Windows PowerShell

An introduction to scripting technologies for people with no real background knowledge

MICROSOFT SWITZERLAND

1 Oktober 2007
Frank Koch (BERN)
Developer & Platform Evangelism

Windows PowerShell

What can you expect from such a short work - An attempt of an foreword

Objectives in drafting this book

This "Windows PowerShell" book will give you an introduction to Windows PowerShell as well as practical examples, in order to give you a quick introduction to this subject even if you have no significant previous scripting experience. The book is explicitly not aimed at professional scripters; the extensive Windows PowerShell help and the many Internet forums and additional literature are designed to give experts everything they need. However, the newbies amongst you will hopefully find everything you need in this book to enable you to think about scripting in more detail, and hopefully quickly learn to enjoy being able to operate a computer even without a mouse.

The sources for this book were mainly Microsoft publications relating to Windows PowerShell. This book presents the information in a new way appropriate for its target audience, initially leaving out the theory whilst using many examples and mini practical exercises so as to hold your interest.

Once you have worked through this text and decided to make Windows PowerShell a fixed part of your everyday IT, feel free to read the original documentation for Windows PowerShell, which is automatically included when you install Windows PowerShell:

- First Steps with Windows PowerShell
- Basic Principles of Windows PowerShell

To really profit from this book you should have access to a PC where you can simultaneously practice the exercises. The only prerequisite is a PC with Windows PowerShell 1.0 installed, which is available free as of Windows XP SP2. For information on downloading and installing PowerShell see the following websites.

Other sources of information online

The introductory page for Widows PowerShell including download link is www.microsoft.com/PowerShell Here you will also find other links to very useful webcasts, books and other help forums.

The best blog pages about Windows PowerShell are found at http://blogs.msdn.com/PowerShell/ Here you can read information about scripting techniques and practical demos. Absolutely everything.

In Switzerland you can also find information in German in the ITPro team blog at http://blogs.technet.com/chITPro-DE Here you can find links to German Windows PowerShell webcasts, and downloads for the exercises in the book in the March/April 2007 archive.

Useful key combinations for Swiss standard keyboards

Symbol	Key combination	Meaning
|	AtgGr 7 (Not: AltGr1 = ¦)	Forward the output from a command
`	Shift ^, then a space	Continue command on next line
{	AltGr Ä	Begin a command sequence (e.g. after an If Statement)
}	AltGr $	End a command sequence (e.g. for an IF Statement
[	AltGr Ü	Sometimes required for objects
]	AltGr !	Sometimes required for objects
TAB	Tab key	Completes commands where necessary Example: Get-He (TAB) produces Get-Help

Windows PowerShell was developed in Redmond and was designed to ideally fit American keyboard layouts. Some commonly used keys are therefore hard to find on Swiss keyboards. I have put together a list for you here.

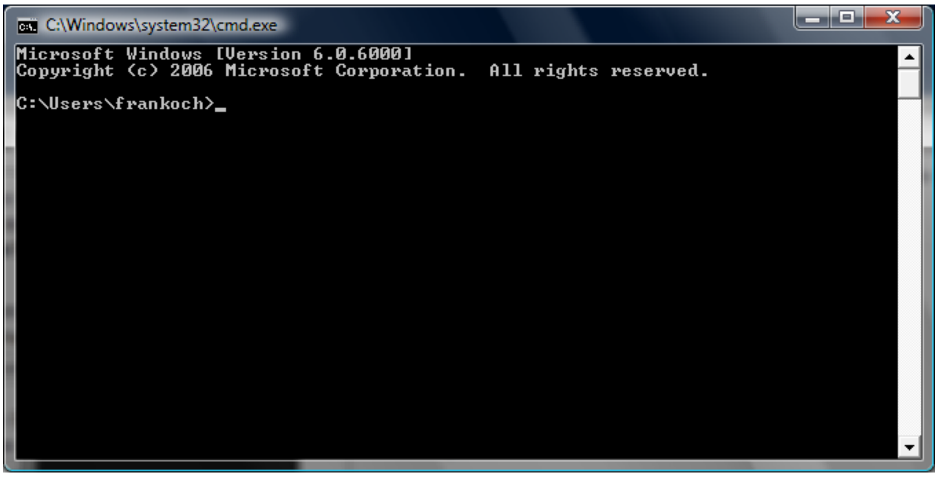
Contents

Windows PowerShell in Practice

Windows PowerShell is a free add-on for Windows XP systems and above, and can be downloaded from Microsoft at http://www.microsoft.com/powershell. The prerequisite is .NET Framework 2.0 that – if not yet installed – requires a separate download and installation. Windows PowerShell itself is a relatively small download of around 1.5 MB and can easily be installed automatically via software distribution. The only thing to consider is that there is a different version of Windows PowerShell for each version of Windows. You must also take into account the relevant 32bit and 64bit architecture. After installation, Windows PowerShell places itself in the Start menu and is then accessible by clicking on the shortcut or by entering “PowerShell” into the Windows run command.

First impressions of Windows PowerShell

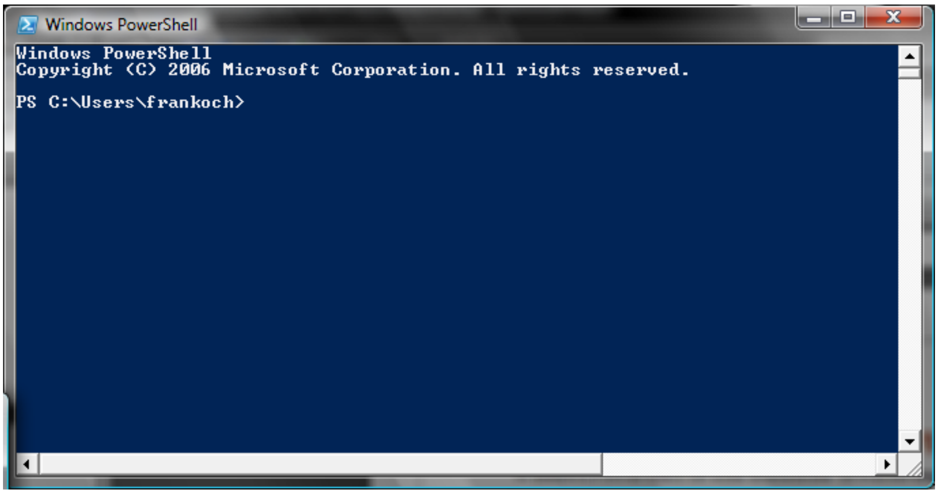
To get your first impressions, start both Windows PowerShell and the classic command line CMD from the Start menu. At first glance both shells appear very similar – apart from their distinctive colors:



```
Microsoft Windows [Version 6.0.6000]
Copyright (c) 2006 Microsoft Corporation.  All rights reserved.

C:\Users\frankoch>_
```

Figure 1: The classical CMD command line



```
Windows PowerShell
Copyright (C) 2006 Microsoft Corporation. All rights reserved.

PS C:\Users\frankoch>
```

Figure 2: Windows PowerShell

And it’s no wonder, since both shells use the same "input container". This unfortunately also means that Windows PowerShell suffers from the real lack of support for Copy/Paste just like CMD has done for years. The following tips may be of some use here:

- Select the required text with the mouse
- Press the right mouse button (= copy)
- Place the cursor on the required position
- Press the right mouse button (= paste)

Try it yourself. Copy the first text line from each shell (e.g. “Copyright (c) 2006 Microsoft Corporation”) then paste them as a 'command line'. Don’t worry about the error message that appears when you press Enter. Get used to these mouse button exercises; you will often use them later on.

Even if the ‘container’ is the same, the content and function of each of the two shells is very different. The easiest way to discover this is to take a look at the online help. A first look makes it clear that Windows PowerShell offers many more functions than CMD; more than 100 commands, also called cmdlets (pronounced “*Commandlet”*). Since only a few commands were directly contained within the CMD, additional help programs were developed over the years. As each of these CMD programs has its own syntax, an experienced CMD expert has to learn by heart many different commands and command logics. For cmdlets the syntax and logic are clearly defined. Windows PowerShell commands follow a specific naming rule:

- Commands in Windows PowerShell consist of a verb and a noun (always singular) separated by a hyphen. Commands are in English. Example:
  get-help calls up the online help in Windows PowerShell syntax
- Parameters are prefixed by – :
  get-help -detailed
- We also included many well-known commands in order to make it easier to start using Windows PowerShell. Example:
  As well as *get-help*, *help* (Windows classic) and *man* (Unix classic) all work the same way.

Display the different help texts for each shell. Enter the *help* command in each shell. You will see the different number of documented commands in each edition of help.

Instead of *help* or *man* you can also use *get-help* in Windows PowerShell. The syntax is as follows:

- *Get-help* gives you help on how to use the help
- *Get-help** lists all Windows PowerShell commands
- *Get-help command* lists the help for the relevant command
- *Get-help command –detailed* lists detailed help with examples of the command

Use the help command to find detailed information about the help command: *get-help get-help –detailed*. Tip: Use the TAB key to automatically complete the command. This avoids typos.

Additional uses for output: “piping” objects

As mentioned before, Windows PowerShell is an object-oriented shell. That means that the input and output for commands are usually objects. Since humans cannot read objects, Windows PowerShell ‘translates’ the objects for output on the screen in text (professionals can even find commands in the Windows PowerShell help that enable them to adjust the output to fit their requirements). Linked commands are represented by the ‘pipe’ command: |

You can use this link to create your own Windows PowerShell book: *Get-help * | get-help –detailed* does this for you: *get-help ** creates a list of known commands that we use to input the command *get-help –detailed*. The output is extensive; you can cancel using CTRL-C.

To be able to use the results of a “help book” at a later point, it is best to move the output into a file instead of displaying it on-screen. Windows PowerShell has its own command *Out-File*, better known as the symbol >.

Create your own ‘book file’ now; input the following command: *Get-help * | get-help –detailed | out-file c:\Powershell-Help.txt* or even
*get-help * | get-help –detailed > c:\PowerShell-Help.txt.*
Note that you must have write authorization for the destination path (here: c:\ ).

Then open your first help file in Notepad and use this as online help for the later exercises.

If you are ever searching for a command then *get-help* can also help you here. If you want to sort something, try to find something suitable using *get-help sort**. *Get-help* now starts to search for a relevant command in the Windows PowerShell command repository. Since all commands start with a verb we can structure the search easily using *get-help“relevant English verb”**. If you don’t know already, the * symbol expresses a wildcard search, meaning that anything can come after the actual search text, we just don’t know yet and want to find everything that starts with our search text.

Once you have displayed a command (here it would be *sort-object*), simply call up *get-help* again, but this time with the relevant command and the parameter *–detailed* in order to find examples of how to use the command:

get-help sort-object –detailed.

You should now be able to solve your problem.

Initial exercises with Windows PowerShell objects

If you have never worked with objects before, the following exercises can help you understand the myriad opportunities in this world. Objects in programming are nothing new, and yet there is nothing to compare in the scripting area. If you are interested in working with objects, you can find detailed secondary literature on the Microsoft MSDN pages at http://msdn.microsoft.com and http://www.microsoft.com/switzerland/msdn/de/default.mspx. Let’s look at objects using the “process” object as an example. If “process” does not mean anything to you, think of what you see on your screen when you call up the Task Manager. If you are interested in the “process” object the MSDN pages can also be of help.

Working with processes

The *get-process* command lists all processes in your system. This list can be very long. To sort the list you can use another cmdlet: *sort-object*. By default, *sort-object* has a fixed sort order (ascending) which can be overridden with a parameter called *-descending*. For the argument you enter the object properties you want to use to sort the results, e.g. CPU time.

A1: Your task now is to generate a list of all processes and to sort these in descending order according to their CPU time. You have already learnt everything you need to know for this: *get-process, sort-object*, and the pipe |. Hint: CPU is not a parameter for *sort-object*, rather it is an argument you can use for sorting. It therefore has no – symbol.

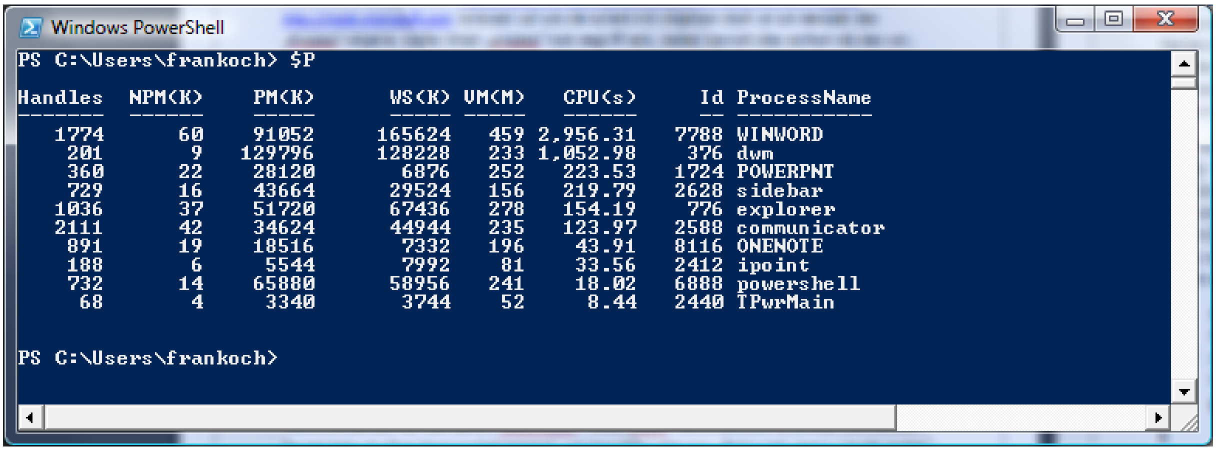
In the next exercise we want to limit the list a little to make it easier to handle. We use the command *select-object*. *select-object* recognizes several parameters (use *get-help* to find these), but we only need *–first x* and *–last y* to find the first x or last y objects in a list e.g. *select-object –first 5. select-object* alone will not work, rather the cmdlet expects input such as from a pipe.

A2: Generate a list of the top 10 processes based on their CPU time. To do this, take the results from exercise A1 and add the command *select-object*. There are two ways to the ideal solution, depending on how you want to sort the list. Try to find both ways. Hint: one way uses the parameter *–first*, the other uses *–last*.

We use this exercise as a quick introduction to variables. In simple terms, variables store all possible values, although they can also be objects! Here too please refer to the additional literature for a more in-depth view of variables. For now you only need to know that variables in PowerShell must always start with $. You can store the result from exercise A2 in a variable, allowing you to access your list of top 10 processes at any time. This allows you to compare with other points in time, perhaps to evaluate what has changed in a system. Assigning a variable is easy:

$a = get-process | sort-object CPU -de…

A3: Assign variable $P to the abbreviated process list from exercise A2. Hint: Use the cursor key “*Up*“ to call up the last used command and use the *Home* key to move the cursor to the beginning of the line, then input your data. You can output the variable in a second command simply by typing *$P* at the prompt.



```
PS C:\Users\frankoch> $P

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
   1774      60    91052     165624   459 2,956.31   7788 WINWORD
    201       9   129796     128228   233 1,052.98    376 dwm
    360      22    28120       6876   252   223.53   1724 POWERPNT
    729      16    43664      29524   156   219.79   2628 sidebar
   1036      37    51720      67436   278   154.19    776 explorer
   2111      42    34624      44944   235   123.97   2588 communicator
    891      19    18516       7332   196    43.91   8116 ONENOTE
    188       6     5544       7992    81    33.56   2412 ipoint
    732      14    65880      58956   241    18.02   6888 powershell
     68       4     3340       3744    52     8.44   2440 TPwrMain


PS C:\Users\frankoch>
```

Figure 3: Output for variable $P

Output in a TXT, CSV or XML file

As default, Windows PowerShell displays the results of a command chain on-screen. Any objects are converted into text so that humans can read them. We use the *out-host* command here. However, because Windows PowerShell wants to be efficient, this is added automatically and invisibly if you do not add it yourself. There are alternatives to *out-host*; find them using *get-help out**.

Outputting results as a text file is quick and easy: *out-file filename* is the solution. Many other shells use the > command, which is also supported by Windows PowerShell. As well as output as a text file you can also convert to a CSV or XML file. As with *out-host* there are separate cmdlets that perform this task for you. They are called *export-CSV* and *export-CliXML*; both require the file name as the argument. And yes, you're right: if you can export, then you can import. Use *import-CSV* or *import-CliXML* to import the files again for viewing.

A4: Take the variable $P from exercise A3 and save its contents in a text file called "A4.txt". Then save the contents of $P in a CSV file called "A4.CSV", and finally again in an XML file "A4.XML". Hint: the command > directly replaces the pipe |, which is only required for true cmdlets such as *out-file*, *export-CSV* etc. Look at the results; Notepad is sufficient for this.

Output in color

Sometimes you want to highlight results to make them easier to read. You can do this, for example, by using color. The *write-host* command recognizes several parameters such as *–foregroundcolor* and *–backgroundcolor*. What do you think, what might be the output for the following?

write-host "Red on blue" –foregroundcolor red –backgroundcolor blue

You've guessed it. *get-help write-host –detailed* gives you a list of possible colors. There are also predefined combinations: with *write-warning "error"* you can also attract the user's attention. Try this now. With this command you can output all processes in color. However, it would be more attractive if we could color the list according to additional conditions. Let's take a closer look at that. For simplicity's sake we use your PC's services instead of the processes. If you do not know what services are, please look it up, for example on the MSDN pages. Simply put, services are the things you see listed under *Control Panel / Administrative Tools / Services*. The nice thing about them is that they appear with status “started“ and “stopped“, which is really useful for color output. But first let’s look at the services using the *get-service* cmdlet.

A5: Generate a list of all services and sort them by status. Hint: Use the same method for sorting processes by CPU time, but use *get-service* and “*status*“ as the argument for *sort-object*.

Now we want to output an entire list in red. *write-host* will be our friend here. Unfortunately, the *get-service | write-host –forgroundcolor red* command does not work as we hoped. *write-host* is not as friendly as other cmdlets and takes a list of objects that are then output in the right color. *write-host* has to know for each object, which object attribute is output in what way. We have to help *write-host* along a bit. First we work through the object list step by step, by making a loop. There are many loops; each has its own meaning and areas for use. For our purposes we use the *ForEach-Object* loop, which goes through the object list and passes each object individually to the next cmdlet. Within the loop we select the relevant object using the *$_* abbreviation, a particularly “random feature” of PowerShell Developer. Therefore you select an object attribute using *$_.name-of-the-property*. Let’s look at that in an example:

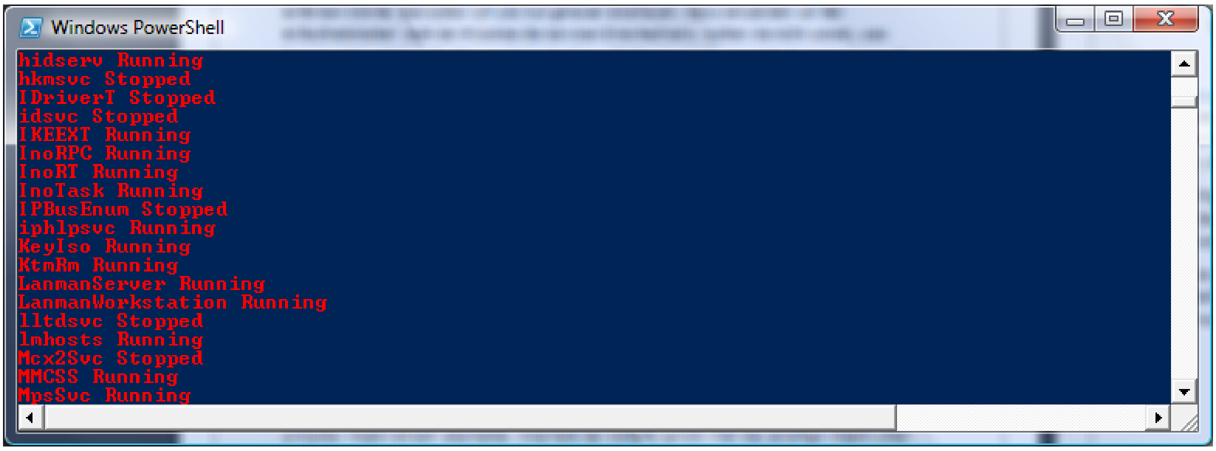
```
get-process | ForEach-Object { write-host $_.ProcessName $_.CPU}
```

Whereas *get-process* gives us a list of processes, in this example we can only see the name and the CPU time, since the first output goes to the pipe (|) and afterwards *write-host* only issues two attributes for each object. (If you call up the example again do not be surprised by the results: not every process has a CPU time, meaning that in certain circumstances a line can consist of only a name).

A6: Generate a list of services and output only the name and status attributes. Use the previously described ForEach loop, even if you can think of other possible solutions.

Now we can use the options for *write-host* to output in color.

A7: Generate a list of services and output only the name and status attributes of your choice in color. Show the colors to your neighbor and decide who chose the prettier combination. Hint: Take your solution from A6 and add the *–foregroundcolor* and *–backgroundcolor* parameters.



```
hidserv Running
hkmsvc Stopped
IDriverT Stopped
idsvc Stopped
IKEEXT Running
InoRPC Running
InoRT Running
InoTask Running
IPBusEnum Stopped
iphlpsvc Running
KeyIso Running
KtmRm Running
LanmanServer Running
LanmanWorkstation Running
lltdsvc Stopped
lmhosts Running
Mcx2Svc Stopped
MMCSS Running
MpsSvc Running
```

Figure 4: Output service names and status in red

Checking conditions using the *if* cmdlet

The last thing we are missing is how to check conditions. Here too there are many options for various scenarios. However, for now in this introduction we will restrict ourselves to the *if* command. You probably already know the syntax from other uses; it is quite simple:

```
if (condition) {command(s) to execute}
elseif (condition2) {command(s) to execute}
else{command(s) to execute}
```

elseif is optional and not always necessary. If you want to execute more than one command in the {} area, you can separate them either with a semi-colon or use a new line for each command. Windows PowerShell simply waits for the } at the end.

For comparison purposes Windows PowerShell recognizes several comparison operators. They all start with “–“ and usually consist of a two-letter English abbreviation: -eq stands for ‘equals’. The most important for us are as follows

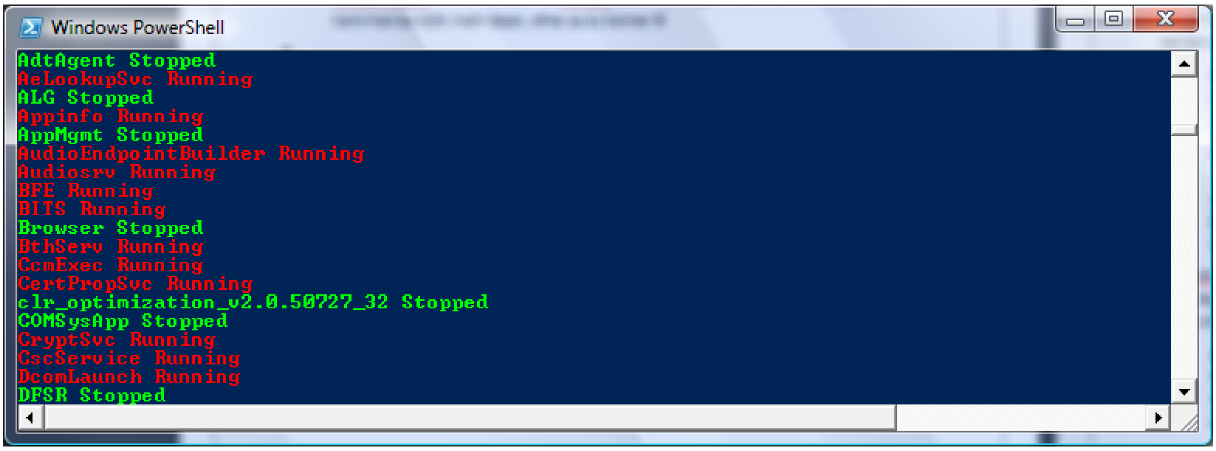
-eq	**Equals**
-match	Match (Not quite as definitive as equals)
-ne	Not equal
-notmatch	Does not match
-gt –ge	Greater than / Greater than or equal to
-lt –le	Less than / Less than or equal to

The last exercise in this block combines all points for initial status monitoring of a system using PowerShell. All services in a system are first sorted by their status and then output in color: services with the status “stopped” in red, services with the status “running” in green.

A8: Call up a list of all services. Sort the list by status, then color the output either red or green, depending on whether the service is "stopped" or "running". Hint: First use *sort-object* as in the previous exercises. Then use the *foreach* loop but instead of only using *write-host*, add an *if* query. You can view the status of services as usual using *$_.status*; possible values are "*stopped*" or "*running"*. About the syntax: The *if* condition is placed in () and the output command in {}. Ignore the theoretical options and for simplicity's sake do not use the *elseif* query. Do not forget the final } for *ForEach*! When you reach the end of a >> line, close it with 2x Return to execute the wrapped lines.

Run the solution again but this time without the cmdlet *sort-object.* Use the cursor keys so that you do not have to enter all the information again.

foreach-object can be abbreviated to *ForEach*. You can make it even shorter, but then you can no longer read it without knowing the meaning of the symbol. And so we will, for now, stick with *foreach-object* or *ForEach.*



```
AdtAgent Stopped
AeLookupSvc Running
ALG Stopped
Appinfo Running
AppMgmt Stopped
AudioEndpointBuilder Running
Audiosrv Running
BFE Running
BITS Running
Browser Stopped
BthServ Running
CcmExec Running
CertPropSvc Running
clr_optimization_v2.0.50727_32 Stopped
COMSysApp Stopped
CryptSvc Running
CscService Running
DcomLaunch Running
DFSR Stopped
```

FIGURE 5: COLOR OUTPUT FOR SERVICES

Output in HTML

Example A8 might be of use in monitoring servers. It would now be good if it was easier to reuse the output. You already know how to output as CSV and XML. And yet there is another version that can sometimes be more useful: HTML. The cmdlet *convertto-html* is used for this. Output is no longer in the form of a file, rather as with the other cmdlets it is now in a form that you can edit directly using a pipe. At the end you should move the text into a file so that, for example, it is easier to view the text in a Web browser. Using a series of mini examples we will now reveal the various options available using *convertto-html*.

A9: Convert the output from *get-service* into HTML. Use the cmdlet *convertto-html*, which can work directly with an object list. Hint: If the list gets too long you can cancel with CTRL-C.

A10: At the end, use the commands you now know to move the output into the file ".\A10.html". View this file. Hint: You can use *invoke-item .\a10.html* to start your default browser and the output file directly from PowerShell. Don't forget to enter the correct path to A10.html! If you prefer you can open the file with the file explorer, too.

convertto-html allows you to limit the output so that the list does not become indecipherable. Use *convertto-html* to call up the list of objects to output, e.g. *… | convertto-html –property name, status*.

A11: Continuing on from A10: Generate a prettier webpage and list only the names and status of all services. You can also sort the output by status **before** conversion. Hint: Your command line will then consist of 4 commands:
list all services, sort by status, convert to HTML, output as a file.

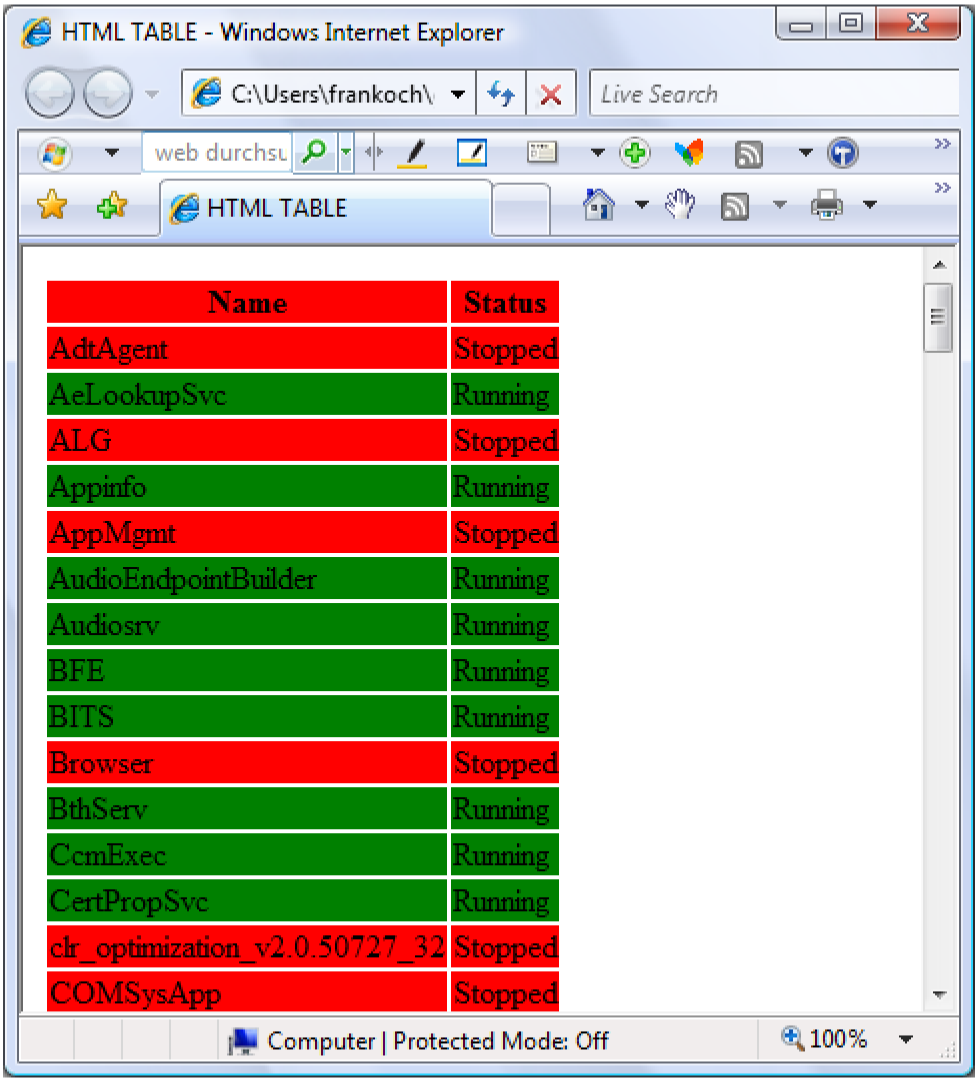
Because *convertto-html* creates HTML text, the output can be easily modified if you have HTML experience. This is not a task for Windows PowerShell, but PowerShell can support you. Try to work out the results of the following code before you copy/type and execute it in Windows PowerShell:

```
get-service | ConvertTo-Html -Property Name,Status | foreach {

if ($_ -like "*<td>Running</td>*") {$_ -replace "<tr>", "<tr bgcolor=green>"}

else {$_ -replace "<tr>", "<tr bgcolor=red>"}} > .\get-service.html
```

The output file should look similar to the one below. In principle this example works in the same way as the *write-host* output, although here the individual lines of the HTML file are reformulated: the HTML command for the table column has a background *bgcolor green* or *bgcolor red*. Because not everyone knows HTML, this example is provided, as an exception, and with the exact solution code.



HTML TABLE - Windows Internet Explorer
C:\Users\frankoch\
Live Search
web durchsu
HTML TABLE
Computer | Protected Mode: Off
100%

Name	Status
AdtAgent	Stopped
AeLookupSvc	Running
ALG	Stopped
Appinfo	Running
AppMgmt	Stopped
AudioEndpointBuilder	Running
Audiosrv	Running
BFE	Running
BITS	Running
Browser	Stopped
BthServ	Running
CcmExec	Running
CertPropSvc	Running
clr_optimization_v2.0.50727_32	Stopped
COMSysApp	Stopped

FIGURE 6: COLOR HTML OUTPUT USING HTML MANIPULATION

Working with files

In the next few exercises we will be working with files. If this book was handed to you as part of a workshop, please ask your instructor for a memory stick containing test files. If you are working through the exercises on your own, simply create your own folder for the exercises. To do this, copy some different files (maybe 40) into a folder. If you cannot find anything else, you can use files from your Internet cache. You should make sure that you use at least two different file types, but the more the better.

Working with files in Windows PowerShell is really straightforward. Popular commands such as *dir* or *ls* can be used directly. However for *cd* you have to take account of the obligatory space afterwards: instead of cd.. it must now be cd .. !

Windows PowerShell also turns all files into objects. The size of a file can be directly queried and does not have to be separated out from a string. Also, Windows PowerShell does not just recognize the 'classic' file system. Using the cmdlet *get-psdrive* you can display all current locations to which Windows PowerShell can offer you direct access. Drives are selected at the end with a colon (:).

List all Windows PowerShell drives. Switch (*cd*) to drive HKLM:. Type in *cd software*. Type in *dir*. Where are you right now? Switch to the ENV: drive. List the content using *ls*, as if it were a classic Unix folder. Finally, switch to the CERT: drive and list the contents using the *get-childitem* cmdlet.

You will see that in Windows PowerShell almost nowhere in your computer is safe from you. The commands *dir, ls* and *get-childitem* have the same functionality everywhere. This means you can use whichever you prefer. To stay on the subject of Windows PowerShell syntax, I will mainly be talking about *get-childitem*. Just a quick note about the registry. If you have ever played around with the registry you will have noticed that you can get along well with *dir* and *cd* but cannot actually see any registry values. This is because registry values are a property of the registry objects, just like *Size* or *Date last accessed* are attributes of a file. To reveal the registry values you need the command *get-ItemProperty* . This command lists all registry values. For more information see, as ever, the help for Windows PowerShell.

To make it easier for you to work with the test files we will create a new 'drive' in Windows PowerShell, which will point to the actual test folder. To do this we need the command *new-psdrive*.

Create a new drive by inputting the command as follows. Change the end of the path to point to your folder with the test files:

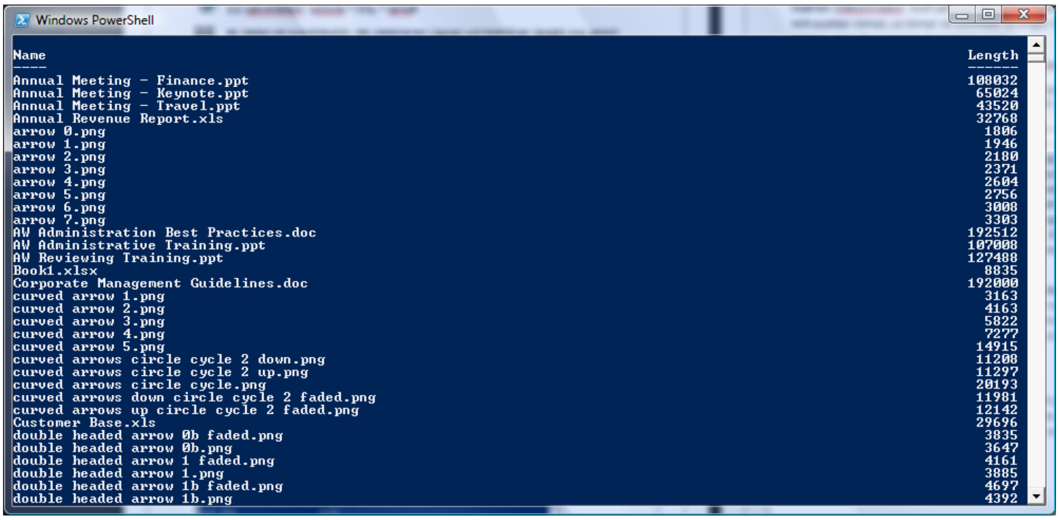
New-PSdrive –name FK –psprovider FileSystem –root c:\pathtofolder

Then use *cd FK:* to switch to that folder and check that you are in the right folder. If not you can use *Remove-PSDrive FK* to delete the drive and try again.

The following exercises should again show you the theoretical possibilities of Windows PowerShell. Of course, at this point we can only scratch the surface but the exercises will show you what you can do in principle. If necessary, modify the file extensions from the examples to match your own exercise files.

Switch to your test drive: cd fk: (In PowerShell syntax, by the way, this would be *set-location fk:*). List the content using *get-childitem*. Hide all temporary files: *get-childitem * –exclude *.tmp, *.temp*

B1: Output only the file name and length, but no temporary files with the extension temp or tmp. Hint: Use the same technique as outputting processes and services.



```
Name                                                  Length
----                                                  ------
Annual Meeting - Finance.ppt                          108032
Annual Meeting - Keynote.ppt                           65024
Annual Meeting - Travel.ppt                            43520
Annual Revenue Report.xls                              32768
arrow 0.png                                             1806
arrow 1.png                                             1946
arrow 2.png                                             2180
arrow 3.png                                             2371
arrow 4.png                                             2604
arrow 5.png                                             2756
arrow 6.png                                             3008
arrow 7.png                                             3303
AW Administration Best Practices.doc                  192512
AW Administrative Training.ppt                        107008
AW Reviewing Training.ppt                             127488
Book1.xlsx                                              8835
Corporate Management Guidelines.doc                   192000
curved arrow 1.png                                      3163
curved arrow 2.png                                      4163
curved arrow 3.png                                      5822
curved arrow 4.png                                      7277
curved arrow 5.png                                     14915
curved arrows circle cycle 2 down.png                  11208
curved arrows circle cycle 2 up.png                    11297
curved arrows circle cycle.png                         20193
curved arrows down circle cycle 2 faded.png            11981
curved arrows up circle cycle 2 faded.png              12142
Customer Base.xls                                      29696
double headed arrow 0b faded.png                        3835
double headed arrow 0b.png                              3647
double headed arrow 1 faded.png                         4161
double headed arrow 1.png                               3885
double headed arrow 1b faded.png                        4697
double headed arrow 1b.png                              4392
```

Figure 7: Output file name and length without TMP files

To minimize the amount of information you have to type in, Windows PowerShell offers you various methods of abbreviating commands. Type *get-alias | sort-object definition* and you will see a list of all possible spellings and 'dialects' for commands. For parameters however you can only use a certain number of letters before the parameter is filled automatically. Thus

*Get-childitem * –exclude *.tmp | select-object name, length*

becomes *ls* –ex *.tmp | select n*, le**

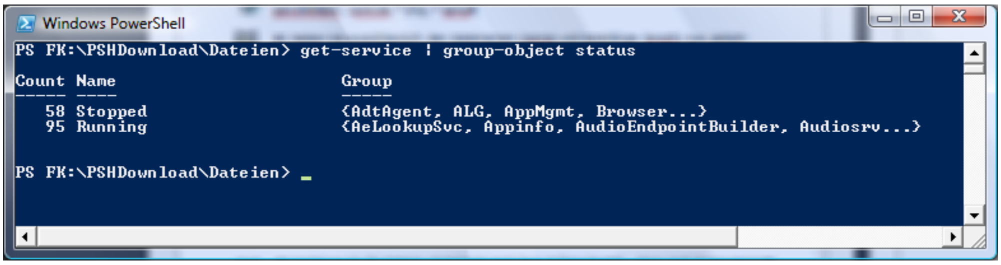
B2: Sort the files in ascending order by size (length), then by name. Hint: Use the same method as for processes in the examples above.

Finding object information using get-member

Use *get-member* to get an overview of all object attributes and functions. To use this command you pass an object to *get-member* via the pipe. You can even pass a list of similar objects without *get-member* running into problems.

B3: Create a list of all possible attributes for a file using the cmdlet *get-member*. Sort all the files by last accessed date. Hint: Use the results from the *get-member* function and 'guess' the appropriate attributes from the list of properties.

The command *group-object* can divide a list of objects into groups. To do this you must use one of the object attributes as an argument. *get-service | group-object status* thus generates a new list containing two (or more) entries. It's handy that it also shows the number of services and their corresponding status:



```
PS FK:\PSHDownload\Dateien> get-service | group-object status

Count Name                      Group
----- ----                      -----
   58 Stopped                   {AdtAgent, ALG, AppMgmt, Browser...}
   95 Running                   {AeLookupSvc, Appinfo, AudioEndpointBuilder, Audiosrv...}

PS FK:\PSHDownload\Dateien> _
```

FIGURE 8: RESULTS OF THE GROUP-OBJECT CMDLET

B4: Group the resulting files by their file extension. Then sort these results using the number of files with each extension. Hint: Output files, group them, then sort the new list by number. To do this use the argument *count.*

As well as *get-member* there is another useful cmdlet for obtaining information about objects: *measure-object*. Even if we cannot fully go into the range of options available using *measure-object*, we can at least guess at its possibilities using a few examples. Try to guess the results of the following command chain:

get-childitem | measure-object length –average –sum –maximum -minimum

You probably managed to guess after a few readthroughs. The whole thing would also work using wildcards and the normal Windows PowerShell cmdlet pipes.

B5: Determine the total size of all TMP files. In a second step output ONLY the total size. Hint: After your first attempt take the cdmlets and place them in (). After running the chain, repeat the commands with the addition of *get-member* to display all attributes for the results (Do you remember? Windows PowerShell works with objects and turns them into texts so we can read them on screen!). Find the property that matches your results in (), which might just correspond to the "Total" attribute. Remember the ForEach loop and how you found the "Status" property? Exactly: "*object.status". But here we want the "total" and not the" status". So change it accordingly.

Deleting files

Windows PowerShell also has all the necessary commands to enable you to delete files. Using the cmdlet *remove-item* you can delete more than just files. It works in the same way as *get-childitem*. At this point it would be a good idea to make a backup copy of your exercise folder. If you accidentally delete too much then you can at least start again.

B6: Delete all TMP files using *remove-item* with the correct arguments!

Sometimes you want to delete files that may be larger than the threshold value. Here you can use the cmdlet *where-object*. As with the *if* command you can define a condition that objects in a list have to fulfill before they can be selected. Let's look at an example using services. Using

```
get-service | where-object {$_.status –eq "stopped"}
```

will only show us stopped services.

B7: Now delete all files that are larger than 2 MB. In simple terms, 2 MB represents 2000000 bytes. Hint: Create your final script step by step. First create a list of all files and filter these by size (…*length –gt 2000000*). This will give you a new list that you can work through using a loop. Then output only the file names *($_.fullname*). You will need these names for the final *remove-item* run. Every once in a while you can also work with variables if you do not want your input lines to become unnecessarily long!

By the way, you do not have to enter 2 MB as 2000000 (after all, it is merely an approximation). It would be better to input 2MB directly as the size; and Windows PowerShell accepts this with no problems. You can even ask it to calculate the sum of 512KB + 512KB. For calculation you only need to input the numbers directly into the shell and do not have to work with any special cmdlets.

Creating folders

Now let's try to bring a little order to the chaos of our files. We will create a separate subfolder for each file type, then move the relevant files into the folders. To do this we need the cmdlet for creating a new "item" in the file system[1]: *new-item*. It takes the name as an argument and the type as a parameter, e.g. *directory* for a directory. You can create a new directory "Test" as follows:

```
new-item .\test   –type directory
```

To make your life easier we will now look at the sorting command again: *get-service | sort-object status* you already know, so now try

```
get-service | sort-object status –unique
```

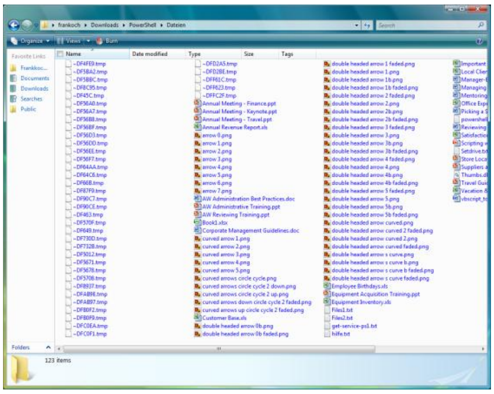
This only returns one representative for each status. Try it now. Now you have everything you need to create folders and directories.

B8: Create a separate subfolder for each file extension in your exercise folder. Hint: Create a list of all files and select them only by the "Extension" attribute. Now sort these using the –*unique* parameter. You will see a list of all file extensions but only once each. When you are ready you can assign this list to a variable and then go to the next step: using a loop, work through the object list and create a subfolder with the name of each extension (.*extension*). Remember that this has to be a complete path name, including at least one \ symbol! If you have any problems with the path then try (".*New"+$_.Extension*) as the argument. Don't forget to input the type (*directory*) at the end in order to create a directory!

[1] File systems such as FAT or NTFS are not true object-oriented systems. This is why we use the command *new-item* instead of *new-object*. This may change in future.

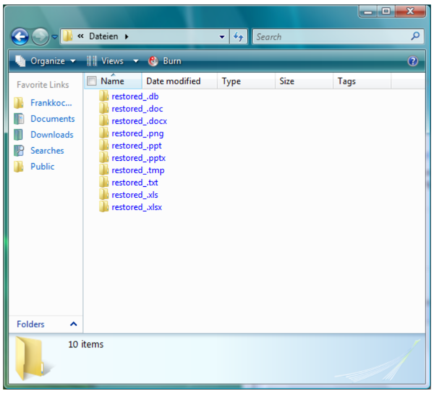
To move the files to their final position we use the command *move-item*. As an argument the cmdlet uses the whole name of the initial object and the path pointing to the end position, e.g. *move-item .\test.txt .\newfolder*

B9: Move all files from the test folder to the subfolders you created. Hint: The list *get-childitem* for the original folder now also includes the newly created subfolders, which you now have to filter out. Create another list of all items (look carefully through the list first). Filter the list using the comparison operator (…type –*notmatch "d")*. Then you need a loop for the list, which now contains only files. The final step is simple: For each object, find the correct destination folder using the file extension and move the file to that folder. Variables for saving interim results are always useful.



~DF4FE9.tmp
~DF5BA2.tmp
~DF5BBC.tmp
~DF8C95.tmp
~DF45C.tmp
~DF56A0.tmp
~DF56A7.tmp
~DF56B8.tmp
~DF56BF.tmp
~DF56D3.tmp
~DF56DD.tmp
~DF56EE.tmp
~DF56F7.tmp
~DF64AA.tmp
~DF64C6.tmp
~DF66B.tmp
~DF87F9.tmp
~DF90C7.tmp
~DF90CE.tmp
~DF463.tmp
~DF570F.tmp
~DF649.tmp
~DF730D.tmp
~DF732B.tmp
~DF5012.tmp
~DF5671.tmp
~DF5678.tmp
~DF5708.tmp
~DF8937.tmp
~DFAB9E.tmp
~DFAB97.tmp
~DFB0F2.tmp
~DFB0F9.tmp
~DFC0EA.tmp
~DFC0F1.tmp
~DFD2A5.tmp
~DFD2BE.tmp
~DFF61C.tmp
~DFF623.tmp
~DFFC2F.tmp
Annual Meeting - Finance.ppt
Annual Meeting - Keynote.ppt
Annual Meeting - Travel.ppt
Annual Revenue Report.xls
arrow 0.png
arrow 1.png
arrow 2.png
arrow 3.png
arrow 4.png
arrow 5.png
arrow 6.png
arrow 7.png
AW Administration Best Practices.doc
AW Administrative Training.ppt
AW Reviewing Training.ppt
Book1.xlsx
Corporate Management Guidelines.doc
curved arrow 1.png
curved arrow 2.png
curved arrow 3.png
curved arrow 4.png
curved arrow 5.png
curved arrows circle cycle.png
curved arrows circle cycle 2 down.png
curved arrows circle cycle 2 up.png
curved arrows down circle cycle 2 faded.png
curved arrows up circle cycle 2 faded.png
Customer Base.xls
double headed arrow 0b.png
double headed arrow 0b faded.png
double headed arrow 1 faded.png
double headed arrow 1.png
double headed arrow 1b.png
double headed arrow 1b faded.png
double headed arrow 2 faded.png
double headed arrow 2.png
double headed arrow 2b.png
double headed arrow 2b faded.png
double headed arrow 3 faded.png
double headed arrow 3.png
double headed arrow 3b.png
double headed arrow 3b faded.png
double headed arrow 4 faded.png
double headed arrow 4.png
double headed arrow 4b.png
double headed arrow 4b faded.png
double headed arrow 5 faded.png
double headed arrow 5.png
double headed arrow 5b.png
double headed arrow 5b faded.png
double headed arrow curved.png
double headed arrow curved 2 faded.png
double headed arrow curved 2.png
double headed arrow curved faded.png
double headed arrow s curve.png
double headed arrow s curve b.png
double headed arrow s curve b faded.png
double headed arrow s curve faded.png
Employee Birthdays.xls
Equipment Acquisition Training.ppt
Equipment Inventory.xls
Files1.txt
Files2.txt
get-service-ps1.txt
hilfe.txt
Important
Local Clie
Manager-
Managing
Mentoring
Office Exp
Picking a S
powershell
Reviewing
Satisfactio
Scripting w
Setdrive.tx
Store Loca
Suppliers a
Thumbs.d
Travel Gui
Vacation &
vbscript_t

123 items



restored_.db
restored_.doc
restored_.docx
restored_.png
restored_.ppt
restored_.pptx
restored_.tmp
restored_.txt
restored_.xls
restored_.xlsx

10 items

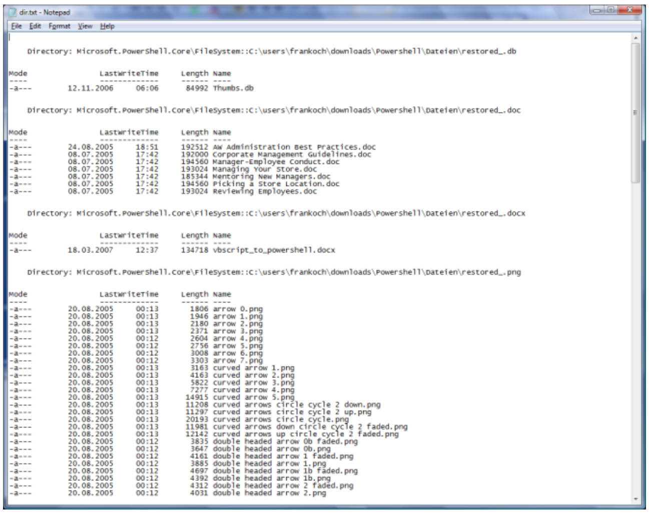
FIGURE 9: FILE FOLDER BEFORE SORTING FIGURE 10: FILE FOLDER AFTER SORTING

Finally we again output all of the files. The original path should now be empty although the subfolders are full. *get-childitem –recurse* will show this in details. Let’s save these results in a TXT file so we can study it in Notepad.

B10: Output the content of the exercise folder including all subfolders into a text file and save this as FinalOutput.txt.

B11: If you have the original exercise files, you can do one more thing: To make life even easier we can reset the read-only attribute for each Word file. To do this go to the subfolder for .doc files and call up all objects. The object attribute to set is called *IsReadOnly* and must be set to 0 (numeric zero). Hint: Use two commands: create a list of all objects, then use a loop to work through the objects, just as you have done in several other exercises before.

Windows PowerShell also assists you for ACL, your security specifications. With *get-acl* and *set-acl* you can easily transfer these from one object to another or even generate new. However, this is beyond the scope of this workshop. For more information about ACL see the online help instead.



```
    Directory: Microsoft.PowerShell.Core\FileSystem::C:\users\frankoch\downloads\Powershell\Dateien\restored_.db


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---        12.11.2006     06:06      84992 Thumbs.db


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\users\frankoch\downloads\Powershell\Dateien\restored_.doc


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---        24.08.2005     18:51     192512 AW Administration Best Practices.doc
-a---        08.07.2005     17:42     192000 Corporate Management Guidelines.doc
-a---        08.07.2005     17:42     194560 Manager-Employee Conduct.doc
-a---        08.07.2005     17:42     193024 Managing Your Store.doc
-a---        08.07.2005     17:42     185344 Mentoring New Managers.doc
-a---        08.07.2005     17:42     194560 Picking a Store Location.doc
-a---        08.07.2005     17:42     193024 Reviewing Employees.doc


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\users\frankoch\downloads\Powershell\Dateien\restored_.docx


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---        18.03.2007     12:37     134718 vbscript_to_powershell.docx


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\users\frankoch\downloads\Powershell\Dateien\restored_.png


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---        20.08.2005     00:13       1806 arrow 0.png
-a---        20.08.2005     00:13       1946 arrow 1.png
-a---        20.08.2005     00:13       2180 arrow 2.png
-a---        20.08.2005     00:13       2371 arrow 3.png
-a---        20.08.2005     00:12       2604 arrow 4.png
-a---        20.08.2005     00:12       2756 arrow 5.png
-a---        20.08.2005     00:12       3008 arrow 6.png
-a---        20.08.2005     00:12       3303 arrow 7.png
-a---        20.08.2005     00:13       3163 curved arrow 1.png
-a---        20.08.2005     00:13       4163 curved arrow 2.png
-a---        20.08.2005     00:13       5822 curved arrow 3.png
-a---        20.08.2005     00:13       7277 curved arrow 4.png
-a---        20.08.2005     00:13      14915 curved arrow 5.png
-a---        20.08.2005     00:13      11208 curved arrows circle cycle 2 down.png
-a---        20.08.2005     00:13      11297 curved arrows circle cycle 2 up.png
-a---        20.08.2005     00:13      20193 curved arrows circle cycle.png
-a---        20.08.2005     00:13      11981 curved arrows down circle cycle 2 faded.png
-a---        20.08.2005     00:13      12142 curved arrows up circle cycle 2 faded.png
-a---        20.08.2005     00:12       3835 double headed arrow 0b faded.png
-a---        20.08.2005     00:12       3647 double headed arrow 0b.png
-a---        20.08.2005     00:12       4161 double headed arrow 1 faded.png
-a---        20.08.2005     00:12       3885 double headed arrow 1.png
-a---        20.08.2005     00:12       4697 double headed arrow 1b faded.png
-a---        20.08.2005     00:12       4392 double headed arrow 1b.png
-a---        20.08.2005     00:12       4312 double headed arrow 2 faded.png
-a---        20.08.2005     00:12       4031 double headed arrow 2.png
```

FIGURE 11: OUTPUT OF CONTENTS WITH SUBFOLDERS

If you have time …

Let’s go back to the beginning and the cmdlets *export-csv* and *export-CliXML*. If still available, use your variable $P, if not enter the following line into Windows PowerShell:

```
$p = get-process
```

Now write the variable to a CSV file and a CliXML file:

```
$p | export-CSV .\test.csv

$p | export-CliXML .\test.xml
```

Now import these values into two new variables:

```
$p1 = import-csv .\test.csv

$p2 = import-CliXML .\test.xml
```

C1: Calculate the average CPU usage, the maximum value and the minimum value. Hint: Use the command *measure-object* as in the examples above. Do this separately for each of the three variables $p, $p1 and $p2.

C2: Now sort the variables, which are shown as a list, by CPU usage and select the first 5. Do this again for each of the three variables $p, $p1 and $p2. Are all the results really the same? Which variable is different from the others in certain situations? What is wrong here?

The solution is relatively simple. During the export to the CSV file and subsequent import, Windows PowerShell loses the information context for all the values. This means that when you sort the list the numbers become strings and 8.0 suddenly is placed ahead of 800. In XML files this information is saved and is therefore available to be sorted correctly. It is easy to calculate the average value etc.; here 8.0 + 800 equals 808.0 meaning that the evaluation for *measure-object* is correct, but not correct for sorting. Make sure to take this into account when saving your variables; XML might be the safer format!

WORKING WITH OTHER OBJECTS

Windows PowerShell as generic object processing machine

With Windows PowerShell you do not have to work only with your own objects; you have access to the entire world of objects including WMI, .NET and even COM. These areas are workshops in themselves, requiring separate training. You will find a wide range of secondary literature for each of the aforementioned topics. At this time we cannot even begin to scratch the surface and will therefore limit ourselves to a small example of each, hopefully piquing your interest for more.

WMI objects

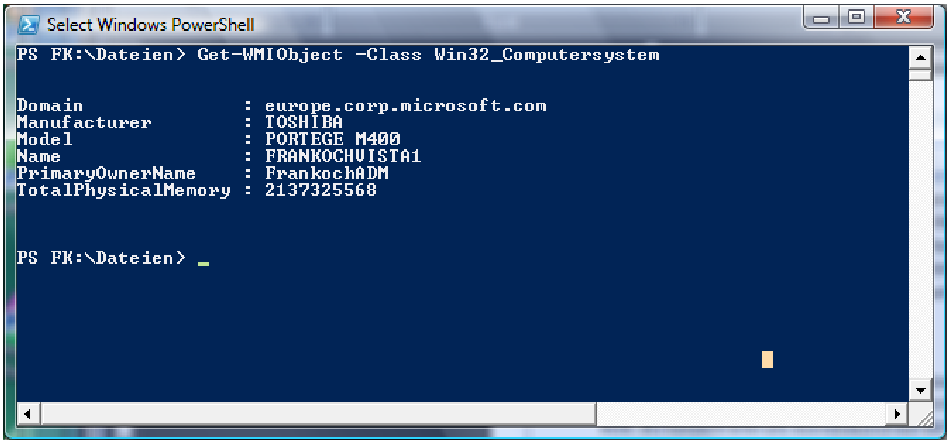
You probably know about WMI objects from Windows Scripting Host WSH and VBScript. If not, welcome to the topic but please do not expect any in-depth discussions about WMI. We will concentrate solely on the Windows PowerShell context.

You generate a WMI object in Windows PowerShell with the specific cmdlet *get-wmiobject*. This alone shows how important WMI is. Go to Windows PowerShell and enter the following command:

get-wmiobject –class win32_computersystem

You will see some basic information about your system. In contrast to VBScript or other languages, Windows PowerShell saves you from complex syntax, reducing your input to the absolute minimum. You only need

- The cmdlet *get-wmiobject* to define that you want to work with WMI
- The relevant WMI class you want to work with e.g. *-class win32_computersystem*



```
PS FK:\Dateien> Get-WMIObject -Class Win32_Computersystem


Domain              : europe.corp.microsoft.com
Manufacturer        : TOSHIBA
Model               : PORTEGE M400
Name                : FRANKOCHVISTA1
PrimaryOwnerName    : FrankochADM
TotalPhysicalMemory : 2137325568


PS FK:\Dateien> _
```

FIGURE 12: OUTPUT OF WMI OBJECTS WIN32_COMPUTERSYSTEM

The output is of course just a small proportion of all the data for the object. Use Windows PowerShell command *get-member* to display the list of attributes.

D1: Display the “User name” attribute for your system. Hint: Use the initial WMI example and determine the appropriate attribute for the user name.

WMI is, as we said, a world on its own. Here you can not only read information but can also write it. This even applies beyond the network to external systems, as long as they can authenticate themselves. To enter the world of WMI you can use various graphical object browsers such as CIM Studio. Literature on this subject will give you all the information you might need. However, I would like to show you a few small examples. The following WMI object classes often provide useful assistance in everyday IT:

Listing information for your desktop PC

get-wmiobject -class win32_desktop -computername .

The period is a part of the command and shows that you mean the local computer. In other cases use another computer name (server1, server4.mycompany.ch, etc.

Information on your system BIOS:

Get-wmiobject –class win32_bios

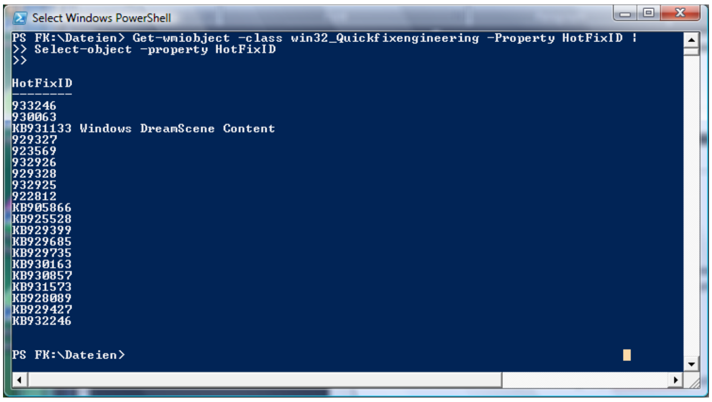
If you mean your own computer then “*–computername .*” is optional.

Listing all installed hotfixes

get-wmiobject –class win32_quickfixengineering written differently:

get-wmiobject –class win32_quickfixengineering –property hotfixid |

select-object –property hotfixid



```
PS FK:\Dateien> Get-wmiobject -class win32_Quickfixengineering -Property HotFixID ¦
>> Select-object -property HotFixID
>>

HotFixID
--------
933246
930063
KB931133 Windows DreamScene Content
929327
923569
932926
929328
932925
922812
KB905866
KB925528
KB929399
KB929685
KB929735
KB930163
KB930857
KB931573
KB928089
KB929427
KB932246


PS FK:\Dateien>
```

Figure 13: Output Hotfixes only as KB numbers

Writing data using WMI objects is the same as for objects in Windows PowerShell. Call up the object with *get-member* to see attributes and methods. You will also see whether these are simply readable values (*get*) or whether they can be written to (*set*). The *“Visible”* attribute in the later COM examples is one of these values, which is both readable and writeable.

Working with .NET objects and XML

The possibilities when using .NET and XML are as impressive as WMI. In the appendix you will find two examples. One calls up a website online, connects to the RSS feed and reads all the topics in the feed as well as their exact URLs, almost like your own personal RSS reader. The first example is impressively short and is given here for demonstration purposes. We must forego detailed discussions in order not to go beyond the scope of this book. In the fantastic Windows PowerShell book "Windows PowerShell in Action" by Bruce Payette, a founding father of Windows PowerShell, you will find extensive explanations for both scripts.

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

Yes, you're seeing that right; the whole thing is just two lines of script.

The syntax is again similar to WMI. For now let's ignore the brackets [XML]. There is an easy-to-read cmdlet *new-object*. You can use this to generate a new object. Since we enter the type directly (*net.webclient*), Windows PowerShell also knows immediately what it's dealing with: a .NET object of type *Webclient*.

From this object we use the method *DownloadString(url)*, which reads the given URL (the website).

The rest is just 'fiddling': Because we know what is hidden behind the URL, we do not check the whole thing; instead we assume that it is an RSS feed and therefore call up the object with the attributes we know apply to RSS feeds. If you were to enter a different URL, then the script would fail unless it was by chance another RSS feed.

format-table is also a normal cmdlet and simply issues a list of objects as a table. You can directly enter the columns you require as a title (*title*) and URL (*link*) for the RSS feed.

You can find the second example in the appendix listed as example 4. This script also uses the option of including .NET objects in Windows PowerShell scripts. This example uses a Windows application. Copy the example once into your Windows PowerShell and start the application. The result is hopefully impressive enough that you will be hungry for more. For information about .NET programming and .NET objects see either Microsoft's MSDN webpages or one of the many published works about .NET.

Working with COM objects

For our last example I would like to briefly discuss the possibilities of COM objects. I am using an Excel example. If your PC does not have Excel you can view the alternative Internet Explorer version. However, I am not going to give an extensive introduction to COM. There are so many books available that I really have nothing new to add.

First we have to tell Windows PowerShell that we want to work with a COM object. To make sure we don't have to input as much later on we immediately assign a variable to the object. This means we have our own COM object of the desired type. There is not a specific cmdlet for using COM objects as there is for WMI objects; instead we use the generic cmdlet for new objects: *new-object*.

```
$a = new-object -comobject excel.application
```

As an argument we input which COM object we want to use. Here it is Excel, selected with the argument *Excel.Application*. As you will discover, there are COM objects available in your system where the exact names are not so obvious. I would suggest a look at Microsoft's MSDN pages or one of the many books about COM. As well as using Excel as a data collection point for generating reports, in everyday IT administration you could also use Visio to graphically display system values, for automation with Windows PowerShell. At MS Press you will find a few books that describe Visio's COM objects really well. But let us get back to our Excel example.

To add data Excel now needs a workbook. Since we have entered the world of COM we also have to use the COM syntax. Windows PowerShell helps us to keep everything as simple as possible. We can explain the syntax using *$a | get-member*. For COM objects too the *get-member* command lists all attributes and methods. Simply expect more from those in an application such as Excel than with a smaller object that we have previously looked at. The *Workbooks.Add()* method creates a new workbook for us. However, you will also find methods for loading an existing workbook. To do this you must specify the path. Bu we will just create a new workbook from scretch:

```
$b = $a.Workbooks.Add()
```

It is possible that you will see an error message here in the form of "*Error: 0x80028018 (-2147647512) Description: Old Format or Invalid Type Library" when creating the Excel workbook*. Microsoft Knowledge Base Article 320369 describes the problem. Usually this error occurs when Excel is installed in a different language (e.g. *English (US)*) than the Windows regional settings (e.g. *English (UK)*). This is a bug in the Excel object. At the time of writing this book there is no fix available for this bug from Microsoft. As a workaround in this case for test purposes in the system control, set your *Regional Settings* to *English (US)* and restart Windows PowerShell. The example for Excel should then work with no errors.

Worksheets automatically belong to a workbook. We select the first one:

```
$c = $b.Worksheets.Item(1)
```

If we want to write to the worksheet then we do so not in the worksheet but in a line in the Excel sheet. This means we must input the line exactly:

```
$c.Cells.Item(1,1) = "Windows PowerShell rocks!"
```

And that's it. Our entry in Excel is ready. You don't believe me? OK, how about we display Excel and you can have a look? The command

```
$a.Visible = $True
```

sets the "Visible" attribute for our Excel object to $true. And as if by magic there is Excel showing our input.

As well as the "Visible" attribute we can also call up methods (functions) from Excel and use these for our own purposes. You can surely guess what the following line produces:

```
$b.SaveAs(".\Test.xls")
```

And if you were wondering: No, we are not saving Excel here, but instead we are saving the relevant workbook as Excel. We are interested in the XLS file, not the program.

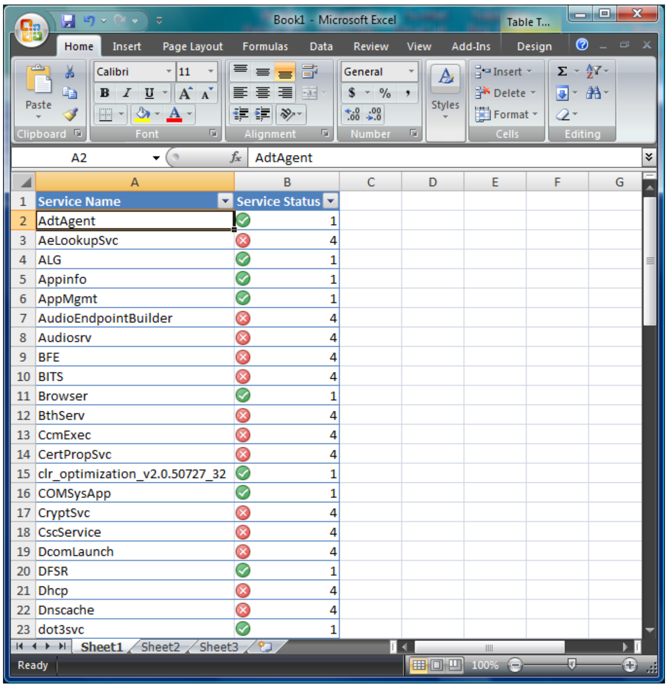
Of course, this demo is fake. I haven't showed you in detail how you get to information in order to manipulate Excel. How do you know that Excel work folders are called workbooks? And how do we get to the command "SaveAs" and its syntax? All of this is already available for COM objects; you can use any COM information sources to learn these things and then use them in Windows PowerShell. *get-member* is of great help everywhere. Because here we are only talking about the tool Windows PowerShell, I have limited myself to using the information and not showing its source. At the end we should clean up and close Excel, if you haven't already done so manually:

```
$a.Quit()
```

If you want to, try to solve this small exercise:

D2: Create a list of all services and enter the name of the services and their status into an Excel sheet. Hint: Use the aforementioned example to create an Excel object and assign it to a variable. To address the line in the Excel sheet we use a separate variable *$i*. Take the script for color output of the services and replace the color output with a *$c.Cells.Item($i,1)* entry. Do not forget to increase *$i* after every line, e.g. with *$i = $i + 1*. You can input several commands in one line using a semicolon ';'. Save the results from Excel in a XLS file, but please automatically using your script, not manually from the file menu in Excel.

In the results you will see that the status of the services is displayed in Excel as a number. Windows PowerShell is really helpful and replaces these numbers with a text such as "running" or "stopped" for us to read.



A2 | *fx* AdtAgent

	A	B
1	Service Name	Service Status
2	AdtAgent	✓ 1
3	AeLookupSvc	✗ 4
4	ALG	✓ 1
5	Appinfo	✓ 1
6	AppMgmt	✓ 1
7	AudioEndpointBuilder	✗ 4
8	Audiosrv	✗ 4
9	BFE	✗ 4
10	BITS	✗ 4
11	Browser	✓ 1
12	BthServ	✗ 4
13	CcmExec	✗ 4
14	CertPropSvc	✗ 4
15	clr_optimization_v2.0.50727_32	✓ 1
16	COMSysApp	✓ 1
17	CryptSvc	✗ 4
18	CscService	✗ 4
19	DcomLaunch	✗ 4
20	DFSR	✓ 1
21	Dhcp	✗ 4
22	Dnscache	✗ 4
23	dot3svc	✓ 1

Figure 14: Output services and status in Excel 2007. Also uses the specific format of Excel 2007 in order to display the status values as an icon instead of a number.

If your PC does not have Excel, you can use a different exercise using Internet Explorer. Instead of entering data into a line, we surf to a website. Combined with the previously mentioned RSS reader we only need one extra step: If you read RSS feeds automatically, search for keywords in the title and call up the webpages automatically. Surfing from the couch has never been easier.

The script begins like the Excel script by calling *new-object:*

$ie = new-object –comobject InternetExplorer.application

Here too the new object is initially invisible, and as in Excel there is a very easy solution:

$ie.Visible = $True

To find the options offered by the Internet Explorer object we also use the cmdlet get-member:

$ie | get-member

To keep things simple we surf to a popular website:

$ie.Navigate(http://www.microsoft.com/powershell)

If you like, you can combine this script with the RSS reader. You will then see a huge number of interesting links and titles. Take the list of titles, search for keywords and where you have a hit, surf to the page. The keyword search in the list of titles has not yet been mentioned in this workshop; use Windows PowerShell help to find more information. In this book however I do still owe you the solution to this problem.

Working with event logs

To complete our practical exercises in this book we will briefly discuss event logs. Windows PowerShell allows access to an event log using quite complex methods. WMI, .NET and COM objects bring together a range of commands. But even in Windows PowerShell there are already a few cmdlets that can help you every day. The most important of these is *get-eventlog.*

The command *get-eventlog –list* shows you all event logs in the system. To be able to access a specific event log we can use the cmdlet *where-object*. Access the system log as follows:

```
get-eventlog -list | where-object {$_.logdisplayname -eq "System"}
```

And yet there is a simpler way, this line of code would otherwise be far too long to use everyday. Therefore, if you want to evaluate entries from the event log, simply call up *get-eventlog*, adding the name of the required event log. The results can sometimes be very long and can be canceled at any time using CTRL-C. If you only want to see the 20 most recent entries, then you can use the parameter *–newest 20*. Of course you can replace 20 with any number.

```
get-eventlog system -newest 3

get-eventlog system -newest 3 | format-list
```

The events are then of course available for processing as usual in Windows PowerShell and can be sorted and grouped as normal.

E1: Search for the name of the Windows PowerShell event log. Group the event entries by event ID, then sort them by name. In a second step list only the events with the ID 403. Hint: If the event log name contains a space then set the entire name in quotation marks “My Event Log”.

E2: Sort the 15 most recent entries in the system event log in descending order by their event ID. Hint: Read the book through one more time if you cannot write down the answer immediately.

Solutions to the Exercises

Solution scripts to the exercises in this book

A1

```
 get-process | sort—object CPU
```

A2a

```
get-process | sort—object CPU –descending | select-object –first 10

get-process | sort—object CPU | select-object –last 10
```

A3

```
$P = get-process | sort—object CPU –descending | select-object –first 10
```

A4

```
$P > .\A4.txt

$P | export-csv .\a4.csv

$P | export-CliXML .\a4.xml
```

A5

```
get-service | sort-object status
```

A6

```
get-service | foreach-object{ write-host $_.name $_.status}
```

A7

```
get-service | foreach-object{ write-host –f yellow –b red $_.name $_.status}
```

Instead of –f you can also write –foregroundcolor and instead of –b also –backgroundcolor

A8

```
get-service | foreach-object{

if ($_.status -eq "stopped") {write-host -f green $_.name $_.status}`

else{ write-host -f red $_.name $_.status}}
```

A9

```
get-service | convertto-html
```

A10

```
get-service | convertto-html > .\a10.html
```

A11

```
get-service | sort-object status | convertto-html name, status > .\a10.html
```

B1

```
get-childitem * -exclude *.tmp | select-object name, length
```

B2

```
get-childitem * -exclude *.tmp | select-object name, length | sort-object length, name
```

B3

```
get-childitem | get-member
```

B4

```
get-childitem | group-object extension | sort-object count
```

B5

```
(get-childitem .\*.tmp | measure-object length –sum).sum
```

B6

```
remove-item .\*.tmp
```

B7

```
get-childitem | where-object {$_.length –gt 2000000} | foreach-object {remove-item $_.fullname}
```

B8

```
get-childitem | select-object extension | sort-object extension -unique `

| foreach-object {new-item (".\New" + $_.extension) -type directory}
```

B9

```
get-childitem | where-object {$_.mode -notmatch "d"} | foreach-object {$b= ".\New" + `

 $_.extension; move-item $_.fullname $b}
```

B10

```
get-childitem –recurse > .\finaloutput.txt
```

B11

```
get-childitem *.doc | foreach-object {$_.Isreadonly = 0}
```

C1

```
$p | measure-object CPU -min -max -average
```

C2

```
$p | sort-object CPU -Descending | Select-Object -first 5
```

D1

```
(get-wmiobject -class win32_computersystem).username
```

D2

```
$a = new-object -comobject excel.application

$a.Visible = $True

$b = $a.Workbooks.Add()

$c = $b.Worksheets.Item(1)

$c.Cells.Item(1,1) = "Service Name"

$c.Cells.Item(1,2) = "Service Status"

$i = 2

get-service | foreach-object{ $c.cells.item($i,1) = $_.name; $c.cells.item($i,2) = $_.status; $i=$i+1}

$b.SaveAs("C:\Users\frankoch\Downloads\Test.xls")

$a.Quit()
```

E1

```
get-eventlog "Windows PowerShell" | group-object eventid | sort-object name

get-eventlog "Windows PowerShell" | where-object {$_.eventid -eq 403}
```

E2

```
get-eventlog system -newest 15 | sort-object eventid -descending
```

Appendix
Script Examples

Windows PowerShell examples – from simple to complex

You can simply copy the following scripts and run them directly in Windows PowerShell. They show the theoretical possibilities of PowerShell but also that a short introduction is in no way able to cover all of Windows PowerShell. To copy the scripts, highlight the text lines under the first > symbol through the comments (not including the comments). These exercises are mainly taken from the highly recommendable book "Windows PowerShell in Action" by Bruce Payette, a founding father of Windows PowerShell. In this book the examples are also explained in more detail in case you would like more information.

Example 1: Direct output of a string

The shortest ever "Hello world" program: >

```
"Hello world"
```

Comments:
Windows PowerShell can directly recognize strings and outputs them directly.

Example 2: Log file analysis

Create a list of all log files in "Windir" folder; search the log files for the word "Error", Output the log file name and the line with the error: >

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

Comments:
There might possibly be an error message due to access authorizations etc. Ignore any such messages.

Example 3: Your own RSS reader

Call up a webpage, read the RSS feed, display the RSS posts and their URLs: >

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

Comments:
At the very beginning the connection to the website might be quite slow.

Example 4: Adding windows into scripts

Create a separate WinForm for graphical output of information etc.: >

```
[void][reflection.assembly]::LoadWithPartialName("System.Windows.Forms")

$form = new-object Windows.Forms.Form

$form.Text = "My First Form"

$button = new-object Windows.Forms.Button

$button.text="Push Me!"

$button.Dock="fill"

$button.add_click({$form.close()})

$form.controls.add($button)

$form.Add_Shown({$form.Activate()})

$form.ShowDialog()
```

Comments:
To quit, simply click on the newly created window (might be hidden by Windows PowerShell).

APPENDIX
THE THEORY BEHIND WINDOWS POWERSHELL

Theoretical principles for Windows PowerShell

Windows PowerShell – A Brief Introduction

Microsoft's previous attempts at command line shells were rather unsatisfactory. The old command.com may have been adequate for the first few versions of MS DOS, but the growing number of operating system functions soon burst the boundaries. The cmd.exe shell introduced with Windows NT offered many more possibilities. And yet compared to popular Unix shells such as Bash, Microsoft's command line was certainly found lacking.

Now Microsoft has changed that completely. With Windows PowerShell (previously Monad Shell, MSH) the software group administrators want to give us a shell under Windows that we can use to write as many scripts as our hearts desire in order to control our systems. Windows PowerShell follows a completely new concept than that used by text-oriented shells such as Bash.

Aims of developing Windows PowerShell

Windows PowerShell is a new Windows command line shell that was specifically developed for system administrators. The shell consists of an interactive input line and script environment that can be used either alone or in combination. In contrast to most shells that accept and return text, Windows PowerShell is based on an object model made available by .NET Framework 2.0. This fundamental change in the environment allows for completely new tools and methods for managing and configuring Windows.

Windows PowerShell introduces the idea of cmdlets (pronounced "commandlet"). A cmdlet is a simple command line tool that is integrated into the shell and executes a single function. Although you can use cmdlets in isolation, their power is more obvious when used in combination to perform complex tasks. Windows PowerShell contains several hundred basic cmdlets and you can also write your own cmdlets and pass these on to other users.

Like many other shells, Windows PowerShell gives you access to the computer's file system. Thanks to Windows PowerShell providers, you have easy access to other data stores such as the registry and to storage areas for digital signature certificates.

On texts, parsers and objects

PowerShell is completely object-oriented. Compared to the usual shells, it behaves differently to the results of a command, instead of a text, with an object (more on this to follow). And yet in a similar way to previous popular shells it does have a *pipeline*, where the results of individual commands can be processed and passed on. The only difference is that the input values and results are objects instead of texts.

PowerShell objects are no different from those in a C++ or C# program. You can imagine an object as a data unit with attributes and methods. Methods are actions that can be executed on the object.

If, for example, you call up a *service* in Windows PowerShell, you are actually using an object that represents that service. If you display information about an object, you are displaying the attributes of the relevant service object. And if you start a service, i.e. change the **status** attribute of the service to **started**, you are using a method for the service object. With increasing experience you will better understand the advantages of object processing and will consciously work with objects.

PowerShell's object-oriented concept makes the standard parsers for Unix shells (analyze/evaluate) and text-based information with all its problems and error proneness completely superfluous. To make this clearer we provide the following example:
Assume that you would like to have a list of all processes that consume more than 100 handles. With a traditional Linux shell you would call up the command for displaying processes (ps -A). The command then returns a text list. Each line would contain information about a process, separated by spaces. You would parse these lines with a tool, filter out the process ID and then query this with another program to find the handle number. You would then parse these text-based results, filter out the relevant lines and then finally display the relevant text.

Depending on how well cutting and filtering of information from the text lists functions, this approach is more or less reliable. But, for example, if the title of a column in the output changes and the process names are then too long, you will certainly have problems.

PowerShell uses a fundamentally different approach. You also start with the command *get-process*, which returns all running processes in the operating system. Only here they are returned as an *object list* made of process objects. These objects can then be investigated for their attributes and directly queried – therefore you do not have to examine any text lines and split them into columns. We will take a closer look at this in more detail.

A new scripting language

No existing language is used in Windows PowerShell, instead it uses a separate language for the following reasons:

- Windows PowerShell needed a language for managing .NET objects.
- The language had to support complex tasks without making simple tasks complicated.
- The language had to meet the conventions for other languages used in .NET programming such as C#.

Each language now has its own commands. In Windows PowerShell we made sure that these commands all followed a specific logic during construction and naming. A *cmdlet* is a specialized command that works through objects in Windows PowerShell. You can recognize cmdlets from their names: a verb and a noun, always in the singular, separated by a hyphen (-), e.g. *get-help, get-process* and *start-service*. In Windows PowerShell most cmdlets are very simple and were designed to be used together with other cmdlets. Therefore, for example 'Get' cmdlets only call up data, 'Set' cmdlets generate or change data, 'Format' cmdlets format data and 'Out' cmdlets only forward output to a specific destination.

Windows commands and service programs

Over time you will have got used to certain commands. A new language that does not take account of this would quickly fail. This is why in Windows PowerShell you can also execute standard Windows command programs and start Windows programs that have a graphical interface such as Notepad and Calculator. Furthermore, as in **Cmd.exe** you can import text output from other programs and use these texts in the shell. Even if commands such as *dir, ls* or *cd* do not follow the official syntax of Windows PowerShell, they will work and can be used with no problems.

An interactive environment

As with other shells, Windows PowerShell supports a completely interactive environment. If you enter a command at the input prompt then the command is processed and the output is displayed in the shell window. You can send the output from a command to a file or printer, or you can use a pipeline operator (|) to send it to another command.

Script support

If you always repeat the same commands, we recommend not inputting the commands or command sequences individually, but instead saving them in a file and then executing this file. A file containing commands is called a script.

As well as the interactive interface, Windows PowerShell also offers complete script support. You can run a script by entering the name of the script at the prompt. The file extension for Windows PowerShell scripts is **.ps1**; entering a file name extension is optional.

Although scripts are very useful, they can also be used to spread malicious code. This is why you can define security guidelines in Windows PowerShell (also called the execution guidelines) to specify which scripts may be run and whether they must be digitally signed. To avoid unnecessary risks it is not permitted in any of the execution guidelines in Windows PowerShell to execute a script by double-clicking on its symbol, as you can with, for example, the good old .bat, .cmd or .vbs files.

CMD, WScript or PowerShell? Do I have to decide?

As of Windows XP you have three scripting shells available: the good old CMD shell, Windows Scripting Host for your VB or J scripts and now Windows PowerShell. And no fear, you don't have to decide between the shells or worry that one might become obsolete. Even in the new Windows versions such as Vista or Longhorn Server you will find that all 3 shells are still equal. You can use whichever you prefer, depending on your taste. Or you can use the shell that fits best for a specific task. If you have not written many scripts until now, then this could be the time to start with Windows PowerShell, to depend on the newest and easiest-to-use technology. Using the practical sections you have, or will have, seen how easy it is and how powerful simple scripts can be, without digging out huge tomes and books from the 60s and 70s talking about batch programming.

Windows PowerShell 1.0

Even if Windows PowerShell is only a 1.0 Version, the quality of the product is utterly convincing and I can recommend its use in practice with a clear conscience. This is mainly because Windows PowerShell is actually only a new way of using the popular .NET Framework 2.0. And so you will hardly notice the product's low version number.

Only the function range would tell you that not everything you might desire is already contained in Windows PowerShell. Windows PowerShell can already directly access file systems, event logs, registry entries, .NET, WMI and ADSI interfaces and objects; however, true, separate remoting is not yet available. This means that accessing other computers works as well as WMI or .NET commands allow. In future we could perhaps imagine using new Web service interfaces for administration that were certified in 2006. For now we can content ourselves with using today's tested methods such as WMI and .NET.

APPENDIX
SCRIPTS & SECURITY

Security when using scripts

With Windows Scripting Host (WSH) included in Windows 2000, Microsoft introduced a new, powerful scripting engine. This engine was so powerful that it was soon being used as a new battleground for virus authors. Inexperienced users received the first few emails with promisingly attractive images, and when they opened attachments there was nothing to see but it turned out to be a VBScript that set about exploiting their systems. Windows PowerShell does its best to counteract this type of threat.

Therefore the basic settings of Windows PowerShell are set to execute absolutely no scripts at all. The function has to be explicitly activated by a system administrator. Activation allows various levels all related to script signing. In addition, the extension for Windows PowerShell (PS1) is linked to Notepad. Even if your environment allows scripts, an ill-advised double-click on an attachment or file would simply open Notepad and show you the source code. And last, Windows PowerShell always demands explicit input of the path ".\"" for files that should be called directly from the current folder. This can prevent executing a program in Windows PowerShell that you didn't expect to be using.

To be able to run scripts you have to modify the security settings for Windows PowerShell. There are two cmdlets for this purpose: *get-executionpolicy* and *set-executionpolicy*. With *get: executionpolicy* you query the current settings. There are four levels:

Policy Value	Description
Restricted (Default)	No scripts are run
Allsigned	Only signed scripts are run
RemoteSigned	Locally created scripts are allowed but other scripts must be signed
Unrestricted	All scripts are run

To change these settings a system administrator must call up, for example, the command

set-executionpolicy remotesigned

Microsoft provides a Group Policy template to set this key automatically in larger organizations. For more information on this subject, also see the documentation for Windows PowerShell.

FINAL WORDS

I would like to thank my wife Petra for her love and patience. She had to forego many weekend trips and evenings with me, whilst I spent time with my computer instead of her.

This book could not have been written without the Windows PowerShell developers, and so my heartfelt thanks to them. Special thanks go of course to Bruce Payette who gave me the initial impetus for my first draft with his book “Windows PowerShell in Action”. Highly recommended, give it a read!

Not all parts of this book are my own work, in particular the theoretical sections came from content published on the Microsoft MSDN pages or Windows PowerShell help files. Here you will find more in-depth information which I encourage you to read.

And if you have any comments or feedback for this book, send me your thoughts by email. I may not be able to respond to everyone personally, but will be very grateful for any and all constructive criticism and praise: frankoch@microsoft.com

PowerShell Cheat Sheet

Essential Commands

To get help on any cmdlet use get-help

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

To get all available cmdlets use get-command

```
Get-Command
```

To get all properties and methods for an object use get-member

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

Setting Security Policy

View and change execution policy with Get-Execution and Set-Execution policy

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

To Execute Script

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

Functions

Parameters separate by space. Return is optional.

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

Variables

Must start with $

```
$a = 32
```

Can be typed

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

Arrays

To initialise

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

To query

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

Constants

Created without $

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

Referenced with $

```
$b
```

Creating Objects

To create an instance of a com object

New-Object -comobject <ProgID>

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

To create an instance of a .Net Framework object. Parameters can be passed if required

New-Object –type <.Net Object>

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

Writing to Console

Variable Name

```
$a
```

or

```
Write-Host $a –foregroundcolor “green”
```

Capture User Input

Use Read-Host to get user input

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

Passing Command Line Arguments

Passed to script with spaces

```
myscript.ps1 server1 benp
```

Accessed in script by $args array

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

Miscellaneous

Line Break `

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

Comments #

```
# code here not executed
```

Merging lines ;

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

Pipe the output to another command |

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

Do While Loop

Can repeat a set of commands while a condition is met

```
$a=1
Do {$a; $a++}
While ($a –lt 10)
```

Do Until Loop

Can repeat a set of commands until a condition is met

```
$a=1
Do {$a; $a++}
Until ($a –gt 10)
```

For Loop

Repeat the same steps a specific number of times

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

ForEach - Loop Through Collection of Objects

Loop through a collection of objects

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

If Statement

Run a specific set of code given specific conditions

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

Switch Statement

Another method to run a specific set of code given specific conditions

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

Reading From a File

Use Get-Content to create an array of lines. Then loop through array

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

Writing to a Simple File

Use Out-File or > for a simple text file

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

Or use > to output script results to file

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

Writing to an Html File

Use ConvertTo-Html and >

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

Writing to 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
```