = a;
    a = d;
  } // Иначе - не меняем.
  // Сторона "a" является гипотенузой.
```

```

if (((a%1) == 0) && ((b%1) == 0) && ((c%1) == 0)) { // Проверка
// чисел - целые ли они?
integerCheck(a, b, c); // Функция для целых чисел.
}
else
floatCheck(a, b, c); // Иначе вызываем функцию для чисел с
//плавающей точкой
} // Окончание проверки правила Пифагора (триплета).
// Практическая проверка - объявляем три переменные.
var sideA = 5;
var sideB = 5;
var sideC = Math.sqrt(50);
checkTriplet(sideA, sideB, sideC); // Функция проверки на триplet.

```

Объекты JScript

В Microsoft JScript объекты по сути являются совокупностями методов и свойств. Метод - функция, которая выполняется внутри объекта, а свойство - значение или набор значений (в виде матрицы или другого объекта), являющееся частью объекта. В JScript объекты можно разделить на три вида: встроенные, созданные и браузерные.

В JScript, обработка объектов и массивов идентична. Вы можете обратиться к любой части объекта (его свойствам и методам) либо по имени, либо по индексу. Нумерация индексов в JScript начинается с нуля. Для удобства работы, частям можно присвоить имена.

Существует несколько вариантов обращений. Следующие выражения эквивалентны.

```

theWidth = spaghetti.width;
theWidth = spaghetti[3]; // [3] является индексом "width".
theWidth = spaghetti["width"];

```

При использовании числовых индексов, в обращении по имени нельзя использовать точку (.). Следующее выражение вызывает ошибку.

```
theWidth = spaghetti.3;
```

В случаях, когда объект является свойством другого объекта, обращение образуется прямым путем.

```

var init4 = toDoToday.shoppingList[3].substring(0,1); // массив
// shoppingList - свойство toDoToday.

```

Так как объекты могут быть свойствами других объектов, возможно создание массивов с более чем одним измерением, которые непосредственно не поддерживаются. Следующий код создает таблицу умножения для значений от 0 до 16.

```

var multTable = new Array(17); // Создание оболочки для таблицы
for (var j = 0; j < multTable.length; j++) { // Подготовка к заполнению

```

```

//строками
var aRow = new Array(17); // Создание строки
for (var i = 0; i < aRow.length; i++) { // Подготовка к заполнению
// строки
aRow[i] = (i + " times " + j + " = " + i*j); // Создание и размещение
//одного значения
}
multTable[j] = aRow; // Заполнение таблицы строкой
}

```

Обращение к каждому элементу производится указанием нескольких индексов

```

var multiply3x7 = multTable[3][7];
Следующее выражение вызовет ошибку.
var multiply3x7 = multTable[3, 7];

```

Зарезервированные слова JScript

В JScript имеется ряд зарезервированных ключевых слов (таблица 1.5). Эти слова разделяются на три типа: зарезервированные ключевые слова JScript, слова, зарезервированные для будущих версий, и слова, которые следует избегать (таблица 1.6).

Таблица 1.5. Ключевые слова JScript

Break	false	in	this	void
Continue	for	new	true	while
Delete	function	null	typeof	with
Else	if	return	var	

Таблица 1.6. Зарезервированные слова для будущих версий JScript

Case	debugger	export	super
Catch	default	extends	switch
Class	do	finally	throw
Const	enum	import	try

К словам, которых следует избегать, относятся имена уже существующих встроенных объектов и функций. Например, в языке уже применяются слова *String* и *parseInt*.

Применение любого из ключевых слов первых двух типов вызовет ошибку трансляции, когда ваш сценарий начнет выполняться.

Использование зарезервированных слов третьего вида может вызвать непредсказуемое исполнение сценария, если Вы попытаетесь применить вашу переменную и вызвать объект с тем же именем в одном сценарии. Например, следующий сценарий не выполняет того, что нужно:

```
var String = "Новая строка";  
var text = new String("Эта строка - новый экземпляр объекта");
```

В этом случае получим ошибку, выводящую информацию о том, что строка не является объектом.

Рекурсия

Рекурсия - важная методика программирования. Она используется, чтобы иметь обращение к функции непосредственно изнутри себя. Наглядный пример - вычисление факториалов. Факториалы 0 и 1 оба определены заранее, оба равны 1. Факториалы больших чисел рассчитываются так: умножаются $1 * 2 * \dots$, увеличивая множитель на каждый цикл на единицу, пока не достигнет номера, для которого вычисляется факториал.

В данном параграфе опишем функцию, вычисляющую факториал.

"Если число - меньше чем ноль, выход. Если это не целое число, округляем до меньшего целого числа. Если число - ноль или единица, факториал - 1. Если число больше единицы, умножаем его на факториал числа, меньшего на единицу".

Чтобы вычислить факториал любого числа, которое больше единицы, необходимо вычислить факториал по крайней мере одного другого числа. Функция, которую Вы используете, должна вызвать саму себя для вычисления факториала меньшего числа, до тех пор, пока факториал какого либо числа не будет вычислен. Это - пример рекурсии

Понятно, этот способ имеет недостатки. Вы можете легко создать рекурсивную функцию, которая никогда не примет конечный или определенный результат, и не достигнет конечной точки. Такая рекурсия заставляет компьютер выполнять так называемый "бесконечный" цикл. Пример: удалим первое правило (относительно вычисления факториала отрицательных чисел) из описания вычисления факториала, и попробуем вычислить факториал любого отрицательного числа. Эта попытка потерпит неудачу, потому что, чтобы вычислять факториал, скажем, -24 Вы сначала должны вычислить факториал -25; но чтобы делать это Вы сначала должны вычислить факториал -26; и так далее. Очевидно, что функция никогда не достигнет промежуточного определенного результата.

Таким образом, чрезвычайно важно определять рекурсивные функции с большой осторожностью. Если Вы подозреваете, что возможен случай бесконечной рекурсии, можно задать функцию, вызывающую себя

определенное количество раз, то есть если обращений к функции слишком много, автоматически происходит выход из нее.

Ниже приведена функция, написанная на JScript.

```
function factorial(aNumber) {  
    aNumber = Math.floor(aNumber); // Если число не цело, "откинем"  
    //дробную часть.  
    if (aNumber < 0) { // Если число меньше нуля, выход.  
        return "Значение не определено.";  
    }  
    if ((aNumber == 0) || (aNumber == 1)) { // Если число 0 или 1,  
    //факториал равен 1.  
        return 1;  
    }  
    else return (aNumber * factorial(aNumber - 1)); // Вычисляем до  
    //выполнения  
}
```

Копирование, прием и сравнение данных

В Microsoft JScript обработка данных зависит от типа данных.

Значение и ссылка

Числовые и логические значения (*true* и *false*) копируются, принимаются и сравниваются значениями. При копировании или приеме значения, определяется место в памяти компьютера и в него помещается значение оригинала. Если оригинал меняется, на значение копии это не влияет (и наоборот), так как копия и оригинал являются двумя различными объектами.

Объекты, массивы и функции копируются, принимаются и сравниваются в большинстве случаев по ссылке. При копировании или приеме по ссылке, вы по существу создаете указатель на оригинал, применяя указатель как копию. При изменении оригинала, вы изменяете и копию, и оригинал. Это только один объект; "копия" на самом деле не является копией, это всего лишь еще одна ссылка на данные.

И наконец, строки копируются и принимаются по ссылке, а сравниваются по значению.

Прием параметров к функциям

Когда прием параметров к функциям происходит по значению, создается отдельная копия параметра, которая существует внутри функции. С другой стороны, при приеме параметра по ссылке, если функция изменяет значение этого параметра, он изменяет значение во всем сценарии.

Проверка данных

При проверке элементов по значению, происходит сравнение на равенство друг другу. Обычно, сравнение происходит поразрядно, байт к байту. При проверке по ссылке, определяется, являются ли оба элемента указателями на один оригинальный элемент. Если являются, то они равны; если нет (даже при поразрядном равенстве), то они не являются равными.

Скопированные и принятые по ссылке строки сохраняются в памяти; но невозможно изменить строки, как только они созданы, это становится возможным при их сравнении по значению. Это позволяет проверять, имеют ли две строки то же самое содержание, даже если каждая из них была создана отдельно от другой.

Использование массивов

Массивы в JScript разряжены. Так, если в массиве три значения пронумерованы как 0, 1 и 2, вы можете создать элемент 50, не волнуясь об элементах от 3 до 49. Если массив имеет автоматическую переменную размера (смотрите встроенные объекты (Intrinsic Objects) для определения размера массива), переменная размера установлена как 51, несмотря на размер 4. Конечно, можно создавать массивы без разрывов в нумерации элементов, но это не обязательное условие. Фактически в JScript, массивы вообще могут не иметь нумерации.

В JScript объекты и массивы идентичны друг другу. Настоящая разница не в данных, а скорее в адресации элементов массива или свойств (properties) и методов объекта.

Существует два основных способа для адресации элементов массива. Обычно, для адресации используют индексы. Индексы содержат числовое значение или выражение (expression), которое оценивается как неотрицательное целое. В следующем примере предполагается, что переменная *entryNum* определена, и ей присвоено значение в другом месте в сценарии.

```
theListing = addressBook[entryNum];  
theFirstLine = theListing[1];
```

Метод адресации эквивалентен методу адресации объектов, хотя при адресации объекта, индекс должен быть именем существующего свойства. Если такого свойства нет, возникает ошибка при исполнении кода.

Второй способ адресации массива состоит в том, чтобы создать объект (массив), который содержит свойства, которые пронумерованы числами в цикле. В следующем примере создается два массива, один для имени, второй для адреса, внесенных в список *addressBook*. Каждый из

них содержит четыре свойства. Образец *theName*, например, формируется от [Name1] до [Name4] свойств *theListing*, может содержать "G." "Edward" "Heatherington" "IV" или "George" "" "Sand" "".

```
theListing = addressBook[entryNum];
for (i = 1; i < 4; i++) {
  theName[i] = theListing["Name" + i];
  theAddress[i] = theListing["Address" + i];
}
```

В то же время код мог бы легко быть написан в "dot"-стиле системы обозначений (то есть адресуя *theListing*, *theName* и *theAddress* скорее как объекты, чем матрицы, через точку), но подобное не всегда возможно. Иногда какое-либо свойство не может существовать до времени выполнения, или нельзя его узнать заранее. Например, если массив *addressBook* упорядочен по фамилии вместо нумерации, пользователь будет вероятно вводить названия "на лету", в то время как сценарий функционирует, просматривая людей. Следующий пример показывает применение определений функций в другом месте сценария.

```
theListing = addressBook[getName()];
theIndivListing = theListing[getFirstName()];
```

Это ассоциативная адресация массива, то есть адресация посредством полностью произвольных строк. Объекты в JScript являются ассоциативными массивами. Хотя чаще всего используется "dot"-стиль, это не всегда требуется.

Организация диалога с пользователем

Благодаря тому, что при написании скриптов WSH, используется либо VBScript, либо JScript (довольно мощные языки программирования), появляется возможность создавать сценарии, позволяющие получить от пользователя какую-либо информацию, влияющую на процесс работы сценария. Эта важная возможность помогает создавать более гибкие и функциональные сценарии, предусматривающие различные нужды пользователей.

Получить информацию от пользователя можно с помощью диалогового окна или строки ввода информации. Рассмотрим, для начала диалоговые окна. Ниже приведен простой скрипт, выводящий пример диалогового окна и выдающий сообщение о выборе пользователя:

```
// Диалоговое окно. JScript
// http://www.whatis.ru
var WSHShell = WScript.CreateObject("WScript.Shell");
// Подготовка переменных для диалогового окна
var vbOKCancel = 1;
var vbInformation = 64;
```

```

var vbCancel = 2;
var Message = "Пример создания диалогового окна";
var Title = "Нажмите ОК или Cancel";
// Вызов диалогового окна
var intDoIt;
intDoIt=WSHShell.Popup(Message,0,Title, vbOKCancel+vbInformation );
// Результат выбора пользователя
WScript.Echo(intDoIt);

```

Вызов диалогового окна осуществляется с помощью метода **Popup** объекта **WSHShell**. Первым параметром передается текст, выводимый в диалоговом окне, третьим - заголовок окна, четвертым - набор кнопок и иконка в диалоговом окне. Вот на последнем и остановимся подробнее.





Каждому набору кнопок соответствует цифровая переменная:

- 0 - ОК;
- 1 - ОК и Отмена;
- 2 - Прервать, Повтор, Пропустить;
- 3 - Да, Нет, Отмена;
- 4 - Да, Нет;
- 5 - Повтор, Отмена;
- 6 - Отмена, Повторить, Продолжить.

Для лучшей читаемости кода удобнее определить переменную с названием, отражающим набор кнопок, в начале сценария, как это сделано в примере, а непосредственно при вызове диалогового окна использовать не цифру, а эту переменную.

Аналогично наборам кнопок, иконки в диалоговом окне определяются с помощью цифровой переменной. Соответствие переменных иконкам (таблица 1.7).

Таблица 7.1. Коды иконок диалогового окна

Код	16	32	48	64
Иконка				

Четвертый параметр метода **Popup** представляет собой сумму переменных набора кнопок и иконки, выводимых в диалоговом окне. Так, если вам надо вывести иконку вопроса и кнопки Да, Нет, Отмена, нужно передать в параметр $32 + 3$, т.е. 35.

Создавать диалоговое окно мы научились, теперь неплохо бы узнать какой выбор сделал пользователь. В примере результат выбора в диалоговом окне сохраняется в переменной **intDoIt**, а потом выводится на экран. Всем кнопкам диалогового окна соответствует числовое значение,

которое и возвращается при выборе одной из них. Полный список приведен ниже:

- 1 - ОК;
- 2 - Отмена;
- 3 - Прервать;
- 4 - Повтор;
- 5 - Пропустить;
- 6 - Да;
- 7 - Нет;
- 10 - Повторить;
- 11 - Продолжить.

Таким образом, получив результат выбора пользователя, можно предусмотреть несколько вариантов работы скрипта.

Однако, только кнопками Да, Нет, Отмена и т.п. не всегда можно обойтись. Например, как узнать у пользователя каталог, куда он хочет сохранить какой-то файл? Или букву диска, куда подмапировать сетевой ресурс? В таких случаях поможет строка ввода информации (**InputBox**). Ниже приведен простой сценарий, демонстрирующий работу такого диалогового окна.

```
' Диалоговое окно. VBScript
' http://www.whatis.ru
Dim s,s1
s1="Введите ваше имя"
' Выводим диалоговое окно со строкой ввода на экран
s=InputBox(s1,"Пример получения данных от пользователя")
' Результат ввода
MsgBox "Вас зовут " & s
```

Дальше уже можно обрабатывать введенную информацию в своем сценарии. Вот только есть один небольшой подводный камень: **InputBox** присутствует только в **VBScript**. А как быть, если вам надо написать сценарий на языке **JScript**? В WSH есть возможность объединять несколько сценариев, написанных на одном или разных языках, в один файл. Для этого служат **wsf-файлы**. Тема эта заслуживает отдельной статьи, но если кратко, то это **XML** файл, имеющий определенную структуру, в которой, каждый сценарий помещается в отдельный элемент XML, и может обращаться к функциям и переменным из других сценариев. Подробно эта тема будет рассмотрена в одной из будущих статей.

1.2. Практический блок

Задание 1. Основы работы с JScript

Создайте сценарий JScript, выполняющий следующие действия:

- 1) Расчет значений функции $\text{tg}(x)$ в промежутке $[0,1]$, с шагом 0.0001;
- 2) Вывод значений в консоль.

Задание 2. Пользовательские функции и массивы

Создайте сценарий JScript, выполняющий следующие действия:

- 1) формирование матрицы (NxN) вида:

```
1 7 7 7... 7
4 1 7 7 ...7
4 4 1 7 ...7
...
4 4 4 4... 1
```

- 2) вычисление количества элементов кратных 7;
- 3) вычисление суммы элементов матрицы в каждой нечетной строке;
- 4) вычисление произведения четных элементов в каждом нечетном столбце;
- 5) вычисление суммы элементов выше побочной диагонали;
- 6) сортировка каждой четной строки матрицы по возрастанию, каждой нечетной – по убыванию;
- 7) вычисление обратной матрицы для исходной;
- 8) умножение исходной матрицы на сортированную;
- 9) вывод результата умножения в одну строку.

Каждый пункт реализуйте в виде отдельной пользовательской функции. Организуйте вывод результатов работы функций в консоль.

Задание 3. Рекурсия

Создайте сценарий JScript, рассчитывающий сумму n первых чисел ряда при помощи рекурсивной функции. Ряд задается рекуррентно:

$$F(1)=1,$$

$$F(i)=\ln(f(i-1)*i)+9$$

2. Основы работы с Windows Script Host

2.1. Теоретический блок

2.1.1. Стандартные объекты WSH 5.6

С помощью внутренних объектов версии WSH 5.6 из сценариев можно выполнять следующие основные задачи [1,2]:

- 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 именованным 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.

Перейдем теперь к рассмотрению свойств и методов внутренних объектов WSH.

Объект WScript

Свойства объекта *WScript* позволяют получить полный путь к используемому 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.

Таблица 2.1. Свойства объекта *WScript*

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

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

Свойство Arguments

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

```
/*Пример 2.1. 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 *****/
```

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. 2.1).



Рис. 2.1. 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. 2.2.

Таблица 2.2. Методы для работы с потоками

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

```
/*Пример 2.2. 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.

2.3.

Приведем дополнительные пояснения и примеры использования для методов, приведенных в табл. 2.3.

Метод CreateObject

Строковый параметр *strProgID*, указываемый в методе *CreateObject*, называется программным идентификатором объекта (Programmatic 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*.

Таблица 2.3. Методы объекта *WScript*

Свойство	Описание
<i>CreateObject(strProgID [,strPrefix])</i>	Создает объект, заданный параметром <i>strProgID</i>
<i>ConnectObject(strObject, strPrefix)</i>	Устанавливает соединение с объектом <i>strObject</i> , позволяющее писать функции-обработчики его событий (имена этих функций должны начинаться с префикса <i>strPrefix</i>)
<i>DisconnectObject(obj)</i>	Отсоединяет объект <i>obj</i> , связь с которым была предварительно установлена в сценарии
<i>Echo([Arg1] [, Arg2] [...])</i>	Выводит текстовую информацию на консоль или в диалоговое окно
<i>GetObject(strPathname [,strProgID], [strPrefix])</i>	Активизирует объект автоматизации, определяемый заданным файлом (параметр <i>strPathName</i>), или объект, заданный параметром <i>strProgID</i>
<i>Quit([intErrorCode])</i>	Прерывает выполнение сценария с заданным параметром <i>intErrorCode</i> кодом выхода. Если параметр <i>intErrorCode</i> не задан, то объект <i>WScript</i> установит код выхода равным нулю
<i>Sleep(intTime)</i>	Приостанавливает выполнения сценария (переводит его в неактивное состояние) на заданное параметром <i>intTime</i> число миллисекунд

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);
```

Method DisconnectObject

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

Method 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* (пример 2.3) с помощью *cscript.exe* на экран будут выведены пустая строка, три числа и строка текста (рис. 2.2).

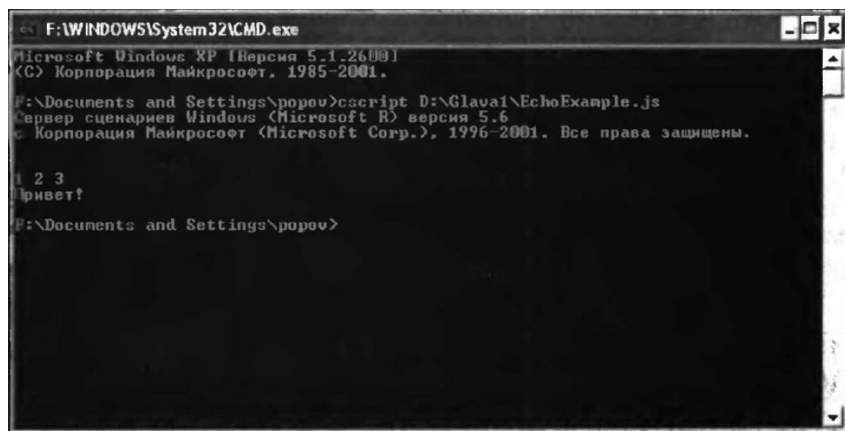


Рис. 2.2. Вывод информации с помощью метода *Echo*

```
/*Пример 2.3. 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 *****/
```

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*.

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 (пример 2.4).

```
/*Пример 2.4. Вывод всех параметров сценария */
/*****
/* Имя: Enum.Args.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.

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

Таким образом, предыдущий пример можно переписать более компактным образом (пример 2.5).

```
/*Пример 2.5. 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;
    WScript.Echo(objArgs(i)); //Выводим значение i-го параметра
/***** Конец *****/
```

С помощью объекта *WshArguments* можно также выделять и отдельно обрабатывать аргументы сценария, у которых имеются имена (например, /*NametAndrey*) 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 именованным 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.

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 примере 2.6 приведен сценарий, который удаляет две переменные (*example_1* и *example_2*) из окружения среды пользователя.

Если в окружении среды пользователя нет переменных с именами *example_1* и *example_2*, то при вызове метода *Remove* произойдет ошибка.

```
/*Пример 2.6. 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 9x поддерживаются следующие имена специальных папок:

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

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

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

Объект *WshSpecialFolders* создается с помощью свойства *SpeciaiFolders* объекта *WshShell*:

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

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 (пример 2.7).

```
/*Пример 2.7. 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 Favorites
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)); }
/***** Конеч *****/

```

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. 2.4.

Таблица 2.4. Свойства объекта *WshShell*

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

Опишем теперь некоторые методы, имеющиеся у объекта *WshShell* (табл. 2.5).

Таблица 2.5. Методы объекта *WshShell*

Свойство	Описание
<i>AppActivate(title)</i>	Активизирует заданное параметром <i>title</i> окно приложения. Строка <i>title</i> задает название окна (например, " <i>calc</i> " или " <i>notepad</i> ") или идентификатор процесса (Process ID, PID)
<i>CreateShortcut(strPathname)</i>	Создает объект <i>WshShortcut</i> для связи с ярлыком Windows (расширение .lnk) или объект <i>WshUrlShortcut</i> для связи с сетевым ярлыком (расширение .url). Параметр <i>strPathname</i> задает полный путь к создаваемому или изменяемому ярлыку
<i>Environment(strType)</i>	Возвращает объект <i>WshEnvironment</i> , содержащий переменные среды заданного вида
<i>Exec(strCommand)</i>	Создает новый дочерний процесс, который запускает консольное приложение, заданное параметром <i>strCommand</i> . В результате возвращается объект <i>WshScriptExec</i> , позволяющий контролировать ход выполнения запущенного приложения и обеспечивающий доступ к потокам <i>StdIn</i> , <i>StdOut</i> и <i>StdErr</i> этого приложения
<i>ExpandEnvironmentStrings(strString)</i>	Возвращает значение переменной среды текущего командного окна, заданной строкой <i>strString</i> (имя переменной должно быть окружено знаками "%")
<i>LogEvent (int Type, strMessage [,strTarget])</i>	Протоколирует события в журнале Windows NT/2000/XP или в файле <i>WSH.log</i> . Целочисленный параметр <i>intType</i> определяет тип сообщения, строка <i>strMessage</i> — текст сообщения. Параметр <i>strTarget</i> может задаваться только в Windows NT/2000/XP, он определяет название системы, в которой протоколируются события (по умолчанию это локальная система). Метод <i>LogEvent</i> возвращает <i>true</i> , если событие записано успешно и <i>false</i> в противном случае
<i>Popup(strText, [nSecToWait], [strTitle], [nType])</i>	Выводит на экран информационное окно с сообщением, заданным параметром <i>strText</i> . Параметр <i>nSecToWait</i> задает количество секунд, по истечении которых окно будет автоматически закрыто, параметр <i>strTitle</i> определяет заголовок окна, параметр <i>nType</i> указывает тип кнопок и значка для окна
<i>RegDelete(strName)</i>	Удаляет из системного реестра заданный параметр или раздел целиком
<i>RegRead(strName)</i>	Возвращает значение параметра реестра или значение по умолчанию для раздела реестра
<i>RegWrite(strName, anyValue [,strType])</i>	Записывает в реестр значение заданного параметра или значение по умолчанию для раздела
<i>Run[strCommand, [intWindowStyle], [bWaitOnReturn])</i>	Создает новый независимый процесс, который запускает приложение, заданное параметром <i>strCommand</i>

Таблица 2.5. Методы объекта *WshShell* (продолжение)

Свойство	Описание
<i>SendKeys(string)</i>	Посылает одно или несколько нажатий клавиш в активное окно (эффект тот же, как если бы вы нажимали эти клавиши на клавиатуре)
<i>SpecialFolders</i> (<i>strSpecFolder</i>)	Возвращает строку, содержащую путь к специальной папке Windows, заданной параметром <i>strSpecFolder</i>

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. 2.6 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 2.6. Типы кнопок и иконок для метода *PopUp*

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

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 vbOkCancel1;*). Естественно, в любых сценариях вместо имен констант можно использовать их числовые значения.

В методе *PopUp* можно комбинировать значения параметра, приведенные в табл. 2.6. 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. 2.3, которое автоматически закроется через 5 секунд.

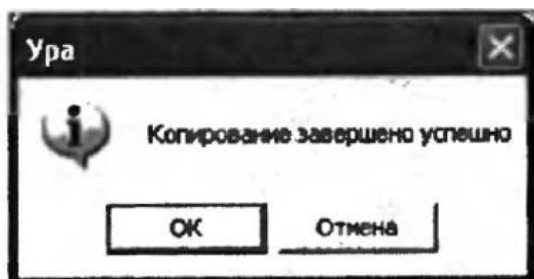


Рис. 2.3. Информационное окно, созданное методом *PopUp*

Метод *PopUp* возвращает целое значение, с помощью которого можно узнать, какая именно кнопка была нажата для выхода (табл. 2.7).

Таблица 2.7. Возвращаемые методом *PopUp* значения

Значение	Константа Visual Basic	Описание
-1		Пользователь не нажал ни на одну из кнопок в течение времени, заданного параметром <i>nSecToWait</i>
1	<i>vbOk</i>	Нажата кнопка ОК
2	<i>vbCancel</i>	Нажата кнопка Отмена (Cancel)
3	<i>vbAbort</i>	Нажата кнопка Стоп (Abort)
4	<i>vbRetry</i>	Нажата кнопка Повтор (Retry)
5	<i>vbIgnore</i>	Нажата кнопка Пропустить (Ignore)
6	<i>vbYes</i>	Нажата кнопка Да (Yes)
7	<i>vbNo</i>	Нажата кнопка Нет (No)

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*. Ниже представлен пример сценария, в котором создается ярлык на этот самый сценарий (ярлык будет находиться в текущем каталоге).

*/*Пример 2.8 .Создание ярлыка на выполняемый сценарий*/*

```

/* Имя: MakeShortcutl.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.Script.FullName;
//Сохраняем ярлык
oShellLink.Save();
*****/

```

Свойства объекта *WshShortcut* описаны в таблице 2.8.

Таблица 2.8. Свойства объекта *WshShortcut*

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

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

Свойство *Arguments*

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

```

/*Пример 2.8. Создание ярлык на сценарий с двумя параметрами
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 "+".

"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 на рабочем столе создается ярлык для Блокнота, которому назначается комбинация "горячих" клавиш <Ctrl>+<Alt>+<D>.

```

/*Пример 2.9. Назначение горячих клавиш на ярлык*/
/*****/
/* 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 создается ярлык на выполняющийся сценарий с первым значком (индекс 0) из файла *notepad.exe*.

Пример 2.10.

```
/* *****/
/* Имя: MakeShortcut4.js */
/* Язык: JScript */
/* Описание: Создание ярлыка на выполняемый сценарий со */
/* значком из notepad.exe */
/* *****/
var WshShell,oShellLink;
//Создаем объект WshShell
WshShell = WScript.CreateObject("WScript.Shell");
//Создаем ярлык в текущем каталоге
oShellLink = WshShell.CreateShortcut("Current Script.Ink");
//Устанавливаем путь к файлу
oShellLink.TargetPath = WScript.ScriptFullName;
//Выбираем значок 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. 2.9.

Таблица 2.9. Значения параметра *IntWindowStyle*

<i>IntWindowStyle</i>	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 создается ярлык для Блокнота, причем в качестве рабочего каталога указан корневой каталог диска C:.

Пример 2.11.

```

/*****/
/* Имя: MakeShortcutS.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.Ink");
//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* [1,2].

2.2. Практический блок

Задание 1. Работа с параметрами командной строки

Создайте сценарий JScript, который в зависимости от введенного пользователем параметра командной строки выполняет следующие действия:

а) Сравнение двух чисел, введенных с клавиатуры

б) Выводит на экран путь к исполняемому файлу сервера сценариев, имя запущенного сценария и версию WSH

Предусмотрите вывод справки по работе скрипта.

Задание 2. Работа с объектами коллекции

Создайте сценарий JScript, который в зависимости от введенного пользователем параметра командной строки выполняет следующие действия:

а) Выводит имена и содержимое всех системных переменных окружения

б) Выводит все специальные папки

Если введен неверный аргумент командной строки, необходимо предусмотреть сообщение об ошибке.

Задание 3. Работа с методом *Popup* и *WshShortcut*

Создайте сценарий JScript, который выполняет следующие действия:

а) При помощи всплывающего окна запрашивает у пользователя необходимость создания на рабочем столе ярлыков для запуска сценария из задания 2

б) При положительном ответе пользователя создает два ярлыка для выполнения задания 2.а и 2.б

Предусмотрите вывод всех возможных ошибок в журнал

3. Вызов приложений при помощи Windows Script Host. Работа с реестром Windows

3.1. Теоретический блок

3.1.1. Вызов приложений

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 "abb": `string="abb" [1-3]`.

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. 3.2.

Для задания комбинаций клавиш с <Shift>, <Ctrl> или <Alt>, перед соответствующей клавишей нужно поставить один или несколько кодов из табл. 3.1.

Таблица 3.1. 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 симитировать 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.

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

Таблица 3.2. 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}
 или <Delete>	{DELETE} или {DEL}	<F3>	{F3}
<End>	{END}	<F4>	{F4}
<Enter>	{ENTER} или ~	<F5>	{F5}
<Esc>	{ESC}	<F6>	{F6}
<Home>	{HOME}	<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}		

Запуск из сценариев внешних программ

Внешние программы и команды можно запускать из сценариев различными способами. 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 maximized (распахнутом на весь экран) окне (список всех возможных значений параметров метода *Run* приведен в табл. 3.3).

Таблица 3.3. Типы окна

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 <i>Run</i>)
10		Устанавливает режим отображения, опирающийся на режим программы, которая запускает приложение

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");
```

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*.

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 симитировать нажатия клавиш в этом окне. Для этого используется метод *SendKeys* объекта *WshShell*.

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

3.1.2. 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*.

С помощью методов объекта *WshShell* из сценариев WSH можно [1-3]:

- 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*).

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

В примере 3.1 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". После этого параметр *ExampleValue* и раздел *ExampleKey* последовательно удаляются из реестра.

'Пример 3.1. 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.Prompt("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.Prompt "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
'***** Конеч *****

```

3.2. Практический блок

Задание 1. Использование методов SendKeys, Run и Exec

Создайте JScript-сценарий выполняющий следующие действия:

- 1) запускает программу "Калькулятор";
- 2) выполняет с ее помощью некоторое арифметическое действие, например 3+2;
- 3) копирует в буфер результат вычислений;
- 4) закрывает программу "Калькулятор";
- 5) запускает программу "Блокнот" и вводит в нее следующие строки:
*«Результат вычислений
3+2=5
Конец»*
- 6) сохраняет текстовый файл.

При этом при запуске программ и сохранении результатов для пользователя выводится диалоговое окно, запрашивающее разрешение этих действий.

Результаты действий пользователя заносятся в журнал.

Задание 2. Работа с реестром. Метод Sleep

Создайте JScript-сценарий выполняющий следующие действия:

- 1) Сохраняет копию системного реестра;
- 2) Вносит в реестр следующие изменения:
 - а) изменяет цветовую схему для текущего пользователя
 - б) скрывает стрелки ярлыков
 - в) скрывает отображение диска А: в папке "Мой компьютер"
- 3) Перезагружает компьютер
- 4) После перезагрузки компьютера через 30 секунд восстанавливает исходные значения реестра

4. Сценарии WSH как приложения XML

4.1. Теоретический блок

4.1.1. Возможности WSH 2.0

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 [1]. 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.

4.1.2. 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 [1], 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 [1]:

```
<element [attributed="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 ниже 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.

```
<?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: GUTD" | progid=ProgID] />
    <reference
      [object="ProgID" | guid="typelibGUID"] [version="version"] />
    <script language="language" [src="strFileURL"]\>
    < script language="language" >
    <![CDATA[
      Код сценария
    ]]>
    </script>
  </job>
</package>
```

Таким образом, из листинга 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 "*]]>*".

В WSH 5.6 названия и значения атрибутов в элементе *<?XML?>* должны быть именно такими, как в листинге (*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. 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>*. 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. В примере 4.1, в сценарии *twojobs.wsf* определяются два задания с именами *"Task1"* и *"Task2"*.

```
<!-- Пример 4.1. Файл twojobs.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="Task2") -->
```

```
<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" twojobs.wsf
```

запускает с помощью `cscript.exe` задание с именем `"Task1"` из файла `twojobs.wsf`.

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. 4.1).

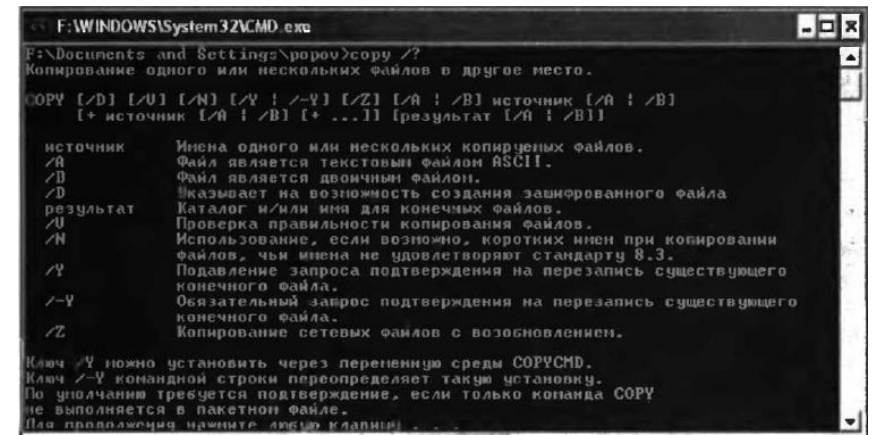


Рис.4.1. Встроенная справка для команды `Copy`

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>*.

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. 4.1 приведено описание аргументов элемента *<named>*.

Таблица 4.1. Аргументы элемента *<named>*

Аргумент	Описание
<i>name</i>	Задаёт имя параметра командной строки
<i>helpstring</i>	Строка, содержащая описание параметра командной строки
<i>type</i>	Определяет тип параметра командной строки. Может принимать значения "string" (символьный тип), "boolean" (логический тип), "simple" (в сценарий передаётся только имя параметра без дополнительного значения). По умолчанию используется тип "simple"
<i>required</i>	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 примере 4.2 приведен сценарий *named.wsf*, в котором в блоке *<runtime>* описываются три именованных 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. 4.2). 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 именованных аргументов командной строки, и используется метод *Exists* этого объекта:

```
//Создаем объект WshNamed — коллекция именованных 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";
```



Рис.4.2. Информационное окно с информацией о параметрах сценария *named.wsf*

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 *namesLwsf* следующим образом: *wscript.exe*

named.wsf /Имя:Роров /Компьютер:404_Роров /Новый+

то на экран будет выведено диалоговое окно, показанное на рис. 4.3.

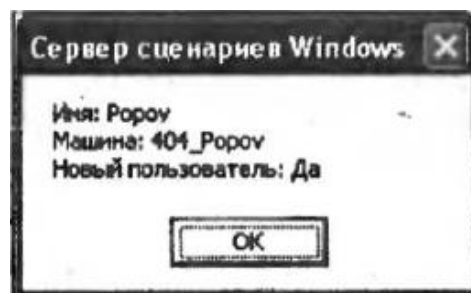


Рис.4.3. Значения именованных аргументов командной строки, переданных в *named.wsf*

```

<!--Пример 4.2. Файл named.wsf-->
<job id="Named">
  <runtime>
    <description>
      Имя: named.wsf
    </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;
    //Вызываем метод ShowUsage для вывода на экран описания сценария
    WScript.Arguments.ShowUsage();
    //Создаем объект WshNamed — коллекция именованных 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";
  </script>
</job>

```

```

//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. 4.2 приведено описание аргументов элемента `<unnamed>`.

Таблица 4.2. Аргументы элемента `<unnamed>`

Аргумент	Описание
<i>name</i>	Задаёт имя, которое будет указано для описываемого параметра командной строки при выводе информации о сценарии
<i>helpstring</i>	Строка, содержащая описание параметра командной строки
<i>many</i>	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
<i>required</i>	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` (пример 4.3), в который в качестве параметров командной строки должны передаваться

расширения файлов, причем обязательно должны быть указаны хотя бы два таких расширения.

Для создания информации об использовании этого сценария создается элемент *<unnamed>* следующего вида:

```
<unnamed  
  name="Расш"  
  helpstring="Расширение 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. 4.4). Затем в сценарии создается коллекция *objUnnamedArgs* (объект *WshUnnamed*), которая содержит все безымянные аргументы командной строки, реально переданные в сценарий:

```
objUnneittedArgs=WScript.Arguments.Unnamed; //Создаем объект  
                                              //WshUnnamed
```

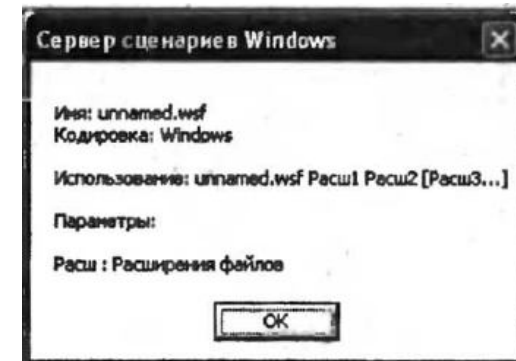


Рис.4.4. Диалоговое окно с информацией о параметрах сценария *unnamed.wsf*

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. 4.5.

```
<!--Пример 4.3. Файл unnamed.wsf-->
<job id="Unnamed">
  <runtime>
    <description>
      Имя: unnamed.wsf
    </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>
```




Рис 4.5. Значения безымянных аргументов командной строки, переданных в *unnamed.wsf*

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* (пример 4.4). Здесь сразу вызывается метод *Wscript.Arguments.ShowUsage*, в результате чего на экран выводится диалоговое окно (в случае запуска сценария с помощью *wscript.exe*) (рис. 4.6, а) или просто строки текста (в случае запуска сценария с помощью *cscript.exe*) с описанием запущенного сценария (рис. 4.6, б).

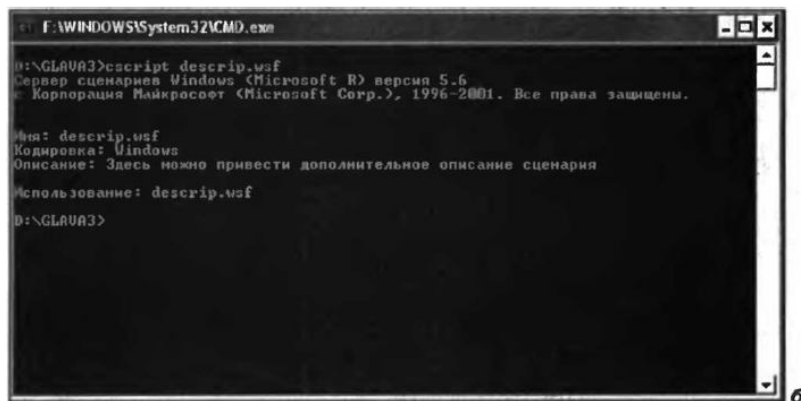
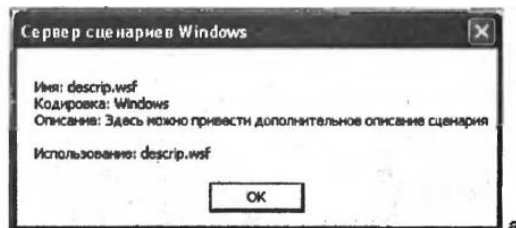


Рис. 4.6. Вывод текста, описывающего сценарий: а – в графическом режиме; б – в консольном режиме

<!--Пример 4.4.Файл descrip.wsf-->

<job id="Descrip">

<runtime>

<description>

Имя: descrip.wsf

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. 4.7).



Рис. 4.7. Диалоговое окно, формируемое элементами `<description>` и `<example>`

Сценарий `example.wsf`, диалоговое окно, с описанием которого показано на рис. 4.7, приведен в примере 4.5..

```
<!--Пример 4.5. Файл example.wsf-->
<job id="Example">
<runtime>
<description>
Имя: example.wsf
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 приере 4.6. 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* объекта *WScript* и метода *getResource*:

```
WScript.Echo(getResource("MyName"));
```

```
<!--Пример 4.6. 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 СОМ-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 (Programmatic Identifier). Из этих двух необязательных атрибутов может быть указан только один. Например, создать объект *FileSystemObject* (GUID="OD43FE01-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. 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.

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!

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.

Элемент `<script>` является вторым обязательным элементом в сценариях WSH с разметкой XML.

4.1.3. Примеры сценариев с разметкой XML

Приведем примеры сценариев, иллюстрирующие основные свойства WS-файлов [1,2].

Строгий режим обработки ws -файла

Напомним, что здесь обязательными являются элементы `<?XML?>` и `<![CDATA[]]>`. Соответствующий пример сценария *strict.wsf* приведен в примере 4.7.

```
<!--Пример 4.7. Файл strict.wsf-->
<?XML version=" 1.0" standalone="yes" ?>
<job id="JS">
  <runtime>
    <description>
      Имя: strict.wsf
      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 примере 4.8. Здесь описываются два задания с идентификаторами "VBS" (сценарий на языке VBScript) и "JS" (сценарий на языке JScript).

```
<!--Пример 4.8. Файл multijob.wsf-->
<package>
  <job id="VBS">
    <!-- Описываем первое задание (id="VBS") -->
    <runtime>
      <description>
        Имя: multijob.wsf
        Описание: Первое задание из multijob.wsf
      </description>
    </runtime>
  </job>
  <job id="JS">
    <!-- Описываем второе задание (id="JS") -->
    <runtime>
      <description>
        Имя: multijob.wsf
        Описание: Второе задание из multijob.wsf
      </description>
    </runtime>
  </job>
</package>
```

```

</description>
</runtime>
<script language="VBScript">
WScript.Echo "Первое задание (VBScript)"
</script>
</job>
<job id="JS">
<!-- Описываем второе задание (id="JS") -->
<description>
Имя: multijob.wsf
Описание: Второе задание из multijob.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 примере 4.9 приведен сценарий *refer.wsf*, в котором с помощью элемента *<reference>* производится доступ к трем константам объекта *FileSystemObject* (*ForReading*, *ForWriting* и *For Appending*), которые определяют режим работы из сценария с внешним текстовым файлом.

```

<!--Пример 4.9. Файл refer.wsf -->

```



```

<job id="Example">
<runtime>
<description>
Имя: refer.wsf
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 For Appending
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. 4.8).



Рис.4.8. 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 примерер 4.10.

```
<!--Пример 4.10. Подключение внешнего сценария (файл main.wsf) -->
<job id="Example">
  <runtime>
    <description>
      Имя: main.wsf
      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)

Ни в WSH, ни в JScript нет метода или функции, которые позволяли бы в графическом режиме создать диалоговое окно для ввода текста [1,2]. 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* (пример 4.11).

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 `a`:

```

var a;
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.

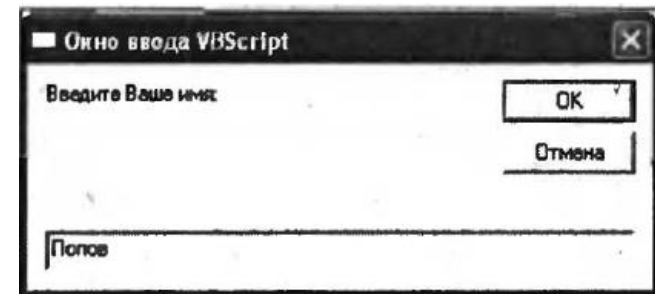


Рис. 3.9. Окно ввода функции `InputBox`

После ввода информации на экран выводится окно, содержащее введенное пользователем значение.

```

<!--Пример 4.11. Использование различных языков внутри-->
<!--одного задания (файл multilang.wsf) -->
<job id="Example">
<runtime>
<description>
Имя: multilang.wsf
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>
```

4.2. Практический блок

Задание 1. Атрибуты файла

Создайте самодокументируемый JScript-сценарий с использованием технологии XML, который выполняет следующие действия:

- 1) запрашивает в оконном режиме имя файла
- 2) проверяет, установлен ли для данного файла атрибут "Архивный"
- 3) если атрибут установлен, то он сбрасывается, если не установлен - устанавливается.

Задание 2. Информация о пользователе

Создайте самодокументируемый JScript-сценарий с использованием технологии XML, который для текущего пользователя выводит следующую информацию:

- 1) Диск, на котором находится профиль пользователя
- 2) Домашний каталог
- 3) Идентификатор пользователя
- 4) Раскладку клавиатуры по умолчанию

Для выполнения задания воспользуйтесь разделом реестра HKEY_CURRENT_USER.

5. Доступ из сценариев к файловой системе

5.1. Теоретический блок

Сценарии WSH позволяют получить полный доступ к файловой системе компьютера [1-3], 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* [1,2]. 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 списка всех файлов заданного каталога	Коллекция <i>Files</i> , содержащаяся в свойстве <i>Folder.Files</i>
Открытие текстового файла для чтения, записи или добавления	Методы <i>FileSystemObject.CreateTextFile</i> или <i>File.OpenAsTextStream</i>
Чтение информации из заданного текстового файла или запись ее в него	Методы объекта <i>TextStream</i>
Получение сведений об определенном диске (тип файловой системы, метка тома, общий объем и количество свободного места и т. д.)	Свойства объекта <i>Drive</i> . Сам объект <i>Drive</i> создается с помощью метода <i>GetDrive</i> объекта <i>FileSystemObject</i>
Получение сведений о заданном каталоге или файле (дата создания или последнего доступа, размер, атрибуты и т. д.)	Свойства объектов <i>Folder</i> и <i>File</i> . Сами эти объекты создаются с помощью методов <i>GetFolder</i> и <i>GetFile</i> объекта <i>FileSystemObject</i>
Проверка существования определенного диска, каталога или файла	Методы <i>DriveExists</i> , <i>FolderExists</i> и <i>FileExists</i> объекта <i>FileSystemObject</i>
Копирование файлов и каталогов	Методы <i>CopyFile</i> и <i>CopyFolder</i> объекта <i>FileSystemObject</i> , а также методы <i>File.Copy</i> и <i>Folder.Copy</i>
Перемещение файлов и каталогов	Методы <i>MoveFile</i> и <i>MoveFolder</i> объекта <i>FileSystemObject</i> или методы <i>File.Move</i> и <i>Folder.Move</i>
Удаление файлов и каталогов	Методы <i>DeleteFile</i> и <i>DeleteFolder</i> объекта <i>FileSystemObject</i> или методы <i>File.Delete</i> и <i>Folder.Delete</i>
Создание каталога	Методы <i>FileSystemObject.CreateFolder</i> или <i>Folders.Add</i>
Создание текстового файла	Методы <i>FileSystemObject.CreateTextFile</i> или <i>Folder.CreateTextFile</i>
Получение списка всех доступных дисков	Коллекция <i>Drives</i> , содержащаяся в свойстве <i>FileSystemObject.Drives</i>
Получение списка всех подкаталогов заданного каталога	Коллекция <i>Folders</i> , содержащаяся в свойстве <i>Folder.SubFolders</i>

Перейдем теперь к подробному рассмотрению некоторых объектов, используемых при работе с файловой системой.

Объект *FileSystemObject*

Объект *FileSystemObject* является основным объектом, обеспечивающим доступ к файловой системе компьютера [1,2]; 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. Методы объекта *FileSystemObject* представлены в табл. 5.2.

Таблица 5.2. Методы объекта *FileSystemObject*

Метод	Описание
<i>BuildPath(path, name)</i>	Добавляет к заданному пути (параметр <i>path</i>) новое имя (параметр <i>name</i>)
<i>CopyFile(source, destination [,overwrite])</i>	Копирует один или несколько файлов из одного места (параметр <i>source</i>) в другое (параметр <i>destination</i>)
<i>CopyFolder(source, destination [, overwrite])</i>	Копирует каталог из одного места (параметр <i>source</i>) в другое (параметр <i>destination</i>)
<i>CreateFolder(foldername)</i>	Создает новый каталог с именем <i>foldername</i> . Если каталог <i>foldername</i> уже существует, то произойдет ошибка
<i>CreateTextFile(filename [,overwrite[, Unicode]])</i>	Создает новый текстовый файл с именем <i>filename</i> и возвращает указывающий на этот файл объект <i>TextStream</i>
<i>DeleteFile(filespec [,force])</i>	Удаляет файл, путь к которому задан параметром <i>filespec</i>
<i>DeleteFolder (folderspec [,force])</i>	Удаляет каталог, путь к которому задан параметром <i>folderspec</i> , вместе со всем его содержимым
<i>DriveExists(drivespec)</i>	Возвращает <i>True</i> , если заданное параметром <i>drivespec</i> устройство существует и <i>False</i> в противном случае
<i>FileExists(filespec)</i>	Возвращает <i>True</i> , если заданный параметром <i>filespec</i> файл существует и <i>False</i> в противном случае
<i>FolderExists(folderspec)</i>	Возвращает <i>True</i> , если заданный параметром <i>folderspec</i> каталог существует и <i>False</i> в противном случае

Таблица 5.2. Методы объекта *FileSystemObject* (продолжение)

Метод	Описание
<i>GetAbsolutePathName (pathspec)</i>	Возвращает полный путь для заданного относительного пути <i>pathspec</i> (из текущего каталога)
<i>GetBaseName(path)</i>	Возвращает базовое имя (без расширения) для последнего компонента в пути <i>path</i>
<i>GetDrive(drivespec)</i>	Возвращает объект <i>Drive</i> , соответствующий диску, заданному параметром <i>drivespec</i>
<i>GetDriveName(path)</i>	Возвращает строку, содержащую имя диска в заданном пути. Если из параметра <i>path</i> нельзя выделить имя диска, то метод возвращает пустую строку ("")
<i>GetExtensionName(path)</i>	Возвращает строку, содержащую расширение для последнего компонента в пути <i>path</i> . Если из параметра <i>path</i> нельзя выделить компоненты пути, то <i>GetExtensionName</i> возвращает пустую строку (""). Для сетевых дисков корневой каталог (\\) рассматривается как компонент пути
<i>GetFile(filespec)</i>	Возвращает объект <i>File</i> , соответствующий файлу, заданному параметром <i>filespec</i> . Если файл, путь к которому задан параметром <i>filespec</i> , не существует, то при выполнении метода <i>GetFile</i> возникнет ошибка
<i>GetFileName (pathspec)</i>	Возвращает имя файла, заданного полным путем к нему. Если из параметра <i>pathspec</i> нельзя выделить имя файла, метод <i>GetFileName</i> возвращает пустую строку ("")
<i>GetFolder(folderpec)</i>	Возвращает объект <i>Folder</i> , соответствующий каталогу, заданному параметром <i>folderspec</i> . Если каталог, путь к которому задан параметром <i>folderspec</i> , не существует, при выполнении метода <i>GetFolder</i> возникнет ошибка
<i>GetParentFolderName (path)</i>	Возвращает строку, содержащую имя родительского каталога для последнего компонента в за данным пути. Если для последнего компонента в пути, заданном параметром <i>path</i> , нельзя определить родительский каталог, то метод возвращает пустую строку (" ")
<i>GetSpecialFolder (folderpec)</i>	Возвращает объект <i>Folder</i> для некоторых специальных папок Windows, заданных числовым параметром <i>folderspec</i>
<i>GetTempName()</i>	Возвращает случайным образом сгенерированное имя файла или каталога, которое может быть использовано для операций, требующих наличия временного файла или каталога
<i>MoveFile(source, destination)</i>	Перемещает один или несколько файлов из одного места (параметр <i>source</i>) в другое (параметр <i>destination</i>)

Таблица 5.2. Методы объекта *FileSystemObject* (продолжение)

Метод	Описание
<i>MoveFolder(source, destination)</i>	Перемещает один или несколько каталогов из одного места (параметр <i>source</i>) в другое (параметр <i>destination</i>)
<i>OpenTextFile(filename [,iomode [,create [,format]]])</i>	Открывает заданный текстовый файл и возвращает объект <i>TextStream</i> 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.

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 сору операционной системы).

Метод *CreateTextFile*

Параметр *Unicode* является логическим значением, указывающим, в каком формате (ASCII или Unicode) следует создавать файл.

Для дальнейшей работы с созданным файлом, т. е. для записи или чтения информации, нужно использовать методы объекта *TextStream*. Соответствующий сценарий приведен в примере 5.1.

```
/*Пример 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();
```

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.

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*.

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*

"C:."	"C:\MyDocuments\Reports"
"C:..."	"C:\MyDocuments"
"C:\\"	"C:\\"
"Region1"	"C:\MyDocuments\Reports\Region1"
"C: \. . \. \MyDocuments"	"C:\MyDocuments"

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 *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
<i>WindowsFolder</i>	0	Каталог Windows (например, "C:\Windows")
<i>SystemFolder</i>	1	Системный каталог, содержащий файлы библиотек, шрифтов и драйверы устройств
<i>TemporaryFolder</i>	2	Каталог для временных файлов, путь к которому хранится в переменной среды TMP

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
<i>ForReading</i>	0	Файл открывается только для чтения, записывать информацию в него нельзя
<i>ForWriting</i>	1	Файл открывается для записи. Если файл с таким именем уже существовал, то при новой записи его содержимое теряется
<i>ForAppending</i>	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
<i>TristateUseDefault</i>	-2	Файл открывается в формате, используемом системой по умолчанию
<i>TristateTrue</i>	-1	Файл открывается в формате Unicode
<i>TristateFalse</i>	0	Файл открывается в формате ASCII

Для дальнейшей работы с открытым файлом, т. е. для записи или чтения информации, нужно использовать методы объекта *TextStream*.

Объект *Drive*

С помощью объекта *Drive* можно получить доступ к свойствам заданного локального или сетевого диска. Создаётся 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.

В примере 5.2 приведен сценарий *Driveinfo.js*, в котором объект *Drive* используется для доступа к некоторым свойствам диска C: (рис. 5.1).

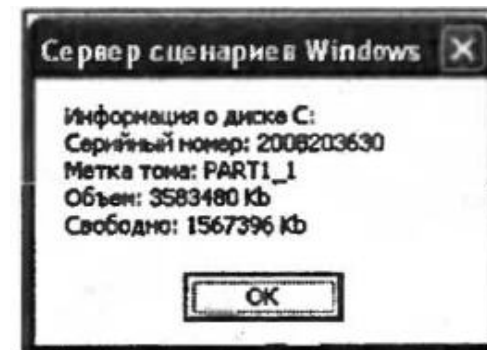


Рис. 5.1. Свойства диска C:

Таблица 5.7. Свойства объекта *Drive*

Свойство	Описание
<i>AvailableSpace</i>	Содержит количество доступного для пользователя места (в байтах) на диске
<i>DriveLetter</i>	Содержит букву, ассоциированную с локальным устройством или сетевым ресурсом. Это свойство доступно только для чтения
<i>DriveType</i>	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
<i>FileSystem</i>	Содержит тип файловой системы, использующейся на диске (FAT, NTFS или CDFS)
<i>FreeSpace</i>	Содержит количество свободного места (в байтах) на локальном диске или сетевом ресурсе. Доступно только для чтения
<i>IsReady</i>	Содержит <i>true</i> , если устройство готово, и <i>false</i> в противном случае. Для устройств со сменными носителями и приводах CDROM. <i>IsReady</i> возвращает <i>true</i> только в том случае, когда в дисковод вставлен соответствующий носитель и устройство готово предоставить доступ к этому носителю
<i>Path</i>	Содержит путь к диску (например, "C:", но не "C:\ ")
<i>RootFolder</i>	Содержит объект Folder, соответствующий корневому каталогу на диске. Доступно только для чтения
<i>SerialNumber</i>	Содержит десятичный серийный номер тома заданного диска
<i>ShareName</i>	Содержит сетевое имя для диска. Если объект не является сетевым диском, то в свойстве <i>ShareName</i> содержится пустая строка ("")
<i>TotalSize</i>	Содержит общий объем в байтах локального диска или сетевого ресурса
<i>VolumeName</i>	Содержит метку тома для диска. Доступно для чтения и записи

```

/*Пример 5.2. Получение свойств диска C: */
/*****/

/* Имя: DrivelInfo.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 примере 5.3 приведен файл *ListDrives.js*, в котором с помощью объекта

Enumerator на экран выводятся сведения обо всех доступных дисках (рис. 5.2).



Рис.5.2. Список всех дисков, имеющихся в системе

```
/* Пример 5.3. Построение списка всех имеющихся */  
/* 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 элемент 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
<i>Attributes</i>	Позволяет просмотреть или установить атрибуты каталога
<i>DateCreated</i>	Содержит дату и время создания каталога. Доступно только для чтения
<i>DateLastAccessed</i>	Содержит дату и время последнего доступа к каталогу. Доступно только для чтения
<i>DateLastModified</i>	Содержит дату и время последней модификации каталога. Доступно только для чтения
<i>Drive</i>	Содержит букву диска для устройства, на котором находится каталог. Доступно только для чтения
<i>Files</i>	<i>Files</i> Содержит коллекцию <i>Files</i> , состоящую из объектов <i>File</i> для всех файлов в каталоге (включая скрытые и системные)
<i>IsRootFolder</i>	Содержит <i>true</i> , если каталог является корневым, и <i>false</i> в противном случае
<i>Name</i>	Позволяет просмотреть и изменить имя каталога. Доступно для чтения и записи
<i>ParentFolder</i>	Содержит объект <i>Folder</i> для родительского каталога. Доступно только для чтения

Следующий пример показывает, как объект *Folder* используется для получения даты создания каталога (пример 5.4).

```

/*Пример 5.4. 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*

Свойство	Описание
<i>Copy(destination [, overwrite])</i>	Копирует каталог в другое место
<i>CreateTextFile [filename[, overwrite [, Unicode]])</i>	Создает новый текстовый файл с именем <i>filename</i> и возвращает указывающий на этот файл объект <i>TextStream</i> (этот метод аналогичен рассмотренному выше методу <i>CreateTextFile</i> объекта <i>FileSystemObject</i>)
<i>Delete ([force])</i>	Удаляет каталог
<i>Move(destination)</i>	Перемещает каталог в другое место

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 примере 5.5 приведен сценарий *MakeSubFold.js*, который создает в каталоге "C:\Мои документы" подкаталог "Новая папка".

```

/*Пример 5.5. 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*****/

```

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*.

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 документы\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
<i>Attributes</i>	Позволяет просмотреть или установить атрибуты файлов
<i>DateCreated</i>	Содержит дату и время создания файла. Доступно только для чтения
<i>DateLastAccessed</i>	Содержит дату и время последнего доступа к файлу. Доступно только для чтения
<i>DateLastModified</i>	Содержит дату и время последней модификации файла. Доступно только для чтения
<i>Drive</i>	Содержит букву диска для устройства, на котором находится файл. Доступно только для чтения
<i>Name</i>	Позволяет просмотреть и изменить имя файла. Доступно для чтения и записи
<i>ParentFolder</i>	Содержит объект <i>Folder</i> для родительского каталога файла. Доступно только для чтения
<i>Path</i>	Содержит путь к файлу
<i>ShortName</i>	Содержит короткое имя файла (в формате 8.3)
<i>ShortPath</i>	Содержит путь к файлу, состоящий из коротких имен каталогов (в формате 8.3)
<i>Size</i>	Содержит размер заданного файла в байтах
<i>Type</i>	Возвращает информацию о типе файла. Например, для файла с расширением <i>txt</i> возвратится строка "Text Document"

Методы объекта *File* представлены в табл. 5.11.

Таблица 5.11. Методы объекта *File*

Свойство	Описание
<i>Copy (destination [, overwrite])</i>	Копирует файл в другое место
<i>Delete ([force])</i>	Удаляет файл
<i>Move(destination)</i>	Перемещает файл в другое место
<i>OpenAsTextStream([iomode],[format])</i>	Открывает заданный файл и возвращает объект <i>TextStream</i> , который может быть использован для чтения, записи или добавления данных в текстовый файл

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 *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*.

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*.

В примере 5.6 приведен сценарий *WriteTextFile.js*, иллюстрирующий использование метода *OpenAsTextStream* для записи строки в файл и чтения из него.

*/*Пример 5.6. 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);
```

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*.

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 FSO=WScript.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();
```

Свойства объекта *Textstream* описаны в табл. 5.12.

Таблица 5.12. Свойства объекта *Textstream*

Свойство	Описание
<i>AtEndOfLine</i>	Содержит <i>true</i> , если указатель достиг конца строки в файле, и <i>false</i> в противном случае. Доступно только для чтения
<i>AtEndOfStream</i>	Содержит <i>true</i> , если указатель достиг конца файла, и <i>false</i> в - противном случае. Доступно только для чтения
<i>Column</i>	Содержит номер колонки текущего символа в текстовом файле. Доступно только для чтения
<i>Line</i>	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*

Свойство	Описание
<i>Close()</i>	Закрывает открытый файл
<i>Read(n)</i>	Считывает из файла <i>n</i> символов и возвращает полученную строку
<i>ReadAll()</i>	Считывает полностью весь файл и возвращает полученную строку
<i>ReadLine()</i>	Возвращает полностью считанную из файла строку
<i>SkipLine()</i>	Пропускает целую строку при чтении
<i>Write(string)</i>	Записывает в файл строку <i>string</i> (без символа конца строки)
<i>WriteBlankLines(n)</i>	Записывает в файл <i>n</i> пустых строк (символы перевода строки и возврата каретки)
<i>WriteLine([string])</i>	Записывает в файл строку <i>string</i> (вместе с символом конца строки). Если параметр <i>string</i> опущен, то в файл записывается пустая строка

В примере 5.7 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. 5.3).

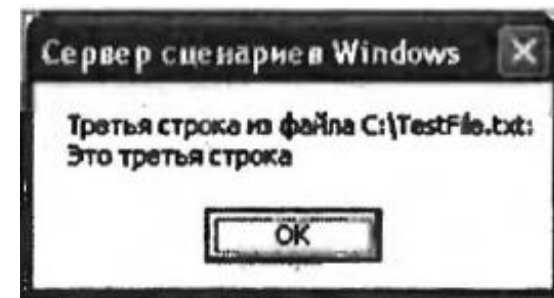


Рис.5.3. Результат работы сценария *TextFile.js*

```

/*Пример 5.7. 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:WTestFile.txt", true);
//Записываем в файл первую строку
F.Write("Это ");
F.WriteLine("первая строка");
//Записываем в файл пустую строку
F.WriteLine(1);
//Записываем в файл третью строку
F.WriteLine("Это третья строка");
//Закрываем файл
F.Close();
//Открываем файл для чтения
F=FSO.OpenTextFile("C:WTestFile.txt", ForReading);
//Пропускаем в файле две первые строки
F.SkipLine();
F.SkipLine();
s="Третья строка из файла C:WTestFile.txt:\n";
//Считываем из файла третью, строку
s+=F.ReadLine();
//Выводим информацию на экран
WScript.Echo(s);
/***** Конеч *****/

```


5.2. Практический блок

Задание 1. Использование дискового пространства

Создайте сценарий, выводящий в файл отчет об использовании дискового пространства. Отчет должен выводить информацию в текстовый файл и иметь вид, представленный на рисунке.

Задание 2. Удаление временных файлов

Создайте сценарий, который удаляет ненужные временные файлы с указанного пользователем диска. 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. 5.4).

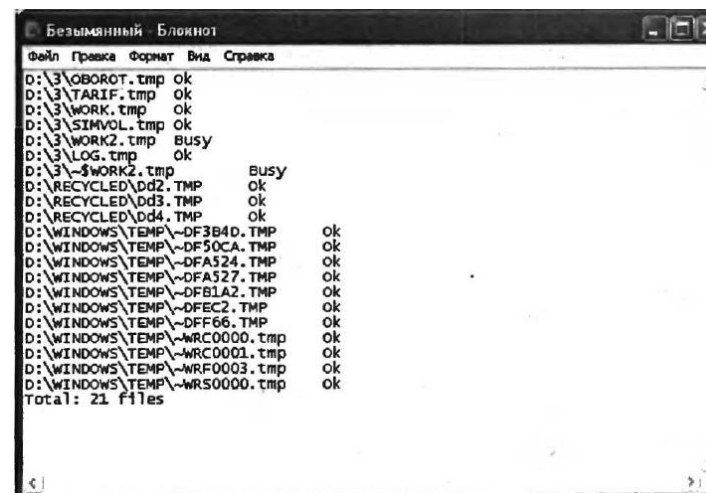


Рис. 5.4. Возможный вариант выходной информации для сценария из задания 2

Список использованной литературы

1. Попов А.В. Windows Script Host для Windows 2000/XP. СПб.: БВХ-Петербург. 2004
2. Попов А.В. Командные файлы и сценарии Windows Script Host. СПб.: БВХ-Петербург. 2002
3. Торрес Дж. Скрипты для администратора Windows. Специальный справочник. СПб.: Питер. 2002

Содержание

1. Введение в Windows Script Host. Основы JScript.....	3
1.1. Теоретический блок	3
1.1.1. Windows Script Host	3
1.1.2. Создание и запуск простейших сценариев WSH	4
1.1.3. Язык JScript.....	9
1.2. Практический блок.....	30
2. Основы работы с Windows Script Host	31
2.1. Теоретический блок	31
2.1.1. Стандартные объекты WSH 5.6.....	31
2.2. Практический блок.....	51
3. Вызов приложений при помощи Windows Script Host. Работа с реестром Windows	52
3.1. Теоретический блок	52
3.1.1. Вызов приложений	52
3.1.2. Работа с системным реестром Windows	55
3.2. Практический блок.....	58
4. Сценарии WSH как приложения XML.....	59
4.1. Теоретический блок	59
4.1.1. Возможности WSH 2.0.....	59
4.1.2. Основные принципы XML	60
4.1.3. Примеры сценариев с разметкой XML.....	79
4.2. Практический блок.....	85
5. Доступ из сценариев к файловой системе.....	86
5.1. Теоретический блок	86
5.2. Практический блок.....	105
Список использованной литературы	106

Учебное издание

Бречка Денис Михайлович

ОПЕРАЦИОННЫЕ СИСТЕМЫ.
ЧАСТЬ 2.
WINDOWS SCRIPT HOST

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ