CHAPTER 1

Objects and Object Types

Starting a journey in an unknown programming language and an unfamiliar environment requires that we start with the fundamentals. What are the most basic, the absolutely essential parts of any environment? Variables of course! Variables are those tiny virtual boxes that hold our data and let us mold it into whatever we like. PowerShell is unusual in that all variables hold references to objects. All objects conform to a known, published contract; an object contains a set of operations, usually called methods, which allow us to execute actions. Additionally, PowerShell objects expose properties that allow us to get and set object attributes.

PowerShell is based on the Microsoft .NET framework, but a fully featured shell would go further than supporting .NET objects only. PowerShell does exactly that through its flexible type system. It adapts and extends objects of different origins, so that we, the users, can not only work with them but do so in a uniform way. Learning about the type system and getting to know the most commonly used types is the most important step on the road to PowerShell mastery.

The Extended Type System

Rule number one for PowerShell is that everything is an object. Objects can have different types and origins and can contain various data too. Nevertheless, they must all look the same and expose services in a similar fashion, so that shell scripters do not need to learn different syntaxes for different objects. The first and most important characteristic of an object is its type. A type holds information about the operations that an object supports and is most often a .NET class. To get an object type, we use the `GetType()` method that all .NET objects have:

```
PS C:\> (42).GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Int32                                    System.ValueType

PS C:\> "Hello, world".GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     String                                   System.Object
```

```
PS C:\> (1,2,3).GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Object[]                                 System.Array
```

Accessing Object Properties

An object type contains object members. Members are usually properties and methods. Technically, they can be fields or events, but we will rarely see those in scripts. Properties are generally used to expose object data, and guess what? Data, stored in a property, is usually another object. We can access property values using the following dot notation:

```
PS C:\> $user = @{}
PS C:\> $user.Name = "John"
PS C:\> $user.Name
John
```

The preceding example creates a new dictionary object and stores it in the `$user` variable. The code then assigns the "John" string as the `Name` property value. The last line gets the property value—as you can see, it holds the same string we just put in there.

Properties can be read only or read-write, depending on whether their values can be modified. Of course, the shell will complain if we try to assign a new value to a read-only property:

```
PS C:\> "Hello".Length = 7
"Length" is a ReadOnly property.
At line:1 char:9
+ "Hello".L <<<< ength = 7
```

Strings are immutable objects, so to change a string's length, we would have to create a new string object.

Many objects contain special properties, called indexer properties. Those properties expose collections of objects and allow accessing them via a name or an index. Most often we use that with an array:

```
PS C:\> $fruit = "apples", "oranges"
PS C:\> $fruit[0]
apples
PS C:\> $fruit[1]
oranges
```

Using Object Methods

An object method is a piece of code that represents an operation that the object supports. It usually takes parameters and returns other objects. As an example, here is how we can check if a string contains another string:

```
PS C:\> "Hello".Contains("Hell")
True
```

Methods do not necessarily take parameters. Here is the syntax that converts a string object to uppercase by calling its ToUpper method:

```
PS C:\> "uppercase, please".ToUpper()
UPPERCASE, PLEASE
```

Note that we still need to add the parentheses at the end even if we do not need to pass parameters for the method. Forgetting the parentheses will make PowerShell think we are trying to get hold of the method object itself:

```
PS C:\> "uppercase, please".ToUpper


MemberType          : Method
OverloadDefinitions : {System.String ToUpper(), System.String ToUpper(Cultu
                      reInfo culture)}
TypeNameOfValue     : System.Management.Automation.PSMethod
Value               : System.String ToUpper(), System.String ToUpper(Cultur
                      eInfo culture)
Name                : ToUpper
IsInstance          : True
```

That may be of use to us if we wish to get more information about the method itself, but most often we just want to call it.

Object Adapters

.NET objects are not the only kids in the object-oriented game. A self-respecting shell would allow us to use various application's automation objects, Active Directory data, and Windows Management Instrumentation classes and objects at the least—which all have different origins and are implemented using different technologies. We need a mechanism that will make those look and work just like .NET objects do. PowerShell calls that mechanism object adaptation. Internally, the shell knows how to work with the various object types, and that knowledge is spread among a set of adapter objects. When a foreign object comes along, the shell wraps it in a native .NET object that serves as a view of the original object. The view object is of the PSObject type object and uses the adapter to get properties and methods. Most of the time, a PSObject behaves just like the original object, and the user can hardly tell he or she is using a special object. If we need the original object, we can always get it using the special PSBase property. Now, here is a list of the adapters the shell uses when it works with objects.

ManagementObjectAdapter

This adapter knows about Windows Management Instrumentation (WMI) objects and makes working with them easier by, for example, exposing their Properties collection as object properties. Here is a taste of the clumsy syntax that we would have had to use to get a process's caption if we did not have that adapter:

```
PS C:\> (Get-WmiObject win32_process)[0].PSBase.Properties["Caption"].Value
System Idle Process
```

Note the use of the PSBase property to get to the raw, unadapted object. Using the adapted version, we can just say:

```
PS C:\> (Get-WmiObject win32_process)[0].Caption
System Idle Process
```

DirectoryEntryAdapter

This adapter works with the Windows active directory in a similar way to the ManagementObjectAdapter. Here is how it hides the ugliness of working with a nested Properties collection:

```
$user = [ADSI]"WinNT://./Hristo,user"
$user.PSBase.Properties["Name"]
```

behind a normal property:

```
PS C:\> $user = [ADSI]"WinNT://./Hristo,user"
PS C:\> $user.Name
Hristo
```

DataRowAdapter

This class knows about the ADO.NET DataRow object and exposes row properties accessible via the column names as object properties. Let's take as an example a data table saved as XML in the following format:

```
<Users>
    <User>
        <Name>John</Name>
        <Age>25</Age>
    </User>
    <User>
        <Name>Mike</Name>
        <Age>20</Age>
    </User>
</Users>
```

Here is how we access row data:

```
PS C:\> $ds = New-Object Data.DataSet
PS C:\> $ds.ReadXml("C:\PowerShell\data.xml")
InferSchema
PS C:\> $ds.Tables[0].Rows[0].Name
John
PS C:\> $ds.Tables[0].Rows[0].PSBase["Name"]
John
```

The last line uses the raw row object as an illustration of what row data access would have looked like without the adapter.

DataRowViewAdapter

This guy works in the same way as `DataRowAdapter`—the only difference is that it adapts ADO.NET `DataRowView` objects.

XmlNodeAdapter

This very important adapter class transfers inner objects and XML attributes as object properties to make things easier for the user. The .NET XML classes make it all too complicated by making the user think if a value is stored as an XML attribute or as an element value. The adapter makes all values look like properties to PowerShell code. Here is how we can navigate our XML-based user data by using the property syntax:

```
PS C:\> $xmlDoc = New-Object Xml.XmlDocument
PS C:\> $xmlDoc.Load("C:\PowerShell\data.xml")
PS C:\> $xmlDoc.Users.User[0]

Name                                    Age
----                                    ---
John                                    25
```

Convenient, isn't it? Exposing inner elements as properties allows us to quickly query an XML tree without using heavy-duty tools like XPath at all. XML finally made easy!

ComAdapter

Working with COM objects involves dealing with the compiled type information and accessing the COM type libraries. The PowerShell COM adapter knows how to do that and spares us the details. Look how the Internet Explorer COM object works just like any other object you have seen so far:

```
PS C:\> $ie = New-Object -COM InternetExplorer.Application
PS C:\> $ie.Visible = $true
```

Type Extensions

Object adapters are the primary mechanism of changing how an object looks, but they are internal to PowerShell: they are implemented as .NET objects and are an integral part of the PowerShell code base. Since there is no way for a user to create his or her own adapter or extend an existing one, we have another mechanism of modifying object types: type extensions. The type system allows users to provide additional code and data and add members, both properties and methods, to objects and object types at runtime. PowerShell supports extending objects by adding many different types of members, and we will go through doing that in greater detail in Chapter 12. For now, this is how you can get some of the extended properties of the `System.Diagnostics.Process` type:

```
PS C:\> (Get-Process)[0] | Get-Member -type AliasProperty


   TypeName: System.Diagnostics.Process
```

```
Name    MemberType    Definition
----    ----------    ----------
Handles AliasProperty Handles = Handlecount
Name    AliasProperty Name = ProcessName
NPM     AliasProperty NPM = NonpagedSystemMemorySize
PM      AliasProperty PM = PagedMemorySize
VM      AliasProperty VM = VirtualMemorySize
WS      AliasProperty WS = WorkingSet
```

The code gets the first of all processes on the system and displays its alias properties. You can see that PowerShell extends the `Process` type, so that we can use shorter names for some of the properties it exposes: `Name` instead of `ProcessName`, `WS` instead of `WorkingSet`, and so on.

Alias properties just reference other properties by name and return their values. Property aliasing is usually done to shorten some of the most commonly used properties to save typing and to make objects look consistent. For example, one of my pet peeves with .NET has always been that collections have a `Count` property that returns the number of items inside. Arrays, on the other hand, do not have a `Count` property—they have a `Length` property instead. I have to remember which property to use all the time. PowerShell solves that problem by adding an extended `Count` property to arrays, relieving me of this burden:

```
PS C:\> Get-Member -input (1,2,3) -type AliasProperty


   TypeName: System.Object[]

Name  MemberType    Definition
----  ----------    ----------
Count AliasProperty Count = Length
```

Remember how the `PSBase` special property gave us access to the raw unadapted object? We can get access to the extended object view only by accessing the `PSExtended` special property in much the same manner. This is how to get all the extended members for the `Process` type:

```
PS C:\> (Get-Process)[0].PSExtended | Get-Member


   TypeName: System.Management.Automation.PSMemberSet

Name            MemberType     Definition
----            ----------     ----------
Handles         AliasProperty  Handles = Handlecount
Name            AliasProperty  Name = ProcessName
NPM             AliasProperty  NPM = NonpagedSystemMemorySize
PM              AliasProperty  PM = PagedMemorySize
VM              AliasProperty  VM = VirtualMemorySize
WS              AliasProperty  WS = WorkingSet
__NounName      NoteProperty   System.String __NounName=Process
PSConfiguration PropertySet    PSConfiguration {Name, Id, PriorityClass,...
PSResources     PropertySet    PSResources {Name, Id, Handlecount, Worki...
```

```
Company         ScriptProperty System.Object Company {get=$this.Mainmodu...
CPU             ScriptProperty System.Object CPU {get=$this.TotalProcess...
Description     ScriptProperty System.Object Description {get=$this.Main...
FileVersion     ScriptProperty System.Object FileVersion {get=$this.Main...
Path            ScriptProperty System.Object Path {get=$this.Mainmodule....
Product         ScriptProperty System.Object Product {get=$this.Mainmodu...
ProductVersion  ScriptProperty System.Object ProductVersion {get=$this.M...
```

Again, we will go through all types of extended members in Chapter 12. For now, just think about `Note` properties, as a way to return a constant value from a property, and for `Script` properties, as a way to attach our own script code that will get executed whenever a property is gotten or set. Going back to our array extension and the `Count` property, this is how to get to it:

```
PS C:\> (1,2,3).PSExtended | Get-Member


   TypeName: System.Management.Automation.PSMemberSet

Name  MemberType    Definition
----  ----------    ----------
Count AliasProperty Count = Length
```

You might have noticed that the `PSExtended` property returns a special type of object: `PSMemberSet`. Member sets are logical groups of members that we can use to create different views of objects. To get the member sets that are defined for an object, we can again use the `Get-Member` command and indicate explicitly that we are interested in `PropertySet` and `MemberSet` property types:

```
PS C:\> (Get-Process)[0] | Get-Member -type PropertySet,MemberSet


   TypeName: System.Diagnostics.Process

Name            MemberType  Definition
----            ----------  ----------
PSConfiguration PropertySet PSConfiguration {Name, Id, PriorityClass, Fi...
PSResources     PropertySet PSResources {Name, Id, Handlecount, WorkingS...
```

In the preceding example, you can see two views defined for a `Process` object: one, `PSConfiguration`, that focuses on how a process has been configured and another, `PSResources`, that reports on the system resources that are being used. We can use those property sets when displaying information about an object using one of the `Format-*` commands:

```
PS C:\> Get-Process WinWord | Format-List PSResources


Name                : WINWORD
Id                  : 3368
HandleCount         : 356
WorkingSet          : 44331008
```

```
PagedMemorySize    : 20971520
PrivateMemorySize  : 20971520
VirtualMemorySize  : 331403264
TotalProcessorTime : 00:03:12.5937500



PS C:\> Get-Process WinWord | Format-List PSConfiguration


Name          : WINWORD
Id            : 3368
PriorityClass : Normal
FileVersion   : 11.0.8134
```

As a summary, each object has a set of views that it must support. They are accessible through special properties:

- `PSObject`: The default view is what we get when we work with the object directly.
- `PSBase`: The raw object view that gives us access to a .NET or COM object that has not been adapted by the type system.
- `PSAdapted`: The adapted object view has members added or filtered out after the object has been processed by its adapter.
- `PSExtended`: The extended view of the object contains only members added as type extensions.

Built-in Types

PowerShell not only allows us access to many types of objects and lets us modify them in a uniform way—it also includes several built-in types that are worth knowing, as they are the ones we will be using in our everyday work with the shell. To be honest, PowerShell does not build those types from scratch; it relies heavily on the .NET framework, and all the built-in types are really .NET types. The shell designers have extended those types and added special syntax shortcuts to the language to make it easier to use them.

Strings and String Operations

Character strings are arguably the most commonly used data type in any programming language. Most likely that is the case because strings contain free-form data and are used to transport data across system boundaries. PowerShell's string support is built on top of the .NET `System.String` object type. That means that a string in PowerShell will have all operations that .NET strings have. On top of that, strings have a number of useful extensions that increase the expressiveness and effectiveness of string manipulation.

String Literals

Creating string literals is easy; we just need to surround a sequence of characters in single or double quotation marks:

```
PS C:\> "This is a string"
This is a string
PS C:\> 'This is a string too'
This is a string too
```

Why support both types of quotes? That is a convenience feature. Mixing quotes allows us to embed quotation marks inside strings without having to escape them:

```
PS C:\> 'John said: "OK."'
John said: "OK."
PS C:\> "Let's go"
Let's go
```

Sometimes, strings get complex, and mixing types of quotation marks becomes a burden. In that case, we can just escape a quotation mark and move along. To do so, simply type the symbol twice:

```
PS C:\> "John said: ""OK."""
John said: "OK."
PS C:\> 'Let''s go'
Let's go
```

Another method of escaping quotes is to use the general escape symbol, the backtick (`` ` ``):

```
PS C:\> "John said: `"OK.`""
John said: "OK."
```

One thing to keep in mind is that backtick escaping works in double-quoted strings only. The backtick can be used as an escape character for various symbols that are hard to type:

- `` `0 ``: The null symbol is used as a string end delimiter. It does not have a printable representation.
- `` `a ``: The alert symbol is the character having the `7` ASCII character code. When a string that contains that character gets printed on the screen, it will produce a beep on the computer speaker. The symbol itself will have no visible representation.
- `` `b ``: Backspace will delete the previous symbol:

  ```
  PS C:\> "aa`b bb"
  a bb
  ```

- `` `f ``: The form feed or page break symbol will make a printer start printing on a new page. Again, it has no printable representation.

- `` `n ``: The line feed symbol causes a printer or terminal to start printing on a new line:

  ```
  PS C:\> "aa`nbb"
  aa
  bb
  ```

- `` `r ``: The carriage return symbol causes a printer or terminal to start printing on the beginning of the line. The character will not generate a new line, and subsequent characters will be printed at the beginning of the current line possibly overwriting previous output. In Windows systems, this symbol is used in conjunction with `` `n `` as a line separator in text files. The `` `r`n `` sequence indicates a new line.

- `` `t ``: The horizontal tab symbol moves several symbols forward, according to the current console settings:

  ```
  PS C:\> "aa`tbb"
  aa      bb
  ```

- `` `v ``: Though the vertical tab symbol is rarely, it may prove useful when doing printing or formatting console-based output.

A useful feature of the backtick symbol is that you can use it to escape the new line symbol when typing at the console. Ending your line with a backtick signals to the shell that the command continues on the next line:

```
PS C:\> Get-Item C:\Windows\System32\WindowsPowerShell\v1.0\types.ps1xml `
>> | Get-Content  | Measure-Object
>>


Count    : 3131
Average  :
Sum      :
Maximum  :
Minimum  :
Property :
```

Often, we want to input a large block of text without having to take care of escaping quotes. This is where here strings come handy! Here strings are special markers that indicate that everything between them is a part of the string. To create such a string, we surround it with `@"` and `"@` or `@'` and `'@`:

```
PS C:\> @"
>> line one
>>     line two
>>
>> line three
>> "@
>>
line one
    line two
```

```
line three
PS C:\> @'
>> "Let's
>>     go dowtown!", John said.
>> '@
>>
"Let's
    go dowtown!", John said.
```

Note that all white space, like new lines and tabs, is preserved in here strings. Keep in mind that the string start and end markers *must* be placed alone on separate lines.

String Interpolation

One of the greatest productivity boosters when working with strings is string interpolation, a widely used feature, also known as variable substitution, that is present in many script and shell languages. In essence, these terms refer to the ability to nest variable names in string literals and have the shell expand those variables and embed their values instead. Typically, we use that feature to format messages for display or to craft specially formatted strings before passing them to other programs. Here is how we can send a message to the user:

```
PS C:\> $processCount = (Get-Process).Count
PS C:\> "$processCount processes running in the system."
61 processes running in the system.
```

Sometimes, we do not want to treat a dollar-sign expression as a variable name. To prevent that, we have to escape the dollar sign using a backtick:

```
PS C:\> "`$processCount is the property that we need."
$processCount is the property that we need.
```

Another option is to use a single-quoted string. Interpolation only works on double-quoted strings:

```
PS C:\> '$processCount processes running in the system.'
$processCount processes running in the system.
```

String interpolation works for here strings too:

```
PS C:\> @"
>> Processes
>> ---------
>> $processCount
>> "@
>>
Processes
---------
61
```

Analogous to ordinary single-quoted strings, there is no interpolation when using single-quoted here strings.

We can use some really complex variable expressions inside strings, but we have to be careful and obey the formatting rules. Variable parsing stops at the end of the first word, and to use an expression, we have to put it inside a `$()` block:

```
PS C:\> $processes = (Get-Process)
PS C:\> "$($processes.Count) processes running in the system."
60 processes running in the system.
```

Note that variables inside the `$()` block still need to be prefixed with dollar signs. Now, let's go a step further and get rid of the `$processes` variable entirely:

```
PS C:\> "$((Get-Process).Count) processes running in the system."
60 processes running in the system.
```

We can embed any legal PowerShell expression:

```
PS C:\> "Total due: $(12 * 5000)"
Total due: 60000
```

We can even cause side effects when generating strings and modify variable values from embedded expressions:

```
PS C:\> $times = 0
PS C:\> "Operation performed $($times++; $times) times"
Operation performed 1 times
PS C:\> $times
1
```

■ **Caution** While useful at times, relying on side effects triggered by string interpolations is usually considered bad programming practice and is best avoided. Most people expect that variable expressions do not modify the system state, and doing the opposite will lead to many hard-to-track bugs.

Productivity Boosting String Operators

In PowerShell, strings have been extended with special syntax so that commonly used operations are easier to perform. PowerShell uses the familiar dash notation that looks like providing a command parameter. The most important operations follow.

Formatting

Instead of calling the static `String.Format` method like this:

```
PS C:\> [string]::Format("{0} running processes.", $processCount)
61 running processes.
```

PowerShell can format a string once it sees the `-f` parameter:

```
PS C:\> "{0} running processes." -f $processCount
61 running processes.
```

Wildcard Matching

This is similar to matching using a regular expression without all the intrinsic power regular expressions have. The advantage to this method is that it is really intuitive and simple to use. For example, we can test if a string starts with the word "Test" as follows:

```
PS C:\> "Test string" -like "Test*"
True
PS C:\> "Sample string" -like "Test*"
False
```

The asterisk symbol means "zero or more symbols of any type." We can use a question mark as a match for exactly one symbol of any type:

```
PS C:\> "notepad.exe" -like "notepad.???"
True
PS C:\> "notepad.exe" -like "notepad.?"
False
PS C:\> "notepad.exe" -like "?otepad.exe"
True
```

We can also use a character range. Here is how to check if a string starts with any letter and ends with "s" or "n":

```
PS C:\> "notepads" -like "[a-z]*[sn]"
True
PS C:\> "notepadz" -like "[a-z]*[sn]"
False
PS C:\> "_notepads" -like "[a-z]*[sn]"
False
```

Regular Expression Matching

Regular expression matching is similar to wildcard matching with the only difference being that we use the `-match` operator and provide a regular expression pattern. Regular expressions are a vast subject on which entire books have been written, so I will not be covering them here. As an example, this is how you match something that looks like a domain name:

```
PS C:\> "www.yahoo.com" -match "(www\.)?\w+\.(com|org|net)"
True
PS C:\> "yahoo.com" -match "(www\.)?\w+\.(com|org|net)"
True
PS C:\> "yahoo.org" -match "(www\.)?\w+\.(com|org|net)"
True
```

The expression matches strings that optionally start with "www. ", followed by one or more alphanumeric characters, followed by a dot and ending in "com", "org", or "net". Note that the dot (`.`) symbol has a special meaning in regular expressions. It signifies any character, so we escape it with a backslash.

Tip For more information you can visit the excellent online resource that is the `http://www.regular-expressions.info` site and go through its tutorials and references on the subject. The site covers regular expression usage in different programming languages and environments. While the syntax and support are more or less the same on most platforms, there are some subtle differences, so make sure you go through the article about .NET regular expressions at `http://www.regular-expressions.info/dotnet.html`. PowerShell uses the .NET regular expressions under the hood.

Replacing Strings or Substrings

PowerShell provides the `-replace` operator to replace string occurrences. This operator behaves much differently than the .NET `String.Replace()` method! `String.Replace()` takes a string as a parameter and replaces all occurrences in the target string:

```
PS C:\> "one.test, two!test".Replace(".test", "-->DONE")
one-->DONE, two!test
```

`-replace` takes a regular expression and internally calls `Regex.Replace()`. Be careful when providing the replace pattern, and escape regular expression special characters if needed. The preceding example might not work as expected if we use `-replace`:

```
PS C:\> "one.test, two!test" -replace ".test", "-->DONE"
one-->DONE, two-->DONE
```

The dot instructs the regular expression engine to look for any character, so our exclamation mark gets replaced too. To correct that, we have to escape the dot:

```
PS C:\> "one.test, two!test" -replace "\.test", "-->DONE"
one-->DONE, two!test
```

Having interpolation and regular expressions under your belt will significantly boost your scripting productivity, so practicing those two until they feel natural might be an excellent idea.

Numeric Types

PowerShell supports all native .NET numeric types. Probably the only ones that we will eventually need to construct from literals are those in the following sections.

System.Int32

`System.Int32`, also known as `int`, is a 32-bit integer value that is the default for most operations. Creating one is a matter of typing a sequence of digits:

```
PS C:\> (3).GetType().FullName
System.Int32
```

Floating Point Numbers

Single- and double-precision floating point numbers are also known as `single` or `float` and `double`. The most commonly used type is `double`, and it can be created using different numeric sequences:

```
PS C:\> (3.0).GetType().FullName
System.Double
PS C:\> (-3E2).GetType().FullName
System.Double
PS C:\> (4.5E-2).GetType().FullName
System.Double
```

Note the scientific notation that allows us to specify a mantissa and an exponent.

Decimal Numbers

A `System.Decimal`, also referred to as `decimal`, number represents a very large decimal number that is appropriate for financial operations. These numbers perform well when we need a large number of integral and fractional digits without round-off errors. To create a decimal number, we need to add a `d` suffix to our number literal:

```
PS C:\> 2d
2
PS C:\> 2.5d
2.5
PS C:\> 2.5d * 10000000000000000000000000000
25000000000000000000000000000
```

Hexadecimal Numbers

Just as in all languages sharing a common heritage with C, in PowerShell, we specify hexadecimal numbers by prefixing them with `0x`:

```
PS C:\> 0x10
16
PS C:\> 0x1A
26
PS C:\> 0xABCD
43981
```

Most computer-related numbers, however, are not multiples of ten. Computers use only binary, and many important numbers are multiples of two. Fortunately, the PowerShell designers have added a clever trick: binary multiplier suffixes. We can append KB, MB, or GB after any number and get a multiple of 1,024, 1,048,576, and 1,073,741,824, respectively. How nice! Here is how we can use that to get information about a running process's memory usage:

```
PS C:\> (Get-Process winword).VM -ge 20MB
True
PS C:\> (Get-Process winword).VM / 1MB
310.69921875
```

Yes, my `winword` process is using more than 20MB of virtual memory. In fact, it is eating up about 310 of them. You can use a similar technique for files and other objects in your computer:

```
PS C:\> (Get-Item C:\pagefile.sys -Force).Length / 1GB
1.5
```

`Pagefile.sys` is a system file, so I had to use the `-Force` parameter to get to it.

Numbers as Strings

Finally in this section we will look at how numbers can be formatted and converted to strings. We do that by using the `-f` string operator. Some of the particularly useful formatting options follow:

- *Currency*: Use a `{0:c}` format string. The currency symbol will be obtained from the current culture.

  ```
  PS C:\> "{0:c}" -f 2.5
  $2.50
  ```

- *Percentage*: The format string is `{0:p}`:

  ```
  PS C:\> "{0:p}" -f 0.2
  20.00 %
  ```

- *Fixed number of digits*: Use the pound sign (`#`) as a digit placeholder. For example, this line will format a number and round it to the third digit after the decimal point:

  ```
  PS C:\> "{0:#.###}" -f 3.4567
  3.457
  ```

- *Scientific notation*: Some people find scientific notation odd; others, useful. Passing a `{0:e}` format string will get us a properly formatted normalized mantissa and an exponent:

  ```
  PS C:\> "{0:e}" -f 333.4567
  3.334567e+002
  ```

There are a number of other formats available in .NET that we will not cover here. To get all of them, peruse the `System.Globalization.NumberFormatInfo` class documentation on MSDN.

Arrays and Collections

Arrays and other ordered collections play a very important part in all programming tasks. I will use the words "array" and "ordered collection" interchangeably in this section, as PowerShell adapts all list collections so that they work in the same way, and we cannot tell one type of collection from another (it is possible to do so, but there is little value in that).

Creating an array is as simple as listing all its members and separating them with commas. Here is a small array containing numbers:

```
PS C:\> 1, 2, 3, 4
1
2
```

```
3
4

PS C:\> (1, 2, 3, 4).GetType().FullName
System.Object[]
```

Note that the array is of the `Object` type, which means we can hold all types of objects inside:

```
PS C:\> 1, 2.5, "apples", (Get-Process winword)
1
2.5
apples

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    352      15    19804      46920   311   112.83   3252 WINWORD


PS C:\> (1, 2.5, "apples", (Get-Process winword)).Count
4
```

The preceding code creates an array of objects that contains an integer, a double precision floating point number, a string, and a `Process` object—that's four types of objects in total: an `int`, a `double`, a `string`, and a `System.Diagnostics.Process` instance.

Array creation has a very free-form syntax. The most formal syntax is to wrap the members in a `@()` block:

```
PS C:\> @("one", "two")
one
two
```

This is the only way to create an empty array:

```
PS C:\> (@()).Count
0

PS C:\> (@()).GetType().FullName
System.Object[]
```

As a convenience, arrays can be created using the numeric range notation. It requires that we provide the start and end numbers and fills in the values between them for us:

```
PS C:\> 5..1
5
4
3
2
1

PS C:\> 1..5
```

```
1
2
3
4
5


PS C:\> (0..255).Count
256
```

That syntax is extremely powerful for enumerating ports, addresses, and all sorts of sequential data.

The array creation syntax recognizes nested arrays and flattens them into one. This allows us to add arrays and even ranges in place:

```
PS C:\> 1, (5, 6), 2
1
5
6
2

PS C:\> 1, (10..7), 2
1
10
9
8
7
2
```

We can access the array items by using the bracket notation:

```
PS C:\> $a = 2,3,4
PS C:\> $a[0]
2
PS C:\> $a[1]
3
PS C:\> $a[2] = 5
PS C:\> $a
2
3
5
```

Note that we modify an item by directly assigning a new value to it. PowerShell arrays, just like the .NET ones, are zero-based, which means that the first item is always at index zero, and the last one at the item count minus one.

Interestingly enough, we can pass more than one index inside the brackets. The return is an array that contains the items at those indexes. This is called an array slice. Here is how to get the first and the last item from an array:

```
PS C:\> $a = 2,3,4
PS C:\> $a[0,2]
2
4
```

The preceding example is a bit problematic, because it requires us to know that the array has three items, so that we request the one at index `2`. Many programming languages, PowerShell included, allow us to address items starting from the end by using negative indexes. Here is how to change the previous example, so that our code does not have to use a hard-coded number for the array length in order to get the last value:

```
PS C:\> $a = 2,3,4
PS C:\> $a[0,-1]
2
4
```

We can create slices by passing ranges as our array indexes too! Here is how to get five items, starting from the third:

```
PS C:\> $a = (1, 2, 3, 4, 5, 6, 7, 8)
PS C:\> $a[2..6]
3
4
5
6
7
```

Note that the ranges are inclusive, so we have to pass `2..6` , instead of `2..7`, to get the five items.

Assigning arrays to items will flatten the arrays. This makes for a very convenient way to insert several items at a specific location:

```
PS C:\> $a = 1, 2, 3
PS C:\> $a[1] = 10, 11, 12
PS C:\> $a
1
10
11
12
3
```

Note that the original value of the second item got destroyed. To preserve it, we need to put forth some extra effort. Prepending the inserted array with the original value does the trick:

```
PS C:\> $a = 1, 2, 3
PS C:\> $a[1] = $a[1], 10, 11, 12
PS C:\> $a
1
2
10
11
```

```
12
3
```

Similar to strings, arrays have the plus operator defined for concatenation. “Adding” an object or another array to an array will result in a new array that contains all values:

```
PS C:\> $a = 1,2
PS C:\> $a = $a + 3
PS C:\> $a
1
2
3
```

The preceding operation adds an item to a variable and assigns the result to the same variable, effectively modifying the variable in place; it is so common that most languages define a shortcut operator (+=) that does exactly that. PowerShell supports that too:

```
PS C:\> $a += 4
PS C:\> $a
1
2
3
4

PS C:\> $a += 5,6
PS C:\> $a
1
2
3
4
5
6
```

Searching for items inside collections is a routine task in many languages. How do we search an array for a specific value? Here is an attempt:

```
foreach ($item in (2,3,4)){ if ($item -eq 3){ echo "Found" }}
```

That feels too clumsy, and we have to repeat the same loop everywhere we need to search for an array item! Checking if an array contains an item is so common that the PowerShell designers have provided two specific operators for that: `-contains` and `-notcontains`:

```
PS C:\> (2, 3, 4) -contains 3
True
PS C:\> (2, 3, 4) -notcontains 3
False
PS C:\> (2, 3, 4) -notcontains 5
True
```

Of course, those operators work for different variable types as well:

```
PS C:\> (2, "some value", 4) -contains "some value"
True
```

Beware of the built-in type conversions though! You may get something quite different than what you expect:

```
PS C:\> (2, 3, 4) -contains "3"
True
PS C:\> (2, 3, 4) -contains "3.0"
True
```

See how the “3.0” string got converted to the integer `3` and signaled `True`? If you need to look for strings and avoid type coercion, you may have to use a loop or the `Where-Object` command:

```
PS C:\> (2, 3, 4) | Where-Object { $_ -is [string] -and $_ -eq "3.0"}
PS C:\> (2, "3.0", 4) | Where-Object { $_ -is [string] -and $_ -eq "3.0"}
3.0
```

I will describe loops in the next chapter and talk about the `Where-Object` command in greater detail in Chapter 3.

Dictionaries and Hash Tables

Dictionaries, also called associative arrays, are objects that map a set of keys to a set of values. They contain a number of key-value pairs and allow for easy lookup of values given their keys. Dictionaries are a convenient way to store related data that usually comprises a logical record about a real-world entity. The syntax for creating a dictionary in PowerShell looks like the formal array creation syntax; the only difference is that we use braces. Here is a sample personal record dictionary that has two items with keys “Name” and “Address” respectively:

```
PS C:\> $d = @{"Name"="John"; "Address"="12 Easy St."}
PS C:\> $d

Name                           Value
----                           -----
Name                           John
Address                        12 Easy St.
```

PowerShell uses `System.Collection.Hashtable` objects internally whenever it creates a dictionary object. `Hashtable` implements the .NET `IDictionary` interface, and PowerShell’s extended type system allows us to work in the same way with all `IDictionary` objects.

```
PS C:\> $d.GetType().FullName
System.Collections.Hashtable
```

Dictionaries do not impose any restrictions on the type of objects that we use as keys or values. Here is how to add different objects to a `Hashtable`:

```
PS C:\> $d = @{"Name"="John"; "Age"=30; `
>>     "File"=(Get-Item C:\PowerShell\Test.txt)}
>>
PS C:\> $d

Name                           Value
----                           -----
Name                           John
Age                            30
File                           C:\PowerShell\Test.txt
```

$d contains a string, an int, and a FileInfo object. Our dictionary is starting to look like a person's record taken from a real database already!

You are not restricted to using only strings as dictionary keys. We can certainly use numbers, but in general, any object will do. Assuming we have Windows Calculator and Microsoft Word running on the system, here is how to create a set of Process objects and map them to string values:

```
PS C:\> $set = @{(Get-Process winword)="MS Word"; (Get-Process calc)="calc"}

PS C:\> $set

Name                           Value
----                           -----
System.Diagnostics.Process ... calc
System.Diagnostics.Process ... MS Word
```

Empty dictionaries are created with the @{} expression:

```
PS C:\> $empty = @{}
PS C:\> $empty.GetType().FullName
System.Collections.Hashtable
```

There are several ways to access values in a dictionary. The most important one is the array-like notation; we provide the key name in brackets:

```
PS C:\> $d = @{"Name"="John"; "Age"=30; "Address"= "12 Easy St."}
PS C:\> $d["Name"]
John
```

Dictionaries can behave like objects, and we can access their values as properties using the dot notation:

```
PS C:\> $d = @{"Name"="John"; "Age"=30; "Address"= "12 Easy St."}
PS C:\> $d.Name
John
```

The object notation is more limiting than the array one in a sense that it is hard to provide a key that contains spaces, new lines, or other fancy symbols. The key also has to be convertible from a string. To provide free-form key values, you have to use the array notation.

The object notation supports some fancy ways of providing key values too. This is how to use a string as the property name:

```
PS C:\> $d."Name"
John
```

We can use a variable as the key:

```
PS C:\> $property = "Name"
PS C:\> $d.$property
John
```

and even an expression:

```
PS C:\> $d.$($property)
John
PS C:\> $d.$("Na" + "me")
John
```

Similar to array slices, dictionaries support getting many values at once by providing an array of keys in the brackets. The return object is an array that contains the values. This is how to get only a person's name and address from our previous array:

```
PS C:\> $d = @{"Name"="John"; "Age"=30; "Address"= "12 Easy St."}
PS C:\> $d["Name", "Address"]
John
12 Easy St.
```

Adding items to a dictionary is a just matter of assigning a new key-value pair with either the object or array notation:

```
PS C:\> $d.Department = "Accounting"
PS C:\> $d["SSN"] = 123456789
PS C:\> $d

Name                           Value
----                           -----
Department                     Accounting
Name                           John
Age                            30
SSN                            123456789
Address                        12 Easy St.
```

Removing items is a bit trickier. We have to use the `Remove()` method and provide the key:

```
PS C:\> $d.Remove("Age")
PS C:\> $d.Remove("SSN")
PS C:\> $d

Name                           Value
----                           -----
Department                     Accounting
Name                           John
Address                        12 Easy St.
```

Other important dictionary methods that are worth knowing are the ones that verify if an item is present. `Contains()` and `ContainsKey()` are synonyms that check if an item with the given key exists in the dictionary:

```
PS C:\> $d.Contains("John")
False
PS C:\> $d.Contains("Name")
True
PS C:\> $d.ContainsKey("Name")
True
```

`ContainsValue()` is our friend when we want to look for a value:

```
PS C:\> $d.ContainsValue("John")
True
PS C:\> $d.ContainsValue("Name")
False
```

In addition, dictionaries expose all their keys and values as arrays through the `Keys` and `Values` properties:

```
PS C:\> $d.Keys
Department
Name
Address
PS C:\> $d.Values
Accounting
John
12 Easy St.
```

We can even substitute our calls to `ContainsKey()` and `ContainsValue()` with array searches:

```
PS C:\> $d.Keys -contains "Name"
True
PS C:\> $d.Values -contains "John"
True
```

Remember that `-contains` will silently convert types under the hood! `ContainsKey()` and `ContainsValue()` do not.

Of course, just like any .NET collection, dictionaries have a `Count` property that tells us the number of key-value pairs stored inside:

```
PS C:\> $d = @{"Name"="John"; "Age"=30; "Address"= "12 Easy St."}
PS C:\> $d.Count
3
```

Type Literals

The PowerShell language allows us to access types by using a special syntax called type literals. A type literal is just a type name enclosed in brackets. At their very simplest, type literals return the underlying .NET `System.Type` object instance:

```
PS C:\> [System.Int32]

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Int32                                    System.ValueType


PS C:\> [System.String]

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     String                                   System.Object


PS C:\> [System.Diagnostics.Process]

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     False    Process                                  System.Compon...
```

We do not really need to repeat that `System` namespace prefix all the time. Most of the types that we work with are below the `System` namespace, and PowerShell already knows that. It allows us to omit the prefix altogether:

```
PS C:\> [Diagnostics.Process]

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     False    Process                                  System.Compon...
```

We also have a number of type aliases defined for our convenience. The most popular follow:

- `int`, `short`, `long` for referencing the `System.Int32`, `System.Int16` and `System.Int64` types respectively.
- `byte` and `sbyte` for the unsigned `System.Byte` and the signed `System.SByte` types.
- `bool` for `System.Boolean`.
- `void` for no type at all.
- `string` for the `System.String` type.
- `float` and `single` for `System.Single` and `double` for `System.Double`.

- `decimal` aliases `System.Decimal`.
- `regex` points to `System.Text.RegularExpressions.Regex`.
- `adsi` refers to `System.DirectoryServices.DirectoryEntry`.
- `wmi`, `wmiclass`, and `wmisearcher` refer to `System.Management.ManagementObject`, `System.Management.ManagementClass`, and `System.Management.ManagementObjectSearcher`, respectively.

You do not need to remember what each alias stands for; you can always check:

```
PS C:\> ([regex]).FullName
System.Text.RegularExpressions.Regex

PS C:\> ([adsi]).FullName
System.DirectoryServices.DirectoryEntry

PS C:\> ([wmisearcher]).FullName
System.Management.ManagementObjectSearcher

PS C:\> ([wmi]).FullName
System.Management.ManagementObject
```

Type Conversion

Most often, type literals are used to instruct the shell to convert an object from one type to another. The syntax involved requires that we place a type literal before the original object, and the operation will return a new object if nothing goes wrong. Here is how to convert a string to an integer:

```
PS C:\> $str = "10"
PS C:\> $str.GetType().FullName
System.String
PS C:\> $num = [int] $str
PS C:\> $num
10
PS C:\> $num.GetType().FullName
System.Int32
```

Of course, things can and will go wrong. There are a lot of conversions that are just impossible. In that case, an exception will be thrown:

```
PS C:\> [int] "not an integer string"
Cannot convert value "not an integer string" to type "System.Int32". Error:
 "Input string was not in a correct format."
At line:1 char:6
+ [int]  <<<< "not an integer string"
```

How PowerShell Converts Between Types

PowerShell really does its best when trying to convert between types. It relies heavily on the .NET framework type conversion mechanisms and exposes a mechanism that programmers can use to provide conversion facilities. Here are the strategies the shell tries when tasked with a conversion operation:

1. It checks if the new type is directly assignable from the old. For that to be true in .NET, the types have to be the same or one of them must inherit directly or indirectly from the other.
2. Well-known ways to convert from one object to another have been built into the language itself; PowerShell next attempts to convert using these. See the "Built-in Type Conversion Rules" section for details.
3. Looks for a `TypeConverter` or a `PSTypeConverter` associated class. `TypeConverter` classes are a part of the .NET native mechanism of converting types, and `PSTypeConverter` is a PowerShell-specific converter object. Both `TypeConverter` and `PSTypeConverter` classes can be associated with types at compile type, by marking our types with the correct .NET attributes. Many existing .NET classes already are associated with type converters, and PowerShell will seamlessly convert them to and from other types. Additionally, we can associate a type with a type converter in the shell's extended type system configuration files.
4. PowerShell next tries to convert the original object to a string and to find and call a `Parse()` method on the new object type that will create a new object from a string representation.
5. It then looks for a constructor on the new type that will accept an object of the original type. If it finds one, it uses it to construct the new object.
6. After that, PowerShell looks for both explicit and implicit type cast operators on both object types. Appropriate cast operators are compiled as static methods with the names `op_Explicit` and `op_Implicit` for explicit and implicit casts, respectively. If the shell finds an appropriate method, it calls the method and uses the returned object.
7. Finally, PowerShell checks if the original type implements the `IConvertible` .NET interface. If it does, then the type can be converted to the basic types like `int`, `double`, `decimal`, and so on.

Built-in Type Conversion Rules

PowerShell has several mechanisms for converting between objects that do not require the conversion strategies listed in the previous section. There are well-known ways to convert one object to another, and they are built in the language. Here they are:

- Every object can be converted to a `PSObject`. In fact, the shell does so automatically, and most of the time, we are not holding real references to objects but are working with `PSObject` instances that point to the actual object.
- Conversions to `void` return `$null`:

```
PS C:\> ([void] 5) -eq $null
True
```

- We can convert all types of collections and objects to arrays. A collection here is defined as any .NET type that implements the `ICollection` interface. The .NET framework has many collection classes built in, and collections implemented by any programmer implement that interface too. That means we can convert to an array from any collection. Here is how:

```
PS C:\> ([object[]] 5).GetType().FullName
System.Object[]
PS C:\> $strings = New-Object Collections.Specialized.StringCollection
PS C:\> ([object[]]$strings).GetType().FullName
System.Object[]
```

- We can convert all objects to Boolean values by treating all nonnull and nonempty objects as `true`. We will discuss that in detail in Chapter 2.
- Anything can be converted to a string.
- `IDictionary` objects can be converted to `Hashtable`. The `IDictionary` interface in the .NET framework is implemented by objects that maintain a mapping between a set of key objects to a set of values. The `Hashtable` class is the most commonly used dictionary object, and here is how we can convert an `OrderedDictionary` object to a `Hashtable`:

```
PS C:\> $dict = New-Object Collections.Specialized.OrderedDictionary
PS C:\> $dict["item1"] = 3
PS C:\> $hash = [Hashtable] $dict
PS C:\> $hash

Name                           Value
----                           -----
item1                          3


PS C:\> $hash.GetType().FullName
System.Collections.Hashtable
PS C:\> $dict.GetType().FullName
System.Collections.Specialized.OrderedDictionary
```

- "Converting" a variable to a `PSReference` creates a reference variable pointing to the original. Modifying a reference value modifies the original:

```
PS C:\> $a = 3
PS C:\> $b = [ref] $a
PS C:\> $b.Value = 4
PS C:\> $b.Value
4
PS C:\> $a
4
```

We will discuss references in greater detail in Chapter 5, because they are particularly helpful when passing parameters to functions.

- A properly formatted string can be converted to an XML document:

```
PS C:\> $doc = [xml] "<root><item>1</item></root>"
PS C:\> $doc.root.item
1
```

- Script blocks can be converted to .NET delegates. Delegates are a special type of object that point to a method or a block of code. They are typically involved in associating event handler code with the event and are used by objects to fire an event. We usually need to convert a script block to a delegate when working with a .NET type that can fire an event that we want to consume.

A number of objects can be created or obtained by converting from a string that contains a name, path, or a query. Here they are:

- Strings can be converted to character arrays:

```
PS C:\> [char[]]"chars"
c
h
a
r
s
```

- Strings can be turned to regular expressions. The string is treated as the match pattern:

```
PS C:\> ([regex] "a.*?b").GetType().FullName
System.Text.RegularExpressions.Regex
```

- Strings can also create WMI objects (I will demonstrate the power of those conversions in Chapter 20, where we will work with WMI):

```
PS C:\> [wmisearcher]'Select * from Win32_Process'


Scope     : System.Management.ManagementScope
Query     : System.Management.ObjectQuery
Options   : System.Management.EnumerationOptions
Site      :
Container :

PS C:\> [WMI]'\\.\root\cimv2:Win32_Process.Handle=0'
ProcessName                 : System Idle Process
…
PS C:\> ([wmiclass] "Win32_Share").GetType().FullName
System.Management.ManagementClass
```

- Strings can be converted to Active Directory entries too. The conversion process uses the string as the `DirectoryEntry` path. Note how we use `PSBase` to get to the real object type instead of the adapted view:

```
PS C:\> $user = [ADSI]"WinNT://./Hristo,user"
PS C:\> $user.PSBase.GetType().FullName
System.DirectoryServices.DirectoryEntry
```

- Strings can point to .NET types. The input string is treated as the type name. The conversion works with the full type name. Again, we can omit the `System` namespace prefix:

```
PS C:\> [type] "System.Diagnostics.Process"

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     False    Process                                  System.Compon...


PS C:\> [type] "Diagnostics.Process"

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     False    Process                                  System.Compon...
```

Accessing Static Members

Type literals can be used for another important purpose: they are the only way to access a static property or call a static method. The syntax requires us to separate the type literal and the member name with a double colon (`::`). This is how we can get a static property value:

```
PS C:\> [datetime]::NowTuesday, September 04, 2007 9:25:38 AM


PS C:\> [datetime]::Today
Tuesday, September 04, 2007 12:00:00 AM
```

Calling static methods looks similar. In the following example, we create a `double` object by manually calling the `Parse` method:

```
PS C:\> [double]::Parse("2.5")
2.5
```

Summary

PowerShell’s extended type system is really an impressive piece of work. It is very powerful and complex, and the really remarkable thing about it is that it has not turned out too restrictive or too newbie unfriendly. Users find their way around quickly because of the uniform access they get to all types of objects through type adaptation and extension. The most commonly used objects get pragmatic extensions that boost a scripter’s productivity and lower the learning curve. The type conversion system silently transforms objects under the hood, adapting them as necessary, thus saving valuable keystrokes and increasing script readability. All these assets make the PowerShell language a formidable blend of a script language’s simplicity with the raw power of .NET and Windows