G

Creating Your Own Scriptable Objects

Why Create Your Own Objects?

Several chapters have been devoted to discussing just a few of the scriptable objects provided with Windows. In this appendix, I show you how to create objects of your own.

Why would you want to do this? Well, remember what objects are about: They do a particular job while hiding the details of how the job is actually accomplished. Using this "divide-and-conquer" approach has three advantages:

- Faster development and debugging
- Simplification of the scripts and programs using objects
- Code reusability

Let's look at these advantages one by one. First, when creating a new object, your immediate task is to make sure that the object does what it is supposed to do and that its methods and properties do the right thing. Your coding and debugging job is to write a small program to perform a particular, discrete task, which is easier than writing and debugging a huge script with all the steps necessary to attack some larger job.

Second, after your new object has been added to the Windows repertoire and you are writing a script to do some larger maintenance task, you can focus on this larger task without worrying about the details now handled by the object. Because objects are off "somewhere else," your Windows Script Host (WSH) scripts are smaller and easier to write and debug.

Third, you can use a new object's capabilities in many different scripts without repeating the object's code in each of them. If you later find a bug in the object's code, or need to change the way it does its job, you only have to modify one program, not several.

Why do objects make it easier to write programs that are more reliable? Objects can protect the data they hold. An object lets you change its data only on the *object's* terms. For example, whereas you can change a script variable's value willy-nilly, an object can intercept attempts to change a property and can determine whether the value you want to assign is legitimate. The variables that hold the property's value can only be changed from inside the object's program. It's simpler to write correct programs when you can limit what parts of the program have access to your data.

Another good reason to write an object is to provide a new data type for your scripts to use. VBScript provides numbers and text strings and has tools to manipulate them, but when you work with more concrete information, such as people's names, computers' IP addresses, and the like, you find yourself doing the same things over and over: combining first and last names into proper names; validating that an entered IP address is correctly formatted; and so on. Objects enable you to write one program to do these things that you can take advantage of in scripts, Word macros, Visual Basic, C++ programs., and so on—in other words, in any language that can use Component Object Model (COM) objects.

Finally, you can create a new object that either extends or simplifies the capabilities of an existing object. This is called *subclassing*. You can use subclassing to add new properties or methods to an object, such as the built-in `WScript.Network`. The new object can simply pass on most properties and methods to an instance of the original object and only needs to handle the ones it has added. I give an example later in this appendix.

Programming Language Options

The technical term for an object that is usable in WSH programs is *Automation object*. An Automation object has several required attributes:

- It's based on the COM. This means it has a standard application programming interface (API) that enables other Windows programs to gain access to its properties and methods.
- Other programs can "query" the object about its methods and properties. The software has a built-in mechanism for describing its interfaces so any external program can find what properties and methods are available, what arguments they take, and what sorts of data values are returned.
- An object is implemented as a program that represents the object's class; that is, one program takes care of all instances of a given object type.

- The class program has functions to return each of the object's property values, and it has subroutines that accept new values to assign to the object's properties.
- A class program has subroutines or functions to implement each of the object's methods. These might take arguments and return values.

Several languages can be used to create Automation objects. The most common are C++, Visual Basic, VBScript, and JScript.

Visual Basic

Visual Basic is a popular language for COM/Automation object programming. Visual Basic is a superset of the VBScript language: It's largely the same, but has additional powerful features. It has access to the entire Windows API, which means Visual Basic programs can take advantage of any facility Windows offers, from networking and cryptographic encoding to graphical display and database access. Because Visual Basic is a compiled language, object programs are fast. It is designed with COM in mind, so as with VBScript, creating and using objects is relatively easy. You can also develop Automation objects using the free "express" version of Microsoft Visual Studio available at www.microsoft.com/express. These are significant advantages. Be aware, however, that compared to VBScript, the learning curve of Visual Basic is much steeper. You have to learn how to use the Visual Studio development environment, and several steps are involved in creating and installing a compiled Automation object.

Still, Visual Basic is the language of choice for entry-level programmers who need the simplicity of the Visual Basic language but require access to Windows API functions or the higher speed of a compiled language.

C++ and C

C++ and C are the old standby programming languages of systems programmers. They produce fast, efficient programs, have low-level access to all parts of Windows, and are considered somewhat more "highbrow" than Visual Basic. However, these are not easy languages to learn, and working with COM objects in C++ is a *huge* pain. In C, it's practically torture.

If you want to create low-level objects, or fast, compiled objects using Visual Basic, C++, or C, I suggest you visit your local bookstore and look through the books devoted to COM programming before deciding which language to use. If that is too much trouble, you can create new objects using just the scripting techniques described in this book in chapters 1 through 9.

VBScript and JScript

WSH lets you write object programs in VBScript or JScript. There are some limitations:

- The objects can't do anything that you can't do with WSH. For example, the nitty-gritty power of the core Windows API routines is not available.
- The objects run in an interpreted language, so they're somewhat slower than objects created with a compiled language.
- The objects don't stand alone; they require WSH to be installed and enabled on any computer where they're to be used.

Although WSH objects can lack speed and some power, they still give you the benefits of object-oriented programming that I discussed at the beginning of this appendix.

Creating Objects with Windows Script Component Files

You can create useful objects using the Windows Script Component (WSC) technology provided with WSH, using *any* WSH scripting language. With advance apologies to JScript programmers, all the examples in this appendix are written in VBScript.

Note

You can find more information on WSC files by visiting www,microsoft.com and searching for the phrase "Script Components." Select the search result with this phrase as its title. The document introduces a whole section on Script Components under the MSDN section titled "Microsoft Windows Script Technology."

An object based on WSH uses a script program with the usual parts: variables, functions, and subroutines. The script's global variables hold the object's data (its property values), and the script's functions and subroutines serve as the interface between outside programs and the object. They implement the object's methods and properties.

When a client program (that is, a script or other program that wants to use your object) asks Windows to create an instance of your object, Windows locates your WSC file and runs the script inside. The script's variables hold the data for each instance of the object. If client programs create more than one instance, you don't have to worry about keeping track of what data goes with which instance—WSH takes care of this for you. Each time an object is created, WSH sets aside memory to hold all the script's variables. Figure G.1 shows how this works.

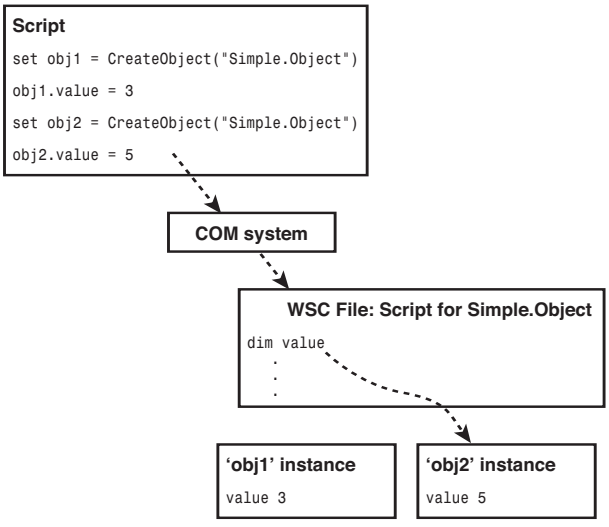

Figure G.1 WSC objects use a script to implement methods, properties, and global variables to hold the data for each object instance.

Each object gets its own separate copy of all the variables, so the script doesn't have to worry about keeping track of what data goes with which object instance. Each instance of the object is separate and doesn't share data with other instances.

WSC File Format

Objects based on scripts have to be packaged in a file with the extension `.wsc`, which stands for Windows Script Component. Script-based objects are referred to as "WSC objects" and script files are referred to as "WSC files" for the remainder of the appendix.

A WSC file contains the script that manages your object and additional information Windows uses to tell client programs or scripts just how your object works: what properties it has, what methods are available, and what arguments it takes. The file is structured as an Extensible Markup Language (XML) file. The various parts of the file are marked with tags that make XML files look a lot like the HTML files used to create web pages.

Let's look at a simple object that has one property named `value`, which can contain any number, and one method named `multiply`, which multiplies the property by some factor. A client script might use such an object the following way:

```
set obj = CreateObject("Simple.Object")  ' create the object

obj.value = 3                            ' set the "value" property to 3
```

```
obj.multiply 4                          ' use the method to multiply value by 4
wscript.echo "Result:", obj.value       ' print the result
```

This should be familiar and sensible after all the objects you read about in the book's first nine chapters.

Now, for the other side: Figure G.2 shows a WSC file that implements this object.

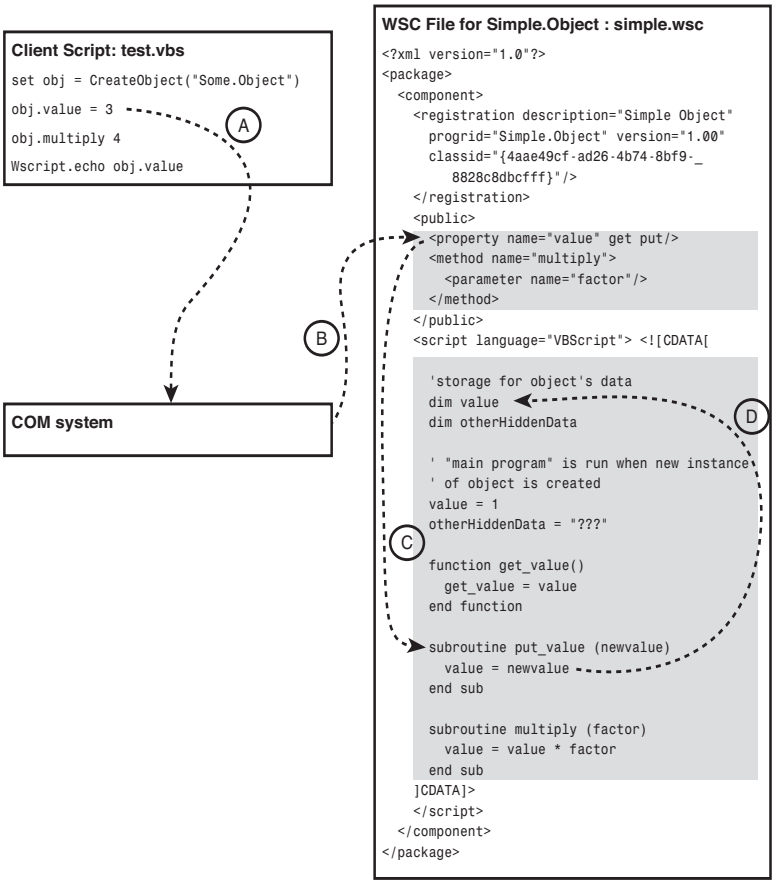

Figure G.2 A WSC file defines an object's properties and methods and contains a script to implement them.

To see how it works, we must take a look at this file, bit by bit. Two parts of the file are shaded in gray. The first is the descriptive part that tells Windows what properties and methods the object has. The second is the script itself—the program that makes the object work. Let's follow what happens when the client script sets the object's `value` property to `3`, as illustrated in Figure G.2:

A. The client script tells the object to set its `value` property to `3`. This request is passed to the Windows COM system.

B. COM identifies the WSC file that is associated with this object and passes along the request. WSH examines the script's property definitions and finds that there is indeed a property named `value`, writing to it is acceptable, and because no alternate subroutine name is specified, the new property value is to be passed to a subroutine named `put_value`.

C. Subroutine `put_value` is called with the value `3` as its argument. The script assigns this new property value to the global variable `value`.

D. The number `3` is stored in `value`, which is located in the memory set aside for just this object instance.

Reading properties or calling methods follows a similar sequence. Note that in this scheme, the object's program maintains its data, and the calling script or program cannot access the object's variables directly. The object's methods and procedures do all the work. The object therefore has complete control over its little universe.

XML Basics

If you look again at the WSC file contents in Figure G.2, besides the VBScript program, you see other text surrounding it. If you've ever created web pages or looked at the Hypertext Markup Language (HTML) files describing them, you might notice the contents of a WSC file are similar. This similiar format is called XML. Before going into the details of this example, we need a crash course in XML terminology. You run into several specialized terms for XML if you read the Microsoft documentation for WSC files.

Items such as `<public>` are called *tags*. Tags are enclosed in angle brackets (< >) and can contain descriptive values within them called *attributes* (such as `name="value"`). Many tags enclose text or other information. For example, a tag such as `<public>` has a corresponding *end tag*, which is written `</public>`. Anything between a tag and its end tag is the tag's *content*. In XML lingo, a tag and its content are called an *element*. For WSC objects, the `<public>` element describes what is to be made available to programs that want to use the object.

Not every element has to have content. You can see this in the `<property/>` and `<parameter/>` tags. The `/>` at the end means there is no corresponding end tag and no content.

Caution

If you're used to working with HTML, beware. Although HTML is flexible on this point, XML requires that every tag *must* either have a matching end tag or end with `/>` to indicate there is no end tag.

Now look at the <registration> tag. It has a several attributes, but no content. There's just the tag followed immediately by its end tag. Could I have written <registration description=[…]/> and omitted </registration>? Yes, but this version is also correct. In this case, it appears this way because the Windows Script Component Wizard, which is discussed in the next section, wrote the bulk of this particular script, and it used the lengthier version.

The strange tag that starts with <?xml is called a *directive*. Directives don't have end tags because they're technically not really tags, and they never enclose content. They provide special information about how the XML is to be interpreted.

By default, XML doesn't care about extra blank lines, spaces inside tags, or the capitalization of tag and attribute names. To an XML application, <tag>something</tag> and

```
<tag>
something
</tag>
```

are equivalent, as are <tag> and <TAG> as well as <tag> and < tag >. Typically, attribute values are enclosed in single or double quotation marks (' or "), but if an attribute value has no embedded spaces, XML tolerates <tag attribute=something> and <tag attribute="something">. For the value "something else", you need the quotes.

However, if you put the <?xml?> directive in the first line of an XML file, this tells the XML application it is to use *strict compliance* rules, thus making formatting requirements much stricter. This can be a good thing, because it's easier for Windows and XML editing programs to detect errors. This topic is discussed later in this appendix under the section called "WSC File Format Reference."

Understanding the Example

Now, let's look back at the sample object script. The <package> element encloses the whole file. Everything inside the <component> element describes a single object. A WSC file can have several <component> elements, each of which manages a separate object type.

The <registration> element gives the object its name, its progid (the name other programs use in calls to CreateObject), its version number, and its classid. A classid consists of long, essentially random numbers that distinguish one object from all others in the world. You don't need to make these numbers up; the Windows Script Component Wizard I discuss later in this appendix provides one for you. All this information is ultimately stored in the Windows Registry so a program can say CreateObject("Simple.Object") and Windows knows to run your script.

The <public> element lists all the object's properties and methods, which are described by the <property> and <method> elements it contains.

Note

An object can also define *events*, which provide a way for the object to trigger subroutines in the calling program when the object deems it necessary. Events are often used in web server and web browser scripts. However, they're not frequently used in WSH scripts and are not covered in this book.

The markup for the `value` property has the attributes `get` and `put`, which indicate the value can be read and written, respectively. That is, it's a read/write property. The markup for the `multiply` method lists one parameter: `factor`. Notice the data types are not specified as Integer, String, or other types. For WSC objects, all parameters and return values are type Variant.

The `<public>` information is made available to any program that wants to use `Simple.Object`.

Finally, there is the script itself—the program that actually implements the object. The strange `<![CDATA[` tag and its end tag, `]]>` tell XML what's inside is not XML but rather "private" data for WSH. Otherwise, XML might interpret the < and > symbols in the program as XML tag markers.

The program itself is straightforward. The script's "main program" is run when an object of this type is created. Here, the script declares a global variable named `value` and initializes it to `1`. This variable holds the object's one property.

The script has the functions `get_value` and `put_value`, which return and set the property, respectively. These are called *accessor* functions, and they're invoked by WSH when a client program using this object uses or assigns to the property. In this script, the "get" subroutine returns the property's value from the variable it's kept in, and the "put" subroutine only has to store a new value in the variable.

Note

In this example, I use a variable named "value" to represent the property named "value," but there is no requirement that the names match up. The variable could have been named "bobsyouruncle," as long as I wrote accessor functions `get_value` and `put_value` to use it. Later in this appendix, I discuss how to create a "bound" property that doesn't need to use accessor functions.

Finally, the subroutine `multiply` takes care of the multiply method.

After the script component has been registered by typing

```
regsvr32 c:\path\simple.wsc
```

at an elevated command prompt (I discuss this later, just accept it as a magical incantation for now), it works. The following is the sample client script shown earlier:

```
                                        ' Example script scriptxh01.vbs
set obj = CreateObject("Simple.Object")  ' create the object
```

```
obj.value = 3                          ' set the "value" property to 3
obj.multiply 4                         ' use the method to multiply value by 4
wscript.echo "Result:", obj.value      ' print the result
```

It prints out the value `12`, just as it should. The following is a more complex client script:

```
                                        ' Example script scriptxh02.vbs
set obj1 = CreateObject("Simple.Object")  ' create an object
set obj2 = CreateObject("Simple.Object")  ' create another instance

obj1.value = 3                          ' set the properties to different
obj2.value = 5                          ' values

wscript.echo "value 1:", obj1.value
wscript.echo "value 2:", obj2.value
```

It prints `3` and `5`, proving that there really are two separate instances of the object and two separate copies of the script's variable `value`.

WSC File Format Reference

The overall structure of a WSC file is shown here. Not every one of these directives and elements must be used in every case; this listing just shows the ordering that is typically used:

```
<?xml?>            <!-- optional                                              -->
<package>          <!-- required only if there is more than one component    -->
  <comment>        <!-- can appear anywhere in the file                       -->
  <component>      <!-- there can be more than one component element         -->
    <?component?>  <!-- optional                                              -->
    <public>       <!-- there can be any number of property, method, element -->
      <property><get/><put/></property>…
      <method><parameter/>…</method>…
      <event>…
    </public>
    <registration>
    <object/>…     <!-- zero or more                                      -->
    <reference/>…  <!-- zero or more                                      -->
    <resource>…    <!-- zero or more                                      -->
    <script>
  </component>…    <!-- can be followed by another component              -->
</package>
```

Reference List G.1 describes all these tags and their attributes.

Note

You can also find information on WSC in Microsoft's Windows Script Documentation. Visit www.microsoft.com and search for "Windows Script Documentation" using the quotes. Download file script56.chm, and then look at the section titled "Script Components." If Windows doesn't display the help file contents after you download it, right-click its icon, select Properties, and click Unblock.

Before we get into the further details, here are some notes on the syntax descriptions in Reference List G.1:

- Text in `this typeface` is typed literally; text in *`italics`* is replaced with an appropriate name or value.
- Square brackets, [ ], enclose optional items; curly brackets, { }, indicate a set of alternate choices that are separated by vertical bars, |; and ellipses, …, indicate that the previous item can be repeated any number of times. (An exception to this is in the CDATA directive in which the square brackets must be typed literally.)
- XML ignores extra whitespace or line breaks between tags, attributes, or elements. However, text inside `<description>`, `<resource>`, `<helptext>`, and other display elements is displayed literally, so you might want to type such text in a concise format. For example, you want to type

  ```
  <description>text goes here</description>
  ```

  rather than

  ```
  <description>
      text goes here
  </description>
  ```

  In the latter case, the carriage returns and indenting spaces would be printed when the script prints its usage information.
- This reference only lists WSC elements appropriate for scripting of objects for use with WSH. Elements used only when creating objects for Active Server Pages (ASP) scripting and Dynamic HTML (DHTML) web pages are not listed.

REFERENCE LIST G.1 WSC Tag Listing

`<?XML Version="1.0" [standalone="yes"]>`

Requests that the WSH interpreter perform strict XML validation. The `<?XML?>` element in the tag accomplishes this. With validation enabled, XML's stringent formatting and interpretation rules are applied. "Strict" mode is typically required when you are editing the WSC or WSF file with an XML editor.

Strict mode affects the interpretation of the file in the following ways:

- **Case sensitivity**. In strict mode, element names and attribute names are case sensitive. In default mode, case is ignored.
- **Attribute quotation**. In strict mode, all attribute values must be enclosed in single (') or double (") quotes. In default mode, quotes are required only when the attribute value has embedded spaces.
- **Opacity**. In strict mode, the XML parser can "see" into the `<script>` element. Therefore, the content of each `<script>` element (script program code in

VBScript, JScript, or another scripting language) must be enclosed in a `<![CDATA[ … ]]>` section to prevent it from being interpreted as XML. In default mode, `<script>` elements are opaque; the parser does not look into them, and the `<![CDATA [ … ]]>` tag must not be used inside them.

If present, the `<?XML?>` element must be the first line in the file and must not be preceded by blank lines. The `version` number should be specified as `1.0`. The optional `standalone` attribute indicates that the XML file does not reference an external Document Type Definition (DTD) and can be omitted or specified as `"yes"`.

`<?component error="`*value*`" debug="`*value*`" ?>`

Enables the reporting of error messages from and the debugging of the component's script. By default, errors are *not* reported and debugging of component scripts is *not* enabled. The `<?Component?>` element lets you alter this default behavior by specifying `True` or `False` to enable or disable error message reporting and debugging with the Windows Script Debugger or an equivalent Script Debugger.

Place this directive as the first item inside the `<component>` element.

```
<! [CDATA[
    protected text
   ⋮
]]>
```

Encapsulates script code inside a `<script>` element and indicates it is not to be treated as XML markup. The `CDATA` directive in the tag accomplishes this. The `CDATA` section is used only in conjunction with the `<?XML?>` directive; without `<?XML?>`, the presence of `CDATA` markup generates a syntax error.

Note

Here the square bracket characters don't indicate optional parts; they're typed literally as part of the tag. The start tag is `<![CDATA[` and the end tag is `]]>`.

```
<!-- any text
    ⋮
-->
```

Treats anything inside `<!-- … -->` as comment text, including any XML markup. This can be used to comment out blocks of code during testing or to enter comments or descriptive information. You can also use the `<comment>` element to enter text comments, but `<comment>` cannot contain any XML tags or other markup.

```
<comment>
    any text
    ⋮
</comment>
```

Indicates comment text that is to be ignored by the parser. You can embed multiple lines in a comment, but not XML tags. `<comment>` tags are typically used to enter notes to yourself about how the script works.

```
<component [id="componentID"]>
    component definition: <registration>, <public>,
    and <script> elements
    ⋮
</component>
```

Encapsulates the definition of a WSC (object). If an `id` is specified, this is the name that appears as the class name in object browsers. The default value for `id` is `"ComponentCoClass"`. If multiple components are defined in one WSC file, the `<component>` elements must be enclosed in a `<package>` element, and each component must have a distinct `id` property.

```
<event name="name" [dispid="id"]/>
```

Defines an event the object can generate. Events are beyond the scope of this book.

```
<get [internalName="functionname"]/>
```

Uses the inside of a property element and indicates the property is readable and a function supplies the value. When the object client's program requests the value of the property, WSH calls the function `get_propertyname()`, which must be defined in the script. An alternate function can be specified with the `internalName` attribute.

```
<method name="methodname" [internalName="functionname"] [dispid="dispID"]>
    [<parameter name="paramname"/>…]
</method>
```

Declares a method that the object provides. This tag is placed inside `<public>`. The associated script must implement a function or subroutine of the same name as the method, or you might specify an alternate name with the `internalName` attribute. In either case, the procedure must exist; otherwise, WSH generates an error. The method's arguments are specified by `<parameter/>` tags placed between `<method>` and `</method>`.

One method might be designated the default method by specifying attribute `dispid="0"`. You can find more information about declaring methods later in this appendix under the section titled "Defining Properties and Methods."

```
<object id="name" (classid="clsid:GUID" | progid="progid")
   events="boolval"/>
```

Creates an instance of a specified object, in a global variable with the indicated *name*, as if you had used the statement `set name = CreateObject(progid)`. You must specify either the object's `progid` name (for example, `"Scripting.FileSystemObject"`) or its `classid`. Be sure to close the object tag with `/>`. The `events` attribute specifies whether the script is prepared to handle events fired by the object. Events are beyond the scope of this book.

```
<package>
    one or more <component> elements
    ⋮
</package>
```

Encloses one or more separate components in a WSC file. The <package> element is optional when the file contains only one component.

```
<property name="propertyname" [internalName="varname"]/>
                or
<property name="propertyname" [get] [put]/>
                or
<property name="propertyname">
    [<get [internalName="functionname"]/>]
    [<put [internalName="subroutinename"]/>]
</property>
```

Declares a property provided by the object. This tag is used inside <public>.

If neither get nor put are specified as attributes or as tags, the property is bound to a variable of the same name as the property and is read/write. This is the first syntax form shown. You can specify a different variable with the internalName attribute.

If you want to use accessor functions, the property must be designated as readable and/or writable with get and/or put either as attribute names or as <get/> and/or <put/> tags, using one of the other two forms shown.

By default, the property is read by function get_*propertyname*() and written by subroutine put_*propertyname*(*newvalue*), but alternate names can be specified with the <get/> and/or <put/> tags.

You can find more information about specifying properties later in this appendix under the section titled "Defining Properties and Methods."

```
<public>
    Interface definition: <property>, <method>, <event> tags
</public>
```

Contains <property>, <method>, and/or <event> tags to define the public interface of your object. The public element in the tag accomplishes this.

Every property and method listed in the interface definition *must* correspond to an actual variable or function in the script code, even if the methods or properties are never actually used, as mentioned under <method> and <property>.

```
<put [internalName="subroutinename"]/>
```

Uses the inside of a property element and indicates the property is writable by a function or subroutine. When the object client's program assigns a new value to the property, WSH calls put_*propertyname*(*newvalue*) with the new value as its argument. An alternate procedure name can be specified with the internalName attribute.

```
<reference {object="progid" | guid="GUID"}
    [version="version"]/>
```

Instructs WSH to load the type library for a specified object. In effect, this loads all the predefined constants associated with an object type. The object can be specified by its `progid` name (for example, to load the WMI constants, `object="WbemScripting.SWbemLocator"`) or its GUID number. By default, WSH loads the most recent version of the object class, but if necessary, you can specify an older version with the `version` attribute. Be sure to close the tag with `/>`.

```
<registration progid="progid" [classid="GUID"]
    [description="text"] [version="number"]
    [remotable="value"]>
  [<script>
      <! [CDATA[
      registration and unregistration script
      ]]>
   </script>]
</registration>
```

Defines the attributes necessary to register the object with Windows. This information is used by the `regsvr32` command, which is described later in the appendix. Here are the attributes:

- **`progid`**—The text name for this object. It is used, for example, by client programs in calls to `CreateObject()`. The usual form for a `progid` is `"category.objectType"`, where *`category`* is a name that describes an entire category of object type (for instance, `"MyObjects"`) and *`objecttype`* specifies a particular object. This is entirely up to you, but you must be sure that the name you choose doesn't conflict with an existing object. To check, run `regedit` and look under `HKEY_CLASSES` to verify the name you want to use is available.
- **`classid`**—The class ID number for this object (a string with lots of numbers inside curly braces). If you omit this, the script registration program that I describe later creates one when it installs the script component. However, it's best to specify one so the `classid` is the same on every computer on which your object is installed. The easiest way to get a valid GUID is to use the Windows Script Component Wizard to create a skeleton WSC file. If you have Microsoft Visual Studio, you can also use the GUIDGEN program—if you do, select Registry Form. There are also websites that can generate truly random GUIDs for you. But do not attempt to just make up a GUID in your head!
- **`description`**—A brief description of the purpose of the object. This information is displayed in object browsers.
- **`version`**—The version number for this object ("1"). If you make changes to an object that are not backward-compatible and have programs or scripts that depend on the old object's behavior, you can create different versions of the same object in different WSC files or different `<component>` elements, and have them all available. If you specify a version number, use only digits (no decimal points or other characters).

- **remotable**—Can be specified with the value "True" if you want to allow this object to be created by remote scripts (scripts running on other computers) using the remote server version of the CreateObject function. When the WSC script is installed on a given computer, remotable="True" makes it possible for the WSC script to be activated by scripts or programs running on other computers.

Normally, the attributes of the <registration> tag are enough to let Windows register and use the object. When this is the case, you can use the <registration *attributes* /> form of the tag and omit the </registration> end tag.

If you need to perform additional actions when the object is installed—for example, if the object depends on finding certain Registry entries—you can specify a script to be run when the object is registered and unregistered. The registration script should be placed inside a <script> element between <registration> and </registration>. If the registration script defines a function named register(), it runs when the object is registered. The function can install Registry keys, check for the existence of required files, and so on. If the script defines a function named unregister(), it runs when the object is unregistered. This function can remove Registry information, although doing so is probably unnecessary.

WSC objects can be registered with Windows using the following command at an elevated Windows command prompt: regsvr32 *drive*:*path**filename*.wsc

<resource id="*resourceid***"><![CDATA[***text or number***]]>**
</*resource*>

or

<resource id="*resourceid***">***text or number***</*resource*>**

Creates a named value that can be retrieved in the script using the GetResource() function. <resource> tags provide a way to concentrate all language-specific text in one place in the file so alternate language versions can be created at a later date. A <resource> tag assigns a name *resourceid* to some text or numeric data. The *resourceid* string is passed to GetResource() to retrieve the data into the script. The content should be encased in <![CDATA[…]]> if the WSC file uses strict XML compliance. For more information about <resource>, see the section titled "Defining Resources" later in the appendix.

```
<script [language="name"]>
<![CDATA[
    script code
    ⋮
]]>
</script>
```

Contains the actual script program that includes the object's initialization procedure, methods, properties, and any additional subroutines and functions necessary. You can specify the script language; the default is VBScript.

If you used the <?xml?> strict-compliance directive at the top of your script, you should enclose the script inside <![CDATA[and]]> to prevent any < or > characters in your script from being interpreted as tags. If you did not use the <?xml?> directive, do not use <![CDATA[and]]>.

Creating a WSC

The first step in creating a component is, of course, deciding what you want your object to do. You should spend some time thinking about what its properties should be and what its methods should do. You can even have some script programs already written that do most of the work that the object will be doing. After you've made your design, the next step is to create the WSC file that contains your object's property and method definitions, and the script program itself.

Setting up even a simple WSC object is pretty tricky; there's a lot to type, every tag has to have its proper end mark or end tag, and every *i* has to be dotted and *t* crossed. Thankfully, Microsoft has a wizard program that gives you correctly formatted WSC files for your objects; all you have to do is fill in the program steps for the methods and procedures.

Using the Windows Script Component Wizard

You can download the wizard from www.msdn.microsoft.com. Search for "Windows Script Component Wizard" (with the quotation marks). Follow the download and installation instructions. When you install the wizard, it appears on your Start menu under All Programs, Microsoft Windows Script, Windows Script Component Wizard.

The wizard displays six screens, requesting the following information:

- The name to give the object and the WSC file, the object's desired `progid` (the name you want to use in `CreateObject` calls), and the version number, as shown in Figure G.3.
- The language to use (VBScript or JScript), whether to enable the implements feature for DHTML and ASP scripting (this is not applicable to WSC objects), and whether to enable error checking and debugging. I recommend to select both Error Checking and Debugging when you are creating a new object.
- Properties to define such as their read/write attributes and default values, as shown in Figure G.4.
- Methods to define and their argument names, if any.
- Events to define, which are not discussed in this book.

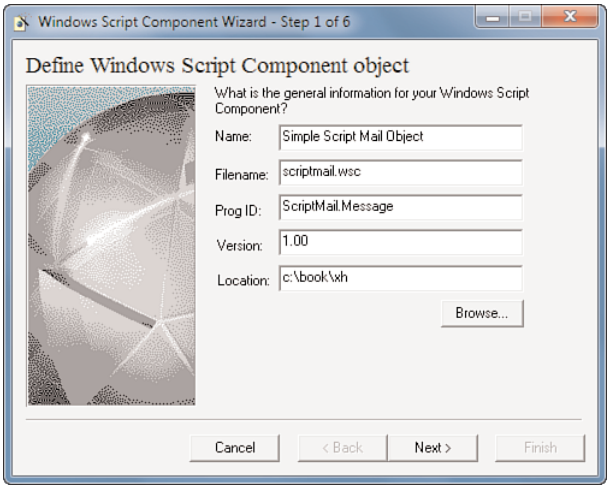

Figure G.3 The general information page lets you assign the name and *progid* for the object.

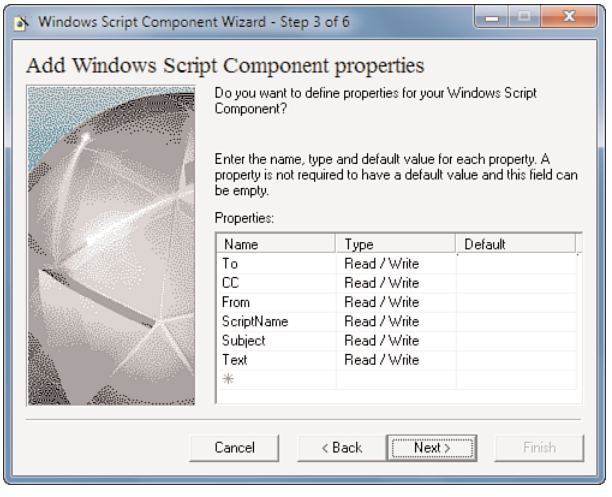

Figure G.4 The properties form lets you define properties and set initial (default) values.

The last screen lets you confirm the information entered. When you click Finish, the wizard writes a `.wsc` file with proper XML tags to define the properties and methods you specified. The script section contains definitions of variables to hold the properties and basic `get_`, `put_`, and methods functions, followed by any additional helper functions and subroutines you want to use. All you have to do is flesh out these functions to get a working object.

Tip

When you're first writing and debugging a script, the `<?component error="yes" debug="yes"?>` element enables script debugging. If you're using VBScript, you can use the `stop` statement in your procedures, which stops the script program and activates the Script Debugger. You can then trace through the script and use breakpoints to help debug it.

In the next sections, I discuss the formatted XML needed to describe an object. Regardless of whether you use the wizard, you need to know how the file is organized.

Defining Properties and Methods

The `<public>` section of the WSC file lists the object's properties and methods, and it describes how they're linked up to your script. As shown earlier in this appendix, the object's properties and methods are managed by functions and subroutines in the associated script program. The names of the properties and methods and their associated program parts are defined by `<property>` and `<method>` elements.

Each of your object's properties must be defined by a `<property>` element.

If you look at the `<property>` syntax in Reference List G.1, you'll see that there are several different allowed formats for this element. You can write it as a single, contentless tag using `<property ...attributes/>`, or you can use the end-tag version: `<property attributes>…</property>`.

Binding a Property to a Variable

Properties can be implemented in one of two ways: They can be directly *bound* (linked) to a global variable in the script, or they can be implemented by functions that return and set the property's value. The first method can be used when the property is a simple value that can be stored in a variable and its value doesn't need to be validated by the object or determined from other variables or objects. The client's program is able to assign any value to the property; WSH simply sticks the value into the associated variable. This method is simpler than providing `get_` and `put_` functions, but you do lose the capability to validate the values being stored in the property. Properties bound to variables are always read/write.

To specify that a property should be stored directly in a variable, omit `get` and `put` as attributes and as tags. By default, the property is stored in a global variable of the same name as the property; you can specify a different variable name with the `internalName` attribute in the `<property>` tag. You might need to do this if the property has a name that is not a legal variable name in your scripting language (for example, "to" in VBScript).

Here are some examples that show how this works:

- A read/write property named `color` stored in the variable `color`:

```
<property name="color"/>
```

- A read/write property named `To` stored in the variable `msgTo`:

```
<property name="To" internalName="msgTo"/>
```

Your script should initialize the property variable to a sensible default value in the script's "main body" (that is, before any of its functions or subroutines).

Implementing a Property with Functions

The second method for managing properties is to bind them to functions that set and return the property value. These are called *accessor functions.* This technique lets you inspect and validate values the user is trying to assign to the property, and it must be used when the property is obtained from another object (as is the case when you are subclassing another object) or is calculated from other values.

If you specify `get` and/or `put` as attributes of the `<property/>` tag, or as `<get/>` or `<put/>` tags inside the `<property>` element, WSH assumes the property is read and written with functions you have supplied. By default, WSH calls a function named `get_`*`propertyname`*`()` when the client wants to read the property's value, and it calls a subroutine named `put_`*`propertyname`*`(`*`newvalue`*`)` when the client wants to write a new value to the property, with the new value passed as an argument.

As before, you must specify whether reading and/or writing is allowed using the `put` and `get` attributes or tags. If you want to specify a different name for the `get` or `put` functions, you must use the long form and supply the new function name as the `internalName` attribute of the `<get/>` or `<put/>` tag. Here are some examples showing the alternatives:

- A read-write property named `age` whose value is returned by a function named `get_age()` and set by a function named `put_age(`*`newvalue`*`)`:

```
<property name="age" get put/>
```

or

```
<property name="age"><get/><put/></property>
```

- A read-only property named `age` whose value is returned by function `calculate_age()`:

```
<property name="age">
    <get internalname="calculate_age"/>
</property>
```

Note

Whether you use a variable or functions to implement a property, the variable or functions *must* be defined in your script, even if they are never used; otherwise, WSH generates an error. If you don't use the `<?component?>` directive with attribute `error="True"`, you get an "unspecified error." If you do use the directive, you get the message "The property accessor is not defined: *MissingItemName*."

Defining Methods

Methods are subroutines or functions provided as part of the object. Like any other subroutine or function, methods can take arguments and return a value to the client.

By default, when the client invokes your object's method, WSH calls a function or subroutine of the same name as the method. An alternate procedure name can be specified with the `internalName` property. If the method does not return a value, the associated procedure can be a function or a subroutine. If the method is to return a value, the associated procedure must be a function.

The `dispid` attribute can be used to specify the COM dispatch ID for the method. This has uses beyond the scope of this book, but for our purposes it's handy to know that you can specify `dispid=0` for *one* of the object's methods, making this the object's *default* method. The default method should return a value such as a name or other identifier because this is the value WSH uses when you attempt to print the value of the object itself. That is,

```
set obj = CreateObject("Some.WSCObject")
WScript.echo obj
```

prints the value returned by the default method.

If the method takes arguments, they must be listed, in order, by one or more `<parameter/>` tags placed between `<method>` and `</method>`. If the method takes no arguments, the method tag can be closed with `/>`, and if so, the `</method>` end tag is omitted.

If you declare a method with a `<method>` tag, the associated function or subroutine *must* exist in your script, even if it is never used; otherwise, WSH generates an error. If you don't use the `<?component?>` directive with the attribute `error="True"`, you get an "unspecified error." If you do use the directive, you get the message, "The method is not defined: *methodname*."

Using Other Objects and Type Libraries

If your object script uses other helper objects, such as `Scripting.FileSystemObject`, instead of creating them with a `CreateObject()` call, you can declare them with the `<object>` tag. The `<object>` tag creates an object reference in a global variable you can use in your script. For example, using

```
<object id="msg" progid="CDO.Message">
```

accomplishes the same thing as beginning your script with something like this:

```
dim msg
set fso = CreateObject("CDO.Message")
```

Strangely, although you would expect the `<object/>` tag to automatically import the object's predefined constants while it's creating the object, it doesn't. You have to use the `<reference/>` tag to get those.

If want to use the predefined constants associated with an object, you can pull in the definitions with a `<reference/>` tag. For instance, in the previous example, if I want to create a `CDO.Message` and use its predefined constants, I could use

```
<component>
   <registration>…
   <public>…
   <object id="msg" progid="CDO.Message"/>
   <reference object="CDO.Message"/>
   <script>
   ⋮
</component>
```

This connects the script to the CDO *type library*, which WSH uses to import all the CDO constants. You can use `<reference/>` whether you use `<object/>` to create objects.

Note

In rare cases, `<reference/>` doesn't work due to problems with COM, WSH, or the referenced object itself. If you find your script or new object stalls out for 30 seconds or so when you try to run it, or you get an error saying the `<reference/>` tag cannot load the required type information, you might be better off removing the `<reference/>` tag and entering the constant definitions into your program by hand. This is unfortunately the case with the Active Directory ADSI objects; `<reference object="ADs"/>` does not work.

Defining Resources

If you plan to make versions of your script in several languages, the `<resource>` tag lets you put all language-specific text in one place in your script file. This makes it easier to edit and maintain the script. Here's how it works: Create `<resource>` elements for

each of the text strings or numbers you want to maintain in separate localized versions. Give each a distinct name in the `id` attribute. Place the text or numbers between `<resource>` and `</resource>`, being careful not to add any extra carriage returns or spaces. If you used the `<?xml?>` strict-compliance directive at the top of your script, you should enclose the text with `<![CDATA[` and `]]>` to prevent any `<` or `>` characters in your text from being interpreted as tags.

Resource elements *cannot* be enclosed in the `<public>` or `<script>` elements of your WSC file. They must be placed inside the `<component>` element at the same level as `<public>` and `<script>`.

Then, in the body of your script, use the function `GetResource("idvalue")` to retrieve the text or number. An example of this would be the following:

```
<? xml version="1.0">
<component id="MyObjects.Multilingual">
    <public>
        <property name="value"/>
        ⋮
    </public>

    <resource id="uninit"><![CDATA[Value has not been set!]]></resource>
    <resource id="msg"><![CDATA[Here is the message:]]></resource>
    ⋮

    <script language="VBScript"><![CDATA[
        dim value
        value = GetResource("uninit")
        ⋮

        wscript.echo GetResource("msg")
     ]]></script>
</component>
```

With all the language-specific information in one place, it's easier to create different versions later.

Registering the Component

When you have created a WSC object file, you need to register the script with Windows to begin testing and using it. This is done at an elevated Windows command prompt with the command,

```
regsvr32 drive:\path\filename.wsc
```

where *drive:\path\filename*`.wsc` is the full pathname of the WSC file. `Regsvr32` is used to register objects.

Note

To open an elevated command prompt using Windows 7 or Vista, click Start, type **cmd** in the Search box, right-click the result cmd.exe, and select Run As Administrator. On Windows XP, log on as a Computer Administrator or Power User and open a regular Command Prompt window.

`Regsvr32` stores the object's `classid`, `progid`, and location information in the Windows Registry, where it's available to any Windows program that wants to create your new object type.

If you need to relocate the WSC file, you need to reregister the object using its new pathname. If you want to remove the object script entirely, unregister the object with the following command line:

```
regsvr32 /u drive:\path\filename.wsc
```

Tip

You can also quickly register and unregister WSC files by locating them in Windows Explorer. Right-click and select Register or Unregister.

Testing

When you are debugging and testing a new WSC object for the first time, be sure to place

```
<?component error="True" debug="True"?>
```

on the line just after each `<component>` tag. This lets WSH display error messages when there are syntax or other programming errors in the script. It also lets you trace into the object's script with the Windows Script Debugger.

You should write small test scripts to exercise all aspects of your object. Besides testing each of the properties and methods to ensure they work correctly, be sure to see whether the object behaves gracefully when it's given invalid data. Give it invalid parameter values and data types, try methods out of order, and if you fail to initialize properties, make sure the object still behaves well.

However, debugging WSC scripts isn't a simple matter of running the script with the following command line:

```
cscript //D myobject.wsc
```

Remember, object scripts are run at the behest of some other client program when that program creates an object defined by your script file. If you use a script program to create your object and debug that, you find that the debugger doesn't step through the `CreateObject` call into the WSC file. How do you debug a WSC file, then?

The solution is to use your scripting language's breakpoint statement to cause the debugger to turn its attention to the object script. In VBScript, use the `stop` statement. In JScript, use `debugger;`. Place the appropriate statement somewhere in your script program: You can place it in the main body of the script to cause a breakpoint when the object is first created, or you can place it in a method or procedure function. If you include the `<?component debug="true"?>` directive, when the breakpoint statement is encountered, you are able to set other breakpoints in the script file, as needed, and debug it in the usual way.

Note

You can't use `WScript.echo` to print values and debugging information from a WSC script. The only way to see inside the object is to use the Script Debugger or an alternate debugger such as Visual Studio.

After the object has been thoroughly tested and debugged, you might want to remove the `<?component?>` directive or change the attribute values to `False`. This disables any `stop` or `debugger;` statements and prevents the object from generating error messages of its own.

COM objects generally don't generate error message by themselves. Your object's script program should do all it can to avoid errors and to handle bad situations gracefully. If it would be helpful, you could communicate error information back to client programs by defining an "error" property for your object that indicates when an error has occurred by returning a number, a descriptive message, or the value `True`.

Using Scripted Objects from Other Programs

After your WSC file is registered and working, you can create the object it represents just as you would any other object—built in or otherwise—in any program that supports Automation objects. Of course, you can use `CreateObject("yourobjectname")` in WSH programs, but also in C++, Visual Basic programs, and scripts written in Visual Basic for Applications (VBA).

Deploying the Object to Other Computers

To use the object on other computers in your organization, you must either copy the WSC file to a specific location on each of your organization's computers or make the WSC file available on a shared folder or web server that is accessible by each computer.

Then, you have to register the object on each computer. There are two ways of doing this. You can run the `regsvr32` program from an elevated command prompt on each computer, using one of these versions of the command:

- **regsvr32 *drive:\path\filename.wsc***—Use this version if the file is copied to a specific place on each computer.
- **regsvr32 *\\server\sharename\path\filename.wsc***—Use this version if the file is located on a network share.
- **regsvr32 http://*servername/path/filename.wsc***—Use this version if the file is placed on a web server.

You can also manually create the necessary Registry entries. Register the object script on a sample computer. Then, look in the Windows Registry under `HKEY_CLASSES_ROOT` under the following keys:

- *`Object.Name`* (that is, the object's `progid`)
- *`Object.Name.versionnumber`*
- `CLSID\{`*`object's classid`*`}`

You can record all the information in the subkeys and values under these keys and write a script to re-create them on other computers, using the objects described in Chapter 4, "File and Registry Access," in the section "Working with the Registry."

Creating a Practical Object

As you saw in Chapter 6, "Messaging and Faxing Objects," sending email from scripts can be complex, and the script has to know a lot about your ISP's or your organization's email system set up. Email systems and ISPs get replaced from time to time. Because I don't want to have to update dozens of separate scripts when this happens, I've decided to create a simplified object I use to send email from all my scripts. That way, when the procedure for sending messages changes, I only have to update one WSC file. I call this object `ScriptMail.Message`.

Wrapping up management of emails into an object has other benefits. For example, I can build in a lot of standard setup information. I could make the default destination for all messages the system administrator; if scripts don't need to change it, they don't have to. Automatically addressing email to my sysadmin is something that the Windows CDO object can't do by itself!

Finally, at a later date, I could extend this object to add paging, pop-up messages, or other forms of delivery that might be useful. With your own objects, you can add whatever features you want. What's more, adding a feature to the definition of an object in the WSC file makes the feature available to every script or program that uses the object.

The first step in designing a useful object is to decide how it should appear to the client. What are the properties and methods? How do they work? As a first step, I suggest you write the object's documentation as if it already existed. That's what we do

here. This guides us later as we implement the object. I've decided I want a simple email object whose properties and methods are listed in Reference List G.2.

REFERENCE LIST G.2 Properties and Methods of the Not-Yet-Written `ScriptMail.Message` Object

Properties:

`To`
The recipient of the message. By default, the message is addressed to `Administrator@mycompany.com`.

`CC`
The recipient for a copy of the message. By default, the recipient is the user who is running the script, with the email address *`username`*`@mycompany.com`. If the administrator is running the script, however, the default CC recipient is left blank because he is already the primary recipient.

`From`
The email address of the person sending the message. By default, this is set to the user who is running the script, with the email address *`username`*`@mycompany.com`.

`ScriptName`
The name of the script file or program from which I'm sending a message (the client script, not the object script). If no subject line is specified, this is used to create a default subject line. This property makes sense for my particular object because it's going to be used mostly to help other scripts send email success and failure notifications.

`Subject`
The subject of the message. If this property is not set, the message is sent with the default subject line "Script *scriptname* on *computername*," where *computername* is the name of the computer on which the script is running and *scriptname* is the name set by the `ScriptName` property, if any.

`Text`
The message to send.

Methods:

`Send`
Sends the message.

Because this object sets sensible defaults for each of its properties, I could send a message from a script with just a few lines, as illustrated here:

```
set msg = CreateObject("ScriptMail.Message")
msg.ScriptName = "diskclean.vbs"
msg.text = "The disk cleanup script was run"
msg.send
```

This sure beats using CDO directly!

Now, it's time to build the object. To start with, I ran the Windows Script Component Wizard and entered the information shown earlier in Figure G.3.

I told the wizard to use VBScript, unchecked Special Implements Support, and checked Error Checking and Debugging. I entered the properties and methods listed earlier, but entered no default values because the default values have to be calculated.

I took the script that the wizard created and made a few changes. Take a look at the following script, and afterward I point out the changes I made to the wizard's original version. Here's the finished product, file `scriptmail.wsc`:

```
'                                ' Example script scriptmail.wsc
<?xml version="1.0"?>
<component>
    <?component error="true" debug="true"?>

    <!-- the registration information was created by the Wizard -->
    <registration
        description="Simple Script Mail Object"
        progid="ScriptMail.Message"
        version="1"
        classid="{3b977054-5cb4-4d7a-98b8-d3bed070f3ce}">
    </registration>

    <public>
        <!-- The properties are bound to variables of the same name
             except "To", which makes VBScript barf, so we use msgTo -->
        <property name="To" internalName="msgTo"/>
        <property name="CC"        />
        <property name="From"      />
        <property name="ScriptName"/>
        <property name="Subject"   />
        <property name="Text"      />
        <method   name="Send"      />
    </public>

    <!-- objects we need: -->
    <object id="netobj" progid="WScript.Network"/>
    <object id="msg"    progid="CDO.Message"/>
    <object id="conf"   progid="CDO.Configuration"/>

    <!-- get the CDO constants like cdoSendUsingPort -->
    <reference object="CDO.Message"/>

    <script language="VBScript">
    <![CDATA[
        ' the object's properties
        dim msgTo, CC, From, Subject, Text,  ScriptName
```

```
' extra variables
dim SMTPServer, maildomain, admin

' set company-specific stuff not changeable by object user
SMTPServer = "mail.mycompany.com"
admin      = "Administrator"
maildomain = "mycompany.com"

' set default property values
msgTo      = admin & "@" & maildomain
From       = netobj.UserName & "@" & maildomain
Subject    = ""
Text       = ""

' address a copy to the script user, if it's not the administrator
if lcase(netobj.username) = lcase(admin) then
    CC = ""
else
    CC = netobj.username & "@" & maildomain
end if

' end of main body & object initialization
'-----------------------------------------------------

' Send method - send the message using CDO or whatever
function Send()
    dim prefix

    ' if no subject was specified, make up a reasonable one
    if Subject = "" then
       if ScriptName = "" then
           Subject = "Script on " & netobj.ComputerName
       else
           Subject = "Script '" & ScriptName & "' on " &_
               netobj.ComputerName
       end if
    end if

    With msg                          ' build the message
        .to       = msgTo
        .cc       = CC
        .from     = From
        .subject  = Subject
        .textBody = Text
    End With

    ' set delivery information. Predefined constants are available

    prefix = "http://schemas.microsoft.com/cdo/configuration/"
    With conf.fields                  ' set delivery options
        .item(prefix & "sendusing")           = cdoSendUsingPort
        .item(prefix & "smtpserver")          = smtpserver
```

```
            .item(prefix & "smtpauthenticate")    = cdoAnonymous
            .update
        End With

        set msg.configuration = conf      ' deliver the message
        on error resume next
        msg.Send

        Send = err.number = 0             ' return True if it went OK
    end function

    ]]>
  </script>
</component>
```

Now let's look at what I changed. The wizard wrote a script that stores its properties in variables, and it used `get` and `put` functions to set them. This is overkill for this object; here, I do all the hard work during initialization and in the `Send` method. The users can set the properties to anything they want, and I don't need to do any processing when the properties are set, so I took the simple approach and decided to just bind the properties to variables.

To do this, I removed the `get` and `put` tags and the `get` and `put` accessor functions the wizard created. They were fine for illustrating a point back in Figure G.2, but I don't need them here. Unfortunately, I can't have a VBScript variable named "`to`" because this is a reserved word. So, I stored the `To` property in a variable named `MsgTo`, which is named with an `internalName` attribute. All the `<property/>` tags use the short form ending with `/>`.

Because the `Send` method has no parameters, I used the short form for the `<method>` tag as well.

Now, on to the real work: In the script's main body, I created default values for each of the properties. The `WScript.Network` object from Chapter 5, "Network and Printer Objects," helps with this job by providing the `UserName` and `ComputerName`. I only needed one network object, so I used an `<object>` tag to create it.

From Chapter 6, I used one each of `CDO.Message` and `CDO.Configuration`, which I made the same way. Now, I could have created these objects at the moment they were needed with `CreateObject()` and released them right afterward, which would keep memory usage to a minimum. But, that's not as much fun as using `<object>`.

When the script's main body finishes, the object is ready to go and is given to the client program. It can change the properties as it pleases, and it finally calls the `Send` method. Back in the object script, `Send` constructs the subject line and sends the message using CDO. It returns the value `True` or `False`, indicating whether it was able to send the message.

To install and use the object, I located it in Explorer, right-clicked, and chose Register. The first few times I tried this I am treated with messages informing me of the syntax errors I had made in the script file. Only when the blatant syntax errors were fixed did the registration procedure work.

After the object is registered, I ran the sample script, and to my utter amazement, it works:

```
                                ' Example script scriptxh03.vbs
set msg = CreateObject("ScriptMail.Message")
msg.ScriptName = "diskclean.vbs"
msg.text = "The disk cleanup script was run"
if msg.send then wscript.echo "Message sent"
```

This is just one example of a useful WSC-based script. If you find others, let me know—visit www.helpwin7.com/scripting and tell me what you've done.