

# PowerShell AND WMI

Richard Siddaway



MEAP

 MANNING



**MEAP Edition**  
**Manning Early Access Program**  
**PowerShell and WMI MEAP Production Version**

Copyright 2012 Manning Publications

For more information on this and other Manning titles go to  
[www.manning.com](http://www.manning.com)

# Table of Contents

## **Part 1 Tools of the trade**

1. Solving administrative challenges
2. Using PowerShell
3. WMI in depth
4. Best practices and optimizing the use of PowerShell and WMI

## **Part 2 WMI in the enterprise**

5. System documentation
6. Disk systems
7. Registry administration
8. Filesystem administration
9. Services and processes
10. Printers
11. Configuring network adapters
12. Managing IIS
13. Configuring a server
14. Users and security
15. Logs, jobs, and performance
16. Administering Hyper-V with PowerShell and WMI

## **Part 3 The future: PowerShell v3 and WMI**

17. WMI over WSMAN
18. Your own WMI Cmdlets
19. CIM cmdlets and sessions

## **Appendixes**

- Appendix A: PowerShell reference
- Appendix B: WMI reference
- Appendix C: Best Practices
- Appendix D: Useful links

# 1

## *Solving administrative challenges*

### ***This chapter covers***

- The administrator's headache
- Solving the challenge with automation
- PowerShell and WMI—the automation tools

Ask any Windows administrator about their biggest problems, and somewhere in the list, usually near the top, will be too much work and not enough time to do it. They know that automation is possible, will be at least aware of some of the technologies that could solve their problems, such as Windows Management Instrumentation (WMI) and PowerShell, but don't have the time to spend investigating the technologies. That's a shame because it's commonly accepted that 70 percent of an organization's IT budget is used to "keep the lights on." Automation can make a worthwhile contribution to reducing that percentage and freeing people and money to contribute to the business bottom line.

It's also possible that they've looked at WMI or PowerShell and decided they were too difficult. This is an understandable view, given the issues with WMI in VBScript—especially the amount of work involved in getting WMI to work in VBScript, and the lack of usable examples that also explain the techniques that have to be used. Some horrendous examples of PowerShell have been posted on the web that put me off, never mind someone wanting to start with the subject! Unfortunately, administrators then miss out on the possibilities that automation provides to reduce their workload and accomplish more.

The aim of this book is to radically lower the entry bar to using WMI productively in your environment. The examples that are provided can be used with few or no, changes. You'll also gain a deeper understanding of WMI that can be used to work with areas we don't cover.

PowerShell itself is constructed to make WMI usage much easier and more intuitive. PowerShell is Microsoft's automation engine that, among other things, provides easy-to-use

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=719>

access to the rich management toolset available in WMI. Together, PowerShell and WMI provide a set of tested techniques that will enable you to administer your Windows environment more quickly and easily. You'll be able to automate many of the standard tasks that currently consume too much of your attention, freeing up time to do the more interesting things that otherwise couldn't be fitted into your normal working day.

The first thing I'll do in this chapter is define the problem we're trying to solve. There are a number of issues that affect any Windows environment of significant size:

- Number of systems
- Rising infrastructure complexity
- Rate of change

The second part of the chapter shows why PowerShell and WMI provide a great toolset for solving these problems. Getting the most out of PowerShell involves investing a little time in learning it, especially when using WMI. Automation is the key to making your life as an administrator easier. The benefits you can achieve with PowerShell and WMI automation provide an excellent return on the investment you make in learning to use the technologies.

The chapter closes with two examples showcasing the power this combination of technologies delivers to us. The first example shows how you can shut down all the servers in your data center with one command, and the second shows how you can test settings on many machines in one pass.

Let's start with a look at the responsibilities of a modern Windows administrator and the problems administrators face.

## **1.1 Administrative challenges**

Administrators are very busy people. They seem to be continually asked to do more with fewer resources. Figure 1.1 illustrates this with a sketch graph that I'll refer to in the following sections. One thing the graph illustrates is the ever-decreasing cost, in real terms, of hardware. For example, I recently acquired a laptop with a quad-core processor (hyperthreading allows Windows to see eight cores) and 16 GB of RAM as a mobile lab. A few years ago, a machine with those specifications was a mid-range server not a laptop!

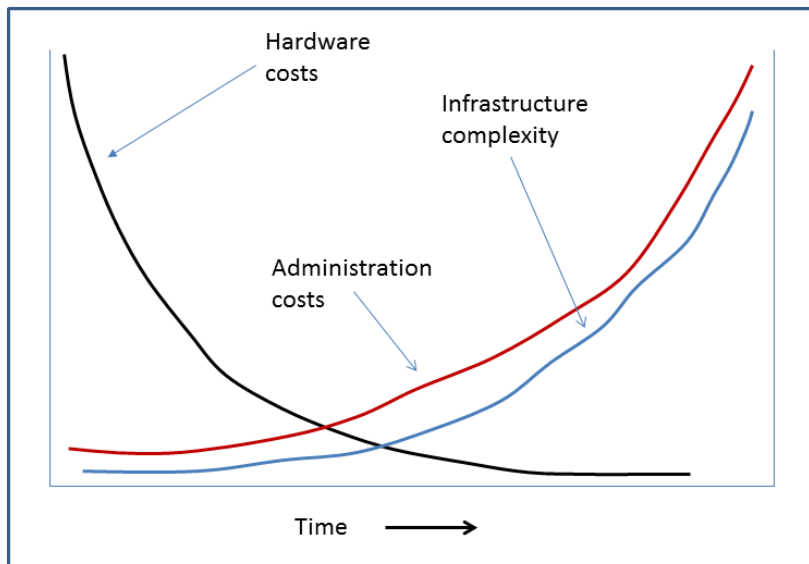


Figure 1.1 The relationship between decreasing hardware costs, increasing infrastructure complexity, and the cost of administering the evolving infrastructure

The same is true in the server market—4-, 8-, or even 10-core processors and lots of relatively cheap memory mean that we can afford to run applications and business processes that were previously only considered by large corporations with huge budgets.

This leads directly to the other components of the graph, which show the steep rise in infrastructure complexity and the even faster rise in administration costs. The continual upward growth of infrastructure complexity and cost isn't sustainable. PowerShell and WMI can help you break out of this growth curve. First, though, we need to examine the problem in a little more depth—where do the complexity and cost of administration come from?

### 1.1.1 Too many machines

This may seem to be an odd way to look at infrastructure, but do you really need every server you've created? Many, if not most, organizations have too many servers. This comes about for a number of reasons:

- *The decreasing cost of hardware*—This change leads to it being easier to add a new server than to think about using an existing one.
- *Department- or project-based purchasing*—This approach raises questions about server ownership and makes departments or projects unwilling to share resources.
- *The "one application—one server rule"*—Separating applications so that a problem in one doesn't affect others may still be valid for business-critical applications, but it's

not necessarily required for second- or third-line applications. It's definitely not required for testing and training versions.

- *Weak or reactive IT departments*—The lack of controls and processes in IT leads to departments and projects introducing systems that IT doesn't know about and has had no involvement with until the systems hit production.

An administrator's workload increases faster than the rate of increase of machines due to the time spent switching one's focus between machines (often requiring a new remote connection to be made) and the additional complexity each machine and its supported applications bring to the environment.

Virtualization is one of the hot topics in IT at present, with most organizations virtualizing at least part of their server estates. There are several advantages of virtualization:

- Reduced numbers of physical servers
- Reduced requirement for data center facilities, including space, power, and air conditioning
- Increased use of physical assets, giving a better return on investment

The organization as a whole benefits from virtualization, but the administrator's load is increased. If you have 100 servers to administer before virtualization, and you change to use 4 physical hosts and virtualize the 100 servers, you now have 104 systems to administer. The complexity may increase as well, because the virtualization platform may introduce a different operating system into the environment. The increase in the total (physical plus virtual) number of systems also means that there will be more change happening as the environment evolves.

### **1.1.2 Too many changes**

Change can be viewed as an administrator's worst headache. Unfortunately environments aren't static:

- Operating system and application patches are released on a regular basis.
- New versions of software are released.
- Storage space needs to be readjusted to match usage patterns.
- Application usage patterns force hardware upgrades.
- Virtualization and other disruptive new technologies change the way environments are created and configured.

This level of activity, multiplied across the 10s, 100s, or even 1000s of machines, builds on top of the day-to-day activity, such as monitoring and backups.

This situation isn't supportable in the long term. Organizations can't absorb ever-increasing administration costs, and today's economic realities prevent other mechanisms, such as increased revenue, from providing an escape. The situation has to be resolved by reducing the cost of administration. But administrators are hampered in doing this by the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=719>

fact that many changes bring new technologies into the environment without ensuring they're supportable.

### **1.1.3 Complexity and understanding**

Complexity is the real problem in many cases. It can arise due to a number of causes:

- Multiple operating systems bring different toolsets and terminology, even between versions of Windows.
- Different types of applications, such as databases, email, Active Directory, and web-based applications, require different skill sets, use different tools, and place different stresses on a server that the administrator must accommodate.
- Many machines perform the same or similar roles, but subtle, potentially undocumented, differences increase the likelihood of error.

Complexity is often compounded by incomplete knowledge and skill sets on the part of the administrators. Too often a project will introduce a new technology and administrators are expected to immediately pick up and manage the systems. Do the administrators have the skills? Do they have the time to learn the intricacies of the new technology? Sadly, the answer to both questions is often no.

This leads administrators to make best-guess decisions about how to do things. Sometimes, if the new technology is a version change from something already in use, administrators will continue to use the old methods even if there's now a better way to perform the task.

This lack of skills and knowledge leads to mistakes, and these mistakes cost money, often in terms of lost revenue for the organization. This puts more pressure on the administrators and leads to a lack of trust from the business. The IT department is often then excluded from discussions about new technologies until it's too late, and the cycle takes another spiral downwards.

Not only are major changes introduced by projects, administrators also face the host of minor changes required to keep their environments secure and running smoothly.

## **1.2 Automation: the way forward**

The way to overcome these issues is to introduce automation. Get the machines to do the mundane, repetitive work—that's what we invented them for!

Automation means many things to many people. There's a hierarchy of automation activity that can be considered, as shown in figure 1.2.

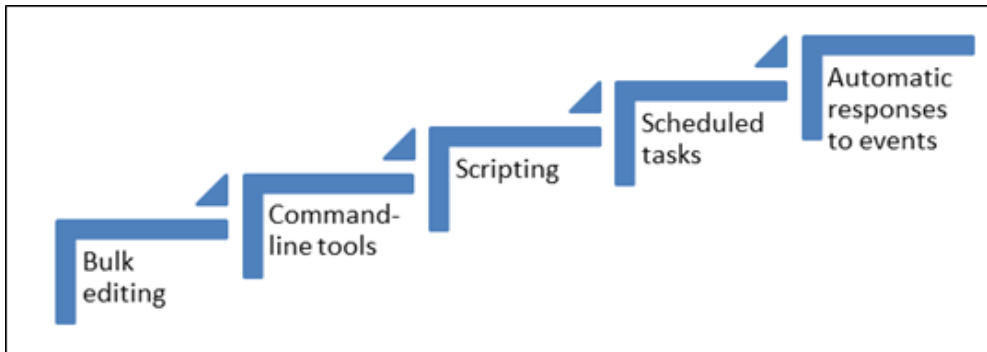


Figure 1.2 Hierarchy of automation activity

The question that needs to be answered by every organization is, “Where do I get the most benefit?” The answer depends on what you’re trying to achieve and where you are now. I know of a number of organizations that are quite happy using the standard Windows tools and a few bulk-editing tools. Others attempt to schedule everything or even create automated responses to events. Automation, for most organizations, involves a mixture of command-line tools, scripting, and scheduled tasks.

That leads to the second big question, which is, “How do I automate my administrative tasks?” PowerShell provides a set of command-line tools (called cmdlets) that can be used interactively. As the commands become longer and more ambitious, there’s a natural progression into scripting. One of the great strengths of PowerShell is that you can use exactly the same commands in a script or at the command prompt, so everything you’ve learned about commands is still usable in scripts.

PowerShell by itself is a wonderful tool (OK, I am fanatical about it), but you can take it a stage further and layer WMI on top. This opens a standards-based management toolset that you can use on local and remote machines and that can potentially include non-Windows systems when PowerShell v3 is used. The scripts can be run interactively or they can be scheduled to run at a specific time by using PowerShell and WMI. But before we get into those delights, let’s have a look at automation in general.

In this book, we’ll be concentrating on scripting as the primary automation activity. It could be argued that because you’re using PowerShell, you could do much of your work from the command line. The benefit of scripting, though, is that you can reuse the code and save even more time by not having to rewrite the code each time you want to use it. This topic is covered in depth in chapter 4 of *PowerShell in Practice* (Manning 2010).

Scheduled tasks and automatic responses are too dependent on the particular environments for this book, so in chapter 3 we’ll start to look at how you can automate responses to events that occur on your systems. We’ll revisit this in later chapters as we

consider specific areas of administration. We won't neglect the use of the command line, though. Many of the examples are short enough to use interactively.

Let's look at an example. Suppose you need to determine the amount of free space on the C: drive of a number of machines in your environment. One way is to go to the data center, assuming they're all in the same data center, and log onto the console of each machine. You'd then need to open Windows Explorer or another tool and find the free space on the C: drive. Write down the answer, and repeat for the next machine on the list.

A slightly easier option is to use Windows' Remote Desktop functionality to connect to each machine. Then you'd need to manually obtain the information. With this approach you don't have to move from your desk, but it still takes too much time.

My favorite solution is to use PowerShell as shown in listing 1.1. Don't worry if you don't understand the code right now. We'll return to this script in chapter 6 when we look at how to administer the disks in servers.

### SCRIPTING CONVENTIONS

I discussed these conventions in the introductory material but if you're like me you skipped that part of the book.

I will usually refer to servers when discussing the types of machines you're administering but many of the techniques covered in this book can be applied to desktop machines as well.

PowerShell commands (cmdlets and functions) can have shortcut names, known as aliases, defined. I don't normally use aliases in scripts as I want to ensure that the scripts are readable and as easy to understand as possible. I also use the full parameter names in cmdlets.

There is one exception to this rule and that's for the utility cmdlets where I use the following conventions:

- `Where-Object` aliased as `where` but never as `?`
- `ForEach-Object` aliased as `foreach` but never as `%`
- `Select-Object` aliased as `select`
- `Sort-Object` aliased as `sort`

In the discussion around a script I always use the full cmdlet name.

I have adopted this convention for a number of reasons:

- On the advice of the PowerShell team.
- Because it represents accepted practice and usage.
- Because it's more readable.

- It saves some space on the page.

In this PowerShell example you start with a list of server names taken from my lab setup. This list is piped into a `ForEach-Object` cmdlet (aliased as `foreach`) that calls `Get-WmiObject` for each server in the list in order to find the information on the logical disk used as the C: drive. You then format the information and output it as a table, as shown in the following listing.

### Listing 1.1 Find free disk space

```
"dc02", "W08R2CS01", "W08R2CS02", "W08R2SQL08",
"W08R2SQL08A", "WSS08" | foreach {
    Get-WmiObject -Class Win32_LogicalDisk `
-ComputerName $_ -Filter "DeviceId='C:'" } |
Format-Table SystemName, @{Name="Free";
Expression={[math]::round($_.FreeSpace/1GB), 2}} -auto
```

The free space is recalculated from bytes to GB to make the results more understandable. Notice that PowerShell understands GB, as well as KB, MB, TB, and PB. The results look like this:

SystemName	Free
-----	----
W08R2CS01	119.04
W08R2CS02	118.65
W08R2SQL08	114.8
W08R2SQL08A	115.17
WSS08	111.41
DC02	118.53

**NOTE** I don't intend to show output from every script we discuss in the book, but I will show output occasionally where it aids in the discussion of a particular issue.

There are a number of enhancements that you could apply to this script:

- Put the computer names into a CSV file (as we'll do in listing 1.4)
- Add the results to an Excel spreadsheet, or a database, so that trends can be seen
- Schedule the task to run on a periodic basis

I use a similar script, with the first two enhancements, to regularly report on disk space trends for the organization I'm currently working with. I now have a tool that takes seconds to run against each machine and provides vital information. It's also quickly and easily extensible to cover other machines that may become of interest. The script took me a few minutes to write and test, and there's an immediate payback every time I use it.

PowerShell is designed to provide this type of return. In the words of Jeffrey Snover, the architect of PowerShell, "I firmly believe that economics determine what people do and don't do so PowerShell is designed from the ground up to make composable, high-level task

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=719>

oriented abstractions be the cheapest things to produce and support.” The full article, “The Semantic Gap,” is available from the *Windows PowerShell Blog* at <http://blogs.msdn.com/b/powershell/>. A search for *semantic gap* will take you to the post.

The second part of this book will show many examples of this concept in action, but for now we’ll have a closer look at PowerShell and discuss why it’s the ideal platform for automating your administration.

### 1.3 PowerShell overview

In this section, I want to show you why PowerShell is the ideal platform for automating your Windows administration.

PowerShell is now on its second version (with the third in beta at the time of writing). It’s part of the default installation of Windows 7 and Windows Server 2008 R2 (for Server Core it’s an optional install). PowerShell v2 also can be installed on Windows Server 2008, Windows Server 2003, Windows Vista, and Windows XP. PowerShell v3 is an integral part of the Windows 8 family of operating systems. This level of support means you can use PowerShell to manage all of your Windows systems.

**WINDOWS 2000 SUPPORT** Windows 2000 is now out of support and won’t be considered in this book. PowerShell doesn’t have an option to install on Windows 2000.

There are also an increasing number of applications that have PowerShell support built into them. It’s a requirement for all new versions of the major Microsoft products, and adoption by third-party vendors is steadily increasing the scope of PowerShell.

#### PowerShell resources

Chapter 2 provides an overview of PowerShell’s features, the language, and how to use it. It isn’t a full PowerShell tutorial, but it will explain what you need to understand the examples in the second part of the book.

Bruce Payette’s *Windows PowerShell in Action*, second edition (Manning 2011) provides the most detailed coverage of PowerShell from a language perspective. My *PowerShell in Practice* (Manning 2010) supplies many examples of using PowerShell to administer Windows systems.

The ability to access remote machines (which we’ll look at in chapter 2) simplifies administration, because you can automate your whole Windows environment from a single administration console. This is how you can break the curve of rising infrastructure complexity. We’ll look at how you can achieve this after we’ve examined PowerShell’s scope.

### **1.3.1 PowerShell scope**

PowerShell enables you to administer a range of applications, from those having direct PowerShell support built into them to community-inspired and -provided additions that are available for download.

A number of major applications have direct PowerShell support:

- Exchange 2007/2010 (probably the poster child of PowerShell support)
- SQL Server 2008/2012
- SharePoint 2010
- Various members of the System Center family

Other elements of the Windows environment have PowerShell support available through Microsoft or third-party additions, including the following:

- Active Directory
- IIS
- Clustering
- Terminal Services
- Graphical presentation tools

The availability of functionality is good, but to get the most from it you need to get it into production use, and the sooner the better. One way to achieve this is to take advantage of the PowerShell community, which supplies sample code that can shorten the development cycle. PowerShell has a very strong and productive community that starts with the PowerShell team but also includes the following resources (links are provided in appendix D):

- Blogs, including mine
- Code repositories for community contributions, such as [www.poshcode.org](http://www.poshcode.org) and [www.powershell.com](http://www.powershell.com)
- Forums, such as [www.powershell.com](http://www.powershell.com)

This provides a breadth and depth of support and additional functionality that almost guarantees you'll be able to find help with solving your problem.

### **1.3.2 PowerShell and .NET**

Whenever PowerShell is discussed, the fact that it's .NET-based and can access most of the .NET Framework is brought up. At this point, I find that the eyes of many administrators begin to glaze over and they slide down into their seats. WAKE UP!

Yes, PowerShell is .NET-based and there are some really clever things that can be done by working directly with .NET code in PowerShell, some of which we'll see in later chapters. But you don't need to do this until you're ready to work at this level. There are a huge number of administration tasks you can perform without dipping your toes any further into the .NET waters than we did in listing 1.1. Just don't forget that .NET is there when you need

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=719>

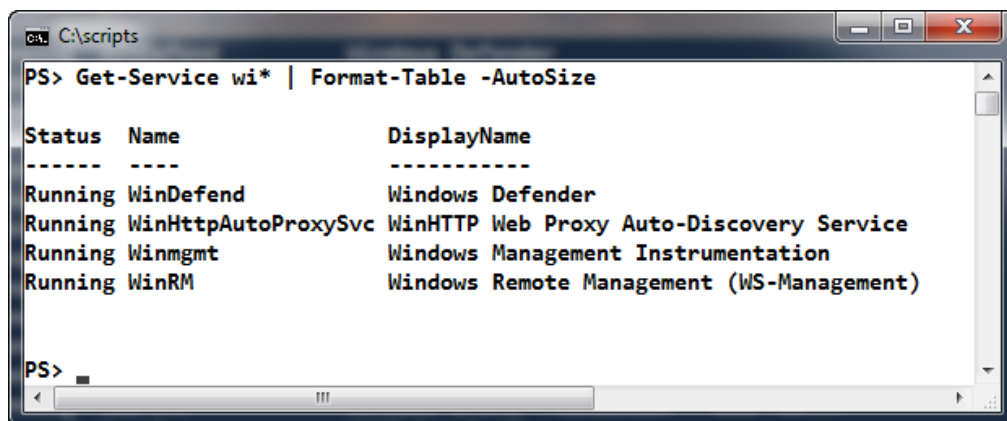
it, and there are lots of great examples of how to work with .NET available from the PowerShell community.

### WMI and .NET

It's possible to use WMI functionality through .NET code created and run in PowerShell. This is an advanced technique that we'll look at in chapter 12, when we're working with IIS.

There are generally alternatives to using .NET in this way, and I'll always choose those over a .NET-based solution. I'm an administrator, not a developer, and I'll present solutions for administrators.

As an example of how you can use .NET with PowerShell, let's look at the services running on a system. A subset of the installed services on my test system is shown in figure 1.3.



```

C:\scripts
PS> Get-Service wi* | Format-Table -AutoSize

Status Name                DisplayName
-----
Running WinDefend         Windows Defender
Running WinHttpAutoProxySvc WinHTTP Web Proxy Auto-Discovery Service
Running Winmgmt          Windows Management Instrumentation
Running WinRM            Windows Remote Management (WS-Management)

PS>

```

Figure 1.3 Using `Get-Service` to display a subset of the running services

The `Get-Service` cmdlet (a PowerShell command) returns a list of the running services. I have restricted the output by using `wi*` to only return services starting with "wi." The results are piped into a `Format-Table` cmdlet that outputs the results as a nicely formatted table.

**NOTE** The PowerShell pipeline passes .NET objects rather than text as in with other shells. This supplies a large measure of PowerShell's power.

I deliberately chose to use `wi*` because it demonstrates two services we'll be seeing a lot more of later: WMI and Windows Remote Management (WinRM). (It also keeps the figure to a reasonable size.)

Underneath the hood, `Get-Service` is using a .NET class called `System.ServiceProcess.ServiceController`, which is fascinating but doesn't mean much to me without looking up the .NET documentation. The beauty is that you don't need to know this 99.99 percent or more of the time. PowerShell abstracts all of this, and you can perform your discovery with an easy-to-use command that has a descriptive name.

### 1.3.3 *Breaking the curve*

In figure 1.1 you saw that there's a continuous rise in the complexity of organizations and the cost of performing the administration in those organizations. This continuous increase isn't supportable in the long term, and we need a way to break the upward curve.

PowerShell can help us break that curve by providing the following:

- A set of tools to interactively administer servers and applications
- An automation engine that works across the entire Windows estate
- The ability to apply those concepts to a number of applications
- Remote administration engines that enable multiple machines to be administered with a single command
- Asynchronous and scheduled tasks to further enhance automation

PowerShell offers a productivity boost that will easily repay the time you spend learning to get the best from the technology. And using PowerShell and WMI together will further enhance your productivity gains.

## 1.4 *WMI overview*

In this section, we'll look at what WMI offers us as administrators. We'll examine WMI in much greater detail in chapters 3 and 4, where we'll drill into the details of how to use it to automate administration tasks.

WMI has been available to Windows administrators since the days of Windows NT 4, but it isn't a static technology. Each new version of Windows brings changes to the functionality available through WMI, usually by adding extra capabilities but occasionally by removing or radically changing functionality. New versions of other applications can have a similar impact. For instance, Exchange 2003 had WMI support, but that was removed in Exchange 2007/2010.

**WMI AND OFFICE** Microsoft Office 2007 supplied a WMI provider in the shape of the `root\MSAPPS12` namespace. This functionality was removed in Office 2010. Remnants of WMI classes will remain on a system if an upgrade from Office 2007 to 2010 is performed, but they won't be usable.

The only way to be sure that particular functionality is available on your version of Windows is to check the documentation on the Microsoft Developer Network (MSDN). The WMI functionality available on a particular system can be discovered in a number of ways using PowerShell and other tools. Chapter 3 supplies full instructions on this.

There are many automation scripts available using WMI, and most of them use VBScript due to the efforts of the people behind the Microsoft Script Center after Windows 2000 shipped. This gives the unfortunate impression that WMI requires a lot of coding for you to achieve any gains. This is no longer true, as you'll see in a little while, and it's slowly becoming apparent on the internet as the PowerShell community supplies example code using WMI.

To get started, let's look at what WMI actually is.

### **1.4.1 What is WMI?**

Just what is WMI? The abbreviation stands for Windows Management Instrumentation and the functionality is automatically installed with Windows. The base functionality can be enhanced by adding features and roles to Windows or by installing additional applications.

At first glance, it may seem like a very large set of stuff that you might be able to use if you get lucky and someone has mapped out how to use the bit you're interested in. When we get to chapter 3, though, you'll see that there is a structure to WMI that you can exploit to discover what is available, and to some degree how to use it.

At this stage you need to be aware that WMI doesn't exist in a vacuum. It's Microsoft's implementation of the Common Information Model (CIM) that's produced by the Distributed Management Task Force (DMTF). The CIM (and WMI) defines a series of classes that supply information about Windows systems, and they may allow you to directly interact with aspects of local and remote systems.

You can look at the WMI classes available for working with disks by using `Get-WmiObject`:

```
Get-WmiObject -List *disk* | sort name | select name
```

This command will produce output like the following:

```
Name
----
CIM_DiskDrive
CIM_DisketteDrive
CIM_DiskPartition
CIM_DiskSpaceCheck
CIM_LogicalDisk
CIM_LogicalDiskBasedOnPartition
CIM_LogicalDiskBasedOnVolumeSet
CIM_RealizesDiskPartition
Win32_DiskDrive
Win32_DiskDrivePhysicalMedia
Win32_DiskDriveToDiskPartition
Win32_DiskPartition
Win32_DiskQuota
Win32_LogicalDisk
Win32_LogicalDiskRootDirectory
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=719>

```

Win32_LogicalDiskToPartition
Win32_LogonSessionMappedDisk
Win32_MappedLogicalDisk
Win32_PerfFormattedData_PerfDisk_LogicalDisk
Win32_PerfFormattedData_PerfDisk_PhysicalDisk
Win32_PerfRawData_PerfDisk_LogicalDisk
Win32_PerfRawData_PerfDisk_PhysicalDisk

```

There are a number of classes that start with CIM\_ and others that start with Win32\_. There isn't always a one-to-one pairing of the two types, but major object types such as logical disks are paired. The CIM\_ class is the parent, corresponding to the definition supplied by the DMTF; the Win32\_ classes are child classes that Microsoft has implemented. In some cases the classes are identical, and in others there is additional functionality in the Microsoft class. We'll usually be working with the Win32\_ classes.

### POWERSHELL v3

Microsoft invested very heavily in WMI for PowerShell v3, and it offers several improvements:

- A new API and associated .NET classes
- Closer adherence to the CIM standards (so expect less deviation from the standard in Microsoft's implementations)
- Simplified creation of WMI providers (see chapter 3 for details on providers)
- The ability to create cmdlets directly from WMI objects (see chapters 18 and 19)

The scripts in chapters 2–16 will use the existing WMI cmdlets for compatibility between PowerShell v2 and v3. Annotated versions of the scripts using the new PowerShell v3 CIM cmdlets will be available in the book's source code.

There are also classes for working with performance counters that you'll experiment with in chapter 15. As you'll see in future chapters, you can work with many parts of your systems.

Technologies that have this level of power tend to seem very complicated when you're first introduced to them. WMI is no exception.

#### 1.4.2 *Is WMI really too hard?*

In the years since its introduction, WMI has gained a poor reputation for a number of reasons:

- Many administrators don't think to look on MSDN for documentation. I know I didn't when I started using WMI.
- Discovering which classes are available isn't always easy. (Chapter 3 will show you how to discover detailed information on the WMI classes installed on a system.)
- Coding WMI can be time consuming. The examples in this book are ready to use,

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=719>

requiring few or no changes for your environment. They can also be used as templates for your own scripts.

- A lot of information is held in a coded form. For instance, the `Win32_LogicalDisk` class has a `media type` property that returns a numeric value—hard drives are type 3. If you don't know that, you can get into problems. The information about these values is available in the WMI documentation, and simple techniques are presented throughout this book to decode these values. The new CIM classes in PowerShell v3 also provide alternatives that simplify use.
- Class names aren't always consistent. For instance, there is a `Win32_LogicalDisk` class, but the physical counterpart is `Win32_DiskDrive` rather than a class called `Win32_PhysicalDisk`. This book's chapters are broken into topics that highlight the common classes you'll need to use, and this information is also gathered in appendix B.

WMI is a powerful technology that provides low-level access to the workings of your servers. It has been shunned by the administrator community because it's viewed as being too hard to use, but this need not be the case, as you've seen in this section. The rest of this book shows how you can make the most of this free power. Microsoft has also realized how much can be done with WMI and is providing a huge boost to the technology in the Windows 8 family.

But before we look at how to use WMI, we need to see how it works with PowerShell.

## **1.5 Automation with WMI and PowerShell**

PowerShell v1 has good WMI support. You can use `Get-WmiObject` to access existing WMI objects, as you saw in listing 1.1. You can also create new WMI objects, perform searches, and manipulate the objects that are available. This is all explained in detail in chapter 3.

This capability is raised to a new level with PowerShell v2. The functionality of PowerShell v1 is at least maintained in v2, and it's often enhanced. For example, you can work directly with WMI security levels in the WMI cmdlets. PowerShell v2 also provides additional cmdlets to modify and even remove objects. You also get the capability to work directly with WMI events, as you'll see in a number of chapters, including 7, 8, and 15.

### **Internet code**

You may not always have time to create a script to solve a problem, or you may not have the knowledge. Fortunately, there are a many scripts available on the Internet.

It's essential that you test any script that you downloaded from the internet or obtain from a book (including this one) in your environment to ensure that it works as advertised and doesn't have any adverse effects. The original script writer is always working based on the assumptions inherent in their environment—they can't know any of the quirks in your environment that would cause the scripts to cause accidental damage.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=719>

This is a warning that I will repeat periodically throughout the book.

Let's look at a WMI example. The starting point is VBScript and WMI, which you'll translate into PowerShell. This will provide a template that can be followed if you need to translate other scripts. The following code listing retrieves some information regarding the processes running on your system. The code is modified from Microsoft's Scripting Guide.

### Listing 1.2 VBScript to retrieve process information

```

set objWMIService = GetObject("winmgmts:" _                               #1
    & "{impersonationlevel=impersonate}!\\" _
    & ".\root\cimv2")

set colProcesses = objWMIService.ExecQuery _                               #2
    ("SELECT * FROM Win32_Process")

for each objProcess in colProcesses                                       #3
    WScript.Echo " "
    WScript.Echo "Process Name : " + objProcess.Name
    WScript.Echo "Handle       : " + objProcess.Handle
    WScript.Echo "Total Handles: " + Cstr(objProcess.HandleCount)
    WScript.Echo "ThreadCount  : " + Cstr(objProcess.ThreadCount)
    WScript.Echo "Path         : " + objProcess.ExecutablePath
next

```

#### 1 Enable interrogation of WMI

#### 2 Select from WMI class

#### 3 Output results

This example uses just a small subset of the available properties to keep the script manageable. The script starts by creating an object, `objWMIService`, to enable interrogation of the WMI service #1. A list of active processes is retrieved by running a WQL query #2. The collection of processes is iterated through, and you write a caption and the value of a particular property to the screen #3.

Writing a script like this takes time, even with cut and paste in your editor. You have to set up the link to WMI, run a query, and then manually define the formatting of your display.

This code can be directly translated to PowerShell as shown in the next listing. The PowerShell commands are explained in more detail in later chapters, but their basic functions should be understandable by comparison to the VBScript example.

### Listing 1.3 PowerShell translation

```

$procs = Get-WmiObject -Query "SELECT * FROM Win32_Process"
foreach ($proc in $procs) {
    Write-Host "Name           : " $proc.ProcessName
    Write-Host "Handle         : " $proc.Handle
    Write-Host "Total Handles : " $proc.Handles
    Write-Host "ThreadCount   : " $proc.ThreadCount
    Write-Host "Path          : " $proc.ExecutablePath
}

```

You first run the WMI query to select the information you need and put the results into a variable. The variable is a collection of objects representing the different processes. You can

then loop through the collection of processes (using the `foreach` command), and for each process in that collection use the `Write-Host` cmdlet to output a caption and the value of the properties you're interested in.

### VBScript to PowerShell conversions

When you're working with WMI, it's inevitable that you'll end up translating VBScript code into PowerShell because of the sheer number of examples that are available from sites such as the Microsoft TechNet Script Center.

The "VBScript-to-Windows PowerShell Conversion Guide" is available on the TechNet Script Center at <http://technet.microsoft.com/en-us/scriptcenter/default.aspx>. You'll need to find it through a search engine because it does move around on the site.

Consult this guide if it isn't obvious how to change a particular piece of VBScript into PowerShell.

Using this approach will produce the results that you need, but it doesn't use PowerShell to its full capabilities. You end up doing the formatting work yourself rather than leaving it to PowerShell. Your goal is to get the machines to do as much of the work as possible. You could just run `Get-WmiObject -Class Win32_Process`, but this displays a lot of information that you'd need to wade through, which is another manual process. You need to select the data you want to see and format it in a sensible way, which leads to the following PowerShell code:

```
Get-WmiObject Win32_Process |
Format-Table ProcessName, Handle, Handles,
ThreadCount, ExecutablePath -AutoSize
```

This final version uses the `Get-WmiObject` cmdlet directly. `Get-WmiObject` returns an object for each process, and you use the PowerShell pipeline to pass them into a `Format-Table` cmdlet. This combines the data selection and display functionality you saw earlier and produces neatly formatted tabular output. If you wanted the output in a list format, as in the previous two examples, you could substitute `Format-List` for `Format-Table`.

In these simple examples, you've progressed from a VBScript that requires a large amount of manual formatting to a one-line PowerShell version that does the formatting automatically. The final PowerShell version is small enough that you could type it and run it directly from the command line if required. But a better solution is to turn it into a script, or function, that can accept a machine name as a parameter, and you can use that across your server estate. You'll learn how to do this in chapters 2 and 3.

So far, the examples in this chapter have focused on day-to-day tasks we face as Windows administrators. We'll close the chapter with a couple of examples showing how PowerShell and WMI can help with some of the bigger tasks we might be asked to perform on a less frequent basis.

## 1.6 *Putting PowerShell and WMI to work*

The two examples we'll look at in this section are things that you may not need to perform on a frequent basis, but they'll involve a lot of work if you have to do them manually.

The first example involves shutting down all the Windows machines in your data center. This isn't an everyday task, but as an example of the power of PowerShell and WMI, it's difficult to beat.

The second example involves auditing a large number of machines to discover their capabilities. This can be especially useful when starting a new role or if you're performing any kind of investigation. In many cases you may have the base information but need other data to complete your knowledge. The techniques we develop in later chapters will build on this example.

Could you achieve these results without PowerShell and WMI? Yes, but it would not be as easy or efficient. You could shut down all the machines in your environment in a few different ways:

- Physically visiting each machine
- Using a remote desktop utility to access each system and shut it down
- Accessing an "out of band" management card in the server and forcing a shutdown
- Creating a script in another language to access a utility that forces a shutdown

All but the last one involve shutting down the system manually. That may be acceptable for a handful of servers, but not for tens or hundreds of machines.

Auditing can be achieved with a number of utilities, but they involve extra expense, infrastructure, and a learning curve. If you have PowerShell, WMI, and this book, you can perform these tasks for a fraction of the cost and time required to set up alternative systems.

Now, how can you shut down that data center?

### 1.6.1 *Example 1: Shutting down a data center*

Shutting down a data center isn't something that you want to do every day. At least, it isn't if you want to keep your job. But there are times when it's necessary, such as if there will be major work on the power supplies or air conditioning systems and it would be safer to have all the systems offline.

I first used this technique when I had to shut down all of the servers in a data center because we were moving them to a different location. Instead of having to travel 150 miles to supervise the shutdown, I did it all remotely and closed the gateway machines as I exited the environment.

**NOTE** I used this script on a regular basis to shut down the lab I used to develop the scripts presented in this book.

The `Win32_OperatingSystem` class has a method called `Win32Shutdown` that can be used to stop all your machines, as shown in the next listing.

#### Listing 1.4 Shut down a data center

```
Import-Csv computers.csv |
foreach {
  (Get-WmiObject -Class Win32_OperatingSystem `
    -ComputerName $_.Computer ).Win32Shutdown(5)
}
```

This script uses a CSV file called `computers.csv`, which contains a list of computer names.

The version that I use to shut down my lab contains the following lines:

```
Computer
W08R2CS01
W08R2CS02
W08R2SQL08
W08R2SQL08A
WSS08
DC02
```

The first line is a header, and each subsequent line has one computer name (computer names aren't case sensitive). You can generate this file manually for a small number of machines or create it with a script that queries Active Directory for a larger environment.

In some cases, you need to control the order in which machines shut down; for example a front end SharePoint server should be closed down before the back end database server. This is achieved by editing the order in which computer names appear in the CSV file.

In listing 1.4, `Import-Csv` is used to read the CSV file. The contents are piped into a `ForEach-Object` cmdlet (`foreach` is an alias). For each computer name that's passed along the pipeline, you use `Get-WmiObject` and the `Win32_OperatingSystem` class. The computer name is passed as `$_Computer`, where the `$_` symbol refers to the current object on the pipeline and the `Computer` part comes from the CSV header.

The `Get-WmiObject` is contained in parentheses, `()`, so you can treat it as an object on which you can call the `Win32Shutdown` method. The value of 5 that's passed to the method forces a shutdown even if users are still logged on. This approach will be revisited in chapter 19 when we look at new ways to work with WMI in PowerShell v3.

In PowerShell v2, you could make this script even simpler by using the `Stop-Computer` cmdlet instead of a WMI call. Other possible enhancements included pinging the server prior to shutdown to ensure you can contact it and putting a delay between each machine you shut down to ensure that linked servers don't have problems.

The next example involves auditing servers. You can never have too much information about your servers' configuration.

### 1.6.2 Example 2: Auditing hundreds of machines

This example shows how you can gather basic information from many machines. You could get this same level of data by connecting to each machine in turn, running utilities on the

system to get the information, and then recording it, but that's a lot of work for more than a few machines.

The audit should return the following information:

- Server make and model
- CPU data (numbers, cores, logical processors, and speed)
- Memory
- Windows version and service pack level

You can use a number of WMI classes, as shown in the following listing, to accomplish this task.

### Listing 1.5 Computer audit

```

Import-Csv computers.csv | #1
foreach {
  $system = "" | #2
  select Name, Make, Model, CPUs, Cores,
  LogProc, Speed, Memory, Windows, SP

  $server = Get-WmiObject -Class Win32_ComputerSystem ` #3
  -ComputerName $_.Computer

  $system.Name = $server.Name
  $system.Make = $server.Manufacturer
  $system.Model = $server.Model
  $system.Memory = $server.TotalPhysicalMemory
  $system.CPUs = $server.NumberOfProcessors

  $cpu = Get-WmiObject -Class Win32_Processor ` #4
  -ComputerName $_.Computer | select -First 1

  $system.Speed = $cpu.MaxClockSpeed

  $os = Get-WmiObject -Class Win32_OperatingSystem ` #5
  -ComputerName $_.Computer

  $system.Windows = $os.Caption
  $system.SP = $os.ServicePackMajorVersion

  if (($os.Version -split "\.")[0] -ge 6) {
    $system.Cores = $cpu.NumberOfCores
    $system.LogProc = $cpu.NumberOfLogicalProcessors
  }
  else {
    $system.CPUs = ""
    $system.Cores = $server.NumberOfProcessors
  }

  $system
} |
Format-Table -AutoSize -Wrap #6
#1 Computer list

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=719>

```

#2 Create object
#3 Computer system
#4 Processor
#5 Operating system
#6 Output

```

As in listing 1.4, you again have a CSV file #1 that contains a list of computer names. This file is read using `Import-Csv` and the results are piped into `foreach` (an alias of `ForEach-Object`).

The easiest way to present the final data is in a table, so you need to create an object #2 to hold the results. One method of creating such an object is to pipe an empty string, "", into a `select` statement with the names of the properties you want the object to have. Note that this only works for properties that are strings. There are other ways of creating objects, which you'll see in later chapters.

Once you have your object, you can start gathering the data. The first data concerns the computer system itself, which you can find with the `Win32_ComputerSystem` class #3. `Get-WmiObject` is used, and the results are put into a variable. You can then map the required properties across to the object you're using to store the results.

This process is repeated for CPU data #4 and operating system details #5 using the `Win32_Processor` and `Win32_OperatingSystem` classes respectively. You test the operating system version:

- If the major version number (the first part) is greater than or equal to 6, you're dealing with Windows Server 2008, Windows Vista, or later. In this case, the version of WMI will correctly return the number of cores and logical processors per physical CPU so you can populate the fields.
- If the major version number is less than 6, you're dealing with Windows Server 2003, Windows XP, or earlier, and you can only retrieve the total number of cores, so you show that and remove the number of processors.

The differences between the output for the two types of operating system are shown in the following output extract:

```

Make      Model      CPUs Cores LogProc Windows
----      -
Dell Inc. PowerEdge 1950    2    2      2 Windows Server® 2008
Dell Inc. PowerEdge R710    16           Windows(R) Server 2003
Dell Inc. PowerEdge 1950    8           Windows(R) Server 2003

```

When your object is fully populated, you can pipe it into `Format-Table` #6. The `-Autosize` parameter will control the width of the columns to best display them on screen, and the `-Wrap` parameter will cause any data that's too long to fit in the column to wrap onto multiple lines to ensure you see all of the results.

This script could be enhanced in a number of ways:

- Ping the server to ensure it's reachable
- Add further information, such as disks, installed applications, hotfixes, and page file configuration

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=719>

- Output the data to a file that could become the basis of your server documentation
- Add the data directly into a CMDB for configuration management

The first two points are covered in later chapters, whereas information on the second two can be found in *PowerShell in Practice* (Manning 2010).

These two examples show what can be achieved with the right knowledge and a few lines of very simple code. I don't know of any other combination of out-of-the-box tools that performs so much work for so little effort. WMI really does put the power in PowerShell.

## 1.7 Summary

As Windows administrators, we're under increasing pressure due to the rise in complexity of the environments we have to work in and the ever-rising costs of administration. On the one hand, we're being asked to take on more work and administrate a steadily increasing number of servers and applications. On the other hand, we're facing demands to cut costs.

The way out of this dilemma is to automate as much of our day-to-day work as possible. In a Windows environment, the pairing of PowerShell and WMI provides an unmatched set of capabilities:

- PowerShell is available on all currently supported Windows platforms except Windows Server 2008 Server Core.
- PowerShell support is built into an increasing number of Microsoft and third-party applications.
- WMI provides low-level access to hardware, operating systems, and applications enabling full lifecycle management.
- The ability to work with remote systems simplifies administration and stretches the envelope of automation.
- PowerShell enables WMI to be used much more easily and shortens the learning curve to productive use of the technologies.
- There is an existing body of knowledge regarding WMI that can be readily adapted for use in PowerShell.
- A thriving PowerShell community provides support for PowerShell and WMI usage.
- Both technologies will be available for the foreseeable future, ensuring that your investment will continue to show returns.

In the next chapter, we'll dive deeper into PowerShell to make sure you have all of the tools you need to get the most out of WMI.