

## Kapitel 11

# Wie PowerShell Objekte erweitert

### In diesem Kapitel:

|  |     |
|--|-----|
| PowerShell-Objekte verstehen               | 486 |
| AliasProperty: Eigenschaften »umbenennen«  | 490 |
| NoteProperty: Taggen von Objekten          | 491 |
| ScriptProperty: »Berechnete« Eigenschaften | 492 |
| ScriptMethod und ParameterizedProperty     | 499 |
| Membertypen für den internen Gebrauch      | 504 |
| Objekte permanent erweitern                | 515 |
| Testen Sie Ihr Wissen                      | 523 |
| Zusammenfassung                            | 524 |

Fast alles, was Ihnen in PowerShell zu Gesicht kommt, sind Objekte, wie Sie seit dem letzten Kapitel wissen. PowerShell verwendet dabei nicht etwa eigene Objekte, sondern nutzt die vorhandenen Objekttypen, die das Betriebssystem (über .NET Framework) zur Verfügung stellt. Allerdings gibt sich PowerShell damit nicht zufrieden, sondern »verbessert« und erweitert diese Objekte. Zuständig dafür ist das *Extended Type System* (ETS), und was das genau ist, was es unternimmt und wie Ihnen das zu Gute kommt, ist Thema dieses Kapitels.

## PowerShell-Objekte verstehen

Irgendwie ähneln sich die Objekte, die Low-Level-Systemfunktionen und Cmdlets liefern:

```
# aktuellen PowerShell-Prozess liefern:
```

```
PS> [System.Diagnostics.Process]::GetCurrentProcess()
```

| Handles | NPM(K) | PM(K) | WS(K) | VM(M) | CPU(s) | Id    | ProcessName |
|---------|--------|-------|-------|-------|--------|-------|-------------|
| 345     | 26     | 51456 | 30864 | 584   | 1,48   | 12164 | powershell  |

```
PS> Get-Process -id $pid
```

| Handles | NPM(K) | PM(K) | WS(K) | VM(M) | CPU(s) | Id    | ProcessName |
|---------|--------|-------|-------|-------|--------|-------|-------------|
| 383     | 26     | 51456 | 30860 | 584   | 1,50   | 12164 | powershell  |

Tatsächlich sind die Low-Level-Zugriffe auf Objekte und das Betriebssystem nichts anderes als das, was Cmdlets in ihrem Inneren auch unternehmen. Deswegen sind die Ergebnisse von beiden gleich. Man kann also sagen: Cmdlets sind lediglich freundliche Verpackungen, die es besonders einfach und sicher machen, Systemfunktionen aufzurufen. Sie können dasselbe aber immer auch ohne Cmdlets durch direkten Low-Level-Code erreichen.

### HINWEIS

Fundament beider Aufrufe ist *.NET Framework*, das eigentlich dazu dient, Anwendungsentwicklung einfacher und schneller zu machen. Es funktioniert wie ein riesiger Zoo mit unzähligen Tierarten und Tieren, auf die Anwendungsentwickler zugreifen können. Standardaufgaben, die fast jede Anwendung benötigt, sind darin also schon vorbereitet, sodass ein Anwender nicht das Rad ständig neu erfinden muss. PowerShell greift genau wie die Programmiersprachen Visual Basic und C# einfach nur darauf zurück.

Eine gute Idee ist das indes nicht immer: Gibt es für eine Aufgabe bereits ein Cmdlet, dann nutzen Sie natürlich besser dieses. Ein Cmdlet ist verständlicher, verfügt über eine Hilfe und vernünftiges Fehlerhandling. Direkter Zugriff auf Objekte und Systemfunktionen, so wie er in diesem Kapitel gezeigt werden wird, entspricht dem Hinzuprogrammieren von Funktionalitäten, für die es (noch) keine Cmdlets gibt. Falls die Aufgabe mit einem Cmdlet schon gelöst werden kann, sollten Sie die Finger von (unnötigen) direkten Zugriffen lassen. Sobald Sie auch Teil D dieses Buchs gelesen haben, werden Sie nützliche Systemfunktionen sogar »bergen« können, also mithilfe einer erweiterten Funktion (Advanced Function) als Cmdlet verpacken.

## Erweiterte PowerShell-Objekte

PowerShell setzt also lediglich auf .NET Framework auf, was clever ist: Es wäre zu viel Arbeit (und sinnlose noch dazu), für alle Cmdlets sämtliche Typen und Objekte neu zu erfinden. Stattdessen greift PowerShell auf die Typen und Objekte von .NET Framework zu. Allerdings gibt sich PowerShell damit nicht zufrieden. Es erweitert diese Typen und Objekte bei Bedarf und hat dazu einen eingebauten Mechanismus, den Sie ebenfalls nutzen dürfen. Die Erweiterungen, die PowerShell vornimmt, lassen sich mit *Get-Member* sichtbar machen. Die folgende Zeile etwa liefert eine Übersicht sämtlicher Membertypen, die ein Cmdlet wie *Get-Process* zurückliefert:

```
PS> Get-Process | Get-Member | Group-Object MemberType -NoElement | Sort-Object Count
```

```
Count Name
-----
1 NoteProperty
2 PropertySet
4 Event
6 AliasProperty
7 ScriptProperty
19 Method
51 Property
```

Darin finden sich 51 Properties und 19 Methoden. Das sind diejenigen, die in den Originalobjekten vorkommen, die direkt von .NET Framework stammen. Sie wurden ergänzt durch 1 *NoteProperty*, 7 *ScriptProperties* und 6 *AliasProperties*. Außerdem sind noch 2 *PropertySets* und 4 *Events* vorhanden. Wollen Sie nur das sehen, was PowerShell »hinzuerfunden« hat, setzen Sie den Parameter *-View* ein:

```
PS> Get-Process | Get-Member -View Extended
```

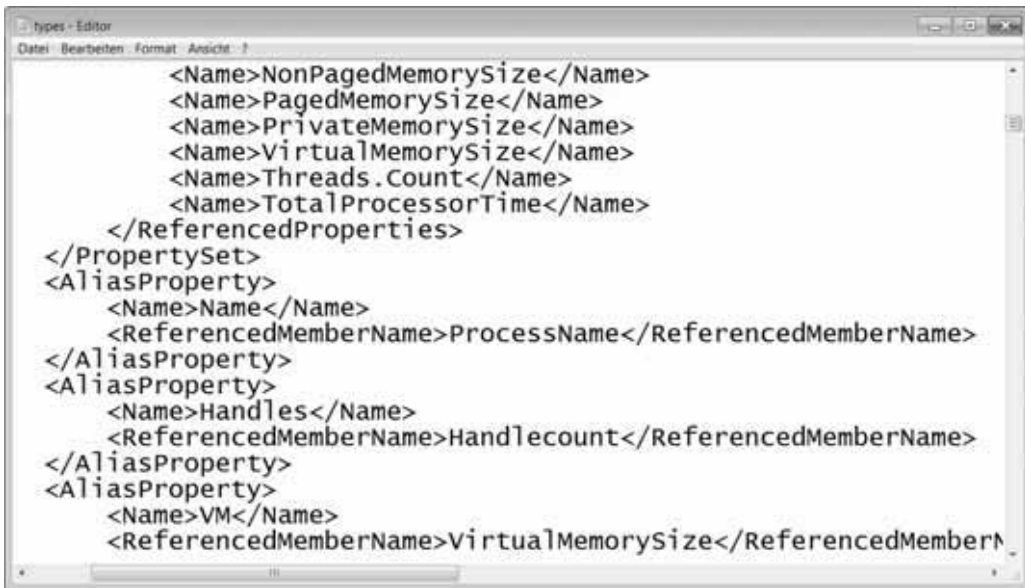
```
TypeName: System.Diagnostics.Process

Name           MemberType      Definition
----           -
Handles        AliasProperty   Handles = Handlecount
Name           AliasProperty   Name = ProcessName
NPM            AliasProperty   NPM = NonpagedSystemMemorySize
PM            AliasProperty   PM = PagedMemorySize
VM            AliasProperty   VM = VirtualMemorySize
WS            AliasProperty   WS = WorkingSet
__NounName     NoteProperty    System.String __NounName=Process
PSConfiguration PropertySet     PSConfiguration {Name, Id, PriorityClass, FileVersion}
PSResources    PropertySet     PSResources {Name, Id, Handlecount, WorkingSet, NonPagedMemorySiz...
Company        ScriptProperty  System.Object Company
{get=$this.Mainmodule.FileVersionInfo.Compa...
CPU            ScriptProperty  System.Object CPU {get=$this.TotalProcessorTime.TotalSeconds;}
Description    ScriptProperty  System.Object Description
{get=$this.Mainmodule.FileVersionInfo.F...
FileVersion    ScriptProperty  System.Object FileVersion
{get=$this.Mainmodule.FileVersionInfo.F...
Path           ScriptProperty  System.Object Path {get=$this.Mainmodule.FileName;}
Product        ScriptProperty  System.Object Product
{get=$this.Mainmodule.FileVersionInfo.Produ...
ProductVersion ScriptProperty  System.Object ProductVersion
{get=$this.Mainmodule.FileVersionInf...
```

Das wirft eine Reihe von Fragen auf: Was unterscheidet eine *NoteProperty* zum Beispiel von einer normalen *Property*? Und wo kommen die Erweiterungen eigentlich her? Alle Eigenschaften und Methoden, vor denen ein weiterer Begriff steht (also *NoteProperty*, *AliasProperty* etc.), werden von PowerShell zu verschiedenen Zwecken hinzugefügt, die gleich beleuchtet werden. Diese Erweiterung erledigt das ETS. Immer also, wenn PowerShell ein Objekt empfängt, fügt das ETS die für den jeweiligen Objekttyp nützlichen Erweiterungen hinzu.

Welche Erweiterungen das sind, regeln XML-Dateien, in die Sie gern schon einmal mit einem Texteditor hineinschauen können, die aber erst etwas später genauer untersucht werden. Ändern sollten Sie in diesen Dateien indes lieber einstweilen nichts.

```
PS> $Host.Runspace.InitialSessionState.Types | Select-Object -ExpandProperty FileName
C:\Windows\System32\WindowsPowerShell\v1.0\types.ps1xml
C:\Windows\System32\WindowsPowerShell\v1.0\typesv3.ps1xml
PS> notepad C:\Windows\System32\WindowsPowerShell\v1.0\types.ps1xml
```



**Abbildung 11.1** Typenerweiterungen sind textbasierte XML-Dateien und bestimmen, was PowerShell hinzufügt

Die verschiedenen Erweiterungen, die PowerShell beispielsweise bei *Process*-Objekten hinzugefügt hat, haben unterschiedliche Aufgaben: *AliasProperties* etwa liefern keine grundsätzlich neuen Erkenntnisse, aber machen eine vorhandene Eigenschaft unter einem anderen (vielleicht verständlicheren oder konsistenteren) Namen sichtbar. Das *Process*-Objekt aus .NET Framework beispielsweise liefert die Größe des virtuellen Speichers in der Eigenschaft *VirtualMemorySize* und den Namen des Prozesses in der Eigenschaft *ProcessName*. In PowerShell dürfen diese beiden Eigenschaften auch als *VM* und *Name* abgerufen werden. Die Originaleigenschaften sind aber auch noch da:

```
# auf den ersten verfügbaren explorer.exe-Prozess zugreifen:
PS> $ProzessObjekt = Get-Process -Name explorer | Select-Object -First 1
```

```
# Originaleigenschaft und PowerShell-AliasProperty:
PS> $ProzessObjekt.VirtualMemorySize
766840832
PS> $ProzessObjekt.VM
766840832

# Originaleigenschaft und PowerShell-AliasProperty:
PS> $ProzessObjekt.ProcessName
explorer
PS> $ProzessObjekt.Name
explorer
```

Tabelle 11.1 gibt eine Übersicht über die verschiedenen Membertypen, die nachträglich von PowerShell in Objekte eingefügt werden.

| Membertyp                    | Beschreibung   |
|------------------------------|--|
| <i>AliasProperty</i>         | Zweitname für eine vorhandene Eigenschaft  |
| <i>NoteProperty</i>          | Eigenschaft mit fest hinterlegtem Inhalt. Mit <i>NoteProperties</i> kann man Zusatzinformationen an ein Objekt anheften.   |
| <i>ParameterizedProperty</i> | Eigenschaft mit Parametern. Formal sieht der Aufruf einer solchen Eigenschaft aus wie die einer Methode.   |
| <i>ScriptMethod</i>          | Methode mit hinterlegtem Skriptcode, der ausgeführt wird, wenn die Methode aufgerufen wird   |
| <i>ScriptProperty</i>        | Eigenschaft mit hinterlegtem Skriptcode, der ausgeführt wird, wenn die Eigenschaft abgefragt wird. Der Skriptcode berechnet also zur Laufzeit den Wert dieser Eigenschaft. |
| <i>CodeMethod</i>            | Interner Gebrauch (siehe Abschnitt »Membertypen für den internen Gebrauch«; Seite 504)   |
| <i>CodeProperty</i>          |  |
| <i>MemberSet</i>             |  |
| <i>PropertySet</i>           |  |

**Tabelle 11.1** Nachträglich in Objekte eingefügte Eigenschaften und Methoden

**PROFITIPP** Die Wahrheit ist: PowerShell kann eigentlich *überhaupt keine* neuen eigenen Typen (=Klassen) definieren. Es kann nur mit Objekten arbeiten, die .NET Framework liefert. Genau deshalb hat PowerShell daraus eine Tugend gemacht und mit dem eben skizzierten Mechanismus dafür gesorgt, dass es doch eigene Objekte herstellen kann: Zugrunde liegt stets ein .NET-Objekt von der Stange, das über die Erweiterungen aus Tabelle 11.1 maßgeschneidert erweitert werden kann.

In Kapitel 17 erfahren Sie zum Beispiel, wie selbstgeschriebene Funktionen eigene Objekte generieren, um darin ihre Ergebnisse zu verpacken. Tatsächlich werden diese eigenen Objekte auch dort immer nur erschaffen, indem ein bestehendes Objekt aus .NET Framework erweitert wird. Weil PowerShell andererseits Low-Level-Zugang zu .NET Framework bietet und da .NET Framework natürlich sehr wohl neue Typen (=Klassen) definieren kann, ist das (indirekt) auch in PowerShell möglich. Mit dem Cmdlet *Add-Type* setzt man dazu klassischen .NET-Code ein. Beispiele hierfür finden Sie in Kapitel 15.

Schauen wir uns nun den Einsatzbereich (und Nutzen) der Erweiterungen aus Tabelle 11.1 an. Das wird Ihnen nicht nur wertvolle Einblicke geben, wie PowerShell intern funktioniert. Sie dürfen auch selbst auf den Erweiterungsmechanismus zugreifen und das eröffnet ungeahnte Möglichkeiten.

## AliasProperty: Eigenschaften umbenennen

Eine *AliasProperty* macht vorhandene Eigenschaften unter einem neuen Namen verfügbar. Da man keine vorhandenen Eigenschaften löschen oder umbenennen kann, ohne das Objekt dabei zu zerstören, sind *AliasProperties* ein Weg, vorhandene Eigenschaften »umzubenennen«. Wirklich umbenannt werden sie zwar nicht, sie sind aber ab sofort auch unter dem gewünschten neuen Namen ansprechbar. Man greift auf diese Möglichkeit zurück, wenn der Originalname zu lang, zu unverständlich oder inkonsistent mit anderen Objekttypen ist. Das erklärt, warum die Größe eines Arrays sowohl über *Length* als auch über *Count* abgefragt werden darf:

```
PS> (1..10).Count
10
```

```
PS> (1..10).Length
10
```

Ein Blick hinter die Kulissen zeigt, dass das .NET-Objekt vom Typ *Array* eigentlich nur die Eigenschaft *Length* kennt, die von PowerShell aber über eine *AliasProperty* auch unter dem Namen *Count* zugänglich gemacht wurde:

```
PS> Get-Member -InputObject (1..10) -Name Count, Length
```

```
    TypeName: System.Object[]
```

| Name   | MemberType    | Definition        |
|--------|---------------|-------------------|
| Count  | AliasProperty | Count = Length    |
| Length | Property      | int Length {get;} |

Dies ist ein Beispiel für eine Konsistenzverbesserung: Über *AliasProperties* sorgt PowerShell dafür, dass Objekte unterschiedlichen Typs ähnliche Informationen über dieselben Eigenschaften anbieten.

Dasselbe dürfen Sie auch tun. Hier eine Aufgabe, die sich mit *AliasProperties* lösen lässt:

»Es sollen über die WMI BIOS-Informationen abgerufen werden: *Hersteller*, *Version* und *Sprache*. Diese Informationen sollen genau unter diesen Namen ausgegeben werden.«

Die gewünschten BIOS-Informationen liefert *Get-WmiObject* in einer Zeile:

```
PS> Get-WmiObject -Class Win32_BIOS | Select-Object -Property Manufacturer, Version, CurrentLanguage
```

| Manufacturer             | Version          | CurrentLanguage |
|--------------------------|------------------|-----------------|
| American Megatrends Inc. | _ASUS_ - 1072009 | en US iso8859-1 |

Um die Spaltennamen umzubenennen, kann man `AliasProperties` einsetzen:

```
PS> $infos = Get-WmiObject -Class Win32_BIOS
PS> $infos | Add-Member -MemberType AliasProperty -Name Hersteller -Value Manufacturer
PS> $infos | Add-Member -MemberType AliasProperty -Name Sprache -Value CurrentLanguage
PS> $infos | Select-Object -Property Hersteller, Version, Sprache
```

Die gewünschten Eigenschaften stehen nun doppelt zur Verfügung: unter dem Originalnamen und unter dem neuen Aliasnamen. Mit `Select-Object` suchen Sie sich anschließend die Eigenschaften aus, die Sie anzeigen wollen. Das Ergebnis verwendet jetzt deutsche Spaltenüberschriften:

| Hersteller               | Version          | Sprache         |
|--------------------------|------------------|-----------------|
| American Megatrends Inc. | _ASUS_ - 1072009 | en US iso8859-1 |

## NoteProperty: Taggen von Objekten

Eine *NoteProperty* ist eine »Notiz«, also eine Eigenschaft mit statischem Inhalt. Mit `NoteProperties` kann man Zusatzinformationen an ein Objekt anfügen (*Taggen* genannt). PowerShell nutzt `NoteProperties` fast gar nicht. Umso wichtiger sind diese für Sie. Im folgenden Beispiel wird der Inhalt des Systemordners `System32` von zwei verschiedenen Servern gelesen. Damit klar ist, welche Datei von welchem Server stammt, fügt `Add-Member` zu jedem Objekt eine *NoteProperty* mit dem Namen des jeweiligen Servers hinzu. Der Parameter `-PassThru` sorgt dafür, dass `Add-Member` das ergänzte Objekt einfach weitergibt:

```
$server1 = '\\storage1'
$server2 = '\\powershellpc'

$fileList1 = Get-ChildItem $server1\c$\windows\system32\*.dll |
Sort-Object -Property Name |
# eine neue Eigenschaft namens "ComputerName" anfügen und den Herkunftsserver darin angeben:
Add-Member -MemberType NoteProperty -Name ComputerName -Value $server1 -PassThru

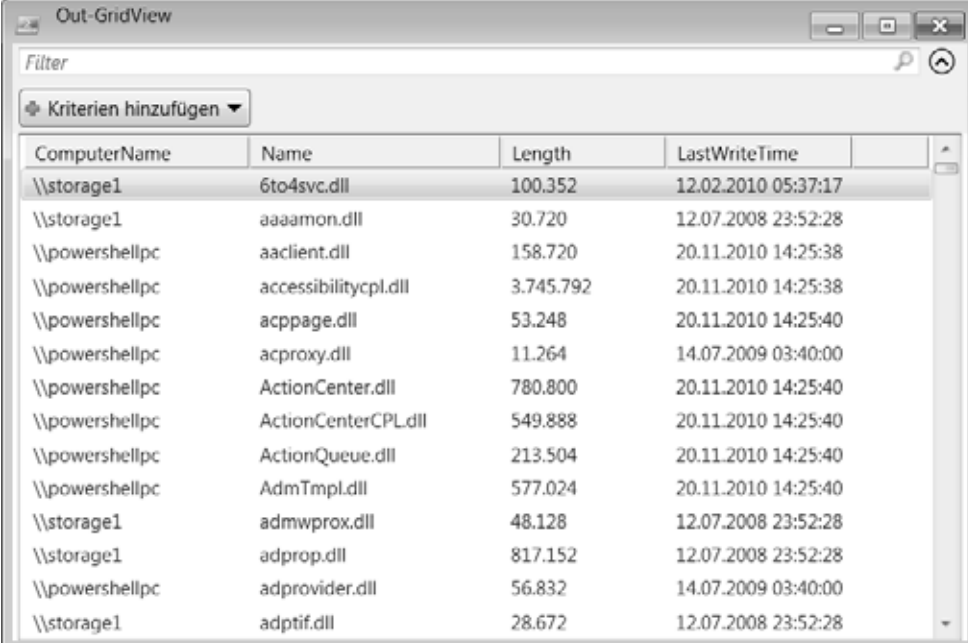
$fileList2 = Get-ChildItem $server2\c$\windows\system32\*.dll |
Sort-Object -Property Name |
# eine neue Eigenschaft namens "ComputerName" anfügen und den Herkunftsserver darin angeben:
Add-Member -MemberType NoteProperty -Name ComputerName -Value $server2 -PassThru

# Unterschiedliche Dateien finden (basierend auf "Name" und "Length") und Objekte mit -PassThru
# weitergeben:
Compare-Object -ReferenceObject $fileList1 -DifferenceObject $fileList2 -Property Name, Length
-PassThru |
Sort-Object -Property Name |
Select-Object -Property ComputerName, Name, Length, LastWriteTime |
Out-GridView
```

**Listing 11.1** Das Skript `compare_filelist.ps1`

Das Ergebnis wird im `GridView` angezeigt und meldet alle Dateien, also solche, die entweder nur auf dem einen oder nur auf dem anderen System mit unikalen (einmaligen) Eigenschaften gefunden wurden. Dateien mit gleichem Dateinamen, aber anderen variierenden Eigenschaften wie der Datei-

größe gelten folglich als unterschiedlich und werden entsprechend aufgelistet. Auf welchem System die Datei vorliegt, verrät die Spalte *ComputerName*, also die hinzugefügte *NoteProperty*:



The screenshot shows a window titled 'Out-GridView' with a 'Filter' search bar and a 'Kriterien hinzufügen' button. Below is a table with four columns: ComputerName, Name, Length, and LastWriteTime. The table lists various DLL files from two different servers: \\storage1 and \\powershellpc.

| ComputerName   | Name                 | Length    | LastWriteTime       |
|----------------|----------------------|-----------|---------------------|
| \\storage1     | 6to4svc.dll          | 100.352   | 12.02.2010 05:37:17 |
| \\storage1     | aaaamon.dll          | 30.720    | 12.07.2008 23:52:28 |
| \\powershellpc | aaclient.dll         | 158.720   | 20.11.2010 14:25:38 |
| \\powershellpc | accessibilitycpl.dll | 3.745.792 | 20.11.2010 14:25:38 |
| \\powershellpc | acppage.dll          | 53.248    | 20.11.2010 14:25:40 |
| \\powershellpc | acproxy.dll          | 11.264    | 14.07.2009 03:40:00 |
| \\powershellpc | ActionCenter.dll     | 780.800   | 20.11.2010 14:25:40 |
| \\powershellpc | ActionCenterCPL.dll  | 549.888   | 20.11.2010 14:25:40 |
| \\powershellpc | ActionQueue.dll      | 213.504   | 20.11.2010 14:25:40 |
| \\powershellpc | AdmTpl.dll           | 577.024   | 20.11.2010 14:25:40 |
| \\storage1     | admwprox.dll         | 48.128    | 12.07.2008 23:52:28 |
| \\storage1     | adprop.dll           | 817.152   | 12.07.2008 23:52:28 |
| \\powershellpc | adprovider.dll       | 56.832    | 14.07.2009 03:40:00 |
| \\storage1     | adptif.dll           | 28.672    | 12.07.2008 23:52:28 |

Abbildung 11.2 Dateiunterschiede auf zwei Servern feststellen

## ScriptProperty: »Berechnete« Eigenschaften

Eine *ScriptProperty* ist ganz besonders flexibel, denn sie kann beliebigen PowerShell-Code ausführen, wenn die Eigenschaft *abgerufen* wird. Sie kann auch PowerShell-Code ausführen, wenn die Eigenschaft *geändert* wird.

### Lesbare Eigenschaften

PowerShell nutzt ScriptProperties an vielen Orten. ScriptProperties werden eingesetzt, wenn eine Information in den vorhandenen Eigenschaften nicht vorliegt und zunächst durch eine Berechnung oder von einem anderen Ort beschafft werden soll. Dazu ein Beispiel:

```
# auf eine Datei zugreifen:
PS> $datei = Get-Item -Path $env:windir\explorer.exe

# Namensbestandteile über Eigenschaften abrufen:
PS> $datei.Name
explorer.exe
```



```
PS> $datei.Extension
.exe
```

```
PS> $datei.BaseName
explorer
```

Woher diese Eigenschaften stammen, verrät *Get-Member*:

```
PS> $datei | Get-Member -Name Name, Extension, BaseName
```

```
    TypeName: System.IO.FileInfo
```

| Name      | MemberType     | Definition   |
|-----------|----------------|--|
| Name      | Property       | System.String Name {get;}  |
| Extension | Property       | System.String Extension {get;}   |
| BaseName  | ScriptProperty | System.Object BaseName {get;if (\$this.Extension.Length -gt 0){\$this.Nam... |

Die Eigenschaften *Name* und *Extension* sind also reguläre Eigenschaften. *BaseName* dagegen wurde von PowerShell als *ScriptProperty* hinzugefügt. Ein Teil des Quellcodes ist bereits in der Spalte *Definition* zu sehen. Der vollständige Quellcode sieht so aus:

```
PS> ($datei | Get-Member -Name BaseName).Definition
System.Object BaseName {get;if ($this.Extension.Length -gt 0){$this.Name.Remove($this.Name.Length - $this.Extension.Length)}else{$this.Name};}
```

Wenn Sie also *BaseName* aufrufen, führt PowerShell diesen Code aus:

```
if ($this.Extension.Length -gt 0)
{
    $this.Name.Remove($this.Name.Length - $this.Extension.Length)
}
else
{
    $this.Name
}
```

Die Variable *\$this* repräsentiert das Objekt selbst. Wenn also dessen Eigenschaft *Extension* nicht leer ist, wird der Name des Objekts aus der Eigenschaft *Name* gelesen. Das Ergebnis ist ein Objekt vom Typ *String*. Solche Objekte verfügen über die Methode *Remove()*, mit der Zeichen abgeschnitten werden. Sie müssen nur sagen, wie viele:

```
PS> 'Dieser Text ist zu lang'.Remove(10)
Dieser Tex
```

Die Anzahl der abzutrennenden Zeichen ist die Gesamtlänge des Namens (Eigenschaft *Length*) minus der Gesamtlänge der Dateierweiterung.

---

**PROFITIPP** Die PowerShell-Entwickler hätten es sich auch sehr viel leichter machen können, denn das Abschneiden einer Dateierweiterung kommt häufig vor, und so gibt es dafür bereits eine Systemfunktion:

```
PS> [System.IO.Path]::GetFileNameWithoutExtension('c:\test.txt')
test
```

Die *ScriptProperty* hätte also auch diesen Code verwenden können:

```
[System.IO.Path]::GetFileNameWithoutExtension($this.Name)
```

Und Sie dürfen das auch, denn mit *Add-Member* kann man einzelnen Objekten von Hand Erweiterungen hinzufügen. Ob dieser Code nicht nur übersichtlicher ist, sondern auch schneller, ergibt ein kleiner Test. Dabei wird einer größeren Dateiliste mit *Add-Member* eine neue *ScriptProperty* namens *BaseNameNeu* hinzugefügt, die den neuen Code verwendet. Danach werden die Ausführungsgeschwindigkeiten beim Abruf der Eigenschaften verglichen:

```
# eine neue ScriptProperty "BaseNameNeu" anfügen, die verbesserten Code nutzt:
$liste = Get-ChildItem -Path $env:windir\system32 | Add-Member -MemberType ScriptProperty -Name
BaseNameNeu -Value { [System.IO.Path]::GetFileNameWithoutExtension($this.Name) } -PassThru
```

```
# Performance prüfen:
Measure-Command {
    $liste | Select-Object -Property BaseName
} | Select-Object -ExpandProperty TotalMilliseconds
```

```
Measure-Command {
    $liste | Select-Object -Property BaseNameNeu
} | Select-Object -ExpandProperty TotalMilliseconds
```

Das Ergebnis ist allerdings nicht unbedingt spektakulär:

```
239,4955
197,5284
```

Die neue Eigenschaft *BaseNameNeu* arbeitet folglich etwas schneller, hier sind es rund 20 %.

## Lesbare und schreibbare Eigenschaften

Manche Eigenschaften wie *BaseName* dürfen nur gelesen, aber nicht verändert werden. Andere akzeptieren auch neue Werte. Das gilt auch für *ScriptProperties*. Zertifikate verfügen zum Beispiel über die Eigenschaft *SendAsTrustedIssuer* (zumindest in PowerShell 3.0, denn diese Eigenschaft ist neu):

```
# erstbestes Root-Zertifikat verwenden:
PS> $cert = Get-ChildItem -Path Cert:\CurrentUser\root | Select-Object -first 1
```

```
PS> $cert | Get-Member -Name *send*
```

```
TypeName: System.Security.Cryptography.X509Certificates.X509Certificate2
```

| Name                | MemberType     | Definition   |
|---------------------|----------------|--|
| SendAsTrustedIssuer | ScriptProperty | System.Object SendAsTrustedIssuer {get=[Microsoft.PowerShell.... |

Diese Eigenschaft kann gelesen, aber auch verändert werden. Die Definition sieht so aus:

```
PS> ($cert | Get-Member -Name *send*).Definition
System.Object SendAsTrustedIssuer {
    get=[Microsoft.PowerShell.Commands.SendAsTrustedIssuerProperty]::ReadSendAsTrustedIssuerProperty
($this);

    set=$sendAsTrustedIssuer = $args[0]
    [Microsoft.PowerShell.Commands.SendAsTrustedIssuerProperty]::WriteSendAsTrustedIssuerProperty($this,
    $this.PsPath,$sendAsTrustedIssuer);
}
```

Die Definition legt also diesmal *zwei* Skripts fest, eines zum Lesen der Eigenschaft (*get*) und eines zum Ändern (*set*). Wird die Eigenschaft gelesen, liefert sie das Ergebnis von *ReadSendAsTrustedIssuerProperty()*. Wird ihr dagegen ein neuer Wert zugewiesen, setzt sie diesen mit *WriteSendAsTrustedIssuerProperty()*. *\$args(0)* steht hierbei für den neuen Wert, der der Eigenschaft zugewiesen wurde, und *\$this* steht für das konkrete Objekt, dessen Eigenschaft gerade verwendet wird.

Formal dürfen Sie *SendAsTrustedIssuer* also auch ändern, aber ob das wirklich funktioniert oder eine Fehlermeldung liefert, hängt von den Umständen ab: Verfügen Sie über die nötigen Rechte und ist das ausgewählte Zertifikat dafür überhaupt geeignet? *SendAsTrustedIssuer* ist sicher eine eher exotische Eigenschaft, aber leider die einzige beschreibbare *ScriptProperty*, die PowerShell hinzugefügt. Mit dem Wissen, das Sie an diesem Anschauungsobjekt gewonnen haben, können Sie nun allerdings auch Ihre eigenen lesbaren und schreibbaren *ScriptProperties* entwickeln.

*ScriptProperties* werden häufig dazu verwendet, ansonsten schwer zugängliche Informationen einfacher bereitzustellen. Interessieren Sie sich zum Beispiel für die Dateiversion einer Anwendung oder einer DLL-Bibliothek, finden Sie diese Informationen in der Eigenschaft *VersionInfo*. Die Version der Datei *explorer.exe* ermitteln Sie also so:

```
PS> $file = Get-Item $env:windir\explorer.exe
PS> $file.VersionInfo
```

*VersionInfo* ist wiederum eine *ScriptProperty*:

```
PS> ($file | Get-Member -Name VersionInfo ).Definition
System.Object VersionInfo
{get=[System.Diagnostics.FileVersionInfo]::GetVersionInfo($this.FullName);}
```

Tatsächlich können die Versionsinformationen einer beliebigen Datei also auch mit *GetVersionInfo()* unter Angabe eines Pfadnamens abgerufen werden:

```
PS> [System.Diagnostics.FileVersionInfo]::GetVersionInfo("$env:windir\regedit.exe")
```

| ProductVersion | FileVersion      | FileName               |
|----------------|------------------|------------------------|
| 6.1.7600.16385 | 6.1.7600.1638... | C:\Windows\regedit.exe |

Das, was *VersionInfo* liefert, ist wiederum ein Objekt mit verschiedenen Eigenschaften. Genau das kann problematisch sein: Vielleicht möchten Sie eine Dateiliste generieren, in der auch die Dateiversion eingeblendet ist. Mit einer *ScriptProperty* ist das problemlos möglich:

```
PS> Get-ChildItem -Path $env:windir -Filter *.exe | Add-Member -MemberType ScriptProperty -Name
Version -Value { $this.VersionInfo.ProductVersion } -PassThru | Add-Member -MemberType
ScriptProperty -Name Description -Value { $this.VersionInfo.FileDescription } -PassThru |
Select-Object -Property Mode, LastWriteTime, Length, Version, Name, Description | Out-GridView
```

| Mode | LastWriteTime       | Length    | Version        | Name                                  | Description  |
|------|---------------------|-----------|----------------|---------------------------------------|--|
| ---- | 07.07.2012 14:38:26 | 3.058.304 | 1.0.0.9        | AsScrPro.exe                          | AsScrPro   |
| ---- | 07.07.2012 14:38:27 | 80.512    | 1.0.0.1        | ASUS_Scr_ZenbookPrime Uninstaller.exe | AsScrUninst  |
| ---- | 20.11.2010 14:24:28 | 71.168    | 6.1.7600.16385 | bfsvc.exe                             | Startdatei-Wartungsprogramm                        |
| ---- | 24.02.2012 01:55:29 | 2.871.808 | 6.1.7600.16385 | explorer.exe                          | Windows-Explorer                                   |
| ---- | 14.07.2009 03:39:10 | 15.360    | 6.1.7600.16385 | Aeuupdate.exe                         | BitLocker-Laufwerkverschlüsselung-Wartungsprogramm |
| ---- | 14.07.2009 03:39:12 | 732.696   | 6.1.7600.16385 | HelpPane.exe                          | Microsoft-Hilfe und Support                        |
| ---- | 14.07.2009 03:39:12 | 16.896    | 6.1.7600.16385 | hh.exe                                | Ausführbare Microsoft®-HTML-Hilfsdatei             |
| ---- | 14.07.2009 03:39:25 | 193.536   | 6.1.7601.17514 | notepad.exe                           | Editor   |
| ---- | 14.07.2009 03:39:29 | 427.008   | 6.1.7600.16385 | regedit.exe                           | Registrierungs-Editor                              |
| ---- | 11.02.2012 07:36:01 | 67.072    | 6.1.7601.17777 | tptwow64.exe                          | Print driver host for 32bit applications           |
| ---- | 10.06.2009 23:41:17 | 49.680    | 1.7.0.0        | twunk_16.exe                          | Twain_32.dll Client's 16-Bit Thunking Server       |
| ---- | 14.07.2009 03:14:42 | 31.232    | 1.7.1.0        | twunk_32.exe                          | Twain.dll Client's 32-Bit Thunking Server          |
| ---- | 14.07.2009 03:14:45 | 9.728     | 6.1.7600.16385 | winhlp32.exe                          | Windows Winhlp32-Stub                              |
| ---- | 14.07.2009 03:39:57 | 10.240    | 6.1.7600.16385 | write.exe                             | Windows Write                                      |

Abbildung 11.3 Zwei neue Eigenschaften in der Dateiliste: Version und Description

Erinnern Sie sich noch an Listing 11.1? Dort wurden zwei Server miteinander verglichen und die Dateien gemeldet, die unterschiedlich waren. Wenn die Dateiversion in die Überprüfung einbezogen werden kann, wird das Ergebnis noch sehr viel nützlicher: Jetzt sehen Sie alle Dateien, die in unterschiedlichen Versionen auf den beiden Computern liegen:

```
$server1 = '\\storage1'
$server2 = '\\powershellpc'

$fileList1 = Get-ChildItem $server1\c$\windows\system32\*.dll |
Sort-Object -Property Name |
# eine neue Eigenschaft namens "ComputerName" anfügen und den Herkunftsserver darin angeben:
Add-Member -MemberType NoteProperty -Name ComputerName -Value $server1 -PassThru |
Add-Member -MemberType ScriptProperty -Name Version -Value { $this.VersionInfo.ProductVersion
} -PassThru

$fileList2 = Get-ChildItem $server2\c$\windows\system32\*.dll |
Sort-Object -Property Name |
# eine neue Eigenschaft namens "ComputerName" anfügen und den Herkunftsserver darin angeben:
Add-Member -MemberType NoteProperty -Name ComputerName -Value $server2 -PassThru |
Add-Member -MemberType ScriptProperty -Name Version -Value { $this.VersionInfo.ProductVersion
} -PassThru

# Unterschiedliche Dateien finden (basierend auf "Name" und "Length") und Objekte mit -PassThru
# weitergeben:
Compare-Object -ReferenceObject $fileList1 -DifferenceObject $fileList2 -Property Name, Version
-PassThru |
Sort-Object -Property Name |
Select-Object -Property ComputerName, Name, Version |
Out-GridView
```

Listing 11.2 Das Skript *compare\_fileversion.ps1*

| ComputerName   | Name                 | Version        |
|----------------|----------------------|----------------|
| \\storage1     | 6to4svc.dll          | 5.2.3790.4662  |
| \\storage1     | aaaamon.dll          | 5.2.3790.3959  |
| \\powershellpc | aaclient.dll         | 6.1.7600.16385 |
| \\powershellpc | accessibilitycpl.dll | 6.1.7600.16385 |
| \\powershellpc | ACCTRES.dll          | 6.1.7600.16385 |
| \\storage1     | acctres.dll          | 6.00.3790.0    |
| \\powershellpc | acedit.dll           | 6.1.7600.16385 |
| \\storage1     | acedit.dll           | 5.2.3790.3959  |
| \\storage1     | actui.dll            | 5.2.3790.3959  |
| \\powershellpc | actui.dll            | 6.1.7600.16385 |
| \\powershellpc | acppage.dll          | 6.1.7601.17514 |
| \\powershellpc | acproxy.dll          | 6.1.7600.16385 |
| \\powershellpc | ActionCenter.dll     | 6.1.7600.16385 |
| \\powershellpc | ActionCenterCPL.dll  | 6.1.7600.16385 |

**Abbildung 11.4** Unterschiedliche Dateiversionen auf zwei verschiedenen Computern finden

Auch lesbare und schreibbare *ScriptProperties* sind möglich: Die Eigenschaft *BaseName* konnte Dateinamen zwar auslesen, aber nicht ändern. Wie müsste eine *ScriptProperty* aussehen, die das kann? Schauen Sie sich dieses Beispiel dazu an und versuchen Sie, den Code nachzuvollziehen:

```
$code_get = { [System.IO.Path]::GetFileNameWithoutExtension($this.Name) }
$code_set = {
    try {
        $extension = $this.Extension
        $newname = $args[0] + $extension
        Write-Host ("Benenne '{0}' um in '{1}'." -f $this.FullName, $newname)
        Rename-Item -Path $this.FullName -NewName $newname -ErrorAction Stop
    }
    catch {
        Throw "Unable to change base name: $_"
    }
}

# Testdatei anlegen und neue Eigenschaft hinzufügen:
$file = New-Item -Path $HOME\Desktop\eine_testdatei.txt -ItemType File |
Add-Member -MemberType ScriptProperty -Name BaseNameNeu -Value $code_get -SecondValue
$code_set -PasThru
```

**Listing 11.3** Das Skript *readwrite\_scriptproperty.ps1*

In *\$file* liegt nun ein Dateiojekt, das sich auf den ersten Blick wie jedes andere verhält. Es repräsentiert eine neue leere Textdatei auf Ihrem Desktop:

```
PS> $file
```

```
Verzeichnis: C:\Users\Tobias\Desktop
```

| Mode  | LastWriteTime    | Length | Name               |
|-------|------------------|--------|--------------------|
| ----  | -----            | -----  | ----               |
| -a--- | 18.11.2012 18:30 | 0      | eine_testdatei.txt |

Beim Lesen unterscheiden sich *BaseName* und *BaseNameNeu* nicht (außer dass *BaseNameNeu* ein paar Millisekunden schneller ist):

```
PS> $file.BaseName
eine_testdatei
```

```
PS> $file.BaseNameNeu
eine_testdatei
```

Beim Schreiben (Ändern) allerdings gibt es nun fundamentale Unterschiede, denn während *BaseName* keine Änderungen zulässt, kann *BaseNameNeu* den Dateinamen unter Beibehaltung der aktuellen Dateierweiterung ändern:

```
# BaseName unterstützt keine Änderungen:
PS> $file.BaseName = 'anderer Name'
"BaseName" ist eine schreibgeschützte Eigenschaft.
```

```
# BaseNameNeu hingegen schon:
PS> $file.BaseNameNeu = 'anderer Name'
Benenne 'C:\Users\Tobias\Desktop\eine_testdatei.txt' um in 'anderer Name.txt'.
```

Ein Blick auf den Desktop beweist: Der Name der Datei wurde tatsächlich wie gewünscht geändert. Allerdings wird wenig später klar, warum die PowerShell-Entwickler nicht selbst auf diese Idee gekommen sind. Beide Eigenschaften halten nämlich trotz der Namensänderung am alten Dateinamen fest:

```
PS> $file.BaseName
eine_testdatei
```

```
PS> $file.BaseNameNeu
eine_testdatei
```

Woher sollten sie auch wissen, dass sich der Dateiname geändert hat? Damit hat *BaseNameNeu* eine schwere Inkonsistenz verursacht: Sämtliche Eigenschaften, die Bezug zum Dateinamen haben, stimmen nicht mehr mit der Realität überein. Sie alle müssten aktualisiert werden, was aber nicht möglich ist, da sie alle schreibgeschützt sind. Das ist der Grund, warum *BaseName* nur lesbar ist: Zwar hätte man wie gezeigt auch die Datei umbenennen können, aber es wäre viel zu aufwändig gewesen, alle davon betroffenen weiteren Eigenschaften zu aktualisieren.

## ScriptMethod und ParameterizedProperty

Eine *ScriptMethod* ist im Grunde dasselbe wie eine *ScriptProperty*, nur verhält sie sich wie eine Methode (ein Befehl) und kann vom Aufrufer deshalb Argumente empfangen. Dasselbe gilt für eine *ParameterizedProperty*: Dabei handelt es sich um eine Eigenschaft, die wie eine Methode Argumente empfangen kann. Sie sehen schon: Die Übergänge zwischen den verschiedenen Membertypen sind fließend. Rufen Sie über die WMI Datums- oder Zeitinformationen ab, zum Beispiel das Installationsdatum Ihres Betriebssystems oder den letzten Systemstart, dann liefert WMI diese zum Beispiel in einem sonderbaren Format:

```
PS> $os = Get-WmiObject -Class Win32_OperatingSystem
PS> $InstallDate = $os.InstallDate
PS> $LastBootDate = $os.LastBootUpTime
```

```
PS> $InstallDate
20120806185927.000000+120
```

```
PS> $LastBootDate
20121113065721.356498+060
```

**Listing 11.4** Das Skript *WMI\_dateconverter.ps1*

In jedes WMI-Objekt ist jedoch die passende Methode zum Entschlüsseln eingebaut: *ConvertToDateTime()*.

```
PS> $os.ConvertToDateTime($InstallDate)
Montag, 6. August 2012 18:59:27
```

```
PS> $os.ConvertToDateTime($LastBootDate)
Dienstag, 13. November 2012 06:57:21
```

*ConvertToDateTime()* ist eigentlich aber gar nicht vorhanden, sondern wurde von PowerShell zuvor-kommenderweise als *ScriptMethod* hinzugefügt. Was genau diese *ScriptMethod* macht, ist diesmal allerdings nicht so leicht zu entschlüsseln, denn der zugrunde liegende PowerShell-Code wird bei *ScriptMethods* nicht in der Eigenschaft *Definition* verraten:

```
PS> $os | Get-Member -Name ConvertToDateTime
```

```
TypeName: System.Management.ManagementObject#root\cimv2\Win32_OperatingSystem
```

| Name              | MemberType   | Definition                         |
|-------------------|--------------|------------------------------------|
| ConvertToDateTime | ScriptMethod | System.Object ConvertToDateTime(); |

Über einen kleinen Kniff kommen Sie aber doch noch elegant an den Code: Rufen Sie die Signatur der Methode ab, indem Sie die Methode ohne die runden Klammern angeben:

```
PS> $os.ConvertToDateTime
```

```
Script           : [System.Management.ManagementDateTimeConverter]::ToDateTime($args[0])
OverloadDefinitions : {System.Object ConvertToDateTime();}
MemberType       : ScriptMethod
TypeNameOfValue   : System.Object
Value            : System.Object ConvertToDateTime();
```

```
Name           : ConvertToDateTime
IsInstance     : False
```

Oder aber Sie begeben sich in den zugrunde liegenden *TypeDefinition*-Dateien auf Spurensuche. Wissen Sie noch, welche das waren? Genau:

```
PS> $Host.Runspace.InitialSessionState.Types | Select-Object -ExpandProperty FileName
C:\Windows\System32\WindowsPowerShell\v1.0\types.ps1xml
C:\Windows\System32\WindowsPowerShell\v1.0\typesv3.ps1xml
```

In *types.ps1xml* würden Sie dann nach etwas Suchen fündig werden:

```
<Type>
  <Name>System.Management.ManagementObject</Name>
  <Members>
    <ScriptMethod>
      <Name>ConvertToDateTime</Name>
      <Script>
        [System.Management.ManagementDateTimeConverter]::ToDateTime($args[0])
      </Script>
    </ScriptMethod>
    <ScriptMethod>
      <Name>ConvertFromDateTime</Name>
      <Script>
        [System.Management.ManagementDateTimeConverter]::ToDmtfDateTime($args[0])
      </Script>
    </ScriptMethod>
  </Members>
</Type>
```

Der Blick hinter die Kulissen verrät also wieder einmal grundsätzliches Know-how: wie man Datums-typen hin- und herkonvertiert:

```
PS> $datum = Get-Date
PS> $wmidatum = [System.Management.ManagementDateTimeConverter]::ToDmtfDateTime($datum)
PS> $datum
Sonntag, 18. November 2012 19:28:08
```

```
PS> $wmidatum
20121118192808.366713+060
```

```
PS> [System.Management.ManagementDateTimeConverter]::ToDateTime($wmidatum)
Sonntag, 18. November 2012 19:28:08
```

Nun folgt noch der Auftritt einer *ParameterizedProperty*:

```
PS> 'Hallo'.Chars(3)
l
```

Auf den ersten Blick sieht hier alles so aus, als wäre *Chars()* eine Methode, denn es stehen ja runde Klammern am Namensende. In Wirklichkeit aber handelt es sich um eine Eigenschaft, die Argumente empfangen kann, eine *ParameterizedProperty*.

Im Grunde bestehen oft nur philosophische Unterschiede zwischen Eigenschaften und Methoden und dem, was sie tun. Eigenschaften sind für den Anwender leichter abzurufen und suggerieren, dass dabei nur statische Daten bewegt, aber kein Code ausgeführt wird. Sie wissen inzwischen, dass das Humbug ist. Auch beim Abruf einer Eigenschaft kann Code ausgeführt werden (*ScriptProperty*) und Eigenschaften und Methoden können grundsätzlich dasselbe leisten.

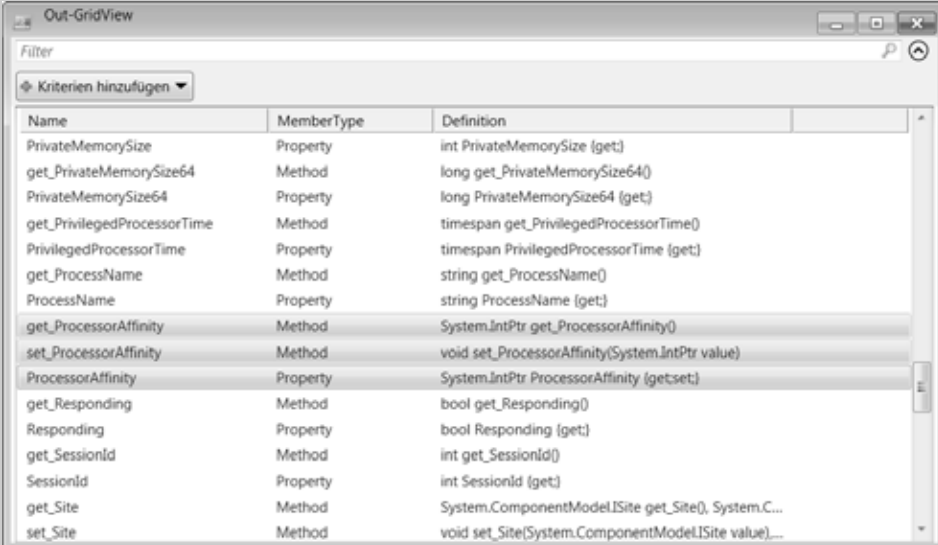


Allerdings gibt es eine wesentliche Regel: Der Abruf einer Eigenschaft muss *sicher* sein. Zwar könnten Sie durchaus eine *ScriptProperty* entwerfen, die beim Abruf die Festplatte formatiert, aber das wäre ein unverzeihlicher Regelverstoß. Eigenschaften dürfen *beim Lesen* also durchaus Code ausführen, aber dieser darf keine bleibenden Änderungen am System verursachen oder langwierig sein. Schließlich zeigt PowerShell den Inhalt von Eigenschaften jedes Mal automatisch an, wenn Sie Objekte ausgeben, und dabei werden alle angezeigten Eigenschaften stets abgerufen. Es wäre also schlimm, wenn dabei jedes Mal Systemveränderungen eintreten oder es zu längeren Verzögerungen käme.

Technisch gesehen gibt es überhaupt keine Eigenschaften. Denn Eigenschaften werden hinter den Kulissen ebenfalls über Methoden abgebildet, die lediglich versteckt sind. Wie das alles zusammenhängt, zeigt die folgende Zeile, die jeweils die Eigenschaften mit ihren zugrunde liegenden Methoden anzeigt:

```
# alle Prozesse abrufen:
PS> Get-Process |
  # für diese Prozesse ALLE Member auflisten, auch die normalerweise versteckten:
  Get-Member -force |
  # gruppieren nach Member-Name, aber "get_" und "set_" dabei nicht berücksichtigen:
  Group-Object { $_.Name -replace '(get_|set_)' } |
  # nur Gruppen mit mindestens 2 Elementen anzeigen (Eigenschaft und zugehörige Methode):
  Where-Object { $_.Count -gt 1 } |
  # Gruppenmitglieder ausgeben:
  Select-Object -ExpandProperty Group |
  # im GridView anzeigen:
  Out-GridView
```

Das Ergebnis sind Gruppen bestehend aus der Eigenschaft und ihrer *get\_-*Methode (wenn sie nur lesbar ist) sowie ihrer *set\_-*Methode (wenn sie auch änderbar ist):



| Name                        | MemberType | Definition  |
|-----------------------------|------------|---|
| PrivateMemorySize           | Property   | int PrivateMemorySize {get;}                        |
| get_PrivateMemorySize64     | Method     | long get_PrivateMemorySize64()                      |
| PrivateMemorySize64         | Property   | long PrivateMemorySize64 {get;}                     |
| get_PrivilegedProcessorTime | Method     | timespan get_PrivilegedProcessorTime()              |
| PrivilegedProcessorTime     | Property   | timespan PrivilegedProcessorTime {get;}             |
| get_ProcessName             | Method     | string get_ProcessName()                            |
| ProcessName                 | Property   | string ProcessName {get;}                           |
| get_ProcessorAffinity       | Method     | System.IntPtr get_ProcessorAffinity()               |
| set_ProcessorAffinity       | Method     | void set_ProcessorAffinity(System.IntPtr value)     |
| ProcessorAffinity           | Property   | System.IntPtr ProcessorAffinity {get;set;}          |
| get_Responding              | Method     | bool get_Responding()                               |
| Responding                  | Property   | bool Responding {get;}                              |
| get_SessionId               | Method     | int get_SessionId()                                 |
| SessionId                   | Property   | int SessionId {get;}                                |
| get_Site                    | Method     | System.ComponentModel.ISite get_Site(), System.C... |
| set_Site                    | Method     | void set_Site(System.ComponentModel.ISite value)... |

Abbildung 11.5 Eigenschaften und die zugrunde liegenden *get\_-* und *set\_-*Methoden

Die Entscheidung, ob Sie für eigene Zwecke also zu einer *ScriptProperty* oder doch lieber zu einer *ScriptMethod* greifen, sollte also auf der Überlegung beruhen, ob der Abruf von Daten sicher ist und ob vom Benutzer zusätzliche Argumente erforderlich sind. Wichtig wird die Wahl auch dann, wenn Sie eine vorhandene Methode überschreiben wollen. In diesem Fall müssen Sie dafür natürlich entsprechend auf eine *ScriptMethod* zurückgreifen.

Jedes Objekt besitzt zum Beispiel die Methode *ToString()*, mit der es in darstellbaren Text verwandelt wird. Wie diese Umwandlung passieren muss, bestimmt das Objekt selbst – normalerweise. Wenn Sie eine *ScriptMethod* namens *ToString()* hinzufügen, können plötzlich Sie bestimmen, wie ein Objekt sich darstellt. Das kann nützlich sein, wenn Sie beispielsweise numerische Daten besser lesbar machen wollen, ohne ihren numerischen Charakter zu zerstören.

Der folgende Code speichert in *\$zahl* eine große Zahl. Danach wird die eingebaute Methode *ToString()* mit einer eigenen überschrieben. Diese teilt den Inhalt der Zahl (*\$this* ist der Inhalt) durch *1GB* und formatiert die Ausgabe mit dem Operator *-f* als Zahl mit Tausendertrennzeichen und zwei Nachkommastellen. Das Ergebnis ist also ein Text.

```
PS> $zahl = 56757564723234
PS> $zahl | Add-Member -MemberType ScriptMethod -Name ToString -Value { '{0:n2} GB' -f ($this/1GB) } -Force
PS> $zahl
52.859,60 GB
```

Tatsächlich wird die Zahl jetzt in GB angezeigt. Ihr Inhalt hat sich aber nicht verändert und ist immer noch eine Zahl, mit der man auch immer noch sortieren oder Vergleiche durchführen kann:

```
PS> $zahl.GetType().FullName
System.Int64

PS> $zahl -gt 435675364
True

PS> $zahl -gt 4356753646786387
False
```

Allerdings geht die neue *ScriptMethod* sofort verloren, wenn der Zahl ein neuer Wert zugewiesen wird, was indirekt auch beim Inkrementieren passiert. PowerShell speichert dabei die alte Zahl nämlich in einer neuen Variable:

```
PS> $zahl++
PS> $zahl
56757564723235
```

Obwohl die Erweiterung hier also eher kurzlebig war, kann man die Technik durchaus in Kombination mit anderen Erweiterungen sinnvoll nutzen. Die folgende *ScriptProperty* fügt beispielsweise die Eigenschaft *LengthKB* hinzu und zeigt die Dateigröße nun wahlweise auch in KB an:

```
Get-ChildItem -Path $env:windir -File |
  Add-Member -MemberType ScriptProperty -Name LengthKB -Value {
    $this.Length | Add-Member -MemberType ScriptMethod -Name ToString -Value { '{0:n2} KB'
-f ($this/1KB) } -Force -PassThru
  } -Force -PassThru |
  Select-Object -Property Mode, LastWriteTime, LengthKB, Name
```

**Listing 11.5** Das Skript *filesize\_kb.ps1*

Das Ergebnis sieht so aus:

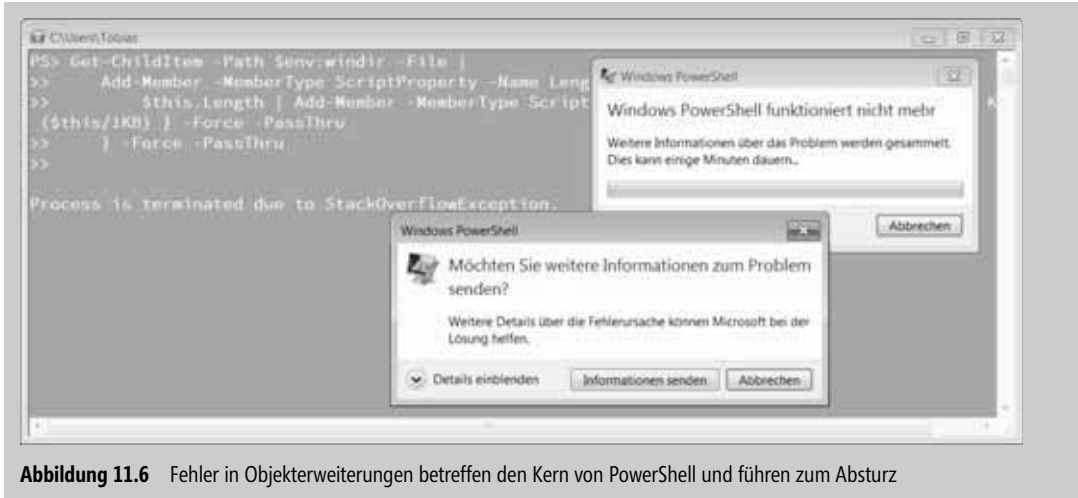
| Mode  | LastWriteTime       | LengthKB    | Name         |
|-------|---------------------|-------------|--------------|
| d---- | 29.07.2009 07:20:19 | 0,00 KB     | ABLKSR       |
| d---- | 14.07.2009 07:32:39 | 0,00 KB     | addins       |
| d---- | 06.08.2012 22:06:45 | 0,00 KB     | AppCompat    |
| d---- | 11.10.2012 20:29:08 | 0,00 KB     | AppPatch     |
| d---- | 11.04.2011 14:04:45 | 0,00 KB     | ar-SA        |
| -a--- | 06.08.2012 18:59:32 | 535,78 KB   | AsCDProc.log |
| -a--- | 07.07.2012 14:39:58 | 66,35 KB    | AsChkDev.txt |
| ----- | 30.04.2012 10:03:31 | 0,02 KB     | AsDCDVer.txt |
| -a--- | 06.08.2012 18:59:32 | 4.217,78 KB | AsDebug.log  |
| -a--- | 24.02.2012 02:33:34 | 82,76 KB    | AsFac.log    |
| (...) |                     |             |              |

*Add-Member* wird häufig zum Testen und als Prototyper verwendet, und auch hier offenbart das Ergebnis noch einige Defizite: Ordner zeigen jetzt eine Größe von *0,00 KB* an, denn sie erhalten genau wie Dateien die neue Eigenschaft *LengthKB*, obwohl sie keine Eigenschaft *Length* besitzen. Wenn Sie das Ergebnis an *Out-GridView* weiterleiten, ist zudem die Spalte *LengthKB* leer. Offenkundig kommt dieses Cmdlet nicht mit dem Konflikt zwischen Datentyp und Darstellung zurecht.

Auch sonst ist es nicht immer trivial, stabile und sichere Erweiterungen herzustellen. Falls Sie beispielsweise nicht eine neue Eigenschaft namens *LengthKB* anfügen, sondern sich dazu entschließen, die bestehende Eigenschaft *Length* zu ändern, nimmt PowerShell das zwar hin, stürzt danach aber umgehend ab:

```
Get-ChildItem -Path $env:windir -File |
  Add-Member -MemberType ScriptProperty -Name Length -Value {
    $this.Length | Add-Member -MemberType ScriptMethod -Name ToString -Value { '{0:n2} KB'
-f ($this/1KB) } -Force -PassThru
  } -Force -PassThru
```

Wenn Sie nämlich diejenige Eigenschaft überschreiben, auf die Sie intern in Ihrer *ScriptMethod* zugreifen, produzieren Sie eine Endlosschleife, denn wenn Ihre *ScriptMethod* die Eigenschaft *Length* ausliest, ruft sie sich jetzt immer wieder selbst auf. Ohnehin ist es keine gute Idee, vorhandene Eigenschaften zu überschreiben, weil das zu Inkonsistenzen führt: Dort, wo die Erweiterung vorhanden ist, liefern dann dieselben Aktionen andere Resultate als anderswo.



**Abbildung 11.6** Fehler in Objekterweiterungen betreffen den Kern von PowerShell und führen zum Absturz

## Membertypen für den internen Gebrauch

Einige Erweiterungstypen werden von PowerShell für interne Zwecke benötigt. Sie können diese Membertypen zwar ignorieren, wenn Sie gerade in Zeitnot sind. Allerdings verstehen Sie deutlich besser, wie PowerShell eigentlich funktioniert, wenn Sie noch einen Moment am Ball bleiben.

## PropertySet: Gruppen von Eigenschaften

*PropertySet*-Erweiterungen sind Gruppen von Eigenschaften. Möchte man beispielsweise von bestimmten Objekten für gewisse Fragestellungen immer wieder genau dieselben Eigenschaften abrufen, braucht man sie nicht jedes Mal einzeln anzugeben. Stattdessen erweitert man den Objekttyp um ein neues *PropertySet*, in dem dann die gewünschten Eigenschaften zusammengefasst sind. *Process*-Objekte verfügen zum Beispiel über zwei vordefinierte *PropertySets*:

```
PS> Get-Process | Get-Member -MemberType PropertySet
```

```
    TypeName: System.Diagnostics.Process
```

| Name            | MemberType  | Definition                                       |
|-----------------|-------------|--|
| PSConfiguration | PropertySet | PSConfiguration {Name, Id, PriorityClass, Fil... |
| PSResources     | PropertySet | PSResources {Name, Id, Handlecount, WorkingSe... |

Sie können Prozesslisten also gezielt nach den Fragestellungen »Konfiguration« und »Ressourcenlast« abrufen. Ein und derselbe Befehl (*Get-Process* in diesem Fall) liefert so ganz unterschiedliche Ergebnisse:

```
# Standard-Eigenschaften anzeigen:
PS> Get-Process | Select-Object -First 5
```

| Handles | NPM(K) | PM(K) | WS(K) | VM(M) | CPU(s) | Id   | ProcessName |
|---------|--------|-------|-------|-------|--------|------|-------------|
| 95      | 14     | 2848  | 3252  | 59    | 6,01   | 5724 | ACEngSvr    |
| 108     | 11     | 2368  | 2768  | 75    | 3,63   | 7048 | ACMON       |
| 281     | 29     | 56792 | 3624  | 229   | 86,21  | 924  | AcroRd32    |
| 291     | 20     | 6668  | 1800  | 101   | 0,61   | 6896 | AcroRd32    |
| 39      | 6      | 1868  | 340   | 55    | 0,05   | 8096 | ADDEL       |

# Eigenschaften für Fragestellung "Konfiguration" abrufen:

```
PS> Get-Process | Select-Object -Property PSConfiguration -First 5
```

| Name     | Id   | PriorityClass | FileVersion |
|----------|------|---------------|-------------|
| ACEngSvr | 5724 | Normal        | 1, 0, 0, 4  |
| ACMON    | 7048 | Normal        | 1, 0, 9, 0  |
| AcroRd32 | 924  | Normal        | 10.1.4.38   |
| AcroRd32 | 6896 | Normal        | 10.1.4.38   |
| ADDEL    | 8096 |               |             |

# Eigenschaften für Fragestellung "Ressourcen-Einsatz" abrufen:

```
PS> Get-Process | Select-Object -Property PSResources -First 5
```

```
Name          : ACEngSvr
Id             : 5724
HandleCount    : 95
WorkingSet     : 3330048
PagedMemorySize : 2916352
PrivateMemorySize : 2916352
VirtualMemorySize : 61751296
TotalProcessorTime : 00:00:06.0060385
```

```
Name          : ACMON
Id             : 7048
HandleCount    : 108
WorkingSet     : 2834432
PagedMemorySize : 2424832
PrivateMemorySize : 2424832
VirtualMemorySize : 78458880
TotalProcessorTime : 00:00:03.6348233
```

```
Name          : AcroRd32
Id             : 924
(...)
```

## MemberSet: Wie soll PowerShell das Objekt behandeln?

Ein *MemberSet* wird ausschließlich für interne Zwecke eingesetzt und legt fest, wie PowerShell mit den Mitgliedern eines Objekts umgehen soll. Hier wird zum Beispiel bestimmt, welche Eigenschaften von PowerShell als Vorgabe angezeigt werden:

```
PS> Get-Process -ID $pid | Format-List
```

```
Id          : 11724
Handles     : 492
CPU         : 15,9433022
```

```
Name      : powershell
```

Warum zum Beispiel zeigt *Format-List* ausgerechnet diese Eigenschaften an? Die Antwort liefert das *MemberSet* mit dem Namen *PSStandardMembers*, das allerdings normalerweise versteckt ist und deshalb nur gezeigt wird, wenn Sie auf den Tisch hauen und *-Force* sagen:

```
PS> Get-Process -ID $pid | Get-Member PSStandardMembers
```

```
PS> Get-Process -ID $pid | Get-Member PSStandardMembers -Force
```

```
    TypeName: System.Diagnostics.Process
```

```
Name              MemberType Definition
----              -
PSStandardMembers MemberSet   PSStandardMembers {DefaultDisplayPropertySet}
```

In diesem *MemberSet* ist in der Eigenschaft *DefaultDisplayPropertySet* hinterlegt, welches die Standard-eigenschaften sind, die PowerShell dann anzeigt, wenn nicht anderweitig bestimmt wird, welche Eigenschaften anzuzeigen sind:

```
PS> (Get-Process -ID $pid).PSStandardMembers.DefaultDisplayPropertySet
```

```
ReferencedPropertyNames : {Id, Handles, CPU, Name}
MemberType               : PropertySet
Value                   : DefaultDisplayPropertySet {Id, Handles, CPU, Name}
TypeNameOfValue         : System.Management.Automation.PSPropertySet
Name                    : DefaultDisplayPropertySet
IsInstance              : False
```

```
PS> (Get-Process -ID $pid).PSStandardMembers.DefaultDisplayPropertySet.ReferencedPropertyNames
Id
Handles
CPU
Name
```

### PROFITIPP

Was die Frage aufwirft, wieso *Get-Process* ganz andere Eigenschaften liefert, wenn anstelle von *Format-List* das Cmdlet *Format-Table* verwendet wird – oder gar keins. Die Liste der Eigenschaften im *MemberSet* wird nur verwendet, wenn *nicht anderweitig* die gewünschten Eigenschaften angegeben sind. Muss PowerShell ein Objekt in Text umwandeln, geht es so vor:

- **Objekttyp bestimmt Formatierung** PowerShell ermittelt zuerst den Typ des Objekts. Alle Objekte desselben Typs werden von PowerShell auf dieselbe Weise formatiert.
- **Interne Format-»Datenbank« bestimmt Standardformatierung** Danach schlägt PowerShell in seinen *.format.ps1xml*-Dateien nach, ob für diesen Objekttyp eine eigene *View* definiert ist. Die Standardformatdateien liegen im Ordner *\$PSHOME*, tragen die Erweiterung *.format.ps1xml* und enthalten XML. Ihr Aufbau ist relativ kompliziert. Diese Formatinformationen können von Modulen erweitert werden.

Die aktuelle Liste aller geladenen Formatdateien liefert diese Zeile:

```
PS> $Host.Runspace.InitialSessionState.Formats | Select-Object -ExpandProperty FileName
C:\Windows\System32\WindowsPowerShell\v1.0\Certificate.format.ps1xml
C:\Windows\System32\WindowsPowerShell\v1.0\DotNetTypes.format.ps1xml
C:\Windows\System32\WindowsPowerShell\v1.0\FileSystem.format.ps1xml
C:\Windows\System32\WindowsPowerShell\v1.0\Help.format.ps1xml
```

```
C:\Windows\System32\WindowsPowerShell\v1.0\HelpV3.format.ps1xml
C:\Windows\System32\WindowsPowerShell\v1.0\PowerShellCore.format.ps1xml
C:\Windows\System32\WindowsPowerShell\v1.0\PowerShellTrace.format.ps1xml
C:\Windows\System32\WindowsPowerShell\v1.0\Registry.format.ps1xml
C:\Windows\system32\WindowsPowerShell\v1.0\WSMan.format.ps1xml
```

- **DefaultDisplayPropertySet wird nur bei Bedarf konsultiert** Nur wenn für die gewünschte Anzeige in keiner der *.format.ps1xml*-Dateien eine Anweisung gefunden wurde, schaut PowerShell, ob das Objekt über ein *MemberSet* namens *DefaultDisplayPropertySet* verfügt. Falls ja, werden die darin genannten Eigenschaften angezeigt. Allerdings wird bei Tabellendarstellung die Spaltenbreite jetzt nicht mehr vorgegeben, weswegen die Spalten gleichmäßig über die Breite verteilt werden müssen.
- **Standardverhalten** Ist auch kein *MemberSet* vorhanden, zeigt PowerShell sämtliche Objekteigenschaften an. Bei vier oder weniger erscheint eine Tabelle, sonst eine Liste. Bei der Tabellendarstellung werden die Spalten gleichmäßig über die Breite verteilt.

In den *.format.ps1xml*-Dateien von PowerShell ist für Objekte vom Typ *System.Diagnostics.Process* hinterlegt, dass die Standardanzeige (*Default View*) ein *TableControl* sein soll. Hier steht auch, welche Spalten dann unter welchem Namen in der Tabelle angezeigt werden und wie breit sie sein sollen.

Setzen Sie also keinen besonderen *Format*-Befehl ein, verwendet PowerShell diese Angaben und produziert eine Tabelle. Wünschen Sie ausdrücklich mit *Format-Table* eine Tabelle, ist das ebenfalls kompatibel zur Standardanzeige, die ja eine Tabelle wünscht. *Format-List* hingegen hat keine Standardanzeige. Deshalb nutzt PowerShell in diesem Fall als absolut letzte Möglichkeit den Inhalt des *MemberSets*.

Was es genau mit den *.format.ps1xml*-Dateien auf sich hat und wie Sie eigene zusätzliche Views definieren, lesen Sie in Kapitel 20. So können Sie nämlich PowerShell beibringen, wie Ihre eigenen Objekttypen formatiert und angezeigt werden sollen. Das allerdings funktioniert nur, wenn Sie Ihre Funktionen in einem Modul verpacken.

Ohne Modul und *.format.ps1xml*-Dateien geht es aber auch, wie Sie gerade erfahren haben. Deshalb lesen Sie in Kapitel 17, wie Sie die Formatierung und Darstellung der Ergebnisse Ihrer Funktionen auch mit einem selbstgemachten *MemberSet* festlegen können.

---

*MemberSets* definieren nicht nur die Standardeigenschaften eines Objekts. Sie legen zum Beispiel auch fest, wie Objekte serialisiert werden (also welche Eigenschaften auf welche Weise bis zu welcher Verschachtelungstiefe in XML gespeichert werden). Dies allerdings soll hier nicht weiter vertieft werden.

## CodeProperty: Statische Methoden aufrufen

Eine *CodeProperty* ruft eine statische Methode eines .NET-Typs auf, um den Inhalt der Eigenschaft zu liefern. Das ist zwar eine tolle Sache, aber vielleicht doch erst nach Lektüre der nächsten beiden Kapitel. Eventuell sollten Sie danach noch einmal an diese Stelle zurückkehren. Bevor geklärt wird, was eine *CodeProperty* ist und wie sie funktioniert, soll zuerst ein Praxisbeispiel bei Ihnen für eine positive Grundstimmung sorgen. Es liefert nämlich sehr nützliche Ergebnisse, die – wie sich herausstellen wird – nur dank *CodeProperties* möglich waren. Haben Sie es eilig, dürfen Sie den Inhalt des Kastens aber auch überlesen.

Hier eine typische Fragestellung: »Wer hat alle Berechtigungen auf bestimmte Ordner oder Dateien?« Das soll mit *Get-Acl* herausgefunden werden, und Ihr zunehmendes Objektverständnis hilft enorm dabei:

```
PS> $zugang = Get-Acl -Path c:\windows
PS> $zugang
```

Verzeichnis: C:\

| Path    | Owner                     | Access                   |
|---------|---------------------------|--------------------------|
| -----   | -----                     | -----                    |
| windows | NT SERVICE\TrustedInst... | ERSTELLER-BESITZER A1... |

Die notwendigen Details sehen Sie allerdings erst, wenn Sie sich alle Eigenschaften des Objekts anzeigen lassen:

```
PS> $zugang | Select-Object -Property *
```

```
PSPath           : Microsoft.PowerShell.Core\FileSystem::C:\windows
PSParentPath     : Microsoft.PowerShell.Core\FileSystem::C:\
PSChildName      : windows
PSDrive          : C
PSProvider       : Microsoft.PowerShell.Core\FileSystem
CentralAccessPolicyId :
CentralAccessPolicyName :
AccessToString   : ERSTELLER-BESITZER Allow 268435456
                  NT-AUTORITÄT\SYSTEM Allow 268435456
                  NT-AUTORITÄT\SYSTEM Allow Modify, Synchronize
                  VORDEFINIERT\Administratoren Allow 268435456
                  VORDEFINIERT\Administratoren Allow Modify,
                  Synchronize
                  VORDEFINIERT\Benutzer Allow -1610612736
                  VORDEFINIERT\Benutzer Allow ReadAndExecute,
                  Synchronize
                  NT SERVICE\TrustedInstaller Allow 268435456
                  NT SERVICE\TrustedInstaller Allow FullControl

AuditToString    :
Path             : Microsoft.PowerShell.Core\FileSystem::C:\windows
Owner            : NT SERVICE\TrustedInstaller
Group            : NT SERVICE\TrustedInstaller
Access           : {System.Security.AccessControl.FileSystemAccessRule, System.Security.AccessControl.FileSystemAccessRule, System.Security.AccessControl.FileSystemAccessRule, System.Security.AccessControl.FileSystemAccessRule...}

Sddl             : O:S-1-5-80-956008885-3418522649-1831038044-1853292631-2271478464G:S-1-5-80-956008885-3418522649-1831038044-1853292631-2271478464D:PAI(A;OICIIO;GA;;;CO)(A;OICIIO;GA;;;SY)(A;0x1301bf;;;SY)(A;OICIIO;GA;;;BA)(A;0x1301bf;;;BA)(A;OICIIO;GXGR;;;BU)(A;0x1200a9;;;BU)(A;CIIIO;GA;;;S-1-5-80-956008885-3418522649-1831038044-1853292631-2271478464)(A;FA;;;S-1-5-80-956008885-3418522649-1831038044-1853292631-2271478464)

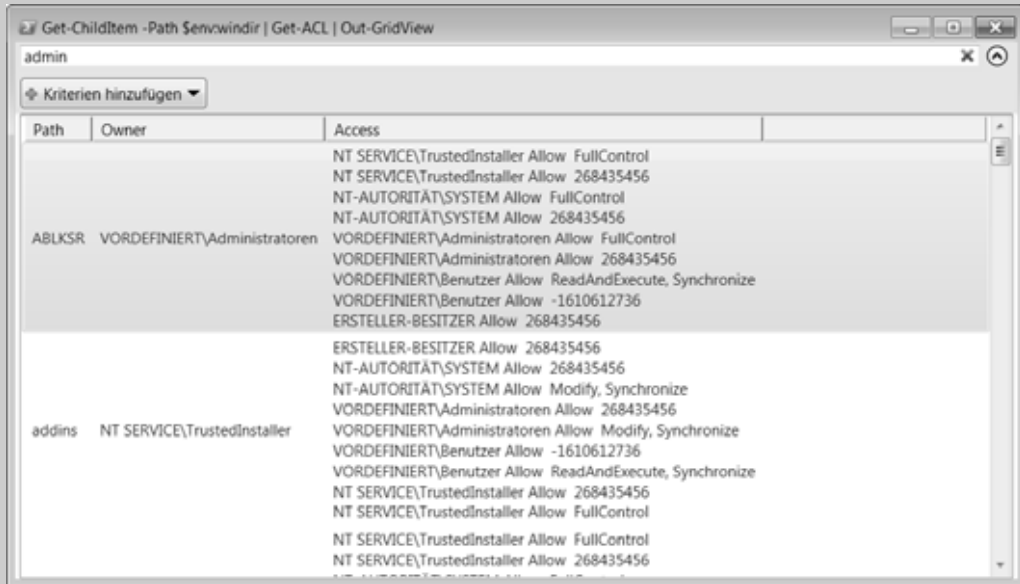
AccessRightType  : System.Security.AccessControl.FileSystemRights
AccessRuleType   : System.Security.AccessControl.FileSystemAccessRule
(...)
```



**TIPP**

Vielleicht genügen Ihnen auch schon die Standardinformationen, solange sie nicht von PowerShell abgeschnitten und verstümmelt werden. Leiten Sie das Ergebnis einfach weiter an *Out-GridView*, um alle Informationen ungekürzt zu sehen. Die folgende Zeile zeigt die Berechtigungsinformationen für alle Ordner innerhalb des Windows-Ordners an:

```
PS> Get-ChildItem -Path $env:windir | Get-Acl | Out-GridView
```



**Abbildung 11.7** Alle Berechtigungen der Unterordner im Windows-Ordner anzeigen

Allerdings können Sie in dieser Darstellung nicht nach einzelnen Benutzern filtern, sodass der Zugang zu den Objekteigenschaften vielleicht doch eine gute Idee ist.

Die Sicherheitsinformationen stecken in den Eigenschaften *AccessToString* (als bequeme Textdarstellung), *Access* (als detailreiche Unterobjekte) und *Sddl* (als formalisierter Berechtigungs-String), je nachdem, in welcher Form Sie die Informationen gerade gebrauchen können:

```
PS> $zugang.AccessToString
ERSTELLER-BESITZER Allow 268435456
NT-AUTORITÄT\SYSTEM Allow 268435456
NT-AUTORITÄT\SYSTEM Allow Modify, Synchronize
VORDEFINIERT\Administratoren Allow 268435456
VORDEFINIERT\Administratoren Allow Modify, Synchronize
VORDEFINIERT\Benutzer Allow -1610612736
VORDEFINIERT\Benutzer Allow ReadAndExecute, Synchronize
NT SERVICE\TrustedInstaller Allow 268435456
NT SERVICE\TrustedInstaller Allow FullControl
```

```
PS> $zugang.Access
```

```
FileSystemRights : 268435456
AccessControlType : Allow
IdentityReference : ERSTELLER-BESITZER
IsInherited       : False
InheritanceFlags  : ContainerInherit, ObjectInherit
PropagationFlags  : InheritOnly
```

```
FileSystemRights : 268435456
AccessControlType : Allow
IdentityReference : NT-AUTORITÄT\SYSTEM
IsInherited       : False
InheritanceFlags  : ContainerInherit, ObjectInherit
PropagationFlags  : InheritOnly
```

```
FileSystemRights : Modify, Synchronize
(...)
```

```
PS> $zugang.Sddl
O:S-1-5-80-956008885-3418522649-1831038044-1853292631-2271478464G:S-1-5-80-956008885-3418522649-1831038044-1853292631-2271478464D:PAI(A;OICIIO;GA;;;CO)(A;OICIIO;GA;;;SY)(A;;0x1301bf;;;SY)(A;OICIIO;GA;;;BA)(A;;0x1301bf;;;BA)(A;OICIIO;GXGR;;;BU)(A;;0x1200a9;;;BU)(A;CIIIO;GA;;;S-1-5-80-956008885-3418522649-1831038044-1853292631-2271478464)(A;;FA;;;S-1-5-80-956008885-3418522649-1831038044-1853292631-2271478464)
```

Welche Darstellung für Sie besser ist, hängt vom Fall ab, aber wie immer sind die von *Access* gelieferten Objekte am flexibelsten. Damit könnten Sie eine Funktion namens *Get-NTFSPermission* schreiben, die künftig die NTFS-Berechtigungen für beliebige Ordner und Dateien analysiert. *Get-NTFSPermission* ist ein vollständiges Script-Cmdlet und verpackt die Low-Level-Zugriffe genauso wie echte Cmdlets. Die Funktion ist ausführlich kommentiert, sodass Sie die Prinzipien sicher nachvollziehen können. Was genau innerhalb der Funktion geschieht, ist aber nicht Gegenstand dieses Kapitels, sondern wird in Teil D dieses Buchs ausführlich besprochen.

```
function Get-NTFSPermission
{
    param
    (
        [Parameter(ValueFromPipeline=$true,ValueFromPipelineByPropertyName=$true,Mandatory=$true)]
        [String[]]
        # ...und auch in Objekten, die die Eigenschaft "FullName" oder "Path" besitzen
        # (also zum Beispiel Ergebnisse von Get-ChildItem)
        [Alias('FullName')]
        $Path
    )

    # der process-Block wird für jedes empfangene Pipeline-Element wiederholt:

    Process
    {
        # falls mehrere Pfade kommasepariert angegeben wurden, einzeln bearbeiten:
        $Path | ForEach-Object {
            # Pfad merken für später
            $currentPath = $_
            # NTFS-Berechtigungen lesen
            # das kann zu terminierenden Fehlern führen (bei fehlenden Berechtigungen)
            # daher ist dieser Teil in einen try-Block gestellt (siehe "Fehlerhandler")
            try
```

```

{
  $ACL = Get-Acl -Path $currentPath

  # Access Control Entries einzeln bearbeiten und für jeden davon ein
  # Ergebnisobjekt liefern:
  $ACL.Access | ForEach-Object {
    # Ergebnisobjekt setzt sich zusammen aus VORHANDENEN und NEUEN Eigenschaften
    # Neue Eigenschaften sind "Path", "Identity", "Right" und "Type", denn diese waren
    # vorher nicht vorhanden. Alle bestehenden Eigenschaften, die "Inheri" enthalten,
    # werden übernommen:
    $Result = $_ | Select-Object -Property Path, Identity, Right, Type, *Inheri*
    # "Path" wird der aktuelle Pfad zugewiesen. Diese Information fehlte in ACL-Objekten
    # bisher:
    $Result.Path = $currentPath
    # den übrigen neuen Eigenschaften werden die unveränderten alten Eigenschaften
    # zugewiesen.
    # dies geschieht nur, um den Eigenschaften bessere (schönere, verständlicherer) Namen
    # zu geben:
    $Result.Identity = $_.IdentityReference
    $Result.Type = $_.AccessControlType
    $Result.Right = $_.FileSystemRights
    # danach wird das Ergebnisobjekt zurückgeliefert:
    $Result
  }
}
catch
{
  Write-Warning "Kein Zugriff auf $currentPath. Fehler: '$_'"
}
}
}
}

```

**Listing 11.6** Das Skript *Get-NTFSPermission.ps1*

Sie können *Get-NTFSPermission* nun genauso einfach verwenden wie Cmdlets. Die nächste Zeile generiert zum Beispiel einen NTFS-Report für die Dateien und Ordner im Windows-Ordner:

```

# Alle NTFS-Rechte aller Dateien und Ordner im Windows-Ordner
Get-ChildItem -Path $env:windir |
Get-NTFSPermission |
Out-GridView

```

#### TIPP

Was im Report steht, bestimmen Sie: Fügen Sie zu *Get-ChildItem* den Parameter *-Recurse* hinzu, dann werden auch alle Unterordner berücksichtigt. Und wenn Sie nach bestimmten Dateierweiterungen suchen, werden eben diese analysiert. *Get-NTFSPermission* ist es egal, welche und wie viele Dateien oder Ordner Sie der Funktion liefern.

| Path   | Identity                     | Right                       | Type  | Inherited | InheritanceFlags |
|--|------------------------------|-----------------------------|-------|-----------|------------------|
| C:\Windows\AsScrPro.exe                          | NT-AUTORITÄT\SYSTEM          | FullControl                 | Allow | True      | None             |
| C:\Windows\AsScrPro.exe                          | VORDEFINIERT\Administratoren | FullControl                 | Allow | True      | None             |
| C:\Windows\AsScrPro.exe                          | VORDEFINIERT\Benutzer        | ReadAndExecute, Synchronize | Allow | True      | None             |
| C:\Windows\ASUS_Scr_ZenbookPrime Uninstaller.exe | NT-AUTORITÄT\SYSTEM          | FullControl                 | Allow | True      | None             |
| C:\Windows\ASUS_Scr_ZenbookPrime Uninstaller.exe | VORDEFINIERT\Administratoren | FullControl                 | Allow | True      | None             |
| C:\Windows\ASUS_Scr_ZenbookPrime Uninstaller.exe | VORDEFINIERT\Benutzer        | ReadAndExecute, Synchronize | Allow | True      | None             |
| C:\Windows\bfsvc.exe                             | NT SERVICE\TrustedInstaller  | FullControl                 | Allow | False     | None             |
| C:\Windows\bfsvc.exe                             | NT-AUTORITÄT\SYSTEM          | ReadAndExecute, Synchronize | Allow | False     | None             |

**Abbildung 11.8** Im GridView werden die einzelnen Berechtigungen angezeigt und lassen sich sogar filtern

Wenn Sie Abbildung 11.8 mit Abbildung 11.7 vergleichen, werden Sie sicher zustimmen, dass sich der Aufwand gelohnt hat. Durch den direkten Objektzugriff listet der Report jetzt die Berechtigungen als Einzeleinträge mit sehr viel verständlicheren Eigenschaftennamen auf.

Im GridView könnten Sie nun bereits spannende Sicherheitsanalysen fahren: welche Personen haben eigentlich Zugriff auf welche Dateien (und warum?). Geben Sie dazu lediglich Stichworte oben ins Textfeld des GridViews ein. In Abbildung 11.8 werden so beispielsweise nur Dateien mit der Erweiterung *.exe* angezeigt.

Der neue Befehl *Get-NTFSPermission* basiert hauptsächlich auf dem Cmdlet *Get-Acl*, aber die wahren Sicherheitsinformationen lieferten die Eigenschaften *Access* und *Sddl*. Wie sich herausstellt, gibt *Get-Acl* ein Objekt vom Typ *System.Security.AccessControl.DirectorySecurity* zurück, und die genutzten Eigenschaften sind in diesem Objekt eigentlich gar nicht vorhanden. Sie wurden von PowerShell als CodeProperties hinzugefügt:

```
PS> Get-Acl -Path $env:windir | Get-Member -MemberType CodeProperty
```

```
TypeName: System.Security.AccessControl.DirectorySecurity
```

| Name                    | MemberType   | Definition                              |
|-------------------------|--------------|---|
| <b>Access</b>           | CodeProperty | System.Security.AccessControl.Author... |
| CentralAccessPolicyId   | CodeProperty | System.Security.Principal.SecurityId... |
| CentralAccessPolicyName | CodeProperty | System.String CentralAccessPolicyNam... |
| Group                   | CodeProperty | System.String Group{get=GetGroup;}      |
| Owner                   | CodeProperty | System.String Owner{get=GetOwner;}      |
| Path                    | CodeProperty | System.String Path{get=GetPath;}        |
| <b>Sddl</b>             | CodeProperty | System.String Sddl{get=GetSddl;}        |

Welchen Code PowerShell tatsächlich ausführt, wenn Sie die Eigenschaften *Access* oder *Sddl* (oder eine der anderen CodeProperties) abrufen, verrät die Spalte *Definition* indes nicht. Sie müssten den Quellcode dort suchen, wo er dem Objekt hinzugefügt wird, nämlich in einer der Typdateien. Die folgende Zeile listet alle geladenen Typdateien auf:

```
PS> $Host.Runspace.InitialSessionState.Types | Select-Object -ExpandProperty FileName
C:\Windows\System32\WindowsPowerShell\v1.0\types.ps1xml
```

C:\Windows\System32\WindowsPowerShell\v1.0\typesv3.ps1xml

In einer der beiden Dateien müssen die CodeProperties definiert worden sein. Wer sich tatsächlich auf die Suche darin begibt, findet diesen Abschnitt in der Datei *types.ps1xml*:

```
<Type>
  <Name>System.Security.AccessControl.ObjectSecurity</Name>
  <Members>
    <CodeProperty>
      <Name>Path</Name>
      <GetCodeReference>
        <TypeName>Microsoft.PowerShell.Commands.SecurityDescriptorCommandsBase</TypeName>
        <MethodName>GetPath</MethodName>
      </GetCodeReference>
    </CodeProperty>
    <CodeProperty>
      <Name>Owner</Name>
      <GetCodeReference>
        <TypeName>Microsoft.PowerShell.Commands.SecurityDescriptorCommandsBase</TypeName>
        <MethodName>GetOwner</MethodName>
      </GetCodeReference>
    </CodeProperty>
    <CodeProperty>
      <Name>Group</Name>
      <GetCodeReference>
        <TypeName>Microsoft.PowerShell.Commands.SecurityDescriptorCommandsBase</TypeName>
        <MethodName>GetGroup</MethodName>
      </GetCodeReference>
    </CodeProperty>
    <CodeProperty>
      <Name>Access</Name>
      <GetCodeReference>
        <TypeName>Microsoft.PowerShell.Commands.SecurityDescriptorCommandsBase</TypeName>
        <MethodName>GetAccess</MethodName>
      </GetCodeReference>
    </CodeProperty>
    <CodeProperty>
      <Name>Sddl</Name>
      <GetCodeReference>
        <TypeName>Microsoft.PowerShell.Commands.SecurityDescriptorCommandsBase</TypeName>
        <MethodName>GetSddl</MethodName>
      </GetCodeReference>
    </CodeProperty>
    (...)
  </Members>
</Type>
```

Und so wird der Codeabschnitt gelesen: Wenn die Eigenschaft *Access* eines Objekts vom Typ *System.Security.AccessControl.ObjectSecurity* aufgerufen wird, dann soll die statische Methode *GetAccess()* aus dem Typ *Microsoft.PowerShell.Commands.SecurityDescriptorCommandsBase* aufgerufen werden. Technisch wird also jedes Mal, wenn Sie die Eigenschaft *Access* abrufen, in Wirklichkeit die statische Methode *GetAccess()* zum Aufruf gebracht:

```
PS> $ntfs = Get-Acl -Path $env:windir
```

```
# CodeProperty aufrufen:
```

```
PS> $e1 = $ntfs.Access
```

```
# zugrunde liegende statische Methode direkt aufrufen:
PS> $e2 = [Microsoft.PowerShell.Commands.SecurityDescriptorCommandsBase]::GetAccess($ntfs)
```

Der Inhalt von *\$e1* und *\$e2* ist identisch.

## CodeMethod: Abkürzung für statische Methoden

Eine *CodeMethod* ähnelt sehr einer *CodeProperty*: Auch hier werden die Informationen durch den Aufruf einer statischen Methode eines .NET-Typs gewonnen. Der einzige Unterschied ist, dass die *CodeMethod* eben keine Eigenschaft ist, sondern eine Methode. Ein Beispiel liefert dieser Code:

```
PS> $xml = [XML] '<xml><test>Testwert</test></xml>'
PS> $xml.ToString()
#document
```

Das Ergebnis ist wenig spektakulär: Die Methode *ToString()* wandelt ein XML-Objekt in eine Textdarstellung um. Hinter dieser Methode steckt eine *CodeMethod*:

```
PS> $xml | Get-Member -Name ToString
```

```
    TypeName: System.Xml.XmlDocument
```

```
Name      MemberType Definition
----

```

```
ToString CodeMethod static string XmlNode(psubject instance)
```

Den Quellcode der Methode *ToString* findet man wie bei den *CodeProperties* in einer der Typdateien. Und tatsächlich steht in *types.ps1xml* Folgendes:

```
<Type>
  <Name>System.Xml.XmlNode</Name>
  <Members>
    <CodeMethod>
      <Name>ToString</Name>
      <CodeReference>
        <TypeName>Microsoft.PowerShell.ToStringCodeMethods</TypeName>
        <MethodName>XmlNode</MethodName>
      </CodeReference>
    </CodeMethod>
  </Members>
</Type>
```

Und so wird der Codeabschnitt gelesen: Wenn die Methode *ToString()* eines Objekts vom Typ *System.Xml.XmlNode* aufgerufen wird, dann soll die statische Methode *XmlNode()* aus dem Typ *Microsoft.PowerShell.ToStringCodeMethods* aufgerufen werden. Technisch wird also jedes Mal, wenn Sie bei einem XML-Dokument die Methode *ToString()* aufrufen, in Wirklichkeit dieser Code ausgeführt:

```
PS> [Microsoft.PowerShell.ToStringCodeMethods]::XmlNode($xml)
#document
```

Ein paar Dinge sollten Ihnen eigentlich sonderbar vorkommen. Wieso wurde zum Beispiel die *CodeMethod ToString()* ohne Argumente aufgerufen, aber die echte zugrunde liegende Methode

*XmlNode()* verlangte ein XML-Objekt als Argument? Wenn Sie sich die Signatur der Methode anschauen, wird deutlich, was PowerShell unternimmt:

```
PS> [Microsoft.PowerShell.ToStringCodeMethods]::XmlNode
```

```
OverloadDefinitions
```

```
-----
```

```
static string XmlNode(psubject instance)
```

Der zugrunde liegenden Originalmethode wird die Instanz übergeben, also das Objekt, dessen Code-Methode aufgerufen wurde. Auch ist der Auszug aus der Typdatei *types.ps1xml* eigentlich für Typen namens *System.Xml.XmlNode* gedacht. Das XML-Objekt im Test ist aber vom Typ *System.Xml.XmlDocument*.

```
PS> $xml.GetType().FullName
System.Xml.XmlDocument
```

Wieso wurde dennoch die Erweiterung für *System.Xml.XmlNode* angefügt? Die Antwort liegt in der versteckten Objekteigenschaft *PSTypeNames*, die es bei jedem Objekt gibt:

```
PS> $xml.PSTypeNames
System.Xml.XmlDocument
System.Xml.XmlNode
System.Object
```

Ein Objekt kann also aus mehreren Erweiterungen zusammengesetzt sein. In diesem Fall ist das XML-Objekt zum einen ein ganz allgemeines *System.Object*. Zum anderen ist es auch ein etwas spezielleres *System.Xml.XmlNode*-Objekt (und erhält dessen Erweiterungen). Schließlich ist es auch ein sehr spezielles *System.Xml.XmlDocument*-Objekt (und würde auch dessen Erweiterungen bekommen, wenn es denn welche gäbe). Auch das deckt sich mit den Erkenntnissen aus dem wahren Leben. Eine Katze würde sich beispielsweise so darstellen:

```
PS> $katze.RealWorldTypeNames
Jürgen
Hauskatze
Wildkatze
Felis
Kleinkatze
Säugetier
Tier
Mehrzeller
Zelle
```

Die Aufzählung erstreckt sich also vom Allgemeinen bis zum Speziellen, obwohl PowerShell bei der Anwendung von Typerverweiterungen die Reihenfolge im Ergebnis weitgehend egal ist.

## Objekte permanent erweitern

Sie haben bereits gesehen, dass die Erweiterungsmöglichkeiten, die das ETS bietet, auch Ihnen zur Verfügung stehen. Über *Add-Member* lassen sich alle Erweiterungstypen von Hand und einzeln einem konkreten Objekt hinzufügen. Sie wissen inzwischen aber auch, dass die bereits mitgelieferten Erweiterungen von PowerShell durch Typdateien mit der Erweiterung *.ps1xml* automatisch angefügt

werden. Während Sie diese mitgelieferten Typdateien besser nicht ändern (es sei denn, Sie wollen das Verhalten vorhandener Erweiterungen ändern und das Risiko von daraus resultierenden Wesensänderungen Ihrer PowerShell ist Ihnen klar), dürfen Sie sehr wohl eigene Dateien anlegen und bei Bedarf (oder im Rahmen der Profilskripts automatisch) nachladen. Der Aufbau einer Typenerweiterung ist streng schematisch. Jede Typenerweiterung sieht zunächst einmal so aus:

```
<Types>
  <Type>
    <Name>Typ-Name, für die die Erweiterung gilt</Name>
    <Members>
      [Erweiterung]
    </Members>
  </Type>
</Types>
```

**Listing 11.7** Vorlage\_TypErweiterung.ps1xml

Die Typenerweiterung binden Sie also über *Name* an einen bestimmten Typnamen. Sie wählen damit aus, für welche Objekttypen die Erweiterung gelten soll. *Get-Member* verrät Ihnen im Zweifelsfall stets hinter *TypeName*, wie der genaue Typname eines Objekts heißt. Im Bereich *Members* werden danach die einzelnen Erweiterungen untereinander aufgeführt. Wie diese Erweiterungen jeweils definiert werden, zeigen die folgenden Vorlagen und Beispiele.

Eine komplette Typenerweiterung könnte beispielsweise so aussehen wie in Listing 11.8. Bevor Sie weiterlesen und sich die Vorlage erklären lassen, schauen Sie zuerst selbst darüber und versuchen, die Erweiterung zu verstehen:

```
$Path = "$env:TEMP\autotagger.type.ps1xml"
$erweiterung = '@'
<Types>
  <Type>
    <Name>System.Object</Name>
    <Members>
      <ScriptMethod>
        <Name>AddTag</Name>
        <Script>
          Add-Member -InputObject $this -Name $args[0] -value $args[1] -MemberType Noteproperty
        </Script>
      </ScriptMethod>
    </Members>
  </Type>
</Types>
'@
```

```
Set-Content -Value $erweiterung -Path $Path
Update-TypeData -PrependPath $Path
```

**Listing 11.8** Typenerweiterung *autotagger.type.ps1*

Das Skript definiert eine ETS-Erweiterung, die als *ScriptMethod* namens *AddTag* beliebigen Objekten (Basistyp *System.Object*) hinzugefügt werden soll. Die Methode verarbeitet zwei Argumente: *\$args[0]* und *\$args[1]*. Das erste Argument soll der Name eines *Tags* sein, also einer an das Objekt angehefteten



Information. Das zweite liefert dessen Inhalt. *Add-Member* fügt die Information dann an das aktuelle Objekt (*\$this*) an.

Die Erweiterung wird als Datei gespeichert und dann von *Update-TypeData* ins ETS eingelesen. Danach sind plötzlich ganz erstaunliche neue Dinge möglich:

```
# aktuellen Prozess abrufen:
PS> $process = Get-Process -id $PID

# zwei Zusatzinfos an das Objekt anheften:
PS> $process.AddTag('test', 'mein eigener Prozess')
PS> $process.AddTag('id', 12)

# vorhandene und hinzugefügte Eigenschaften ausgeben:
PS> $process | Select-Object -Property Name, test, id
```

```
Name      test          id
----      -
powershell mein eigener Prozess 12
```

Ab PowerShell 3.0 können solche Typereicherungen auch ohne Umweg über eine Typdatei und ohne kompliziertes XML direkt von *Update-TypeData* geladen werden. Das folgende Skript definiert für alle Objekte eine neue *ScriptMethod* namens *GetHelp()*. Diese ruft intern den Objekttyp ab, generiert dazu die passende Internetadresse bei *MSDN* und ruft die Seite dann in Ihrem Standardbrowser auf:

```
$code = {
    $url = 'http://msdn.microsoft.com/de-de/library/{0}(v=vs.80).aspx' -f $this.GetType().FullName
    Start-Process $url
}
```

```
Update-TypeData -MemberType ScriptMethod -MemberName GetHelp -Value $code -TypeName System.Object
```

**Listing 11.9** Das Skript *GetHelp.ps1*

Sobald Sie das Skript ausgeführt haben, verfügen alle Objekte über einen neuen Befehl namens *GetHelp()*, über den Sie bald bestens gelaunt die zugehörige Hilfe abrufen – so lange jedenfalls, bis Microsoft die Internetadressen für diese Hilfeseiten wieder einmal anpasst:

```
PS> $datum = Get-Date
PS> $datum.GetHelp()
PS> (Get-Date).GetHelp()
```

## DateTime-Struktur

msdn

.NET Framework 2.0 | Andere Versionen ▾ | 3 von 12 fanden dies hilfreich - Dieses Thema bewerten.

Stellt einen Zeitpunkt dar, der normalerweise durch Datum und Uhrzeit dargestellt wird.

**Namespace:** System

**Assembly:** mscorlib (in mscorlib.dll)

### Syntax

```
C# C++ VB
[SerializableAttribute]
public struct DateTime : IComparable, IFormattable, IConvertible,
    ISerializable, IComparable<DateTime>, IQuotable<DateTime>
```

```
J#
/** @attribute SerializableAttribute() */
public final class DateTime extends ValueType implements IComparable, IFormattable,
    IConvertible, ISerializable, IComparable<DateTime>, IQuotable<DateTime>
```

```
JScript
JScript unterstützt die Verwendung von Strukturen, aber nicht die Deklaration von neuen Strukturen.
```

### Hinweise

Ein **DateTime**-Wert stellt Angaben über Datum und Uhrzeit vom 1. Januar 0001, 00:00:00 u. Z. (unserer Zeitrechnung) bis zum 31. Dezember 9999, 23:59:59 u. Z. dar.

Zeitwerte werden in Einheiten von 100 Nanosekunden gemessen, die als Ticks bezeichnet werden. Ein bestimmtes Datum ist die Anzahl Ticks seit dem 1. Januar 0001, 00:00:00 u. Z. nach dem GregorianCalendar-Kalender. Ein Tickswert von 31241376000000000 stellt z. B. Freitag, den 1. Januar 0100, 00:00:00 dar. Ein **DateTime**-Wert bezieht sich immer auf den Kontext eines expliziten Kalenders oder eines Standardkalenders.

Versionaspekte

Abbildung 11.9 Künftig ruft *GetHelp()* für alle Objekte automatisch die Hilfe im Internet ab

Es ist schon erstaunlich, was die Befehlerweiterung mit so wenig Code leistet. Sie ist deshalb allerdings auch ein wenig eingeschränkt:

- **Arrays** liefert die Hilfe zum zugrunde liegenden Array-Datentyp, aber nicht die des Inhalts
- **WMI** liefert keine Hilfe zu WMI-Objekten, weil bei diesen hinter dem Typname der Name der WMI-Klasse steht

Aber natürlich steht einer Überarbeitung nichts im Wege. Die Objekterweiterung, die hier genutzt werden soll, kennt diese Einschränkungen nicht und liefert auch Informationen zu WMI-Objekten. Das allerdings ist nicht leicht, weil die Internetadressen dafür leider nicht so »sprechend« sind wie die für .NET-Typen. Daher musste zuerst mit *Get-WMIHelpLocation* ein kreativer »Übersetzungsbefehl« gebastelt werden, der WMI-Klassennamen in die zugehörige Internetadresse umwandelt:

```
function Get-WMIHelpLocation
{
    param ($WMIClassName)

    $uri = 'http://www.bing.com/search?q={0}+site:msdn.microsoft.com' -f $WMIClassName
    (Invoke-WebRequest -Uri $uri -UseBasicParsing).Links |
    Where-Object href -Like 'http://msdn.microsoft.com*' |
    Select-Object -ExpandProperty href -First 1
}
```

**ACHTUNG** Die Funktion *Get-WMIHelpLocation* nutzt das Cmdlet *Invoke-WebRequest*, um online nach Informationen zu suchen. Dieses Cmdlet ist neu in PowerShell 3.0, weswegen diese Lösung nicht in PowerShell 2.0 funktioniert.

Falls Sie keinen direkten Internetzugang zur Verfügung haben, sondern einen Proxyserver einsetzen oder sich anmelden müssen, fügen Sie die entsprechenden Informationen über die Parameter von *Invoke-WebRequest* hinzu.

*Get-WMIHelpLocation* sendet den Namen der gewünschten WMI-Klasse an eine Suchseite im Internet und wertet dann die zurückgegebenen Links aus. Der erste Link, der von MSDN stammt, wird zurückgeliefert:

```
PS> Get-WMIHelpLocation Win32_BIOS
http://msdn.microsoft.com/en-us/library/windows/desktop/aa394077(v=vs.85).aspx
```

```
PS> Get-WMIHelpLocation Win32_LogicalDisk
http://msdn.microsoft.com/en-us/library/windows/desktop/aa394173(v=vs.85).aspx
```

```
PS> Get-WMIHelpLocation Win32_Volume
http://msdn.microsoft.com/en-us/library/windows/desktop/aa394515(v=vs.85).aspx
```

```
PS> Get-WMIHelpLocation Win32_GibtsNix
PS>
```

Die zugehörige Erweiterung können Sie zum Abschluss dieses Kapitels vielleicht als so etwas wie eine kleine Herausforderung nehmen: Schauen Sie, ob Sie den Code verstehen. Falls nicht, dann machen Sie für heute Schluss und gehen das Kapitel morgen noch einmal genauer durch.

```
$code = {
    function Get-WMIHelpLocation
    {
        param ($WMIClassName)

        $uri = 'http://www.bing.com/search?q={0}+site:msdn.microsoft.com' -f $WMIClassName
        (Invoke-WebRequest -Uri $uri -UseBasicParsing).Links |
        Where-Object href -Like 'http://msdn.microsoft.com*' |
        Select-Object -ExpandProperty href -First 1
    }

    if (-not $args[0])
    {
        $types = $this | Get-Member | Select-Object -ExpandProperty TypeName | Sort-Object -Unique
    }
    else
    {
        $types = Get-Member -InputObject $this | Select-Object -ExpandProperty TypeName | Sort-Object
        -Unique
    }

    $types |
    ForEach-Object {
        if ($_ -like '*#root*')
        {
            $WMIClassName = $_.Split('#')[-1].Split('\')[-1]
            Write-Host $WMIClassName
            $url = Get-WMIHelpLocation -WMIClassName $WMIClassName
        }
    }
}
```

```

else {
    $template = 'http://msdn.microsoft.com/de-de/library/{0}(v=vs.80).aspx'
    $url = $template -f $_
}

try { Start-Process $url -ErrorAction SilentlyContinue } catch {}
}
}

```

```
Update-TypeData -MemberType ScriptMethod -MemberName GetHelp -Value $code -TypeName System.Object -Force
```

**Listing 11.10** Das Skript mit der verbesserten Onlinehilfe: *Get-Help2.ps1*

Oder Sie pfeifen einfach auf die Aufgabe, laden das Skript von den Begleitmaterialien und freuen sich über eine enorm praktische Befehlsenerweiterung:

```
# kann mit mehreren Ergebnissen umgehen:
```

```
PS> (Get-Process).GetHelp()
```

```
# öffnet mehrere Webseiten bei verschiedenen Objekttypen:
```

```
PS> (dir $env:windir).GetHelp()
```

```
# kommt mit WMI zurecht:
```

```
PS> $os = Get-WmiObject -Class Win32_OperatingSystem
```

```
PS> $os.GetHelp()
```

**Abbildung 11.10** *GetHelp()* findet die Onlinehilfe zu beliebigen WMI-Objekten

## Vorlagen für Typ-Erweiterungsdateien

Wie Sie in PowerShell 3.0 dynamisch neue Objektmember mit *Update-TypeData* für die aktuelle PowerShell-Sitzung anlegen, ist dank der neuen Parameter nicht mehr allzu schwierig. Wer hingegen Typ-Erweiterungen für PowerShell 2.0 herstellen möchte oder aus anderen Gründen lieber *.ps1xml*-Dateien generiert, findet in den nächsten Abschnitten die entsprechenden Vorlagen für die verschiedenen Erweiterungstypen.

### AliasProperty

```
<AliasProperty>
  <Name>NameDerEigenschaft</Name>
  <ReferencedMemberName>
    NameDerVorhandenenEigenschaft
  </ReferencedMemberName>
</AliasProperty>
```

### NoteProperty

```
<NoteProperty>
  <Name>NameDerEigenschaft</Name>
  <Value>
    Inhalt der Eigenschaft
  </Value>
</NoteProperty>
```

### ScriptProperty (nur lesbar):

```
<ScriptProperty>
  <Name>NameDerEigenschaft</Name>
  <GetScriptBlock>
    # wird beim Abruf der Eigenschaft aufgerufen
    # $this ist das Objekt, von dem die Eigenschaft abgerufen wird
  </GetScriptBlock>
</ScriptProperty>
```

### ScriptProperty (lesbar und schreibbar):

```
<ScriptProperty>
  <Name>NameDerEigenschaft</Name>
  <GetScriptBlock>
    # wird beim Abruf der Eigenschaft aufgerufen
    # $this ist das Objekt, von dem die Eigenschaft abgerufen wird
  </GetScriptBlock>
  <SetScriptBlock>
    # wird beim Ändern der Eigenschaft aufgerufen
    # $this ist das Objekt, von dem die Eigenschaft abgerufen wird
    # $args sind die Werte, die der Eigenschaft zugewiesen werden
    # $args[0] ist der erste Wert
  </SetScriptBlock>
</ScriptProperty>
```

## ScriptMethod

```
<ScriptMethod>
  <Name>NameDerMethode</Name>
  <Script>
    # wird beim Ändern der Eigenschaft aufgerufen
    # $this ist das Objekt, von dem die Eigenschaft abgerufen wird
    # $args sind die Werte, die der Eigenschaft zugewiesen werden
    # $args[0] ist der erste Wert
  </Script>
</ScriptMethod>
```

## CodeProperty

```
<CodeProperty>
  <Name>NameDerEigenschaft</Name>
  <GetCodeReference>
    <TypeName>
      VollständigerNameDesVorhandenenTyps
    </TypeName>
    <MethodName>
      NameDerStatischenMethode
    </MethodName>
  </GetCodeReference>
</CodeProperty>
```

## CodeMethod

```
<CodeMethod>
  <Name>NameDerMethode</Name>
  <CodeReference>
    <TypeName>
      VollständigerNameDesVorhandenenTyps
    </TypeName>
    <MethodName>
      NameDerStatischenMethode
    </MethodName>
  </CodeReference>
</CodeMethod>
```

## PropertySet

```
<Name>NameDesPropertySets</Name>
<ReferencedProperties>
  <Name>NameEinerVorhandenenEigenschaft</Name>
  <Name>NameEinerVorhandenenEigenschaft</Name>
  <Name>NameEinerVorhandenenEigenschaft</Name>
  (...)
</ReferencedProperties>
```

## MemberSet

```
<Name>NameDesMemberSets</Name>
<Members>
  [Erweiterung(en)]
</Members>
```

# Testen Sie Ihr Wissen!

Auch diesmal finden Sie in diesem Abschnitt eine Reihe von Aufgaben, mit denen Sie Ihr neues Wissen überprüfen können.

**Aufgabe** Der Typ eines Objekts lässt sich immer mit der Eigenschaft *PSTypeNames* ermitteln:

```
PS> "Hallo".PSTypeNames
System.String
System.Object
PS> "Hallo" | Get-Member PSType*
```

Sonderbarerweise zeigt *Get-Member* diese Eigenschaft aber gar nicht an. Wieso?

**Lösung** *Get-Member* zeigt nur die üblichen Objektmember an, mit denen man im Alltag konfrontiert wird, und versteckt einige andere, damit die Liste der Objektmember nicht ausufernd und unübersichtlich wird. Möchten Sie diese Sicherung ausschalten, verwenden Sie den Parameter *-Force*:

```
PS> "Hallo" | Get-Member PSType* -Force
```

```
TypeName: System.String
```

| Name        | MemberType   | Definition   |
|-------------|--------------|--|
| -----       | -----        | -----  |
| pstypenames | CodeProperty | System.Collections.ObjectModel.Collection`1[[System... |

Voilà! Schon wird die Eigenschaft *PSTypeNames* sichtbar. Es lohnt sich durchaus, nachzuforschen, welche Objektmember genau von *Get-Member* versteckt werden. Die folgenden Zeilen finden heraus, welche Eigenschaften und Methoden *Get-Member* bei Objekten vom Typ *DateTime* versteckt:

```
PS> $normal = Get-Date | Get-Member
PS> $alle = Get-Date | Get-Member -Force
PS> Compare-Object $normal $alle -Property Name -PassThru
```

```
TypeName: System.DateTime
```

| Name        | MemberType   | Definition   |
|-------------|--------------|--|
| -----       | -----        | -----  |
| pstypenames | CodeProperty | System.Collections.ObjectModel.Collection`1[[Sy... |
| psadapted   | MemberSet    | psadapted {Date, Day, DayOfWeek, DayOfYear, Hou... |
| psbase      | MemberSet    | psbase {Date, Day, DayOfWeek, DayOfYear, Hour, ... |
| psextended  | MemberSet    | psextended {DisplayHint, DateTime}                 |
| psoject     | MemberSet    | psoject {Members, Properties, Methods, Immedia...  |
| get_Date    | Method       | System.DateTime get_Date()                         |
| get_Day     | Method       | int get_Day()                                      |
| (...)       |              |  |

Die Eigenschaft *PSTypeNames* ist eine interne von PowerShell hinzugefügte Information, die man im normalen Alltag nicht benötigt. Sie zeigt, von welchen Typen das Objekt abgeleitet ist.

| MemberSet         | Beschreibung  |
|-------------------|---|
| <i>PSBase</i>     | Das Rohobjekt, bevor es vom PowerShell-ETS bearbeitet wurde. Häufig sind beide Versionen identisch, aber manchmal bietet nur das Rohobjekt Zugriff auf Low-Level-Eigenschaften und -Methoden. |
| <i>PSAdapted</i>  | Das vollständig vom ETS bearbeitete Objekt  |
| <i>PSExtended</i> | Nur die vom ETS hinzugefügten Elemente  |
| <i>PSObject</i>   | Der Adapter, der das Objekt repräsentiert   |

**Tabelle 11.2** Spezialansichten für Objekte

## Zusammenfassung

PowerShell erstellt keine eigenen Objekte, sondern nutzt (ausschließlich) die Objekte, die .NET Framework bietet. Das interne Extended Type System (ETS) kann diese Objekte jedoch nachträglich erweitern. Die mitgelieferten Erweiterungen stammen aus *.ps1xml*-Typdateien, die im Stammordner von PowerShell lagern. Sie legen fest, welcher Objekttyp mit welchen zusätzlichen Eigenschaften und Methoden ausgerüstet werden soll. Zusätzlich können aber auch weitere solcher Dateien mit *Update-TypeData* oder über ein nachgeladenes Modul aktiv werden. Damit haben auch Sie selbst die Möglichkeit, bestimmte Objekttypen mit neuen Eigenschaften und Methoden auszustatten. Alle diese Erweiterungen gelten jedoch immer nur für die aktuelle PowerShell-Sitzung. Erweiterungen, die ständig benötigt werden, sollten daher über ein Profilskript automatisch beim PowerShell-Start geladen werden.

Mit *Add-Member* kann man Objekterweiterungen auch »von Hand« einem konkreten einzelnen Objekt zuweisen. In PowerShell 3.0 wird auch *Update-TypeData* zu einem nützlichen Testwerkzeug, weil es nun erweiterte Objektmember auch ohne *.ps1xml*-Typdatei allein über seine Parameter erzeugen und einlesen kann.

PowerShell nutzt die erweiterten Objektmember auch für den internen Gebrauch und legt mit sogenannten *MemberSets* fest, wie PowerShell die Objekte behandeln soll. Mit einem *MemberSet* kann man beispielsweise die Standardeigenschaften eines Objekts oder seine Serialisierungstiefe festlegen.