When you need to automate a Windows administration task, the chances are that you will use VBScript or batch scripting. There are several more powerful options available, but when you need to get the job done right away, taking the time to learn a new scripting language is rarely at the forefront of your mind. This could be a significant reason why there has been a less than explosive take up of Microsoft PowerShell. This is a genuine shame, because it is an incredibly powerful tool that can be as simple or as complex as you want it to be.

PowerShell (formerly known as Monad) is Microsoft’s next-generation shell scripting environment. It takes the best elements from other shells such as Bash, KSH and CMD.exe, as well as many advanced scripting features from languages such as VBScript and Python. It was specifically designed as an environment for managing Windows, which means that the developers considered what administrators would use this tool for when they designed it: automating mundane tasks and creating tools to manage their systems.

When you think about all of the issues that we’ve been putting up with when using command prompt for all these years, it’s surprising that nothing else came along to replace it earlier. The main issues with command prompt were inconsistent syntax (both in the naming of commands and use of switches), minimal help and strong reliance on EXEs and scripts to bolt on missing functionality. The great news is that all of these things have now been addressed with the new shell environment.

Starting out
Getting up and running is easy. All that’s required is .NET Framework 2.0 and the latest build of PowerShell (which is currently RC1). When you run PowerShell for the first time you will be prompted to run a number of scripts that will set up options such as aliases, snap-ins and user profiles. These options allow plenty of scope for all sorts of customisation.

After that, the interface looks remarkably similar to the traditional command prompt interface. The only obvious difference to be seen is that the new shell sports a ‘PS’ in front of the prompt.

Reassuringly, all of the traditional commands such as Dir and Copy still work, but what isn’t immediately obvious is that they now exist as aliases to cmdlets (these are pronounced commandlets). A cmdlet is the new shell’s take on the command-line utilities that Cmd used, such as Copy and Del. This is a really sensible move because it has allowed the developers to be obsessive about maintaining a strict naming convention, while still allowing a gentle learning curve of familiar commands.

All cmdlets are named in the form verb-noun, which makes it easy to guess the command you’re looking for. For example, knowing that Get-Service gives you all services, it’s easy to figure out that Stop-Service will stop a specified service.

Getting help
But it may not always be obvious what the command you’re looking for is or what the syntax is once you’ve found it, which is why two of the most useful commands are Get-Help and Get-Command. The Get-Help cmdlet is very similar to the UNIX man pages function and will spew pages and pages of help and examples on any cmdlet. Get-Command is really useful for finding a command because it can filter by verb or noun. For example, the Get-Command –noun service command will return a list of all of the service-related cmdlets (Figure 1).

More and more Microsoft systems use Windows Management Instrumentation (WMI). Most of my VBScripts query it in one form or another in half a dozen or so lines of code. Reading objects from WMI couldn’t be simpler because PowerShell comes fully equipped with the Get-WMIObject cmdlet: it does exactly what it says on the tin.

Another neat trick PowerShell has when playing around with other data sources is its ability to treat...
things like the registry as just another drive, allowing you to traverse it as though it were a folder structure.

PowerShell’s party piece is its object orientation. Previously, output to the shell was in the form of a text stream. Now with PowerShell it’s a text representation of the actual objects that can be operated on individually. What’s more, it can use any of the .NET 2.0 classes. This allows for really interesting possibilities. Imagine an SMS systems administrator has been asked to provide a tool for first line support to perform a task. A VBScript controlled with message box prompts would be the obvious choice. Because of the .NET class exposure you could just use the full power of Windows Forms that would normally be reserved for a developer rather than an administrator. The example below shows how only a few lines of script could build up a rich GUI to drive tools that you might write to assist with day-to-day tasks (Figure 2):

```powershell
[void] [Reflection.Assembly]::LoadWithPartialName("System.Windows.Forms")
$form = new-object Windows.Forms.Form
$lblCollection = new-object Windows.Forms.Label
$lblComputer = new-object Windows.Forms.Label
$txtCollection = new-object Windows.Forms.Textbox
$txtComputer = new-object Windows.Forms.Textbox
$btnAdd = new-object Windows.Forms.Button
$form.Text = "SMS Collection Populator"
$form.Size = "330,130"
$lblCollection.Text = "Collection ID:"
$lblCollection.Location = "10,12"
$txtCollection.Location = "110, 8"
$txtCollection.Size = "200, 25"
$lblComputer.Text = "Computer Name:"
$lblComputer.Location = "10,36"
$txtComputer.Location = "110, 38"
$txtComputer.Size = "200, 25"
$btnAdd.Text = "Add to Collection"
$btnAdd.Location = "10,68"
$btnAdd.Size = "300,25"
$form.Controls.Add($lblCollection)
$form.Controls.Add($txtCollection)
$form.Controls.Add($lblComputer)
$form.Controls.Add($txtComputer)
$form.Controls.Add($btnAdd)
$form.ShowDialog()
```

Of course, this doesn’t do anything until more lines of code are added behind the button. Also, a combo box would be nice to allow the user to select values in a more friendly fashion. But the possibilities for enhancing scripts and tools are obvious. All of the .NET objects, properties and methods are well documented on the Microsoft web site and because there’s no need to compile it’s really fast and easy just to play. For this reason it’s possible PowerShell might enjoy a following as a prototyping language among developers.

I’d like to make a suggestion about searching for PowerShell information. Because it was previously known as MSH and Monad, it’s worth trying the old names as well as the new in any web searches: some of the useful stuff was written in the pre-PowerShell days. Also, when looking for information on .NET objects, poking around C# sites can give ideas on how things have been done using the same objects, then it’s simply a matter of adapting the syntax.

It’s worth familiarising yourself with PowerShell for two main reasons. First, Microsoft has decreed that future admin GUIs will be layered on top of PowerShell. This will mean that in order to manage and automate these new products, starting with Exchange 12, there probably won’t be traditional command-line utilities provided, just cmdlets. Secondly, it will give you a warm fuzzy feeling the next time a UNIX administrator is droning on about Windows’ lack of support for their favourite command-line feature and you say: “Actually Windows can do that too – and better.”

Geoff Collins is an SMS consultant at 1E, a specialist in Windows deployment and management. You can reach him at editorial@server-management.co.uk