SECOND EDITION

Windows PowerShell IN ACTION

Bruce Payette

MEAP Unedited Draft

MEAP Edition
Manning Early Access Program

Copyright © 2010 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

Part 1: Learning PowerShell

1 Welcome to PowerShell
2 The basics in detail
3 Working with types
4 Operators and expressions
5 Advanced operators and variables
6 Flow control in scripts
7 PowerShell Functions
8 Advanced functions and scripts
9 Using and Authoring Modules
10 Module Manifests and Metadata
11 ScriptBlocks and Dynamic Code
12 Remoting and Background Jobs
13 Remoting: Advanced Configuration and Application
14 Errors and exceptions
15 Script debugging

Part 2: Using PowerShell

16 Processing text, files, and XML
17 Getting fancy — .NET and WinForms
18 Windows objects: Come, WMI and WSMan
19 Security, security, security

Appendix A Comparing PowerShell to other languages

Appendix B Admin examples

Appendix C The PowerShell grammar

Appendix D World-Ready Scripting

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

1
Welcome to PowerShell

Space is big. Really big! You just won't believe how vastly hugely mind-bogglingly big it is. I mean you may think it's a long way down the road to the chemist, but that's just peanuts compared to space!

Don't Panic.

—Douglas Adams, The Hitchhiker's Guide to the Galaxy

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

Welcome to Windows PowerShell, the new command and scripting language from Microsoft. We begin this chapter with two quotes from The Hitchhiker's Guide to the Galaxy. What do they have to do with a new scripting language? In essence, where a program solves a particular problem or problems, a programming language can solve any problem, at least in theory. That's the "big, really big" part. The "Don't Panic" bit is, well—don't panic. While PowerShell is new and different, it has been designed to leverage what you already know, making it easier to learn. It's also designed to allow you to learn it a bit at a time. Starting at the beginning, here's the traditional "Hello world" program in PowerShell.

```
"Hello world."
```

As you can see, no panic needed. But "Hello world" by itself is not really very interesting. Here's something a bit more complicated:

```
dir $env:windir\*.log | select-string -List error |
format-table path,linenumber -auto
```

Although this is more complex, you can probably still figure out what it does. It searches all the log files in the Windows directory, looking for the string "error", then prints the full name of the matching file and the matching line number. "Useful, but not very special," you might think, because you can easily do this using cmd.exe on Windows or bash on UNIX. So what about the "big, really big" thing? Well, how about this example?

```
([xml](new-object net.webclient).DownloadString(
"http://blogs.msdn.com/powershell/rss.aspx"
)).rss.channel.item | format-table title,link
```

Now we're getting somewhere. This script downloads the RSS feed from the PowerShell team weblog, and then displays the title and a link for each blog entry.

NOTE

RSS stands for Really Simple Syndication. This is a mechanism that allows programs to download web logs automatically so they can be read more conveniently than in the browser.

By the way, you weren't really expected to figure this example out yet. If you did, you can move to the head of the class!

Finally, one last example:

```
[void][reflection.assembly]::LoadWithPartialName(
    "System.Windows.Forms")
$form = new-object Windows.Forms.Form
$form.Text = "My First Form"
$button = new-object Windows.Forms.Button
$button.text="Push Me!"
$button.Dock="fill"
$button.add_click({$form.close()})
$form.controls.add($button)
$form.Add_Shown({$form.Activate()})
$form.ShowDialog()
```

This script uses the Windows Forms library (WinForms) to build a graphical user interface (GUI) that has a single button displaying the text "Push Me". The window this script creates is shown in figure 1.1.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

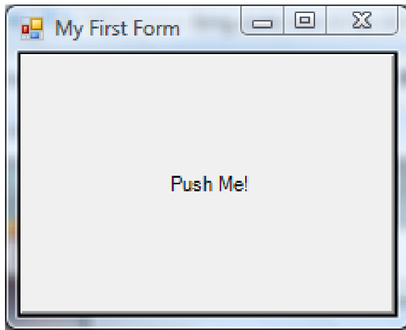

Figure 1,1 When you run the code from the example, this window will be displayed. If you don't see it, it may be hidden behind another window.

When you click the button, it closes the form and exits the script. With this you go from `"Hello world"` to a GUI application in less than two pages.

Now let's come back down to earth for minute. The intent of chapter 1 is to set the stage for understanding PowerShell—what it is, what it isn't and, almost as important—why we made the decisions we made in designing the PowerShell language. Chapter 1 covers the goals of the project along with some of the major issues we faced in trying to achieve those goals. By the end of the chapter you should have a solid base from which to start learning and using PowerShell to solve real-world problems. Of course all theory and no practice is boring, so the chapter concludes with a number of small examples to give you a feel for PowerShell. But first, a philosophical digression: while under development, the codename for this project was Monad. The name Monad comes from The Monadology by Gottfried Wilhelm Leibniz, one of the inventors of calculus. Here is how Leibniz defined the Monad:

> "The Monad, of which we shall here speak, is nothing but a simple substance, which enters into compounds. By 'simple' is meant 'without parts.'"
>
> From The Monadology by Gottfried Wilhelm Leibniz (translated by Robert Latta)

In *The Monadology*, Leibniz described a world of irreducible components from which all things could be composed. This captures the spirit of the project: to create a toolkit of simple pieces that you compose to create complex solutions.

1.1 What is PowerShell?

What is PowerShell and why was it created? As we said, PowerShell is the new command-line/scripting environment from Microsoft. The overall goal for this project was to provide the best shell scripting environment possible for Microsoft Windows. This statement has two parts, and they are equally important, as the goal was not just to produce a good generic shell

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

environment, but rather to produce one designed specifically for the Windows environment. While drawing heavily from existing command-line shell and scripting languages, the PowerShell language and runtime were designed from scratch to be an optimal environment for the modern Windows operating system.

Historically, the Windows command line has been weak. This is mainly the result of the early focus in Microsoft on computing for the average user, who is neither particularly technical nor particularly interested in computers. Most of the development effort for Windows was put into improving the graphical environment for the non-technical user, rather than creating an environment for the computer professional. Although this was certainly an enormously successful commercial strategy for Microsoft, it has left some segments of the community under-served.

In the next couple of sections, we'll go over some of the other environmental forces that led to the creation of PowerShell. By environmental forces, we mean the various business pressures and practical requirements that needed to be satisfied. But first we'll refine our definitions of shell and scripting.

1.1.1 Shells, command-lines, and scripting languages

In the previous section, we called PowerShell a command-line shell. You may be asking, what is a shell? And how is that different from a command interpreter? What about scripting languages? If you can script in a shell language, doesn't that make it a scripting language? In answering these questions, let's start with shells.

Defining what a shell is can be a bit tricky, especially at Microsoft, since pretty much everything at Microsoft has something called a shell. Windows Explorer is a shell. Visual Studio has a component called the shell. Heck - even the Xbox has something they call a shell.

Historically, the term *shell* describes the piece of software that sits over an operating system's core functionality. This core functionality is known as the operating system kernel (shell... kernel... get it?). A shell is the piece of software that lets you access the functionality provided by the operating system. Windows Explorer is properly called a shell because it lets you access the functionality of a Windows system. For our purposes, though, we're more interested in the traditional text-based environment where the user types a command and receives a response. In other words, a shell is a command-line interpreter. The two terms can be used for the most part interchangeably.

SCRIPTING LANGUAGES VS. SHELLS

If this is the case, then what is scripting and why are scripting languages not shells? To some extent, there isn't really a difference. Many scripting languages have a mode in which they take commands from the user and then execute those commands to return results. This mode of operation is called a Read-Evaluate-Print loop or REP loop. Not all scripting languages have these interactive loops, but many do. In what way is a scripting language with a REP loop not a shell? The difference is mainly in the user experience. A proper command-line shell is also a proper user interface. As such, a command line has to provide a number of features to make the user's experience pleasant and customizable. The features that improve the user's experience include aliases (shortcuts for hard-to-type commands), wildcard matching so you don't have to type out full names, and the ability to start other programs without having to do

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

anything special such as calling a function to start the program. Finally, command-line shells provide mechanisms for examining, editing, and re-executing previously typed commands. These mechanisms are called *command history*.

If scripting languages can be shells, can shells be scripting languages? The answer is, emphatically, yes. With each generation, the UNIX shell languages have grown more and more powerful. It's entirely possible to write substantial applications in a modern shell language, such as `bash` or `zsh`. Scripting languages characteristically have an advantage over shell languages, in that they provide mechanisms to help you develop larger scripts by letting you break a script into components or *modules*. Scripting languages typically provide more sophisticated features for debugging your scripts. Next, scripting language runtimes are implemented in a way that makes their code execution more efficient, so that scripts written in these languages execute more quickly than they would in the corresponding shell script runtime. Finally, scripting language syntax is oriented more toward writing an application than toward interactively issuing commands.

In the end, there really is no hard and fast distinction between a shell language and a scripting language. Some of the features that make a good scripting language result in a poor shell user experience. Conversely, some of the features that make for a good interactive shell experience can interfere with scripting. Since PowerShell's goal is to be both a good scripting language and a good interactive shell, balancing the trade-offs between user-experience and scripting authoring was one of the major language design challenges.

1.1.2 Why a new shell? Why now?

In the early part of this decade, Microsoft commissioned a study to identify areas where it could improve its offerings in the server space. Server management, and particularly command-line management of Windows systems, was called out as a critical area for improvement. While some might say that this is like discovering that water is wet, the important point is that people cared about the problem. When comparing the command-line manageability of a Windows system to a UNIX system, Windows was found to be limited, and this was a genuine pain point with customers.

There are a number of reasons for the historically weak Windows command line. First, as mentioned previously, limited effort had been put into improving the command line. Since the average desktop user doesn't care about the command line, it wasn't considered important. Secondly, when writing graphical user interfaces, you need to access whatever you're managing through programmer-style interfaces called Application Programmer Interfaces (APIs). APIs are almost universally binary (especially on Windows), and binary interfaces are not command-line friendly.

Managing Windows through objects

Another factor that drove the need for a new shell model is that, as Windows acquired more and more subsystems and features, the number of issues you had to think about when managing a system increased dramatically. To deal with this increase in complexity, the manageable elements were factored into structured data objects. This collection of *management objects* is known internally at Microsoft as the Windows management surface.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

AUTHOR'S NOTE

Microsoft wasn't the only company that was running into issues due to increased complexity. Pretty much everyone in the industry was having this problem. This led to the Desktop Management Taskforce, an industry organization, creating a standard for management objects called the Common Information Model (CIM). Microsoft's implementation of this standard is called the Windows Management Instrumentation (WMI). Chapter 16 covers PowerShell's support for WMI.

While this factoring addressed overall complexity and worked well for graphical interfaces, it made it much harder to work with using a traditional text-based shell environment.

Finally, as the power of the PC increased, Windows began to move off the desktop and into the corporate data center. In the corporate data center, you have a large number of servers to manage, and the graphical point-and-click management approach that worked well for one machine doesn't scale. All these elements combined to make it clear that Microsoft could no longer ignore the command line.

1.1.3 The last mile problem

Why do we care about command-line management and automation? Because it helps to solve the Information Technology professional's version of the last mile problem. The last mile problem is a classical problem that comes from the telecommunications industry. It goes like this: the telecom industry can effectively amortize its infrastructure costs across all its customers until it gets to the last mile where the service is finally run to an individual location. Installing service across this last mile can't be amortized because it serves only a single location. Also, what's involved in servicing any particular location can vary significantly. Servicing a rural farmhouse is different and significantly more expensive than running service to a house on a city street.

In the Information Technology (IT) industry, the last mile problem is figuring out how to manage each IT installation effectively and economically. Even a small IT environment has a wide variety of equipment and applications. One approach to solving this is through consulting: IT vendors provide consultants who build custom last-mile solutions for each end-user. This, of course, has problems with recurring costs and scalability (it's great for the vendor, though). A better solution for end-users is to empower them to solve their own last mile problems. We do this by providing a toolkit to enable end-users to build their own custom solutions. This toolkit can't merely be the same tools used to build the overall infrastructure as the level of detail required is too great. Instead, you need a set of tools with a higher level of abstraction. This is where PowerShell comes in—its higher-level abstractions allow you to connect the various bits of your IT environment together more quickly and with less effort.

Now that we understand the environmental forces that led to the creation of PowerShell, the need for command-line automation in a distributed object-based operating environment, let's look at the form the solution took.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

1.2 Soul of a new language

The title of this section was adapted from Tracey Kidder's Soul of a New Machine, one of the best non-technical technical books ever written. Kidder's book described how Data General developed a new 32-bit minicomputer, the Eclipse, in a single year. At that time, 32-bit minicomputers were not just new computers; they represented a whole new class of computers. It was a bold, ambitious project; many considered it crazy. Likewise, the PowerShell project is not just about creating a new shell language. We are developing a new class of object-based shell languages. And we've been told more than a few times that we were crazy.

In this section, we're going to cover some of the technological forces that shaped the development of PowerShell. A unique set of customer requirements in tandem with the arrival of the new .NET wave of tools at Microsoft led to this revolution in shell languages.

1.2.1 Learning from history

In section 1.1.2, we described why Microsoft needed to improve the command line. Now let's talk about how we decided to improve it. In particular, let's talk about why we created a new language. This is certainly one of the most common questions people ask about PowerShell (right after "What, are you guys nuts?"). People ask "why not just use one of the UNIX shells?" or "why not extend the existing Windows command line?"

In practice, we did start with an existing shell language. We began with the shell grammar for the POSIX standard shell defined in IEEE Specification 1003.2. The POSIX shell is a mature command-line environment available on a huge variety of platforms including Microsoft Windows. It's based on a subset of the UNIX Korn shell, which is itself a superset of the original Bourne shell. Starting with the POSIX shell gave us a well-specified and stable base. Then we had to consider how to accommodate the differences that properly supporting the Windows environment would entail. We wanted to have a shell optimized for the Windows environment in the same way that the UNIX shells are optimized for this UNIX environment.

To begin with, traditional shells deal only with strings. Even numeric operations work by turning a string into a number, performing the operation, and then turning it back into a string. Given that a core goal for PowerShell was to preserve the structure of the Windows data types, we couldn't simply use the POSIX shell language as is. This factor impacted the language design more than any other. Next, we wanted to support a more conventional scripting experience where, for example, expressions could be used as you would normally use them in a scripting language such as VBScript, Perl, or Python. With a more natural expression syntax, it would be easier to work with the Windows management objects. Now we just had to decide how to make those objects available to the shell.

1.2.2 Leveraging .NET

One of the biggest challenges in developing any computer language is deciding how to represent data in that language. For PowerShell, the key decision was to leverage the .NET object model. .NET is a unifying object representation that is being used across all of the groups at Microsoft. It is a hugely ambitious project that has taken years to come to fruition.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

By having this common data model, all the components in Windows can share and understand each other's data.

One of .NET's most interesting features for PowerShell is that the .NET object model is self-describing. By this, we mean that the object itself contains the information that describes the object's structure. This is important for an interactive environment, as you need to be able to look at an object and see what you can do with it. For example, if PowerShell receives an event object from the system event log, the user can simply inspect the object to see that it has a data stamp indicating when the event was generated.

Traditional text-based shells facilitate inspection because everything is text. Text is great—what you see is what you get. Unfortunately, what you see is all you get. You can't pull off many interesting tricks with text until you turn it into something else. For example, if you want to find out the total size of a set of files, you can get a directory listing, which looks something like the following:

```
02/26/2004  10:58 PM            45,452 Q810833.log
02/26/2004  10:59 PM            47,808 Q811493.log
02/26/2004  10:59 PM            48,256 Q811630.log
02/26/2004  11:00 PM            50,681 Q814033.log
```

You can see where the file size is in this text, but it isn't useful as is. You have to extract the sequence of characters starting at column 32 (or is it 33?) until column 39, remove the comma, and then turn those characters into numbers. Even removing the comma might be tricky, because the thousands separator can change depending on the current cultural settings on the computer. In other words, it may not be a comma—it may be a period. Or it may not be present at all.

It would be easier if you could just ask for the size of the files as a number in the first place. This is what .NET brings to PowerShell: self-describing data that can be easily inspected and manipulated without having to convert it to text until you really need to.

Choosing to use the .NET object model also brings an additional benefit, in that it allows PowerShell to directly use the extensive libraries that are part of the .NET framework. This brings to PowerShell a breadth of coverage rarely found in a new language. Here's a simple example that shows the kinds of things .NET brings to the environment. Say we want to find out what day of the week December 13, 1974 was. We can do this in PowerShell as follows:

```
PS (1) > (get-date "December 13, 1974").DayOfWeek
Friday
```

In this example, the get-date command returns a .NET `DateTime` object, which has a property that will calculate the day of the week corresponding to that date. The PowerShell team didn't need to create a library of date and time manipulation routines for PowerShell—we got them for free by building on top of .NET. And the same `DateTime` objects are used throughout the system. For example, say we want to find out which of two files is newer. In a text-based shell, we'd have to get a string that contains the time each file was updated, covert those strings into numbers somehow, and then compare them. In PowerShell, we can simply do:

```
PS (6) > (dir data.txt).lastwritetime -gt
>> (dir hello.ps1).lastwritetime
>>
True
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

We use the dir command to get the file information objects and then simply compare the last write time of each file. No string parsing is needed.

Now that we're all sold on the wonders of objects and .NET (I'm expecting my check from the Microsoft marketing folks real soon), let's make sure we're all talking about the same thing when we use words like object, member, method, and instance. The next section discusses the basics of object-oriented programming.

1.3 Brushing up on objects

Since the PowerShell environment uses objects in almost everything it does, it's worth running through a quick refresher on objects and how they're used in programming. If you're comfortable with this material, feel free to skip most of this section, but do please read the section on objects and PowerShell.

There is no shortage of "learned debate" (also known as bitter feuding) about what objects are and what object-oriented programming is all about. For our purposes, we'll use the simplest definition. An object is a unit that contains both data (properties) and the information on how to use that data (methods). Take a light bulb object as a simple example. This object would contain data describing its state—whether it's off or on. It would also contain the mechanisms or methods needed to change the on/off state. Non-object-oriented approaches to programming typically put the data in one place, perhaps a table of numbers where 0 is off and 1 is on, and then provide a separate library of routines to change this state. To change its state, the programmer would have to tell these routines where the value representing a particular light bulb was. This could be complicated and is certainly error prone. With objects, because both the data and the methods are packaged as a whole, the user can work with objects in a more direct and therefore simpler manner, allowing many errors to be avoided.

1.3.1 Reviewing object-oriented programming

That's the basics of what objects are. Now what is object-oriented programming? Well, it deals mainly with how you build objects. Where do the data elements come from? Where do the behaviors come from? Most object systems determine the object's capabilities through its type. In the light bulb example, the type of the object is (surprise) `LightBulb`. The type of the object determines what properties the object has (for example, `IsOn`) and what methods it has (for example, `TurnOn` and `TurnOff`).

Essentially, an object's type is the blueprint or pattern for what an object looks like and how you use it. The type `LightBulb` would say that that it has one data element—IsOn—and two methods—`TurnOn()` and `TurnOff()`. Types are frequently further divided into two subsets:

- Types that have an actual implementation of `TurnOn()` and `TurnOff()`. These are typically called classes.
- Types that only describe what the members of the type should look like but not how they work. These are called interfaces.

The pattern `IsOn/TurnOn()/TurnOff()` could be an interface implemented by a variety of classes such as `LightBulb`, `KitchenSinkTap`, or `Television`. All these objects have the same basic pattern for being turned on and off. From a programmer's perspective, if they all

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

have the same interface (that is, the same mechanism for being turned on and off), once you know how to turn one of these objects on or off, you can use any type of object that has that interface.

Types are typically arranged in hierarchies with the idea that they should reflect logical taxonomies of objects. This taxonomy is made up of classes and subclasses. An example taxonomy is shown in figure 1.2.

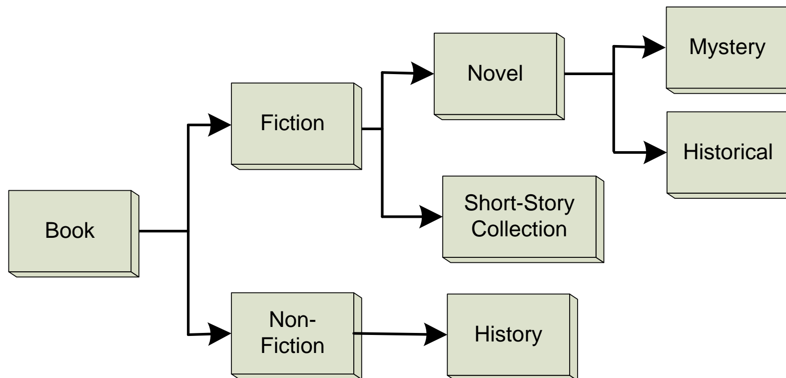

Figure 1.2 This diagram shows how books can be organized in a hierarchy of classes, just as object types can be organized into classes.

In this taxonomy, `Book` is the parent class, `Fiction` and `Non-fiction` are subclasses of `Book`, and so on. While taxonomies organize data effectively, designing a good taxonomy is hard. Frequently, the best arrangement is not immediately obvious. In figure 1.2, it might be better to organize by subject matter first, instead of the Novel/Short-story Collection grouping. In the scientific world, people spend entire careers categorizing items. Since it's hard to categorize well, people also arrange instances of objects into collections by containment instead of by type. A library contains books, but it isn't itself a book. A library also contains other things that aren't books, such as chairs and tables. If at some point you decide to re-categorize all of the books in a library, it doesn't affect what building people visit to get a book. It only changes how you find a book once you reach that building. On the other hand, if the library moves to a new location, you have to learn where it is. Once inside the building, however, your method for looking up books hasn't changed. This is usually called a has-a relationship—a library has-a bunch of books. Now let's see how these concepts are used in the PowerShell environment.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

1.3.2 Objects in PowerShell

We've said that PowerShell is an object-based shell as opposed to an object-oriented language. What do we mean by object-based? In object-based scripting, you typically use objects somebody else has already defined for you. While it's possible to build your own objects in PowerShell, it isn't something that you need to worry about—at least not for most basic PowerShell tasks.

Returning to the LightBulb example, PowerShell would probably use the LightBulb class like this:

```
$lb = get-lightbulb -room bedroom
$lb.TurnOff()
```

Don't worry about the details of the syntax for now—we'll cover that later. The key point is that you usually get an object "foo" by saying:

```
get-foo -option1 -option2 bar
```

rather than saying something like:

```
new foo()
```

as you would in an object-oriented language.

PowerShell commands, called *cmdlets*, use *verb-noun* pairs like `Get-Date`. The `Get-*` verb is used universally in the system to get at objects. Note that we didn't have to worry about whether `LightBulb` is a class or an interface, or care about where in the object hierarchy it comes from. You can get all of the information about the member properties of an object though the get-member command (see the pattern?), which will tell you all about an object's properties.

But enough talk! By far the best way to understand PowerShell is to use it. In the next section, we'll get you up and going with PowerShell, and quickly tour through the basics of the environment.

1.4 Up and running with PowerShell

In this section, we'll look at the things you need to know to get going with PowerShell as quickly as possible. This is a brief introduction intended to provide a taste of what PowerShell can do and how it works. We begin with how to download and install PowerShell and how to start the interpreter once it's installed. Then we'll cover the basic format of commands, command-line editing, and how to use command completion with the Tab key to speed up command entry. Once you're up and running, we'll look at what you can do with PowerShell.

NOTE

The PowerShell documentation package also includes a short Getting Started guide that will include up-to-date installation information and instructions. You may want to take a look at this as well.

1.4.1 Installing PowerShell

How you get PowerShell depends on what operating system you're using. If you're using Windows 7 or Windows Server 2008 R2, you have nothing to do - it's already there. All Microsoft operating systems beginning with Windows 7 include PowerShell as part of the system.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

If you're using Windows Server 2008, PowerShell was included with this operating system but as an optional component which will need to be turned on before you can use it. For earlier Microsoft operating systems, you'll have to download and install the PowerShell package on your computer. For details about supported platforms, etc., go to the PowerShell page on the Microsoft website:

```
http://microsoft.com/powershell
```

This page contains links to the appropriate installers as well as documentation packages and other relevant materials. Alternatively, you can go to Microsoft Update and search for the installer there. Once you've located the installer, follow the instructions to install the package.

1.4.2 Starting PowerShell

Now let's look at how we start PowerShell running. PowerShell follows a model found in many modern interactive environments. It's actually composed of two main parts:

1. The PowerShell engine which interprets the commands
2. A host application that passes commands from the user to the engine.

While there is only one PowerShell engine, there can be many hosts, including hosts written by third-parties like Quest Software's PowerGUI. In version 1 of PowerShell, Microsoft only provided one basic PowerShell host based on the old-fashioned Windows console. Version 2 introduced a much more modern host environment, called the PowerShell Integrated Scripting Environment (PowerShell ISE). We'll look at both of these hosts in the next few sections.

1.4.3 The PowerShell console host

To start an interactive PowerShell session using the console host go to:

```
Start -> Programs -> Windows PowerShell -> Windows PowerShell
```

PowerShell will start and you'll see a screen like that shown in figure 1.3:

Figure 1.3 When you start an interactive PowerShell session, the first thing you see is the PowerShell logo and then the prompt. As soon as you see the prompt, you can begin entering commands.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

This window looks a lot like the old Windows command window (except that it's blue and yellow instead of black and white.) Now type the first command most people type: "dir". This produces a listing of the files on your system, as shown in figure 1.4.

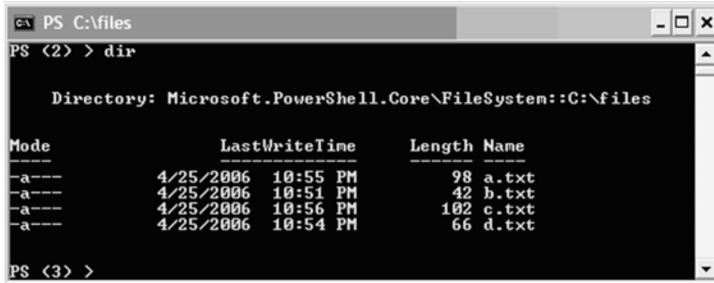

Figure 1.4 At the prompt, type "dir" and press the Enter key. PowerShell will then execute the dir command and display a list of files in the current directory.

As you would expect, the dir command prints a listing of the current directory to standard output.

NOTE

Let's stop for a second and talk about the conventions we're going to use in examples. Since PowerShell is an interactive environment, we'll show a lot of example commands as the user would type them, followed by the responses the system generates. Before the command text, there will be a prompt string that looks like "PS (2) > ". Following the prompt, the actual command will be displayed in bold font. PowerShell's responses will follow on the next few lines. Since PowerShell doesn't display anything in front of the output lines, you can distinguish output from commands by looking for the prompt string. These conventions are illustrated in figure 1.5.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

First Prompt
User enters "1+2+3+4"
PowerShell Outputs the Result 10
PS (1) > **1+2+3+4**
10
PS (2) >
Next Prompt

Figure 1.5 This diagram illustrates the conventions we're using for showing examples in this book. The text that the user types is shown in bold. Prompts and other output from the interpreter are shown in normal weight text.

COMMAND EDITING IN THE CONSOLE

Typing in commands is all well and good but you also want to be able to edit and rerun commands. Command-line editing works the same way in the PowerShell console window as it does for cmd.exe. The available editing features and keystrokes are listed in table 1.1.

Table 1.1 Command editing features

Keyboard Sequence	Editing Operation
Left/Right Arrows	Move the editing cursor left and right through the current command line.
Ctrl-Left Arrow, Ctrl-Right Arrow	Holding the control (CTRL) key down while pressing the left and right arrow keys will move the editing cursor through the current command line one word at a time, instead of one character at a time.
Home	Moves the editing cursor to the beginning of the current command line.
End	Moves the editing cursor to the end of the current command line
Up/Down Arrows	Moves up and down through the command history.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

Insert Key	Toggles between character insert and character overwrite modes.
Delete Key	Deletes the character under the cursor.
Backspace Key	Deletes the character to the left of the cursor.

These key sequences let you create and edit commands effectively at the command line. In fact, they aren't really part of PowerShell at all. These command-line editing features are part of the Windows console subsystem, so they are the same across all console applications.

Users of cmd.exe or any modern UNIX shell will also expect to be able to do command completion. Since this component is common to both host environments, we'll cover how it works in its own section.

Now let's leap into the 21st Century and look at a modern shell environment - the PowerShell ISE.

1.4.4 The PowerShell Integrated Scripting Environment

Starting with version 2, PowerShell includes a modern integrated environment for working with PowerShell - the Integrated Scripting Environment (or ISE). To start the PowerShell ISE go to:

```
Start -> Programs -> Windows PowerShell -> Windows PowerShellISE
```

PowerShell will start and you'll see a screen like that shown in figure 1.4:

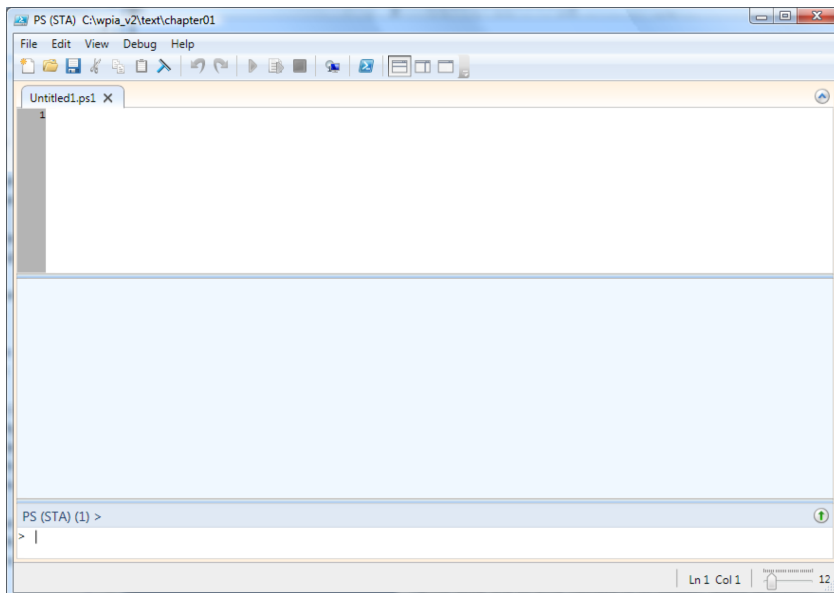

Figure 1.4 The PowerShell Integrated Scripting Environment

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

You can see that, by default, the window is divided into three parts: the text entry area at the bottom, the output window in the middle and an editor at the top. As we did in the console window, let's run the `dir` command. Type `dir` into the bottom pane and hit enter. The command will disappear from the bottom pane, appear in the middle pane followed by the output of the command as shown in figure 1.5.

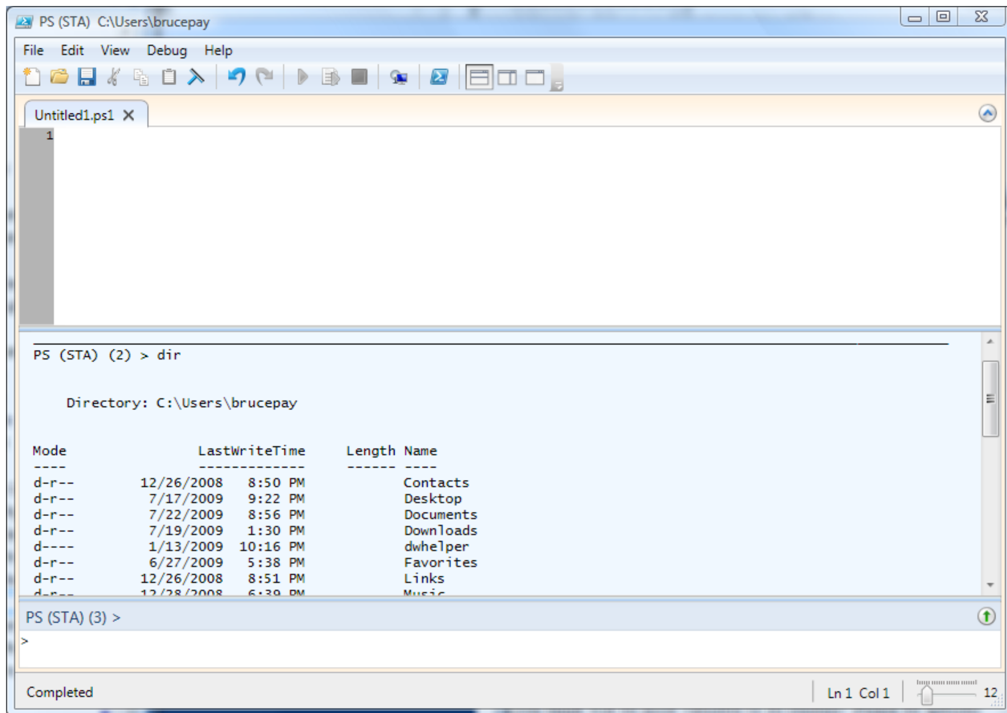

Figure 1.5 This figure shows running the dir command in the PowerShell ISE. The command is entered in the bottom pane and the result of the command is shown in the output pane in the middle.

Because the ISE is a real Windows application, it follows all of the Windows Common User Access (CUA) guidelines. The left and right arrows work as expected. The up and down arrows will move you through the history of commands that you've entered.

Something that requires special mention is how Ctrl-C works. By default, this is the key sequence for copying into the clipboard in Windows. It is also, however, the way to interrupt a running command in most shells. As a result, the ISE has to treat Ctrl-C in a special way. When there is something selected, Ctrl-C copies the selection. If there is a command running and there is no selection, then the running command will be interrupted.

There is also another way to interrupt a running command. You may have noticed the two buttons immediately above the command entry pane - the ones that look like the play and stop buttons on a media player.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

As one might expect, the green "play" button will run a command just like if you hit enter. If there is a command running, the "play" button is disabled (grayed out) and the red "stop" button is enabled. Clicking on this button will stop the currently running command.

USING THE EDITOR PANE

The topmost pane in the ISE is a script editor that understands the PowerShell language. This editor will do syntax highlighting as you type in script text. It will also let you select a region and either hit the "play" button above the pane or press the F8 key to execute the part of the script you want to test out. If there is nothing selected in the window, then the whole script will be run. If you are editing a script file, the ISE will ask if you want to save the file before running it.

Another nice feature of the ISE editor is the ability to have multiple files open at once, each in individual tabs as shown in figure 1.6.

Figure 1.6 Multiple tabs in the ISE editor

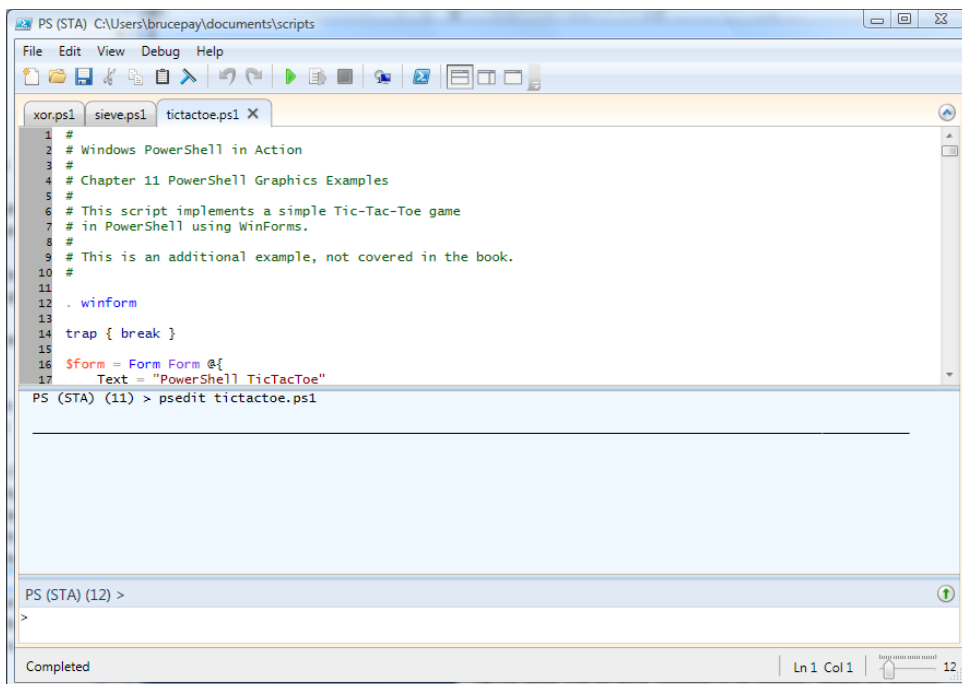

Figure 1.6 This figure shows using multiple tabs in the ISE editor. Each new file that is opened gets its own tab. Files can be opened from the file menu or by using the 'psedit' command in the command window as shown.

And finally, as well as multiple editor tabs, the ISE also allows you to have multiple session tabs as shown in figure 1.7. In this figure you can see that there are 4 session tabs and within

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

each tab, there can be multiple editor tabs. This makes the ISE a very powerful way to organize your work and easily multitask between different activities.

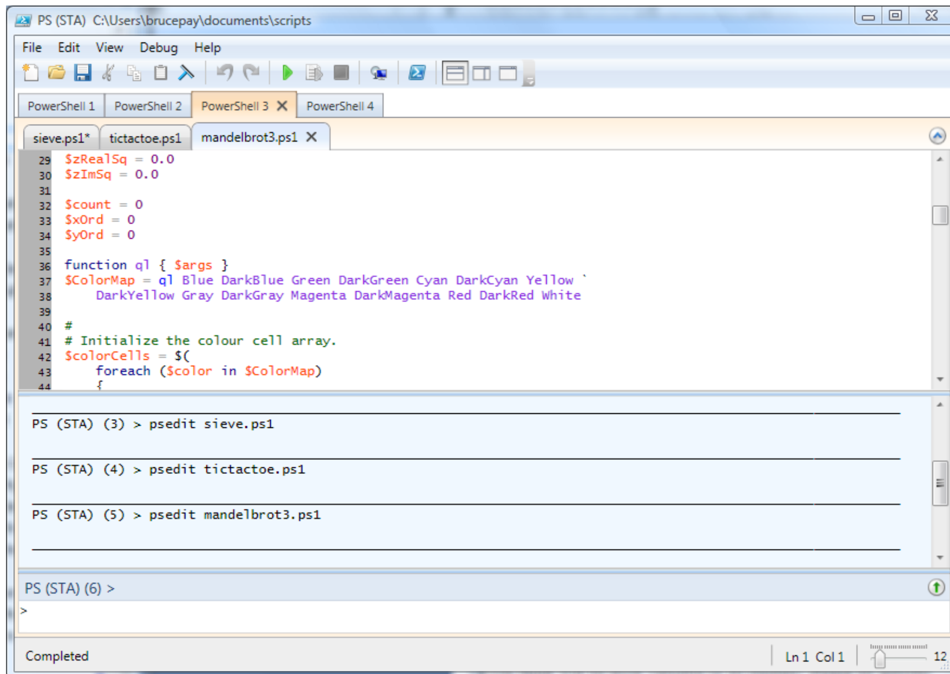

Figure 1.7 This figure shows how multiple session tabs are displayed in the ISE. Note that each session tab has its own set of editor tabs.

These are the basic concepts in the ISE. But the ISE is not just a tool for writing, testing and debugging PowerShell scripts. It's also *scriptable* by PowerShell. This means that you can use scripts to manipulate the contents of buffers, create new tabs and menu items, etc. This allows you to use the ISE as part of your application in much the same way as the EMACS editor was a component of custom applications. There are some limitations to this in the first version of the ISE - we didn't have time to do everything we wanted (there's never enough time) but the result is still very powerful. We'll see more of this later on.

NOTE:

Ok - so why is this an 'ISE' instead of an 'IDE' like Visual Studio? The big difference is that the ISE is intended for interactive use of PowerShell, not just the creation of PowerShell applications. One of the biggest differences between the two approaches is the lack of a project system in the ISE.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

1.4.5 Command completion

One of the most useful editing features in PowerShell is *command completion*, also called *tab-completion*. While cmd.exe does have tab-completion, PowerShell's implementation is significantly more powerful. Command completion allows you to partially enter a command, then hit the Tab key and have PowerShell try to fill in the rest of the command. By default, PowerShell will do tab completion against the file system, so if you type a partial file name and then hit Tab, the system matches what you've typed against the files in the current directory and returns the first matching file name. Hitting Tab again takes you to the next match, and so on. PowerShell also supplies the powerful capability of tab-completion on wild cards (see chapter 4 for information on PowerShell wild cards). This means that you can type:

```
PS (1) > cd c:\pro*files<tab>
```

and the command is expanded to:

```
PS (2) > cd 'C:\Program Files'
```

PowerShell will also do tab-completion on partial cmdlet names. If you enter a cmdlet name up to the dash and then hit the Tab key, the system will step through the matching cmdlet names.

So far, this isn't much more interesting than what cmd.exe provide. What is significantly different is that PowerShell also does completion on parameter names. If you enter a command followed by a partial parameter name and hit Tab, the system will step through all of the possible parameters for that command.

PowerShell also does tab-completion on variables. If you type a partial variable name and then hit the Tab key, PowerShell will complete the name of the variable.

And finally, PowerShell does completion on properties in variables. If you've used the Microsoft Visual Studio development environment, you've probably seen the Intellisense feature. Property completion is kind of a limited Intellisense capability at the command line. If you type something like:

```
PS (1) > $a="abcde"
PS (2) > $a.len<tab>
```

The system expands the property name to:

```
PS (2) > $a.Length
```

Again, the first Tab returns the first matching property or method. If the match is a method, an open parenthesis is displayed:

```
PS (3) > $a.sub<tab>
```

which produces:

```
PS (3) > $a.Substring(
```

Note that the system corrects the capitalization for the method or property name to match how it was actually defined. This doesn't really impact how things work. PowerShell is case-insensitive by default whenever it has to match against something. (There are operators that allow you to do case-sensitive matching, which are discussed in chapter 3).

Version 2 of PowerShell introduced an additional tab-completion feature (suggested by a PowerShell user no less). PowerShell remembers each command you type. You can access previous commands using the arrow keys or show them using the `Get-History` command. A new feature was added to allow you to do tab completion against the command history. To recall the first command containing the string "abc", type the # sign, followed by the pattern of the command you want to find and then hit the Tab key:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

```
PS (4) > #abc<tab>
```

This will expand the command line to

```
PS (4) > $a="abcde"
```

You can also select a command from the history by number (this is why we have numbers in the prompt). To do this, type the # sign, then the number of the command to run followed by tab

```
PS (5) > #2<tab>
```

And this should expand to

```
PS (5) > $a="abcde"
```

NOTE:

The PowerShell tab completion mechanism is user extendable. While the path completion mechanism is built into the executable, features such as parameter and property completion are implemented through a shell function that users can examine and modify. The name of this function is `TabExpansion`. Chapter 7 describes how to write and manipulate PowerShell functions.

1.5 Dude! Where's my code?

Ok enough talk, let's see some more example code! First, we'll revisit the `dir` example we saw earlier. This time, instead of simply displaying the directory listing, we'll save it into a file using output redirection just like in other shell environments. In the following example, we'll use `dir` to get information about a file named "somefile.txt" in the root of the `C:` drive. Using redirection, we direct the output into a new file "c:\foo.txt" and then use the `type` command to display what was saved. This looks like

```
PS (2) > dir c:\somefile.txt > c:\foo.txt
PS (3) > type c:\foo.txt

    Directory: Microsoft.PowerShell.Core\FileSystem::C:\

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---        11/17/2004   3:32 AM          0 somefile.txt
PS (4) >
```

As you can see, commands work more or less as you'd expect.

AUTHOR'S NOTE

Ok - nobody really has a file named "somefile.txt" in the root of their C drive (except me). For the purpose of this example, just choose any file that does exist and the example will work fine, though obviously the output will be different.

Let's go over some other things that should be familiar to you.

1.5.1 Navigation and Basic Operations

The PowerShell commands for working with the file system should be pretty familiar to most users. You navigate around the file system with the `cd` command. Files are copied with the `copy` or `cp` commands, moved with the `move` and `mv` commands and removed with the `del` or

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

`rm` commands. Why two of each command you might be asking? One set are the names familiar to cmd.exe/DOS users and the other are familiar to UNIX users. In practice they are actually *aliases* for the same command, designed to make it easier for people to get going with PowerShell. One thing to keep in mind however, is that while the commands are similar they are not exactly the same as either of the other two systems. You can use the `Get-Help` command to get help about these commands. Here is the output of `Get-Help` for the `dir` command.

```
PS (1) > get-help dir

NAME
    Get-ChildItem

SYNOPSIS
    Gets the items and child items in one or more specified
     locations.


SYNTAX
    Get-ChildItem [-Exclude <string[]>] [-Force] [-Include
    <string[]>] [-Name] [-Recurse] [[-Path] <string[]>] [[-
    Filter] <string>] [<CommonParameters>]

    Get-ChildItem [-Exclude <string[]>] [-Force] [-Include
    <string[]>] [-Name] [-Recurse] [[-Filter] <string>] [-L
    iteralPath] <string[]> [<CommonParameters>]


DETAILED DESCRIPTION
    The Get-Childitem cmdlet gets the items in one or more
    specified locations. If the item is a container, it get
    s the items inside the container, known as child items.
     You can use the Recurse parameter to get items in all
    child containers.

    A location can be a file system location, such as a dir
    ectory, or a location exposed by another provider, such
     as a registry hive or a certificate store.


RELATED LINKS
    About_Providers
    Get-Item
    Get-Alias
    Get-Location
    Get-Process

REMARKS
    To see the examples, type: "get-help Get-ChildItem -exa
    mples".
    For more information, type: "get-help Get-ChildItem -de
    tailed".
    For technical information, type: "get-help Get-ChildIte
    m -full".
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

The PowerShell help subsystem contains information about all of the commands provided with the system and is a great way to explore what's available. You can even use wildcard characters to search through the help topics (version 2 and later). Of course this is the simple text output. The PowerShell ISE also includes help in the richer windows format and will even let you select an item then hit `F1` to view the help for the item.

1.5.2 Basic expressions and variables

In addition to running commands, PowerShell can also evaluate expressions. In effect, it operates as a kind of calculator. Let's evaluate a simple expression:

```
PS (4) > 2+2
4
```

Notice that as soon as you typed the expression, the result was calculated and displayed. It wasn't necessary to use any kind of print statement to display the result. It is important to remember that whenever an expression is evaluated, the result of the expression is output, not discarded. We'll explore the implications of this in later sections.

Here are few more examples of PowerShell expressions examples:

```
PS (5) > (2+2)*3
12
PS (6) > (2+2)*6/2
12
PS (7) > 22/7
3.14285714285714
```

You can see from these examples that PowerShell supports most of the basic arithmetic operations you'd expect, including floating point.

NOTE

PowerShell supports single and double precision floating point, as well as the .NET decimal type. See chapter 3 for more details.

Since we've already shown how to save the output of a command into a file using the redirection operator, let's do the same thing with expressions:

```
PS (8) > (2+2)*3/7
1.71428571428571
PS (9) > (2+2)*3/7 > c:\foo.txt
PS (10) > type c:\foo.txt
1.71428571428571
```

Saving expressions into files is useful; saving them in variables is more useful:

```
PS (11) > $n = (2+2)*3
PS (12) > $n
12
PS (13) > $n / 7
1.71428571428571
```

Variables can also be used to store the output of commands:

```
PS (14) > $files = dir
PS (15) > $files[1]
    Directory: Microsoft.PowerShell.Core\FileSystem::C:\Document
    s and Settings\brucepay
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

```
Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----         4/25/2006  10:32 PM            Desktop
```

In this example, we extracted the second element of the collection of file information objects returned by the dir command. We were able to do this because we saved the output of the dir command as an array of objects in the `$files` variable.

> **NOTE:**
>
> Note that collections in PowerShell start at 0, not at 1. This is a characteristic we've inherited from the .NET Common Language Runtime specification. This is why $files[1] is actually extracting the second element, not the first.

PowerShell has a very rich set of operators which are covered in detail in chapters 4 and 5.

1.5.3 Processing data

As we've seen in the preceding sections, we can run commands to get information and then store it in files and variables. Now let's do some processing on that data. First we'll look at how to sort objects and how to extract properties from those objects. Then we'll look at using the PowerShell flow control statements to write scripts that use conditionals and loops to do more sophisticated processing.

SORTING OBJECTS

First let's sort a list of files. Here's the initial list, which by default is sorted by name.

```
PS (16) > cd c:\files
PS (17) > dir
    Directory: Microsoft.PowerShell.Core\FileSystem::C:\files

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---         4/25/2006  10:55 PM         98 a.txt
-a---         4/25/2006  10:51 PM         42 b.txt
-a---         4/25/2006  10:56 PM        102 c.txt
-a---         4/25/2006  10:54 PM         66 d.txt
```

The output of this shows the basic properties on the file system objects sorted by the name of the file. Now, let's run it through the sort utility:

```
PS (18) > dir | sort
    Directory: Microsoft.PowerShell.Core\FileSystem::C:\files

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---         4/25/2006  10:55 PM         98 a.txt
-a---         4/25/2006  10:51 PM         42 b.txt
-a---         4/25/2006  10:56 PM        102 c.txt
-a---         4/25/2006  10:54 PM         66 d.txt
```

Granted, it's not very interesting. Sorting an already sorted list by the same property gives you the same result. Let's do something a bit more interesting. Let's sort by name in descending order:

```
PS (19) > dir | sort -descending
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

```
Directory: Microsoft.PowerShell.Core\FileSystem::C:\files

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---         4/25/2006  10:54 PM         66 d.txt
-a---         4/25/2006  10:56 PM        102 c.txt
-a---         4/25/2006  10:51 PM         42 b.txt
-a---         4/25/2006  10:55 PM         98 a.txt
```

So there you have it—files sorted by name in reverse order. Now let's sort by something other than the name of the file. Let's sort by file length. You may remember from an earlier section how hard it would be to sort by file length if the output were just text.

NOTE:

In fact, on a UNIX system, this sort command looks like:

```
ls -l | sort -n -k 5
```

which, while pithy, is pretty opaque. Here's what it's doing. The -n option tells the sort function that you want to do a numeric sort. -k tells you which field you want to sort on. (The sort utility considers space-separated bits of text to be fields.) In the output of the `ls -l` command, the field containing the length of the file is at offset 5, as shown in the following:

```
-rw-r--r--  1 brucepay  brucepay  5754 Feb 19   2005 index.html

-rw-r--r--  1 brucepay  brucepay   204 Aug 19  12:50 page1.htm
```

We need to set things up this way because `ls` produces unstructured strings. We have to tell sort how to parse those strings before it can sort them.

In PowerShell, when we use the `Sort-Object` cmdlet, we don't have to tell it to sort numerically—it already knows the type of the field, and we can specify the sort key by property name instead of a numeric field offset.

```
PS (20) > dir | sort -property length
    Directory: Microsoft.PowerShell.Core\FileSystem::C:\files

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---         4/25/2006  10:51 PM         42 b.txt
-a---         4/25/2006  10:54 PM         66 d.txt
-a---         4/25/2006  10:55 PM         98 a.txt
-a---         4/25/2006  10:56 PM        102 c.txt
```

In this example, we're working with the output as objects; that is, things having a set of distinct characteristics accessible by name.

SELECTING PROPERTIES FROM AN OBJECT

In the meantime, let's introduce a new cmdlet—Select-Object. This cmdlet allows you to either select some of the objects piped into it or select some properties of each object piped into it.

Say we want to get the largest file in a directory and put it into a variable:

```
PS (21) > $a = dir | sort -property length -descending |
>> select-object -first 1
>>
PS (22) > $a
    Directory: Microsoft.PowerShell.Core\FileSystem::C:\files

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---          4/25/2006  10:56 PM        102 c.txt
```

From this we can see that the largest file is c.txt.

> **NOTE**
>
> Note the secondary prompt ">>" in the previous example. The first line of the command ended in a pipe symbol. The PowerShell interpreter noticed this, saw that the command was incomplete, and prompted for additional text to complete the command. Once the command is complete, you type a second blank line to send the command to the interpreter.

Now say we want only the name of the directory containing the file and not all of the other properties of the object. We can also do this with Select-Object. As with the sort cmdlet, Select-Object also takes a -property parameter (you'll see this frequently in the PowerShell environment—commands are consistent in their use of parameters).

```
PS (23) > $a = dir | sort -property length -descending |
>> select-object -first 1 -property directory
>>
PS (24) > $a

Directory
---------
C:\files
```

We now have an object with a single property.

PROCESSING WITH THE FOREACH-OBJECT CMDLET

The final simplification is to get just the value itself. Let's introduce a new cmdlet that lets you do arbitrary processing on each object in a pipeline. The Foreach-Object cmdlet executes a block of statements for each object in the pipeline.

```
PS (25) > $a = dir | sort -property length -descending |
>> select-object -first 1 |
>> foreach-object { $_.DirectoryName }
>>
PS (26) > $a
C:\files
```

This shows that we can get an arbitrary property out of an object, and then do arbitrary processing on that information using the Foreach-Object command. Combining those features, here's an example that adds up the lengths of all of the objects in a directory.

```
PS (27) > $total = 0
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

```
PS (28) > dir | foreach-object {$total += $_.length }
PS (29) > $total
308
```

In this example, we initialize the variable $total to 0, then add to it the length of each file returned by the dir command and finally display the total.

PROCESSING OTHER KINDS OF DATA

One of the great strengths of the PowerShell approach is that once you learn a pattern for solving a problem, you can use this same pattern over and over again. For example, say we want to find the largest three files in a directory. The command line might look like this:

```
PS (1) > dir | sort -desc length | select -first 3
    Directory: Microsoft.PowerShell.Core\FileSystem::C:\files

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---         4/25/2006  10:56 PM        102 c.txt
-a---         4/25/2006  10:55 PM         98 a.txt
-a---         4/25/2006  10:54 PM         66 d.txt
```

We ran the dir command to get the list of file information objects, sorted them in descending order by length, and then selected the first three results to get the three largest files.

Now let's tackle a different problem. We want to find the three processes on the system with the largest working set size. Here's what this command line looks like:

```
PS (2) > get-process | sort -desc ws | select -first 3
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
   1294      43    51096      81776   367    11.48   3156 OUTLOOK
    893      25    55260      73340   196    79.33   5124 iexplore
   2092      64    42676      54080   214   187.23    988 svchost
```

This time we run Get-Process to get the data and sort on the working set instead of the file size. Otherwise the pattern is identical to the previous example. This command pattern can be applied over and over again. For example, to get the three largest mailboxes on an Exchange mail server, the command might look like:

```
get-mailboxstatistics | sort -desc  TotalItemSize | select -first 3
```

Again the pattern is repeated except for the `Get-MailboxStatistics` command and the property to filter on.

Even when we don't have a specific command for the data we're looking for and have to use other facilities such as WMI, we can continue to apply the pattern. Say we want to find the three drives on the system that have the most free space. To do this we need to get some data from WMI. Not surprisingly, the command for this is `Get-WmiObject`. Here's how we'd use this command:

```
PS (4) > get-wmiobject win32_logicaldisk |
>> sort -desc freespace | select -first 3 |
>> format-table -autosize deviceid, freespace
>>
deviceid   freespace
--------   ---------
C:       97778954240
T:       31173663232
D:         932118528
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

Once again, the pattern is almost identical. The Get-WmiObject command returns a set of objects from WMI. We pipe these objects into sort and sort on the freespace property, then use Select-Object to extract the first three.

NOTE:

Because of this ability to apply a command pattern over and over, most of the examples in this book are deliberately generic. The intent is to highlight the pattern of the solution rather than show a specific example. Once you understand the basic patterns, you can effectively adapt them to solve a multitude of other problems.

1.5.4 Flow control statements

Pipelines are great, but sometimes you need more control over the flow of your script. PowerShell has the usual script flow control statements found in most programming languages. These include the basic if statements, a very powerful switch statement, and various loops like a while loop, for and foreach loops and so on. Here's an example showing use of the while and if statements.

```
PS (1) > $i=0
PS (2) > while ($i++ -lt 10) { if ($i % 2) {"$i is odd"}}
1 is odd
3 is odd
5 is odd
7 is odd
9 is odd
PS (3) >
```

In this example, we're using the while loop to count through a range of numbers, printing out only the odd numbers. In the body of the while loop, we have an if statement that tests to see whether the current number is odd, and then writes out a message if it is. We can do the same thing using the foreach statement and the range operator (..), but much more succinctly:

```
PS (3) > foreach ($i in 1..10) { if ($i % 2) {"$i is odd"}}
1 is odd
3 is odd
5 is odd
7 is odd
9 is odd
```

The foreach statement iterates over a collection of objects and the range operator is a way to generate a sequence of numbers. The two combine to make looping over a sequence of numbers very clean.

Of course, since the range operator generates a sequence of numbers and numbers are objects like everything else in PowerShell, you can implement this using pipelines and the ForEach-Object cmdlet:

```
PS (5) > 1..10 | foreach { if ($_ % 2) {"$_ is odd"}}
1 is odd
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

```
3 is odd
5 is odd
7 is odd
9 is odd
```

These examples only scratch the surface of what you can do with the PowerShell flow control statements (just wait 'till you see the `switch` statement!) The complete set of control structures is covered in detail in chapter 6 with lots of examples.

1.5.5 Scripts and Functions

What good is a scripting language if you can't package commands into scripts? PowerShell lets you do this by simply putting your commands into a text file with a ".ps1" extension and then running that command. You can even have parameters in your scripts. Put the following text into a hello.ps1

```
param($name = "bub")
"Hello $name, how are you?"
```

Notice that the `param` keyword is used to define a parameter called `$name.` The parameter is given a default value "bub". Now we can run this script from the PowerShell prompt by typing the name as ".\hello". We need the ".\" to tell PowerShell to get the command from the current directory (chapter 13 explains why this is needed.).

The first time we run this script, we won't specify any arguments.

```
PS (1) > .\hello
Hello bub, how are you?
```

and we see that the default value was used in the response. Let's run it again but this time we'll specify an argument.

```
PS (2) > .\hello Bruce
Hello Bruce, how are you?
```

Now the argument is in the output instead of the default value. Of course sometimes we just want to have subroutines in our code. PowerShell address this need through functions. Let's turn our hello script into a function. Here's what it looks like:

```
PS (3) > function hello {
>> param($name = "bub")
>> "Hello $name, how are you"
>> }
>>
```

The body of the function is exactly the same as the script. The only thing we've added is the `function` keyword, the name of the function and braces around the body of the function. Now let's run it, first with no arguments like we did with the script

```
PS (4) > hello
Hello bub, how are you
```

and then with an argument:

```
PS (5) > hello Bruce
Hello Bruce, how are you
```

Obviously the function operates in the same way as the script except that PowerShell didn't have to load it from a disk file so it's a bit faster to call. Scripts and functions are covered in detail in chapter 7.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

1.5.6 Remoting and the Universal Execution Model

In the previous sections we've see the kinds of things we can do with PowerShell on a single computer. But the computing industry has long since moved beyond a one-computer world. Being able to manage groups of computers, without having to physically visit each one is critical in the modern information technology world. To address this, PowerShell Version 2 introduced remote execution capabilities (remoting) and the *Universal Execution Model* - a fancy term that really just means that if it works locally, then it should work remotely.

AUTHOR'S NOTE

At this point you should be asking "if this is so important why wasn't it in version 1"? In fact it was planned from shortly *before* day 1 of the PowerShell project. But - to make something universal, secure and simple is, in fact, very hard.

The core of PowerShell remoting is the `Invoke-Command` command, aliased to `icm`. This command allows you to invoke a block of PowerShell script on the current computer, on a remote computer or on a thousand remote computers. Let's see some of this in action. Our example scenario will be to check the version of the Microsoft Management Console host program installed on a computer. We might need to do this because we want to install an MMC extension (called a *snapin)* on a set of machines and this snapin requires a minimum version of the MMC host. We can do this locally by using the `Get-Command` command (`gcm`) to retrieve information about `mmc.exe` - the executable for the MMC host.

```
PS (1) > (gcm mmc.exe).FileVersionInfo.ProductVersion
6.1.7069.0
```

This is a pretty simple one-line command and it shows us that we have version 6.1 of `mmc.exe` installed on the local machine. Now let's run it on a remote machine. (We'll assume that our target machine names are stored in the variables `$m1` and `$m2`.) Let's check the version on the first machine. We'll run the same command as before, but this time we'll enclose it in braces as an argument to `icm`. We also give the `icm` command the name of the host to execute on.

```
PS {2) > icm $m1 {
>> (gcm mmc.exe).FileVersionInfo.ProductVersion
>> }
>>
6.0.6000.16386
```

Oops - this machine has an older version of `mmc.exe`. OK - let's try machine 2. We run exactly the same command but pass in the name of the second machine this time.

```
PS {3) > icm $m2 {
>> (gcm mmc.exe).FileVersionInfo.ProductVersion
>> }
>>
6.1.7069.0
```

and this machine is up-to-date. At this point we've addressed the need to physically go to each machine but we're still executing the commands one at a time. Let's fix that too by putting the machines to check into a list. We'll use an array variable for this example however this list

could come from a file, an Excel spreadsheet, a database or a web service. Let's run the command with the machine list:

```
PS {4) > $mlist = $m1,$m2
PS {5) > icm $mlist {
>> (gcm mmc.exe).FileVersionInfo.ProductVersion
>> }
>>
6.0.6000.16386
6.1.7069.0
```

and we same the same results as last time, but as a list instead of one at a time. In practice, most of our machines are probably up to date, so we really only care about the ones that don't have the correct version of the software. We can use the `where` command to filter out the machines we don't care about.

```
PS {6) > icm $mlist {
>> (gcm mmc.exe).FileVersionInfo.ProductVersion
>> } | where { $_ -notmatch '6\.1' }
>>
6.0.6000.16386
```

This returns the list of machines that have out-of-date `mmc.exe` versions. There's still a problem however - we see the version number but not the computer's name. Obviously we'll need this too if we're going to update those machines. To address this, PowerShell remoting automatically adds the name of the originating computer to each received object using a special property called "PSComputerName".

Now let's jump a head a bit and see how much effort it would be to produce a nice table of computer names and version numbers. We'll run the remote command again, use `where` to filter the results, extract the fields we want to display using the `select` command and finally format the report using the `Format-Table` command. For the purpose of this example, we'll add the machine lists together so we know we'll get two records in our report. Here's what the command looks like:

```
PS {7) > icm ($mlist + $mlist) {
>>           (gcm mmc.exe).FileVersionInfo.ProductVersion } |
>>       where { $_ -notmatch '6\.1' } |
>>       select @{n="Computer"; e={$_.PSComputerName}},
>>              @{n="MMC Version"; e={$_}} |
>>       format-table -auto
>>

Computer       MMC Version
--------       -----------
brucepaydev07 6.0.6000.16386
brucepaydev07 6.0.6000.16386
```

While the command may look a bit scary, we were able to produce our report with very little effort. And the techniques we used for formatting the report can be used over and over again. This example shows how easily PowerShell remoting can be used to expand the reach of your scripts to cover 10, hundreds or thousands of computers all at once. But sometimes you just want to work with a single remote machine interactively. Let's see how to do that.

The `Invoke-Command` command is the way to programmatically execute PowerShell commands on a remote machine. When we want to connect to a machine so we can interact with it on a 1-1 basis, we use the `Enter-PSSession` command. This command allows you to

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

start an interactive 1-1 session with a remote computer. Running `Enter-PSSession` looks like:

```
PS (11) > enter-pssession server01

[server01]: PS > (gcm mmc.exe).FileVersionInfo.ProductVersion
6.0.6000.16386
[brucepaydev07]: PS > get-date

Sunday, May 03, 2009 7:40:08 PM

[server01]: PS > exit
PS (12) >
```

As shown in the example, when we connect to the remote computer, our prompt changes to indicate that we are working remotely. Otherwise, once connected, we can pretty much interact with the remote computer the same way we would with a local machine. When we're done, we exit the remote session with the `exit` command and this pops us back to the local session. This brief introduction cover some very powerful techniques but we've only begun to cover all of the things remoting let's you do.

At this point, we'll end of our "Cook's tour" of PowerShell. We've only breezed over the features and capabilities of the environment. There are many other areas of PowerShell that we haven't covered here, especially in Version 2 which introduced advanced functions and scripts, modules, transactions, eventing support and many more features. In the subsequent chapters, we'll cover each of the elements discussed here in detail and a whole lot more.

1.6 Summary

This chapter covered what PowerShell is and, just as important, why it is. We also took a whirlwind tour through some simple examples of using PowerShell interactively. Here are the key points that were covered:

- PowerShell is the new command-line/scripting environment from Microsoft Corporation. Since its introduction with Windows Server 2008, PowerShell has rapidly moved to the center of Microsoft server and application management technologies. Many of the most important server products now use PowerShell as their management layer including Exchange, Active Directory and SQL Server.
- The Microsoft Windows management model is primarily object-based, through .NET, COM and WMI. This required us to take a novel approach to command-line scripting, incorporating object-oriented concepts into a command-line shell. PowerShell uses the .NET object model as the base for its type system but can also access other object types like WMI.
- PowerShell is an interactive environment with two different host applications - the console host `powershell.exe` and the graphical host `powershell_ise.exe`. It can also be "hosted" in other applications to act as the scripting engine for those applications.
- Shell operations like navigation and file manipulation in PowerShell are very similar to what you're used to in other shells.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542

- The way to get help about things in PowerShell is to use the Get-Help command (or selecting text and pressing F1 in the ISE).
- PowerShell is a very sophisticated with a full range of calculation, scripting and text processing capabilities.
- PowerShell Version 2 introduced a comprehensive set of remoting features to allow you to do scripted automation of large collections of computers.

In the following chapters, we'll look at each of the PowerShell features we showed you here in much more detail. Stay tuned!

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=542