PART I

Introducing the WSH and VBScript

Chapter 1: Getting Started with the WSH and VBScript

Chapter 2: An Introduction to the Windows Script Host
Getting Started with the WSH and VBScript

In this chapter, you’ll be introduced to a number of topics. These include a high-level overview of the Windows Script Host (WSH) and VBScript. You will learn how the WSH and VBScript work together to provide a comprehensive scripting environment. You will also be introduced to HTML Applications (HTAs) and learn how an HTA can be used to provide your scripts with a graphical user interface (GUI). In addition, you’ll learn a little bit about VBScript’s history and its relationship to other languages in the Visual Basic programming family. As a wrap-up, you’ll learn how to create and execute your very first VBScript.

Specifically, you will learn the following:

- The basic mechanics of the WSH
- How to write and execute VBScripts using the WSH
- Background information about VBScript and its capabilities
- How you can use HTAs to add GUIs to your scripts
- How to create your first VBScript game

Project Preview: The Knock Knock Game

In this chapter, as in all the chapters to follow, you will learn how to create a computer game using VBScript. This chapter’s game is called the Knock Knock game. Actually, it’s more of a riddle than a game, but it provides a great starting point for demonstrating how VBScript works and how it can be used to develop games and other useful scripts.
The Knock Knock game begins by displaying a pop-up dialog box that reads “Knock Knock.” It then waits for the user to respond with “Who’s there?” The dialog between the game and the player continues until the computer finally displays the game’s punch line. Figures 1.1 through 1.3 demonstrate operations of the script on Windows 7 and show the flow of the conversation between the game and the player. Figure 1.4 shows the message that appears if the player does not play the game correctly.

By the time you have created and run this game, you’ll have learned the fundamental steps involved in writing and executing VBScripts. At the same time, you will have prepared yourself for the more advanced programming concepts developed in later chapters, including how to use the WSH and VBScript to develop some really cool games.

What Is the WSH?

The Windows Script Host (WSH) is a programming environment that allows you to write and execute scripts that run on Windows operating systems. You can use the WSH to create and execute scripts—small text-based files written in an English-like programming language—from the Windows command prompt or directly from the Windows desktop. Scripts provide quick and easy ways to automate lengthy or mundane tasks that take too much time or effort using the Windows graphical user interface (GUI). Scripts are also better suited for automating tasks that are not complex enough to justify the development of an entire application using a language such as C++ or Visual Basic.
The WSH is made up of a number of different components. These components include the following:

- Script engines
- Script execution hosts
- The WSH core object model

The relationship of each of the components to one another is shown in Figure 1.5.

**Figure 1.5** The components that comprise the WSH.

**WSH Scripting Engines**

A *script execution engine* is a program that processes (interprets) the statements that make up scripts and translates them into machine-readable code that the computer can understand and execute. By creating an environment in which scripts can execute, the WSH makes script development a straightforward task.

The WSH provides each script with a number of resources. The WSH provides script engines for processing scripts. By default, Microsoft provides two script engines for the WSH:

- **VBScript.** A scripting language based on Microsoft’s Visual Basic programming language.
- **JScript.** A scripting language based on Netscape’s JavaScript Web-scripting language.

Therefore, by default, the WSH can process scripts written in either VBScript or JScript. The WSH is designed in a modular fashion, allowing Microsoft and third-party software developers to add support for additional scripting engines. For example, script execution engines have been developed for Perl, Python, and Rexx.
Selecting a WSH Script Execution Host

To actually run a script, the WSH uses a script execution host to process a script after a script engine has interpreted that script. The WSH supplies two different script execution hosts:

- **CScript.exe.** An execution host that enables scripts to execute from the Windows command prompt and display text-based messages.
- **WScript.exe.** An execution host that enables scripts to execute from the Windows desktop, display messages, and collect user input using graphical pop-up dialog boxes.

With the exception of the WScript.exe execution host’s capability to display graphical pop-up dialog boxes, the functionality provided by the WSH’s two execution hosts is identical. In fact, if you run a script using the CScript.exe execution host, the script can, depending on how it is written, still display messages using pop-up dialog boxes.

As both execution hosts provide the same basic functionality, you’re probably wondering which one you should use. There’s no right or wrong answer here. Often, the selection of an execution host is simply a matter of personal preference. However, there are some circumstances in which you may want to choose one over the other. For example, if you plan to run your scripts in the background, or if you want to schedule the execution of your scripts using the Windows Task Scheduler service and have no requirement for interacting with the user, you might want to use CScript.exe. However, if your scripts need to interact with the user—which will be the case with the games you’ll create with this book—you’ll want to use the WScript.exe execution host. Another factor that may affect your selection of a script execution host is your personal comfort level in working with the Windows command prompt.

Introducing the WSH Core Object Model

The WSH provides one final component, called the *core object model*, which is critically important to the development and execution of scripts. The WSH core object model provides VBScript with direct access to Windows resources.

Examples of the types of Windows resources to which the WSH core object model provides access include the following:

- Windows desktop
- Windows Start menu
- Windows applications
- Windows file system
- Network printers
- Network drives
- Windows Registry
The Windows operating system can be viewed as a collection of objects. For example, a file is an object. So is a folder, disk drive, printer, or any other resource that is part of the computer. What the core object model does is expose these objects in a format that allows scripts to view, access, and manipulate them. Each exposed object has associated properties and methods that scripts can then use to interact with an object, as well as affect its behavior or status. For example, a file is an object, and a file has a number of associated properties, such as its name and file extension. By exposing the Windows file system, the WSH enables scripts to access files and their properties and to perform actions, such as renaming a particular file or its file extension. Files also have methods associated with them. Examples of these methods are those that perform the copy and move operations. Using these methods, you can write scripts that can move or copy files from one folder to another or, if you are working on a network, from one computer to another.

Don’t worry if the WSH core object model seems a little confusing right now. You will learn more about it in Chapter 2, “An Introduction to the Windows Script Host.” In addition, you can jump to Appendix C, “The WSH Core Object Model,” at any time for additional insight. The important thing to understand for now is that the WSH enables scripts to access Windows resources (objects) and to change their attributes (properties) or perform actions that affect them (using object methods).

**How Does the WSH Compare to Windows Shell Scripting?**

Windows shell scripts are plain text files that have a .bat or .cmd file extension. Unlike scripts written to work with the WSH, which are written using specific scripting languages like VBScript and JScript, Windows shell scripts are developed using regular Windows commands and a collection of shell-scripting statements. The WSH provides a more complete scripting environment due in large part to its core object model. However, Windows shell scripts still offer a powerful scripting solution. This is partly because you can execute any Windows command or command-line utility from within a shell script. Windows shell scripting also provides a complete collection of programming statements that include support for variables, looping, conditional logic, and procedures. For non-programmers, shell scripts may be easier to read, understand, and modify.

Another difference between scripts written using the WSH and Windows shell scripts is that Windows shell scripts only support text-based communications with the user. In other words, shell scripts cannot display messages or prompt the user for information using graphical pop-up dialog boxes. Windows shell scripting does not provide support for any type of object model like the WSH does. Therefore, Windows shell scripts are not capable of directly interacting with many Windows resources. For example, Windows shell scripts cannot directly edit the Windows Registry or create desktop shortcuts. However, Windows Resource Kits
provide Windows shell scripts with access to a number of command-line utilities that provide indirect access to many Windows resources.

To write shell scripts, you must have a good understanding of Windows commands and their syntax. You must also be comfortable working with the Windows command prompt. Conversely, to effectively use the WSH, you must be well versed in one of its supported scripting languages. There are many cases in which you can accomplish the same task using either Windows shell scripting or the WSH. As a general rule, however, the more complex the task, the more likely you’ll want, or need, to use the WSH.

**Hint**


**WSH Versus Windows PowerShell**

PowerShell is fully integrated into Microsoft’s .NET Framework, providing system administrators with access to system resources. Like VBScript and the WSH, Windows PowerShell is object oriented. PowerShell lets you execute PowerShell commands, referred to as cmdlets, and develop and execute small scripts that use those cmdlets.

Like WSH and VBScript, Windows PowerShell provides access to system resources and can be used to programmatically interact with the file system, Windows Registry, .NET, and WMI. WSH, VBScript, and Windows PowerShell support a robust collection of language constructions like variables, conditional logic, loops, and functions.

Unlike WSH and VBScript, PowerShell does not support the use of pop-up dialog boxes and is restricted to the command line. Unlike VBScript, which is based on the widely popular and easy-to-use BASIC programming language, Windows PowerShell represents a completely new scripting language, which is arguably more difficult for new programmers to learn and understand.

Microsoft has put a lot of time and resources into the development of Windows PowerShell and is promoting it as the future of Windows scripting. However, Microsoft is continuing to support the WSH as a Windows scripting environment, as evidenced by the recent release of WSH 5.8. Microsoft will continue to support the WSH—and for good reason. Companies all over the world have invested significant time and resources in it and have developed hundreds of millions of lines of code that are used to run mission-critical applications and administer servers and workstations.
Companies continue to rely on the WSH and VBScript and extend their use. As such, WSH and VBScript programming will remain essential for application developers and systems administrators for the foreseeable future.

**Hint**


**Understanding How the Windows Shell Works**

Even if you have used Windows operating systems for many years, chances are that you have only limited experience working with the Windows shell. To become a really efficient and proficient script programmer, you’ll need a solid understanding of what the Windows shell is and how to work with it.

An understanding of how to work with the Windows shell is also important when learning how to work with the Cscript.exe execution host, because scripts run by this execution host are generally started from the Windows command prompt. Finally, it’s important to understand the Windows shell when working with the WScript.exe execution host because it provides support for command-line script execution.

You cannot touch the Windows operating system itself. This would be far too complex and difficult. Instead, you must go through an interface. Windows operating systems support two such interfaces:

- **The Windows GUI**. The Windows GUI is provided in the form of the Windows desktop, Start menu, and other graphical elements with which you normally interact when using your computer. The purpose of the GUI is to make the operating system easier to work with.

- **The Windows shell**. The Windows shell is a text-based interface between you or your scripts and the operating system. You communicate with the Windows shell by typing commands in the Windows command prompt; the Windows shell translates these commands into a format that the operating system can process. The operating system then returns any results to the Windows shell, which displays them in the Windows Console.

**Accessing the Windows Console in Normal Mode**

To access the Windows shell and begin working with it using the command prompt, you must first open a Windows Console. To open a Windows Console on a computer running Windows Vista or Windows 7 (see Figure 1.6), open the Start menu, choose All Programs, choose Accessories, and then choose Command Prompt.
To open a Windows Console on Windows 8.1, press the Windows key on the keyboard, type cmd, and then press the Enter key.

Accessing the Windows Console in Elevated Mode

In previous editions of this book, all interaction with the Windows shell was done through the Windows Console in what now can be referred to as normal mode. Starting with Windows Vista, however, Microsoft introduced the concept of elevated command line access, such that there are now two different modes in which the Windows Console can operate. In normal mode, you can execute any command or script as long as it does not require administrative level privileges to run.

**Hint**

For most of the examples and work done in this book, accessing the command prompt through a Windows Console operating in normal mode will be sufficient.

Starting an Elevated Windows Console

To execute commands and scripts requiring elevated access privileges, you need to open an elevated instance of the Windows Console. Doing so is easy. On Windows Vista and Windows 7, all you have to do is open the Start menu, type cmd in the Search field, right-click the cmd utility, and then choose Run as Administrator from the pop-up menu that appears (see Figure 1.7).
In response, Windows Vista or Windows 7 will display the User Account Control dialog box shown in Figure 1.8. This dialog box requires you to confirm your command to run the Windows Console in elevated mode. Click Yes; Windows will display the window shown in Figure 1.9. You can tell the Windows Console is running in elevated mode by the appearance of the word “Administrator” in the text string displayed in the Windows Console's title bar (see Figure 1.9).
To open a Windows Console in elevated mode on Windows 8.1, move the cursor to the bottom-right corner of the screen. This will display a list of icons, the top-most of which is the Search icon. Click this icon and type cmd. When the Windows Console option appears, right-click it and select Run as Administrator from the menu that appears.

Adding a Menu Command for Starting a Elevated Windows Console
If you find yourself frequently needing to work with Windows Console in elevated mode, you can configure a Run as Administer menu option for the Windows Console to make things easier on yourself. The following procedure outlines how to set this up on a computer running Windows 7.

1. Click the Start button, type Regedit in the Search field, and press the Enter key to open the Regedit utility.
2. Click Yes when prompted by the User Account Control dialog box.
3. Navigate to the following key: HKEY_Classes_Root\VBSFile\Shell.
4. Right-click the key, select New > Key, and type RunAs as its name.
5. Right-click the RunAs key and select New > Key. Then type Command as its name.
6. Select the Command key and then double-click its (Default) value to open the Edit String dialog box.
7. Type "C:\Windows\System32\WScript.exe" "%1" %" as its Value data and click OK.
8. Right-click on the Command key and select New > String Value. Then type IsolatedCommand and press the Enter key.

9. Double-click the IsolatedCommand key to display the Edit String dialog box.

10. Type "C:\Windows\System32\WScript.exe" "%1" %" as its Value data and click OK.

11. Close the Regedit utility.

Now the next time you need to run a VBScript using the WScript host with elevated privileges, all you have to do is right-click on the script and select Run as Administrator from the menu that appears, as shown in Figure 1.10.
Interacting with the Windows Shell through the Command Prompt

As you can see, when the Windows Console first opens, it displays information about the version of Windows in use and Microsoft’s copyright information. Then the command prompt appears. Just to the right of the command prompt, you’ll see a blinking cursor or underscore character. This character indicates that the command prompt is ready to accept input. For example, type the command `DIR` and then press the Enter key. The `DIR`, or directory, command instructs Windows to display a list of all the files and folders in the current working directory. The following output shows the results that were returned when I executed this command on my computer:

```
C:\>dir
   Volume in drive C is OS
   Volume Serial Number is 2A2C-3CC5

   Directory of C:\

   03/29/2012  05:37 PM    <DIR>          Apps
   03/29/2012  06:27 PM    <DIR>          Drivers
   10/08/2013  09:44 AM    <DIR>          history
   11/10/2013  11:14 PM    <DIR>          Program Files
   11/10/2013  11:19 PM    <DIR>          Program Files (x86)
   10/04/2013  04:23 PM    <DIR>          Scripts
   01/23/2013  09:46 PM    <DIR>          Temp
   03/03/2013  11:25 AM    <DIR>          Users
   11/02/2013  10:39 AM    <DIR>          Windows

   5 File(s)         40,180 bytes
   17 Dir(s)  283,386,662,912 bytes free

C:\>
```

As you can see, the last line in the output is the Windows command prompt. The Windows shell redisplays the command prompt as soon as the output of the `DIR` command is complete, allowing for the entry of another command. For example, if I have a VBScript named `Hello.vbs` in a folder named Scripts on the computer C: drive, I could execute it by typing `CScript C:\Scripts\Hello.vbs` and pressing the Enter key. After the script finishes its execution, I could type additional commands, run more scripts, or end my Window shell session by closing the Windows Console. The Windows Console is closed just like any other Windows application: by clicking the Close (×) button in the upper-right corner of the Windows screen or by right-clicking on the icon in the upper-left corner of the screen and selecting Close.

**Hint**

You also can close the Windows Console by typing `Exit` and pressing the Enter key.
How Does It All Work?

To execute a script using the WSH, you must first create the script using one of the WSH’s supported scripting languages. In this book, that language is VBScript. Windows operating systems recognize the type of data stored in files based on the file extension assigned to the file. For example, a file with a .txt file extension is a text file. Windows automatically associates files with this file extension with its Notepad application. Therefore, when you double-click on a file with a .txt extension to open it, Windows automatically loads the file into Notepad.

When you create your VBScripts, you need to save them as plain-text files and assign them a .vbs file extension. That way, Windows will know that the file contains VBScripts. In a similar fashion, to write a script using JScript, you must save the file with a .js file extension so that Windows can properly identify it as well.

As long as the WSH has been installed on your computer, all you have to do to execute a script that has been saved with the appropriate file extension is to run it. There are several ways to run a script. One way is to simply double-click on the file. Windows will recognize the file as a script and then automatically process it using the appropriate WSH script engine (based on the script’s file extension). What happens next depends on how you have configured the WSH. By default, the WSH is configured to run all scripts using the WScript.exe execution host, although you can modify this default behavior to make the CScript.exe execution host the default if you want. However, the WScript.exe execution host allows scripts to display messages and to collect text input using graphical pop-up dialog boxes, but the CScript.exe execution host does not. As the script runs in the execution host, it can access and manipulate Windows resources, thanks to the core object model.

**Trap**

Windows runs a script based on the authority of the person who starts it. Therefore, your scripts have no more access to Windows and its resources than you do. If you try to create a script to perform a task that you cannot perform manually via the GUI, your script will not work. If this is the case, you might want to talk with your system administrator to see if you can be assigned additional access permissions and user rights.

Operating System Compatibility

The current version of the WSH is 5.8. This is the fifth version of the WSH released by Microsoft. The four previous versions were versions 5.7, 5.6, 2.0, and 1.0. Depending on which operating system your computer runs, you may already have access to one of these versions. For example, if you are using Windows Server 2008 R2, Windows 7, Windows 8, or Windows 8.1, then you already have WSH 5.8. However, if you work with other Windows operating systems, you likely have an older version of the WSH installed. Table 1.1 provides a list of Windows operating systems and the version of the WSH that is supplied with them.

This book covers WSH 5.8 and VBScript 5.8. However, because WSH 5.6 and WSH 5.7 and VBScript 5.6 and VBScript 5.7 are nearly identical, everything that you learn in this book should apply to these earlier versions.
Microsoft provides free downloads for WSH 5.7 for Windows 2000, Windows XP, and Windows 2003 from the Microsoft Download Center (www.microsoft.com/downloads/). As of the writing of this book, no downloads were available for WSH 5.8. Windows XP users can also upgrade to WSH 5.7 by installing Service Pack 3. WSH upgrades are not longer available for Windows 95, NT, 98, or Me.

### How Do You Install It?

You can download and upgrade to WSH 5.7 on Windows 2000, XP, or 2003. Microsoft provides separate downloads for each of these three operating systems at www.microsoft.com/downloads/. The steps involved in upgrading to version 5.7 once you’ve downloaded it are outlined here:

1. Double-click on the Windows script host file that you downloaded to begin the installation process.
2. Click Next when the Software Update Installation wizard appears.
3. Click I Agree when prompted by the wizard to accept the license agreement.
4. The wizard will complete the installation process. Click Finish when prompted to close the wizard.

When the installation process is complete, the following components will have been installed. At this point, your computer is ready to support the development and execution of VBScripts using the latest and most reliable version of WSH and VBScript.

- Visual Basic Script Edition (VBScript) 5.7
- Jscript 5.7
- Windows Script Host 5.7
**Hint**

WSH 5.7 downloads as a Windows Package Installer file, allowing you to manage it using the Add/Remove Programs option in the Windows Control Panel.

---

**How Does It Work with VBScript?**

Microsoft originally designed VBScript to operate as a Web-scripting language. That means it could run only when embedded within HTML pages that were executed by Internet Explorer. VBScript’s success as a Web-scripting language has always been limited. One reason for this is that Netscape never provided support for it in its Internet browser. In addition, from the beginning, Netscape provided JavaScript free of charge. There was hesitation on the part of many programmers to abandon JavaScript for VBScript, which Microsoft maintained as a proprietary technology, meaning that Microsoft and Microsoft alone owned and controlled VBScript.

Microsoft has since created a modified version of VBScript that is designed to work with the WSH. This version of VBScript lacks many of the features found in browser-based versions of VBScript. For example, it does not work with forms and frames. Then again, as a WSH scripting language, VBScript doesn’t need this functionality because these types of resources are beyond the scope of its environment.

---

**Hello World: Creating and Executing Your First VBScript**

Instead of being embedded within HTML pages, VBScripts run by the WSH are saved as standalone files with a .vbs file extension. For example, take a look at the following VBScript:

```vbnet
MsgBox "Hello World!"
```

As you can see, the script consists of just one line of code. To create this script, open your editor and type the line of code exactly as I’ve shown it here and then save the script as Hello.vbs. That’s it. Now run it: First locate the folder in which you saved the script and then double-click on it. You should see a graphical pop-up dialog box similar to the one shown in Figure 1.11.

Let’s talk about the script that you just wrote and executed. First of all, because you executed it by double-clicking it, you ran it using the default execution host. The default execution host is WScript.exe unless you’ve changed it. (I’ll go over how to change the execution host in the next chapter.) The script itself executes a VBScript function called `MsgBox()`.
The MsgBox() function is a built-in VBScript function that you can call within your scripts to display messages in pop-up dialog boxes. As you can see, the text “Hello World!” was displayed when you ran the script. This VBScript was run using a WSH execution engine (for example VBScript) and one of the WSH’s two execution hosts (either WScript.exe or CScript.exe). However, the code itself was all VBScript.

Let’s modify the script just a little bit to demonstrate how to incorporate the WScript object. The WScript object is one of a small number of objects that make up the WSH core object model. (I’ll go over this object and the rest of the WSH core object model in greater detail in Chapter 2 and Appendix C.) Using your editor, open the Hello.vbs script and modify it so that it looks exactly like the following example:

```
Set WshShl = WScript.CreateObject("WScript.Shell")
WshShl.Popup "Hello World!"
```

Now save the script and run it again. This time, unless you entered a typo, you should see a pop-up dialog box similar to the one shown in Figure 1.12.

![Figure 1.12](image)

As you can see, things look pretty much the same. The same message is displayed and the words “Windows Script H…” are now displayed in the pop-up dialog box’s title bar. Let’s break it down and examine exactly how the script is now written. Don’t worry if you don’t fully understand everything that is covered here—it’s fairly complex and you’ll be better prepared to understand it soon. For now, I’d like you to just read along with the steps I’ll present so that you’ll understand the process involved in creating and executing scripts using VBScript and the WSH.

First, the script uses the Set command to define a variable named WshShl. This variable is then assigned a value using the following expression:

```
WScript.CreateObject("WScript.Shell")
```

This statement executes the WScript object’s CreateObject() method. This method is used to instantiate (that is, create a new instance of) the WshShell object, which is another WSH core object. The second line of code in the example uses the WshShell object’s Popup() method to display a pop-up dialog box.
Hint

The WScript object is one of the WSH's core objects. Do not confuse it with the WSH WScript.exe execution host. It is unfortunate that they share the same name because they are very different.

As the two versions of the previous script show, you can often perform the same task using either a VBScript function or a WSH method. This script also demonstrates how easy script creation and execution can be, and how even a one- or two-line script can perform some pretty neat tricks, such as displaying a pop-up dialog box.

In the Real World

In the previous example, you created your first VBScript by following the steps that I set down. Often, depending on the size and complexity of the script that you're going to develop, you can get away with simply writing the script as you go. More often than not, however, you'll want to take a more methodical approach to script development. First, make sure you know exactly what you want to achieve. Then break the task down into specific steps that, when combined, complete the task. Spend a little time sketching out the design of your script and try to break the script into different sections. Then develop a section at a time, making sure that one section works before moving on to the next. I'll try to point out ways to do this throughout the book.

Executing Your Script from the Command Prompt

In the previous example, you executed your script by double-clicking on it, and everything worked fine. That's because the scripts were written so that they could run from the Windows desktop. Sometimes, however, the execution host you use to run your script can have a big impact on how the script operates. Let's take a look at an example.

1. Open the Hello.vbs script again and replace the contents of the script with the following statement:
   
   ```vbscript
   WScript.Echo "Hello World"
   ```

   This statement uses the WScript object's Echo() method to display a text message.

2. Save the script and execute it by double-clicking it. Unless you have modified the default WSH configuration, the script will run using the WScript.exe execution host. The result is that the message is displayed in a pop-up dialog box.

3. Copy the file to the C: drive on your computer and open a Windows Console.

4. At the command prompt, type `CD \` and press the Enter key. This command changes the current working directory to the root of the C: drive, where Hello.vbs script now resides.

5. Type the following command and press the Enter key:
   
   ```cmd
   CScript Hello.vbs
   ```

   What you see this time is quite different. Instead of a pop-up dialog box, the script’s output is written to the Windows console, as shown in Figure 1.13.
6. As a final experiment, type the following command at the Windows command prompt:

```
WScript Hello.vbs
```

As you see, the message produced by the script is once again displayed in a pop-up dialog box because even though the script was run from the Windows command prompt, the WScript.exe execution host displays its output graphically.

**What Other Scripting Languages Does the WSH Support?**

As I have already alluded to, the WSH supports other languages besides VBScript. Microsoft ships the WSH with both JScript and VBScript. JScript is Microsoft’s implementation of the ECMAScript language, originally developed by Netscape as LiveScript and later renamed JavaScript. Like VBScript, the version of JScript that is shipped with the WSH is a modified version of the browser-based scripting language. Also like VBScript, JScript is a complete programming language replete with support for variables, conditional logic, looping, arrays, and procedures.

JScript’s overall syntax structure is a little more difficult to master than VBScript’s unless you are already familiar with JavaScript. VBScript provides better support for arrays, whereas JScript provides a stronger collection of mathematical functions. JScripts are created as plain-text files and saved with a .js file extension.

**Hint**

To learn more about JScript, check out the JScript Documentation link on http://msdn.microsoft.com/scripting.
In addition to VBScript and JScript, a number of third-party scripting languages are also designed to work with the WSH, as outlined in Table 1.2.

**Table 1.2 Third-Party WSH-Compatible Script Engines**

<table>
<thead>
<tr>
<th>Language</th>
<th>Website</th>
</tr>
</thead>
<tbody>
<tr>
<td>PerlScript</td>
<td><a href="http://www.activestate.com">www.activestate.com</a></td>
</tr>
<tr>
<td>PythonScript</td>
<td><a href="http://www.sourceforge.com">www.sourceforge.com</a></td>
</tr>
<tr>
<td>ooRexx</td>
<td><a href="http://www.oorexx.org">www.oorexx.org</a></td>
</tr>
<tr>
<td>RubyScript</td>
<td><a href="http://www.ruby-lang.org">www.ruby-lang.org</a></td>
</tr>
<tr>
<td>ActiveScript</td>
<td><a href="http://www.php.net">www.php.net</a></td>
</tr>
<tr>
<td>TclScript</td>
<td><a href="http://www.sourceforge.com">www.sourceforge.com</a></td>
</tr>
</tbody>
</table>

### Introducing VBScript

As you now know, VBScript is a scripting language that enables you to develop scripts that automate tasks that would otherwise have to be manually performed in the environment in which they execute. VBScripts are stored as plain-text files with a .vbs file extension and can be created using any text editor. This makes them easy and quick to develop.

Unlike the standalone implementations of many scripting languages, such as Perl or Python, VBScripts cannot execute without an execution host. VBScript was originally designed to execute as text embedded within HTML pages inside the Internet Explorer browser. Over the years, however, Microsoft has extended VBScript’s capabilities to enable it to function in numerous different settings. VBScript is now supported in a number of different environments, including the following:

- **Windows Script Host.** VBScript provides a host automation language for performing system and network tasks.
- **Internet Explorer.** VBScript supplies a client-side Web-scripting language.
- **Microsoft Windows Script Console.** This allows VBScript to be added to third-party applications to incorporate its scripting capabilities.
- **Internet Information Server (IIS) and Active Server Pages (ASP).** VBScript can be embedded into ASP to access local databases and help deliver dynamic Web content.
- **Outlook Express.** VBScript provides the ability to automate a number of Outlook’s functions.

As you can see, after you master VBScript within the context of WSH script development, you’ll have a number of other avenues in which you can begin using your new VBScript programming skills.
VBScript Capabilities

VBScripts cannot execute without an execution host. Therefore, the language’s capabilities vary greatly based on where they are run. For example, when embedded in HTML pages, VBScript can access and manipulate forms, frames, links, images, and other objects that are based on Web pages. When placed inside ASP pages, VBScripts have access to server-based resources such as databases. However, because the purpose of this book is to teach you how to program using VBScript within the context of the WSH, I think it’s best that we focus on the capabilities that VBScript has when executed in this environment.

As I’ll show you throughout this book, you can create games using VBScript and the WSH. While game development is a great way to have fun while learning a new language, it’s important to understand the reason Microsoft enabled VBScript to operate in the WSH and to be familiar with the capabilities that Microsoft has given to VBScript within the context of WSH script development.

VBScript provides programmers with a quick development tool for creating small applications and utilities and for prototyping new applications. System and network administrators use these tools to automate system administrative tasks, such as the following:

- Creating user and group accounts
- Configuring the desktop
- Creating ad hoc reports
- Automating network file, folder, and drive administration
- Managing Windows services
- Administering local and network printers

Some tasks simply take a long time to perform manually or must be done so frequently that they become bothersome. By providing the ability to automate these tasks, VBScript provides a powerful yet easy way to use programming tools. Once developed, script execution can be automated using the Windows scheduling service. This allows you to run your scripts at the times that are most convenient for you.

For example, suppose you wrote a script that reorganizes files on your computer by moving them from various folders into a centralized location. That way, at the end of each month, you can run the script and reorganize a month’s worth of messy file placement. The number of files to be moved may be such that it takes the script a while to complete its work, during which time the computer runs slowly and is no fun to use. Fortunately for you, however, VBScripts can be scheduled. You can set up the execution of this script to run at night, over the weekend, or any time you don’t plan on using your computer.

VBScript’s Roots

Microsoft first released VBScript in 1996 as a Web-based client-side scripting language for Internet Explorer 3.0. At the time, another Web-based client-side scripting language, JavaScript, was already making big waves in the Internet community. Despite the similarity in name, JavaScript had very little in common with Java, which was also fast becoming popular in the mid to late 1990s.
As mentioned, JavaScript’s popularity as a client-side Web-scripting language has continued over the years, while VBScript’s stalled. Even today, the only way to perform client-side Web scripting and to be sure that everyone with an Internet browser has access is to use JavaScript.

Still, Microsoft has remained committed to the development of VBScript over the years. It released VBScript 2.0, along with IIS 3.0, turning VBScript into a server-side Web-development language. Now Web developers could embed VBScripts into their ASP pages, giving them the ability to access local databases and create dynamic HTML pages.

VBScript’s big break came with VBScript 3.0. This version was packaged with multiple Microsoft products, including the following:

- Internet Explorer 4.0
- IIS 4.0
- Outlook 98
- Windows Scripting Host

VBScript 3.0 now could be used as a scripting language for Microsoft’s email client. However, VBScript really took off when it was included as a scripting language for the WSH. Visual Basic programmers, computer administrators, and technology enthusiasts with a background in Visual Basic found VBScript easy to learn. It quickly proved to be a great language for developing small scripts to perform tasks that did not merit the development of a complete standalone application.

Microsoft later released VBScript 4.0 as part of its Microsoft Visual Studio application development suite. Microsoft gave VBScript 4.0 the capability to access the Windows file system; otherwise, VBScript 4.0 remained pretty much unchanged from the previous version.

In 2000, Microsoft released VBScript 5.0 as a component of Windows 2000, which included Internet Explorer 5 and WSH 2.0. In 2001, Microsoft released Windows XP Professional, Windows XP Home Edition, and Internet Explorer 6.0. Along with these goodies came WSH 5.6 and VBScript 5.6. VBScript 5.7 was made available in 2007 as part of WSH 5.7. It was also made available as part of the install of Internet Explorer 7. Likewise, VBScript 5.8 was made available in 2009 as part of WSH 5.8 and as part of the install of Internet Explorer 8.

VBScript’s Cousins: Visual Basic and VBA
VBScript is the third member in a family of three closely related programming languages:

- Visual Basic
- Visual Basic for Applications (VBA)
- VBScript
Visual Basic is the original member of this family. Microsoft first introduced it in 1991. Microsoft has steadily improved Visual Basic, releasing a number of versions along the way. The most current version of Visual Basic is Visual Basic 2012. As a .NET-compliant language, Visual Basic supports Microsoft’s .NET framework.

Visual Basic is generally used to create standalone programs. This means that once written and compiled into executable code, a Visual Basic application does not need anything other than a Windows operating system to execute. Visual Basic earned a reputation very early on for being easy to learn. As a result, it did not take Visual Basic long to become one of the most popular programming languages ever developed. Today Visual Basic is taught in colleges around the world and is used to build applications in companies of all sizes and types.

Visual Basic applications are created using Visual Basic’s built-in integrated development environment (IDE). Visual Basic’s IDE includes a built-in compiler, debugger, help system, and tools for managing Visual Basic projects. Although Visual Basic’s IDE provides a rich and powerful programming development environment, it takes a substantial amount of time and effort to learn. Because of the complexities of its IDE, Visual Basic is not well suited to the development of small scripts. Visual Basic’s strength lies in aiding the development of larger and more complex programs that justify the time and effort required to develop them.

The next language in the Visual Basic family is Visual Basic for Applications (VBA), which Microsoft first released in 1993. VBA represents a subset of Visual Basic and is designed to provide applications with a Visual Basic–like programming language. For example, using VBA for Microsoft Excel, programmers can develop entire applications using features provided by Excel. Similarly, VBA for Microsoft Access provides a powerful programming language for creating applications that require a Microsoft Access database.
Like Visual Basic applications, VBA applications are created using a sophisticated IDE program. Unlike Visual Basic applications, which can be compiled into fully executable programs, VBA can only be compiled into a format known as p-code, which you can think of as partial compilation. Using p-code, VBA code can load and run more quickly than VBScript, which is an interpreted language, but will still run more slowly than a Visual Basic application. VBA also requires a host application such as Microsoft Excel or Microsoft Access.

VBA 7.1 was released as part of Microsoft Office 2010 and is still the current version. Using VBA, you can develop programs for any of the following Microsoft applications:

- Word
- PowerPoint
- Excel
- Outlook
- Access
- FrontPage

**Hint**

To learn more about VBA and Microsoft Excel, check out *Microsoft Excel VBA Programming for the Absolute Beginner*, by Duane Birnbaum. To learn more about VBA and Microsoft Access, check out *Microsoft Access VBA Programming for the Absolute Beginner*, by Michael Vine.

**Back to the Knock Knock Game**

Let’s turn the focus of this chapter back to the development of the Knock Knock game. This project will demonstrate the steps involved in creating and running your first VBScript game. Along the way, you’ll learn how to use VBScript to create a script that can communicate with the user via pop-up dialog boxes. You will also learn a little about conditional programming logic.

**Designing the Game**

The Knock Knock game’s design is very straightforward, involving basic programming techniques. The game begins by displaying the message “Knock Knock” in a pop-up dialog box. It then waits for the player to reply by typing “Who’s there?” The game then replies “Panther” and waits for the player to respond by typing “Panther who?” at which time the punch line, “Panther no panths I’m going swimming” is displayed. If the player fails to exactly type the proper responses at any point of the game, an error message will be displayed inviting the player to try again.
This project will be completed in five steps, as follows:

1. Present the player with the Knock Knock pop-up dialog box and collect the player’s response.
2. Validate the player’s reply and continue the game if appropriate. Otherwise, display an error message.
3. Present the player with the name of the person at the door and collect his or her reply.
4. Validate the player’s reply and continue the game if appropriate. Otherwise, display an error message.
5. Display the game’s punch line.

Starting the Script Development Process
The first step in creating the Knock Knock game is to start your script editor and use it to create an empty VBScript file. For example, to create the script using the Notepad text editor on a computer running Windows XP, you would execute the following steps:

1. Click the Start button, choose All Programs, select Accessories, and choose Notepad. The Notepad application opens.
2. Open the File menu and choose Save. The Save As dialog box appears.
3. Specify the location where you want the script to be stored. Then type KnockKnock.vbs in the File Name field at the bottom of the dialog box and click Save.

The Notepad editor should now display the name of the Knock Knock script in its title bar.

Starting the Game and Collecting Initial User Input
Now let’s begin the script by writing its first VBScript statement. The first thing the game is supposed to do is display a pop-up dialog box with the “Knock Knock” message and then wait for the user response. This task is performed surprisingly easily using VBScript, and can be done with a single statement:

```
Reply1 = InputBox("Knock Knock!")
```

In plain English, this VBScript statement displays a pop-up dialog box with a “Knock Knock” message and then waits for the player to type something into the dialog box’s text field and click the OK button.

Let’s break this statement down into pieces and see how it works. First, the statement executes a built-in VBScript function called `InputBox()`. This function displays a pop-up dialog box with a text entry field that allows the script to collect text input from the player.
The VBScript `InputBox()` function is just one of a number of options for collecting input. The `InputBox()` function facilitates direct interaction with users. When direct user interaction is not required, you can also develop VBScripts that can read input from text files or the Windows Registry. I’ll show you how to read data from text files in Chapter 8, “Storing and Retrieving Data,” and how to interact with the Windows Registry in Chapter 10, “Using the Windows Registry to Configure Script Settings.” You can also create VBScripts that process data passed to them at run-time. I’ll show you how this works in Chapter 4, “Constants, Variables, Arrays, and Dictionaries.”

To communicate with the player, the `InputBox()` function allows you to display a message. In this example, the message is simply “Knock Knock,” but could just as easily be “Hello, what is your name?” or any other question that helps the player understand the type of information the script is trying to collect.

Finally, the text typed by the player in the pop-up dialog box’s text field is temporarily assigned to a variable called `Reply1`. Variables provide scripts with the capability to store and later reference data used by the script.

Functions and variables are fundamental components of VBScript. Unfortunately, it is difficult to write even the simplest scripts without using them. For now, don’t worry too much about them and keep your focus on the overall steps used to create and run the Knock Knock game. I’ll go over the use of variables in great detail in Chapter 4 and the use of functions in Chapter 7, “Using Procedures to Organize Scripts.”

Validating User Input

The player’s role in this game is to first type the phrase “Who’s there?” Any variation in spelling or case will result in an error. After the player has typed this message and clicked the OK button, the script needs to perform a test that validates whether the player is playing the game properly.

The following three lines of code accomplish this task:

```
If Reply1 = "Who's there?" Then
  .
  .
  .
End If
If Reply1 <> "Who's there?" Then MsgBox "Incorrect answer. Try again."
```

The first two lines of actual code—`If Reply1 = "Who's there?" Then` and `End If`—go together. The three dots between these lines of code are placeholders for more statements that will be inserted in the next section. The first of these two lines tests the value of `Reply1`. Remember that `Reply1` is a variable that contains the response typed by the player. This statement checks to see if the values stored in `Reply1` match the phrase “Who’s there?” If there is an exact match, then the lines of code that you will soon place within the first two statements are executed. Otherwise, these statements are not processed. The third line of code
inverts the test performed by the first two lines of code by checking to see if the player’s reply is not equal to (that is, $<>$) the expected phrase. If this is the case, then the rest of the third statement executes the display of an error message. The text performed by the third statement may prove true for a number of reasons, including the following:

- The player clicked the Cancel button.
- The player clicked the OK button without typing a response.
- The player typed an incorrect response.

**Finishing Input Collection**

If you are creating the script as you read along, then your script should now contain the following statements:

```vbscript
Reply1 = InputBox("Knock Knock!")
If Reply1 = "Who's there?" Then
    
    
End If
If Reply1 <> "Who's there?" Then MsgBox "Incorrect answer. Try again."
```

It's now time to add three lines of code that will reside in the lines currently marked with periods. The first of these three lines of code is as follows:

```vbscript
Reply2 = InputBox("Panther!")
```

This statement is very similar to the first statement in the script, except that instead of displaying the message “Knock Knock,” it displays the message “Panther” and then waits for the player to type a response (that is, “Panther who?”). The text typed by the player is then stored in a variable named `Reply2`.

**Validating the User’s Last Response**

The following two lines of code need to be inserted just after the previous statement:

```vbscript
If Reply2 = "Panther who?" Then 
    MsgBox "Panther no panths I'm going swimming."
If Reply2 <> "Panther who?" Then MsgBox "Incorrect answer. Try again."
```

The first line checks to see if the value stored in `Reply2` is equal to the phrase “Who is it?” If it is, then the rest of the statement displays the joke’s punch line. If the player typed something other than “Who is it?” then the second of these two statements executes, displaying a message that informs the player that he did not provide the correct response.
The Final Result
Now let's take a look at the fully assembled script.

Reply1 = InputBox("Knock Knock!")
If Reply1 = "Who's there?" Then
    Reply2 = InputBox("Panther!")
    If Reply2 = "Panther who?" Then
        MsgBox "Panther no panths I'm going swimming."
    If Reply2 <> "Panther who?" Then MsgBox "Incorrect answer. Try again."
End If
If Reply1 <> "Who's there?" Then MsgBox "Incorrect answer. Try again."

As you can see, the script only has seven lines of code, and yet it displays multiple graphical pop-up dialog boxes that collect player text input and display any of three additional messages in pop-up dialog boxes. In addition, this script demonstrates one way of testing player input and then altering the execution of the script based on that input.

Save and then run the script, and make sure everything works as expected. If not, open the script and double-check each statement to make sure you typed it correctly.

Summary
This chapter has covered a lot of ground for an introductory chapter. Not only did you create your first VBScript, but you also learned how to use the WSH to execute it and to incorporate WSH elements within your scripts. In addition, you learned a lot about VBScript and how it relates to other languages that make up the Visual Basic family of programming languages. Finally, you created your first computer game, learning how to collect and validate user input and to display output. All in all, I’d say that this has been a very good start.

Challenges
1. The Knock Knock game is a very simple game. Its main purpose was to introduce you to the basics of script and game development. Try to improve the game by adding additional jokes so that the game does not end after the first joke.
2. Try running the Knock Knock game using both the CScript.exe and WScript.exe WSH execution hosts. How does the execution of the script change and why?
3. See if you can create a new script that prompts you for your name and then displays a personalized greeting message that includes your name. Hint: When displaying the customized greeting message, you will need to concatenate (glue together) the name of the user with a greeting message as follows:

   MsgBox "Greetings " & UserName