Windows® PowerShell™ Unleashed
Copyright © 2009 by Pearson Education, Inc.

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-10: 0-672-32988-3

Library of Congress Cataloging-in-Publication Data:
Kopczynski, Tyson.
    Windows PowerShell unleashed / Tyson Kopczynski, Pete Handley, Marco Shaw.
      p. cm.
    Includes index.
    1. Windows PowerShell (Computer program language) I. Handley, Pete. II. Shaw, Marco. III. Title.
QA76.73.W56K67 2008
005.2’82—dc22
2008041296

Printed in the United States of America
First Printing December 2008

Trademarks
All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer
Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Bulk Sales
Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact:

U.S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales
international@pearson.com
Introduction

Well, we are back for yet another *PowerShell Unleashed* book. However, unlike just a simple revision of the existing book, which most likely would have resulted in only just a few updated chapters, I decided to instead treat this release in the series as almost a completely new book. Granted, the Community Technology Release (CTP) of PowerShell 2.0 did help drive the need to update all aspects of this edition. Nonetheless, there was also a lot of feedback (some positive and some negative) about how the first book could be improved.

So, based on this feedback and the looming PowerShell 2.0 feature list, I set about making a major revision to the book. To start off right, I decided to address how the PowerShell language was covered in the series. After all, the first book in the series was script heavy, but lacking when it came to explaining some of the basics about the PowerShell language. Additionally, we wanted to go into greater detail about how PowerShell could be used to manage Windows resources while further addressing some of the finer technical details of PowerShell's architecture. Needless to say, all of these changes required a reorganization to not only the layout of the book, but also its size.

The bottom line, in this new edition, there are six completely new chapters with the rest of the existing chapters either being extensively rewritten or updated. With all this extra content, the book needed additional authors to jump on board and help pound out the book's technical prose. Thus, joining me on this book as coauthors were Marco Shaw (PowerShell MVP) and Peter Handley (contributing author from the first book). Together, Marco and Peter made great additions to this book and infused fantastic ideas together with even better content—all while writing their chapters.

Finally, the primary goal of this book was to start down the path of explaining the features found in the future 2.0 release of PowerShell. After all, with the 2.0 CTP release late last year, the PowerShell product team ignited our imaginations with the possibilities for what might come down the road (remoting). So, we simply had to do our best to explain the new features. However, given that the 2.0 version is still just a CTP and not a beta, we also walked down a slippery slope, considering that some of these features may not exist in the PowerShell 2.0 RTM. Naturally, like a good reporter might do, we did our best. In the end, we tried to include 2.0 content where applicable while also dedicating an entire chapter to only 2.0 features deemed too important to ignore or voted most likely to survive the beta.

We hope our efforts result in a more comprehensive PowerShell book that can act as both a reference for the current PowerShell 1.0 release while also providing insight into where PowerShell might go with the 2.0 release.
Who Is This Book’s Intended Audience?

This *Unleashed* book is intended for an intermediate level of systems administrators who have invested time and energy learning Windows scripting and want to translate those skills into PowerShell skills while learning how it can meet their real-world needs. This book has been written so that anyone with a scripting background can understand what PowerShell is and how to use it, but by no means is it meant to be a complete PowerShell reference. Instead, think of it as a resource for learning how PowerShell can be applied in your own environment. Therefore, the structure of this book reflects that focus by including numerous command examples and working scripts.

How This Book Is Organized

The book is divided into the following three parts:

- **Part I, “Introduction to PowerShell”—**In this section, you are introduced to PowerShell and some of its internal workings. Topics covered include items such as why PowerShell came into existence, general concepts about PowerShell and how it is constructed, and an in-depth review of PowerShell security.

- **Part II, “Using PowerShell”—**In this section, you learn more about the PowerShell scripting language, how to use PowerShell to manage Windows resources, and important best practices to follow when using PowerShell. Specific topics covered range from working with the Windows file system, the Registry, permissions, strings, and Windows Management Instrumentation (WMI) to understanding PowerShell language concepts such as loops, functions, arrays, and so on.

- **Part III, “Managing Microsoft Technologies with PowerShell”—**In this section, you learn how PowerShell can be used to manage Microsoft technologies. Topics covered include using PowerShell to manage Active Directory, Exchange Server 2007, and Systems Center Operations Manager 2007. Additionally, you learn how to programmatically use PowerShell to manage systems and gain insight and understanding into important PowerShell 2.0 features.

Conventions Used in This Book

Commands, scripts, and anything related to code are presented in a special monospace computer typeface. Bold indicates key terms being defined, and italic is used to indicate variables or for emphasis. Great care has been taken to be consistent in letter case, naming, and structure, with the goal of making command and script examples more readable. In addition, you might find instances in which commands or scripts haven’t been fully optimized. This lack of optimization is for your benefit, as it makes those code samples more intelligible and follows the practice of writing code for others to read.
Other standards used throughout this book are as follows:

**Black Code Boxes**

These code boxes contain commands that run in a PowerShell or Bash shell session.

**Gray Code Boxes**

These code boxes contain source code from scripts, configuration files, or other items that aren’t run directly in a shell session.

Please note that although PowerShell can display text in multiple colors, all script output from the examples is printed here in black and white. If you run one of the example scripts on your lab system, the text will be displayed in color.

**CAUTION**

Cautions alert you to actions that should be avoided.

**NOTE**

Notes give you additional background information about a topic being discussed.
CHAPTER 1
Introduction to Shells

Shells are a necessity when using key components of nearly all operating systems, because they make it possible to perform arbitrary actions such as traversing the file system, running commands, or using applications. As such, every computer user has interacted with a shell by typing commands at a prompt or by clicking an icon to start an application. Shells are an ever-present component of modern computing, frequently providing functionality that is not available anywhere else when working on a computer system.

In this chapter, you take a look at what a shell is and see the power that can be harnessed by interacting with a shell. To do this, you walk through some basic shell commands, and then build a shell script from those basic commands to see how they can become more powerful via scripting. Next, you take a brief tour of how shells have evolved over the past 35 years. Finally, you learn why PowerShell was created, why there was a need for PowerShell, what its inception means to scripters and system administrators, and what some of the differences between PowerShell 1.0 and PowerShell 2.0 CTP2 are.

What Is a Shell?

A shell is an interface that enables users to interact with the operating system. A shell isn’t considered an application because of its inescapable nature, but it’s the same as any other process that runs on a system. The difference between a shell and an application is that a shell’s purpose is to enable users to run other applications. In some operating systems (such as UNIX, Linux, and VMS), the shell is a command-line interface (CLI); in other operating systems (such as Windows and Mac OS X), the shell is a graphical user interface (GUI).
In addition, two types of systems in wide use are often neglected in discussions of shells: networking equipment and kiosks. Networking equipment usually has a GUI shell (mostly a Web interface on consumer-grade equipment) or a CLI shell (in commercial-grade equipment). Kiosks are a completely different animal; because many kiosks are built from applications running atop a more robust operating system, often kiosk interfaces aren’t shells. However, if the kiosk is built with an operating system that serves only to run the kiosk, the interface is accurately described as a shell. Unfortunately, kiosk interfaces continue to be referred to generically as shells because of the difficulty in explaining the difference to nontechnical users.

Both CLI and GUI shells have benefits and drawbacks. For example, most CLI shells allow powerful command chaining (using commands that feed their output into other commands for further processing, this is commonly referred to as the pipeline). GUI shells, however, require commands to be completely self-contained and generally do not provide a native method for directing their output into other commands. Furthermore, most GUI shells are easy to navigate, whereas CLI shells do not have an intuitive interface and require a preexisting knowledge of the system to successfully complete automation tasks. Your choice of shell depends on what you’re comfortable with and what’s best suited to perform the task at hand.

Even though GUI shells exist, the term “shell” is used almost exclusively to describe a command-line environment, not a task you perform with a GUI application, such as Windows Explorer. Likewise, shell scripting refers to collecting commands normally entered on the command line or into an executable file.

As you can see, historically there has been a distinction between graphical and nongraphical shells. An interesting development in PowerShell 2.0 CTP2 is the introduction of an alpha version of Graphical PowerShell, which provides a CLI and a script editor in the same window. Although this type of interface has been available for many years in IDE (Integrated Development Environment) editors for programming languages such as C, this alpha version of Graphical PowerShell gives a sense of the direction from the PowerShell team on where they see PowerShell going in the future—a fully featured CLI shell with the added benefits of a natively supported GUI interface.

Basic Shell Use
Many shell commands, such as listing the contents of the current working directory, are simple. However, shells can quickly become complex when more powerful results are required. The following example uses the Bash shell to list the contents of the current working directory.

```
$ ls
apache2 bin etc include lib libexec man sbin share var
```
However, often seeing just filenames isn’t enough and so a command-line argument needs to be passed to the command to get more details about the files.

NOTE

Please note that the Bash examples shown here are included only to demonstrate the capabilities of a command-line shell interface. Although it may be helpful, knowledge of the Bash shell will not be needed to understand any of the scripts or examples in this book.

The following command gives you more detailed information about each file using a command-line argument.

```
$ ls –l
```

```
total 8
```

```
drwxr-xr-x  13 root  admin   442 Sep 18 20:50 apache2
```

```
drwxrwxr-x  57 root  admin  1938 Sep 19 22:35 bin
```

```
drwxrwxr-x   5 root  admin   170 Sep 18 20:50 etc
```

```
drwxrwxr-x  30 root  admin  1020 Sep 19 22:30 include
```

```
drwxrwxr-x 102 root  admin  3468 Sep 19 22:30 lib
```

```
drwxrwxr-x  3 root  admin  102 Sep 18 20:11 libexec
```

```
lrwxr-xr-x   1 root  admin     9 Sep 18 20:12 man -> share/man
```

```
drwxrwxr-x  3 root  admin   102 Sep 18 20:11 sbin
```

```
drwxrwxr-x  3 root  admin   102 Sep 18 20:11 sbin
```

```
drwxrwxr-x   3 root  admin   102 Sep 18 20:11 var
```

Now you need to decide what to do with this information. As you can see, directories are interspersed with files, making it difficult to tell them apart. If you want to view only directories, you have to pare down the output by piping the ls command output into the grep command. In the following example, the output has been filtered to display only lines starting with the letter d, which signifies that the file is a directory.

```
$ ls –l | grep '^d'
```

```
drwxr-xr-x  13 root  admin   442 Sep 18 20:50 apache2
```

```
drwxrwxr-x  57 root  admin  1938 Sep 19 22:35 bin
```

```
drwxrwxr-x   5 root  admin   170 Sep 18 20:50 etc
```

```
drwxrwxr-x  30 root  admin  1020 Sep 19 22:30 include
```

```
drwxrwxr-x 102 root  admin  3468 Sep 19 22:30 lib
```

```
drwxrwxr-x  3 root  admin  102 Sep 18 20:11 libexec
```

```
drwxrwxr-x  3 root  admin  102 Sep 18 20:11 sbin
```

```
drwxrwxr-x 13 root  admin  442 Sep 19 22:35 share
```

```
drwxrwxr-x   3 root  admin  102 Jul 30 21:05 var
```
However, now that you have only directories listed, the other information such as date, permissions, size, and so on is superfluous because only the directory names are needed. So in this next example, you use the awk command to print only the last column of output shown in the previous example.

```
$ ls -l | grep '^d' | awk '{ print $NF }'
apache2
bin
etc
include
lib
libexec
sbin
share
var
```

The result is a simple list of directories in the current working directory. This command is fairly straightforward, but it’s not something you want to type every time you want to see a list of directories. Instead, we can create an alias or command shortcut for the command that we just executed.

```
$ alias lsd="ls -l | grep '^d' | awk '{ print $NF }"
```

Then, by using the lsd alias, you can get a list of directories in the current working directory without having to retype the command from the previous examples.

```
$ lsd
apache2
bin
etc
include
lib
libexec
sbin
share
var
```

As you can see, using a CLI shell offers the potential for serious power when you’re automating simple, repetitive tasks.
Basic Shell Scripts

Working in a shell typically consists of typing each command, interpreting the output, deciding how to put that data to work, and then combining the commands into a single, streamlined process. Anyone who has gone through dozens of files, manually adding a single line at the end of each one, will agree that scripting this type of manual process is a much more efficient approach than manually editing each file, and the potential for data entry errors is greatly reduced. In many ways, scripting makes as much sense as breathing.

You’ve seen how commands can be chained together in a pipeline to manipulate output from the preceding command, and how a command can be aliased to minimize typing. Command aliasing is the younger sibling of shell scripting and gives the command line some of the power of shell scripts. However, shell scripts can harness even more power than aliases.

Collecting single-line commands and pipelines into files for later execution is a powerful technique. Putting output into variables for further manipulation and reference later in the script takes the power to the next level. Wrapping any combination of commands into recursive loops and flow control constructs takes scripting to the same level of sophistication as programming.

Some may say that scripting isn’t programming, but this distinction is quickly becoming blurred with the growing variety and power of scripting languages these days. With this in mind, let’s try developing the one-line Bash command from the previous section into something more useful.

The lsd command alias from the previous example (referencing the Bash command ls -l | grep ‘^d’ | awk '{ print $NF }') produces a listing of each directory in the current working directory. Now, suppose you want to expand this functionality to show how much space each directory uses on the disk. The Bash utility that reports on disk usage does so on a specified directory’s entire contents or a directory’s overall disk usage in a summary. It also reports disk usage amounts in bytes by default. With all that in mind, if you want to know each directory’s disk usage as a freestanding entity, you need to get and display information for each directory, one by one. The following examples show what this process would look like as a script.

Notice the command line you worked on in the previous section. The for loop parses through the directory list the command returns, assigning each directory name to the DIR variable and executing the code between the do and done keywords.

```bash
#!/bin/bash

for DIR in $(ls -l | grep '^d' | awk '{ print $NF }'); do
    du -sk ${DIR}
done
```
Saving the previous code as a script file named directory.sh and then running the script in a Bash session produces the following output.

```
$ big_directory.sh
17988   apache2
5900    bin
72      etc
2652    include
82264   lib
0       libexec
0       sbin
35648   share
166768  var
```

Initially, this output doesn’t seem especially helpful. With a few additions, you can build something considerably more useful. In this example, we add an additional requirement to report the names of all directories using more than a certain amount of disk space. To achieve this requirement, modify the directory.sh script file as shown in this next example.

```
#!/bin/bash

PRINT_DIR_MIN=35000

for DIR in $(ls -l | grep '^d' | awk '{ print $NF }'); do
    DIR_SIZE=$(du -sk ${DIR} | cut -f 1)
    if [ ${DIR_SIZE} -ge ${PRINT_DIR_MIN} ]; then
        echo ${DIR}
    fi
done
```

One of the first things that you’ll notice about this version of directory.sh is that we have started adding variables. PRINT_DIR_MIN is a value that represents the minimum number of kilobytes a directory uses to meet the printing criteria. This value could change fairly regularly, so we want to keep it as easily editable as possible. Also, we could reuse this value elsewhere in the script so that we don’t have to change the amount in multiple places when the number of kilobytes changes.

You might be thinking the find command would be easier to use. However, although find is terrific for browsing through directory structures, it is too cumbersome for simply viewing the current directory, so the convoluted ls command is used instead. If we were looking for files in the hierarchy, the find command would be the most appropriate
choice. However, because we are simply looking for directories in the current directory, the \texttt{ls} command is the best tool for the job in this situation.

The following is an example of the output rendered by the script so far.

```
$ big_directory.sh
lib
share
var
```

This output can be used in a number of ways. For example, systems administrators might use this script to watch user directories for disk usage thresholds if they want to notify users when they have reached a certain level of disk space. For this purpose, knowing when a certain percentage of users reaches or crosses the threshold would be useful.

\textbf{NOTE}

Keep in mind that plenty of commercial products on the market notify administrators of overall disk thresholds being met, so although some money can be saved by writing a shell script to monitor overall disk use, it’s not necessary. The task of finding how many users have reached a certain use threshold is different, as it involves proactive measures to prevent disk use problems before they get out of control. The solution is notifying the administrator that certain users should be offloaded to new disks because of growth on the current disk. This approach isn’t foolproof, but is an easy way to add a layer of proactive monitoring to ensure that users don’t encounter problems when using their systems. Systems administrators could get creative and modify this script with command-line parameters to serve several functions, such as listing the top disk space users and indicating when a certain percentage of users have reached the disk threshold. That kind of complexity, however, is beyond the scope of this chapter.

In our next Bash scripting example, we modify the directory.sh script to display a message when a certain percentage of directories are a specified size.

```
#!/bin/bash

DIR_MIN_SIZE=35000
DIR_PERCENT_BIG_MAX=23

DIR_COUNTER=0
BIG(DIR_COUNTER)=0

for DIR in $(ls -l | grep '^d' | awk '{ print $NF }'); do
    DIR_COUNTER=$(expr ${DIR_COUNTER} + 1)
```
Now, the preceding example barely looks like what we started with. The variable name PRINT_DIR_MIN has been changed to DIR_MIN_SIZE because we’re not printing anything as a direct result of meeting the minimum size. The DIR_PERCENT_BIG_MAX variable has been added to indicate the maximum allowable percentage of directories at or above the minimum size. Also, two counters have been added: one (DIR_COUNTER) to count the directories and one (BIG_DIR_COUNTER) to count the directories exceeding the minimum size.

Inside the for loop, DIR_COUNTER is incremented, and the if statement in the for loop now simply increments BIG_DIR_COUNTER instead of printing the directory’s name. An if statement has been added after the for loop to do additional processing, figure out the percentage of directories exceeding the minimum size, and then print the message if necessary. With these changes, the script now produces the following output.

```
$ big_directory.sh
33 percent of the directories are larger than 35000 kilobytes.
```

The output shows that 33 percent of the directories are 35MB or more. By modifying the echo line in the script to feed a pipeline into a mail delivery command and tweaking the size and percentage thresholds for the environment, systems administrators can schedule this shell script to run at specified intervals and produce directory size reports easily. If administrators want to get fancy, they can make the size and percentage thresholds configurable via command-line parameters.

As you can see, even a basic shell script can be powerful. With a mere 22 lines of code, we have a useful shell script. Some quirks of the script might seem inconvenient (using the expr command for simple math can be tedious, for example), but every programming language has its strengths and weaknesses. As a rule, some tasks you need to do are convoluted to perform, no matter what language you’re using.
The moral of this story is that shell scripting, or scripting in general, can make life much easier. For example, say your company merges with another company. As part of that merger, you have to create 1,000 user accounts in Active Directory or another authentication system. Usually, a systems administrator grabs the list, sits down with a cup of coffee, and starts clicking or typing away. If an administrator manages to get a migration budget, he can hire an intern or consultants to do the work or purchase migration software. But why bother performing repetitive tasks or spending money that could be put to better use (such as a bigger salary)?

Instead, the answer should be to automate those iterative tasks by using scripting. Automation is the purpose of scripting. As a systems administrator, you should take advantage of scripting with CLI shells or command interpreters to gain access to the same functionality developers have when coding the systems you manage. However, scripting tools tend to be more open, flexible, and focused on the tasks that you as an IT professional need to perform, as opposed to development tools that provide a framework for building an entire application from a blank canvas.

A Shell History

The first shell in wide use was the Bourne shell, the standard user interface for the UNIX operating system, which is still required by UNIX systems during the startup sequence. This robust shell provided pipelines and conditional and recursive command execution. It was developed by C programmers for C programmers.

Oddly, however, despite being written by and for C programmers, the Bourne shell didn’t have a C-like coding style. This lack of a similarity to the C language drove the invention of the C shell, which introduced more C-like programming structures. While the C shell inventors were building a better mousetrap, they decided to add command-line editing and command aliasing (defining command shortcuts), which eased the bane of every UNIX user’s existence: typing. The less a UNIX user has to type to get results, the better.

Although most UNIX users liked the C shell, learning a completely new shell was a challenge for some. So the Korn shell was invented, which added a number of the C shell features to the Bourne shell. Because the Korn shell is a commercially licensed product, the open-source software movement needed a shell for Linux and FreeBSD. The collaborative result was the Bourne Again Shell, or Bash, invented by the Free Software Foundation.

Throughout the evolution of UNIX and the birth of Linux and FreeBSD, other operating systems were introduced along with their own shells. Digital Equipment Corporation (DEC) introduced Virtual Memory System (VMS) to compete with UNIX on its VAX systems. VMS had a shell called Digital Command Language (DCL) with a verbose syntax, unlike that of its UNIX counterparts. Also, unlike its UNIX counterparts, it wasn’t case sensitive nor did it provide pipelines.

Somewhere along the line, the PC was born. IBM took the PC to the business market, and Apple rebranded roughly the same hardware technology and focused on consumers. Microsoft made DOS run on the IBM PC, acting as both kernel and shell and including some features of other shells. (The pipeline syntax was inspired by UNIX shells.)
Following DOS was Windows, which quickly evolved from an application to an operating system. Windows introduced a GUI shell, which has become the basis for Microsoft shells ever since. Unfortunately, GUI shells are notoriously difficult to script, so Windows provided a DOSShell-like environment. It was improved with a new executable, cmd.exe instead of command.com, and a more robust set of command-line editing features. Regrettably, this change also meant that shell scripts in Windows had to be written in the DOSShell syntax for collecting and executing command groupings.

Over time, Microsoft realized its folly and decided systems administrators should have better ways to manage Windows systems. Windows Script Host (WSH) was introduced in Windows 98, providing a native scripting solution with access to the underpinnings of Windows. It was a library that enabled scripting languages to use Windows in a powerful and efficient manner. WSH is not its own language, however, so a WSH-compliant scripting language was required to take advantage of it, such as JScript, VBScript, Perl, Python, Kixstart, or Object REXX. Some of these languages are quite powerful in performing complex processing, so WSH seemed like a blessing to Windows systems administrators.

However, the rejoicing was short lived because there was no guarantee that the WSH-compliant scripting language you chose would be readily available or a viable option for everyone. The lack of a standard language and environment for writing scripts made it difficult for users and administrators to incorporate automation by using WSH. The only way to be sure the scripting language or WSH version would be compatible on the system being managed was to use a native scripting language, which meant using DOSShell and enduring the problems that accompanied it. In addition, WSH opened a large attack vector for malicious code to run on Windows systems. This vulnerability gave rise to a stream of viruses, worms, and other malicious programs that have wreaked havoc on computer systems, thanks to WSH’s focus on automation without user intervention.

The end result was that systems administrators viewed WSH as both a blessing and a curse. Although WSH presented a good object model and access to a number of automation interfaces, it wasn’t a shell. It required using Wscript.exe and Cscript.exe; scripts had to be written in a compatible scripting language, and its attack vulnerabilities posed a security challenge. Clearly, a different approach was needed for systems management; over time, Microsoft reached the same conclusion.

**Enter PowerShell**

Microsoft didn’t put a lot of effort into a CLI shell; instead, it concentrated on a GUI shell, which is more compatible with its GUI-based operating systems. (Mac OS X didn’t put any effort into a CLI shell, either; it used the Bash shell.) However, the resulting DOSShell had a variety of limitations, such as conditional and recursive programming structures not being well documented and heavy reliance on goto statements. These drawbacks hampered shell scripters for years, and they had to use other scripting languages or write compiled programs to solve common problems.

The introduction of WSH as a standard in the Windows operating system offered a robust alternative to DOSShell scripting. Unfortunately, WSH presented a number of challenges,
as discussed in the preceding section. Furthermore, WSH didn’t offer the CLI shell experience that UNIX and Linux administrators had enjoyed for years, which resulted in Windows administrators being made fun of by the other chaps for the lack of a CLI shell and its benefits.

Luckily, Jeffrey Snover (the architect of PowerShell) and others on the PowerShell team realized that Windows needed a strong, secure, and robust CLI shell for systems management. Enter PowerShell. PowerShell was designed as a shell with full access to the underpinnings of Windows via the .NET Framework, Component Object Model (COM) objects, and other methods. It also provided an execution environment that’s familiar, easy, and secure. PowerShell is aptly named, as it puts the power into the Windows shell. For users wanting to automate their Windows systems, the introduction of PowerShell was exciting because it combined the power of WSH with the familiarity of a traditional shell.

PowerShell provides a powerful native scripting language, so scripts can be ported to all Windows systems without worrying about whether a particular language interpreter is installed. You might have gone through the rigmarole of scripting a solution with WSH in Perl, Python, VBScript, JScript, or another language, only to find that the next system you worked on didn’t have that interpreter installed. At home, users can put whatever they want on their systems and maintain them however they see fit, but in a workplace, that option isn’t always viable. PowerShell solves that problem by removing the need for nonnative interpreters. It also solves the problem of wading through Web sites to find command-line equivalents for simple GUI shell operations and coding them into .cmd files. Last, PowerShell addresses the WSH security problem by providing a platform for secure Windows scripting. It focuses on security features such as script signing, lack of executable extensions, and execution policies (which are restricted by default).

For anyone who needs to automate administration tasks on a Windows system, PowerShell provides a much-needed injection of power. Its object-oriented nature boosts the power available to you, too. If you’re a Windows systems administrator or scripter, becoming a PowerShell expert is highly recommended.

PowerShell is not just a fluke or a side project at Microsoft. The PowerShell team succeeded at creating an amazing shell and winning support within Microsoft for its creation. For example, the Exchange product team adopted PowerShell as the backbone of the management interface in Exchange Server 2007. That was just the start. Other product groups at Microsoft, such as System Center Operations Manager 2007, System Center Data Protection Manager V2, and System Center Virtual Machine Manager, are being won over by what PowerShell can do for their products. In fact, PowerShell is the approach Microsoft has been seeking for a general management interface to Windows-based systems. Over time, PowerShell could replace current management interfaces, such as cmd.exe, WSH, CLI tools, and so on, and become integrated into the Windows operating system as its backbone management interface. With the introduction of PowerShell, Microsoft has addressed a need for CLI shells. The sky is the limit for what Windows systems administrators and scripters can achieve with it.
New Capabilities in PowerShell 2.0 CTP2

With the release of PowerShell 2.0 CTP2, the PowerShell team has expanded the capabilities of PowerShell 1.0 to include a number of key new features. Although PowerShell 2.0’s final feature set is likely to change from the CTP2 release, these features are central to PowerShell 2.0 and are expected to make it into the final release of the product.

The first major new feature of PowerShell 2.0 CTP2 is the addition of **PowerShell Remoting**. In a major step forward from the original release of PowerShell 1.0, PowerShell 2.0 CTP2 provides support for running cmdlets and scripts on a remote machine. The Windows Remote Management Service (WS-Man) is used to accomplish this, and a new cmdlet named **Invoke-Expression** is used to designate the target machine and the command to be executed. The following code example shows the general usage of the **Invoke-Expression** cmdlet to run the command `get-process powershell` on a remote computer named `XP1`.

```
PS C:\> invoke-expression -comp XP1 -command "get-process powershell"
```

<table>
<thead>
<tr>
<th>Handles</th>
<th>NPM(K)</th>
<th>PM(K)</th>
<th>WS(K)</th>
<th>VM(M)</th>
<th>CPU(s)</th>
<th>Id</th>
<th>ProcessName</th>
</tr>
</thead>
<tbody>
<tr>
<td>522</td>
<td>12</td>
<td>30652</td>
<td>29076</td>
<td>158</td>
<td>3.70</td>
<td>1168</td>
<td>powershell</td>
</tr>
</tbody>
</table>

Another new feature of PowerShell 2.0 CTP2 is the introduction of background jobs or PSJobs. A PSJob is simply a command or expression that executes asynchronously, freeing up the command prompt immediately for other tasks. A new series of Cmdlets related to PSJobs are included, which enable PSJobs to be started, stopped, paused, and listed. It also enables the results analyzed.

Also included in PowerShell 2.0 CTP2 is a new functionality called ScriptCmdlets. Previously, cmdlets had to be written in a .NET framework programming language such as C#, which made it a challenge for many scripters to create their own cmdlets. In this release of PowerShell, the ScriptCmdlets functionality enables scripters to write their own cmdlets with no more effort than writing a PowerShell function. While ScriptCmdlets are handled differently from compiled cmdlets and have certain limitations in this release of PowerShell (such as lack of support for help files), this functionality makes it far easier for scripters to extend PowerShell to address their specific requirements.

The last new feature of PowerShell 2.0 CTP2 that we discuss in this chapter is the introduction of Graphical PowerShell. Graphical PowerShell is currently in an early alpha version, but includes a number of powerful new capabilities that enhance the features of the basic PowerShell CLI shell. Graphical PowerShell provides an interface with both an
interactive shell pane and a multi-tabbed scripting pane, as well as the ability to launch multiple shell processes (also known as runspaces) from within Graphical PowerShell. All of these new features and more are discussed and demonstrated in more detail in subsequent chapters.

**Summary**

In summary, this chapter has served as an introduction to what a shell is, where shells came from, how to use a shell, and how to create a basic shell script. While learning these aspects about shells, you have also learned why scripting is so important to systems administrators. As you have come to discover, scripting enables systems administrators to automate repetitive tasks. In doing so, task automation enables systems administrators to perform their jobs more effectively, freeing them to perform more important business-enhancing tasks.

In addition to learning about shells, you have also been introduced to what PowerShell is and why PowerShell was needed. As explained, PowerShell is the replacement to WSH, which, although it was powerful, had a number of shortcomings (security and interoperability being the most noteworthy). PowerShell was also needed because Windows lacked a viable CLI that could be used to easily complete complex automation tasks. The end result for replacing WSH and improving on the Windows CLI, is PowerShell, which is built around the .NET Framework and brings a much needed injection of backbone to the world of Windows scripting and automation. Lastly, the key new features of PowerShell 2.0 CTP2 were reviewed at a high level, with detailed analysis of these new capabilities to be provided in subsequent chapters.
Symbols

* (asterisk), 131-134, 185-186
@ (at symbol), 137, 194
\ (backslash), 135
` (backtick), 143-144
[] (braces), 39, 170
[] (brackets), 52, 69, 154, 185
& (call operator), 91
^ (caret), 134, 193
, (comma), 151
/ (division) operator, 131
$ (dollar sign), 134, 137, 193
:: (double colon), 52, 138, 140
- (hyphen), 131, 154
% (modulus) operator, 131
? (question mark), 133-135, 185-186
= operator, 131
%= operator, 132
*= operator, 132
+= operator, 131
-= operator, 131
/= operator, 132
() (parentheses), 137
. (period), 91, 134, 138
| (pipe operator), 62
+ (plus sign), 131, 135
.. (range operator), 154
< redirection operator, 143
> redirection operator, 143
>> redirection operator, 143
; (semicolon), 152
_ (underscore) character, 39
$_ variable, 41
1 to 1 (interactive) remoting, 460-461
1 to many (fan-out) remoting, 461-465
2> redirection operator, 143
2>&1 redirection operator, 143
2>> redirection operator, 143

A

accelerators (type), 71-72
Access Control Entry (ACE), 236-237
accounts, disabling, 366
ACE (Access Control Entry), 236-237
Active Directory
ADSI (Active Directory Services Interfaces), 353-355
ChangeLocalAdminPassword.ps1 script. See ChangeLocalAdminPassword.ps1 script
managing with PowerShell
   administrator tips and guidelines, 359-360
   CTP2 improvements, 361
   DirectoryEntry class, 357
   DirectorySearcher class, 357-358
   DirectoryServices.ActiveDirectory
   namespace, 358-359
managing with Window Script Host (WSH), 355-356
objects
   ADSI methods, 363-364
   base object methods, 365
   binding, 361-363
   creating, 365
   deleting, 366
   example, 372-373
   modifying, 366
   moving, 366
   searching for, 367-372
overview, 353
ActiveX Data Objects (ADO), 355
AD. See Active Directory
Add method, 364
Add-ACE cmdlet, 236-237
Add-ConnectorToTier cmdlet, 454
Add-Content cmdlet, 207
Add-Member cmdlet, 67, 76
Add-Module cmdlet, 475
Add-Numbers function, 159
Add-PSSnapin cmdlet, 94, 327
Add-RemotelyManagedComputer cmdlet, 454
Add-RemotelyManagedDevice cmdlet, 454
Add-UserToUserRole cmdlet, 454
add/remove cmdlets (Operations Manager PowerShell interface), 454
addition (+) operator, 131
AddPerm function, 230
ADO (ActiveX Data Objects), 355
ADODB (ADO DB) provider, 355
ADSI (Active Directory Services Interfaces), 353-355, 363-364
ADSI OLE DB (ADODB) provider, 355
[ADSI] type accelerator, 357
[ADSI] type accelerator, 358
agent cmdlets (Operations Manager PowerShell interface), 432-436
agents, 434-436
alias cmdlets, 43-44
Alias property, 332
aliases, 10
   alias cmdlets, 43-44
   avoiding in scripts, 303
   listing supported aliases, 43
   overview, 42
   persistent aliases, 44
All Users host-specific profile, 87-88
All Users profile, 87
AllMatches parameter (Select-String cmdlet), 176
AllSigned execution policy, 101, 294
analyzing objects with Get-Member cmdlet, 56-59
   -and operator, 136
   appending content to files, 207, 210-211
   Approve-AgentPendingAction cmdlet, 455
   arguments, 29
   arithmetic expressions, 130
   arithmetic operators, 131
   array literals. See arrays
   array operators, 137
array subexpressions operator (@), 137, 151
arrays, 69, 329-330
   accessing elements of, 154
   array slicing, 155
   array subexpressions operator (@), 137, 151
   defining, 151-153
   definition of, 151
   multidimensional arrays, 151, 156-157
   negative index, 154
   one-dimensional arrays, 151
   operators, 151
   range operator (..), 154
assemblies, 54-56
assignment expressions, 130
assignment operators, 131-132
asterisk (*), 131-134, 185-186
at symbol (@), 137, 194
attributes. See specific attributes
authentication, 363
AuthenticationLevel property, 280-282
auto-completion feature, 30-32
automating file system management with ProvisionWebFolders.ps1 script. See ProvisionWebFolders.ps1 script
available cmdlets, listing with Get-Command cmdlet, 36-39
awk scripting language, 60
B
background jobs. See PSJobs
backslash (\), 135
backtick (`), 143-144
-band operator, 136
base object methods, 365
Bash shell, 8-15
BeginProcesses method, 323
BIG_DIR_COUNTER counter, 14
binding objects, 361-363
bitwise operators, 135-136
-bnot operator, 136
-bor operator, 136
Bourne Again Shell. See Bash shell
Bourne shell, 15
braces ({ }), 39, 170
brackets ([]), 52, 69, 185
Break keyword, 148
-bxor operator, 136
built-in variables, 40-42

C
C shell, 15
CA signed certificates, 109-111
cache, GAC (Global Assembly Cache), 56, 316
call operator (&), 91
caret (^), 134, 193
CAs (certificate authorities), 107
CategoryInfo property (ErrorRecord object), 146
CEC (Common Engineering Criteria), 268
certificate authorities (CAs), 107
Certificate Creation Tool (Makecert.exe), 101
Certificate.Format.ps1xml file, 75
certificate stores, 112
Certificate Trust List (CTL), 117
certificates
   CA signed certificates, 109-111
certificate stores, 112
   obtaining, 107-108
   PVK Digital Certificate Files Importer, 112
   self-signed certificates, 108-109
ChangeLocalAdminPassword.ps1 script, 373
   DataTable objects, 380
   DirectorySearcher class, 383-384
   Get-ADObject function, 381
   Get-CurrentDomain function, 381
   Get-ScriptHeader function, 380
   header, 377
   LibraryCrypto.ps1 library file, 376
   New-PromptYesNo function, 377-378
   New-RandomPassword function, 377-379
   $OnlineServers variable, 384-385
   Out-File cmdlet, 380
   $Password variable, 384-385
   Read-Host cmdlet, 382-383
   running, 374
   sequence of actions, 374-376
   Set-ChoiceMessage function, 377-378
ChangePassword method, 364
classes. See specific classes
Clear-Inherit cmdlet, 224, 233-234
Clear-SD cmdlet, 235
CLI (command line interface)
   aliases, 43-44
   cmdlets. See specific cmdlets
   command syntax, 28-30
   navigating, 30
   overview, 28
   Tab key auto-completion, 30-32
   variables, 39-42
CLI shells, 8
cmd command prompt, accessing PowerShell from, 27
Cmdlet class, 322
cmdlet keyword, 470
cmdlets. See specific cmdlets
code signing, 294
   CAs (certificate authorities), 107
   certificates. See certificates
definition of, 106
tab public code distribution, 115-117
   overview, 105-106
   process, 106-107
   public code distribution, 117
   signed code distribution, 115-117
   signing scripts, 112-113
   verifying digital signatures, 113-114
column index values (multidimensional arrays), 156
combining strings, 194
comma (,), 151
comma-separated value (CSV) files. See CSV files

How can we make this index more useful? Email us at indexes@samspublishing.com
command line interface. See CLI command shell (Operations Manager PowerShell interface), 426
commands. See specific commands comments, 300
CommitChanges method, 365
committing transactions, 245-247
Common Engineering Criteria (CEC), 268
common parameters for cmdlets, 33-34
Community Technology Preview (CTP), 457
comparison expressions, 130
comparison operators, 132-133, 190-191
Complete-PSTransaction cmdlet, 244
configuration files, digitally signing, 294
configuration information, putting at beginning of scripts, 299
Configure-WSMan.ps1 script, 459
Confirm parameter, 34, 305-306
Connect-Mailbox cmdlet, 396
Console, 30
constructors, 52
Contains method, 180
-contains operator, 133
Continue keyword, 148
conversion errors (types), 70
ConvertFrom-StringData cmdlet, 473-475
converting strings to lowercase/uppercase, 183-184
ConvertTo-Csv cmdlet, 215
ConvertTo-Html cmdlet, 64
Copy-Item cmdlet, 221
core cmdlets, 197-198
counters, 14
Create method, 364
CreateObject method, 356
CreateRunspace method, 486
CSV (comma-separated value) files, 214-216
CTL (Certificate Trust List), 117
CTP (Community Technology Preview), 457
Current User's host-specific profile, 88
Current User's profile, 88
current working directory
  listing contents of, 8-10
  retrieving, 198
  setting, 198
custom cmdlets, 320-323
custom output formats, 75-79
custom parameters, 327-329
custom snap-ins, 323-327
cut command, 60
\d regular expression characters, 135
DATA sections, 473-475
Database parameter (New-Mailbox cmdlet), 396
databases, 392-393
DataTable objects, 380
DCL (Digital Command Language), 15
Debug parameter (cmdlets), 33
debugging in PowerShell 2.0, 475-477
$DebugPreference variable, 42
default security, 100
defining types, 68-71
Delete method, 364
delimiters, 60
descriptive names, 303
designing scripts. See scripts
Detailed Description section (PowerShell help), 36, 337
developing scripts. See scripts
development life cycle model, 296-297
dialog boxes. See specific dialog boxes
Digital Command Language (DCL), 15
digital signatures. See code signing
dir command, 29
DIR_COUNTER counter, 14
DIR_MIN_SIZE variable, 14
DIR_PERCENT_BIG_MAX variable, 14
directories
  current working directory
    listing contents of, 8-10
    retrieving, 198
    setting, 198
  pushing onto list of locations, 199
  recalling from list of locations, 199-200
directory.sh script, 12-14
DirectoryEntry class, 66, 357
DirectorySearcher class, 56-59, 357-358, 367, 383-384
  Filter property, 367-368
  PageSize property, 371
  PropertiesToLoad property, 372
  search example, 372-373
  SearchRoot property, 367
SearchScope property, 371
SizeLimit property, 371
DirectoryServices.ActiveDirectory namespace, 358-359
Disable-Mailbox cmdlet, 396
Disable-NotificationSubscription cmdlet, 455
Disable-PsBreakpoint cmdlet, 476
Disable-Rule cmdlet, 455
disabling user accounts, 366
Dismount-Database cmdlet, 392
distribution, 115-117
division (/) operator, 131
DLLS (dynamic link libraries), 91
do/while loop, 164-165
dollar sign ($), 134, 137, 193
DOSShell, 16
dot sourcing, 91
DotNetTypes.Format.ps1xml file, 75
double colon (:), 138-140

drives
accessing, 82-85
adding, 200
definition of, 81
mounting, 85-86
removing, 86, 201
viewing, 81-82
DumpPerm function, 229-230
dynamic link libraries (DLLs), 91

E

EMC (Exchange Management Console), 388
EMS (Exchange Management Shell)
accessing, 388-389
databases management/reporting, 392-393
definition of, 388
help, 389-390
recipient management/reporting, 394-397
server management/reporting, 390-391
storage group management/reporting, 391-392
Enable-Mailbox cmdlet, 394-395
Enable-NotificationSubscription cmdlet, 455
Enable-PsBreakpoint cmdlet, 476
Enable-Rule cmdlet, 455
enable/disable cmdlets (Operations Manager PowerShell interface), 455
EndProcessing method, 323
EndsWith method, 180
enterprise code distribution, 117
-eq operator, 132, 190
equal operator, 190
error handling
error trapping, 148-150
$Error variable, 145
ErrorRecord object, 145-146
nonterminating errors, 145
terminating errors, 145
with throw keyword, 150
in transactions, 248-249
type conversion errors, 70
with ubiquitous parameters, 146-147
$Error variable, 41, 145
ErrorAction parameter, 34, 146-147
$ErrorActionPreference variable, 42
ErrorDetails property (ErrorRecord object), 146
ErrorRecord object, 145-146
errortraps1.ps1, 149
errortraps2.ps1, 149
ErrorVariable parameter, 34, 146-147
escape sequences, 143-144
ETS (Extended Type System)
Add-Member cmdlet, 67
arrays, 69
hash tables, 69
integers, 68
overview, 65-67
strings, 68
type accelerators, 71-72
type conversion errors, 70
type definitions, 68-71
type references, 69
types.ps1xml file, 68
Event ID 1221 messages, reporting on, 406-416
events, 52
example-cmdlet function, 160-163
Examples section (PowerShell help), 36
Exception property (ErrorRecord object), 146
Exchange Management Console (EMC), 388
Exchange Management Shell. See EMS
Exchange Server 2007
EMC (Exchange Management Console), 388
EMS (Exchange Management Shell)
accessing, 388-389

EndsWith method, 180
enterprise code distribution, 117
-eq operator, 132, 190
equal operator, 190
error handling
error trapping, 148-150
$Error variable, 145
ErrorRecord object, 145-146
nonterminating errors, 145
terminating errors, 145
with throw keyword, 150
in transactions, 248-249
type conversion errors, 70
with ubiquitous parameters, 146-147
$Error variable, 41, 145
ErrorAction parameter, 34, 146-147
$ErrorActionPreference variable, 42
ErrorDetails property (ErrorRecord object), 146
ErrorRecord object, 145-146
errortraps1.ps1, 149
errortraps2.ps1, 149
ErrorVariable parameter, 34, 146-147
escape sequences, 143-144
ETS (Extended Type System)
Add-Member cmdlet, 67
arrays, 69
hash tables, 69
integers, 68
overview, 65-67
strings, 68
type accelerators, 71-72
type conversion errors, 70
type definitions, 68-71
type references, 69
types.ps1xml file, 68
Event ID 1221 messages, reporting on, 406-416
events, 52
example-cmdlet function, 160-163
Examples section (PowerShell help), 36
Exception property (ErrorRecord object), 146
Exchange Management Console (EMC), 388
Exchange Management Shell. See EMS
Exchange Server 2007
EMC (Exchange Management Console), 388
EMS (Exchange Management Shell)
accessing, 388-389

How can we make this index more useful? Email us at indexes@samspublishing.com
database management/reporting, 392-393
definition of, 388
help, 389-390
recipient management/reporting, 394-397
server management/reporting, 390-391
storage group management/reporting, 391-392
GetDatabaseSizeReport.ps1 script, 397-406
GetEvent1221Info.ps1 script, 406-416
overview, 387
ProvisionExchangeUsers.ps1 script, 416-423
EvalQuery method, 270
execution policies
AllSigned, 101
best practices, 294
definition of, 100
RemoteSigned, 101-103
Restricted, 100
setting, 103-105
Unrestricted, 103, 118
explicit scope indicators, 89
explorer.exe, replacing
overview, 480-481
PSShell Secure Kiosk GPO, creating, 481
Windows Shell Replacement settings, configuring, 481-482
Export-Alias cmdlet, 44
Export-CliXml cmdlet, 213-214
Export-Csv cmdlet, 216
Export-DataTable function, 401, 406, 410, 416
Export-Mailbox cmdlet, 396
Export-ManagementPack cmdlet, 455
expressions, 129-130, 191-192
Extended Type System. See ETS
extensions, PowerShell Community Extensions (PSCX) package, 64-65

F

-f (format) operator, 195-196
fan-in remoting, 461
fan-out remoting, 461-465
fields, 52
file systems
automating with ProvisionWebFolders.ps1 script. See ProvisionWebFolders.ps1 script
core cmdlets, 197-198

F

F

F

for loop, 165
test for, 204-205
Format-List cmdlet, 74, 327
Format-Table cmdlet, 61, 74, 291, 406
Format-Wide cmdlet, 74
FormatNumber function, 270
formatting output, 73-79, 195-196
FullControl, granting to user’s Web folder, 224
FullyQualifiedErrorId property (ErrorRecord object), 146
function-parameters argument (functions), 157
function providers, 162
functions. See also specific functions
compared to filters, 163-164
defining, 157-163
definition of, 157
function providers, 162
multiline functions, 158

G

GAC (Global Assembly Cache), 56
gathering script requirements effectively, 297
-gc operator, 132
geocoding in MMC 3.0 (sample interface)
Get-Coordinates cmdlet, 344-347
Get-Coordinates MMC, 349-352
Get-Coordinates User Control, 347-349
Get-ACL cmdlet, 232
Get-ADObject cmdlet, 64
Get-ADObject function, 381, 418, 421
get-agent cmdlet, 432-434, 452
Get-AgentPendingAction cmdlet, 452
Get-Alert cmdlet, 452
Get-AlertDestination cmdlet, 452
Get-AlertHistory cmdlet, 452
Get-Alias cmdlet, 43-44
Get-AuthenticodeSignature cmdlet, 113-114
Get-ChildItem cmdlet, 112, 188
Get-ClientAccessServer cmdlet, 391
Get-Command cmdlet, 33, 36-39, 327
Get-Connector cmdlet, 452
Get-Content cmdlet, 84-85, 206-207
Get-Coordinates cmdlet, 344-347
Get-Coordinates MMC, 349-352
Get-Coordinates User Control, 347-349
Get-CurrentDomain function, 381, 418, 421
get-DefaultSetting cmdlet, 443-447, 452
Get-Diagnostic cmdlet, 452
Get-Discovery cmdlet, 453
Get-Event cmdlet, 453
Get-ExBlog cmdlet, 390
Get-ExCommand cmdlet, 390
Get-ExecutionPolicy cmdlet, 45, 104
Get-FailoverManagementServer cmdlet, 453
Get-GatewayManagementServer cmdlet, 453
Get-Help cmdlet
overview, 34-35, 390
supporting with XML files
Detailed Description section, 337
header, 335
Input Type section, 338
Name and Synopsis sections, 335-336
Notes section, 338-339
Output section, 340-341
Parameters section, 337-338
Related Links section, 339
Return Type section, 338
Syntax section, 336-337
Get-Help command, 30
Get-ItemProperty cmdlet, 241, 263
Get-Location cmdlet, 198
Get-Mailbox cmdlet, 396-397
Get-MailboxDatabase cmdlet, 399
Get-MailboxServer cmdlet, 391, 399, 404, 414
Get-MaintenanceWindow cmdlet, 453
get-MaintenanceWindow cmdlet, 449-451
Get-ManagementGroupConnection cmdlet, 453
Get-ManagementPack cmdlet, 453
get-ManagementServer cmdlet, 441-442, 453
Get-Member cmdlet, 52, 56-59, 277
Get method, 364
Get-Monitor cmdlet, 453
Get-MonitorHierarchy cmdlet, 453
Get-MonitoringClass cmdlet, 453
Get-MonitoringClassProperty cmdlet, 453
Get-MonitoringObject cmdlet, 453
Get-MonitoringObjectGroup cmdlet, 453
Get-MonitoringObjectPath cmdlet, 453
Get-MonitoringObjectProperty cmdlet, 453
Get-MyMessage cmdlet, 471
Get-NotificationAction cmdlet, 453
Get-NotificationEndpoint cmdlet, 453
Get-NotificationRecipient cmdlet, 453
Get-NotificationSubscription cmdlet, 453
Get-OperationsManagerCommand cmdlet, 431, 453

How can we make this index more useful? Email us at indexes@samspublishing.com
Get-Override cmdlet, 453
Get-PerformanceCounter cmdlet, 453
Get-PerformanceCounterValue cmdlet, 453
Get-PrimaryManagementServer cmdlet, 453
Get-Process cmdlet, 62
Get-Process command, 29
Get-PsBreakpoint cmdlet, 476
Get-PsCallStack cmdlet, 476
Get-PsCommand cmdlet, 390
Get-PsJob cmdlet, 467
Get-PsProvider cmdlet, 81
Get-PsSnapin cmdlet, 92, 95, 326
Get-Recovery cmdlet, 453
Get-RegValue cmdlet, 251-253
Get-RelationshipClass cmdlet, 453
Get-RelationshipObject cmdlet, 453
Get-RemoteEventLog cmdlet, 409
Get-RemoteEventLog function, 410, 415
Get-RemotelyManagedComputer cmdlet, 453
Get-RemotelyManagedDevice cmdlet, 453
Get-ResultantCategoryOverride cmdlet, 453
Get-ResultantRuleOverride cmdlet, 453
Get-ResultantUnitMonitorOverride cmdlet, 453
Get-RootManagementServer cmdlet, 453
Get-Rule cmdlet, 453
Get-RunAsAccount cmdlet, 454
Get-Runspace cmdlet, 264
Get-ScriptHeader function, 380, 413, 420
Get-Service cmdlet, 50
Get-State cmdlet, 454
Get-StorageGroup cmdlet, 391
gc_Cmdlet, 437-438, 454
gc_CmdletResult cmdlet, 439-440, 454
Get-Tier cmdlet, 454
Get-Tip cmdlet, 390
Get-TransportServer cmdlet, 391
Get-UserRole cmdlet, 454
Get-WmiObject cmdlet, 271-273, 276, 399
gc_Cmdlet (Operations Manager PowerShell interface), 452-454
GetDatabaseSizeReport.ps1 script, 397-406
GetEvent1221Info.ps1 script, 406-416
GetEx method, 364
GetHelloWorldPSSnapIn, 323, 326
GetInfo method, 364
GetInfoEx method, 364
GetObject method, 355
Global Assembly Cache (GAC), 56, 316
global scope, 88-89
GPOs (Group Policy Objects)
configuring WinRM and WinRS with, 122-125
creating, 104-105
PSShell Secure Kiosk GPO, creating, 481
Graphical PowerShell (PowerShell 2.0), 18, 468-469
Group Policy Administrative Template, 104
Group Policy Objects. See GPOs
grouping operators, 137
-gt operator, 132
GUI shells, 8
H
hard-coding configuration information, avoiding, 300
hashes
hash tables, 69
one-way hashes, 106
help
available cmdlets, listing with Get-Command cmdlet, 36-39
in EMS (Exchange Management Shell), 389-390
help topics, displaying with Get-Help cmdlet, 34-35
in Operations Manager PowerShell interface, 431
PowerShell help sections, 36
Windows PowerShell home page, 21
Help.Format.ps1xml file, 75
HelpMessage property, 331
history of shells, 15-16
hives (Registry), 239, 242-243
HKCU: hive, 239
HKLM: hive, 239
hosting applications, 87
hosts, 87
hyphen (-), 154
I
IADs interface, 354
IADsContainer interface, 354
if/else loop, 167
IIsWebService class, 276-277
ImpersonationLevel property, 280-282
Import-Alias cmdlet, 44
Import-CliXml cmdlet, 213-214
Import-Csv cmdlet, 215, 221, 418
Import-LocalizedData cmdlet, 473
Import-Mailbox cmdlet, 396
Input Type section (PowerShell help), 36, 338
input validation
  Get-Help cmdlet. See Get-Help cmdlet
  required parameters, 302-303
  ValidateCount attribute, 334-335
  ValidateLength attribute, 332
  ValidatePattern attribute, 333
  ValidateRange attribute, 333
  ValidateSet attribute, 334
Insert method, 181
install-agent cmdlet, 434-435, 455
Install-AgentByName cmdlet, 455
Install-ManagementPack cmdlet, 455
install/uninstall cmdlets (Operations Manager PowerShell interface), 455
InstallUtil.exe, 93-94, 96
instance members, 52-53
instructions, including with scripts, 301-302
integers, 68
interactive remoting, 460-461
interfaces, 354
internationalization (scripts), 472-473
InvocationInfo property (ErrorRecord object), 146
Invoke method, 365
Invoke-Command cmdlet, 264, 462-464
Invoke-Expression cmdlet, 18
Invoke-WMI Method cmdlet, 283-284
InvokeGet method, 365
InvokeSet method, 360, 365

J-K-L

jobs
  background jobs (PowerShell 2.0), 466-467
  PSJobs, 18
join operator, 194
Join-Path cmdlet, 221
joining strings, 194

keys (Registry)
  creating, 253-254
  removing, 256-257
keywords. See specific keywords

  kiosks, 8
Korn shell, 15

labels, 167
-le operator, 132
Length property (System.String class), 185
length of strings, determining, 185
library files
  creating, 97
  definition of, 91, 316
  LibraryCrypto.ps1 library file, 376
  registering, 93-95
  removing, 96-97
  snapins, adding to console, 94-95
  snapins, removing from console, 96
LibraryCrypto.ps1 library file, 376
LibraryRegistry.ps1 script
  Get-RegValue function, 251-253
  overview, 249-251
  Remove-RegKey function, 256-257
  sample application, 257-262
  Set-RegKey function, 253-254
  Set-RegValue function, 254-255
Like operator, 133, 190
Load method, 55, 485
LoadFile method, 56, 485
LoadFrom method, 56, 485
LoadWith PartialName method, 55, 485
local scope, 89
locating tasks, 437-438
logical operators, 135-136, 368
loops
  definition of, 164
  do/while loop, 164-165
  for loop, 165
  foreach loop, 166
  if/else loop, 167
  while loop, 164
lowercase, converting strings to, 183
Is command, 8-9
-lt operator, 132

M

mailboxes
  creating, 394-396
  GetDatabaseSizeReport.ps1 script, 397-406

How can we make this index more useful? Email us at indexes@samspublishing.com
GetEvent1221Info.ps1 script, 406-416
managing, 396-397
ProvisionExchangeUsers.ps1 script, 416-423
reporting on, 396-397
maintenance mode cmdlets (Operations Manager PowerShell interface)
        get-MaintenanceWindow cmdlet, 449-451
        maintenance window reason codes, 448
new-MaintenanceWindow cmdlet, 448-449
set-MaintenanceWindow cmdlet, 451-452
maintenance windows, 448-452
Makecert.exe, 101
Managed Object Format (MOF) file, 278
management interface
custom cmdlets, 320-323
        custom snap-ins, creating, 323-327
        Get-Help cmdlet. See Get-Help cmdlet
input validation, 332-335
overview, 315
parameters, 327-332
PowerShell SDK, 316-319
runspaces, 341-344
scenario: geocoding in MMC 3.0
        Get-Coordinates cmdlet, 344-347
        Get-Coordinates MMC, 349-352
        Get-Coordinates User Control, 347-349
management servers
management server cmdlets (Operations Manager PowerShell interface)
default setting paths, 445-447
        get-DefaultSetting, 443-444
        get-ManagementServer, 441-442
set-DefaultSetting, 444-445
        set-ManagementServer, 442-443
returning, 441-442
returning settings for, 443-444
        setting default settings for, 444-445
specifying, 442-443
Mandatory property, 331
many to 1 (fan-in) remoting, 461
Match operator, 190
-match operator, 134
$MaximumAliasCount variable, 42
$MaximumDriveCount variable, 42
$MaximumErrorCount variable, 42
$MaximumFunctionCount variable, 42
$MaximumHistoryCount variable, 42
$MaximumVariableCount variable, 42
method operators, 138-140
methods. See specific methods
Microsoft Technet, 390
Microsoft Visual Studio Express Edition, downloading, 316
MMC, geocoding in MMC 3.0 (sample interface)
        Get-Coordinates cmdlet, 344-347
        Get-Coordinates MMC, 349-352
        Get-Coordinates User Control, 347-349
modules (PowerShell 2.0), 475
modulus (%) operator, 131
MOF (Managed Object Format) files, 278
moniker strings, 269
MonitorMSVS.ps1 script, 285-291
Mount-Database cmdlet, 392
mounting drives, 85-86
Move-DatabasePath cmdlet, 392
Move-Item cmdlet, 202-203, 206
Move-Mailbox cmdlet, 396-397
Move-StorageGroupPath cmdlet, 391-392
MoveHere method, 364
MoveTo method, 365
multidimensional arrays, 151, 156-157
multiline functions, 158
multiplication (*) operator, 131
multiplying strings, 178
N
{n} regular expression characters, 135
{n,} regular expression characters, 135
{n,m} regular expression characters, 135
Name parameter (New-Mailbox cmdlet), 396
Name section (PowerShell help), 36, 335
namespaces
definition of, 316
        DirectoryServices.ActiveDirectory, 358-359
        Root, 275
        WMI providers, 275
naming conventions
        CLI (command line interface), 39
cmdlets, 32
custom cmdlets, 320
        scripting standards, 309-310
-ne operator, 132, 190
negative index, 154
.NET Framework
persistent aliases, 44
Ping class, 53
pipe operator (|), 62
pipeline
  accepting input from, 158
  one-liners, 63-65
  overview, 59-63
PKI (public key infrastructure), 107-108
plus sign (+), 131, 135
policies
  execution policies, 100-105, 118, 294
  GPOs (Group Policy Objects), 104-105, 122-125
Pop-Location cmdlet, 199-200
Position property, 330
PowerShell 2.0
  background jobs, 466-467
  DATA sections, 473-475
  Graphical PowerShell, 468-469
  modules, 475
  Out-GridView cmdlet, 471
  overview, 457
  remoting. See remoting
  script cmdlets, 469-471
  script debugging, 475-477
  script internationalization, 472-473
PowerShell Community Extensions (PSCX) package, 64-65
PowerShell development, 16-17
PowerShell Remoting, 18, 295
PowerShell SDK. See SDK
PowerShell WMI Explorer, 278
PowerShellCore.format.ps1xml file, 75
PowerShellExecutionPolicy.adm, 104
PowerShellTrace.format.ps1xml file, 75
principle of least privileges, 294-295
PRINT_DIR_MIN variable, 12
private scope, 90-91
privileges, principle of least privileges, 294-295
processing XML files, 213
ProcessRecord method, 323
$Processes variable, 88-91
$ProcessReturnPreference variable, 42
$ProgressPreference variable, 42
production environments, dangers of, 297-298
professionalism in script development, 298-299
profiles, 87-88
projects, setting up, 321-322

\P regular expression characters, 135
packages, PowerShell Community Extensions (PSCX) package, 64-65
PageSize property (DirectorySearcher class), 371
parameters. See specific parameters
Parameters section (PowerShell help), 36, 337-338
parentheses, 137
parsing text, 59
Password parameter (New-Mailbox cmdlet), 396
$Password variable, 384-385
pattern matching operators, 133
percent sign (%), 131
period (.), 91, 134, 138
permissions
  folder permissions, 229-234
  managing with PowerShell cmdlets
    Add-Ace, 236-237
    Clear-Inherit, 233-234
    Clear-SD, 235
  overview, 231-233
  Remove-Ace, 237
  Set-Owner, 234-235
  managing in WSH (Windows Script Host) with
    SubInACL, 227-231
  overview, 227
properties. See also specific properties
adding to PSObject, 75-76
definition of, 52, 316
PropertiesToLoad property (DirectorySearcher
class), 372
property operators, 138
providers, 278, 354
core cmdlets, 79-81
drives
accessing, 82-85
definition of, 81
mounting, 85-86
removing, 86
viewing, 81-82
viewing, 81
WMI providers, 275
ProvisionExchangeUsers.ps1 script, 416-423
ProvisionWebFolders.ps1 script, 216
Clear-Inherit cmdlet, 224
Copy-Item cmdlet, 221
FullControl, granting, 224
functions, 218-220
header, 217-218
Import-Csv cmdlet, 221
Join-Path cmdlet, 221
Set-Owner cmdlet, 222
Test-Path cmdlet, 220-221
ps command, 60
PSAdapted view, 66
PSBase view, 66
PSCX (PowerShell Community Extensions)
package, 64-65
pseudocode, 297
.ps1 extension, 45
.ps1xml formatting files, 74-75
PSExtended view, 66
$PSHOME variable, 74
PSJobs, 18
PSObject, 61, 65-66, 75-76
PShell Kiosk
PShell.exe, 482-484
PShell.ps1 script, 479, 484-488
shell replacement, 480-482
PShell.exe, 482-484
PShell.ps1 script, 479, 484-488
PSSnapIn class, 323
PSTypeNames view, 66
public code distribution, 117
public key infrastructure (PKI), 107-108
Push-Location cmdlet, 199
Push-Runspace cmdlet, 461
pushing directories onto list of locations, 199
Put method, 364
PutEx method, 364
PVK Digital Certificate Files Importer, 112
Q-R
quantifiers, 192
question mark (?), 133-135, 185-186
QuickRef cmdlet, 390
range operator (..), 154
Read-Host cmdlet, 382-383, 418, 422
Read-only LDAP, 358
reading
CSV files, 215
files, 206-207
Registry values, 241
recalling directories from list of locations, 199-200
Receive-PsJob cmdlet, 467
recipients
managing in EMS (Exchange Management
Shell), 394-397
reporting on, 394-397
redirection operators, 142-143
references, 69
reflection, 56-59
[RegEx] type accelerator, 193-194
registering library files, 93-95
Registry
2.0 CTP features, 262-265
core cmdlets, 240
hives, 239, 242-243
keys, 253-257
LibraryRegistry.ps1 script
Get-RegValue function, 251-253
overview, 249-251
Remove-RegKey function, 256-257
sample application, 257-262
Set-RegKey function, 253-254
Set-RegValue function, 254-255
overview, 239
transactions
   cmdlets, 244-245
   committing, 245-247
   how they work, 245
   overview, 243-244
   starting, 245-247
   transactions with errors, 248-249
   undoing, 247-248
   values, 241-242, 251-255
Registry.format.ps1xml file, 75
regular expressions, 134-135, 191-192
Reject-AgentPendingAction cmdlet, 456
Related Links section (PowerShell help), 36, 339
releases of PowerShell, 22-23
reliability, 310-311
remote management. See WinRM (Windows Remote Management)
RemoteSigned execution policy, 101-103, 294
RemoteSigned policy, 45
remoting
   centrally managing, 295
   fan-in remoting, 461
   fan-out remoting, 461-465
   interactive remoting, 460-461
   overview, 458-460
   Universal Code Execution Model, 460
   Windows Vista and Windows Server 2008 machines, 465-466
   Windows XP SP2 or greater and Windows Server 2003 machines, 465
Remove method, 181-182, 364
Remove-ACE cmdlet, 237
Remove-ConnectorFromTier cmdlet, 454
Remove-DisabledMonitoringObject cmdlet, 454
Remove-Item cmdlet, 202, 205
Remove-ItemProperty cmdlet, 242
Remove-Mailbox cmdlet, 396
Remove-MailboxDatabase cmdlet, 392
Remove-ManagementGroupConnection cmdlet, 454
Remove-PsBreakpoint cmdlet, 476
Remove-PSDrive cmdlet, 86, 201
Remove-PSSnapin cmdlet, 96
Remove-RegKey cmdlet, 256-257
Remove-RemotelyManagedComputer cmdlet, 454
Remove-RemotelyManagedDevice cmdlet, 454
Remove-Runspace cmdlet, 265
Remove-StorageGroup cmdlet, 391
Remove-Tier cmdlet, 454
Remove-WMIObject cmdlet, 284-285
RemovePerm function, 231
Rename-Item cmdlet, 203-206
Rename method, 365
Replace method, 182
replacing
   operators, 134, 192-193
   strings, 182
   Windows Explorer shell (explorer.exe), 480-482
reporting
   on databases, 392-393
   on recipients, 394-397
   on servers, 390-391
   on storage groups, 391-392
repository (WMI), 275-278
Resolve-Alert cmdlet, 456
Restore-Mailbox cmdlet, 396
Restricted execution policy, 100
results of tasks, checking, 439-440
retrieving Registry values, 251-253
Return keyword, 148
Return Type section (PowerShell help), 36, 338
reusability of scripts, 303
rights, principle of least privileges, 294-295
Root namespace, 275
row index values (multidimensional arrays), 156
Run dialog box, accessing PowerShell from, 26
running scripts, 294-295, 374
RunspaceInvoke object, 486
runspaces, 264, 341-344, 458
   creating, 264, 486
   definition of, 486
   removing, 264-265
   running commands with, 263-264
S
\S regular expression characters, 135
SCOM (System Center Operations Manager), 425
scopes, 88-91, 150
script cmdlets (PowerShell 2.0), 469-471
script scope, 89-90
scriptblocks, 170-172
ScriptCmdlets, 18
Scriptomatic, 275
scripts. See also specific scripts
   benefits, 15
   building with scriptblocks, 170-172
controlling script flow with loops. See loops
creating, 45-47
debugging in PowerShell 2.0, 475-477
decision making in, 167-169
definition of, 45
design
comments, 300
configuration information at beginning of
scripts, 299
descriptive names, 303
hard-coding configuration information,
avoiding, 300
instructions, 301-302
reusability, 303
status information for script users, 304
validity checks on required parameters,
302-303
variables, 301
WhatIf and Confirm parameters, 305-306
development
dangers of production environment,
297-298
development life cycle model, 296-297
gathering script requirements effectively,
297
professionalism, 298-299
pseudocode, 297
testing, 298
treating scripting projects as actual
projects, 295
digitally signing, 294
dot sourcing, 91
internationalization, 472-473
in Operations Manager PowerShell
interface, 432
overview, 11
.ps1 extension, 45
RemoteSigned policy, 45
running, 294-295
script cmdlets (PowerShell 2.0), 469-471
scripting standards, 306-307, 309-311
signing, 112-113
WSH (Windows Script Host), 227-231
SDDL (Security Descriptor Definition
Language), 235
SDK, 316-319
searching
file contents, 207-208
for objects (DirectorySearcher class)
Set-Owner cmdlet, 222, 234-235
Set-ProxyAgent cmdlet, 454
Set-RegKey cmdlet, 253-254
Set-RegValue cmdlet, 254-255
Set-StorageGroup cmdlet, 391
Set-TransportServer cmdlet, 391
Set-WMIInstance cmdlet, 282-283
SetInfo method, 364
SetOwner function, 229
SetPassword method, 364
shell replacement (PSShell Kiosk), 480-482
shell scripts. See scripts
shells
  Bash shell, 13-14
  basic shell use, 8-10
  benefits and drawbacks, 8
  CLI shells, 8
  definition of, 7
  graphical versus nongraphical, 8
  GUI shells, 8
  history of, 15-16
  overview, 7
Shfusion.dll, 56
shortcuts (type), 71-72
ShouldProcess method, 34
$ShouldProcessPreference variable, 42
signing code. See code signing
silent installation, 25-26
SizeLimit property (DirectorySearcher class), 371
slash (/), 131
slicing arrays, 155
snapins
  adding to console, 94-95
  custom snap-ins, creating, 323-327
  removing from console, 96
Snover, Jeffrey, 17, 387
Software Development Kit. See SDK
Software Publishing Certificate (SPC) file, 112
SPC (Software Publishing Certificate) file, 112
Split method, 182
Split operator, 195
splitting strings, 182, 195
square brackets ([]), 52, 69, 154, 185
Start-Discovery cmdlet, 456
Start menu, accessing PowerShell from, 26
Start-PsJob cmdlet, 466
Start-PSTransaction cmdlet, 244
start-Task cmdlet, 438-439, 456
starting
  PowerShell, 26-27
  tasks, 438-439
  transactions, 245-247
StartsWith method, 182
statements. See specific statements
static member accessor (::), 52, 138-140
static members, 52
status information, providing for for script users, 304
Step-Into cmdlet, 476
Step-Out cmdlet, 476
Step-Over cmdlet, 477
Stop-PsJob cmdlet, 467
storage groups
  managing in EMS (Exchange Management Shell), 391-392
  reporting on, 391-392
[String] type accelerator, 174-175
strings, 68
  adding, 177
  binding strings, 362
  combining, 194
  comparison operators, 190-191
  converting to lowercase, 183
  converting to uppercase, 184
  creating, 173-174
  definition of, 173
  determining length of, 185
  extracting from current object, 183-184
  extracting strings from, 183
  Format operator, 195-196
  format strings, 141-142
  formatting, 195-196
  inserting strings into, 181
  join operator, 194
  moniker strings, 269
  multiplying, 178
  overview, 173
  querying for substrings, 180
  [RegEx] type accelerator, 193-194
  regular expressions, 191-192
  removing strings from, 181-182
  replace operators, 192-193
  replacing strings in, 182
Select-String cmdlet, 175-177
Split operator, 195
splitting, 182, 195
[String] type accelerator, 174-175
System.String class, 174, 178-180
Contains method, 180
EndsWith method, 180
Insert method, 181
Length property, 185
Remove method, 181-182
Replace method, 182
Split method, 182
StartsWith method, 182
SubString method, 183
ToLower method, 183
ToString method, 183-184
Trim method, 184
trimming, 184
wildcards, 185-189
subexpressions grouping operator ($), 137
SubInACL utility
AddPerm function, 230
DumpPerm function, 229-230
RemovePerm function, 231
SetOwner function, 229
syntax, 227-228
SubString method, 183
subtraction (-) operator, 131
switch statement, 168-169
Synopsis section (PowerShell help), 36, 335
Syntax section (PowerShell help), 36, 336-337
System Center Operations Manager (SCOM), 425
System.String class, 174, 178-180
Contains method, 18
EndsWith method, 180
Insert method, 181
Length property, 185
Remove method, 181-182
Replace method, 182
Split method, 182
StartsWith method, 182
SubString method, 183
ToLower method, 183
ToString method, 183-184
Trim method, 184
trimming strings, 507
T
Tab key completion, 138
TargetObject property (ErrorRecord object), 146
tasks
checking results of, 439-440
locating, 437-438
starting, 438-439
task cmdlets (Operations Manager PowerShell interface)
get-Task, 437-438
get-TaskResult, 439-440
start-Task, 438-439
templates, Group Policy Administrative Template, 104
terminating errors, 145
Terminating Errors section (PowerShell help), 36
test-Item cmdlet
test-Path cmdlet, 418, 422
in ProvisionWebFolders.ps1 script, 220-221
testing for files, 208
testing for folders, 204-205
testing
for files, 208
for folders, 204-205
scripts, 298
text, parsing, 59
throw keyword, 150
throwing errors, 150
ToLower method, 183
ToString method, 183-184
ToUpper method, 184
transactions (Registry)
cmdlets, 244-245
committing, 245-247
how they work, 245
overview, 243-244
starting, 245-247
transactions with errors, 248-249
undoing, 247-248
trapping errors
errottraps1.ps1 example, 149
errottraps2.ps1 example, 149
syntax, 148
trap scopes, 150
Trim method, 184
trimming strings, 184

How can we make this index more useful? Email us at indexes@samspublishing.com
type accelerators
[ADSI], 357
[ADSIISearcher], 358
[WMIClass] type accelerator, 273-274
[WMISearcher] type accelerator, 274
[WMI] type accelerator, 273

types
arrays, 69
conversion errors, 70
defining, 68-71
definition of, 53
ETS (Extended Type System)
    Add-Member cmdlet, 67
    overview, 65-67
types.ps1xml file, 68
hash tables, 69
integers, 68
references, 69
string. See strings
type accelerators, 71-72
    [ADSIISearcher], 358
    [ADSI], 357
    [RegEx] type accelerator, 193-194
    [WMIClass] type accelerator, 273-274
    [WMISearcher] type accelerator, 274
    [WMI] type accelerator, 273
type operators, 136
types.ps1xml file, 68

U-V
ubiquitous parameters, 146-147
unary operators, 136
underscore (_ ) character, 39
Undo-PSTransaction cmdlet, 244, 248
undoing transactions, 247-248
uninstall-agent cmdlet, 435-436, 455
Uninstall-ManagementPack cmdlet, 455
uninstalling
    agents, 435-436
    Windows PowerShell 1.0, 23-24
Universal Code Execution Model, 460
Unrestricted execution policy, 103, 118, 294
upercase, converting strings to, 184
Use-PSTransaction cmdlet, 245
user accounts, disabling, 366

UserPrincipalName parameter (New-Mailbox
    cmdlet), 396
users, providing status information for, 304
useTransaction parameter (New-Item cmdlet), 247

ValidateCount attribute, 334-335
ValidateLength attribute, 332
ValidatePattern attribute, 333
ValidateRange attribute, 333
ValidateSet attribute, 334
validating input. See input validation
variables. See also specific variables
    best practices, 301
    built-in variables, 40-42
    definition of, 39
    naming, 39
VerbNounPSSnapln naming convention, 323
Verboser parameter (cmdlets), 33
$VerbosePreference variable, 42
verifying digital signatures, 113-114
versions of PowerShell, 22-23
views, 66
Virtual Memory System (VMS), 15
$VirtualMachines variable, 290
Visual Studio Express Edition, downloading, 316
VMS (Virtual Memory System), 15

W
\W regular expression characters, 135
/w parameter (dir command), 29
Wait-PSJob cmdlet, 467
wbemtest.exe, 275
websites, Windows PowerShell home page, 21
WhatIf parameter, 34, 305-306
Where-Object cmdlet, 63
while loop, 164
whitespace delimiters, 60
wildcards, 133, 185-189, 369
Window Script Host (WSH), managing Active
    Directory with, 355-356
Windows Explorer shell (explorer.exe), replacing
    overview, 480-481
PSShell Secure Kiosk GPO, creating, 481
Windows Shell Replacement settings,
    configuring, 481-482
Windows Management Instrumentation Query Language (WQL), 278-279
Windows Management Instrumentation. See WMI
Windows permissions. See permissions
Windows PowerShell home page, 21
Windows Remote Management. See WinRM
Windows Remote Shell (WinRS), 121-125, 295
Windows Script Host. See WSH
Windows Server 2003 remoting settings, 465
Windows Server 2008
   enabling PowerShell 1.0 on, 22-23
   remoting settings, 465-466
Windows Shell Replacement settings, configuring, 481-482
Windows Vista remoting settings, 465-466
Windows XP SP2 remoting settings, 465
WinRM (Windows Remote Management)
   configuring, 119-120
   GPO settings, 122-125
   installing, 24
   overview, 119
   WinRS (Windows Remote Shell), 120-122
WinRS (Windows Remote Shell), 121-125, 295
WMI (Windows Management Instrumentation)
   advantages of, 267-268
   challenges, 268
   definition of, 267
   MonitorMSVS.ps1 script, 285-291
   objects, connecting to, 269
   in PowerShell
      AuthenticationLevel property, 280-282
      Get-WmiObject cmdlet, 271-273
      ImpersonationLevel property, 280-282
      Invoke-WMIObject cmdlet, 283-284
      Remove-WMIObject cmdlet, 284-285
      Set-WMIInstance cmdlet, 282-283
      [WMI] [WMI] [WMI] type accelerator, 273-274
      [WMI][type accelerator, 274
      [WMISearcher] type accelerator, 274
      [WMI] type accelerator, 273
   PowerShell WMI Explorer, 278
   providers and namespaces, 275, 278
   repository, exploring, 275-278
   Windows Management Instrumentation Query Language (WQL), 278-279
   in WSH, 269-270
WMI Query Language (WQL), 270

[WMI] type accelerator, 273
[WMIClass] type accelerator, 273-274
[WMISearcher] type accelerator, 274
WQL (Windows Management Instrumentation Query Language), 278-279
WQL (WMI Query Language), 270
writing CSV files, 216
WSH (Windows Script Host), 16
   managing Active Directory with, 355-356
   permissions, managing with SubInACL, 227-231
   WMI (Windows Management Instrumentation) in, 269-270
WSMAN (Windows Remote Management), 295, 459
WSMan settings, configuring, 27-28

X-Y-Z

XML files
   appending, 210-211
   creating, 209-210
   deleting from, 212
   Export-CliXml cmdlet, 213-214
   Import-CliXml cmdlet, 213-214
   loading, 212-213
   modifying, 211-212
   overview, 208
   processing, 213
   supporting custom cmdlets with
      Detailed Description section, 337
      header, 335
      Input Type section, 338
      Name and Synopsis sections, 335-336
      Notes section, 338-339
      Output section, 340-341
      Parameters section, 337-338
      Related Links section, 339
      Return Type section, 338
      Syntax section, 336-337
   -xor operator, 136
   [xy] operator, 133
   [x-y] operator, 133