Administrative tasks using Windows PowerShell

Administrative tasks using Windows PowerShell: Introduction and examples of how to use the scripting technology in everyday IT administration

MICROSOFT SWITZERLAND

Januar 1, 2008
Frank Koch (BERN)
Developer & Platform Group

Administrative tasks using Windows PowerShell

Requirements for this workshop

This workshop is the follow-up to our highly successful Windows PowerShell foundation course. This time, the focus is on daily IT administration tasks using Windows PowerShell. Of course, it is not possible to cover all aspects of this broad spectrum of tasks here – much more comprehensive books than this booklet are available on the subject. However, we do hope that, by way of examples and tasks, this booklet will awaken your own curiosity, and that you will incorporate Windows PowerShell more and more into your everyday IT administration tasks.

If you don't yet know anything about Windows PowerShell, it's a good idea to acquaint yourself with it a little to start off with. You will find numerous good descriptions and exercises on the Internet for this purpose. I have also created a brief introduction that you can download for free from the Internet. At http://blogs.technet.com/chITPro-de, you can download the "Windows PowerShell" exercise booklet as a PDF file (available in German and English) as well as various practice files. Apart from that, no further requirements exist for the initial foundation course – all you need is an up-to-date Windows system with Windows PowerShell installed, which is available as a free download from Microsoft.

In this follow-up workshop, however, you will be working mainly in a Windows Server environment. This limitation is necessary, as many administrative tasks only make sense when carried out on servers. If you do not have access to a Windows Server for your tests (by this, I DO NOT mean a live server, but instead an additional test system), you can download a similar environment from Microsoft for free. Please note that the download volume is over 1 GB, so do ensure that you are aware of the download time and any potential costs for your own Internet connection beforehand. You will find a detailed description and instructions on how to download and initially configure the test environment as an annex to this workshop. In addition, I recommend that you use the practice files for this workshop, which are also available as a free download. More information on this can be found in the annex.

Please ensure that the correct server names, Active Directory configuration and other details in the following exercises are selected in line with the test environment mentioned. If you use this booklet ***WITHOUT*** the virtual environment, you may not be able to carry out certain exercises, or certain examples and scripts must be adapted for use in other environments.

Other information sources on the Internet

You will find the Windows PowerShell homepage, including a download link, at www.microsoft.com/PowerShell.
Here, you will also find links to very good Webcasts, books and other help forums.

The best blog page on Windows PowerShell is http://blogs.msdn.com/PowerShell/. Here, you will find all the information you need on script techniques, alongside practical demonstrations.

In the Swiss IT-Pro team blog (http://blogs.technet.com/chITPro-DE) you will find the "Windows PowerShell" exercise booklet mentioned beforehand, as well as links to Windows PowerShell Webcasts (in German).

Helpful key combinations for a Swiss standard keyboard

CHARACTER	KEY COMBINATION	MEANING
|	ALTGR 7(NOT: ALTGR1 = ¦)	FORWARD THE OUTPUT OF A COMMAND
`	SHIFT ^,FOLLOWED BY A BLANK	CONTINUE COMMAND ON NEXT LINE
{	ALTGR Ä	START OF A COMMAND SEQUENCE (E.G. FOLLOWING AN "IF" STATEMENT)
}	ALTGR $	END OF A COMMAND SEQUENCE (E.G. IN THE CASE OF AN "IF" STATEMENT)
[	ALTGR Ü	NEEDED FOR SOME OBJECTS
]	ALTGR !	NEEDED FOR SOME OBJECTS
TAB	TAB KEY	COMPLETES COMMANDS WHERE NECESSARY EXAMPLE: GET-HE (TAB) = GET-HELP

Windows PowerShell was developed in Redmond, and is ideally suited to the American standard keyboard layout. Should you be using a Swiss keyboard, you will notice that some commonly used keys are hard to find. A small list of shortcuts is provided for you here.

Contents

Administrative tasks using Windows PowerShell 2
  Requirements for this workshop 2
  Other information sources on the Internet 2
  Helpful key combinations for a Swiss standard keyboard 3
Contents 4
A Automatic installation of Windows PowerShell 5
B Security in Windows PowerShell 6
C Exercises with files 8
D Working in the registry 11
E Exercises using .NET and WinForms 12
  Evaluating file types (extensions) 15
  Setting ACLs in the file system 16
  Sending an e-mail from a script 17
F Exercises with log files and event logs 20
G Exercises with Peedy 21
H Exercises with the Active Directory 26
  Performing a search in the Active Directory 27
  Working with partner add-ons: Quest 28
I Reports using Office Web Components and PowerGadgets 31
J Performing evaluations with WMI 36
K Partner add-ons: community tools 38
  The Active Directory as a PSDrive, thanks to PSCX 39
L Partner add-ons: assigning GPOs using SDM Software 41
M Partner add-ons: FullArmor Workflow Studio 43
N Access to databases via .NET 46
Solution scripts for the exercises in this booklet 49
  Script: Out-Chart.ps1 63
  More sample scripts 69
  Configuring the test environment 71
  Installation of a simple e-mail environment 72
Further reading 73

WORKING WITH WINDOWS POWERSHELL

A Automatic installation of Windows PowerShell

Windows PowerShell is a free add-on for Windows XP systems and higher. It can be downloaded from Microsoft at http://www.microsoft.com/powershell. Windows PowerShell itself is relatively small to download; however, it requires .NET Framework 2.0, which also must be downloaded if it is not already used.

It is easy to automatically install Windows PowerShell. System management tools such as SMS, System Center Essentials or ***System Center Configuration Manager*** from Microsoft can be used for this purpose. As Windows PowerShell is available as a normal Windows patch, it can be installed automatically using generally known command line parameters. However, please note that there is a separate version of Windows PowerShell for each version of Windows. You must also take into account whether you are dealing with a 32bit or 64bit architecture, which means that you may need to create several software packages depending on the environment.

To save you from typing out the links for the exercises in this booklet, all links are listed in a file called ***Link.TXT***, which can be found in the files connected to the workshop. These links can be copied directly into your browser. You will also find a file called ***Master.TXT***, which contains longer scripts so that you do not have to type these out either.

A1: Download Windows PowerShell and install it on your virtual test system. To do this, copy the Windows PowerShell download link into Internet Explorer in your virtual environment (add the page http://downloads.microsoft.com to the list of trusted sites when you are asked, in order to execute the download. You will find all the links in the aforementioned text file in the ZIP file connected to this booklet):
http://www.microsoft.com/downloads/details.aspx?FamilyId=10EE29AF-7C3A-4057-8367-C9C1DAB6E2BF&displaylang=en

Call the EXE file with parameter /? by entering the command ***Run…*** or by means of a normal CMD input prompt. What are the installation parameters? What is the syntax for fully automatic installation with no user interaction?

Now carry out fully automatic installation on your virtual test system.

B Security in Windows PowerShell

In the default setting, Windows PowerShell does not execute any scripts. This function must be explicitly activated by the system administrator. Various policies are permitted for activation, which are all connected to the signing of scripts. There are two cmdlets involved in activation: ***get-executionpolicy*** and ***set-executionpolicy***. Using ***get-executionpolicy*** queries the current settings. There are four security levels:

Policy value	**Description**
Restricted (default)	No scripts are executed
Allsigned	Only signed scripts are executed
RemoteSigned	Locally created scripts are permitted, but all other scripts must be signed
Unrestricted	All scripts are executed

B1: Check the Windows PowerShell settings on your system by calling the relevant cmdlet in PowerShell.
Following this, search for the correct key and value in the registry. To do this, use Windows PowerShell or ***regedit***. An introduction to registry navigation is provided in the first workshop book. Tip: ***PowerShell*** is a piece of ***software*** from ***Microsoft*** installed on the ***local PC***. From there, refer to key ***1/ShellIDs*** and display the item properties for ***Microsoft.PowerShell*** (the cmdlet ***get-itemproperty*** *key name* should help). Is there a suitable entry for the Windows PowerShell ***ExecutionPolicy***?

To change the setting for the ***ExecutionPolicy***, a system administrator must call the command

Set-ExecutionPolicy new-value

The four values from the table above are available for selection.

B2: Use the correct cmdlet to set the Windows PowerShell ***ExecutionPolicy*** to ***RemoteSigned*** for the purpose of these exercises. Check the settings using the appropriate cmdlet. Now look at the registry key again. Is there now a suitable entry? What is this entry called, and what value does it bear?

In practice of course, it is not really feasible to set the ***ExecutionPolicy*** manually. Microsoft offers help in this area in the form of a group policy administrative template, which can be downloaded from Microsoft free of charge.

B3: Create a new group policy using the Windows PowerShell template in order to set script security to ***Unrestricted*** for all systems in your domain. To do this, download the Microsoft group policy administrative template and install it on your server. The download link for the template is: http://www.microsoft.com/downloads/details.aspx?FamilyID=2917a564-dbbc-4da7-82c8-fe08b3ef4e6d&DisplayLang=en

Tip: Following download and installation, launch the menu item ***Active Directory Users and Computers*** located under ***Administrative Tools*** from the Start menu. In the newly opened MMC, click on your domain, ***Contoso.local***, and select the item ***Properties*** from the ***actions menu.*** Select the menu item ***Group Policies*** and click on ***Edit***. This opens the Group Policy Object Editor. Click on ***Computer Configuration Administrative Templates*** and select ***Add/Remove Templates*** from the actions menu. Click on the ***Add*** button, select the previously installed Windows PowerShell group policy administrative template (normally located under ***C:\Program Files\Microsoft Group Policy***) and then click on ***Close***. Under ***Computer Settings / Administrative Templates / Windows Components***, you will now find the item ***Windows PowerShell***. Activate the group policy, and set the value to ***Allow all scripts***. Finally, update the group policies on your server by entering the command ***gpupdate /force*** in Windows PowerShell.

In this example, you have changed the standard group policy for the domain. In practice, you would create a new group policy that summarizes the Windows PowerShell settings (and perhaps other settings as well), and then apply this guideline to the appropriate computer group.

B4: Check the Windows PowerShell security settings using the appropriate cmdlet. Change the security settings to another value using the appropriate cmdlet. What answer do you get? Now look at the registry key again. What value does the key have? And what does the cmdlet output?

Now look at the group policy template. To do this, open the ADM file (normally located under ***C:\Program Files\Microsoft Group Policy***) in Notepad. Which registry key is mentioned here? Now open ***regedit*** and take a look at both keys. Change the group policy value as described in **B3**, then check whether a registry entry changes and, if so, which one changes. Which key therefore takes precedence over which?

In this chapter, you have seen that Windows PowerShell is a shell that can be quickly and easily rolled out on your systems, and that its security settings can be securely and centrally managed via the Active Directory. You can, for instance, define whether only signed scripts are demanded in the entire environment, or only on critical back-end systems, while a lower security level can be used with no problems on developer PCs or in test environments. This means that the first version of Windows PowerShell already meets all the necessary requirements for live use in various environments. In the following chapters, individual possibilities for using PowerShell will be presented. Although we cannot by any means cover all areas, we hope we will spark your interest to experiment a little yourself.

C Exercises with files

In this block of exercises, we will be working in the file system, as is often the case for server administrators. We will begin with exercises from the first booklet in this series, "*An Introduction to Windows PowerShell*". As detailed information concerning exercises **C1** and **C2** is available in the first booklet, we won't go into further detail here.

C1: Copy the practice files from the ZIP file connected to this booklet into the directory ***C:\Downloads\Files*** created on your test system. Then, create a script that sorts out the "chaos" by first creating a subfolder for each file type before automatically moving the files to the correct directory. Detailed information on this step can be found in the first booklet in the series. You can either take the solution directly from there or work it out yourself. However, do not create the script interactively in Windows PowerShell this time, but instead in Notepad to start off with. Add your own detailed comments to the script. Comments are introduced with a hash sign ***#*** in Windows PowerShell scripts, and continue to the end of the current line. Save the script under ***C:\Downloads\Scripts\C1.ps1***, and then execute it in the shell.

C2: Delete the "write-protected" attribute from all files in the folder ***C:\Downloads\Files*** (including subfolders). Again, do not execute the necessary commands for this interactively in the shell. Instead, write another script, ***C2.ps1***, in Notepad. Avoid error messages by factoring out directories, as directories have no ***IsReadOnly*** property. This time, however, do not launch the script from Windows PowerShell – instead, use the classic CMD.exe command prompt from a new batch file, ***C2.cmd***. Tip: Create a batch file that contains the correct path to the script ***C2.ps1*** (the correct path is important and must ALWAYS be specified in PowerShell scripts, where necessary even as "**.**"). The call is as follows:

```
powershell.exe   scriptpath\name.ps1
```

In this exercise, you have learnt how to indirectly call Windows PowerShell scripts. Using this technique, you can execute Windows PowerShell scripts as login scripts or system tasks, or using the software distribution method of your choice, even if this does not know anything about Windows PowerShell. Try to discover other Windows PowerShell command line parameters by calling Windows PowerShell using the switch ***/?***

In the next exercise, we will attempt to create a type of intelligent storage solution using Windows PowerShell. This solution is not intended to compete against commercial solutions. Instead, the aim is to show what you can do using intelligent scripts. How to proceed: Use a script to search a server share for files that have not been used for some time. Then, move these files from the source share (e.g. your expensive SAN with daily backup) to another share (your "archive server"). To avoid confusing end users, also create a link on the source share that directs users to the file on the target server, should someone still wish to access the moved file at some point. As mentioned before, this script solution cannot and is not intended to compete with commercial archive solutions. It is simply intended to present an alternative scripting technique, that's all!

To enable the script to be used as flexibly as possible, the paths to the shares are not coded directly in the script, but are instead transmitted via parameters when the script is called. The same applies to the time interval of the last call, in order to obtain as high a level of flexibility as possible. When the script is called, the script parameters are simply appended to the end (…***myscript.ps1 Parameter1 Parameter2*** …) and separated by means of spaces. The predefined variable field ***$Args[]*** can then be used to access the individual parameters in the script. For instance, if the call is as follows:

Myscript.ps1   hello   17

then the variables would be **$Args[0] = hello**   and   **$Args[1] = 17**.

For the following exercise, you must make a few preparations in your virtual environment. First, create two shares. The source share should point to ***C:\Downloads\Files*** and can be called ***Files***. Now, create a second directory called ***C:\Downloads\Target*** and simply name the share ***Target***. Ensure that your account has not only read access (default value) but ***Full Access*** rights to both shares.

As all the practice files were moved in the previous exercise, the ***LastAccessDate*** attribute cannot be used directly. Using the script ***C1-prepare.ps1*** (included in the practice files) sets the ***LastAccessDate*** for certain files back by one or two years. Therefore, execute this script beforehand. The script source code can be found in the annex to this booklet.

C3: Write a script that is called with three parameters: source share, target share and last access date. Check if the script was called with three parameters. If not, cancel the script, indicating the correct call. Then check whether both of the first two parameters are really existing shares (file paths). Tip: The cmdlet ***test-path*** will help you here. If you obtain the return value ***$False,*** simply cancel (indicating the correct call?). Verification that the third entry is a figure will take place at a later point in time, as the required troubleshooting measures have not yet been introduced in this workshop. Now identify all files with a ***LastAccessTime*** value older than (*Today – Parameter3* (in months)). Generate a list of the files with file names, paths and the last access date. Check the results for the following parameters: 6, 13, and 25 months. **Important**: To successfully compare date values, you must convert them into a floating point value using the method ***.ToOADate()***. These figures can then be compared simply using ***–gt*** or ***–lt***.

Now, we've almost found the solution to the problem.

C4: Go through the list of all the original files and move them if necessary. However, before moving the files, ensure you obtain the NTFS permissions by using the command ***get-acl*** in a variable. After moving the files, assign these permissions to the target files again using ***set-acl***.

Of course, in Windows PowerShell you can not only copy ACLs (using ***get-acl*** and ***set-acl***) but also generate them. The easiest way to do this is via .NET libraries. To avoid jumping too far ahead here, these tasks are described in the .NET section and we will come back to them later.

The last thing that needs to be done is to create the link in the source share to the new file in the target share. There are various methods for creating a link. Firstly, the COM object ***WScript.Shell*** can be used. Another option is using the free tool ***xxMkLink.exe*** from ***Pixelab Inc.***, which is described and available for download at http://www.xxcopy.com/xxcopy38.htm. This tool's advantage is its ease of use in comparison with the shell object. For our purposes, it is sufficient to call ***xxMKlink.exe*** with two parameters:

xxMklink (path to new link but without .lnk ending) (original file with path)

C5: Download the tool xxMkLink.exe from http://www.xxcopy.com/xxcopy38.htm, and copy the tool into your script folders. Then, move the practice files into their original directories and reset the access data using the ***C1-prepare.ps1*** script. Now build on your solution from **C4** by creating the link to the moved file. An appropriate task sequence would be as follows:

- Save the original ACLs and the source path
- Move the file to the target folder
- Restore the ACLs
- Create a link in the original path to the new path of the target file
- Ensure link access rights to the link (restore to the original ACLs)

Should you be dissatisfied that the link was not created using a PowerShell cmdlet but instead using an EXE, take a look at ***section K*** as well. In the ***PowerShell Community Extensions***, you will find an appropriate cmdlet (***new-link***) that works in the same way as ***xxMKLink.exe***, therefore permitting a solution to be found using PowerShell alone.

We have now reached the end of our examples for working with files. If you're looking for more ideas, consider the answers to the following questions (suggested solutions can be found in the annex under **X1** and **X2)**:

Create a report that lists the following statistics concerning a server, share, etc.:

- The number of files per file type and the overall size for each type
- List of the top 10 file owners based on their entire volume

Similar reports can also be obtained using the file server functions of Windows Server 2003 R2, but you should try it out here just using what you have.

The aforementioned archive solution has a disadvantage, namely that all files end up in the same target folder. To lend a clearer structure to the archive directory, you can create subfolders, perhaps one folder per source share, and then continue expanding the rest of the directory structure as required. However, a practical example can clarify this much better. \\Server1\Files and \\Server1\data\Project become \\Archivserver\Server1\Files and \\Archivserver\Server1\data\Projects

Can you also incorporate this extension into your solution script for **C5**?

D Working in the registry

Your system registry is available like a normal drive in Windows PowerShell. This enables you to use scripts in the registry similar to those used for the file exercises. However, instead of files, the registry contains keys and their values. If you wish to display keys, you can do this directly using the cmdlet ***get-childitem***. You can display the values using the cmdlet ***Get-childitemproperty Schlüssel***.

These methods can be used to display registry values in a new way, which we first want to do interactively.

D1: Go to the registry path ***HKLM\Software\Microsoft*** in Windows PowerShell and choose to output only the keys beginning with W. Tip: Use the same command as for files in the file system!

D2: This method also works for subfolders, etc. Display all keys that begin with W and end with R, and second-level subkeys that also begin with W. Tip: This sounds a lot more complicated than it actually is.

A new key is created using the cmdlet ***New-Item***. For instance, to create a new entry in the ***HKLM\Software*** key for a Contoso-specific application, the command would be:

```
New-Item –path "HKLM:\Software" –Name "Contoso"
```

Now, only the required entries and their values are still missing here. As already mentioned, this concerns an ***Itemproperty*** that is created using the cmdlet ***new-itemproperty***. A new string entry for the above application would be created as follows:

```
New-itemproperty –literalpath "HKLM:\Software\Contoso" –Name "Appname" –Value "Contoso
CRM" –type string
```

D3: Create a new registry key for your own software, and write at least one value for this key to the registry. Check the result using regedit. Once you have done that, delete the value and the key using the appropriate cmdlets. Tip: These cmdlets begin with ***remove-***…. You can also use the command ***DEL***, which you will be familiar with from the classic command prompt.

Finally, you can put what you have learnt into practice. To do this, list the autostart entries for the RUN key and write these to a file. Once you have analyzed the syntax of the entries, create a new key that automatically launches the Notepad application. The autostart key is ***HKLM\software\Microsoft\Windows\CurrentVersion\Run***

D4: Launch Notepad automatically on your system using a PowerShell script.
Tip: Output the existing entries in a file first in order to understand their syntax before creating a new entry. Check the entry by logging in again, starting a remote session or similar.

E Exercises using .NET and WinForms

.NET WinForms allow you to display script outputs in individual windows. One example is the output of Windows services and their relevant statuses. In the following exercises, we will be repeating possible outputs in HTML format, and will be extending them by outputting them in table format in a separate window.

E1: List all services with their name and status as a HTML page. This exercise is detailed in the first booklet, and is included here simply to refresh your memory. However, assign the output file the same name as your test computer (e.g. ***Computer5.html***). Tip: The name of your computer can be obtained, for instance, by using the WMI object ***Win32_Computersystem***. A detailed description of how to use WMI objects is also provided in the first booklet.

You can nicely use HTML for reports, but for representing data directly, a graphical output, like in an own windows, is more appropriate. This functionality is provided by the .NET framework which is the foundation of Windows PowerShell. Before we take a look at the so called WinForms, we must understand the basics. .NET commands can be used very easily in Windows PowerShell, and the most commonly used .NET libraries are automatically loaded. If you wish to access a more seldom-used library, you must load it first. Special nomenclature and terms are used for .NET. You can look up the meaning of these in any .NET documentation, or for instance on the Microsoft MSDN website. However, with the help of examples, you can quickly pick up how the principle works: all it takes is the library in square brackets followed by two sets of colons:

```
# Loading a .NET assembly:
[System.Reflection.Assembly]::LoadWithPartialName(“System.Windows.Forms”)
[System.Reflection.Assembly]::LoadWithPartialName("System.Drawing")
```

If you wish to use a .NET object, you can generally do this by creating a new instance for the object class. This instance takes over the cmdlet ***New-Object***, indicating the corresponding .NET class. Here, there are very interesting object classes such as Web clients (a type of mini-browser) and e-mail clients, if you wish to send something by e-mail. The script line for the Web client would look something like this:

```
$web = New-Object net.webclient
```

A new object reference has now been created in the $web variable, and can be analyzed using the cmdlet ***get-member*** . For instance, the method DownloadString() exists, which downloads an entire website. The script line for this is:

```
$Seite = $web.DownloadString(“http://blogs.technet.com/chitpro-de”)
```

The example of the RSS Reader can be taken from the first workshop booklet. The only thing now missing is the square brackets []. These brackets are used to define the type of return value that should be chosen in line with the expected output if this is specified. Examples are [XML] or [void], whereby [void] can also mean <u>no</u> output/return value. This also means <u>no</u> error messages! Let's take another look at the example of the RSS Reader:

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

To start off with, another .NET object of the Web client type is created. The object calls a page on the Internet, which is an RSS feed. As we know that this is an RSS feed, we can directly access the RSS properties of the feed: its ***channel*** and the ***Items*** (entries) in the ***channel***. This list of entries is interpreted as XML, which enables formatted output of the ***title*** and ***link*** for the list entries.

However, it is not always necessary to create a new .NET object. Alongside objects, .NET also offers methods that can be helpful in everyday tasks. For instance, .NET commands exist for creating random numbers and random file names, for instance if you wish to create a temporary file during installation. Methods can either be called directly (in the case of libraries that are loaded directly by Windows PowerShell) or by specifying the relevant .NET assembly. Take a look at the following example:

(new-object Random).next(1,50)

The above example calls the .NET method (random) to create a random number between 1 and 50, and returns the random number. To do this, you don't have to create an object first – you can go straight to the method. This is, however, different when creating a random name for a temporary file

[System.IO.Path]::GetTempFileName()

Here again, square brackets need to be placed around the assembly, and colons must also be entered.

Let’s now take a look at how to create own windows, or Winforms as they are called in .NET, by analyzing an example from the first Windows PowerShell workshop. Using the ***System.Windows.Forms*** assembly, we create a Winform object. We add a button object, too and assign the action to close the Winform when we click on the button. The code is Windows PowerShell looks like this:

E2a: Repeat the .NET example from the first Windows PowerShell workshop, which created a window with a single button:

```
[void][System.reflection.assembly]::LoadWithPartialName("System.Windows.Forms")
$form = new-object Windows.Forms.Form
$button = new-object Windows.Forms.Button
$button.add_click({$form.close()})
$form.controls.add($button)
$form.ShowDialog()
```

The example will work, but it’s not very appealing. With few more line of code, we can change this. Try the next example to learn, how you can adjust Winforms to your own needs.

E2b: Extend the example E2a and add a title to your window. In addition, the button should have a label and should fill out the whole windows, too:

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

Beside buttons, the Winforms objects know many more controls you can add, like drawing areas for graphs or pictures and tables you can use for data like in Excel. The next example shows how to use such a table, or ***datagrid*** as they are called in .NET. Take a look at the example and play a little bit with the parameter to better understand their meaning.

E3: Run the following script to get another presentation form for the services of your system. The idea behind is the same to the first Winforms examples. What's new is the way how to fill out the datagrid. Instead of assigning each cell individually the new value, we take advantage of the array handling of .NET and Windows PowerShell. The special @() command actually executes all Cmdlets within the brackets and return their output which we can store in the array variable. This allows you to easily change or add the Cmdlets you want to output in your new datagrid. But be careful: there must not be a space between the @ and the ()!

```
 [void][System.reflection.assembly]::LoadWithPartialName("System.Windows.Forms")
$form = new-object System.Windows.Forms.Form
$DataGridView = new-object System.windows.forms.DataGridView
$Form.Text = "My First Datagrid"

$array= new-object System.Collections.ArrayList
$array.AddRange( @(  get-service | write-output  ) )
$DataGridView.DataSource = $array

$DataGridView.Dock = "fill"
$DataGridView.AllowUsertoResizeColumns=$True
$form.Controls.Add($DataGridView)
$form.showdialog()
```

E4: Try some variations to better understand the @() idea. First, sort the services by their state before you fill the datagrid. And try to output the processes, sorted by the company, too. Hint: simply change the Cmdlets in the ***@()*** bracket construct.

There are still other things to discover in .NET alongside WinForms. However, .NET is too comprehensive to be described here in any more detail. More detailed Windows PowerShell books and all the .NET literature available will provide you with more information in this area and, by means of countless examples, will show you what can be done using .NET. We will limit ourselves to just a few examples here.

Evaluating file types (extensions)

There is a great command in .NET for evaluating file extensions:

[System.io.path]::GetExtension("c:\meinpfad\datei.txt") would for instance return files with the extension ***.TXT***.

E5: List all practice files and return their file extensions. Use the .NET method ***GetExtension*** to do this. Tip: ***GetExtension*** only accepts one file at a time. For this reason, use a loop to process all file objects. Only display extensions that are not "empty". Group the extensions and sort them according to their frequency. Tip: ***($ext –ne "")*** could be a helpful code segment.

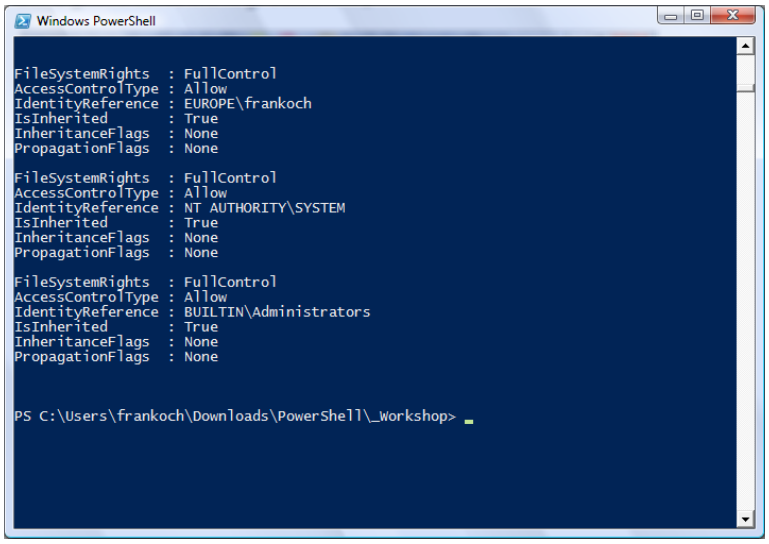
The return value of the sorted, grouped file extensions is an array (provided that several extensions are available). The well-known cmdlet ***Measure-Object*** tells you how many extensions you have altogether, while ***get-member*** gives you the properties of the array.

E6: Only output the extension that is the most frequent.

Setting ACLs in the file system

ACLs have already been described in the section on working with files. The cmdlet ***get-acl*** can provide you with a good overview of who has access to which objects (files, registry values, etc.). For the initial analysis, it is advisable to save the result of ***get-acl*** in a variable, and to then output this as a list ***(…| format-list…)***. It will quickly become apparent that the Access property contains exactly the information you are looking for.

E7: Output the access rights of one of the practice files as a list. To do this, save the ACLs in a variable, ***$MyAcl***. The result should look like this:



```
FileSystemRights  : FullControl
AccessControlType : Allow
IdentityReference : EUROPE\frankoch
IsInherited       : True
InheritanceFlags  : None
PropagationFlags  : None

FileSystemRights  : FullControl
AccessControlType : Allow
IdentityReference : NT AUTHORITY\SYSTEM
IsInherited       : True
InheritanceFlags  : None
PropagationFlags  : None

FileSystemRights  : FullControl
AccessControlType : Allow
IdentityReference : BUILTIN\Administrators
IsInherited       : True
InheritanceFlags  : None
PropagationFlags  : None


PS C:\Users\frankoch\Downloads\PowerShell\_Workshop> _
```

An abbreviation is chosen using .NET in order to set new rights. The following command outputs a list of access constants such as Read, Write, and ListDirectory:

[system.enum]::getnames([System.Security.AccessControl.FileSystemRights])

The required rights can be saved in a variable in order to assign them later to the required file. You'll be pleased to know that all desired rights can be simply listed one after another, separated by commas:

$NewRights = [System.Security.AccessControl.FileSystemRights]"Read, Write, ListDirectory"

If new access rights are to be assigned to a file, this can be done using the method ***AddAccessRights*** of the ACL object, which you obtained using the cmdlet ***get-acl***. The method requires five items of information for this purpose: the name of the user as an NT account, the rights as a ***FileSystemRights*** list, the specification as to whether the rights can be inherited (can be ***None***), and also whether they should be propagated to subordinate objects in the case of a folder (may also be ***None***). The last thing required is the type of access rights rule: ***Access*** or ***Deny***.

A “simple” example looks something like this:

```
$newrights =  [System.Security.AccessControl.FileSystemRights]”Read, Write”
$InheritanceFlag =  [System.Security.AccessControl.InheritanceFlags]::None
$PropagationFlag =  [System.Security.AccessControl.PropagationFlags]::None
$Typ =[System.Security.AccessControl.AccessControlType]::Allow
$ID = new-object System.Security.Principal.NTAccount(“Contoso\Administrator”)
$SecRule =new-object  System.Security.AccessControl.FileSystemAccessRule($ID,
$newrights, $InheritanceFlag, $PropagationFlag, $Typ)
$myACL = get-acl “.\c1-prepare.ps1“
$myACL.AddAccessRule($SecRule)
Set-ACL “.\c1-prepare.ps1t“ $myACL
```

If you find this procedure too complex, you'll be pleased to hear that the trusty ***CACLS.exe*** also works perfectly in Windows PowerShell. The only difference is that you can now, of course, incorporate ***CALCS*** much better into your script pipelines.

Sending an e-mail from a script

There are various approaches for sending an e-mail from a script. Firstly, you can access Outlook, assuming it is installed. However, Outlook's new security mechanisms normally bar access to non-signed scripts in order to keep out viruses and worms. However, a similar result can be achieved using a .NET object. Here, an SMTP connection is set up to an SMTP server for the purpose of sending an e-mail. However, please bear in mind that you must then allow SMTP connections from your clients, a function that I do not normally consider appropriate with regard to keeping viruses, etc. under control. However, for specific clients, or environments in which the SMTP port is open anyway, it may be appropriate.

(To carry out this exercise, you need an SMTP server and an e-mail client to check the result. You can set up both on your virtual test server. You will find a description in the annex of how to do this.)

To send an e-mail, we require several .NET objects of the supertype ***System.Net.Mail***. The object types ***System.Net.Mail.MailMessage*** and ***System.Net.Mail.SmtpClient*** are required as a minimum. The object type ***System.Net.Mail.Attachment*** can also be helpful. These objects can be directly created without the need for loading a further assembly. Using ***get-member*** will tell you more about these objects.

E8: Create a new object (saved in variable ***$Mail-test1***) of the type ***System.Net.Mail.MailMessage*** and display the object properties using ***get-member***. Set the following properties of the object ***$Mail-Test1***

- Sender: “info@contoso.local”
- Subject: “server alarm from ” and the name of the computer
- E-mail body: “Alarm triggered at.....” and the current time; more text can also be added for clarification purposes (everything as a string).
- Recipient: “administrator@contoso.local”

While the e-mail object should be created without any problems and you can change the values for sender, subject and e-mail text, this unfortunately does not work in the case of the recipient. This is due to the fact that the Recipient property (***To***) is only available as a read property and not as a write property. For this reason, the recipient must be specified when creating the e-mail object by, for instance, specifying the sender address followed by the recipient address, in brackets:

$Email = New-Object System.Net.Mail.MailMessage(“Info@contoso.local”, “administrator@contoso.local”)

E9: Create a new object, ***$Mail***, of the type ***System.Net.Mail.MailMessage*** , specifying ***Info@contoso.local*** as the sender and ***Administrator@contoso.local*** as the recipient. Enter the values from exercise **E8** for the subject and the e-mail body. Now create an object, ***$client***, of the type ***SMTPClient*** and also analyse the object using ***get-member***. Set the value of the e-mail server (***Host***) to the name of your server, and call the client method to send the e-mail. Tip: The method adopts the ***MailMessage*** object ***$client.send($mail)*** as a parameter (in rounded brackets).

As this example is very complex for the first time around, here you're allowed to take a look at the solution directly and type it up, so that you can understand it better.

Check your script by using Outlook Express on your test server to look at the e-mail. Sometimes it helps to attach log files or similar to the e-mail, which can be done using the ***Attachment*** object. To do this, create the ***Attachment*** object when writing your e-mail, as well as the ***Message*** and the ***SMTPClient*** object. The syntax for this is a bit different to the other cases, as you must make a direct reference to the file to be attached:

$Attachment = new-object System.Net.Mail.Attachment(“C:\downlaods\test.txt”)

The file is attached to the message using the following command (***$Email*** is the .NET object created previously of the type *MailMessage*):

$Email.Attachment.Add($Attachment)

You now have everything you need in order to send yourself the boot.ini file for your server by e-mail, for instance.

E10: Modify the script to send yourself your server's ***boot.ini*** file as an e-mail attachment.

You can send e-mails not only via .NET but also directly via Outlook if it is installed. Outlook 2007 now finally offers a COM model for this purpose, which can also be addressed very simply by PowerShell. Even though it is not a .NET example, it fits in with this chapter in terms of topic, and is therefore described briefly here. If you don't yet have Outlook 2007, you can download a free trial version from Microsoft. The following example shows how you can output a list of the e-mail subject lines in the inbox:

```
$Outlook = New-Object .-com Outlook.Application
$Inbox = $Outlook.Session.GetDefaultFolder(6)
$Inbox.items | foreach { write-host $_.Subject }
```

E11: Download Outlook 2007 from http://office.microsoft.com/de-ch/outlook/default.aspx and install it on your test system. Configure Outlook for your e-mail server, e.g. the POP3 server of the test environment. Send yourself a few test e-mails and create a few contacts, tasks and meetings. Use Windows PowerShell to display all e-mails in the inbox. Then try to analyse the e-mails in the inbox (***$Inbox.Items)*** using ***get-member***. Note: the call may take a while, as every e-mail is exported from the inbox before being analysed. Can you only display unread e-mails? The solution is really simple:

```
$Inbox.items | where { $_.Unread }| foreach { write-host $_.Subject }
```

Try to find out a bit more about the Outlook object. Use the method ***$Outlook.Session.GetDefaultFolder(x)*** to access the individual functions of Outlook. The individual numerical values for ***x*** are as follows:

Inbox	**6**
Calendar	9
Sent items	5
Outbox	4
Deleted items	3
Contacts	10
Journal	11
Notes	12
Tasks	13

Using ***get-member*** will provide you with ideas of what you can do in each area. For instance, PowerShell scripts write themselves to the Outlook journal with a time and a description for tracking purposes. In ***chapter G***, we'll be combining Outlook with ***Peedy*** to read e-mails out loud. If you've never heard of ***Peedy***, you'll become acquainted with him as well. You'll be surprised how much you can do!

F Exercises with log files and event logs

There are various methods for evaluating log files and event logs. The first example also comes from the first workshop booklet and lists all log files in the Windows directory, looks for the ***Error*** string in them and outputs the log file with the corresponding line number of the first error entry:

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

F1: Modify the example so that all lines with the word “error”, and not just the first line in each case, are output. Tip: Use the help on cmdlet ***select-string*** to find out more about the ***–list*** parameter.

Together with the previous file exercises and the e-mail example, you can now search anywhere in your network for log files with errors, and then either save these centrally or send them to yourself by e-mail. In addition, using the cmdlet ***get-content***, you can choose only to import the lines of a log file up to where an error occurs. The online help on ***get-content*** will tell you how to do this.

There are in principle many ways to access the event log; you can access it directly through PowerShell, or indirectly using COM, WMI and .NET. The cmdlet ***get-eventlog –list*** outputs a list of all event logs in your system. Using ***get-eventlog “Eventlogname”*** provides you with the content of the specified event log. However, as this output can be very long in the case of many event entries, you should restrict the output directly by using the parameter ***–newest xy***, which only outputs the most recent ***xy*** entries.

F2: Output a list of the Windows PowerShell event log. Tip: The name of the corresponding event log can be found by listing all event logs. Only choose to display the last 10 entries. Then, group all entries according to their event ID, and sort the list based on event frequency. Tip: Use the cmdlets ***Group-Object*** and ***Sort-object*** that were presented in the first exercise booklet.

.NET allows you to not only read the event log, but also write in it. To do this, you must first create an object reference to the corresponding object:

```
$a = new-object –type system.diagnostics.eventlog –argumentlist System
$a.source = “Windows PowerShell Labs”
$a.writeentry(“this is my first entry in the system log”,“Information“)
```

F3: Generate a list of all stopped services on your system and enter each service as a separate event in the system event log. Select a suitable event text. To avoid making too many entries, limit the list to the first five services.

You can also make entries in event logs on other computers using the above command. The general syntax for this is, for instance:

```
$a = new-object –type system.diagnostics.eventlog –argumentlist System, servername
```

G Exercises with Peedy

Microsoft Agents are actually already considered to be obsolete. However, a small group of them has survived and, thanks to Windows PowerShell, can be looked at in close detail. The ***MSAgent Peedy*** must be downloaded from http://www.microsoft.com/msagent/downloads/user.aspx#character and installed on your test system in order to carry out these exercises. The installation is nothing spectacular – Peedy is available to you for programming, but you cannot see him yet. The following examples concentrate on Peedy, but in principle also work for the other agents. You can also install voice output depending on your system, if it hasn't already been installed and set up on your computer.

Peedy is a simple COM object that is used here to demonstrate the general handling of COM objects. Alongside .NET and WMI, COM is another, separate dimension that encompasses a lot of variety and power, and which has its own literature and examples. As in the case of .NET, we can only give you a taster here of the possibilities offered by COM.

To contact Peedy, you must first access his object, as in the case of .NET, and create an individual reference to it. ***MSAgents*** have their own syntax, which is as follows:

```
$PC = New-Object -com agent.control.2
$pc.connected = $true
[void]$pc.Characters.load("Peedy","C:\windows\MSAgent\chars\Peedy.acs")
$Peedy = $pc.Characters.Item("Peedy")
```

As you can see, an ***MSAgent*** is initially accessed indirectly via its own ***Control***. The path to the actual character is then assigned to this ***Control***. In the above example, the path to the character is fixed, which is a bad habit, as the Windows directory may of course also be on D: or have a different name. Contary to this, the subpath ***\MSAgents\chars\*** is always correct. In this case, you only need to check whether Peedy exists. You can thankfully also access environment variables from Windows PowerShell. The Windows directory can be found under ***$env:windir***, for instance. Use the cmdlet ***join-path*** to connect to the target path. This cmdlet is capable of creating a valid file path from system variables, file paths and more, which is exactly what we need here:

```
$path = @(Join-path $env:TMP  "my_temp_file.tmp")
```

G1: Improve on the previous example of creating a Peedy object reference by defining the Windows directory using the system variable windir, and adding the path for Peedy. Check whether the Peedy file actually exists. Tip: Use the examples from the file exercises. Following this, you can address Peedy directly using the variables. Analyse the Peedy object using the cmdlet get-member to find out more about its characteristics.

To get hold of Peedy, you may have to lure him out from his hiding place first:

```
[void]$peedy.Show()
```

[void] is specified here to surpress the output of the Peedy methods.. This is highly appropriate for later scripts. However, it makes it more difficult to debug your work at the start. You can therefore choose whether you wish to use [void] or not.

G2: You can move Peedy across your screen by using the following commands and entering figures for x and y. The numerical values correspond to the appropiate screen pixels. However, please take into account that Peedy is also of a certain size: ***[void]$peedy.Moveto(x,y)*** , or, by way of example: ***[void]$peedy.Moveto(10,200)***

If you wish to hide Peedy, the following may come in handy:

```
[void]$Peedy.Hide()
```

Peedy can also talk to you. Try to get him to say the following phrase: "Frank Koch is a great guy from Microsoft." Use the ***speak***() method to do this:

```
[void]$Peedy.Speak("Frank Koch is a great guy from Microsoft")
```

If the task should be performed quietly, let Peedy "think" instead:

```
[void]$Peedy.think("I shall not disturb my master")
```

One reason why ***MSAgents*** became obsolete is because they are so lively. ***MSAgents*** like Peedy just love playing around on your screen. However, each ***MSAgent*** can also do other things as well. You can see in the property ***AnimationNames*** what each character is capable of.

G3: Generate a list of all the Peedy animation sequences. Then pass this list on to Peedy's ***Play()*** method in order to view all animation sequences at once. Tip: First create the list then output it on the screen. Then, using a second command, pass this list on to a loop that plays each animation sequence through once. Please remember here that the animation sequences take a certain amount of time. For this reason, pause the script using the cmdlet ***Start-Sleep*** command for five seconds each time. As some Peedy animation sequences are infinite, stop each sequence once it is complete using the command ***$Peedy.StopAll().***

G4: Create a system monitor that outputs all stopped services as speech using Peedy. To keep this task short, choose five services. Select a suitable animation sequence to capture the system administrator's attention.

Peedy can also introduce your next presentation. Look at the following script for this. To start off with, Peedy is loaded and then moved to a certain position on the screen:

```
# Load peedy as usual; hardcoded Windir, not checking if Peedy exists
$PC = New-Object -com agent.control.2
$pc.connected = $true

[void]$pc.Characters.load("Peedy","C:\windows\MSAgent\chars\Peedy.acs")
$Peedy = $pc.Characters.Item("Peedy")

# Move Peedy to right screen position
[void]$peedy.Show()
[void]$peedy.moveto(100,300)
```

```
# Play some animation to get attention from audience
[void]$peedy.Play("GetAttention")
Start-sleep –seconds 5
[void]$Peedy.StopAll()

# show how polite Peedy can be
[void]$peedy.Play("Greet")
Start-sleep –seconds 5
[void]$Peedy.StopAll()

# spread the word: the session will start soon
[void]$peedy.Speak("The Show starts in 5 minutes")
Start-sleep –seconds 5

# say goodbye to the audience and disappear
[void]$peedy.Play("Greet")
Start-sleep –seconds 5
[void]$Peedy.StopAll()
[void]$peedy.Hide()
```

If you wish to hear the voice output in languages other than English, you can install other languages too. After doing this, all you need to do is tell Peedy what language you want to use by using the ***LanguageID*** number in the Peedy object property of the same name. Please note that many languages are available in both a male and a female voice, meaning you also have to set the ***TTModeID*** value. You will find the “cryptic” values for this purpose on the ***MSAgent*** website under the examples for developers. For instance, if you want a German male or female voice, set the following values:

```
$peedy.LanguageID=1031
$peedy.TTSModeID = "{3A1FB760-A92B-11d1-B17B-0020AFED142E}"  #  --> Anna = Woman
$peedy.TTSModeID = "{3A1FB761-A92B-11d1-B17B-0020AFED142E}"  #  --> Stefan = Man
```

Many thanks to Frank Glattky for investigating these properties of Peedy and telling me about them.

Peedy is also great for defining your own ***Functions***. Let’s take a look at a simple example that you can adapt to suit your own needs. Functions in Windows Powershell are presented as command blocks that can be called with a command, and can thus be easily integrated into other command chains. They are created directly using the command ***function name-of-function { the actual commands are then placed between the curly brackets}***

Here’s a simple example:

```
Function now{ get-date }
```

Each time you enter ***now***, you are now shown today's date. Of course, functions can also execute more complex command chains and be called using parameters:

Function Sum { $Args | $sum = $sum + $_; return $sum}

In this example, several calculations are carried out in the function, and the desired result is then supplied as the return value. Unless otherwise defined, variables remain in their corresponding environment. In the above example, the auxiliary variable ***$sum*** does not exist outside of the function; you cannot access it at the end either – as soon as the function is completed, ***$sum*** vanishes into thin air.

G5: Now create your own function with the name ***Out-Peedy***, which as an argument adopts a text that is then output by Peedy. Tip: Assume that Peedy is not loaded before the function and is therefore not defined. You can therefore take the start of the procedure directly from the other Peedy examples. Also use the ***start-sleep*** command.

Using ***Out-Peedy***, you have learnt how easy it is to define and use functions in Windows PowerShell. You can also expand ***Out-Peedy*** further by allowing a new argument: ***-quiet***. Whenever you call ***Out-Peedy Text –quiet***, output does not ensue via the Peedy command ***Speak***, but instead via ***Think***.

G6: Add the additional argument to your ***Out-Peedy*** function. To make things easy, just check whether there are two arguments and, if so, if the second begins with –q. Everything beforehand is compiled and output as a string, and everything afterwards is ignored. If the trigger –q is used, Peedy should only "think" and not "speak".

To ensure that the ***out-peedy*** function is always available, you can add it to your PowerShell profile. Your profile is automatically loaded when Windows PowerShell is started. There is therefore a personal profile for each user, as well as a PC profile. For this reason, if you want to be able to use a function when using automatically started scripts as well, it should be saved in the PC profile. The profile file is ***Microsoft.PowerShell_profile.ps1*** in the directory ***Own files\WindowsPowerShell*** (or the equivalent depending on your system language and version of Windows). It is loaded when Windows PowerShell is started up. For this reason, try to add ***out-peedy*** to the file if there is one, or create a new file that includes the function ***out-peedy***. Upon restarting Windows PowerShell, ***out-peedy*** should be directly available.

(Please note that script authorizations must be set as required so that the script files can actually be executed!)

The following profile files are available on the systems:

Location of profile file	File name	Refers to
%Systemroot%System32\WindowsPowerShell\v1.0	Profile.ps1	All users, all shells
%Systemroot%System32\WindowsPowerShell\v1.0	Microsoft.PowerShell_ Profile.ps1	All users PowerShell only (not Exchange, etc.)
%userprofile%\Documents\WindowsPowerShell	Profile.PS1	Current user All shells
%userprofile%\Documents\WindowsPowerShell	Microsoft.PowerShell_ Profile.ps1	Current user PowerShell only (not Exchange, etc.)

G7: If you installed Outlook 2007 in ***chapter E***, now use ***out-peedy*** to read all unread e-mails out loud. So Peedy doesn't get hoarse, start off with the e-mail subject lines.

Even though this may be a somewhat hectic solution (as Peedy keeps disappearing), it is impressive to see how Windows PowerShell can implement a solution like this with so little effort. Perhaps you'd like to add another parameter to ***out-peedy*** to stop Peedy from disappearing? ***out-peedy*** can be a very valuable extension to your Windows PowerShell. And, if you don't want to install Peedy on all computers and servers, another ***MSAgent, Merlin***, is always included in Windows. You can call and use Merlin just the same as Peedy:

```
$PC = New-Object -com agent.control.2
$pc.connected = $true
[void]$pc.Characters.load("Merlin","C:\windows\MSAgent\chars\merlin.acs")
$Merlin  = $pc.Characters.Item("Merlin")
```

And, before you ask – yes, Merlin and Peedy can both be active on the screen at the same time and perform different tasks.

H Exercises with the Active Directory

There are various options for working with the Active Directory. Using individual cmdlets, which would be the most user-friendly method, is still not possible. However, we can assume that Microsoft will sort out this drawback in future. Instead of its own cmdlets, Windows PowerShell lets you administer the Active Directory via ADSI. ADSI is an interface protocol that allows access not just to the Active Directory, but also to other directory services. ADSI is very well documented and has been on the market for years, meaning it is not necessary to provide an introduction to it here. If by any chance you haven't heard of ADSI, you can perform a search for ASDI on the Microsoft MSDN website. In the following example, you can see the effort required to create a new organizational unit (OU) in the Active Directory via ADSI for yourself:

```
$objDomain = [ADSI]"LDAP://dc=nwtraders,dc=com"
$objOU = $objDomain.Create("organizationalUnit", "ou=HR")
$objOU.SetInfo()
```

The first line establishes the connection to the directory service and sets the path where the new object is to be created, in this case in the domain nwtraders.com.

In the second line, the object to be created is specified, in this case an OU with the name "HR". Only when the ***SetInfo*** command is given are these changes written to the Active Directory. This is then helpful if you wish to set further properties after creating the object, as is shown below in the example of creating a user:

```
$objOU = [ADSI]"LDAP://ou=HR,dc=nwtraders,dc=com"
$objUser = $objOU.Create(„user", „cn=frankoch")
$objUser.Put(„sAMAccountName", „frankoch")
$objuser.SetInfo()
```

Note that you have to specify all mandatory object properties when creating the user object. Can you analyse the member object using ***Get-Member***, for instance?

If you're not sure what the current directory environment is called, you can query this in a very simple way. By simply entering ***[ADSI]""*** in the Windows PowerShell prompt, the correct LDAP path will be output. You can also save this return value in a variable for later use. ***Get-member*** again lists all the properties that you can use. You will find the LDAP path, for instance, under the property ***distinguishedName***:

```
$dc = [ADSI]""
$dc.distinguishedName
```

The result is, for instance: ***dc=nwtraders,dc=com***

H1: First, create a new OU for human resources (HR) in your virtual environment (***Contoso.local***), and a new group within this OU. The procedure is the same as for creating a user object. This time, however, you must select the "group" object type instead of the "user" object type. Finally, generate two user accounts – one for yourself, and one with the name ***Frankoch***. Please note that a group also has a ***SAM Accountname***!

As soon as you've created an account, you can set your initial password and activate the account. It is important to activate the account, as a new account is always created as a deactivated account by default. The password can be set most easily using the method ***$User.SetPasswort("New Password")*** of the user object ***$USER***. Accounts can only be activated indirectly in Windows PowerShell. To do this, you must use this somewhat cryptic command sequence:
$User.psbase.InvokeSet("AccountDisabled",$false)

H2: Activate both the user objects created in **H1** in your virtual environment, and set an initial password. Use ***get-member*** to check the user objects for other properties, and then try to set these as you wish.

Of course, objects from the Active Directory can also be deleted via the ASDI interface. Instead of the ***Create***, a ***Delete*** is used for this purpose. The following example deletes the user you have just created from the Active Directory:

```
$objOU = [ADSI]"LDAP://ou=HR,dc=nwtraders,dc=com"
$objOU.Delete(„user", „cn=frankoch")
```

No ***SetInfo***() is required for ***Delete***.

Performing a search in the Active Directory

If you wish to alter certain objects in the Active Directory, you first have to find them. The object class ***DirectorySearcher*** (yes, it really does exist!) is available for this purpose, which makes life a bit easier. Here's an example:

```
$ADDomain = [ADSI]"LDAP://dc=contoso,dc=local"
$ADSearch = New-Object System.DirectoryServices.DirectorySearcher
$ADSearch.SearchRoot = $ADDomain
# Definition of the filter: only computer, search for names
$ADSearch.Filter = „(objectCategory=computer)"
$ADSearch.PropertiesToLoad.Add("name")
$results = $ADSearch.FindAll()

# For the result we use a trick to only display the names:
Foreach ($res in $results)
{
        $ADComp = $res.Properties
        $ADcomp.Name
}
```

The result is now located in the array ***$results*** and can be further used from there. The Filter property is a typical ADSI search expression that is based on ADSI nomenclature. This nomenclature is very unusual and different to that used in Windows PowerShell, so we won't delve into the topic further here. I will only say that, if you want all properties to be loaded, and not only the name, replace the line

$ADSearch.PropertiesToLoad.Add(„name“) with ***$ADSearch.PropertiesToLoad.AddRange()***

However, if you would like to load only certain properties, you must add each of these properties using ***PropertiesToLoad.Add(name_of_property)***.

As a search category, you can also use users for instance, instead of computers.

H4: Search for all users in the Active Directory on your test system. Get the system to output the following properties: ***name, description***. Please note that the ADSI query is case-sensitive, so ***name*** and ***description*** must be written in lower-case letters.

Working with partner add-ons: Quest

Working with the free Quest tools is considerably easier than working with ADSI. Quest is a close partner of Microsoft in many areas, and offers practical add-ons to Microsoft solutions. Quest offers a very interesting, free add-on for Windows PowerShell, which contains cmdlets for managing the Active Directory. Quest showed foresight in naming all their cmdlets ***…-QAD…*** (instead of ***–AD***, i.e. ***get-QADUser*** instead of ***get-ADUser***), thus enabling Microsoft to close the existing loophole in a later version of Windows PowerShell. You can download the Quest tools at http://www.quest.com. Here, you will also find a detailed description of the tools for download. The Quest cmdlets are available as a ***PSSnapIn***, which happens when the specific Quest shell is loaded from the Start menu entry. If you want to add the ***PSSnapin*** to the normal Windows PowerShell, all you need to do is call the following command in Windows PowerShell:

Add-PSSnapIn Quest.ActiveRoles.ADManagement

The PowerShells for Exchange 2007, Operations Manager and similar products are individualized in the same way; their individual Cmdlets can be added to the normal Windows PowerShell as shown.

H5: Install the Quest add-ons for Windows PowerShell from http://www.quest.com. Launch the new Quest shell from the Start menu and display all new cmdlets. Tip: Quest commands always contain a ***QA*** in the cmdlet name. Then, load the Quest ***PSSnapin*** in the normal Windows PowerShell and attempt to display all new cmdlets. How can you automatically load the Quest ***PSSnapins***? Tip: Think back to the profile files in the previous exercises.

Quest tools can also be used to choose the precise domain controller you wish to connect to. You will find details on this in the extensive Quest handbook. We only take a closer look at a few functions here, such as creating a user, changing group affiliations or changing user attributes. A new user is created using the new Quest cmdlet ***New-QADUser***. Display the help for the cmdlet for creating a new user in order to take a closer look at the many possibilities available. In any case, the following syntax is enough to create a user:

new-QADUser -name 'user1'
-organizationalUnit 'OU=companyOU,DC=company,DC=com'
-SamAccountName 'user1'
-password 'P@ssword'

You will already notice at first glance that the syntax appears simpler (more similar to PowerShell) than the ADSI/LDAP syntax from the previous exercises.

To modify an existing user, you must choose the relevant user object and specify the new attributes. The ***Set-QADuser*** serves this purpose. Here is an example:

set-QADUser 'CN=John Smith,OU=CompanyOU,DC=company,DC=com'
-description 'Sales person'

The user object can also be selected directly via its *Domain/User* property:
set-QADUser -identity 'company\jsmith'
-city 'New York'
-description ''
-password 'P@ssword'

You will find a list of the attributes that you can change in the descriptions on Quest. Here is a selection for you:

User attribute	PowerShell cmdlet syntax
company	-Company
description	-Description
department	-Department
displayName	-DisplayName
facsimileTelephoneNumber	-FAX
givenName	-FirstName
sn	-LastName
mobile	-MobilePhone
info	-Notes
physicalDeliveryOfficeName	-Office
password	-Password
telephoneNumber	-Phone
samAccountName	-SamAccountName
st	-StateOrProvince
wWWHomePage	-WebPage

H6: Create a new user using the new Quest cmdlet ***New-QADUser***. The user should be created in the HR organizational unit with the name **Frankoch3**. You can choose the password yourself. Create a new group and add the user Frankoch3 as a member of the group. Tip: Look at the syntax for the cmdlets ***New-QADGroup*** and ***Add-QADGroupMember***.

Finally, we will quickly look at how the process of creating multiple users can be easily automated using Windows PowerShell. Ideally, a CSV file containing the relevant user data would be available, for instance. The cmdlet ***Import-CSV*** can be used to import a CSV file. The ADSI or Quest functions described are then used to create the user objects.

H7: In the sample files in the H folder, you will also find a CSV file that you can use for a bulk import of several users into the Active Directory. Load the file using ***import-csv H7-Users.csv*** and work through it line by line in order to create all accounts in a new OU in the Active Directory.

As you see, working in the Active Directory is not as complex as it first looked using ADSI. In particular, the Quest tools make creating users and groups child's play. Alongside the Quest tools, there are also options using .NET classes and an interesting approach via a Windows PowerShell Provider from the ***PowerShell Community Extensions***, which is introduced in ***chapter K***. This provider incorporates the Active Directory as an additional PowerShell drive (in the same way as the registry is also incorporated as a "drive") to enable you to navigate in the Active Directory in the same way as in the file system: with ***cd ..*** through the OUs, or ***DEL **** to delete users. And as if those new possibilities weren't exciting enough, further examples are presented in ***chapter K***.

I Reports using Office Web Components and PowerGadgets

To generate reports directly from Windows PowerShell, the much-used option of generating a HTML output via the cmdlet ***convertto-HTML*** exists. However, this cmdlet cannot be used to create more graphically appealing reports. The Office Web Components offer an additional way to simply produce elegant reports. The Office Web Components can be downloaded free from Microsoft. However, there are certain licencing conditions for using them that you should take a closer look at. To cut a long story short; if you want to use the output in a static way, this is possible. However, if it needs to be possible for readers to make active changes to the reports, each user must have a valid Office licence.

To create the reports, the Office Web Components must first be installed on the system concerned. To find out more about the Office Web Components and download them, visit http://www.microsoft.com/downloads/details.aspx?familyid=7287252C-402E-4F72-97A5-E0FD290D4B76&displaylang=en

The Office web components are used like COM objects. Because they roughly have the same feature set like Excel, the Office web components are as complex as Excel. Therefore we limit ourselves here and only use basic bar charts. If you want to dig deeper, you should use the extensive documentation of the Office web components in the internet at the Microsoft MSDN web pages.

The following script shows the possibilities of Office web components. The first lines initialize the COM objects and create an empty bar chart diagram. To add and store the values we want to print out, we use two arrays called ***values*** and ***categories***.

```
# Author Stephen Hulse shulse@hotmail.com
$categories = @()
$values = @()
$chart = new-object -com OWC11.ChartSpace.11
$chart.Clear()
$c = $chart.charts.Add(0)
$c.Type = 4
$series = ([array] $chart.charts)[0].SeriesCollection.Add(0)

Get-Process | Sort-Object cpu | Select-Object processname,cpu -last 10 | foreach-object {
        $values += $_.cpu * 1
        $categories += $_.processname }

$series.Caption = "chart"
$series.SetData(1, -1, $categories)
$series.SetData(2, -1, $values)

$filename = (resolve-path .).Path + "\chart2.jpg"
$chart.ExportPicture($filename, "jpg", 800, 500)
invoke-item $filename
```

Hard coding the Cmdlets we want to output is not very flexible. We need to change the code a little bit to allow the creating of the values outside the code block for the diagram. The next example shows such a generic diagram script. It uses the Office web components in exactly the same way, but allows the values to be handed over as parameters. The code below is commented in the solution chapter in the appendix of this booklet.

```
param($xaxis, $yaxis)

begin {
$categories = @()
$values = @()
$chart = new-object -com OWC11.ChartSpace.11
 $chart.Clear()
 $c = $chart.charts.Add(0)
 $c.Type = 4
 $c.HasTitle = $true
 $c.Title.Caption = "Chart generated on $(get-date)"
 $series = ([array] $chart.charts)[0].SeriesCollection.Add(0)
}

Process {
$categories += $_.$xaxis
 $values += $_.$yaxis *1
}

End {
 $series.Caption = "chart"
 $series.SetData(1, -1, $categories)
 $series.SetData(2, -1, $values)
 $filename = (resolve-path .).Path + "\chart.jpg"
 $chart.ExportPicture($filename, "jpg", 800,  500)
 invoke-item $filename
}
```

This generic script can now be called within any Windows PowerShell code segment. The usage looks like this and calls the scripts with the 2 parameters (in the right order!) to generate the bar chart:

```
get-process |.\out-chart.ps1 processname  handles
```

I1: Use the sample script ***out-chart.ps1*** to create a graph of the 10 processes that take up the most CPU time, in descending order. To do this, download the Office Web Components from http://www.microsoft.com/downloads/details.aspx?familyid=7287252C-402E-4F72-97A5-E0FD290D4B76&displaylang=en and install them on your test system. You will find a brief description of the source code for ***out-chart.ps1*** in the annex. Tip: If you want to work more using the Office Web Components, please also make sure you read the detailed descriptions and examples on Microsoft's MSDN website.

As an alternative to Office Web Components, you can also use the commercial solution PowerGadgets, which is a great deal more powerful and flexible than the ***out-chart.ps1*** script. You can download a free trial version of PowerGadgets at http://www.powergadgets.com/. After installation, you will find a new folder for the PowerGadgets in the Start menu with a link to Windows PowerShell, which loads the relevant ***PSSnapin*** for the PowerGadgets. Excellent documentation for the PowerGadgets, including training videos, is available on the homepage, so we won't go into the many possibilities here. Instead, we will simply repeat the example from **I1**, the only difference being that we will use the cmdlet ***out-chart*** instead of ***out-chart.ps1***.

I2: Download the free trial version of PowerGadgets and install the tool in your virtual test environment. You will find the download link on the homepage http://www.powergadgets.com/. Start Windows PowerShell from the PowerGadget folder and observe which ***PSSnapin*** is loaded. Also try loading this ***PSSnapin*** in a normal Windows PowerShell.

To view the cmdlets for a specific Windows PowerShell ***PSSnapin***, you can use the command ***Get-command –pssnapin Kompletter-PSSnapin-Name.***

I3: Display all new cmdlets for the PowerGadget tool. You will see that the PowerGadgets are capable of all possible types of graphical output, such as bar charts and vector diagrams (for percentages, etc.) and even maps! Use the solution from **I1** to create a graph of the 10 processes that take up the most CPU time, in descending order. However, create the graph using the PowerGadget cmdlet ***Out-Chart*** instead of ***out-chart.ps1***. So that ***out-chart*** knows which data is to be output, select the two values you are interested in beforehand, in this case ***Processname*** and ***CPU.***

The enormous capabilities of PowerGadgets should not be underestimated, as the graphical output can be a tremendous assistance in everyday work. The following examples clearly demonstrate this. First, copy all test files from the previous exercises back into one single directory. Would you be able to say without hesitation how many files there are of each type? PowerGadgets even lets you see how many there are and lets you quickly find the largest files in the folder.

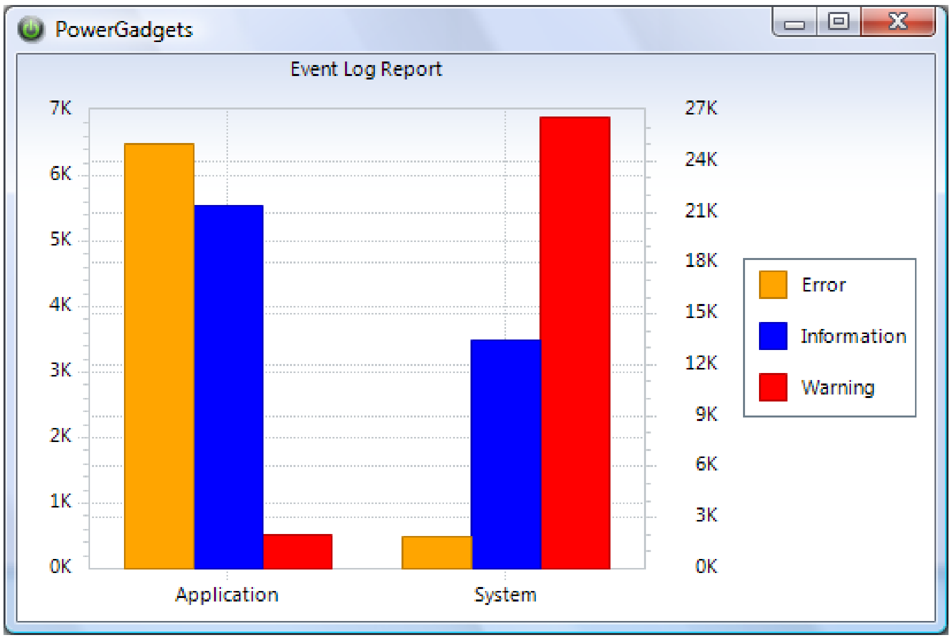
I4: Create a bar chart that shows all file extensions and the number of files with each extension. Analyse the diagram a little using your mouse. Tip: The solution is very quick, and the cmdlet ***Group-Object*** will help.
Now draw up a list of the ten largest files and output this as a chart as well.

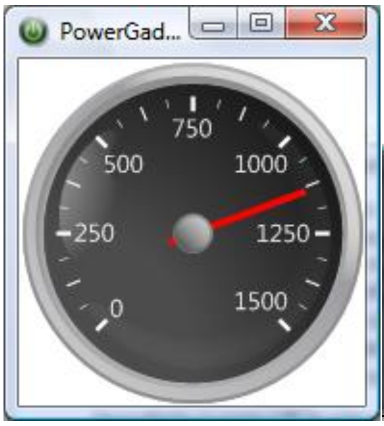
Even if it looks so to start off with, the PowerGadgets must not necessarily represent the end of the pipeline. For instance, values can be passed to the PowerGadgets intially in order to be selected from there by users by double-clicking. This object is returned to the pipeline, where it can be processed further. We will take a closer look at this in the next exercise.

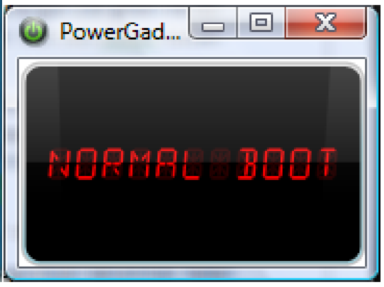
I5: Take your solution from **I3** and, as the simplest example, take the return to the pipeline and add on the output from the object at the end as a new command. Tip: Simply add***| Foreach { $ }*** to directly output the object that you clicked on previously in the chart.

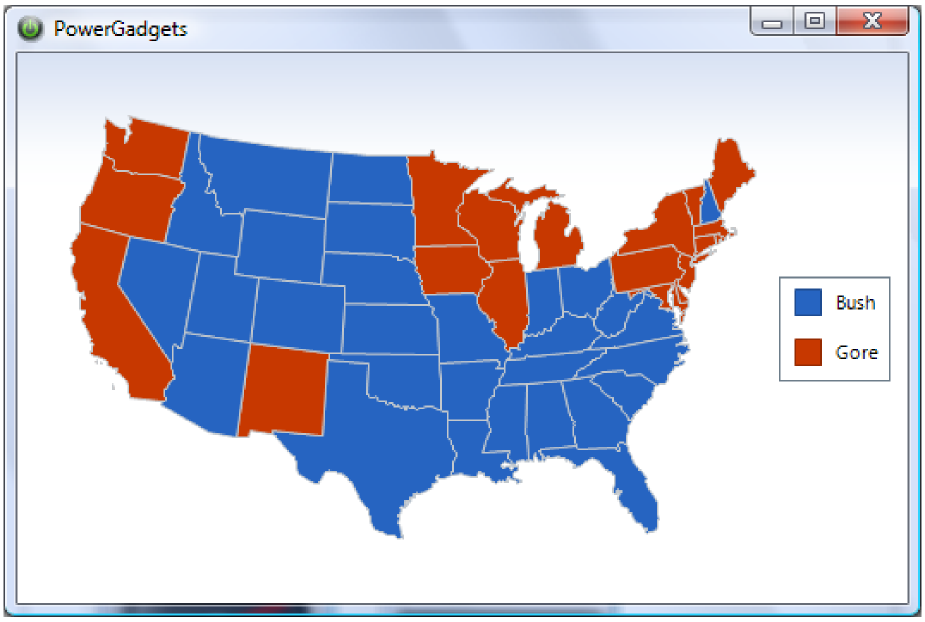
You can now form a combined solution from **I5** and **I4** that initially lists all file types for you. The list of the largest files is then shown for the file type you select, and files you select can be deleted with just one mouse-click. Windows administrators tend to really like this combination of shell and mouse, whereas just thinking about it may give Unix, Linux and Host fans the shivers.

PowerGadgets are much more powerful than I am able to show here. For this reason, please do take a look at the examples that come directly with the PowerGadgets. You will find these in the Samples folder (normally under ***%ProgramFiles%\PowerGadgets\Samples***). These examples also show the options for creating maps, bar charts, vector diagrams and digital displays. Here are a few sample outputs:


PowerGadgets
Event Log Report
7K
6K
5K
4K
3K
2K
1K
0K
27K
24K
21K
18K
15K
12K
9K
6K
3K
0K
Application
System
Error
Information
Warning


PowerGad...
0
250
500
750
1000
1250
1500


PowerGad...
NORMAL BOOT


PowerGadgets
Bush
Gore

J Performing evaluations with WMI

The first workshop booklet provided a short introduction to the world of WMI, so we won't go into it here. Instead, a few exercises are suggested to help you become better acquainted with the almost infinite possibilities of WMI. If, in your work as an administrator up to now, you've managed to steer clear of WMI, please tell me how – I'd like to know how you've managed it, and why you would want to. After all, WMI really does make the everyday work of a Windows administrator much easier.

To give you a first impression of how powerful WMI is, list all WMI classes using the simple PowerShell command ***Get-WMIObject –list***. You should assume the task list will be extremely long, but give it a try anyway.

J1: List all WMI classes of the type Win32 that begin with "n". Output the results as a table that only lists the names of the classes. How many classes of this type are there?

In the results list from exercise J1, you will also find the class for network adapter configuration. All network adapter objects can be found in this class. Using the parameter ***–Filter Eigenschaft=Wert***, you can now choose certain objects. For instance, to get all network adapters that have an IP address, you can use the following syntax:

```
Get-WmiObject -Class "Win32_NetworkAdapterConfiguration" -Filter IPEnabled=True
```

J2: Now list all network adapters that have been assigned an IP address. Output the results as a table containing the index, the IP address and the adapter description. Now also take a look at the WMI class for the network adapters themselves (not their configuration). Use ***get-member*** to output the properties and methods of the adapters. Can you find a method for activating or deactivating adapters? If you do not find the WMI class straight away, try executing this WMI query on your host computer as well (i.e. not in the virtual test environment).

As WMI not only works in Windows PowerShell, and as the WMI syntax is in principle no different to that of VB, I will not list the classic WMI examples here. Instead, I'll be giving you an example that was also described in the first booklet. In this example, Windows PowerShell is used to compare the battery installed in Dell notebook computers against a recall list published on the Internet of batteries at risk of explosion. At the time of writing, Dell unfortunately used the wrong certificate for the website, so you have to initially open the website manually in the browser and confirm that you wish to open it despite the wrong (= unsecure) certificate. If you do not perform this step, the.NET object ***webclient*** will not perform it later in the script.

The example also shows the possibilities offered by ***Regular Expressions***, one of the most powerful ways of comparing expressions of all kinds with a chosen format. Let's take a closer look at the example ***[A-Z0-9][A-Z0-9][0-9][0-9][0-9]***:

- Each set of square brackets stands for a character.
- The items in the square brackets indicate the value that this character can have; for instance, A-Z can be any letter, 0-9 any number, and A-Z0-9 a letter or a number.

Dell marks its batteries with 5 characters (5 sets of square brackets). The first and second characters may be letters or numbers. In contrast, the last three characters must be numbers. The result is the following regular expression:

[A-Z0-9][A-Z0-9][0-9][0-9][0-9]

So, the script for comparing the batteries is as follows:

- First, a Web client object is created and the Dell website is launched.
- The regular expression for Dell batteries is created (the brackets we just looked at now).
- A search is performed on the website for the expression, and you are then informed how many different batteries are listed (upon writing this booklet there were 38).
- Then, the names of the batteries available are output and compared with the list of names. If any matches are found, an alert is triggered.

J3: Execute the following lines of code:

```
$WC = New-Object net.webclient
$Global:BatSeite = $wc.downloadstring(“https://www.dellbatteryprogram.com/“)

$R = [regex]”&nbsp;([A-Z0-9][A-Z0-9][0-9][0-9][0-9])”
$Matches = $R.Matches($Global:BatSeite)
Write-Host “Anzahl gelisteter Akku-Typen: $($Matches.Count) “

$Global:BatterieListe = $Matches | ForEach { $_.Groups[1].Captures[0].value }
Get-WMIObject –Class Win32_Battery | ForEach { $BatName = $_.name; break }

If ((select-string –pattern  $BatName  -InputObject  $BatterieListe –Quiet)  -eq $True) {
   BatterieListe | Where {$BatName –match $_ } | Foreach {
     Write-Warning “Betroffen:Akku $Name – Übereinstimmungen: *$_*`n“ } }
```

Similar exercises can also be used for other notebook manufacturers. The only reason I have used Dell here is because I am personally a big fan of Dell, and I still enjoy using my private Dell notebook – I don’t want you to get the wrong idea from this example!

Regular expressions are extremely powerful but also a topic in their own right, so we’ll leave it at this simple example here. However, you will find detailed descriptions of everything alongside many more examples of usage on both the MSDN website and in literature on the subject.

K Partner add-ons: community tools

As Microsoft published Windows PowerShell, many people were impressed straight away, as not only had a powerful shell been launched – this shell also had interfaces to expand it! So, alongside professional and commercial add-ons such as PowerGadgets, developer communities also quickly developed add-ons. One of the most powerful add-ons that are provided free of charge comes from one of these communities. You will find the ***PSCX*** (***PowerShell Community Extensions***) directly under http://www.codeplex.com/PowerShellCX, where you can read more about the development. Here, you can also make your own contributions and add-ons available to the community. In the PSCX, you will find tools and add-ons for everyday IT tasks, alongside the clever aliases for detecting typical typing errors. Only a few new cmdlets will be presented in this chapter for reasons of space. For this reason, please don't forget to read the detailed descriptions, auxiliary files and websites available for the PSCX in order to find out more.

K1: Download the ***PowerShell Community Extensions*** (PSCX) from the website http://www.codeplex.com/PowerShellCX (top right-hand corner) and install them on your test system. Create the test profile as suggested. Following installation, you will find one of the first add-ons directly in Windows Explorer. In the context menu for any folder you will now find the new entry "Open PowerShell here" for opening a new Windows PowerShell that calls the selected folder directly as a startup directory. Open a Windows PowerShell in this way, and view the loaded ***PSSnapin*** (***get-pssnapin***). You should now see the ***PSCX PSSnapin***. Now display all cmdlets of the ***PSCX PSSnapin*** and list the new help text.

As an example, you will now be introduced to some PSCX cmdlets that could have been put to good use in the previous exercises. If you wish, repeat the named exercises, incorporating the new PSCX add-ons.

K2: In ***chapter C***, you carried out several exercises involving files. In these exercises, links were created in the file directory at the end using the program ***xxMKLink***. Instead of the EXE file, the PSCX add-ons include the cmdlet ***New-Shortcut***, which is called in a similar way:

new-shortcut „Anlegepfad zum neuen Shortcut" „Ziel des Shortscuts"

So, in the solution to exercise **C5** replace ***xxMkLink.exe*** with ***New-Shortcut.***

*In the PSCXs, you will also find nice "dir" add-ons such as **dirS** and **dirT**, which sort a directory by size or time. See the help texts for more details.*

K3: In ***chapter G*** you met Peedy, a COM object that can also be used for voice output. At the end, you created the ***out-peedy*** function for implementing voice output from your PowerShell scripts. At the end of **G7**, your unread e-mails were read out loud. In the PSCX add-ons you will find the cmdlet ***out-speech***, which, although not as charming as Peedy, doesn't cause as much chaos and flapping about on the screen! Try replacing ***out-peedy*** with ***out-speech*** in G7, and decide which you prefer.

K4: If you want to access another computer, you should first check whether it is available on the network. The cmdlet ***Ping-Host*** can help here. In contrast to the old Ping command, here you receive the return value as an object, enabling you to access the individual values more easily. So, ping your own computer and analyse the result using ***get-member***. Following this, output the values for the host name and the number of lost packets (leaving the usual statistics out, of course). If these account to 0, the computer is probably accessible on the network. You can use this knowledge in future scripts to initially check the availability of remote systems. Incidentally, ***Ping-Host*** can also handle name lists and processes the list automatically name by name.

K5: In exercises E9 and E10, you saw how you can send an e-mail via .NET. Calling the .NET objects, which is somewhat awkward, can be simplified using PSCX cmdlet ***Send-SmtpMail***. Here as well, you must of course specify the usual parameters such as the SMTP server, sender and recipient. However, many of these values can be directly transferred to the Windows PowerShell profile, meaning that you don't have to enter them again each time. The help on PSCX will explain this profile file to you and the values that can be saved in it. You can use ***get-help Send-SmtpMail –detailed*** as a simple example for using the cmdlet:

Send-SmtpMail -SmtpHost contoso-DC1.Contoso.local – to Administrator@contoso.local -from info@contoso.local -subject "Warning" – body "This is a generic warning email."

Try sending the e-mail from exercise **E9** using the cmdlet ***Send-SmtpMail***.

The option of automatically creating ZIP files is particularly interesting for e-mail attachments. The cmdlet ***write-zip*** in the PSCX add-ons serves this purpose. The call is surprisingly simple:

dir * | write-zip -outputpath test.zip

This cmdlet compresses all files into one single ZIP file, Test.zip. Other call types are possible, as well as other degrees of compression and the transferal of directory structures to the ZIP file. You can find more information on this in the help for the cmdlet ***write-zip***.

The Active Directory as a PSDrive, thanks to PSCX

Although the PSCX cmdlets are already a powerful add-on for Windows PowerShell, installing the PSCX offers even more. Access to the Active Directory via a PowerShell drive is enabled via an individual provider. ***cd MyDomainName:*** lets you switch directly to the Active Directory, in a similar way to ***cd HKLM:*** for the registry. You can then switch between the OUs via CD. Users, groups and other objects are simply listed using dir, and ***del **** can be used to simply delete objects and subordinate objects. Even if this doesn't give you more rights than you had in the first place, it lets you work quite easily in the Active Directory. Try it for yourself!

Of course, the provider can also be used to create individual Windows PowerShell Drives for other Active Directories and ADAM directories or the local user database.

In addition to the provider, PSCXs also offer several new cmdlets for the Active Directory. Using ***Get-DomainController***, you can obtain a list of all domain controllers, as objects again of course. This means that the DN property also directly gives you the LDAP path for the server object, e.g. ***CN=CONTOSO-DC1,OU=Domain Controllers,DC=Contoso,DC=local*** .

DHCP servers authorized in the Active Directory can also be simply listed using the cmdlet ***Get-DHCPserver***. Finally, ***get-ADObject*** outputs a list of the objects in your Active Directory. If no further parameters are added, all objects are listed; in contrast, ***Get-ADObject -class user*** simply lists users alone. This means you can perform the search exercises in H2 much more easily.

K6: In your virtual test environment, switch over to the Active Directory by simply entering ***cd contoso:*** in Windows PowerShell with ***PSCX PSSnapin***. Entering ***dir*** gives you the available OUs. Now, output a list of all users from the OU ***Users***. Also output all user objects in the Active Directory using the cmdlet ***get-ADObject –class user***.

As ***contoso:*** is a normal ***PSDrive***, you can create new objects using the cmdlet ***New-Item***. To create a new OU called HR, all you have to do is enter ***New-Item   HR -type OrganizationalUnit*** in Windows PowerShell. ***user*** or ***group*** also works as a ***-Type***. After creating a new object, you can also use ***get-itemproperty Objektname*** to list the object's properties, e.g. ***get-itemproperty HR.*** Using ***set-itemproperty***, you can also set object properties. For instance, if a user with the name ***Frankoch*** exists, you can change the first name to the value ***Frank*** by entering the following command:

```
set-ItemProperty .\frankoch  -name FirstName -value "Frank"
```

K7: Create a new OU with the name PSCX in the Active Directory. In the OU, create two users called PSCX1 and PSCX2, as well as a group called PSCX-Group. Change the user first names to PSCX and list the content of the OU. If you have completed the exercises from ***chapter H***, now delete all OUs, users and groups under the HR OU created in ***chapter H***.

PSCX does not have its own cmdlets for directly editing users or other objects in the Active Directory. However, you can do all you need using the item properties. To activate a new account, simply set its ***Disabled*** property to the value ***$False:***

```
Set-Itemproperty User  Disabled  $False
```

If you remember the early days of the Active Directory and compare the time-consuming VB and ADSI scripts to Windows PowerShell, you'll surely agree that Windows PowerShell rightfully has the word ***power*** in its name.

L Partner add-ons: assigning GPOs using SDM Software

You have already been introduced to several interesting add-ons for Windows PowerShell for the Active Directory. For instance, using Quest's ***PSSnapin*** enables you to simply create and change users and groups in the Active Directory, while PSCX add-ons let you simply navigate, list, create or delete users and groups. Thanks to SDM Software, you can now also simply assign group policies and even modify them using a commercial add-on. At www.sdmsoftware.com, you will find both free cmdlets that can be used to assign and list existing group policies, as well as the Scripting Toolkit for Windows PowerShell, which is available subject to a fee. After downloading the free PowerShell add-ons and installing them, you can launch the relevant PowerShell with the SDM ***PSSnapin*** via a link in the Start menu, and use the new cmdlets.

L1: Download the free SDM Software add-ons for Windows PowerShell at www.sdmsoftware.com and install them on your test system. Then launch Windows PowerShell via the SDM link in the Start menu, and output a list of the new ***PSSnapin***s. Display all cmdlets of the ***PSSnapin***. Tip: Take a look back at chapters H and I if you've forgotten how to display information on the ***PSSnapin***. How many new cmdlets can you find?

To display all group policies in the domain, simply call the following command:
Get-SDMgpo *. However, if you would like information on just one group policy, replace the asterisk with the name of the group concerned. SDM has also released a cmdlet that lists which GPO is linked to a specific OU. This is done using ***get-SDMgplink "OUname"***

L2: Display all GPOs in your domain. Following this, display the details of the ***Default Domain Policy***. Which GPO is linked to the "Domain Controllers" OU? Tip: The OU name is specified in the LDAP syntax, and is thus as follows:
„OU=Domain Controllers,dc=contoso,dc=local"

You can use the cmdlet ***New-SDMgpo NeuerName*** to create a new GPO. Using ***Remove-SDMgpo NeuerName*** removes this GPO again. However, a GPO must still always be linked to a group or OU. This can be done, for instance, using:

Add-SDMLink –name GPOname -scope OU -location -1

The ***Name*** is the name of the required group policies, the ***Scope*** is the OU to which the group policy should apply, and the ***Location*** is the order of the GPO (it may be the case that you have several group policies for this OU, so you have to specify which is to be applied first and which last). The value -1 is the lowest position in the list, no matter how long the list is.

L3: Create a new, empty test GPO and assign this to the HR OU created in chapter K. Alternatively, first create the HR OU either manually or using Windows PowerShell, depending on your knowledge.

In the same way as ***Add-SDMgplink*** lets you assign a GPO to an OU, ***remove-SDMgpLink*** lets you remove the link again without specifying the ***–location*** parameter.

In the SDM cmdlets, you will find other helpful functions such as fuctions to export, backup and restore GPOs. However, what is missing here is the option of modifying the settings of GPOs. This is saved for the ***Scripting Toolkit***, which is available for a fee. A trial version can also be found on the

SDM webseite. Following the simple installation prodecure, you receive a further ***PSSnapin*** called ***GetgpObjectPSsnapin*** with one single cmdlet: ***Get-SDMgpobject***

This cmdlet allows you to specifically call a group policy, and assign it to a variable. Using this GPO reference, you can then set and modify individual settings, and adapt them to suit your requirements. Although the syntax isn't complex, it isn't intuitive either, so you should try out the following example to gain a better understanding of the basics:

```
$gpo = Get-SDMgpobject -gponame "gpo://contoso.local/Default Domain Policy"
           -openbyname $true
$stng = $gpo.GetObject("User Configuration/Administrative Templates/Desktop/Active
Desktop/Active Desktop Wallpaper");
$stng.Put("State", [GPOSDK.AdmTempSettingState]"Enabled");
$stng.Put("Wallpaper Name:", "%WINDIR%\Web\Wallpaper\Ascent.jpg");
$stng.Put("Wallpaper Style:", "0");
$stng.Save();
```

The first call, ***"-openbyname $true"***, enables you to address the GPO by its name instead of its GUID. In the second line, the desired setting is then selected. Alongside ***User Configuration***, there is also ***Computer Configuration*** and the subfolders in each case, which you will be familiar with from the graphic console for group policies.
The third line activates the group policy value ("state" set to "enabled").
Finally, the actual value of the group policy is set (in this example, the desktop background image).
The group policy is then saved.

L4: Download the trial version of the Scripting Toolkit from the SDM website and install the tool. Then launch the appropriate Windows PowerShell and display the ***PSSnapin*** as well as the new cmdlets of this ***PSSnapin***. Then, copy the command lines for setting a group policy into Windows PowerShell and execute these lines. Following this, update the group policies using the command ***gpupdate /force.*** To see the new background, you must log out and log back in again.

As you have learned in the past two chapters, Windows PowerShell and partner add-ons let you open a whole new chapter of Active Directory administration. Instead of graphic tools, you can now use effective scripts and therefore ensure a high level of automation. And, alongside lower costs, higher automation also leads to more time for other tasks. Now doesn't that sound appealing?

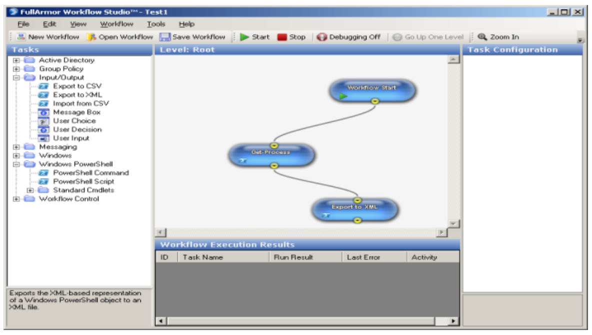
M Partner add-ons: FullArmor Workflow Studio

Workflow Studio, provided by the company FullArmor (www.fullarmor.com), offers you an independent tool that is not only a Windows PowerShell add-on, but also provides companies with a very interesting way of implementing business workflows and process automation. Using a graphic workflow designer, you can quickly and easily implement various requirements in the area of ***Server & Client Provisioning***, right through to Active Directory & GPO tasks. On the website, you will find not only a 30-day free trial version of the Workflow Studio, but also coaching videos and more detailed product descriptions.

M1: Download the free trial version of the Workflow Studio from FullArmor at http://www.fullarmor.com, and install the tool on your virtual server. The installation process is very simple, and pretty much self-explanatory. Then, start the Workflow Studio using the newly created link in the Start menu. Create a new workflow with the name ***Test1*** in the Start wizard.

Once you have opened the Workflow Studio, you can view the pre-prepared task blocks in the task bar on the left-hand side. Alongside Active Directory tasks such as creating users or groups, there are components for working in the file system and sending e-mails, and also generic PowerShell commands that enable any PowerShell scripts to be executed. Let's take a look at the entire procedure using one, very simple example.

M2: Using the mouse, add the task ***Windows PowerShell / Standrad Cmdlets / Get-Process*** from the list on the left-hand side to the working area in the middle, and click on the new block. On the right-hand side, you can now see the properties of the task block, such as input and output variables and filters. Change the value for the ***Process Name Filter*** from * to ***P****. This has the result that only the processes beginning with P are selected. The list that is output is saved in the ***Output Variable*** (which can also be seen on the right-hand side). Copy the value of this variable to the clipboard. Then add another task: ***Input/Output / Export to XML**.* On the right-hand side, set the value for the output file to ***C:\Downloads\Scripts\Test1-out.xml**.* It is now important that the input value for this task is the output value of the get-process task so that a link is really established. To ensure this is the case, simply paste the old ***Output Variable*** from the clipboard to the new ***Input Variable*** value. Now connect the individual blocks using the mouse by clicking on the yellow circles and drawing a line to the next block. To not only view the result in the XML file but also receive a direct output, set the values ***Save As HTML*** and ***Show HTML*** in the ***Export XML*** task to the value ***True***. At the end, the result you obtain should look something like this:

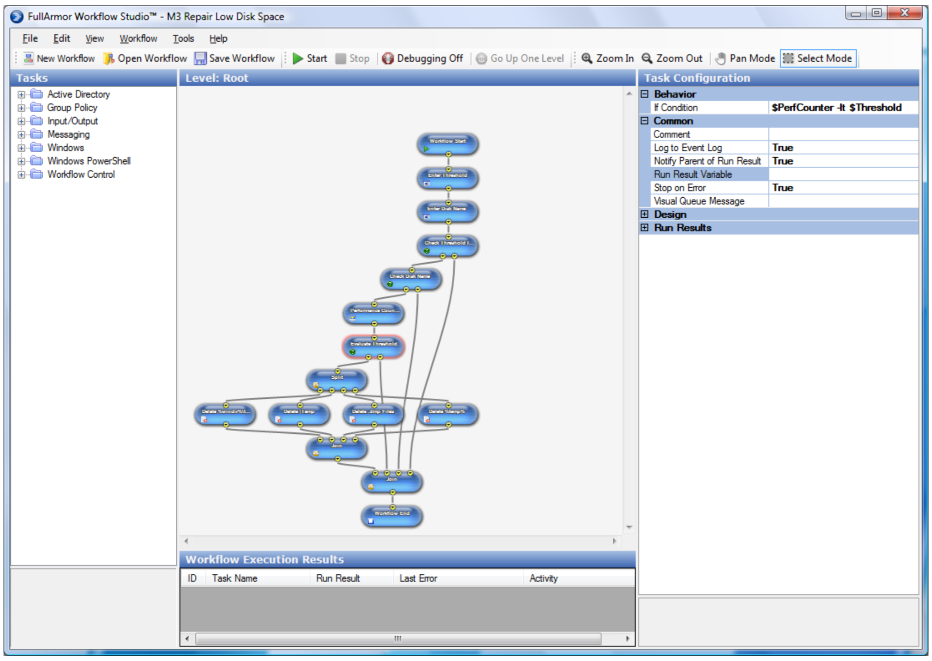


Exports the XML-based representation of a Windows PowerShell object to an XML file.

Launch the script using the green start button in the top-middle of the screen. You should optain at least the POP3 service as a result.

Of course, this is a very simple example. To obtain a better overview of the possibilities offered by the Workflow Studio, you will find four more examples in the practice files for this workshop that I would like to introduce to you in a bit more detail. First, copy the four workflows to your virtual server and open the workflow file required in each case via the File menu in the Workflow Studio.

M3: Repair low disk space This example is in principle very simple. However, in practice you will find that logical, precise implementation involves more steps than you first think. The workflow queries the amount of free space you require and the desired drive. The entries you make are checked, and then compared with the WMI counter for free disk space. If the free space is smaller than required, various files in the TEMP directory are deleted, as well as DUMP files. Finally, the individual strands are united again and the workflow ends. Take your time and look at the example step by step, then try it out in your test environment.



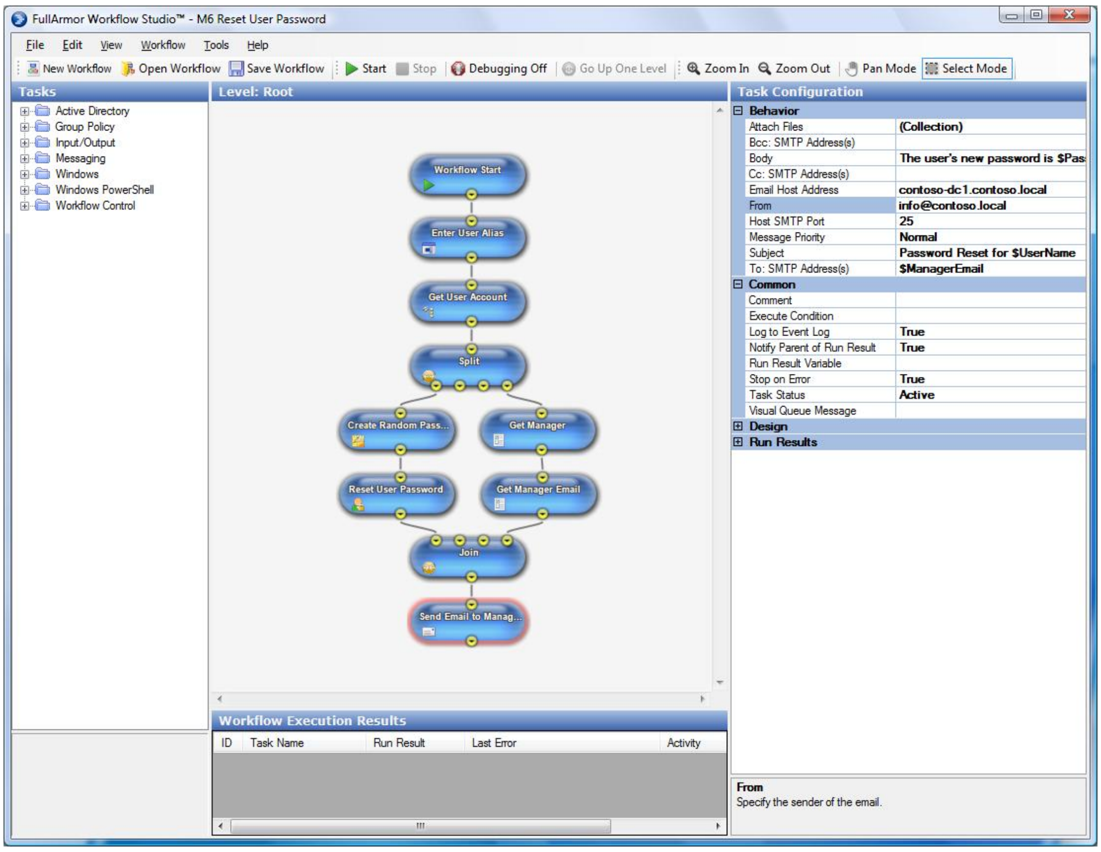
Task Configuration	
Behavior	
If Condition	$PerfCounter -lt $Threshold
Common	
Comment	
Log to Event Log	True
Notify Parent of Run Result	True
Run Result Variable	
Stop on Error	True
Visual Queue Message	
Design	
Run Results	

ID	Task Name	Run Result	Last Error	Activity

M4: Restart service via command line This workflow restarts a service. To start off with, a check is carried out as to whether a service was transmitted as a parameter, otherwise it is queried by means of a dialog box. If an entry has been made, this is interpreted as a service and the service is stopped. The system waits 5 seconds before restarting the service. Take your time and look at the example step by step, then try it out in your test environment.

M5: Create New-User Account This workflow lets you create a new user in the Active Directory. After queries for information such as names, a password is automatically generated. If required, the new user can be directly created as a Domain Administrator. If it is specified that the new user is a manager, this user will receive information on their new user account via e-mail. Take your time and look at the example step by step, then try it out in your test environment. Adapt the Active Directory information to your test environment. As a manager, for instance, you can select the administrator and view the e-mail using Outlook Express.

M6: Reset user Password In this last example, I will show you how to reset a user password. Once the user name (display name with spaces, etc.) has been entered, a new password is generated and a search is performed for the user's manager in the Active Directory. The new password is sent to the manager by e-mail. Take your time and look at the example step by step, then try it out in your test environment. Adapt the Active Directory information to your test environment. As a user and manager you can, for instance, use the user created in exercise M5 whose manager is the administrator, which lets you look at the e-mail in Outlook Express.



Behavior	
Attach Files	(Collection)
Bcc: SMTP Address(s)	
Body	The user's new password is $Pas
Cc: SMTP Address(s)	
Email Host Address	contoso-dc1.contoso.local
From	info@contoso.local
Host SMTP Port	25
Message Priority	Normal
Subject	Password Reset for $UserName
To: SMTP Address(s)	$ManagerEmail
Common	
Comment	
Execute Condition	
Log to Event Log	True
Notify Parent of Run Result	True
Run Result Variable	
Stop on Error	True
Task Status	Active
Visual Queue Message	
Design	
Run Results	

From
Specify the sender of the email.

As you will hopefully have seen, the Workflow Studio is very powerful. Once you've understood the logic and procedure involved, you can quickly and easily implement even complex workflows. Using Windows PowerShell, WMI and .NET, an almost unlimited variety of options are available to you as possible tasks. Alongside the graphic designer, the Workflow Studio also offers license forms for servers and clients to enable distributed workflows. The options for creating log files, tracing and debugging are a huge help in everyday work, and offer one of the quickest ways to create individual, comprehensive solutions.

N Access to databases via .NET

One of the most interesting possibilities offered by Windows PowerShell is direct database access. Creating new tables and reading and writing data are normal daily tasks for a system administrator. .NET and the possibilities it offers for accessing databases registered in the system can be used to carry out these tasks not only for the Microsoft SQL Server. However, Microsoft is going yet another step further with the SQL Server 2008 by creating a script interface that makes access even easier. Nevertheless, as most readers will in all probability be dealing with an older SQL server version, I would like to focus on this first. You can download the Microsoft SQL Server 2005 Express Edition for free at http://www.microsoft.com/downloads/details.aspx?FamilyID=220549b5-0b07-4448-8848-dcc397514b41&displaylang=en. This database is scaleable up to 4 GB in size and 1 processor. Following registration, it can be implemented in individual solutions and distributed free of charge. Although we will be managing the SQL server entirely using scripts in the following exercises, the graphic interface is sometimes very helpful. You can also download the interface free from Microsoft at http://www.microsoft.com/downloads/details.aspx?FamilyId=C243A5AE-4BD1-4E3D-94B8-5A0F62BF7796&DisplayLang=en.

N1: Download the Microsoft SQL Server 2005 Express Edition for free from http://www.microsoft.com/downloads/details.aspx?FamilyID=220549b5-0b07-4448-8848-dcc397514b41&displaylang=en and the graphic user interface from http://www.microsoft.com/downloads/details.aspx?FamilyId=C243A5AE-4BD1-4E3D-94B8-5A0F62BF7796&DisplayLang=en. Install these two tools in your virtual test environment. Installation is very simple, and is carried out by calling either the EXE or the MSI file. Accept the default settings to begin with; on the server, however, in the ***Feature Selection*** dialog, add ***all Client Components***. If you use the computer for tasks other than carrying out these Windows PowerShell exercises, after installation, you should look for further updates for the SQL server and install these as well.

Once the SQL server is installed, all you need is a database. This can be set up using .NET. The following example creates a database with tables for entries from the system event log. This permits, for instance, analysis, archiving or similar at a later date. Of course, you can also adapt the procedure to suit your own requirements.

First, you must create a database. To create the database, a connection is set up via .NET to the required SQL server (which can also be installed on another PC), where the empty database is created:

```
$SQLCn = New-Object
System.Data.SqlClient.SqlConnection("Server=.\SQLEXPRESS;Trusted_Connection=Yes")
$SQLCn.Open()
$SQLCMD = $SQLCn.CreateCommand()
$SQLCMD.CommandText = "CREATE DATABASE EventDB"
$Result= $SQLCMD.ExecuteNonQuery()
If ($Result) { Write-Host "Anlegen der Datenbank erfolgreich." }
$SQLCN.Close()
```

Following this, one or more tables are created in the database, depending on requirements. In this example, we shall create a table with the typical columns for system log entries. The proecdure here is similar to that for creating a database:

```
$SQLCn = New-Object System.Data.SqlClient.SqlConnection( “Server=Contoso-
DC1\SQLEXPRESS;Trusted_Connection=Yes;Database=EventDB”)
$SQLCn.Open()
$SQLCMD = $SQLCn.CreateCommand()
$SQLCMD.CommandText = “CREATE TABLE Events (EvtIX int, EvtTXT varchar(1000), EvtTYPE
varchar(50), EvtTIME datetime, EvtID int, EvtSrc varchar(100))”
$Result= $SQLCMD.ExecuteNonQuery()
If ($Result) { Write-Host “Anlegen der Tabelle erfolgreich.“ }
$SQLCN.Close()
```

After the table has been created, values can be entered in the database. The 10 most recent entries from the system log serve as an example here:

```
$SQLCn = New-Object System.Data.SqlClient.SqlConnection( “Server=Contoso-
DC1\SQLEXPRESS;Trusted_Connection=Yes;Database=EventDB”)
$SQLCn.Open()

Get-eventlog system –newest 10 | foreach-object {
$SQLCMD = $SQLCn.CreateCommand()
$SQLCMD.CommandText = “INSERT Into Events VALUES(@EvtIX, @EvtTXT, @EvtTYPE,
@EvtTIME, @EvtID, @EvtSrc)”
$P = $SQLCMD.Parameters.AddwithValue(“@EvtIX“, $_.Index)
$P = $SQLCMD.Parameters.AddwithValue(“@EvtTXT“, $_.Message)
$P = $SQLCMD.Parameters.AddwithValue(“@EvtTYPE“, $_.Category)
$P = $SQLCMD.Parameters.AddwithValue(“@EvtTIME“, $_.TimeGenerated)
$P = $SQLCMD.Parameters.AddwithValue(“@EvtID“, $_.EventID)
$P = $SQLCMD.Parameters.AddwithValue(“@EvtSrc“, $_.Source)
$Result= $SQLCMD.ExecuteNonQuery()
If ($result –eq $False) { Write-Host “Fehler beim Datenschreiben“ }
}
$SQLCN.Close()
```

Listing data records is just as easy as writing data (provided you can write SQL statements):

```
$SQLCn = New-Object System.Data.SqlClient.SqlConnection( “Server=Contoso-
DC1\SQLEXPRESS;Trusted_Connection=Yes;Database=EventDB”)
$SQLCn.Open()
$SQLCMD = $SQLCn.CreateCommand()
        $SQLCMD.CommandText = “SELECT * FROM Events”
$Results = $SQLCMD.ExecuteReader()

While ( $Results.Read() ) {
Write-Host $Results.Item(“EvtIX”) $Results.Item(“EvtTIME”) $Results.Item(“EvtTXT”)
}
$SQLCN.Close()
```

As you have seen in this chapter, .NET makes working with databases very easy, even though the actual work with SQL statements is made no easier. In the above examples, even in Windows PowerShell you must be able to write classic SQL statements and know how to deal with SELECT statements, and WHERE and JOIN expressions. LINQ, an add-on to the .NET Framework, is an interesting development in this area for simpler handling of databases. You can find out more about LINQ and the powerful oppportunities it offers on the Microsoft MSDN website, for instance at http://msdn2.microsoft.com/en-us/netframework/aa904594.aspx. Its effects on Windows PowerShell are as yet unknown.

While LINQ is aimed more at developers of database applications, it is also good for database administrators to know that a scripting tool comes with the SQL Server 2008, which also helps make life easier. You can read more about this in the SQL Server 2008 documentation, for instance under http://www.microsoft.com/sql/techinfo/whitepapers/sql_2008_manageability.mspx.

SOLUTIONS TO EXERCISES

Solution scripts for the exercises in this booklet

A1

```
WindowsServer2003-KB926139-x86-ENU.exe /?
WindowsServer2003-KB926139-x86-ENU.exe /quiet (/norestart)
```

The /norestart parameter is optional; even if no restart is required for the installation of Windows PowerShell, you can set it to server systems just to make sure. However, you need to remember to perform the restart if necessary.

B1

```
Get-executionpolicy

cd hklm:
cd HKLM:\software\Microsoft\PowerShell\1\ShellIds
Get-ItemProperty .\Microsoft.PowerShell
```

The desired registry key is not available as standard, which is why the default value, ***Restricted***, automatically applies as being set.

B2

```
Set-Executionpolicy RemoteSigned
```

Only once the cmdlet ***Set-ExecutionPolicy*** is used is the registry key set and then adopts the relevant value; in the example, this value is ***RemoteSigned***.

B4

The group policy sets a different registry key to the cmdlet ***Set-ExecutionPolicy.*** For this reason, a warning message appears if a group policy exists and a change to the security environment is carried out via cmdlet. The group policy value always takes precedence. However, the registry key from **B1** remains unchanged.

The group policy administrative template sets the following registry key:

```
HKLM:\Software\Policies\Microsoft\Windows\PowerShell
```

C1

```
# The following script analyzes a folder anc leans up the files
# First we create a new PS drive to minimize the file path and to shorten the script
# Out-NUL avoids any screen output
New-PSDrive -name FK -PSProvider FileSystem -root C:\Dwonloads\Files | out-null

# Now we list all files and sort them by their extensions. Each extension is selected once
# and for each extension, we create an own folder to store later on the files
# We use the well known MD command from the old DOS days instead of new-item
# new-item is longer but knows the –force switch to avoid erros in case the folder exists
# So pick your command as you like!
get-childitem fk:\ | sort-object extension -unique | foreach-object {
 MD ( "fk:\restored_" + $_.extension) | Out-Null }

# Now that we have the folders, we simply move the files to the correct final location
# But we need toskip the folder objects and leave them where they are
get-childitem fk:\ | where-object {$_.mode -notmatch "d"} | foreach-object {
move-item $_.fullname ("fk:\Restored_" + $_.extension) }
# Done with the script
```

C2

```
# The PSDrive simply shortens the PowerShell command line, that's it.
New-PSDrive -name FK -PSProvider FileSystem -root c:\Downloads\files | Out-Null
get-childitem fk:\ –recurse | where-object {$_.mode -notmatch "d"} | foreach-object {
        $_.isreadonly = 0
}
```

The above script is called by a batch file containing the following line (change the path to suit the script where necessary):

```
powershell.exe c:\Downloads\Scripts\C2.ps1
```

C1-Prepare.ps1

```
Dir C:\Downloads\Files -Recurse | where-object {$_.mode -notmatch "d"} | sort extension
-unique | foreach { $_.IsReadOnly = $False; $_.LastAccessTime = (get-date).AddYears(-1) }

dir C:\Downloads\Files -Recurse | where-object {$_.mode -notmatch "d"} | sort extension
-desc -unique|foreach{$_.IsReadOnly = $False;$_.LastAccessTime (get-date).AddYears(-2)}
```

C3

```
# The scripts starts here. First check: do we have 3 parameters?
If ($Args.count –ne 3) {
        write-warning “call the script with 3 parameter: source target date-in-months”
} else {
If (test-path $Args[0]) {
        If (test-path $Args[1] ) {
        $CheckDate = ((Get-date).AddMonths(- $Args[2])).ToOADate()
        Dir $args[0] -recurse | where-object {$_.mode -notmatch "d"} | foreach-object {
                if ( ($_.LastAccessTime).ToOADate() -lt $CheckDate) {
                $_.name + "   " + (($_.LastAccessTime))
                }
        }
} else { Write-Warning “Target path does not exists” }
} else {Write-Warning (“Source path does not exists: ” + $Args[0]) }
}
```

C4

```
# The scripts starts here. First check: do we have 3 parameters?
If ($Args.count –ne 3) {
        write-warning “call the script with 3 parameter: source target date-in-months”
} else {
If (test-path $Args[0]) {
        If (test-path $Args[1] ) {
        $CheckDate = ((Get-date).AddMonths(- $Args[2])).ToOADate()
        $MyPath = $Args[1]
        Dir $args[0] -recurse | where-object {$_.mode -notmatch "d"} | foreach-object {
                if ( ($_.LastAccessTime).ToOADate() -lt $CheckDate) {
                $myACL = get-Acl $_.FullName
                $MyName = $MyPath + "\" + $_.Name
                Move-Item $_.FullName $MyPath
                Set-Acl $Myname $myACL
                }
        }
} else { Write-Warning “Target path does not exists” }
} else {Write-Warning (“Source path does not exists: ” + $Args[0]) }
}
```

C5

```
# The scripts starts here. First check: do we have 3 parameters?
If ($Args.count –ne 3) {
        write-warning “call the script with 3 parameter: source target date-in-months”
} else {
If (test-path $Args[0]) {
        If (test-path $Args[1] ) {
        $CheckDate = ((Get-date).AddMonths(- $Args[2])).ToOADate()
        $MyPath = $Args[1]
        Dir $args[0] -recurse | where-object {$_.mode -notmatch "d"} | foreach-object {
                if ( ($_.LastAccessTime).ToOADate() -lt $CheckDate) {
                $myACL = get-Acl $_.FullName
                $MyName = $MyPath + "\" + $_.Name
                Move-Item $_.FullName $MyPath
# Bes ure that the path to  xxMKLink.exe is correct!
                .\xxMkLink.exe $_.FullName $Myname /q

                Set-Acl $Myname $myACL
                $Myname = $_.FullName + ".lnk"
                Set-Acl $Myname $myACL
                }
        }
} else { Write-Warning “Target path does not exists” }
} else {Write-Warning (“Source path does not exists: ” + $Args[0]) }
}
```

D1

```
CD HKL:
CD HKLM:\Software\Microsoft
Dir W*
```

Displaying registry keys works in exactly the same way as listing files in the file system.

D2

```
CD HKL:
CD HKLM:\Software\Microsoft
dir W*R\*\W*
```

Displaying registry keys works in exactly the same way as listing files in the file system.

D3

```
New-Item –path “HKLM:\Software“ –Name “Contoso“
New-itemproperty –literalpath “HKLM:\Software\Contoso“ –Name “Appname“ –Value
        ”Contoso CRM” –type string
Remove-item .\contoso
```

Registry keys and values can be deleted very easily by deleting the top registry key.

D4

```
Cd HKLM:\Software\Microsoft\Windows\CurrentVersion
Get-itemproperty run
New-itemproperty –literalpath
“HKLM:\Software\Microsoft\Windows\CurrentVersion\Run“ –Name “Notepad“ –Value
      ”c:\windows\notepad.exe” –type string
```

When you log in again, Notepad should automatically start. Ideally, you should replace the path c:\Windows with the environment variable %windir%

E1

```
# Get the computer name with the help of WMI
$Name = (get-wmiobject Win32_Computersystem).name

# The HTML script comes from the first Windows PowerShell workshop
# where you find the comments, too
get-service | ConvertTo-Html -Title "Get-Service" -Body "<H2>The result of get-
service</H2> " -Property Name,Status | foreach {
if($_ -like "*<td>Running</td>*"){
$_ -replace "<tr>", "<tr bgcolor=green>"
} elseif ($_ -like "*<td>Stopped</td>*"){
$_ -replace "<tr>", "<tr bgcolor=red>"
}else { $_} }  >  (“fk:\” + $name + “.html”)
```

E2a

```
[void][System.reflection.assembly]::LoadWithPartialName("System.Windows.Forms")
$form = new-object Windows.Forms.Form
$button = new-object Windows.Forms.Button
$button.add_click({$form.close()})
$form.controls.add($button)
$form.ShowDialog()
```

E2b

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

E3

```
[void][System.reflection.assembly]::LoadWithPartialName("System.Windows.Forms")
$form = new-object System.Windows.Forms.Form
$DataGridView = new-object System.windows.forms.DataGridView
$Form.Text = "My First Datagrid"
$array= new-object System.Collections.ArrayList
$array.AddRange( @( get-service | write-output  ) )
$DataGridView.DataSource = $array
$DataGridView.Dock = "fill"
$DataGridView.AllowUsertoResizeColumns=$True
$form.Controls.Add($DataGridView)
$form.showdialog()
```

E4

```
# The only code changes are highlighted
[void][System.reflection.assembly]::LoadWithPartialName("System.Windows.Forms")
$form = new-object System.Windows.Forms.Form
$DataGridView = new-object System.windows.forms.DataGridView
$Form.Text = "My First Datagrid"
$array= new-object System.Collections.ArrayList
$array.AddRange( @(  get-service | sort-object Status | write-output  ) )
$DataGridView.DataSource = $array
$DataGridView.Dock = "fill"
$DataGridView.AllowUsertoResizeColumns=$True
$form.Controls.Add($DataGridView)
$form.showdialog()
```

E4

```
# The only code changes are highlighted
 [void][System.reflection.assembly]::LoadWithPartialName("System.Windows.Forms")
$form = new-object System.Windows.Forms.Form
$DataGridView = new-object System.windows.forms.DataGridView
$Form.Text = "My First Datagrid"
$array= new-object System.Collections.ArrayList
$array.AddRange( @(  get-process | sort-object company | write-output  ) )
$DataGridView.DataSource = $array
$DataGridView.Dock = "fill"
$DataGridView.AllowUsertoResizeColumns=$True
$form.Controls.Add($DataGridView)
$form.showdialog()
```

E5

```
# Create a list with all file extensions
$ext = Get-ChildItem * -recurse | foreach {[System.io.path]::GetExtension($_) }
# Group the extensions and sort them by their numbers (count)
$ext | where { $_ -ne “”} | group | sort count
```

E6

```
# Get the no.1 extension
($ext | where { $_ -ne “”} | group | sort count | select -last 1).Name
```

E7

```
$newrights =  [System.Security.AccessControl.FileSystemRights]”Read, Write”
$InheritanceFlag =  [System.Security.AccessControl.InheritanceFlags]::None
$PropagationFlag =  [System.Security.AccessControl.PropagationFlags]::None
$Typ =[System.Security.AccessControl.AccessControlType]::Allow
$ID = new-object System.Security.Principal.NTAccount(“Contoso\Administrator”)
$SecRule =new-object  System.Security.AccessControl.FileSystemAccessRule($ID,
$newrights, $InheritanceFlag, $PropagationFlag, $Typ)

$myACL = get-acl „.\c1-prepare.ps1“
$myACL.AddAccessRule($SecRule)
Set-ACL “.\c1-prepare.ps1“ $myACL
```

E8

```
# Send email with .NET
# This example will NOT work by design!
$Mail = New-Object System.Net.Mail.MailMessage
$Mail.Sender = “Info@contoso.local”
$Mail.Subject = “PowerShell information from ” + (dir env:\Computername).Value
$Mail.Body = “Alarm on server “ + (dir env:\Computername).Value + “ at “ + (get-date)
$Mail.To = “Administrator@contoso.local”
```

E9

```
# Send email with .NET
$mailserver = “contoso-dc1.contoso.local”
$Mail = New-Object System.Net.Mail.MailMessage( “Info@contoso.local”,
“administrator@contoso.local”)

$Mail.Subject = “PowerShell information from ” + (dir env:\Computername).Value
$Mail.Body = “Alarm on server “ + (dir env:\Computername).Value + “ at “ + (get-date)
$Mail.IsBodyHTML = $False

$MailClient = New-Object System.Net.Mail.SmtpClient
$MailClient.Host = $Mailserver
$MailClient.Send($Mail)
```

E10

```
# Send email and attachment with .NET
$mailserver = “contoso-dc1.contoso.local”
$Mail = New-Object System.Net.Mail.MailMessage( “Info@contoso.local”,
“administrator@contoso.local”)
$Mail.Subject = “PowerShell information from ” + (dir env:\Computername).Value
$Mail.Body = “Alarm on server “ + (dir env:\Computername).Value + “ at “ + (get-date)
$Mail.IsBodyHTML = $False

$Attached = New-Object System.Net.Mail.Attachment(“C:\boot.ini”)
$Mail.Attachments.Add($Attached)

$MailClient = New-Object System.Net.Mail.SmtpClient($Mailserver)

$MailClient.UseDefaultCredentials = $False
$MailClient.Credentials = New-Object System.Net.NetworkCredential(“contoso\Info”,
“Pass1word”)

$MailClient.Send($Mail)
```

E11

```
$Outlook = New-Object .-com Outlook.Application
$Inbox = $Outlook.Session.GetDefaultFolder(6)
$Inbox.items | where { $_.Unread }| foreach { write-host $_.Subject }
```

F1

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

The parameter –list always lists just the first appearance of the string; if the parameter is not set, all positions of the string will be output.

F2

```
Get-eventlog “Windows PowerShell” –newest 10
Get-eventlog “Windows PowerShell” | group EventID | sort count –descending
```

F3

```
$a = new-object –type system.diagnostics.eventlog –argumentlist System
# Source can be chosen randomly as you like, but maybe use something appropreatly
$a.source = “Windows PowerShell Labs”
get-service | where { $_.status -eq "Stopped"} | select -first 5 | foreach {
$a.writeentry(“Service stopped: ”+ $_.Name,“Information“)
}
get-eventlog System -newest 10
```

G1

```
$path = “” + @(Join-path $env:windir "MSAgent\chars\peedy.acs")
if ((test-path $path) -eq $false) {
  write-host "Please download Peedy from www.microsoft.com/msagents"
} else {
        $AgentControl = New-Object -com agent.control.2
        $AgentControl.connected = $true
        [void]$AgentControl.Characters.load("Peedy", $path)
        $Peedy = $AgentControl.Characters.Item("Peedy")

        [Void]$Peedy.Show()
        [Void]$Peedy.moveto(300,300)
        [void]$Peedy.Speak("I am ready, and you?")
}
```

G2

```
$PC = New-Object -com agent.control.2
$pc.connected = $true
[void]$pc.Characters.load("Peedy","C:\windows\MSAgent\chars\Peedy.acs")
$Peedy = $pc.Characters.Item("Peedy")

$Display = Get-WmiObject -class Win32_DesktopMonitor
# check if multiscreen display is enabled:
if (($display | measure-object).count -gt 1) {
        $height = $display[0].ScreenHeight - $peedy.height
        $width = $display[0].ScreenWidth - $peedy.width} else {
        $height = $display.ScreenHeight - $peedy.height
        $width = $display.ScreenWidth - $peedy.width }
[void]$peedy.show()
[Void]$Peedy.moveto(0,0)
Start-sleep -seconds 3
[Void]$Peedy.moveto(0,$height)
Start-sleep -seconds 3
[Void]$Peedy.moveto($width,$height)
Start-sleep -seconds 3
[Void]$Peedy.moveto($width,0)
Start-sleep -seconds 3
[Void]$Peedy.moveto(0,0)
```

G3

Creation of a list of all Peedy animation sequences by outputting the ***AnimationNames*** property of the object: ***$Peedy.AnimationNames***

Creation of a list of all animation sequences through simply calling and playing all sequences using a "for-each" loop and outputting the name of the sequence concerned:

```
$Peedy.AnimationNames | foreach {
        $_
        [void]$Peedy.Play($_)
        Start-sleep –seconds 5
        [void]$Peedy.StopAll()
}
```

G4

System monitoring using Peedy. To start off with, the following script creates a Peedy instance, presents Peedy on the screen, creates a list of five stopped services and outputs this list via Peedy:

```
$PC = New-Object -com agent.control.2
$pc.connected = $true
[void]$pc.Characters.load("Peedy","C:\windows\MSAgent\chars\Peedy.acs")
$Peedy = $pc.Characters.Item("Peedy")
[void] $speedy.show()
[void]$peedy.moveto(100,300)


[void]$peedy.Play("GetAttention")
Start-sleep –seconds 2
[void]$Peedy.StopAll()

Get-service | where { $_.status –eq “Stopped” } | select-object –first 5 | foreach {
        [void] $Peedy.Speak($_.Name)
        Start-sleep –seconds 5
         [void]$Peedy.StopAll()
}
[void]$peedy.Hide()
```

G5

```
Function   out-peedy {
$PC = New-Object -com agent.control.2
$pc.connected = $true
[void]$pc.Characters.load("Peedy","C:\windows\MSAgent\chars\Peedy.acs")
$Peedy = $pc.Characters.Item("Peedy")
 [void] $speedy.show()
[void]$peedy.moveto(100,300)
$text = “” + $Args[0]
 [void]$peedy.Speak($text)
Start-sleep –seconds 5
 [void]$Peedy.StopAll()
[void]$peedy.Hide()
}
```

G6

```
Function   out-peedy {
$PC = New-Object -com agent.control.2
$pc.connected = $true
[void]$pc.Characters.load("Peedy","C:\windows\MSAgent\chars\Peedy.acs")
$Peedy = $pc.Characters.Item("Peedy")
 [void] $peedy.show()
[void]$peedy.moveto(100,300)
$text = “” + $Args[0]
If ($Args.Count –gt 1) {
        If ($Args[1]  -match “-q”) { [void]$peedy.Think($text)
        } else {  [void]$peedy.Speak($text) }
} Else {  [void]$peedy.Speak($text) }
Start-sleep –seconds 5
 [void]$Peedy.StopAll()
[void]$peedy.Hide()
}
```

G7

```
# Outlook 2007 and the inbox (folder 6)
$Outlook = New-Object -com Outlook.Application
$Inbox = $Outlook.Session.GetDefaultFolder(6)
$Inbox.items | where { $_.Unread }| foreach { out-peedy $_.Subject }
```

H1

```
$objDomain = [ADSI]“LDAP://dc=contoso,dc=local”
$objOU = $objDomain.Create(“organizationalUnit”, “ou=HR”)
$objOU.SetInfo()

$objOU = [ADSI]“LDAP://ou=hr,dc=contoso,dc=local“
$objGrp = $objOU.Create(„group“, „cn=HRGroup“)
$objGrp.Put(„sAMAccountName“, „HRGroup“)
$objGrp.SetInfo()

$objUser = $objOU.Create(„user“, „cn=frankoch“)
$objUser.Put(„sAMAccountName“, „frankoch“)
$objuser.SetInfo()

$objUser2 = $objOU.Create(„user“, „cn=frankoch2“)
$objUser2.Put(„sAMAccountName“, „frankoch2“)
$objuser2.SetInfo()
```

H2

```
$objUser = [ADSI]"LDAP://cn=frankoch,ou=HR,dc=contoso,dc=local"
$objUser.SetPassword("Pass1word")
$objUser.psbase.InvokeSet("AccountDisabled",$false)
$objUser.SetInfo()
$objUser = [ADSI]"LDAP://cn=frankoch2,ou=HR,dc=contoso,dc=local"
$objUser.SetPassword("Pass1word")
$objUser.psbase.InvokeSet("AccountDisabled",$false)
$objUser.SetInfo()
```

H4

```
$ADDomain = [ADSI]"LDAP://dc=contoso,dc=local"
$ADSearch = New-Object System.DirectoryServices.DirectorySearcher
$ADSearch.SearchRoot = $ADDomain
# definition des Filters: nur Computer,Suche nach Namen
$ADSearch.Filter = „(objectCategory=user)"
$ADSearch.PropertiesToLoad.Add("name")
$ADSearch.PropertiesToLoad.Add("description")
$ergebnis = $ADSearch.FindAll()

# Zur Ausgabe wenden wir einen Trick an, um nur den Namen auszugeben
Foreach ($erg in $ergebnis) {
        $ADuser = $erg.Properties
        $ADUser.name + " " + $ADUser.description
}
```

H5

```
Get-command –pssanpin Quest.ActiveRoles.ADManagement
```

In order to ensure that the Quest add-ons are always available, you must enter the call ***Add-PSSnapIn Quest.ActiveRoles.ADManagement*** in one of the profile files.

H6

Creating a new user:

```
New-QADUser -name "Frankoch3" -organizationalUnit "OU=HR,DC=contoso,DC=local"
-samAccountName "Frankoch3" -UserPassword "Pass1word"
```

Creating a new group:

```
new-qadGroup -name "HR Group2" -organizationalUnit "OU=HR,DC=contoso,DC=local" -
samAccountName "hrgroup2" -Grouptype "Security" -Groupscope "Global"
```

Adding a user to a group:

```
add-QADGroupMember  -identity "CN=HR Group2,OU=HR,DC=contoso,DC=local"  -member
"Contoso\frankoch3"
```

H7

```
import-csv  h7-users.csv | foreach { new-QADuser -name $_.Name -sam $_.sam
-password $_.password  -Org $_.OU }
```

I1

```
get-process | sort cpu | select -last 10 | .\out-chart.ps1 processname CPU
```

I2

```
Get-pssnapins
Add-pssnapin PowerGadgets
```

I3

```
Get-command –pssnapin PowerGadgets
get-process | sort cpu | select name, cpu -last 10 | out-chart
```

I4

Outputing file extensions

```
dir *.* | group Extension | out-chart -Values Count -Label Name
dir *.* | sort length | select name, length –last 10 | out-chart
```

I5

To obtain a return value by double-clicking, all you need to do is continue the pipeline after ***Out-Chart***:

```
get-process | sort cpu | select name, cpu -last 10 | out-chart | foreach { $_ }
```

Script: Out-Chart.ps1

```
param($xaxis, $yaxis)

begin {
        $categories = @()
        $values = @()
        $chart = new-object -com OWC11.ChartSpace.11
        $chart.Clear()
        $c = $chart.charts.Add(0)
        $c.Type = 3
        $c.HasTitle = $true
        $c.Title.Caption = "Chart generated on $(get-date)"
        $series = ([array] $chart.charts)[0].SeriesCollection.Add(0)
}
Process {
        $categories += $_.$xaxis
        $values += $_.$yaxis *1
}
End {
        $series.Caption = "chart"
        $series.SetData(1, -1, $categories)
        $series.SetData(2, -1, $values)
        $filename = (resolve-path .).Path + "\chart.jpg"
        $chart.ExportPicture($filename, "jpg", 800,  500)
        invoke-item $filename
}
```

I cannot provide you with a precise explanation of this script in this exercise booklet. However, chapter 7.4.2 of the Windows PowerShell book by Bruce Payette explains the ideas in full. Here are just a few tips:

- In the ***Param*** line, the script arguments are defined and set, provided that they are not further defined when the script is called.
- The ***Begin*** area is for processing commands before the transmitted items are even taken into account, i.e. it is a kind of initialization. $Script variables only exist in the context of the script, and are used to more clearly define the chart item.
- The ***Process*** area is where the individually transmitted items are processed.
- The ***End*** area allows final tasks to be carried out without the transmitted objects.
- The ***$values*** must be multiplied by 1 to enable clean output of decimal figures. You don't believe me? Try it without!

J1

```
Get-WmiObject -list | where {$_ -match "win32_n"}
Get-WmiObject -list | where {$_ -match "win32_n"} | measure-object
```

J2

```
Get-WmiObject -Class "Win32_NetworkAdapterConfiguration" -Filter IPEnabled=True |
format-table index, IPAddress, Description -autosize

Get-WmiObject -Class "Win32_NetworkAdapter" | get-member

Get-WmiObject -Class "Win32_NetworkAdapter" | foreach{ $_.Disable() }
```

J3

```
$WC = New-Object net.webclient
$Global:BatSeite = $wc.downloadstring(“https://www.dellbatteryprogram.com/“)

$R = [regex]”&nbsp;([A-Z0-9][A-Z0-9][0-9][0-9][0-9])”
$Matches = $R.Matches($Global:BatSeite)
Write-Host “Anzahl gelisteter Akku-Typen: $($Matches.Count) “

$Global:BatterieListe = $Matches | ForEach { $_.Groups[1].Captures[0].value }
Get-WMIObject –Class Win32_Battery | ForEach { $BatName = $_.name; break }

If ((select-string –pattern  $BatName  -InputObject  $BatterieListe –Quiet)  -eq $True) {
   BatterieListe | Where {$BatName –match $_ } | Foreach {
     Write-Warning “Betroffen:Akku $Name – Übereinstimmungen: *$_*`n“ } }
```

K1

```
Get-pssnapin
get-command -PSSnapin  PSCX
(get-command -PSSnapin  PSCX | measure-object).count
Get-help *pscx
Get-help about_PSCX
```

K2

```
# The scripts starts here. First check: do we have 3 parameters?
If ($Args.count –ne 3) {
        write-warning “call the script with 3 parameter: source target date-in-months”
} else {
If (test-path $Args[0]) {
        If (test-path $Args[1] ) {
        $CheckDate = ((Get-date).AddMonths(- $Args[2])).ToOADate()
        $MyPath = $Args[1]
        Dir $args[0] -recurse | where-object {$_.mode -notmatch "d"} | foreach-object {
                if ( ($_.LastAccessTime).ToOADate() -lt $CheckDate) {
                $myACL = get-Acl $_.FullName
                $MyName = $MyPath + "\" + $_.Name
                Move-Item $_.FullName $MyPath
#Statt xxMKLink verwenden wir nun das PSCX Cmdlet New-Link
                New-Link $_.FullName $Myname /q

                Set-Acl $Myname $myACL
                $Myname = $_.FullName + ".lnk"
                Set-Acl $Myname $myACL
                }
        }
} else { Write-Warning “Target path does not exists” }
} else {Write-Warning (“Source path does not exists: ” + $Args[0]) }
}
```

K3

```
# Aufruf von Outlook 2007 und des Posteingangs (Folder 6)
$Outlook = New-Object -com Outlook.Application
$Inbox = $Outlook.Session.GetDefaultFolder(6)
$Inbox.items | where { $_.Unread }| foreach { out-speech $_.Subject }
```

K4

```
Ping-Host localhost
Ping-Host localhost –q | get-member
(Ping-Host localhost –q).loss
```

K5

```
Send-SmtpMail  -SmtpHost contoso-DC1.Contoso.local -to Administrator@contoso.local
-from info@contoso.local  -subject “PowerShell Information von ” + (dir
env:\Computername).Value – body  “Serveralarm auf “ + (dir env:\Computername).Value +
“ um “ + (get-date)
```

K6

```
cd contoso:  # Vergessen Sie nicht den Doppelpunkt!
CD \; Dir users
Get-ADObject   -class user
```

K7

```
New-Item  PSCX –type OrganizationalUnit
cd PSCX
New-Item PSCX1    –type User
New-Item PSCX2    –type User
New-Item PSCX-Group   –type Group

Get-Itemproperty Pscx1  # der Vorname ist in der Eigenschaft Surname gespeichert
Set-ItemProperty PSCX  Surname  „Frank“

Cd \; Cd HR
Dir
Del *
Dir
```

L1

```
Get-pssnapin
get-command -PSSnapin  SDMGPOSnapIn
(get-command -PSSnapin  SDMGPOSnapIn | measure-object).count
```

L2

```
get-sdmgpo *
get-sdmgposecurity "Default Domain Policy"
get-sdmgplink "OU=Domain Controllers,dc=contoso,dc=local"
```

L3

```
New-SDMgpo "Test-GPO"
Add-SDMgplin -name Test-GPO -scope "OU=HR, DC=contoso, DC=local" -Location -1
```

L4

```
Get-pssnapin
get-command -pssnapin getgpobjectpssnapin

$gpo = Get-SDMgpobject -gponame "gpo://contoso.local/Default Domain Policy"   -
openbyname $true
$stng = $gpo.GetObject("User Configuration/Administrative Templates/Desktop/Active
Desktop/Active Desktop Wallpaper");
$stng.Put("State", [GPOSDK.AdmTempSettingState]"Enabled");
$stng.Put("Wallpaper Name:", "%WINDIR%\Web\Wallpaper\Ascent.jpg");
$stng.Put("Wallpaper Style:", "0");
$stng.Save();
```

Following this, update the group policies, log out and log back in again:

```
Gpupdate /force
```

N

```
# Create new Database
$SQLCn = New-Object System.Data.SqlClient.SqlConnection(
    "Server=.\SQLEXPRESS;Trusted_Connection=Yes")
$SQLCn.Open()
$SQLCMD = $SQLCn.CreateCommand()
$SQLCMD.CommandText = "CREATE DATABASE EventDB"
$Result= $SQLCMD.ExecuteNonQuery()
If ($Result) { Write-Host "Anlegen der Datenbank erfolgreich." }
$SQLCN.Close()


# Create new Table
$SQLCn = New-Object System.Data.SqlClient.SqlConnection( "Server=
   Contoso-DC1\SQLEXPRESS;Trusted_Connection=Yes;Database=EventDB")
$SQLCn.Open()
$SQLCMD = $SQLCn.CreateCommand()
$SQLCMD.CommandText = "CREATE TABLE Events (EvtIX int, EvtTXT varchar(1000), EvtTYPE
    varchar(50), EvtTIME datetime, EvtID int, EvtSrc varchar(100))"
$Result= $SQLCMD.ExecuteNonQuery()
If ($Result) { Write-Host "Anlegen der Tabelle erfolgreich." }
$SQLCN.Close()
```

```
# Create new row / entry
$SQLCn = New-Object System.Data.SqlClient.SqlConnection( “Server=
      Contoso-DC1\SQLEXPRESS;Trusted_Connection=Yes;Database=EventDB”)
$SQLCn.Open()
Get-eventlog system –newest 10 | foreach-object {
          $SQLCMD = $SQLCn.CreateCommand()
          $SQLCMD.CommandText = "INSERT Into Events VALUES(@EvtIX, @EvtTXT,
                      @EvtTYPE, @EvtTIME, @EvtID, @EvtSrc)"
          $P = $SQLCMD.Parameters.AddwithValue("@EvtIX", $_.Index)
          $P = $SQLCMD.Parameters.AddwithValue("@EvtTXT", $_.Message)
          $P = $SQLCMD.Parameters.AddwithValue("@EvtTYPE", $_.Category)
          $P = $SQLCMD.Parameters.AddwithValue("@EvtTIME", $_.TimeGenerated)
          $P = $SQLCMD.Parameters.AddwithValue("@EvtID", $_.EventID)
          $P = $SQLCMD.Parameters.AddwithValue("@EvtSrc", $_.Source)
          $Result= $SQLCMD.ExecuteNonQuery()
          If ($result –eq $False) { Write-Host "Fehler beim Datenschreiben" }  }
$SQLCN.Close()


# Read row / entry
$SQLCn = New-Object System.Data.SqlClient.SqlConnection( "Server=
           Contoso-DC1\SQLEXPRESS;Trusted_Connection=Yes;Database=EventDB")
$SQLCn.Open()
$SQLCMD = $SQLCn.CreateCommand()
$SQLCMD.CommandText = "SELECT * FROM Events"
$Results = $SQLCMD.ExecuteReader()
While ( $Results.Read() ) {
          Write-Host $Results.Item("EvtIX") $Results.Item("EvtTIME")
              $Results.Item("EvtTXT")
}
$SQLCN.Close()
```

More sample scripts

X1 # List of all files from all subfolders with statistics concerning:

- **File type (.doc, .TXT. etc.)**
- **Number of files per type (5, 10, etc.)**
- **Overall size of all files of this type (1345 KB)**

# Script content:

```
New-PSDrive -name FK -PSProvider FileSystem -root c:\Downloads\Files | out-null
get-childitem FK:\ -recurse | where-object {$_.mode -notmatch "d"} |
Group-Object extension | foreach { write-host $_.name, $_.count, (Get-childitem ("FK:\*"+
$_.name) -recurse |  measure-object length –sum).sum }
```

X2 # Saving the result from X1 in Excel instead of on the screen:

```
New-PSDrive -name FK -PSProvider FileSystem -root c:\downloads\files | out-null
$a = New-Object -comobject Excel.Application
$b = $a.Workbooks.Add()
$c = $b.Worksheets.Item(1)
$c.Cells.Item(1,1) = "Dateityp"
$c.Cells.Item(1,2) = "Anzahl"
$c.Cells.Item(1,3) = "Gesamtgrösse"
$i = 2
get-childitem FK:\ * -recurse | where-object {$_.mode -notmatch "d"} |
Group-Object extension | foreach {
        $c.cells.item($i,1) = $_.Name
        $c.cells.item($i,2) = $_.count
        $c.cells.item($i,3) = (Get-childitem ("FK:\*"+ $_.name) -recurse |  measure-object
length –sum).sum
        $i = $i + 1
}
$b.SaveAs("C:\Downloads\Scripts\Report-X2.xlsx")
$a.quit()
```

ANNEX: PREPARING THE TEST ENVIRONMENT

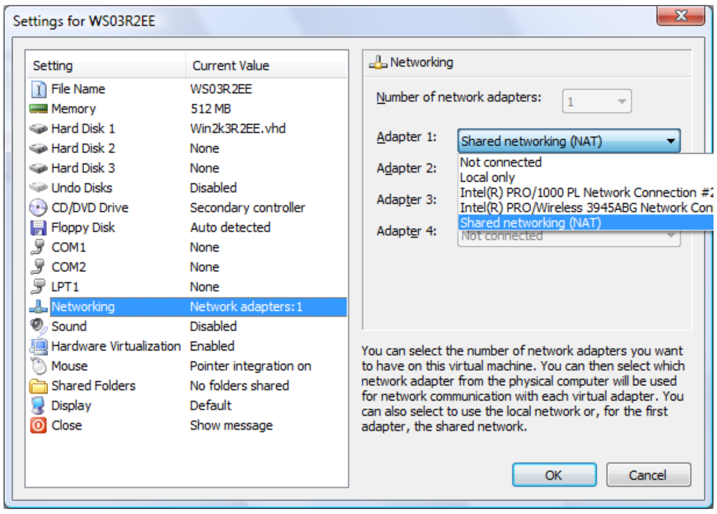
The test environment for this Windows PowerShell exercise booklet consists of a Windows Server 2003, configured as a domain controller. If you have no Windows server available for use, you can download a virtual hard drive free from Microsoft, which you can also use on the (also free) VirtualPC 2007 (or Virtual Server 2005 R2). Please note that your PC must have at least 1 GB of RAM available. You can use this test environment for 30 days. Entering the valid server product key (e.g. from your TechNet or MSDN subscription) will enable you to use the virtual environment until summer 2008. For general queries on using VirtualPC, please also read http://www.microsoft.com/windows/products/winfamily/virtualpc/default.mspx

Download link for VirtualPC 2007:
http://www.microsoft.com/downloads/details.aspx?displaylang=de&FamilyID=04d26402-3199-48a3-afa2-2dc0b40a73b6

Download link for virtual hard drive for Windows Server 2003:
http://www.microsoft.com/downloads/details.aspx?FamilyID=77f24c9d-b4b8-4f73-99e3-c66f80e415b6&DisplayLang=en (two downloads with a total data volume of around 1.5 GB!)

First install VirtualPC 2007 and then unzip the virtual harddisk for the Windows Server 2003 by starting the EXE file. Once all files are unzipped, launch the virtual environment. You can do this most easily by double-clicking on the unzipped WS03R2EE.vmc file.

You must now make one setting on the VirtualPC to use the test environment as required. Set up two network adapters for your new virtual PC. Set virtual network adapter 1 to ***Shared networking (NAT)***, and adapter 2 to the value ***Local Only*** (under *Settings* for the WS03R2EE PC):



Settings for WS03R2EE

Setting	Current Value
File Name	WS03R2EE
Memory	512 MB
Hard Disk 1	Win2k3R2EE.vhd
Hard Disk 2	None
Hard Disk 3	None
Undo Disks	Disabled
CD/DVD Drive	Secondary controller
Floppy Disk	Auto detected
COM1	None
COM2	None
LPT1	None
Networking	Network adapters: 1
Sound	Disabled
Hardware Virtualization	Enabled
Mouse	Pointer integration on
Shared Folders	No folders shared
Display	Default
Close	Show message

Networking

Number of network adapters: 1

Adapter 1: Shared networking (NAT)

Adapter 2:

Adapter 3:

Adapter 4: Not connected

Not connected
Local only
Intel(R) PRO/1000 PL Network Connection #2
Intel(R) PRO/Wireless 3945ABG Network Con
Shared networking (NAT)

You can select the number of network adapters you want to have on this virtual machine. You can then select which network adapter from the physical computer will be used for network communication with each virtual adapter. You can also select to use the local network or, for the first adapter, the shared network.

Configuring the test environment

First, set up an Active Directory for working in the test environment. To do this, log onto the virtual server (details of how to do this can be found in the Readme file for the download). To use the exercises and solution scripts from this exercise booklet without making any changes to them, you must change the name of your server to ***Contos-DC1***. You can do this in the Start menu by clicking on ***My Computer*** with the right mouse button and selecting ***Properties***. On the ***Computer Name*** tab, click on the ***Change*** button and enter the new name. You must then restart your virtual server. After restarting, log back onto the server to set up the Active Directory. First, assign a fixed IP address to the second network adapter of your test system (the network environment containing the network adapters can be found under Control Panel in the Start menu). You can assign, for instance, 10.0.0.1 as a fixed IP address.

You now need to install the Active Directory. To do this, click on ***"RUN…"*** in the server Start menu. Enter ***DCPROMO*** and click on ***OK***. The Active Directory installation wizard is launched, and will ask you for various pieces of information. Please enter the following answers on the corresponding entry screens:

- Create a "*New Domain Controller"* in the domain*"Domain in a new Forest".*
- Select *Contoso.local* as the DNS name and *CONTOSO* as the NetBios name.
- Accept the suggested file paths.
- Let the wizard install and configure the DNS server (default setting).
- Confirm the default value for Windows compatibility.
- Select (and make a note of!) a password of your choice to be used as a restore password.

If you are asked for the path for the installation files, they are located on the C: drive in the directory labelled "WindowsInstallationFiles\i386" on the virtual hard drive. Depending on the network configuration, you may also need to confirm that you wish to work with dynamic IP addresses in your test environment, even if this is ***NOT*** the recommended configuration for a live Active Directory. Installation is completed by rebooting. Following the reboot, log back in again. Close all windows that have opened automatically upon login.

You must now install the .NET Framework 2.0. This can be done in several ways; here, we will use Microsoft Update. First of all, deactivate the "Enhanced Security Configuration" via the menu item "Add / Remove Windows Components" in the Control Panel. However, beware – this setting is ***not recommended*** for live servers, as it subjects your server systems to unnecessary risks. Only use this setting in your virtual test environment. Then, update your PC using the Start menu entry "Microsoft Update", and install all necessary system updates for Microsoft Update.

You can then choose which updates to install on your virtual server. To do this, select "Custom" on the Microsoft Update site. If you don't want to install all updates, delete the default selection SP2 and also remove all other updates from the selection. You will find the .NET Framework under the heading "***Software, optional***" on the Microsoft Update site. This is the only update that you absolutely must install in order to complete the Windows PowerShell course on your virtual server. However, all other updates are highly recommended for your other live systems.

Should you have problems accessing the Internet, ensure that your host system has Internet access, and that your DNS configuration on the guest system is correct. Only the IP address ***192.168.131.254*** must be entered in the DNS server MMC in the properties of the DNS server Contoso-DC1.local as the forwarder. This corresponds to the IP address of your host system, which can now take on the DNS forwarder role.

Installation of a simple e-mail environment

The Windows Server 2003 is supplied complete with a POP3 e-mail server, which is sufficient for the later exercises on automatically sending e-mails using Windows PowerShell. Outlook Express, a simple e-mail client, is also available for checking the e-mails. Although these do not compare with the power of Exchange and Outlook, they still allow you to do a fair amount. So, install the POP3 services by selecting ***Add or Remove Programs*** in the ***Control Panel***, and clicking on ***Add/Remove Windows Components***. In the***Windows Components List***, select ***E-mail-Services*** and click on ***Next***. After installing the e-mail services, launch the administrative interface for the services from the Start menu. You will find the ***POP3 Services*** MMC in the ***Administrative Tools***.

In the POP3 service MMC, select your server (Contoso-DC1) and create a new domain. This is not an Active Directory domain, but instead an e-mail SMTP domain. Call this domain ***contoso.local***

Then, set up two inboxes, one for the administrator and one called ***Info***. Automatically create a new user for Info (default setting).

Now, all that's left to do is set up Outlook Express. To do this, start Outlook Express from the Start menu. A configuration wizard is automatically launched. Choose any display name you like for yourself, and use ***administrator@contoso.local*** as your account. Your mailserver is a POP3 server. Enter the name of your server (normally***contoso-dc1.contoso.local***) as both the incoming and outgoing server. Use "Data Administrator" as your login ID, and your Windows password (you can check the box to save the password), and don't forget to check the box for using Secure Password Authentication (SPA).

You can now write your very first e-mail to check your e-mail environment. Enter ***info@contoso.local*** and ***administrator@contoso.local*** as the recipient. After sending the e-mail, don't forget to check for new e-mails using ***Send/Receive…***, as you will hopefully have received your own e-mail. If the e-mail doesn't appear, or if you receive an error message, check the POP3 configuration and the Outlook Express configuration for possible errors:

- In the POP3 service MMC, you can, for instance, check if the e-mails have even reached the accounts (number of messages on the server).
- Outlook Express provides detailed error descriptions in the event that a server cannot be contacted. Check the name closely for any typing errors, and whether you have checked the box for using SPA as mentioned above.

Further reading

The following are some of my favourite books on Windows PowerShell. As each person has their own preferences in terms of author and writing style, you should take a look at other books as well. If you have any book recommendations or tips, I'd like to hear them.

Language:	English or German
Format:	Soft cover/e-book
No. of pages:	576
ISBN-10:	1932394907
ISBN-13:	978-1932394900

"Windows PowerShell in Action" is a programming tutorial for system administrators and developers. It provides the reader with a comprehensive introduction to the language and environment of Windows PowerShell. The book shows you how to develop scripts and utilities to automate system tasks, or create powerful system management tools to handle the day-to-day tasks involved in Windows system administration. The book also covers topics such as batch scripting and string processing, COM and WMI, as well as .NET and WinForms programming.

Bruce Payette, the author, also shows why PowerShell was made the way it was, and how it works. Bruce's knowledge as one of the chief developers of the language provides the reader with the profound understanding required to get the most out of PowerShell.

http://www.manning.com/payette/

PowerShell cheat sheet

Important commands

To get help on a cmdlet: get-help

```
Get-Help Get-Service
```

List of all available cmdlets: get-command

```
Get-Command
```

All properties and methods of an item: get-member

```
Get-Service | Get-Member
```

Setting the security policy

Reading and changing security policy:

Get-Execution und Set-Execution policy

```
Get-Executionpolicy
Set-Executionpolicy remotesigned
```

Executing a script

```
powershell.exe –noexit &"c:\myscript.ps1"
```

Functions

Parameters are separated by a blank. Return value is optional.

```
function sum ([int]$a,[int]$b)
{
     return $a + $b
}
sum 4 5
```

Variables

Begin with $

```
$a = 32
```

Type specification:

```
[int]$a = 32
```

Fields

Initialization:

```
$a = 1,2,4,8
```

Query:

```
$b = $a[3]
```

Constants

Are created without $:

```
Set-Variable –name b –value 3.142 –option constant
```

And queried with $:

```
$b
```

Using objects

To generate a new instance of a COM object:

New-Object -comobject <ProgID>

```
$a = New-Object –comobject "wscript.network"
$a.username
```

To generate an instance of a .Net Framework object (parameters can be transmitted when necessary): New-Object –type <.Net Object>

```
$d = New-Object -Type System.DateTime 2006,12,25
$d.get_DayOfWeek()
```

Windows PowerShell

Output to console

Variable name

```
$a
```

or

```
Write-Host $a –foregroundcolor "green"
```

Using user entries

Read-Host reads user entries:

```
$a = Read-Host "Enter your name"
Write-Host "Hello" $a
```

Command line arguments

Must be transmitted with blanks

```
myscript.ps1 server1 benp
```

Can be used in script with $args field

```
$servername = $args[0]
$username = $args[1]
```

Miscellaneous

Link break: ` (Shift ^ + space)

```
Get-Process | Select-Object `
name, ID
```

Comments: #

```
# code here not executed
```

Several commands in one $a=1;$b=3;$c=9

$a=1;$b=3;$c=9

Forward output: | (AltGr 7)

```
Get-Service | Get-Member
```

“Do-while” loop

Repeats loop as long as “do-while” condition is met

$a=1

Do {$a; $a++}

While ($a –lt 10)

“Do-until” loop

Repeats loop until the “until” condition is met

$a=1

Do {$a; $a++}

Until ($a –gt 10)

“For” loop

Repeats commands a set number of times

```
For ($a=1; $a –le 10; $a++)
{$a}
```

“For each” – loop for groups of objects

Processes groups of objects:

```
Foreach ($i in Get-Childitem c:\windows)
{$i.name; $i.creationtime}
```

“If” condition

Executes code when “if” condition is fulfiled

```
$a = "white"
if ($a -eq "red")
      {"The colour is red"}
elseif ($a -eq "white")
      {"The colour is white"}
else
      {"Another colour"}
```

“Switch” condition

Another option for executing code when a condition is met

```
$a = "red"
switch ($a)
{
      "red" {"The colour is red"}
      "white"{"The colour is white"}
      default{"Another colour"}
}
```

Get content from a file

Get-Content creates a field from the lines (objects) in the file. Use “for each” loop afterwards:

```
$a = Get-Content "c:\servers.txt"
foreach ($i in $a)
{$i}
```

Write to a simple file

Use Out-File or > for simple text files

```
$a = "Hello world"
$a | out-file test.txt
```

Or use >:

```
.\test.ps1 > test.txt
```

Write to a HTML file

Use ConvertTo-Html followed by >

```
$a = Get-Process
$a | Convertto-Html -property Name,Path,Company > test.htm
```

Create a CSV file

Use Export-Csv and Select-Object to filter output:

```
$a = Get-Process
$a| Select-Object Name,Path,Company | Export-Csv -path test.csv
```

Acknowledgements

Following the overwhelming success of the first workbook, I received lots of requests for a second part. I stuck with the idea of using real-life examples from the day-to-day work of an administrator, meaning that the content was quickly defined. Unfortunately, my actual career took up so much of my time that the publication date kept getting pushed back. So I'm very pleased that at least the year of publication is still being kept. I hope that you, the reader, have enjoyed both this book and Windows PowerShell, and that you are, like me, impressed by its capability and simple elegance.

This book is based on the work of many helpers who have created labs and practice exercises for Windows PowerShell, or have carefully tested first drafts of this book. They deserve the credit here.

If you have any suggestions or feedback regarding this book, or you would like to see another book of this kind (for instance on the Windows Server 2008 or Windows PowerShell V2), please e-mail me. Although I may not be able to send replies to all e-mails, I will be happy to receive any type of constructive criticism or positive feedback. My e-mail address is frankoch@microsoft.com.

And if you're surprised that this little book with its 70-odd pages has an acknowledgements page, you're in the same boat as my colleague Sascha Corti, who I'd like to give my warmest regards to here as well.