Ed Wilson

Microsoft Windows PowerShell-Scripting – Die technische Referenz

Microsoft Press

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Dieses Buch ist die deutsche Übersetzung von:
Microsoft PowerShell Scripting Guide
Microsoft Press, Redmond, Washington 98052-6399

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
10 09 08

ISBN 978-3-86645-919-9

Übertragung ins Deutsche: biblioso Corporation, Las Vegas (www.biblioso.com)
Korrektorat: Jutta Alfes, Karin Baeyens, Siegen
Satz: Gerhard Alfes, mediaService, Siegen (www.media-service.tv)
Umschlaggestaltung: Hommer Design GmbH, Haar (www.HommerDesign.com)
Herstellung, Druck und Bindung: Kösel, Krugzell (www.KoeselBuch.de)

Inhaltsverzeichnis

Einleitung . . . . . **XI**
Wurde dieses Buch für Sie geschrieben? XI
Die Begleit-CD XII
Systemanforderungen XIII
Technischer Support XIII

Kapitel 1 Die Konsole von Windows PowerShell **1**
Installieren von Windows PowerShell 1
Überprüfen der Installation mit VBScript 1
Bereitstellen von Windows PowerShell 3
Arbeiten mit der Windows PowerShell-Konsole 4
Einführung in Cmdlets 6
Konfigurieren von Windows PowerShell 6
Erstellen eines Windows PowerShell-Profils 6
Konfigurieren von Windows PowerShell-Optionen 7
Sicherheitsprobleme mit Windows PowerShell 7
Steuern der Ausführung von Cmdlets 7
Bestätigen von Befehlen 9
Zurückstellen der Bestätigung von Cmdlets 10
Optionen für Cmdlets 11
Arbeiten mit Get-Help 12
Zuordnen von Tastenkombinationen zu Cmdlets mit Aliasen 14
Weitere Einsatzmöglichkeiten für Cmdlets 15
Das Cmdlet Get-ChildItem 16
Formatieren der Ausgabe 16
Das Cmdlet Get-Command 23
Das Cmdlet Get-Member 25
Zusammenfassung 28

Kapitel 2 Skripting mit Windows PowerShell **29**
Warum Skripts verwenden? 29
Konfigurieren der Skriptrichtlinie 32
Ausführen von Windows PowerShell-Skripts 34
Verwenden von Variablen 35
Verwenden von Konstanten 36
Verwenden von Flusskontrollanweisungen 37
Hinzufügen von Parametern zum Cmdlet ForEach-Object 37
Verwenden des Parameters Begin 37
Verwenden des Parameters Process 38
Verwenden des Parameters End 38

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Verwenden der Anweisung *For* 39
Verwenden von Anweisungen für die Entscheidungsfindung 40
Verwenden von *If … Elseif … Else* 40
Verwenden von *Switch* 41
Arbeiten mit Datentypen 44
Die Leistungsfähigkeit regulärer Ausdrücke 48
Verwenden von Befehlszeilenargumenten 51
Zusammenfassung 52

Kapitel 3 Verwalten von Protokollen 53

Identifizieren der Ereignisprotokolle 53
Lesen der Ereignisprotokolle 54
Exportieren in eine Textdatei 54
Exportieren in eine XML-Datei 56
Überprüfen allgemeiner Protokolldateien 58
Überprüfen mehrerer Protokolle 59
Abrufen eines Protokolleintrags 60
Durchsuchen des Ereignisprotokolls 62
Filtern von Eigenschaften 63
Auswählen der Quelle 63
Auswählen des Schweregrads 63
Auswählen der Meldung 64
Verwalten des Ereignisprotokolls 65
Identifizieren der Quellen 65
Ändern der Ereignisprotokolleinstellungen 65
Überprüfen der WMI-Ereignisprotokolle 68
Ändern der WMI-Protokollierungsebene 69
Das Befehlszeilenprogramm Wevtutil.exe 69
Schreiben in Ereignisprotokolle 70
Erstellen einer Quelle 70
Einfügen der Ausgabe in das Protokoll 71
Erstellen von Ereignisprotokollen 72
Zusammenfassung 73

Kapitel 4 Verwalten von Diensten 75

Dokumentieren der vorhandenen Dienstkonfiguration 75
Arbeiten mit ausgeführten Diensten 76
Schreiben einer Textdatei 77
Erfassen in einer Datenbank 80
Festlegen der Dienstkonfiguration 88
Befehlszeilenargumente 91
Beenden von Diensten 91
Ordnungsgemäßes Beenden 93
Starten von Diensten 95
Ordnungsgemäßes Starten 96

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Verwalten der Konfiguration 101
Überprüfen, ob die Dienste beendet wurden 102
Lesen einer Datei, um den Dienststatus zu überprüfen 102
Überprüfen, ob die Dienste ausgeführt werden 103
Bestätigen der Konfiguration 104
Erstellen eines Ausnahmeberichts 104
Zusammenfassung 106

Kapitel 5 Verwalten von Freigaben 107

Dokumentieren von Freigaben 107
Dokumentieren von Benutzerfreigaben 113
Erfassen von Freigaben in Textdateien 116
Dokumentieren administrativer Freigaben 117
Erfassen von Freigabeinformationen in einer Microsoft Access-Datenbank 118
Überwachen von Freigaben 121
Ändern von Freigaben 124
Angeben von Parametern im Skript 125
Umwandeln des Rückgabecodes 126
Erstellen neuer Freigaben 127
Erstellen mehrerer Freigaben 131
Entfernen von Freigaben 133
Entfernen nicht autorisierter Freigaben 135
Zusammenfassung 136

Kapitel 6 Verwalten von Druckern 137

Ermitteln der Druckeranzahl 137
Abfragen mehrerer Computer 138
Protokollieren in eine Datei 140
Erfassen in einer Microsoft Access-Datenbank 142
Überprüfen der Druckerports 147
Identifizieren der Druckertreiber 152
Installieren von Druckertreibern 154
Installieren von Druckertreibern, die auf dem Computer vorhanden sind 154
Installieren von Druckertreibern, die nicht auf dem Computer vorhanden sind 156
Zusammenfassung 158

Kapitel 7 Desktopverwaltung 159

Verwalten der Desktopcomputer 159
Überprüfen der Laufwerke 159
Erfassen der Datenträgerinformationen in Microsoft Access 162
Arbeiten mit Partitionen 166
Vergleichen von Datenträgern und Partitionen 168
Arbeiten mit logischen Datenträgern 171

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Überwachen der Speicherplatzbelegung 175
Protokollieren des Speicherplatzes in einer Datenbank 178
Überwachen gespeicherter Dateien 182
Überwachen der Leistung 185
Verwenden von Leistungsindikatorklassen 185
Ermitteln der Ursachen von Seitenfehlern 189
Zusammenfassung 189

Kapitel 8 Netzwerkverwaltung 191
Arbeiten mit Netzwerkeinstellungen 191
Überprüfen der Netzwerkeinstellungen 191
Arbeiten mit den Netzwerkeinstellungen 196
Filtern von initialisierten Eigenschaften 200
Konfigurieren der Netzwerkeinstellungen 205
Erkennen mehrerer Netzwerkkarten 205
Erfassen der Netzwerkinformationen in einem Microsoft Excel-Arbeitsblatt 206
Erkennen verbundener Netzwerkkarten 209
Festlegen einer statischen IP-Adresse 211
Aktivieren von DHCP 215
Konfigurieren der Windows-Firewall 219
Überprüfen der Firewalleinstellungen 220
Konfigurieren der Firewalleinstellungen 222
Zusammenfassung 223

Kapitel 9 Konfigurieren der Desktopeinstellungen 225
Desktopkonfiguration 225
Konfigurieren des Bildschirmschoners 225
Überprüfen des Bildschirmschoners 225
Auflisten von Eigenschaften mit Werten 232
Überprüfen der Sicherheitseinstellungen für Bildschirmschoner 236
Verwalten der Energieeinstellungen 242
Ändern des Energieschemas 247
Zusammenfassung 253

Kapitel 10 Behandeln von Problemen nach der Bereitstellung 255
Festlegen der Systemzeit 255
Festlegen der Systemzeit auf einem Remotecomputer 256
Aufzeichnen der Ergebnisse im Ereignisprotokoll 261
Konfigurieren der Zeitquelle 266
Der Befehl Net Time 266
Abfragen der Zeitquelle in der Registrierung 269
Aktivieren von Benutzerkonten 274
Erstellen eines lokalen Benutzerkontos 278
Erstellen eines lokalen Benutzers 279
Erstellen einer lokalen Benutzergruppe 281

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Konfigurieren des Bildschirmschoners 285
Umbenennen des Computers 291
Herunterfahren oder Neustarten eines Remotecomputers 294
Zusammenfassung 297

Kapitel 11 Verwalten von Benutzerdaten 299

Arbeiten mit Datensicherungen 299
Konfigurieren von Offline-Dateien 302
Aktivieren von Offline-Dateien 305
Arbeiten mit der Systemwiederherstellung 312
Abrufen der Einstellungen der Systemwiederherstellung 313
Auflisten der verfügbaren Systemwiederherstellungspunkte 316
Zusammenfassung 319

Kapitel 12 Behandeln von Windows-Problemen 321

Beheben von Startproblemen 321
Überprüfen der Startkonfiguration 321
Überprüfen der Startdienste 323
Anzeigen der Dienstabhängigkeiten 326
Überprüfen der Gerätetreiber 331
Überprüfen der Startprozesse 335
Überprüfen von Hardwareproblemen 339
Beheben von Netzwerkproblemen 343
Zusammenfassung 346

Kapitel 13 Verwalten von Domänenbenutzern 347

Erstellen von Organisationseinheiten 347
Erstellen von Domänenbenutzern 350
Ändern der Benutzerattribute 353
Ändern der allgemeinen Benutzerinformationen 353
Bearbeiten der Registerkarte Adresse 355
Bearbeiten der Registerkarte Profil 355
Bearbeiten der Registerkarte Telefon 356
Bearbeiten der Registerkarte Organisation 357
Ändern eines bestimmten Benutzerattributs 358
Erstellen von Benutzern mit einer .csv-Datei 360
Festlegen des Kennworts 361
Aktivieren des Benutzerkontos 361
Erstellen von Domänengruppen 362
Hinzufügen eines Benutzers zu einer Domänengruppe 365
Hinzufügen mehrerer Benutzer mit mehreren Attributen 367
Zusammenfassung 370

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Kapitel 14 Konfigurieren des Clusterdienstes **371**
Überprüfen von Clusterservern 371
Überprüfen der Clusterkonfiguration 377
Überprüfen der Knotenkonfiguration 381
Abfragen mehrerer Clusterklassen 385
Verwalten von Knoten 395
Hinzufügen und Entfernen von Knoten 395
Entfernen eines Clusterservers 401
Zusammenfassung 405

Kapitel 15 Verwalten der Internetinformationsdienste **407**
Aktivieren der IIS-Verwaltung 407
Überprüfen der IIS-Konfiguration 408
Überprüfen der Websiteinformationen 408
Überprüfen der Anwendungspools 411
Überprüfen der Standardwerte eines Anwendungspools 414
Überprüfen der Websiteeinschränkungen 417
Auflisten der virtuellen Verzeichnisse 420
Erstellen einer neuen Website 422
Erstellen eines neuen Anwendungspools 427
Starten und Beenden von Websites 429
Zusammenfassung 433

Kapitel 16 Arbeiten mit dem Zertifikatspeicher **435**
Auffinden bestimmter Zertifikate im Zertifikatspeicher 435
Auflisten von Zertfikaten 441
Ermitteln abgelaufener Zertifikate 444
Identifizieren von Zertifikaten, die in Kürze ablaufen 449
Verwalten von Zertifikaten 453
Überprüfen eines Zertifikats 453
Importieren eines Zertifikats 457
Löschen eines Zertifikats 461
Zusammenfassung 467

Kapitel 17 Verwalten der Terminaldienste **469**
Konfigurieren der Terminaldienst-Installation 469
Dokumentieren der Terminaldienstekonfiguration 469
Deaktivieren der Anmeldung 472
Ändern der Clienteigenschaften 476
Verwalten von Benutzern 481
Benutzern den Zugriff auf den Server gewähren 483
Konfigurieren der Clienteinstellungen 487
Zusammenfassung 497

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Kapitel 18 Konfigurieren der Netzwerkdienste **499**
Überprüfen der DNS-Einstellungen 499
Konfigurieren der DNS-Protokolleinstellungen 505
Überprüfen von Stammhinweisen 513
Abrufen von A-Einträgen 514
Konfigurieren der DNS-Servereinstellungen 519
Überprüfen von DNS-Zonen 524
Erstellen von DNS-Zonen 526
Verwalten von WINS und DHCP 531
Zusammenfassung 536

Kapitel 19 Arbeiten mit der Windows Server 2008 Server Core **537**
Ursprüngliche Konfiguration 537
Aufnehmen eines Computers in eine Domäne 538
Festlegen der IP-Adresse 545
Konfigurieren der DNS-Einstellungen 550
Umbenennen des Servers 558
Verwalten von Windows Server 2008 Server Core 563
Überwachen von Servern 564
Abfragen von Ereignisprotokollen 566
Zusammenfassung 568

Anhang A Cmdlet-Namenskonventionen **569**

Anhang B Anbieternamen für ActiveX-Datenobjekte **571**

Anhang C Häufig gestellte Fragen **573**

Anhang D Skriptrichtlinien **581**
Allgemeiner Skriptaufbau 581
Einbeziehen einer Funktion in das Skript, das die Funktion aufruft 581
Vollständige Cmdlet-Namen und Parameternamen 581
Verwenden Sie Get-Item, um Pfadzeichenfolgen zu konvertieren 582
Lesbarkeit von Skripts 583
Formatieren des Codes 584
Arbeiten mit Funktionen 585
Erstellen von Vorlagendateien 586
Erstellen von Funktionen 587
Erstellen und Benennen von Variablen und Konstanten 587

Anhang E Allgemeine Tipps zur Problembehandlung **589**

Stichwortverzeichnis **593**

Über den Autor **603**

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Einleitung

Die beste Skriptsprache der Welt ist nun in die besten Betriebssysteme der Welt integriert! Windows Vista und Windows Server 2008 sind nicht nur die wichtigsten Produkte in der Geschichte von Microsoft, sondern auch die anpassungsfähigsten Betriebssysteme. Die Neuerungen, die die Benutzeroberfläche für Benutzer so bequem machen, können jedoch ein Problem für Netzwerkadministratoren, Berater und fortgeschrittene Benutzer darstellen. Erfreulicherweise ist aber das Tool zum Verwalten von Exchange Server 2007, Virtual Server 2007 und Windows Server 2008 auch das Tool, das zum Verwalten von Windows Vista verwendet wird. Dieses Tool ist Windows PowerShell.

Als Autor von fünf Büchern über Windows-Scripting und als Consultant für Microsoft reise ich viel durch die Welt und vermittle Informationen bezüglich Visual Basic Script (VBScript), Windows Management Instrumentation (WMI), Active Directory Services Interfaces (ADSI) und nun auch Windows PowerShell.

Mit Windows PowerShell können selbst unerfahrene Netzwerkadministratoren Skripts erstellen, um die Prozesse anzuzeigen, die auf einem Computer die meisten Ressourcen belegen. Dazu ist nur eine einzige Codezeile erforderlich. Obwohl diese Aufgabe auch mit VBScript ausgeführt werden kann, ist die Arbeit mit VBScript wesentlich zeitaufwändiger. Sie können mit der gleichen Codezeile die Prozesse auf einem Windows Server 2008- oder auf einem Windows Vista-Computer ermitteln.

Neue Produkte von Microsoft werden mit Windows PowerShell-Cmdlets (Cmdlets werden in Kapitel 1 erklärt), Schnittstellen und manchmal sogar Tools ausgeliefert. Vielleicht werden wir in naher Zukunft tatsächlich nur noch mit einer Umgebung Anwendungen verwalten und konfigurieren.

Windows PowerShell ist eine neue Skriptsprache, die mit Microsoft Exchange 2007 eingeführt wurde und auf Windows XP, Windows Server 2003 und Windows Vista installiert werden kann. Windows PowerShell ist auch als installierbares Feature in Windows Server 2008 integriert und in der Standardinstallation der nächsten Generation von Desktopclients enthalten. Da die Microsoft Exchange 2007-Verwaltungstools auf Windows PowerShell basieren, zählen Exchange-Administratoren zu den ersten Benutzern von Windows PowerShell. Das Verwalten von Sicherheitseinstellungen, der Registrierung und der Dienstkonfiguration gehört zu den täglichen Aufgaben von Netzwerkadministratoren, die durch die Flexibilität von Windows PowerShell vereinfacht werden.

Wurde dieses Buch für Sie geschrieben?

Windows PowerShell Scripting – Die technische Referenz gibt Ihnen die Tools in die Hand, die Sie zum Automatisieren von Installations-, Bereitstellungs- und Verwaltungsaufgaben auf Windows-Computern benötigen. Außerdem werden in diesem Buch die Cmdlets von Windows PowerShell ausführlich beschrieben. Mehr als 300 Skripts veranschaulichen das Ausführen von Aufgaben, die in den Zuständigkeitsbereich von Netzwerkadministratoren fallen: Sicherheit, Konfiguration, Bereitstellung, Wartung und Problembehandlung.

Windows PowerShell Scripting – Die technische Referenz ist für folgende Zielgruppen ausgelegt:

- **Windows-Netzwerkberater** Zum Standardisieren und Automatisieren der Installation und Konfiguration von .NET-Netzwerkkomponenten.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

- **Windows-Netzwerkadministratoren** Zum Automatisieren der täglichen Verwaltungsaufgaben für Windows-Netzwerke.
- **Microsoft Certified Systems Engineers (MCSEs) und Microsoft Certified Trainers (MCTs)** Für die Vorbereitung auf die neuen Zertifizierungsprüfungen, die Fragen zu Windows PowerShell umfassen.
- **Technisches Personal** Zum Zusammenstellen von Informationen und Konfigurieren von Einstellungen auf Windows-Computern.
- **Hauptbenutzer** Für die maximale Leistungsfähigkeit und Konfigurierbarkeit von Windows-Computern in privaten oder nicht verwalteten Desktopumgebungen.

Windows PowerShell Scripting – Die technische Referenz ist in vier konzeptionelle Abschnitt aufgeteilt: Verstehen von Windows PowerShell, Verwenden von Windows PowerShell mit Windows Vista, Verwenden von Windows PowerShell mit Windows Server 2008 und Verwalten bestimmter Anwendungen. Das Buch ist nicht wirklich in diese Abschnitte aufgeteilt, da sich jedes Kapitel auf ein anderes Thema bezieht. Falls Sie Fragen haben, können Sie die entsprechenden Informationen im Buch nachschlagen. Sehen Sie beispielsweise in Kapitel 15 „Verwalten der Internetinformationsdienste“ nach, wenn Sie IIS 7 verwalten müssen.

Die Begleit-CD

Die Begleit-CD enthält weitere Informationen sowie Softwarekomponenten und zahlreiche Skripts. Es sind tatsächlich 317 Skripts verfügbar. (Das kann ich mit Gewissheit sagen, denn ich habe ein Skript geschrieben, um alle Skripts zu zählen.) Für jedes Kapitel sind Skripts und Beispielsausgaben vorhanden. Da die Ordnernamen mit den Kapitelnamen identisch sind, sollten Sie die gewünschten Skripts einfach finden können.

Die meisten Skripts sind in sich abgeschlossen und erfordern keine bestimmten Werte. Diese Skripts akzeptieren Befehlszeilenparameter, die Sie zur Laufzeit ändern können. Einige Skripts umfassen jedoch Variablen mit einem Beispielwert. Sie können diese Skripts an Ihre Anforderungen anpassen. Die erforderlichen Änderungen sind im Code, im Buch oder an beiden Stellen kommentiert.

Die CD enthält außerdem einige Datenbankdateien. Diese Dateien wurden mit Microsoft Access 2007 erstellt. Da Sie möglicherweise ein ältere Version von Access verwenden, wurden die Datenbankdateien im Kompatibilitätsmodus gespeichert. Die Screenshots, die auf die Datenbankdateien verweisen, wurden jedoch unter Verwendung von Access 2007 erstellt.

Wenn Sie das Skript-Installationsprogramm auf der Begleit-CD ausführen, werden die Beispielskripts standardmäßig im Ordner <*Eigene Dateien*>\Microsoft Press\PowerShell Scripting Guide\Scripts installiert. Sie können während der Installation jedoch ein anderes Verzeichnis angeben.

Achten Sie auf den Ordner *Extras*. Ich habe zahlreiche Skripts geschrieben, die sich nicht auf die Kapitel oder Themen in diesem Buch beziehen. Diese Skripts veranschaulichen Methoden, die Sie möglicherweise nützlich finden. Einige dieser Skripts, beispielsweise *FlashingBunny.ps1*, sind ziemlich nutzlos, aber möglicherweise finden Sie ein Skript, das Ihnen beim Beheben eines Problems helfen kann. (Wenn Sie beispielsweise von Ihrem Manager aufgefordert werden, ein Skript zu schreiben, das ein blinkendes Kaninchen anzeigt, haben Sie das Skript umgehend fertig.)

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Systemanforderungen

- Einen Intel Pentium/Celeron- oder AMD-Prozessor mit mindestens 1 GHz
- 1 GB RAM
- 1,5 GB freien Speicherplatz
- Einen Monitor mit einer Auflösung von mindestens 1024 × 768
- CD-ROM- oder DVD-Laufwerk
- Microsoft-Maus oder ein kompatibles Zeigegerät
- Windows Server 2003 SP1, Windows XP SP2 oder Windows Vista
- Microsoft .NET Framework 2.0

Dieses Buch wurde für Windows Vista und Windows Server 2008 geschrieben. Die Skripts wurden nicht auf Windows XP oder Windows Server 2003 getestet, obwohl sie in den meisten Fällen ohne Änderungen ausgeführt werden können.

Technischer Support

Microsoft Press bemüht sich stets um die Richtigkeit der in diesem Buch sowie der auf der Begleit-CD-ROM enthaltenen Informationen. Microsoft Press veröffentlicht Korrekturen zu Büchern unter *http://www.microsoft-press.de/support.asp.*

Unter *http://www.microsoft.com/learning/support/search.asp* können Sie direkt auf die Microsoft Press Knowledge Base zugreifen.

Anmerkungen, Fragen oder Verbesserungsvorschläge bezüglich dieses Buches oder der Begleit-CD können Sie folgendermaßen an Microsoft Press senden:

Per E-Mail:

presscd@microsoft.com

Per Post:

Microsoft Press
Betrifft: *Windows PowerShell Scripting – Die technische Referenz*
Konrad-Zuse-Straße 1
85716 Unterschleißheim

Beachten Sie bitte, dass unter den oben angegebenen Adressen kein Produktsupport geleistet wird.

Weitere Onlineinhalte Neues oder aktualisiertes Material wird auf der Microsoft Press-Website veröffentlicht. Dieses Material umfasst möglicherweise Aktualisierungen des Buches, Artikel, Links zu ergänzenden Inhalten, Berichtigungen, Beispielkapitel usw. Die Website, die regelmäßig aktualisiert wird, ist in Kürze unter *www.microsoft.com/learning/books/online/serverclient* verfügbar.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

KAPITEL 1

Die Konsole von Windows PowerShell

Nach Abschluss dieses Kapitels können Sie:

- Windows PowerShell installieren und konfigurieren
- Sicherheitsprobleme mit Windows PowerShell beheben
- Grundlagen von Cmdlets erläutern
- Häufig verwendeten Cmdlets einfachere Aliasnamen zuweisen
- Die Hilfe von Windows PowerShell verwenden

Auf der Begleit-CD Sie können die in diesem Kapitel verwendeten Skripts im Ordner *\Scripts\Chapter01* auf der Begleit-CD zu diesem Buch finden.

Installieren von Windows PowerShell

Da Windows PowerShell standardmäßig nicht auf Microsoft-Betriebssystemen installiert ist, müssen Sie vor dem Ausführen von Skripts oder Befehlen das Vorhandensein von PowerShell überprüfen. Führen Sie hierzu einfach einen Windows PowerShell-Befehl aus und überprüfen Sie, ob eine Fehlermeldung angezeigt wird. Am einfachsten lässt sich dies mit einer Batchdatei bewerkstelligen, wenn Sie in dieser den Wert %ERRORLEVEL% abfragen.

Überprüfen der Installation mit VBScript

Eine etwas weiterführende Methode zum Überprüfen der Installation von Windows PowerShell basiert auf dem Einsatz eines Skripts, um die WMI-Klasse (Windows Management Instrumentation) *Win32_QuickFixEngineering* abzufragen. Das Beispielskript *FindPowerShell.vbs* veranschaulicht den Einsatz von *Win32_QuickFixEngineering* in VBScript (Microsoft Visual Basic Scripting Edition) zum Überprüfen der Installation von Windows PowerShell.

Das Skript *FindPowerShell.vbs* verwendet den WMI-Moniker, um eine Instanz des Objekts *SwbemServices* zu erstellen. Dieses Objekt bietet eine Methode namens *ExecQuery* zum Ausführen von Abfragen. Die WQL-Abfrage (WMI Query Language) verwendet den Operator *like*, um Hotfixes anhand der Hotfix-ID zu ermitteln, beispielsweise 928439. 928439 ist die Hotfix-ID für Windows PowerShell auf Windows XP, Windows Vista und Windows Server 2003. Bei Windows Server 2008 wird Windows PowerShell jedoch als Feature und nicht mehr über einen Hotfix installiert. Sie brauchen

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

allerdings nur die WQL-Abfrage anzupassen und die Feature-ID 66 anstatt der Hotfix-ID 928439 abzufragen. Wird der Hotfix oder das Feature gefunden, zeigt das Skript eine entsprechende Mitteilung an, dass Windows PowerShell installiert ist (siehe Abbildung 1.1).

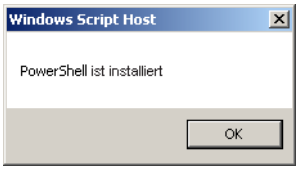


Abbildung 1.1 Das Skript *FindPowerShell.vbs* zeigt an, ob Windows PowerShell installiert ist

Wenn der Hotfix oder das Feature nicht gefunden wird, zeigt das Skript an, dass Windows PowerShell nicht installiert ist. Sie können das Skript *FindPowerShell.vbs* ändern, um weitere netzwerkspezifische Funktionen einzubeziehen. Um mit diesem Skript beispielsweise mehrere Computer zu überprüfen, ersetzen Sie *strComputer* mit einem Array der entsprechenden Computernamen. Sie können auch eine Textdatei zur Angabe der Computernamen verwenden oder diese mittels einer Active Directory-Abfrage bestimmen. Außerdem können Sie die Skriptausgabe in einer Datei protokollieren, statt für jeden Computer eine Mitteilung anzuzeigen.

FindPowerShell.vbs

```
Const RtnImmedFwdOnly = &h30
strComputer = "."
wmiNS = "\root\cimv2"
wmiQuery = "Select * from win32_QuickFixEngineering where hotfixid like '928439'"

Set objWMIService = GetObject("winmgmts:\\" & strComputer & wmiNS)
Set colItems = objWMIService.ExecQuery(wmiQuery,,RtnImmedFwdOnly)

For Each objItem in colItems
    Wscript.Echo "PowerShell ist installiert."
Wscript.quit
Next
wmiQuery = "SELECT * FROM Win32_ServerFeature WHERE ID = 66"
Set objWMIService = GetObject("winmgmts:\\" & strComputer & wmiNS)
Set colItems = objWMIService.ExecQuery(wmiQuery,,RtnImmedFwdOnly)
For Each objItem in colItems
    Wscript.Echo "PowerShell ist installiert."
Wscript.quit
Next
Wscript.Echo "PowerShell ist nicht installiert."
```

Bereitstellen von Windows PowerShell

Wenn Sie nicht Windows Server 2008 verwenden, müssen Sie Windows PowerShell von *http://www.microsoft.com/downloads* herunterladen, um diese Anwendung in Ihrer Umgebung bereitstellen zu können. Wie bereits erwähnt ist Windows PowerShell bei Windows Server 2008 als Feature im Lieferumfang enthalten. Zur Installation des Hotfixes für Windows PowerShell können Sie eine der folgenden Bereitstellungsmethoden verwenden.

- Sie können mit Microsoft Systems Management Server (SMS) ein Installationspaket erstellen und in der entsprechenden Organisationseinheit oder Sammlung zur Verfügung stellen.
- Sie können in Active Directory ein Gruppenrichtlinienobjekt erstellen und dieses mit der entsprechenden Organisationseinheit verknüpfen.
- Sie können die Installationsdatei für Windows PowerShell mit einem Anmeldeskript ausführen.

Wenn Sie Windows PowerShell nicht für das gesamte Unternehmen bereitstellen möchten, besteht die einfachste Installationsmethode darin, die ausführbare Datei manuell zu starten und den Installationsassistenten zu durchlaufen.

Beachten Sie, dass bei diesen Vorgehensweisen Windows PowerShell als Hotfix installiert wird. Das heißt, die PowerShell ist ein Update des Betriebssystems und kein zusätzliches Programm. Die Hotfixmethode hat mehrere Vorteile. Beispielsweise können Updates und Fixes für Windows PowerShell über Service Packs für das Betriebssystem und mit Hilfe des Windows Update-Dienstes eingespielt werden. Es gibt jedoch auch einige Nachteile, da Hotfixes in der gleichen Reihenfolge deinstalliert werden müssen, in der sie installiert wurden. Wenn Sie beispielsweise Windows PowerShell auf Windows Vista installiert haben und nun mehrere Updates und das Service Pack 1 in dieser Reihenfolge installieren, müssen Sie zuerst das Service Pack 1 und alle Updates in der umgekehrten Reihenfolge wieder entfernen, wenn Sie Windows PowerShell deinstallieren möchten. (Ich würde in diesem Fall meine Daten sichern, die Datenträger formatieren und Windows Vista neu installieren. Das wäre wahrscheinlich schneller. Die Frage erübrigt sich jedoch, da es so gut wie keinen Grund gibt, Windows PowerShell zu deinstallieren.)

Grundlagen zu Windows PowerShell

Natürlich ist es wichtig, Windows PowerShell in vollem Umfang zu verstehen. Allerdings ist dies leichter gesagt, als getan. Bei meiner ersten Begegnung mit Jeffrey Snover, Chefarchitekt von Windows PowerShell, stellte mir Jeffrey unter anderem die Frage: „Wie beschreiben Sie denn Ihren Lesern Windows PowerShell?“

Also *was ist* nun Windows PowerShell? Einfach gesagt, ist Windows PowerShell die Befehlsshell und Skriptsprache der nächsten Generation von Microsoft-Betriebssystemen, um den Befehlsinterpreter *Cmd.exe* und die Skriptsprache VBScript zu ersetzen.

Diese dualistische Charakteristik bereitet vielen Netzwerkadministratoren Verständnisschwierigkeiten, insbesondere wenn diese zum Automatisieren von Verwaltungsaufgaben an *Cmd.exe* mit seiner schwachen Batchsprache und die leistungsfähigere aber auch kompliziertere VBScript-Sprache gewöhnt sind. Batchdateien und VBScript sind zwar keine schlechte Wahl, werden aber derzeit für Aufgaben verwendet, für die diese vor mehr als einem Jahrzehnt einfach nicht entwickelt wurden. Der Befehlsinterpreter *Cmd.exe* ist der Nachfolger der DOS-Eingabeaufforderung und VBScript wurde ursprünglich mehr oder weniger in Hinsicht auf Webseiten entwickelt. Keines dieser Tools ist speziell für die Netzwerkverwaltung ausgelegt.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Arbeiten mit der Windows PowerShell-Konsole

Wenn Sie Windows PowerShell starten, können Sie die PowerShell-Konsole wie den Befehlsinterpreter *Cmd.exe* verwenden. Beispielsweise können Sie mit dem Befehl *dir* eine Verzeichnisliste abrufen. Wie in der *Cmd.exe*-Konsole können Sie mit dem Befehl *cd* das aktuelle Verzeichnis wechseln und anschließend mit *dir* eine Verzeichnisliste anzeigen. Die Ergebnisse dieser Befehle sind in folgendem Listing einer Beispieldatei namens *UsingPowerShell.txt* dargestellt.

UsingPowerShell.txt

```
PS C:\Users\edwils> dir


    Verzeichnis: Microsoft.PowerShell.Core\FileSystem::C:\Users\edwils


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d-r--        29.11.2006   13:32            Contacts
d-r--          2.4.2007   12:51            Desktop
d-r--          1.4.2007   18:53            Documents
d-r--        29.11.2006   13:32            Downloads
d-r--          2.4.2007   13:10            Favorites
d-r--          1.4.2007   18:53            Links
d-r--        29.11.2006   13:32            Music
d-r--        29.11.2006   13:32            Pictures
d-r--        29.11.2006   13:32            Saved Games
d-r--          1.4.2007   18:53            Searches
d-r--          2.4.2007   17:53            Videos


PS C:\Users\edwils> cd Music
PS C:\Users\edwils\Music> dir
```

Zusätzlich zu den herkömmlichen Befehlen können Sie aber auch einige neue Befehlszeilenprogramme verwenden, beispielsweise *Fsutil.exe*. Beachten Sie, dass für den Zugriff auf *Fsutil.exe* Administratorrechte erforderlich sind. Wenn Sie Windows PowerShell wie üblich über die Programmgruppe *Windows PowerShell 1.0* starten, arbeiten Sie jedoch unter Umständen nicht mit Administratorrechten und die in Abbildung 1.2 dargestellte Fehlermeldung wird angezeigt.

Abbildung 1.2 Da Windows PowerShell standardmäßig mit normalen Benutzerberechtigungen gestartet wird, werden beim Ausführen von Befehlen mit unzureichenden Berechtigungen Fehlermeldungen generiert

Fsutil.txt

```
PS C:\Users\edwils> sl c:\Test
PS C:\Test> fsutil file createNew c:\Test\meineNeueDatei.txt 1000
Die Datei c:\Test\meineNeueDatei.txt wurde erstellt.
PS C:\Test> dir


    Verzeichnis: Microsoft.PowerShell.Core\FileSystem::C:\Test


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---          8.5./2007   19:30        1000 meineNeueDatei.txt


PS C:\Test>
```

Tipp Ich empfehle, zwei Windows PowerShell-Verknüpfungen zu erstellen und in der Schnellstartleiste zu speichern. Eine Verknüpfung startet mit normalen Benutzerrechten und die andere mit Administratorrechten. Standardmäßig sollten Sie die Verknüpfung mit normalen Benutzerrechten verwenden und die Situationen dokumentieren, die Administratorrechte erfordern.

Nachdem Sie Ihre Arbeit beendet haben, können Sie die Datei mit dem Befehl **del** löschen. Sie können Platzhalter verwenden, beispielsweise ***.txt**, anstatt den vollständigen Dateinamen einzugeben. Es ist sicher angebracht, bei dieser Methode zuerst den Befehl **dir** auszuführen, um sicherzustellen, dass sich nur eine Textdatei im Ordner befindet. Nachdem die Datei gelöscht wurde, können Sie mit dem Befehl **rd** das Verzeichnis entfernen. Wie im folgenden Beispiel der Datei *DeleteFileAndFolder.txt* dargestellt, funktionieren diese Befehle wie in der Eingabeaufforderung.

DeleteFileAndFolder.txt

```
PS C:\> sl c:\Test
PS C:\Test> dir


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\Test


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---          5/8/2007   7:30 PM       1000 meineNeueDatei.txt


PS C:\Test> del *.txt
PS C:\Test> cd C:\
PS C:\> rd C:\Test
PS C:\> dir C:\Test
Get-ChildItem : Der Pfad "C:\Test" kann nicht gefunden werden, da er nicht vorhanden ist.
Bei Zeile:1 Zeichen:4
+ dir  <<<< C:\Test
PS C:\>
```

In diesen Beispielen wird Windows PowerShell auf interaktive Art verwendet. Dies ist einer der Hauptverwendungszwecke von Windows PowerShell. Das Windows PowerShell-Team geht davon aus, dass

80 Prozent der Benutzer interaktiv mit Windows PowerShell arbeitet, da dies eine bequeme Möglichkeit für die Befehlseingabe darstellt. Sie öffnen eine Windows PowerShell-Eingabeaufforderung und geben Befehle ein. Die Befehle können nacheinander oder gleichzeitig wie in einer Batchdatei eingegeben werden. Dieser Prozess wird später in diesem Buch beschrieben.

Einführung in Cmdlets

Zusätzlich zu herkömmlichen Programmen und Befehlen in der CMD-Shell, können Sie in Windows PowerShell integrierte Cmdlets verwenden. *Cmdlet* ist eine Bezeichnung, die das Windows PowerShell-Team für systemeigene Befehle eingeführt hat. Cmdlets sind ausführbaren Programmen ähnlich, aber einfacher zu programmieren, da diese die in Windows PowerShell integrierten Funktionen nutzen. Cmdlets sind keine Skripts, da diese nicht aus unkompiliertem Code bestehen. Stattdessen werden Cmdlets unter Verwendung eines speziellen Microsoft .NET Framework-Namespaces erstellt. Aufgrund der andersgearteten Beschaffenheit hat sich das Windows PowerShell-Team den neuen Begriff *Cmdlet* einfallen lassen.

Windows PowerShell umfasst mehr als 120 Cmdlets, mit denen Netzwerkadministratoren und Consultants die Vorteile von Windows PowerShell nutzen können, ohne die Skriptsprache von Windows PowerShell erlernen zu müssen. Diese Cmdlets sind in Anhang A „Cmdlet-Namenskonventionen“ beschrieben. Im Allgemeinen folgen die Cmdlets einer Standardnamenskonvention, beispielsweise **Get-Help**, **Get-EventLog** oder **Get-Process**. Die „get“-Cmdlets zeigen Informationen in Bezug auf das nach dem Bindestrich angegebene Element an. Die „set“-Cmdlets ändern oder legen Informationen für das nach dem Bindestrich angegebene Element fest. **Set-Service** ist beispielsweise ein „set“-Cmdlet, mit dem der Startmodus eines Diensts geändert werden kann. Eine Erklärung zu dieser Namenskonvention finden Sie in Anhang A „Cmdlet-Namenskonventionen“.

Konfigurieren von Windows PowerShell

Nachdem Windows PowerShell auf einer Plattform installiert wurde, müssen Sie einige Konfigurationsschritte ausführen. Dies ist zum Teil auf die angenommenen typischen Einsatzszenarien zurückzuführen. Da das Windows PowerShell-Team davon ausgeht, dass 80 Prozent der Windows PowerShell-Benutzer keine Skripts verwenden, wurden die Skriptfunktionen standardmäßig deaktiviert. Weitere Informationen zum Aktivieren der Skriptunterstützung in Windows Power Shell finden Sie in Kapitel 2 „Skripterstellung in der Windows PowerShell.“

Erstellen eines Windows PowerShell-Profils

In einem Windows PowerShell-Profil können zahlreiche Einstellungen gespeichert werden. Die Profileinstellungen werden in einer psconsole-Datei gespeichert. Diese Konfigurationsdatei kann mit dem Cmdlet **Export-Console** exportiert werden:

```
PS C:\> Export-Console meineKonsole
```

Die psconsole-Datei wird standardmäßig mit der Erweiterung .pscl im aktuellen Verzeichnis gespeichert. Die psconsole-Dateien verwenden folgendes XML-Format:

```
<?xml version="1.0" encoding="utf-8"?>
<PSConsoleFile ConsoleSchemaVersion="1.0">
  <PSVersion>1.0</PSVersion>
  <PSSnapIns />
</PSConsoleFile>
```

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Konfigurieren von Windows PowerShell-Optionen

Windows PowerShell kann auf mehrere Arten gestartet werden. Beispielsweise können Sie das Logo ausblenden, das angezeigt wird, wenn Sie auf das **Windows PowerShell**-Standardsymbol klicken. Sie können Windows PowerShell auch mit unterschiedlichen Profilen starten oder einen Windows PowerShell-Befehl ausführen und die Shell schließen. Um eine bestimmte Version vpm Windows PowerShell zu starten, geben Sie einen Wert für den Parameter **-version** an. Diese Optionen sind im Folgenden aufgeführt.

- Das Argument ***-nologo*** startet Windows PowerShell ohne das Logo:

  ```
  PowerShell -nologo
  ```

- Das Argument ***-version*** startet eine bestimmte Version von Windows PowerShell:

  ```
  PowerShell -version 1.0
  ```

- Das Argument ***-psconsolefile*** startet Windows PowerShell mit einer angegebenen Konfigurationsdatei:

  ```
  Power Shell -psconsolefile meineKonsole.psc1
  ```

- Das Argument ***-command*** beendet Windows PowerShell, nachdem der angegebene Befehl ausgeführt wurde. Der Befehl muss mit einem kaufmännisches Und-Zeichen beginnen und in geschweifte Klammern eingeschlossen werden:

  ```
  powershell -command "& {get-process}"
  ```

Sicherheitsprobleme mit Windows PowerShell

Wie mit jedem anderen vielseitigen Tool, bestehen auch bei Verwendung von Windows PowerShell einige Sicherheitsbedenken. Die Sicherheit war jedoch eines der Designziele bei der Entwicklung von Windows PowerShell.

Wenn Sie Windows PowerShell starten, wird das Tool im Ordner *Users\Benutzername* geöffnet, um sicherzustellen, dass Sie über die zum Ausführen bestimmter Aktionen und Aktivitäten erforderlichen Berechtigungen verfügen. Diese Methode ist bei Weitem sicherer, als das Arbeiten mit Windows PowerShell im Stammverzeichnis oder im Systemverzeichnis.

Um das Verzeichnis zu ändern, können Sie nicht einfach zur nächsten Ebene wechseln, sondern müssen das Ziel explizit angeben (Sie können jedoch die **Set-Location**-Cmdlets mit Punkten angeben, beispielsweise **Set-Location ...**).

Das Ausführen von Skripts ist standardmäßig deaktiviert. Sie können die Ausführung mit einer Gruppenrichtlinie oder einem Anmeldeskript jedoch aktivieren.

Steuern der Ausführung von Cmdlets

Haben Sie schon einmal eine CMD-Eingabeaufforderung geöffnet, einen Befehl eingegeben und die Eingabetaste gedrückt? Wenn der Befehl **Format C:** ist, werden Sie gefragt, ob Sie Laufwerk C wirklich formatieren möchten. Für Cmdlets sind mehrere Argumente verfügbar, die die Ausführung steuern. Diese Argumente werden in diesem Abschnitt beschrieben.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Tipp Die meisten Windows PowerShell-Cmdlets unterstützen einen „Versuchsmodus“, der mit dem Parameter **-whatif** aktiviert wird. Der Parameter **-whatif** kann vom Entwickler des Cmdlets optional implementiert werden. Das Windows PowerShell-Team empfiehlt jedoch, dass Entwickler den Parameter **-whatif** unterstützen, wenn das Cmdlet Änderungen am System vornimmt.

Diese Argumente werden von den meisten, aber nicht von allen, in Windows PowerShell integrierten Cmdlets unterstützt. Die drei Methoden zum Steuern der Ausführung sind **-whatif**, **-confirm** und **-suspend**. Das Argument **-suspend** wird nicht an das Cmdlet übergeben. Es ist eine Aktion, die Sie als Bestätigung ausführen können und somit eine weitere Methode zum Steuern der Ausführung.

Um den Parameter **-whatif** zu verwenden, geben Sie zuerst das Cmdlet in der Windows PowerShell-Eingabeaufforderung an. Geben Sie dann den Parameter **-whatif** nach dem Cmdlet ein. Die Verwendung des Arguments **-whatif** ist im folgenden Beispiel anhand der Datei *WhatIf.txt* dargestellt. Starten Sie zuerst Notepad, indem Sie **notepad** eingeben. Suchen Sie mit dem Cmdlet **Get-Process** nach den Prozessen, die mit *note* beginnen. In diesem Beispiel beginnen zwei Prozesse mit *notepad*. Führen Sie das Cmdlet **Stop-Process** aus, um einem Prozess mit dem Namen *notepad* zu beenden. Da das Ergebnis unbekannt ist, verwenden Sie den Parameter **-whatif**. Dieser Parameter zeigt an, dass zwei Prozesse mit dem Namen *notepad* abgebrochen werden. Außerdem wird die Prozess-ID angezeigt, mittels der Sie überprüfen können, welcher Prozess abgebrochen wird. Führen Sie das Cmdlet **Stop-Process** erneut aus, um alle Prozesse zu beenden, die mit *n* beginnen. Überprüfen Sie mit dem Parameter **-whatif**, welchen Vorgang der Befehl ausführt.

WhatIf.txt

```
PS C:\Users\edwils> notepad
PS C:\Users\edwils> Get-Process note*

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     45       2     1044       3904    53     0.03   3052 notepad
     45       2     1136       4020    54     0.05   3140 notepad


PS C:\Users\edwils> Stop-Process -processName notepad -WhatIf
WhatIf: Ausführen des Vorgangs "Stop-Process" für das Ziel "notepad (3052)".
WhatIf: Ausführen des Vorgangs "Stop-Process" für das Ziel "notepad (3140)".

PS C:\Users\edwils> Stop-Process -processName n* -WhatIf
WhatIf: Ausführen des Vorgangs "Stop-Process" für das Ziel "notepad (3052)".
WhatIf: Ausführen des Vorgangs "Stop-Process" für das Ziel "notepad (3140)".
```

Was passiert, wenn der Parameter **-whatif** nicht implementiert ist? Im folgenden Beispiel des Skripts *WhatIf2.txt* wird der Parameter **-whatif** implementiert und mit diesem bestätigt, dass das Cmdlet **New-Item** das Verzeichnis *meinNeuerTest* im Stammverzeichnis erstellt.

Beachten Sie, was passiert, wenn Sie den Parameter **-whatif** mit dem Cmdlet **Get-Help** eingeben. Normalerweise würden Sie eine Meldung, wie beispielsweise *„Whatif: Die Hilfeinformationen für das Cmdlet Get-Process werden abgerufen,“* erwarten. Aber wozu wäre das gut? Da das Cmdlet **Get-Help** keine Gefahr darstellt, muss **-whatif** für **Get-Help** nicht implementiert werden.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

WhatIf2.txt

```
PS C:\Users\edwils> New-Item -Name meinNeuerTest -Path c:\ -ItemType directory -WhatIf
WhatIf: Ausführen des Vorgangs "Verzeichnis erstellen" für das Ziel
"Ziel: C:\meinNeuerTest".

PS C:\Users\edwils> get-help Get-Process -whatif
Get-Help : Es wurde kein Parameter gefunden, der dem Parameternamen "whatif" entspricht.
Bei Zeile:1 Zeichen:29
+ get-help Get-Process -whatif  <<<<
```

Bewährte Vorgehensweise Der Parameter **-whatif** ist ein unentbehrlicher Parameter für den Netzwerkadministrator, der unter Umständen viel Arbeit ersparen kann.

Bestätigen von Befehlen

Wie im vorherigen Abschnitt erklärt, können Sie mit **-whatif** Cmdlets in Windows PowerShell testen. Dies ist nützlich, um vorab zu überprüfen, welchen Vorgang der jeweilige Befehl ausführt. Um einen Befehl vor dessen Ausführung zu bestätigen, verwenden Sie jedoch den Parameter **-confirm**. Der Parameter **-confirm** kann **-whatif** ersetzen, da ein Vorgang zuerst bestätigt werden muss. Dies ist in folgendem Beispiel der Datei *ConfirmIt.txt* dargestellt.

Starten Sie in der Datei *ConfirmIt.txt* zuerst den Rechner (*Calc.exe*). Da die Datei im Pfad angegeben ist, müssen Sie weder den Pfad noch die Erweiterung angeben. Führen Sie das Cmdlet **Get-Process** mit dem Platzhalter *c** aus, um nach den Prozessen zu suchen, die mit *c* beginnen. Beachten Sie, dass mehrere Prozesse aufgelistet werden. Rufen Sie als Nächstes den Prozess *Calc.exe* ab. Dieser Schritt liefert ein übersichtlicheres Ergebnis zurück. Führen Sie anschließend das Cmdlet **Stop-Process** mit dem Parameter **-confirm** aus. Das Cmdlet gibt folgende Informationen zurück:

```
Bestätigung
Möchten Sie diese Aktion wirklich ausführen?
Ausführen des Vorgangs "Stop-Process" für das Ziel "calc (2924)".
[J] Ja  [A] Ja, alle  [N] Nein  [K] Nein, keine  [H] Anhalten  [?] Hilfe
(Der Standardwert ist "J"):
```

Diese Informationen sind im Wesentlichen mit den Informationen identisch, die der Parameter **-whatif** anzeigt. Sie haben jedoch die Möglichkeit den gewünschten Vorgang auszuführen. Dies kann beim Ausführen mehrerer Befehle Zeit sparen.

ConfirmIt.txt

```
PS C:\Users\edwils> calc
PS C:\Users\edwils> Get-Process c*

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     43       2     1060       4212    54     0.03   2924 calc
   1408       7     3364       6556    81            372 casha
   1132      16    23156      34680   129           3084 CcmExec
    599       5     1680       4956    88            620 csrss
    480      10    15812      20500   195            688 csrss


PS C:\Users\edwils> Get-Process calc

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
```

```
-------  ------    -----      ----- -----   ------     -- -----------
     43       2     1060       4212    54     0.03   2924 calc


PS C:\Users\edwils> Stop-Process -Name calc -Confirm

Bestätigung
Möchten Sie diese Aktion wirklich ausführen?
Ausführen des Vorgangs "Stop-Process" für das Ziel "calc (2924)".
[J] Ja  [A] Ja, alle  [N] Nein  [K] Nein, keine  [H] Anhalten  [?] Hilfe
(Der Standardwert ist "J"): y
PS C:\Users\edwils> Get-Process c*

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
   1412       7     3364       6556    81             372 casha
   1154      16    23224      34740   130            3084 CcmExec
    598       5     1680       4956    88             620 csrss
    477      10    15812      20488   195             688 csrss
```

Zurückstellen der Bestätigung von Cmdlets

Das Bestätigen der Ausführung von Cmdlets ist ausgesprochen nützlich und kann für eine hohe Systembetriebszeit ausschlaggebend sein. Wenn Sie beispielsweise einen langen Befehl eingegeben haben, aber zuerst eine andere Prozedur ausführen müssen, können Sie die Ausführung des Befehls auch verzögern. Die Befehle für die verzögerte Ausführung eines Cmdlets und die entsprechende Ausgabe sind im folgenden Beispiel der Datei *SuspendConfirmation.txt* dargestellt.

Starten Sie wie in der Datei *SuspendConfirmation.txt* veranschaulicht zuerst Microsoft Paint (Mspaint.exe). Da Mspaint.exe bereits im Pfad angegeben ist, müssen Sie weder den Pfad noch die Erweiterung angeben. Rufen Sie anschließend die Prozessinformationen mit dem Cmdlet **Get-Process** ab. Verwenden Sie den Platzhalter *ms**, um alle Prozesse anzuzeigen, die mit *ms* beginnen. Nachdem Sie den richtigen Prozess identifiziert haben, führen Sie das Cmdlet **Stop-Process** mit dem Parameter **-confirm** aus. Anstatt nun aber mit *Ja* zu bestätigen, verzögern Sie die Ausführung des Befehls, um einen weiteren Befehl auszuführen (wenn Sie beispielsweise die Prozess-ID vergessen haben). Nachdem der zusätzliche Befehl ausgeführt wurde, geben Sie **exit** ein, um zum angehaltenen Befehl zurückzukehren. Führen Sie anschließend das Cmdlet **Get-Process** erneut aus, um den Abbruch des Prozesses **mspaint** zu bestätigen.

SuspendConfirmation.txt

```
PS C:\Users\edwils> mspaint
PS C:\Users\edwils> Get-Process ms*

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     98       4     5404      10492    72     0.09   3064 mspaint


PS C:\Users\edwils> Stop-Process -id 3064 -Confirm

Bestätigung
Möchten Sie diese Aktion wirklich ausführen?
Ausführen des Vorgangs "Stop-Process" für das Ziel "mspaint (3064)".
```

```
[J] Ja  [A] Ja, alle  [N] Nein  [K] Nein, keine  [H] Anhalten  [?] Hilfe
(Der Standardwert ist "J"): h
PS C:\Users\edwils>>> Get-Process ms*

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     97       4     5404      10496    72     0.09     3064 mspaint


PS C:\Users\edwils>>> exit

Bestätigung
Möchten Sie diese Aktion wirklich ausführen?
Ausführen des Vorgangs "Stop-Process" für das Ziel "mspaint (3064)".
[J] Ja  [A] Ja, alle  [N] Nein  [K] Nein, keine  [H] Anhalten  [?] Hilfe
(Der Standardwert ist "J"): j
PS C:\Users\edwils> Get-Process ms*
```

Optionen für Cmdlets

Wie in den vorherigen Abschnitten erklärt, können Sie mit **-whatif** und **-confirm** die Ausführung von Cmdlets steuern. Eine Frage, die mir häufig gestellt wird, ist: „Wie weiß ich, welche Optionen verfügbar sind?“ Die Antwort ist, dass das Windows PowerShell-Team mehrere Standardoptionen entwickelt hat. Diese Standardoptionen werden als *allgemeine Parameter* bezeichnet. Die Syntaxbeschreibung für ein Cmdlet gibt oft an, dass das Cmdlet allgemeine Parameter unterstützt. Dies ist in folgendem Beispiel anhand des Cmdlets **Get-Process** dargestellt:

```
SYNTAX
    Get-Process [[-name] <string[]>] [<AllgemeineParameter>]

    Get-Process -id <Int32[]> [<AllgemeineParameter>]

    Get-Process -inputObject <Prozess[]> [<AllgemeineParameter>]
```

Eines der hilfreichen Features von Windows PowerShell ist die Standardisierung der Cmdlet-Syntax. Dies vereinfacht die Arbeit mit der Shell und das Erlernen der neuen Sprache wesentlich. In Tabelle 1.1 sind die allgemeinen Parameter aufgeführt. Beachten Sie, dass nicht alle Cmdlets diese Parameter unterstützen. Die implementierten Parameter werden jedoch bei allen Cmdlets auf die gleiche Weise verwendet, da das Windows PowerShell-Modul die Parameter interpretiert.

Tabelle 1.1 Allgemeine Parameter

Parameter	Bedeutung
-whatif	Führt den Befehl nicht aus, sondern teilt Ihnen mit, welchen Vorgang das Cmdlet ausführen würde
-confirm	Fordert eine Bestätigung an, bevor der Befehl ausgeführt wird
-verbose	Zeigt umfassendere Informationen an, als ohne den Parameter **-verbose**
-debug	Zeigt Debuginformationen an
-erroraction	Führt eine bestimmte Aktion aus, wenn ein Fehler auftritt. Die zulässigen Aktionen sind **Continue**, **Stop**, **SilentlyContinue** und **Inquire.**
-errorvariable	Verwendet zusätzlich zur Variablen *$error* eine bestimmte Variable für Fehlerinformationen

Tabelle 1.1 Allgemeine Parameter *(Fortsetzung)*

Parameter	Bedeutung
-outvariable	Verwendet eine bestimmte Variable für die Ausgabeinformationen
-outbuffer	Speichert eine bestimmte Anzahl an Objekten, bevor das nächste Cmdlet in der Sequenz aufgerufen wird

Arbeiten mit Get-Help

Windows PowerShell ist intuitiv und die Onlinehilfe macht die Verwendung des Programms sogar noch einfacher. Sie können auf mehrere Arten auf das Hilfesystem von Windows PowerShell zugreifen. Führen Sie das Cmdlet **Get-Help** aus, um Informationen zur Windows PowerShell anzuzeigen:

```
get-help get-help
```

Dieser Befehl gibt die Hilfeinformationen zum Cmdlet **Get-Help** aus. Im Folgenden ist die Ausgabe des Cmdlets dargestellt:

```
NAME
    Get-Help

ÜBERSICHT
    Zeigt Informationen zu Windows PowerShell-Cmdlets und -Konzepten an.


SYNTAX
    Get-Help [[-name] <string>] [-component <string[]>] [-functionality <string
    []>] [-role <string[]>] [-category <string[]>] [-full] [<CommonParameters>]

    Get-Help [[-name] <string>] [-component <string[]>] [-functionality <string
    []>] [-role <string[]>] [-category <string[]>] [-detailed] [<CommonParameters>]

    Get-Help [[-name] <string>] [-component <string[]>] [-functionality <string
    []>] [-role <string[]>] [-category <string[]>] [-examples] [<CommonParameters>]

    Get-Help [[-name] <string>] [-component <string[]>] [-functionality <string
    []>] [-role <string[]>] [-category <string[]>] [-parameter <string>] [<CommonParameters>]


DETAILLIERTE BESCHREIBUNG
Mit dem Cmdlet "Get-Help" werden Informationen zu Windows PowerShell-Cmdlets und -Konzepten angezeigt. Sie können auch
"Help {<Cmdlet-Name> | <Thema>" oder "<Cmdlet-Name> /?" verwenden. Mit "Help" wird pro Seite jeweils ein Hilfethema
angezeigt. Mit "/?" wird die Hilfe zu Cmdlets auf einer einzigen Seite angezeigt.


VERWANDTE LINKS
    Get-Command
    Get-PSDrive
    Get-Member

HINWEISE
    Weitere Informationen erhalten Sie mit folgendem Befehl: "get-help Get-Help -detailed".
    Technische Informationen erhalten Sie mit folgendem Befehl: "get-help Get-Help -full".
```

Die Onlinehilfe für Windows PowerShell zeigt nicht nur wie erwartet die Hilfe zu einem Befehl an, sondern umfasst auch drei verschiedene Ansichten: *normal*, *detailliert* und *vollständig*. Außerdem ist

die Hilfe auch zu Windows PowerShell-Konzepten verfügbar. Dieses Feature entspricht einem Onlinehandbuch. Führen Sie den Befehl **Get-Help about*** wie in folgendem Beispiel aus, um eine Liste der konzeptionellen Hilfeartikel anzuzeigen:

```
get-help about*
```

Wenn Sie beispielsweise den genauen Namen eines Cmdlets vergessen haben, aber wissen, dass es sich um ein "get"-Cmdlet handelt, können Sie einen Platzhalter verwenden (beispielsweise *), um den Namen des Cmdlets zu ermitteln. Beispiel:

```
get-help get*
```

Die Methode mit einem Platzhalter kann noch weiter ausgedehnt werden. Wenn Sie sich erinnern, dass das Cmdlet ein "get"-Cmdlet ist und mit dem Buchstaben *p* beginnt, können Sie das Cmdlet mit folgendem Befehl ermitteln:

```
get-help get-p*
```

Wenn Ihnen der genaue Name des Cmdlets bekannt ist, Sie jedoch die Syntax vergessen haben, können Sie das Argument **-examples** verwenden. Um mehrere Beispiele des Cmdlets **Get-PSDrive** anzuzeigen, führen Sie **Get-Help** mit dem Argument **-examples** aus:

```
get-help get-psdrive -examples
```

Um die Hilfe seitenweise anzuzeigen, können Sie die Hilfefunktion verwenden, die den Hilfetext über die Funktion *more* abruft. Auf diese Weise müssen Sie den Bildlauf nicht ausführen, um die Hilfe anzuzeigen. Beispiel:

```
get-help get-help | more
```

Die formatierte Ausgabe der Funktion *more* ist in Abbildung 1.3 dargestellt.

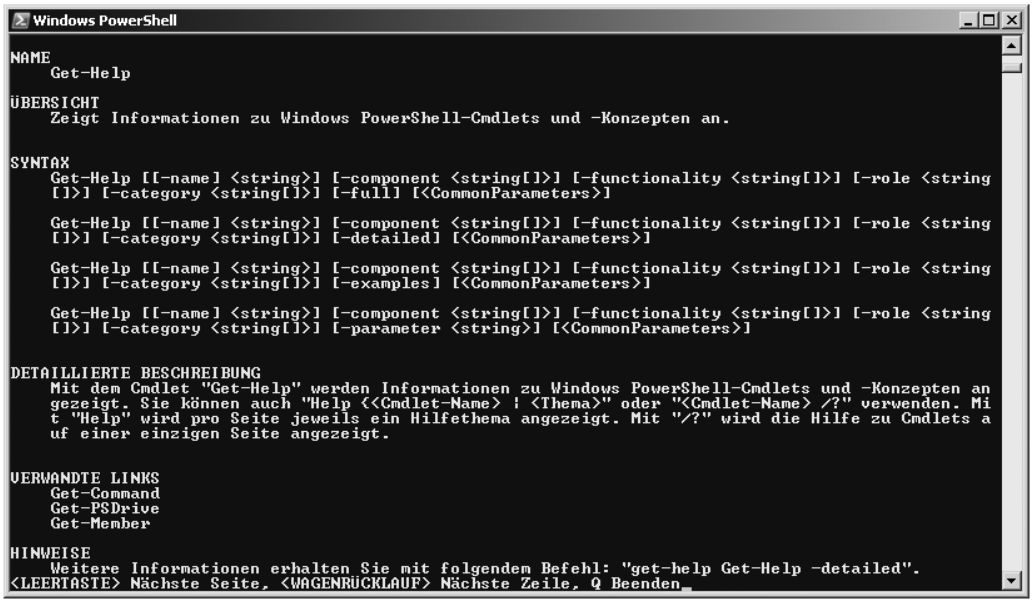


Abbildung 1.3 Die Funktion *more* zeigt langen Hilfetext seitenweise an

Um detaillierte Hilfeinformationen zum Cmdlet **Get-Help** anzuzeigen, geben Sie das Argument **-detailed** an:

```
get-help get-help -detailed
```

Um technische Informationen zum Cmdlet **Get-Help** anzuzeigen, geben Sie das Argument **-full** an:

```
get-help get-help -full
```

Möchten Sie **Get-Help** nicht immer wieder eingeben? Der Befehl ist einschließlich dem Bindestrich immerhin acht Zeichen lang. Die Lösung ist, einen Alias für das Cmdlet **Get-Help** zu erstellen. Ein Alias ist eine Tastenkombination, die ein Programm oder ein Cmdlet startet. Beispielsweise können Sie dem Cmdlet **Get-Help** die Tastenkombination *gh* zuordnen.

Tipp Bevor Sie einen Alias für ein Cmdlet erstellen, stellen Sie mit **Get-Alias** sicher, dass für das Cmdlet nicht bereits ein Alias vorhanden ist. Ordnen Sie anschließend mit **Set-Alias** dem Cmdlet eine eindeutige Tastenkombination zu.

Zuordnen von Tastenkombinationen zu Cmdlets mit Aliasen

Mit Aliasen können Sie Cmdlets Tastenkombinationen zuordnen. Die Anpassung der Befehlssyntax vereinfacht die Arbeit mit der Windows PowerShell-Eingabeaufforderung wesentlich. Beispielsweise können Sie einen Alias für das Cmdlet **Get-Help** erstellen, um anstatt **Get-Help** nur ***gh*** einzugeben. Sie können einen Alias in vier einfachen Schritten erstellen. Stellen Sie zuerst sicher, dass Sie der gewünschten Tastenkombination nicht bereits einen Alias zugeordnet haben. Lesen Sie als Nächstes die Hilfe für das Cmdlet **Set-Alias**. Rufen Sie anschließend das Cmdlet **Set-Alias** auf und geben Sie den neuen Namen und den Namen des Cmdlets ein, für das Sie den Alias erstellen möchten. Führen Sie das Cmdlet **Get-Alias** aus, um zu überprüfen, ob der Alias korrekt erstellt wurde. Den vollständigen Code finden Sie in der Datei *GhAlias.txt* auf der Begleit-CD.

1. Rufen Sie eine alphabetische Liste der definierten Aliase ab und überprüfen Sie, ob dem Cmdlet **Get-Help** oder der Tastenkombination ***gh*** ein Alias zugeordnet ist. Führen Sie folgenden Befehl aus:

   ```
   get-alias |sort
   ```

2. Nachdem Sie sichergestellt haben, dass dem Cmdlet **Get-Help** und der Tastenkombination ***gh*** kein Alias zugeordnet ist, überprüfen Sie die Syntax für das Cmdlet **Set-Alias**. Führen Sie das Cmdlet **Get-Help** mit dem Argument **-full** aus:

   ```
   get-help set-alias -full
   ```

3. Ordnen Sie dem Cmdlet **Get-Help** mit dem Cmdlet **Set-Alias** die Tastenkombination ***gh*** zu. Führen Sie hierzu folgenden Befehl aus:

   ```
   set-alias gh get-help
   ```

4. Um zu überprüfen, ob der Alias korrekt erstellt wurde, verwenden Sie das Cmdlet **Get-Alias**. Führen Sie hierzu folgenden Befehl aus:

   ```
   Get-Alias gh
   ```

Tipp Wenn die Syntax für Set-Alias zu verwirrend ist, können Sie benannte Parameter verwenden. Ich empfehle die Verwendung der Parameter **-whatif** oder **-confirm**. Sie können auch eine Beschreibung für den Alias eingeben. Die geänderte Syntax sieht wie folgt aus:

```
Set-Alias -Name gh -Value Get-Help -Description "mred help alias" -WhatIf
```

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Wie bereits erwähnt, kann Windows PowerShell als Ersatz für den CMD-Befehlsinterpreter verwendet werden. Windows PowerShell umfasst jedoch zahlreiche integrierte Cmdlets, mit der Sie eine Unmenge an Aktivitäten ausführen können. Diese Cmdlets können einzeln oder in Verbindung mit anderen Cmdlets ausgeführt werden.

Zugreifen auf Windows PowerShell

Nachdem Windows PowerShell installiert wurde, kann diese umgehend verwendet werden. Das Drücken des Buchstabens R zusammen mit der Windows-Taste oder das Klicken der linken Maustaste, um das Windows-Dialogfeld *Ausführen* zu öffnen und PowerShell einzugeben, wird mit der Zeit jedoch etwas langweilig. Ich habe deshalb eine Verknüpfung für Windows PowerShell auf meinem Desktop erstellt. Für meine Arbeitsweise ist das ideal. Es ist tatsächlich so hilfreich, dass ich ein Skript geschrieben habe, das diese Funktion ausführt. Das Skript kann über ein Anmeldeskript gestartet werden, um die Verknüpfung automatisch auf dem Desktop zu erstellen. Das Skript hat den Namen *CreateShortCutToPowerShell.vbs*:

CreateShortCutToPowerShell.vbs

```
Option Explicit
Dim objshell
Dim strDesktop
Dim objshortcut
Dim strProg
strProg = "powershell.exe"

Set objshell=CreateObject("WScript.Shell")
strDesktop = objshell.SpecialFolders("desktop")
set objShortcut = objshell.CreateShortcut(strDesktop & "\powershell.lnk")
objshortcut.TargetPath = strProg
objshortcut.WindowStyle = 1
objshortcut.Description = funfix(strProg)
objshortcut.WorkingDirectory = "C:\"
objshortcut.IconLocation= strProg
objshortcut.Hotkey = "CTRL+SHIFT+P"
objshortcut.Save

Function funfix(strin)
funfix = InStrRev(strin,".")
funfix = Mid(strin,1,funfix)
End function
```

Weitere Einsatzmöglichkeiten für Cmdlets

Nachdem Sie nun mit den Hilfetools und Aliasen vertraut sind, ist es an der Zeit, einige weitere Verwendungsmöglichkeiten der Cmdlets von Windows PowerShell zu betrachten.

Tipp Um bei der Eingabe eines Cmdlets Zeit zu sparen, geben Sie einen Teil des jeweiligen Namens ein und drücken Sie die Tab-Taste. Die Tab-Taste vervollständigt den Namen des Cmdlets. Dieses Methode funktioniert auch mit Argumentnamen und anderen Prozeduren. Probieren Sie diese zeitsparende Methode aus. Möglicherweise müssen Sie **get-command** nie wieder eingeben!

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Cmdlets geben Objekte anstatt Zeichenfolgenwerte zurück. Sie können weitere Informationen zu den zurückgegebenen Objekten abrufen. Diese Informationen sind bei der Arbeit mit Zeichenfolgen nicht verfügbar. Um die Informationen anzuzeigen, verwenden Sie den senkrechten Strich (|) und fügen Sie die Informationen eines Cmdlets in ein anderes Cmdlet ein. Obwohl dieser Vorgang möglicherweise kompliziert erscheint, ist er tatsächlich ziemlich einfach. Nach Abschluss dieses Kapitels sollten Sie mit diesem Verfahren vertraut sein.

Das einfachste Beispiel ist das Anzeigen und Formatieren einer Verzeichnisliste. Nachdem Sie die Verzeichnisliste abgerufen haben, können Sie die Ausgabe beispielsweise als Tabelle oder Liste formatieren. Dieser Vorgang besteht aus zwei Aktionen: Dem Abrufen der Liste und dem Formatieren der Liste. Die Formatierung wird nach dem Abrufen der Verzeichnisliste rechts vom senkrechten Strich angegeben. Dieses Verfahren wird in folgendem Abschnitt mit dem Cmdlet **Get-ChildItem** veranschaulicht.

Das Cmdlet Get-ChildItem

In einem vorherigen Abschnitt haben Sie mit dem Befehl **dir** eine Liste der Dateien in einem Verzeichnis angezeigt. Das funktioniert, da die Windows PowerShell einen Alias umfasst, der dem Cmdlet **Get-ChildItem** die Buchstabenkombination **dir** zuordnet. Sie können die Zuordnung mit dem Cmdlet **Get-Alias** in der Datei *GetDirAlias.txt* überprüfen.

GetDirAlias.txt

```
PS C:\> Get-Alias dir

CommandType     Name       Definition
-----------     ----       ----------
Alias           dir        Get-ChildItem
```

In Windows PowerShell gibt es kein Cmdlet mit dem Namen **dir**. Der Befehl **dir** ist nicht implementiert. Der Alias **dir** ist jedoch dem Cmdlet **Get-ChildItem** zugeordnet. Deshalb sind die Ausgaben von **dir** in Windows PowerShell und im CMD-Befehlsinterpreter unterschiedlich. Der Alias **dir** wird angezeigt, wenn Sie die Zuordnung mit dem Cmdlet **Get-Alias** analysieren.

Tipp Wenn Sie mit **Get-ChildItem** eine Verzeichnisliste abrufen, geben Sie den Parameter ***-force*** an, um ausgeblendete Dateien und Ordner sowie Systemdateien und -ordner anzuzeigen. Beispiel: **Get-ChildItem** ***-Force.***

Formatieren der Ausgabe

Windows PowerShell umfasst vier Format-Cmdlets. Die drei Cmdlets **Format-List**, **Format-Wide** und **Format-Table** werden häufig verwendet. Das vierte Cmdlet, **Format-Custom**, kann die Ausgabe unter Verwendung der **.format.ps1xml*-Datei in einem anderen Format anzeigen. Sie können entweder die Standardansicht in der Datei **.format.ps1xml* verwenden oder eine neue **.format.ps1xml*-Datei erstellen.

Im folgenden Abschnitt formatieren Sie die Ausgabe zuerst mit dem nützlichsten der Format-Cmdlets: **Format-List**.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Format-List

Format-List ist eines der wichtigsten Cmdlets, das Sie immer wieder verwenden werden. Wenn Sie beispielsweise mit dem Cmdlet **Get-WmiObject** die Eigenschaften der Klasse *Win32_LogicalDisk* abrufen, werden folgende Standardeigenschaften der Klasse angezeigt:

```
PS C:\> Get-WmiObject Win32_LogicalDisk


DeviceID     : C:
DriveType    : 3
ProviderName :
FreeSpace    : 10559041536
Size         : 78452355072
VolumeName   : Sea Drive
```

Obwohl dieses Verhalten in vielen Fällen ausreichend ist, benötigen Sie möglicherweise auch die anderen Klasseneigenschaften. Um die anderen Eigenschaften anzuzeigen, verwenden Sie den Platzhalter *, der alle Eigenschaften auflistet:

```
PS C:\> Get-WmiObject Win32_LogicalDisk | Format-List *


Status                      :
Availability                :
DeviceID                    : C:
StatusInfo                  :
__GENUS                     : 2
__CLASS                     : Win32_LogicalDisk
__SUPERCLASS                : CIM_LogicalDisk
__DYNASTY                   : CIM_ManagedSystemElement
__RELPATH                   : Win32_LogicalDisk.DeviceID="C:"
__PROPERTY_COUNT            : 40
__DERIVATION                : {CIM_LogicalDisk, CIM_StorageExtent,
CIM_LogicalDevice, CIM_LogicalElement...}
__SERVER                    : M5-1875135
__NAMESPACE                 : root\cimv2
__PATH                      : \\M5-1875135\root\cimv2:Win32_LogicalDisk.DeviceID="C:"
Access                      : 0
BlockSize                   :
Caption                     : C:
Compressed                  : False
ConfigManagerErrorCode      :
ConfigManagerUserConfig     :
CreationClassName           : Win32_LogicalDisk
Description                 : Local Fixed Disk
DriveType                   : 3
ErrorCleared                :
ErrorDescription            :
ErrorMethodology            :
FileSystem                  : NTFS
FreeSpace                   : 10559041536
InstallDate                 :
LastErrorCode               :
MaximumComponentLength      : 255
MediaType                   : 12
Name                        : C:
```

```
NumberOfBlocks              :
PNPDeviceID                 :
PowerManagementCapabilities :
PowerManagementSupported    :
ProviderName                :
Purpose                     :
QuotasDisabled              :
QuotasIncomplete            :
QuotasRebuilding            :
Size                        : 78452355072
SupportsDiskQuotas          : False
SupportsFileBasedCompression : True
SystemCreationClassName     : Win32_ComputerSystem
SystemName                  : M5-1875135
VolumeDirty                 :
VolumeName                  : Sea Drive
VolumeSerialNumber          : F0FE15F7
```

Nachdem Sie alle Eigenschaften einer Klasse überprüft haben, können Sie bestimmte Eigenschaften auswählen. Ersetzen Sie den Platzhalter * durch die entsprechenden in der Liste aufgeführten Eigenschaftenamen:

```
PS C:\> Get-WmiObject Win32_LogicalDisk | Format-List Name, FileSystem, FreeSpace


Name       : C:
FileSystem : NTFS
FreeSpace  : 10559029248
```

Anstatt alle Eigenschaftennamen einzugeben, können Sie mit Platzhaltern einen Eigenschaftenbereich auswählen. Um beispielsweise nur die Eigenschaftennamen anzuzeigen, die mit *f* beginnen, geben Sie Folgendes ein:

```
PS C:\> Get-WmiObject Win32_LogicalDisk | Format-List f*


FileSystem : NTFS
FreeSpace  : 10558660608
```

Um die Eigenschaften anzuzeigen, die mit *n* und *f* beginnen, müssen Sie die entsprechenden Buchstaben in eckige Klammern einschließen:

```
PS C:\> Get-WmiObject Win32_LogicalDisk | Format-List [nf]*


FileSystem     : NTFS
FreeSpace      : 10558238720
Name           : C:
NumberOfBlocks :
```

Diese Befehle und die jeweilige Befehlsausgabe finden Sie in der Datei *Format-List.txt* auf der Begleit-CD.

Format-Table

Durch seine Funktionen eignet sich das Cmdlet **Format-Table** insbesondere zum Ausführen von Netzwerkverwaltungsaufgaben. Dieses Cmdlet erstellt Datenspalten, die schnell überprüft werden können. Wie mit **Format-List** und **Format-Wide** können Sie die anzuzeigenden Eigenschaften auswählen, um die von einigen Cmdlets angezeigten langatmigen Informationen zu unterdrücken. In diesem Beispiel

wird eine rekursive Ansicht der Festplatte angezeigt, um alle Protokolldateien mit der Erweiterung *.log* zu ermitteln. Die normalerweise ziemlich lange Ausgabe wurde für dieses Beispiel gekürzt. Mit dem Cmdlet **Format-Table** wurde beispielsweise folgende Ausgabe des Cmdlets **Get-ChildItem** erstellt:

```
PS C:\> Get-ChildItem c:\ -Recurse -Include *.log | Format-Table


    Verzeichnis: Microsoft.PowerShell.Core\FileSystem::C:\Backup_Extras_92705


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---          8/3/2004   6:34 PM    3931872 setupapi.log
-a---          8/2/2004   9:32 PM     206168 Windows Update.log
-a---          6/8/2004  12:41 AM     170095 wmsetup.log
```

Zusätzlich zum Standardverhalten des Cmdlets können Sie auch bestimmte Eigenschaften festlegen. Ein Problem bei dieser Vorgehensweise ist jedoch, dass die Formatierung in der Fensterauflösung erfolgt und die Spalten an den gegenüberliegenden Fensterrändern angezeigt werden. Dies ist für die schnelle Überprüfung ausreichend, aber die Daten sollten nicht in diesem Format gespeichert werden.

```
PS C:\> Get-ChildItem c:\ -Recurse -Include *.log | Format-Table
-Property name, length, lastWriteTime

Name                                                                  Length
LastWriteTime
----
------ -------------
setupapi.log                                                         3931872
8/3/2004 6:34:53 PM
Windows Update.log                                                    206168
8/2/2004 9:32:06 PM
wmsetup.log                                                           170095
6/8/2004 12:41:32 AM
Debug.log                                                                  0
8/23/2006 8:10:38 PM
AVCheck.Log                                                           191694
5/8/2007 9:28:05 AM
AVCheckServer.Log                                                       7762
5/8/2007 9:28:05 AM
```

Um eine Liste zu erstellen, die die Fenstergröße effizienter verwendet, geben Sie den Parameter **-autosize** an. Beachten Sie jedoch, dass der Parameter **-autosize** das längste Element in jeder Spalte ermitteln muss. Der Parameter wartet bis alle Objekte aufgelistet wurden und bestimmt anschließend die maximale Länge jeder Spalte. Dieser Prozess kann längere Zeit in Anspruch nehmen, da die Ausführung des Befehls erst erfolgen kann, wenn alle Elemente bestimmt sind. Möglicherweise möchten Sie jedoch nicht warten, bis **-autosize** alle Elemente aufgelistet hat (wenn Sie beispielsweise einen Serverausfall beheben müssen). Für eine kleine Objektgruppe ist die Leistungseinbuße geringfügig, kann sich jedoch bei einem Befehl bemerkbar machen, dessen Ausführung längere Zeit in Anspruch nimmt. Der Unterschied in der Ausgabe ist jedoch ebenfalls nicht zu unterschätzen (die Ausgabe ist wesentlich überschaubarer).

```
PS C:\> Get-ChildItem c:\ -Recurse -Include *.log | Format-Table
-Property name, length, lastWriteTime -AutoSize
```

```
Name                                                Length LastWriteTime
----                                                ------ -------------
setupapi.log                                       3931872 8/3/2004 6:34:53 PM
Windows Update.log                                  206168 8/2/2004 9:32:06 PM
wmsetup.log                                         170095 6/8/2004 12:41:32 AM
Debug.log                                                0 8/23/2006 8:10:38 PM
AVCheck.Log                                         191694 5/8/2007 9:28:05 AM
```

Format-Table kann auch in Verbindung mit dem CmdLet **Sort-Object** verwendet werden. **Sort-Object** ermöglicht das Sortieren der Daten nach Eigenschaften. In diesem Beispiel wird der Alias für **Sort-Object** (*Sort*) verwendet, der die erforderliche Eingabe verringert. Da der Befehl trotzdem ziemlich lang ist, wurde er in folgendem Beispiel umgebrochen. (Wenn Befehle so lang werden, schreibe ich ein Skript.) Beachten Sie, dass die Daten vor dem Einfügen in das Cmdlet **Format-Table** sortiert werden. Beachten Sie außerdem, dass das Cmdlet **Sort-Object** standardmäßig in aufsteigender Reihenfolge sortiert. Sie können den Parameter **-descending** angeben, um die Dateien in absteigender Reihenfolge zu sortieren.

```
PS C:\>Get-ChildItem c:\ -Recurse -Include *.log | Sort -Property
length | Format-Table name, lastwriteTime, length -AutoSize
Name                                               LastWriteTime           Length
----                                               -------------           ------
PASSWD.LOG                                         5/10/2007 2:44:58 AM         0
sam.log                                            11/29/2006 1:14:33 PM        0
poqexec.log                                        2/1/2007 6:50:49 PM          0
ChkAcc.log                                         5/10/2007 2:45:00 AM         0
Debug.log                                          8/23/2006 8:10:38 PM         0
setuperr.log                                       3/16/2007 7:18:17 AM         0
setuperr.log                                       4/4/2007 6:34:54 PM          0
netlogon.log                                       2/1/2007 7:04:44 PM          3
```

Es gibt noch andere Möglichkeiten zum Sortieren. Beispielsweise können Sie die Liste der Protokolldateien nach dem Datum der letzten Änderung in absteigender Reihenfolge sortieren. In diesem Fall werden die zuletzt geänderten Protokolldateien zuerst angezeigt. Für dieses Verfahren müssen Sie das **Sort-Objekt** ändern. Der restliche Befehl ist gleich. Ein Teil der Ausgabe ist in folgendem Listing dargestellt. Es ist interessant, dass die meisten Protokolle während des Anmeldeprozesses geändert wurden.

```
PS C:\> Get-ChildItem c:\ -Recurse -Include *.log | Sort -Property
lastWriteTime -descending | Format-Table name, lastwriteTime, length -AutoSize
Name                                               LastWriteTime           Length
----                                               -------------           ------
mtrmgr.log                                         10.5.2007 4:56:52    1538364
LocationServices.log                               10.5.2007 4:56:26     830557
StateMessage.log                                   10.5.2007 4:55:00     129595
Scheduler.log                                      10.5.2007 4:55:00     393352
StatusAgent.log                                    10.5.2007 4:53:24     723564
edb.log                                            10.5.2007 4:51:49     131072
PolicyEvaluator.log                                10.5.2007 4:51:25    1672613
ClientLocation.log                                 10.5.2007 4:51:24     330046
FSPStateMessage.log                                10.5.2007 4:51:18     228879
CBS.log                                            10.5.2007 4:46:55   28940091
CertificateMaintenance.log                         10.5.2007 4:42:17     206472
CcmExec.log                                        10.5.2007 4:00:51     537177
wmiprov.log                                        10.5.2007 3:03:11      19503
PolicyAgentProvider.log                            10.5.2007 2:54:02     252866
UpdatesHandler.log                                 10.5.2007 2:53:19     108552
CIAgent.log                                        10.5.2007 2:53:19      99114
ScanAgent.log                                      10.5.2007 2:53:18     354939
```

```
UpdatesDeployment.log                    10.5.2007 2:53:18    1106297
SrcUpdateMgr.log                         10.5.2007 2:53:02     151452
smssha.log                               10.5.2007 2:52:02     107104
execmgr.log                              10.5.2007 2:52:02     150942
InventoryAgent.log                       10.5.2007 2:52:02      34034
ServiceWindowManager.log                 10.5.2007 2:52:02     139955
SdmAgent.log                             10.5.2007 2:49:46     172101
UpdatesStore.log                         10.5.2007 2:49:43      64787
WUAHandler.log                           10.5.2007 2:49:39      14590
CAS.log                                  10.5.2007 2:49:35     198955
PeerDPAgent.log                          10.5.2007 2:49:35       7900
PolicyAgent.log                          10.5.2007 2:49:35     246873
RebootCoordinator.log                    10.5.2007 2:49:35      20420
InternetProxy.log                        10.5.2007 2:49:34      85825
ClientIDManagerStartup.log               10.5.2007 2:49:34     158351
WindowsUpdate.log                        10.5.2007 2:46:46    1553462
edb.log                                  10.5.2007 2:46:43      65536
setupapi.dev.log                         10.5.2007 2:46:38    6469237
setupapi.app.log                         10.5.2007 2:46:38    2722285
WMITracing.log                           10.5.2007 2:45:57   16777216
ChkAcc.log                               10.5.2007 2:45:00          0
PASSWD.LOG                               10.5.2007 2:44:58          0
```

Die Datei *Format-Table.txt* im Ordner *Chapter01* enthält zahlreiche Fehler in der Protokolldatei, da das Cmdlet **Get-ChildItem** versuchte, auf geschützte Verzeichnisse und Dateien zuzugreifen. Während der Entwicklung sind diese Fehlermeldungen hilfreich. Beim Analysieren von Daten erweisen sie sich jedoch als problematisch. Folgendes Listing zeigt ein Beispiel für diese Fehlermeldung:

```
Get-ChildItem : Der Zugriff auf den Pfad 'C:\Windows\CSC' wurde verweigert.
Bei Zeile:1 Zeichen:14
```

Die Fehlermeldung ist hilfreich, da sie den Namen des Cmdlets und die Aktion angibt, die den Fehler verusacht hat. Sie können diese Fehlertypen unterdrücken, indem Sie das Cmdlet **Get-ChildItem** mit dem allgemeinen Parameter **-ErrorAction** ausführen und das Schlüsselwort **SilentlyContinue** angeben. Die geänderte Codezeile:

```
PS C:\> Get-ChildItem c:\ -Recurse -Include *.log -errorAction SilentlyContinue
| Sort -Property lastWriteTime -descending | Format-Table name, lastwriteTime,
length -AutoSize
```

Format-Wide

Das Cmdlet **Format-Wide** ist nicht annähernd so nützlich wie **Format-Table** oder **Format-List**, da die Anzeige auf eine Eigenschaft pro Objekt beschränkt ist. Die Liste kann jedoch hilfreich sein, um beispielsweise ausschließlich die ausgeführten Prozesse anzuzeigen. Sie können das Cmdlet **Get-Process** ausführen und das Ergebnis in das Cmdlet **Format-Wide** überführen. Beispiel:

```
PS C:\> Get-Process | Format-Wide

ApMsgFwd                                 ApntEx
Apoint                                   audiodg
casha                                    CcmExec
csrss                                    csrss
dwm                                      explorer
FwcAgent                                 Idle
InoRpc                                   InoRT
InoTask                                  lsass
```

```
lsm
MSASCui
powershell
rundll32
SearchIndexer
services
smss
SRUserService
svchost
svchost
svchost
svchost
svchost
svchost
svchost
System
taskeng
ThpSrv
wininit
WINWORD
WmiPrvSE
mobsync
powershell
PowerShellIDE
SearchFilterHost
SearchProtocolHost
SLsvc
spoolsv
svchost
svchost
svchost
svchost
svchost
svchost
svchost
svchost
taskeng
ThpSrv
TODDSrv
winlogon
wmdc
WmiPrvSE
```

Obwohl die Ausgabe brauchbar ist, umfasst sie zahlreiche Zeilen und verschwendet viel Platz auf dem Bildschirm. Mit dem Parameter **-column** erhalten Sie eine bessere Ausgabe. Beispiel:

```
PS C:\> Get-Process | Format-Wide -Column 4
```

Die vierspaltige Ausgabe halbiert die Liste, aber der verfügbare Platz auf dem Bildschirm wird nicht optimal genutzt. Obwohl Sie ein Skript schreiben können, das den optimalen Wert für den Parameter **-column** ermittelt (beispielsweise das folgende Skript *DemoFormatWide.ps1*), ist dieses Unterfangen den Aufwand meist nicht wert.

DemoFormatWide.ps1

```
function funGetProcess()
{
 if ($args)
  {
   Get-Process |
   Format-Wide -autosize
  }
 else
  {
   Get-Process |
   Format-Wide -column $i
  }
}

cls
$i = 1
for
   ($i ; $i -le 10 ; $i++)
{
  Write-Host -ForegroundColor red "`$i ist gleich $i"
  funGetProcess
}
  Write-Host -ForeGroundColor red "Und jetzt mit format-wide -autosize"
  funGetProcess("auto")
```

Eine bessere Methode zum Ermitteln der optimalen Bildschirmkonfiguration für **Format-Wide** ist der Parameter **-autosize**:

```
PS C:\> Get-Process | Format-Wide -AutoSize
```

Das Cmdlet Get-Command

Es sind drei Cmdlets verfügbar, die den drei wichtigsten Zutaten in Cajun-Rezepten entsprechen: Salz, Pfeffer und Paprika. Sie möchten Cajun-Bohnen zubereiten? Fügen Sie Salz, Pfeffer und Paprika hinzu. Sie möchten mit Windows PowerShell arbeiten? Verwenden Sie die Cmdlets **Get-Help**, **Get-Command** und **Get-Member**. Mit diesen drei Cmdlets können Sie Windows PowerShell meistern. Da Sie **Get-Help** bereits kennengelernt haben, wird als Nächstes das Cmdlet **Get-Command** beschrieben.

Der einfachste mit **Get-Command** ausgeführte Vorgang besteht im Erstellen einer Liste der in Windows PowerShell verfügbaren Befehle. Diese Liste ist nützlich, um schnell zu überprüfen, welche Cmdlets verfügbar sind. Beachten Sie, dass die Definition im folgenden Beispiel gekürzt ist.

```
PS C:\> Get-Command

CommandType     Name                                               Definition
-----------     ----                                               ----------
Cmdlet          Add-Content                                        Add-Content [-Path] <String[]> [-Value] <Object[...
Cmdlet          Add-History                                        Add-History [[-InputObject] <PSObject[]>] [-Pass...
Cmdlet          Add-Member                                         Add-Member [-MemberType] <PSMemberTypes> [-Name]...
Cmdlet          Add-PSSnapin                                       Add-PSSnapin [-Name] <String[]> [-PassThru] [-Ve...
Cmdlet          Clear-Content                                      Clear-Content [-Path] <String[]> [-Filter <Strin...
Cmdlet          Clear-Item                                         Clear-Item [-Path] <String[]> [-Force] [-Filter ...
```

Da **Get-Command** standardmäßig auf das Erstellen einer Liste von Cmdlets beschränkt ist, ist das *CommandType*-Feld redundant. Das Format der Liste ist übersichtlicher, wenn Sie das Ergebnis in das Cmdlet **Format-List** einfügen und nur den Namen und die Definition auswählen. Wie in folgendem Beispiel dargestellt, ist die Ausgabe einfacher zu lesen und zeigt die syntaktische Definition der Befehle an.

```
PS C:\> Get-Command | Format-List name, definition


Name       : Add-Content
Definition : Add-Content [-Path] <String[]> [-Value] <Object[]> [-PassThru]
[-Filter <String>] [-Include <String[]>] [-Exclude <String[]>] [-Force]
[-Credential<PSCredential>] [-Verbose] [-Debug] [-ErrorAction <ActionPreference>]
[-ErrorVariable<String>] [-OutVariable <String>] [-OutBuffer <Int32>] [-WhatIf]
[-Confirm][-Encoding <FileSystemCmdletProviderEncoding>]             Add-Content
[-LiteralPath] <String[]> [-Value] <Object[]> [-PassThru][-Filter <String>]
[-Include <String[]>] [-Exclude <String[]>] [-Force] [-Credential<PSCredential>]
[-Verbose] [-Debug] [-ErrorAction <ActionPreference>] [-ErrorVariable
<String>] [-OutVariable <String>] [-OutBuffer <Int32>] [-WhatIf] [-Confirm]
[-Encoding <FileSystemCmdletProviderEncoding>]


Name       : Add-History
Definition : Add-History [[-InputObject] <PSObject[]>] [-Passthru] [-Verbose]
[-Debug] [-ErrorAction <ActionPreference>] [-ErrorVariable <String>] [-OutVariable
String>] [-OutBuffer <Int32>]
```

Bis jetzt wurde die normale Verwendung des Cmdlets **Get-Command** erklärt. Eine interessantere Methode basiert jedoch auf der Substantiv/Verb-Kombination der Cmdlet-Namen. Im folgenden Beispiel suchen wir nach Befehlen mit dem Wort *Process* im Namen des Cmdlets:

```
PS C:\> Get-Command -Noun process

CommandType     Name                                               Definition
-----------     ----                                               ----------
Cmdlet          Get-Process                                        Get-Process
[[-Name] <String[]>] [-Verbose] [-De...
Cmdlet          Stop-Process                                       Stop-Process
[-Id] <Int32[]> [-PassThru] [-Verbo...
```

Um mit diesem Verfahren ein Cmdlet mit dem Buchstaben *p* im Namen zu suchen, können Sie Platzhalter verwenden, um die erforderliche Eingabe zu verringern und die verfügbaren Cmdlets anzuzeigen. Beispiel:

```
PS C:\> get-command -Noun p*

CommandType     Name                                               Definition
-----------     ----                                               ----------
Cmdlet          Add-PSSnapin                                       Add-PSSnapin [-Name] <String[]> [-PassThru] [-Ve...
Cmdlet          Convert-Path                                       Convert-Path [-Path] <String[]> [-Verbose] [-Deb...
Cmdlet          Get-PfxCertificate                                 Get-PfxCertificate [-FilePath] <String[]> [-Verb...
Cmdlet          Get-Process                                        Get-Process [[-Name] <String[]>] [-Verbose] [-De...
Cmdlet          Get-PSDrive                                        Get-PSDrive [[-Name] <String[]>] [-Scope <String...
Cmdlet          Get-PSProvider                                     Get-PSProvider [[-PSProvider] <String[]>] [-Verb...
Cmdlet          Get-PSSnapin                                       Get-PSSnapin [[-Name] <String[]>] [-Registered] ...
Cmdlet          Join-Path                                          Join-Path [-Path] <String[]> [-ChildPath] <Strin...
Cmdlet          New-PSDrive                                        New-PSDrive [-Name] <String> [-PSProvider] <Stri...
Cmdlet          Out-Printer                                        Out-Printer [[-Name] <String>] [-InputObject <PS...
Cmdlet          Remove-PSDrive                                     Remove-PSDrive [-Name] <String[]> [-PSProvider <...
Cmdlet          Remove-PSSnapin                                    Remove-PSSnapin [-Name] <String[]> [-PassThru] [...
Cmdlet          Resolve-Path                                       Resolve-Path [-Path] <String[]> [-Credential <PS...
Cmdlet          Set-PSDebug                                        Set-PSDebug [-Trace <Int32>] [-Step] [-Strict] [...
Cmdlet          Split-Path                                         Split-Path [-Path] <String[]> [-LiteralPath <Str...
Cmdlet          Stop-Process                                       Stop-Process [-Id] <Int32[]> [-PassThru] [-Verbo...
Cmdlet          Test-Path                                          Test-Path [-Path] <String[]> [-Filter <String>] ...
Cmdlet          Write-Progress                                     Write-Progress [-Activity] <String> [-Status] <S...
```

Get-Command zeigt standardmäßig ausschließlich Cmdlets an, kann aber auch andere Elemente abrufen, beispielsweise .exe- und .dll-Dateien. **Get-Command** zeigt Informationen zu allen in Windows PowerShell ausgeführten Elementen an. Im folgenden Beispiel wird eine Liste der Befehle aufgeführt, die das Wort *file* enthalten. Beachten Sie, dass ausschließlich Windows PowerShell-Entitäten angezeigt werden.

```
PS C:\> get-command -Name *file*

CommandType     Name                                               Definition
-----------     ----                                               ----------
Application     avifile.dll
C:\Windows\system32\avifile.dll
Application     filemgmt.dll
C:\Windows\system32\filemgmt.dll
Application     FileSystem.format.ps1xml
```

```
C:\Windows\System32\WindowsPowerShell\v1.0\FileS...
Application     filetrace.mof
C:\Windows\System32\Wbem\filetrace.mof
Application     forfiles.exe
C:\Windows\system32\forfiles.exe
```

Sie können dieses Verhalten ändern, indem Sie den Parameter **-commandType** angeben und die Suche auf Cmdlets beschränken. Beispiel:

```
PS C:\> get-command -Name *file* -CommandType cmdlet

CommandType     Name                                               Definition
-----------     ----                                               ----------
Cmdlet          Out-File                                           Out-File
[-FilePath] <String> [[-Encoding] <Stri
```

Diese Beispiele veranschaulichen die Suchabfragen, die Sie mit dem Cmdlet **Get-Command** ausführen können. Diese Befehle und Befehlsausgaben finden Sie in der Datei *Get-Command.txt* im Ordner *Chapter01* auf der Begleit-CD.

Das Cmdlet Get-Member

Das dritte wichtige Cmdlet von Windows PowerShell ist **Get-Member**. Einige Skriptentwickler sind skeptisch, wenn ich das Cmdlet **Get-Member** als eines der drei „Cajun"-Cmdlets vorstelle. Ein Entwickler fragte mich sogar, für was dieses Cmdlet gut ist. Diese Frage ist durchaus angebracht. **Get-Member** ist deshalb so nützlich, weil es anzeigt, welche Eigenschaften und Methoden von einem Objekt unterstützt werden. Ein einfaches Beispiel veranschaulicht die Nützlichkeit dieses Cmdlets.

Wenn Sie beispielsweise das Cmdlet **Get-Item** ausführen, um ein Objekt abzurufen, das den Ordner *meinTest* repräsentiert, können Sie die Referenz in der Variablen *$a* speichern. Beispiel:

```
PS C:\> $a = Get-Item c:\meinTest
```

Wenn Sie der Variablen *$a* eine Instanz des Ordnerobjekts zuweisen, können Sie die Methoden und Eigenschaften eines Ordnerobjekts überprüfen, indem Sie das Objekt in das Cmdlet **Get-Member** einfügen. Der Befehl und seine Ausgabe:

```
PS C:\> $a | Get-Member


   TypeName: System.IO.DirectoryInfo

Name                      MemberType     Definition
----                      ----------     ----------
Create                    Method         System.Void Create(), System.Void
Create(DirectorySecurity directorySecurity)
CreateObjRef              Method         System.Runtime.Remoting.ObjRef
CreateObjRef(Type requestedType)
CreateSubdirectory        Method         System.IO.DirectoryInfo
CreateSubdirectory(String path), System.IO.Director...
Delete                    Method         System.Void Delete(), System.Void
Delete(Boolean recursive)
Equals                    Method         System.Boolean Equals(Object obj)
GetAccessControl          Method         System.Security.AccessControl.DirectorySecurity GetAccessControl(), System
GetDirectories            Method         System.IO.DirectoryInfo[]
GetDirectories(), System.IO.DirectoryInfo[GetFiles                  Method         System.IO.FileInfo[] GetFiles(Str
ing searchPattern), System.IO.FileInfo[] G...
GetFileSystemInfos        Method         System.IO.FileSystemInfo[] GetFileSystemInfos(String searchPattern), Syst
```

```
em...
GetHashCode               Method         System.Int32 GetHashCode()
GetLifetimeService        Method         System.Object GetLifetimeService()
GetObjectData             Method         System.Void GetObjectData
*(SerializationInfo info, StreamingContext context)
GetType                   Method         System.Type GetType()
get_Attributes            Method         System.IO.FileAttributes get_Attributes()
get_CreationTime          Method         System.DateTime get_CreationTime()
get_CreationTimeUtc       Method         System.DateTime get_CreationTimeUtc()
get_Exists                Method         System.Boolean get_Exists()
get_Extension             Method         System.String get_Extension()
get_FullName              Method         System.String get_FullName()
get_LastAccessTime        Method         System.DateTime get_LastAccessTime()
get_LastAccessTimeUtc     Method         System.DateTime get_LastAccessTimeUtc()
get_LastWriteTime         Method         System.DateTime get_LastWriteTime()
get_LastWriteTimeUtc      Method         System.DateTime get_LastWriteTimeUtc()
get_Name                  Method         System.String get_Name()
get_Parent                Method         System.IO.DirectoryInfo get_Parent()
get_Root                  Method         System.IO.DirectoryInfo get_Root()
InitializeLifetimeService Method         System.Object InitializeLifetimeService()
MoveTo                    Method         System.Void MoveTo(String destDirName)
Refresh                   Method         System.Void Refresh()
SetAccessControl          Method         System.Void
SetAccessControl(DirectorySecurity directorySecurity)
set_Attributes            Method         System.Void set_Attributes(FileAttributes
value)
set_CreationTime          Method         System.Void set_CreationTime(DateTime
value)
set_CreationTimeUtc       Method         System.Void set_CreationTimeUtc(DateTime
value)
set_LastAccessTime        Method         System.Void set_LastAccessTime(DateTime
value)
set_LastAccessTimeUtc     Method         System.Void set_LastAccessTimeUtc(DateTime
value)
set_LastWriteTime         Method         System.Void set_LastWriteTime(DateTime
value)
set_LastWriteTimeUtc      Method         System.Void set_LastWriteTimeUtc(DateTime
value)
ToString                  Method         System.String ToString()
PSChildName               NoteProperty   System.String PSChildName=meinTest
PSDrive                   NoteProperty   System.Management.Automation.PSDriveInfo
PSDrive=C
PSIsContainer             NoteProperty   System.Boolean PSIsContainer=True
PSParentPath              NoteProperty   System.String
PSParentPath=Microsoft.PowerShell.Core\FileSystem::C:\
PSPath                    NoteProperty   System.String
PSPath=Microsoft.PowerShell.Core\FileSystem::C:\meinTest
PSProvider                NoteProperty   System.Management.Automation.ProviderInfo
PSProvider=Microsoft.PowerShell.C...
Attributes                Property       System.IO.FileAttributes Attributes
{get;set;}
CreationTime              Property       System.DateTime CreationTime {get;set;}
CreationTimeUtc           Property       System.DateTime CreationTimeUtc {get;set;}
Exists                    Property       System.Boolean Exists {get;}
Extension                 Property       System.String Extension {get;}
FullName                  Property       System.String FullName {get;}
```

```
LastAccessTime            Property       System.DateTime LastAccessTime {get;set;}
LastAccessTimeUtc         Property       System.DateTime LastAccessTimeUtc {get;set;}
LastWriteTime             Property       System.DateTime LastWriteTime {get;set;}
LastWriteTimeUtc          Property       System.DateTime LastWriteTimeUtc {get;set;}
Name                      Property       System.String Name {get;}
Parent                    Property       System.IO.DirectoryInfo Parent {get;}
Root                      Property       System.IO.DirectoryInfo Root {get;}
Mode                      ScriptProperty System.Object Mode {get=$catr = "";...
```

Die Liste der Ordnermitglieder enthält eine übergeordnete Eigenschaft. Mittels dieser Eigenschaft können Sie weitere Informationen über den Ordner *meinTest* abrufen. Beispiel:

```
PS C:\> $a.parent

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d--hs         5/11/2007   2:39 PM            C:\
```

Um herauszufinden, wann der Ordner zum letzten Mal geöffnet wurde, verwenden Sie die Eigenschaft *LastAccessTime*:

```
PS C:\> $a.LastAccessTime

Freitag, 11. Mai 2007 14:39:12
```

Um zu bestätigen, dass es sich beim Objekt in *$a* um einen Ordner handelt, verwenden Sie die Eigenschaft *PsIsContainer*. Die Ausgabe von **Get-Member** zeigt an, dass *PsIsContainer* ein boolescher Wert ist und gibt entweder *true* oder *false* zurück. Beispiel:

```
PS C:\> $a.PsIsContainer
True
```

Sie können eine der zurückgegebenen Methoden ausführen. Die Methode *moveTo* verschiebt den Ordner in ein anderes Verzeichnis. **Get-Member** gibt an, dass die Methode *moveTo* die Eingabe des Zielverzeichnisses erfordert. Verschieben Sie den Ordner *meinTest* in *C:\verschobenerOrdner* und führen Sie das Cmdlet **Test-Path** aus, um zu überprüfen, ob der Ordner in das neue Verzeichnis verschoben wurde. Beispiel:

```
PS C:\> $a.MoveTo("C:\verschobenerOrdner")
PS C:\> Test-Path c:\verschobenerOrdner
True
PS C:\> Test-Path c:\meinTest
False
PS C:\>
```

Um den Namen des Ordners zu bestätigen, der vom Objekt in der Variablen *$a* repräsentiert wird, verwenden Sie die Eigenschaft *Name*. Beispiel:

```
PS C:\> $a.name
verschobenerOrdner
```

Wenn Sie den Ordner löschen möchten, führen Sie die Methode *delete* aus. Um zu bestätigen, dass der Ordner gelöscht wurde, verwenden Sie **dir m***. Beachten Sie, dass der Ordner gelöscht wurde.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

```
PS C:\> $a.Delete()
PS C:\> dir m*


    Directory: Microsoft.PowerShell.Core\FileSystem::C:\


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----         4/21/2007   4:56 PM            Maps
d----          5/5/2007   3:51 PM            music
-a---          2/1/2007   6:17 PM         54 MASK.txt
```

Alle Befehle und deren Befehlsausgaben finden Sie in der Datei *Get-Member.txt* im Ordner *Chapter01* auf der Begleit-CD.

Arbeiten mit dem .NET Framework

Ein interessanter Aspekt ist, dass diese Befehle eigentlich auf dem .NET Framework basieren und keine Windows PowerShell-Befehle sind. Die Cmdlets **Get-Item**, **Get-Member** und **Test-Path** sind natürlich Windows PowerShell-Befehle, aber *System.IO.DirectoryInfo* kommt nicht aus Windows PowerShell. Das heißt, Sie verwenden in Windows PowerShell die gleichen Methoden und Eigenschaften wie ein Visual Basic .NET- oder C#-Entwickler. Das bedeutet auch, dass Ihnen mit dem Microsoft Developer Network (MSDN) und dem Windows Software Development Kit (SDK) umfassendere Informationen zur Verfügung stehen. Wenn Sie bestimmte Informationen nicht in der Onlinehilfe finden können, können Sie immer noch auf der MSDN-Website oder im Windows SDK nachlesen.

Zusammenfassung

In diesem Kapitel wurden verschiedene Methoden beschrieben, mit denen Sie bestimmen können, ob Windows PowerShell installiert ist. Außerdem wurde die Konfiguration von Windows PowerShell in Unternehmensumgebungen erklärt. Sie haben Windows PowerShell-Profile erstellt und sowohl die Windows PowerShell als auch die Windows PowerShell-Befehle mit verschiedenen Methoden ausgeführt. Das Kapitel erklärte das Erweitern der Features von Windows PowerShell über benutzerdefinierte Aliase und Funktionen. Außerdem wurden drei wichtige Windows PowerShell-Cmdlets beschrieben: **Get-Help**, **Get-Command** und **Get-Member**.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

KAPITEL 2

Skripting mit Windows PowerShell

Nach Abschluss dieses Kapitels können Sie:

- Skriptrichtlinien für Windows PowerShell konfigurieren
- Windows PowerShell-Skripts ausführen
- Flusskontrollanweisungen in Windows PowerShell verwenden
- Anweisungen für die Entscheidungsfindung und für Verzweigungen verwenden
- Datentypen identifizieren und verwenden
- Unter Verwendung von regulären Ausdrücken erweiterte Übereinstimmungsfunktionen bereitstellen
- Befehlszeilenargumente verwenden

Auf der Begleit-CD Alle Skripts in diesem Kapitel befinden sich auf der Begleit-CD im Ordner *\Scripts\Chapter02*.

Warum Skripts verwenden?

Für viele Netzwerkadministratoren ist das Erstellen von Skripts (jeglicher Skripts) ein Buch mit sieben Siegeln oder eine Aufgabe, die mehr mit dem Lesen von Kaffeesätzen als mit dem Verwalten eines Servers zu tun hat. Die meisten großen Unternehmen beschäftigen (trotz der bisherigen Bemühungen von Microsoft, Visual Basic Scripting Edition (VBScript) als Skriptsprache für Verwaltungsaufgaben verstärkt ins Spiel zu bringen) nur einen „Skripter“. Zwar stimmen die meisten professionellen Netzwerkadministratoren zu, dass das Erstellen von Skripts zum Vornehmen von Konfigurationsänderungen an Netzwerkservern eine wertvolle Fähigkeit ist, dennoch verfügen nur wenige Administratoren über die erforderlichen Kenntnisse. Tatsächlich beruhen die Fähigkeiten vieler „Skripter“ einfach nur darauf, ein passendes Skript zu finden, das einfach geändert werden kann, anstatt neue Skripts selbst zu erstellen.

Das wird sich mit der Windows PowerShell hoffentlich ändern. Die Syntax von Windows PowerShell wurde absichtlich so entwickelt, dass die Verwendung und das Erlernen der Syntax einfacher ist. Bei den Zielbenutzern handelt es sich um Windows-Administratoren großer Unternehmen.

Warum sollten Sie also Skripts verwenden? Hierfür gibt es mehrere Gründe. Beispielsweise vereinfachen Skripts das Dokumentieren bestimmter Befehlsfolgen. Um beispielsweise eine Liste aller Freigaben auf einem Computer zu erstellen, können Sie die WMI-Klasse *Win32_share* und das Cmdlet **Get-WmiObject** verwenden, um die Ergebnisse abzurufen:

```
PS C:\> Get-wmiObject win32_share

Name                    Path                          Description
----                    ----                          -----------
ADMIN$                  C:\Windows                    Remoteverwaltung
C$                      C:\                           Standardfreigabe
CCMLogs$                C:\Windows\system32\ccm\logs
```

```
CCMSetup$                C:\Windows\system32\ccmsetup
IPC$                                                    Remote-IPC
Musik                    C:\Musik                       keine
VPCache$                 C:\Windows\system32\VPCache
WMILogs$                 C:\Windows\system32\wbem\logs
```

Möchten Sie nur eine Liste der Dateifreigaben anzeigen? Möglicherweise wissen Sie nicht, dass eine Dateifreigabe eine Freigabe des Typs 0 ist. Allerdings können Sie diese Informationen möglicherweise im Internet finden. Nachdem Sie die erforderlichen Informationen ermittelt haben, führen Sie folgenden Befehl aus:

```
PS C:\> Get-WmiObject win32_share -Filter "type = '0'"

Name                     Path                           Description
----                     ---                            -----------
CCMLogs$                 C:\Windows\system32\ccm\logs
CCMSetup$                C:\Windows\system32\ccmsetup
Musik                    C:\Musik                       keine
VPCache$                 C:\Windows\system32\VPCache
WMILogs$                 C:\Windows\system32\wbem\logs
```

Sie müssen sich den Freigabetyp 0 merken. Allerdings ist die Syntax nicht besonders vielsagend. Wo vermerken Sie also diese Informationen? Hier ist ein Vorschlag: Während ich als Administrator für Digital VAX tätig war, hatte ich immer ein kleines Notebook zur Hand, um kryptische Befehle zu speichern. Natürlich gab es dabei manchmal das Problem, dass ich mein Notebook nicht finden konnte oder dieses vergessen hatte.

Wenn Sie nur die Dateifreigaben anzeigen möchten, denen keine Beschreibung zugeordnet ist, führen Sie folgenden Befehl aus:

```
PS C:\> Get-WmiObject win32_share -Filter "type = '0' AND description = ''"

Name                       Path                         Description
----                       ----                         -----------
CCMLogs$                   C:\Windows\system32\ccm\logs
CCMSetup$                  C:\Windows\system32\ccmsetup
VPCache$                   C:\Windows\system32\VPCache
WMILogs$                   C:\Windows\system32\wbem\logs
```

An dieser Stelle stimmen Sie mir wahrscheinlich zu, dass die Befehlssyntax kompliziert genug ist, um das Erstellen eines Skripts zu rechtfertigen. Das Erstellen eines Skripts ist einfach. Kopieren Sie den in Windows PowerShell angezeigten Befehl in die Zwischenablage und fügen Sie diesen anschließend in eine Textdatei ein. Geben Sie dem Skript einen Namen und speichern Sie dieses mit der Erweiterung *.psl* ab. Sie können dann das Skript in Windows PowerShell ausführen. Sie finden die zuvor ausgeführten Befehle in der Datei *Share.txt* im Ordner *Chapter02* auf der Begleit-CD. Das Skript hat den Namen *GetFileShares.ps1*.

Ein weiterer Vorteil des Konfigurierens von Befehlen mit Hilfe von Skripts ist, dass Sie Änderungen recht einfach vornehmen können. Der vorherige Befehl war auf Dateifreigaben beschränkt. Sie können dieses Skript jedoch ändern, um Druckfreigaben, Remotefreigaben, IPC-Freigaben oder einen anderen definierten Freigabetyp abzurufen. Das Skript kann beispielsweise so angepasst werden, dass Sie beim Aufrufen des Skripts einen Freigabetyp angeben können. Verwenden Sie zu diesem Zweck eine *if … else-Anweisung*, um zu überprüfen, ob das Skript ein Befehlszeilenargument umfasst.

Tipp Überprüfen Sie die automatisch erstellte Variable *$args,* welche die Befehlszeilenargumente enthält.

Ist ein Befehlszeilenargument vorhanden, können Sie den in der Befehlszeile angegebenen Wert direkt verwenden. Wenn beim Ausführen des Skripts jedoch kein Wert angegeben wird, müssen Sie einen Standardwert in das Skript einfügen. Beispielsweise können Sie in Ihrem Skript die Dateifreigaben auflisten und dem Benutzer mitteilen, dass Standardwerte verwendet werden. Die **Get-WmiObject**-Syntax ist mit der VBScript-Syntax identisch. Das folgende Skript mit dem Namen *GetSharesWithArgs.ps1* umfasst einen Beispielbefehl mit der korrekten Syntax. Darüber hinaus zeigt das Skript eine Zeichenfolge mit Hilfeinformationen an, wenn kein Befehlszeilenargument angegeben wurde.

GetSharesWithArgs.ps1

```
if($args)
 {
   $type = $args
   Get-WmiObject win32_share -Filter "type = $type"
 }
ELSE
 {
  Write-Host
  "
  Es wird der Standardwert für Dateifreigabetypen (= 0) verwendet.
  Andere gültige Typen sind:
  2147483648 für die Verwaltung administrativer Dateifreigaben.
  2147483649 für die Verwaltung von Druckfreigaben.
  2147483650 für die Geräteverwaltung.
  2147483651 für die Verwaltung von IPC-Freigaben.
  Beispiel: C:\GetSharesWithArgs.ps1 '2147483651'
  "
  $type = '0'
  Get-WmiObject win32_share -Filter "type = $type"
 }
```

Ein weiterer Grund, warum sich Netzwerkadministratoren mit dem Erstellen von Windows PowerShell-Skripts befassen sollten, ist das Ausführen von Skripts als geplante Tasks. Die Windows-Betriebssysteme umfassen mehrere Taskplanermodule. Mit der WMI-Klasse *Win32_ScheduledJob* können Sie geplante Tasks erstellen, überwachen und löschen. Diese WMI-Klasse ist seit Windows NT 4.0 verfügbar. Allerdings umfassen Windows XP und Windows Server 2003 auch ein Dienstprogramm namens *Schtasks.exe*, das flexibler als die WMI-Klasse *Win32_ScheduledJob* ist. In Windows Vista und Windows Server 2008 ist außer *Schtasks.exe* auch das *Schedule.Service*-Objekt integriert, um die Konfiguration geplanter Tasks zu vereinfachen.

Das Skript *ListProcessesSortResults.ps1* kann mittels eines geplanten Tasks mehrmals täglich ausgeführt werden. Dieses Skript erstellt eine Liste der aktuellen Prozesse und schreibt die Ergebnisse in eine als Tabelle formatierte und sortierte Textdatei.

ListProcessesSortResults.ps1

```
$args = "localhost","loopback","127.0.0.1"

foreach ($i in $args)
   {$strFile = "c:\test\"+ $i +"_Prozesse.txt"
    Write-Host "Testen von -" $i "- Bitte warten...";
    Get-WmiObject -computername $i -class win32_process |
    Select-Object name, processID, Priority, ThreadCount, PageFaults,
      PageFileUsage |
    Where-Object {!$_.processID -eq 0} | Sort-Object -property name |
    Format-Table | Out-File $strFile}
```

Konfigurieren der Skriptrichtlinie

Da das Skripting in Windows PowerShell standardmäßig nicht aktiviert ist, ist es wichtig, vor der Bereitstellung von Skripts oder Befehlen den Grad der Skriptunterstützung zu überprüfen. Wenn Sie ein Windows PowerShell-Skript ausführen möchten, die Skriptunterstützung jedoch nicht aktiviert ist, wird eine Fehlermeldung angezeigt. Die Fehlermeldung ist in Abbildung 2.1 dargestellt.

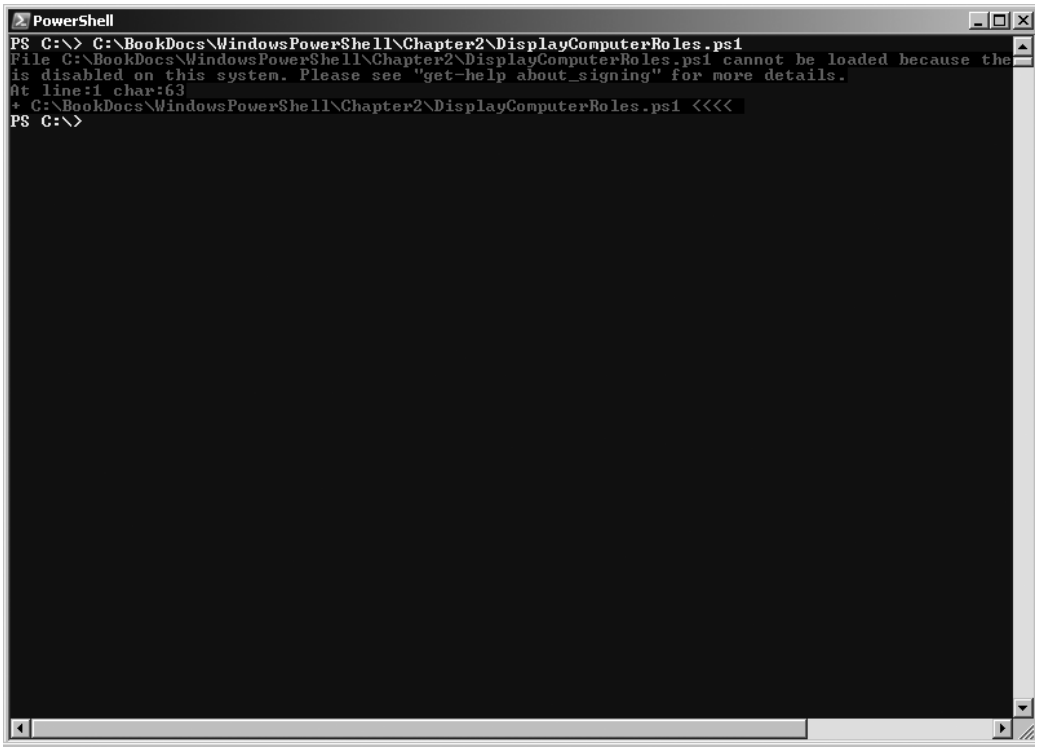


Abbildung 2.1 Beim Versuch, ein Skript ohne Skriptunterstützung auszuführen, wird eine Fehlermeldung generiert

Die Standardeinstellung verwendet eine eingeschränkte Ausführungsrichtlinie. In Windows PowerShell können Sie mit dem Cmdlet **Set-ExecutionPolicy** vier Stufen für die Ausführungsrichtlinie konfigurieren, die in Tabelle 2.1 aufgeführt sind. Die Ausführungsrichtlinie kann über eine Active Directory-Gruppenrichtlinie unter Verwendung der **Windows PowerShell**-Einstellung **Skriptausführung aktivieren** konfiguriert werden. Anschließend können Sie die Richtlinie auf Computerobjekte oder Benutzerobjekte anwenden. Die Einstellungen für Computerobjekte haben Vorrang vor den anderen Einstellungen.

Tipp Verwenden Sie das Cmdlet **Get-ExecutionPolicy**, um die aktuelle Skriptausführungsrichtlinie abzurufen.

Sie können die Benutzereinstellungen für die Ausführungsrichtlinie mit dem Cmdlet **Set-ExecutionPolicy** konfigurieren. Beachten Sie jedoch, dass diese Einstellungen die von der Gruppenrichtlinie konfigurierten

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Einstellungen nicht überschreiben. Zeigen Sie die resultierenden Einstellungen für die Ausführungsrichtlinie mit dem Cmdlet **Get-ExecutionPolicy** an.

Tabelle 2.1 Einschränkungsstufen für Skriptausführungsrichtlinien

Stufe	Bedeutung
Restricted (Eingeschränkt)	Führt keine Skripts oder Konfigurationsdateien aus
AllSigned (Alle signiert)	Alle Skripts und Konfigurationsdateien müssen von einem vertrauenswürdigen Herausgeber signiert sein
RemoteSigned (Remote signiert)	Alle aus dem Internet heruntergeladenen Skripts und Konfigurationsdateien müssen von einem vertrauenswürdigen Herausgeber signiert sein
Unrestricted (Uneingeschränkt)	Alle Skripts und Konfigurationsdateien werden ausgeführt. Aus dem Internet heruntergeladene Skripts müssen vor dem Ausführen bestätigt werden.

Beachten Sie, dass in Windows Vista und Windows Server 2008 der Zugriff auf den Registrierungsschlüssel für die Skriptausführungsrichtlinie eingeschränkt ist. Ein „normaler“ Benutzer kann den Schlüssel nicht ändern. Sogar ein Administrator ist bei aktivierter Benutzerkontensteuerung (User Account Control, UAC) nicht berechtigt, diese Einstellung zu ändern. Beim Versuch die Einstellung zu ändern, wird die in Abbildung 2.2 dargestellte Fehlermeldung angezeigt.

Das UAC-Problem kann jedoch auf mehrere Arten umgangen werden. Sie können beispielsweise die Benutzerkontensteuerung deaktivieren, was jedoch keine optimale Lösung ist. Es ist besser, mit der rechten Maustaste auf das Symbol **Windows PowerShell** im Startmenü zu klicken und die Option **Als Administrator ausführen** auszuwählen (siehe Abbildung 2.3).

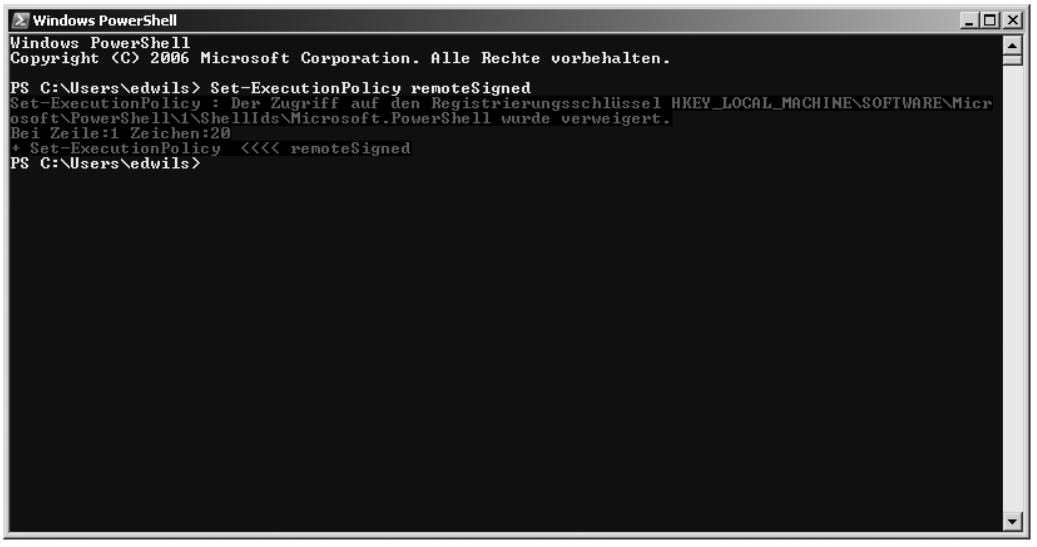


Abbildung 2.2 Das Ausführen des Cmdlets *Set-ExecutionPolicy* schlägt fehl, wenn der Benutzer nicht über Administratorberechtigungen verfügt

Wenn Ihnen das Klicken mit der Maus zu zeitaufwändig ist, können Sie eine zweite Windows PowerShell-Verknüpfung erstellen. Geben Sie der zweiten Verknüpfung den Namen **Admin_PS** und

konfigurieren Sie die Eigenschaften so, dass Windows PowerShell mit Administratorrechten gestartet wird. Für ca. 90 Prozent der Verwaltungsaufgaben sollte die erste Verknüpfung jedoch ausreichend sein. Sollten Sie jedoch mehr Rechte benötigen, wählen Sie die Verknüpfung mit Administratorberechtigungen aus. Die verfügbaren Eigenschaften für die Verknüpfung **Admin_PS** sind in Abbildung 2.4 dargestellt.

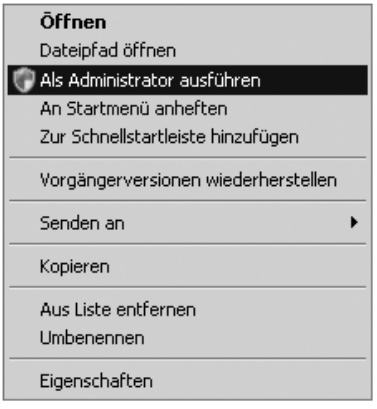


Abbildung 2.3 Um Windows PowerShell mit Administratorberechtigungen zu starten, klicken Sie mit der rechten Maustaste auf das Symbol und wählen Sie *Als Administrator ausführen* aus

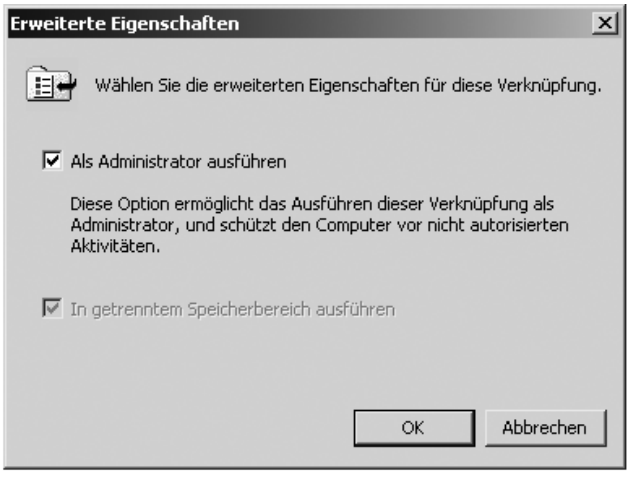


Abbildung 2.4 Um die Windows PowerShell-Verknüpfung mit Administratorberechtigungen auszuführen, aktivieren Sie in den erweiterten Eigenschaften das Kontrollkästchen *Als Administrator ausführen*

Ausführen von Windows PowerShell-Skripts

Sie können nicht einfach auf ein Windows PowerShell-Skript doppelklicken, um dieses auszuführen. Sie können auch nicht einfach den Skriptnamen in das Dialogfeld **Ausführen** im Startmenü eingeben. Sie müssen stattdessen das Skript in Windows PowerShell ausführen, wenn die Ausführungsrichtlinie diesen Vorgang zulässt. Sie müssen jedoch den vollständigen Pfad zum Skript eingeben, einschließlich der Erweiterung *.ps1*.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Um ein Skript außerhalb von Windows PowerShell auszuführen, müssen Sie den vollständigen Pfad zum Skript eingeben und als Argument an *PowerShell.exe* übergeben. Geben Sie außerdem den Parameter **-noexit** an, um die Ausgabe des Skripts in der Windows PowerShell-Konsole anzuzeigen. Die Syntax ist Abbildung 2.5 dargestellt.

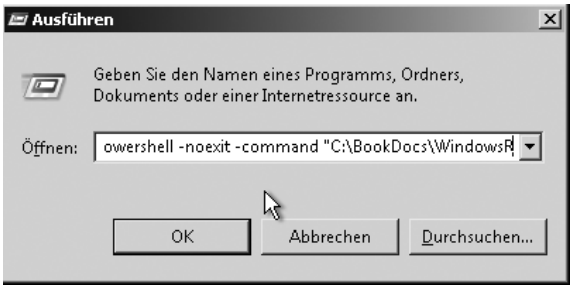


Abbildung 2.5 Um ein Skript außerhalb von Windows PowerShell auszuführen, verwenden Sie das Argument *-noexit*, um die Ausgabe des Skripts anzuzeigen

Verwenden von Variablen

Bei der Arbeit mit Windows PowerShell werden Variablen standardmäßig nicht im Voraus angegeben, sondern beim Zuweisen von Datenwerten deklariert. Allen Variablennamen muss ein Dollarzeichen vorangehen. Windows PowerShell umfasst mehrere spezielle Variablen, die automatisch erstellt werden. In Tabelle 2.2 sind diese Variablen und ihre Bedeutung aufgeführt.

Tabelle 2.2 Verwenden von speziellen Variablen

Name	Verwendung
$^	Enthält das erste Token der letzten Zeileneingabe in der Shell
$$	Enthält das letzte Token der letzten Zeileneingabe in der Shell
$_	Das aktuelle Pipelineobjekt; wird in Skriptblöcken, Filtern, *Where*-Klauseln, *ForEach*-Schleifen und *Switch*-Blöcken verwendet
$?	Enthält den Status (erfolgreich/fehlgeschlagen) der letzten Anweisung
$args	Wird zum Erstellen von Funktionen verwendet, die Parameter erfordern
$error	Wenn ein Fehler auftritt, wird das *error*-Objekt in der Variablen *$error* gespeichert
$executioncontext	Die für Cmdlets verfügbaren *execution*-Objekte
$foreach	Verweist auf den Enumerator in einer *ForEach*-Schleife
$home	Das Basisverzeichnis des Benutzers (%HOMEDRIVE%\%HOMEPATH%)
$input	Die Eingabe wird in eine Funktion oder einen Codeblock übertragen
$match	Eine Hashtabelle, die aus den vom **-match** Operator gefundenen Elementen besteht
$myinvocation	Informationen über das derzeit ausgeführte Skript oder die Befehlszeile
$pshome	Das Verzeichnis, in dem Windows PowerShell installiert ist
$host	Informationen über den aktuellen Host

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Tabelle 2.2 Verwenden von speziellen Variablen *(Fortsetzung)*

Name	Verwendung
$lastexitcode	Der Beendigungscode, der zuletzt ausgeführten systemeigenen Anwendung
$true	Boolescher Wert TRUE
$false	Boolescher Wert FALSE
$null	Ein Null-Objekt
$this	Repräsentiert das aktuelle Objekt in der XML-Datei *Types.ps1* und in einigen Skriptblockinstanzen
$ofs	Feldtrennzeichen in der Ausgabe, das beim Konvertieren eines Arrays in eine Zeichenfolge verwendet wird
$shellid	Die ID für die Shell. Dieser Wert wird von der Shell verwendet, um die Ausführungsrichtlinie und die beim Start auszuführenden Profile zu bestimmen.
$stacktrace	Enthält detaillierte Stapelüberwachungsinformationen zum letzten Fehler

Verwenden von Konstanten

Konstanten werden in Windows PowerShell wie Variablen verwendet, allerdings mit zwei wichtigen Ausnahmen: Der Wert einer Konstanten ändert sich nicht und eine Konstante kann nicht gelöscht werden. Konstanten werden mit dem Cmdlet **Set-Variable** und dem Argument **-option** erstellt.

Tipp Wenn Sie im Text eines Skripts auf eine Konstante verweisen, müssen Sie dieser (wie einer Variablen) ein Dollarzeichen voranstellen. Das Dollarzeichen ist jedoch nicht erforderlich, wenn Sie die Konstante (oder eine Variable) mit dem Cmdlet **Set-Variable** erstellen und das Argument **name** angeben.

Das folgende Skript *GetHardDiskDetails.ps1* enthält die Konstante *$intDriveType* mit dem Wert 3. Diese Konstante wird verwendet, um für die WMI-Klasse *Win32_LogicalDisk* die Eigenschaft *DiskType* auf den Wert 3 festzulegen. Der Wert 3 bezeichnet lokale Festplatten. Die *Where*-Klausel verwendet diesen Wert, um alle Netzlaufwerke, austauschbaren Datenträger und RAM-Laufwerke von den zurückgegebenen Objekten auszuschließen.

Die Konstante *$intDriveType* wird nur in der *Where*-Klausel verwendet. Der Wert von *$strComputer* ändert sich jedoch für jeden im Array *$aryComputers* angegebenen Computernamen. Im Skript *GetHardDiskDetails.ps1* ändert sich der Wert von *$strComputer* zweimal. Das erste Mal entspricht der Computername dem Wert *loopback* und das zweite Mal *localhost*. Auch wenn Sie 250 verschiedene Computernamen hinzufügen, ist das Ergebnis das gleiche: Der Wert von *$strComputer* ändert sich jedes Mal.

GetHardDiskDetails.ps1

```
$aryComputers = "loopback", "localhost"
Set-Variable -name intDriveType -value 3 -option constant

foreach ($strComputer in $aryComputers)

   {"Lokale Festplatten von: " + $strComputer
   Get-WmiObject -class win32_logicaldisk -computername $strComputer|
      Where {$_.drivetype -eq $intDriveType}}
```

Verwenden von Flusskontrollanweisungen

Nachdem Sie die Skriptunterstützung in Windows PowerShell aktiviert haben, können Sie die Cmdlets für die erweiterte Flusskontrolle in Ihren Skripts verwenden. Das bedeutet jedoch nicht, dass Sie die Flusskontrolle nicht auch in der Konsole ausführen können. Sie können die Flusskontrollanweisungen jederzeit in der Konsole verwenden. Beispiel:

```
PS C:\> Get-Process | foreach ( $_.name ) { if ( $_.name -eq "system" ) {
Write-Host "Die ID des Systemprozesses lautet: " $_.ID } }
```

Das Problem ist jedoch die Eingabe. Sie sollten einen Befehl wie diesen in einem Skript speichern. Das spart nicht nur Zeit, sondern verbessert auch die Lesbarkeit. Beispielsweise können Sie die geschwungenen Klammern und die anderen Komponenten des Befehls in einer Textdatei ausrichten. Außerdem können Sie die Prozessnamen als Variablen speichern, anstatt diese in das Skript einzugeben. Das Skript kann einfach geändert werden, um beispielsweise Befehlszeilenargumente einzubeziehen. Das folgende Skript *GetProcessByID.ps1* verdeutlicht diese Vorgehensweise.

GetProcessByID.ps1

```
$strProcess = "system"
Get-Process |
foreach ( $_.name ) {
  if ( $_.name -eq $strProcess )
   {
    Write-Host "Die ID des Systemprozesses lautet: " $_.ID
   }
}
```

Hinzufügen von Parametern zum Cmdlet ForEach-Object

Im Skript *GetWmiAndQuery.ps1* erstellt das Cmdlet **ForEach-Object** eine Liste aller WMI-Klassen deren Namen die Zeichenfolge *usb* enthalten. Dieses Skript ist ausgesprochen nützlich, da es den Prozessnamen zusammen mit der Prozess-ID (PID) auflistet. Außerdem kann das Skript *GetProcessByID.ps1* geändert werden, um Befehlszeilenargumente zu akzeptieren. Mit dem Parameter **-list** zeigt das Cmdlet **Get-WmiObject** eine vollständige Liste der WMI-Klassen im WMI-Standardnamespace an. Fügen Sie das resultierende Objekt in das Cmdlet **Where-Object** ein und filtern Sie das Ergebnis nach der Eigenschaft *Name*, wenn der Wert in der Variablen *$strClass* enthalten ist.

Verwenden des Parameters Begin

Geben Sie den Parameter **-begin** des Cmdlets **ForEach-Object** an, um den zum Generieren der WMI-Klassenliste verwendeten Namen auszugeben. Diese Aktion wirkt sich nicht auf das aktuelle Pipelineobjekt aus. Die Parameter **-begin** und **-end** stehen nicht mit dem aktuellen Pipelineobjekt in Verbindung. Diese Parameter führen jedoch Vor- und Nachverarbeitungsprozesse aus. Der Parameter **-process** wird verwendet, um den Skriptblock einzubeziehen, der mit dem aktuellen Pipelineobjekt in Verbindung steht. Dieser Parameter ist der Standardparameter und muss nicht benannt werden. Das Skript *Get-WmiAndQuery.ps1* umfasst folgenden Code.

GetWmiAndQuery.ps1

```
$strClass = "usb"
Get-WmiObject -List |
Where { $_.name -like "*$strClass*" } |
```

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

```
ForEach-Object -begin `
  {
    Write-Host "$strClass WMI-Aufzählung"
   Start-Sleep 3
   } `
-Process `
   {
     Get-wmiObject $_.name
   }
```

Im Skript *ProcessUsbHub.ps1* ruft das Cmdlet **Get-WmiObject** Instanzen der Klasse *Win32_USBHub* ab. Nachdem die USB Hub-Objekte aufgelistet wurden, können Sie die Objekte in das Cmdlet **ForEach-Object** einfügen. Vorschlag: Damit das Skript einfacher zu lesen ist, richten Sie die Parameter **-begin**, **-process** und **-end** am linken Rand des Skripts aus. Sie müssen die Zeilenfortsetzung jedoch durch ein Graviszeichen (`) angeben.

Tipp Die Umgebungsvariable *%COMPUTERNAME%* ist immer verfügbar und kann zum Extrahieren des Computernamens in einem Skript verwendet werden. Sie können den Wert der Variablen mit dem Cmdlet **Get-Item** ermitteln, das den Wert aus *env:\ psdrive* abruft. Die Eigenschaft *Value* enthält den Computernamen. Beispiel: *(Get-Item env:\computerName) value*.

Der Abschnitt **-begin** verwendet einen Codeblock, um den Computernamen unter Verwendung des Cmdlets **Write-Host** auszugeben. Mit einem Unterausdruck können Sie den Computernamen aus *env:\ psdrive* abrufen. Verwenden Sie die Variable %COMPUTERNAME% und extrahieren Sie den Wert.

Verwenden des Parameters Process

Verwenden Sie im Abschnitt **-process** das aktuelle Pipelineobjekt (mit *$_* angegeben), um die Eigenschaft *PnpDeviceID* der WMI-Klasse *Win32_USBHub* auszugeben. Geben Sie die Zeilenfortsetzung durch ein Graviszeichen an.

Verwenden des Parameters End

Der letzte Abschnitt des Skripts *ProcessUsbHub.ps1* enthält den Parameter **-end**. Mit dem Cmdlet **Write-Host** können Sie eine Zeichenfolge ausgeben, die angibt, dass der Befehl abgeschlossen wurde. Verwenden Sie einen Unterausdruck, um den vom Cmdlet **Get-Date** zurückgegebenen Wert auszugeben. Das Skript *ProcessUsbHub.ps1* umfasst folgenden Code:

ProcessUsbHub.ps1

```
Get-WmiObject win32_usbhub |
foreach-object `
-begin { Write-Host "USB-Hubs auf:" $(Get-Item env:\computerName).value } `
-process { $_.pnpDeviceID}  `
-end { Write-Host "Abschluss der Befehlsverarbeitung: $(get-date)" }
```

Verwenden der Anweisung *For*

Die Anweisung *for* wird ähnlich wie das Cmdlet **ForEach-Object** verwendet, um die Ausführung eines Skriptblocks zu steuern, wenn die angegebene Bedingung *True* ist. Die Anweisung *for* wird hauptsächlich zum mehrmaligen Ausführen einer Aktion verwendet. Beachten Sie in folgender Codezeile den Basiskonstruktor von *for*. Trennen Sie den Ausdruck, der überprüft wird, durch Klammern vom Codeblock in den geschweiften Klammern ab. Der überprüfte Ausdruck setzt sich aus drei Abschnitten zusammen. Der erste Abschnitt ist die Variable *$a*, der Sie den Wert 1 zuweisen. Der zweite Abschnitt enthält die Bedingung, die überprüft werden soll. Solange die Variable *$a* einen Wert von kleiner oder gleich 3 hat, wird der Befehl im Codeblockabschnitt ausgeführt. Der letzte Abschnitt des Ausdrucks fügt den Wert 1 zur Variablen *$a* hinzu. Der Codeblock umfasst einen einfachen Ausdruck zur Ausgabe des Worts *Hallo*.

```
for ($a = 1; $a -le 3 ; $a++) {"Hallo"}
```

Das Skript *PingARange.ps1* kann zum Pingen zahlreicher IP-Adressen verwendet werden und zeigt an, ob der Computer auf ICMP-Pakete (Internet Control Message Protocol) reagiert. Dies ist beim Ausführen von Suchvorgängen im Netzwerk hilfreich oder wenn Sie sicherstellen möchten, dass ein Computer im Netzwerk erreichbar ist. Die Variable *$intPing* ist auf 10 festgelegt und als ganze Zahl definiert. Die Variable *$intNetwork* ist als Zeichenfolge definiert und auf 127.0.0. festgelegt.

Die Anweisung *for* führt den übrigen Code entsprechend der Variablen *$intPing* aus. Die Indikatorvariable wird in der *for*-Anweisungszeile erstellt. Der Indikatorvariablen *$i* ist der Wert 1 zugewiesen. Wenn *$i* kleiner oder gleich dem Wert der Variablen *$intPing* ist, wird das Skript ausgeführt. Anschließend muss im Evaluierungsabschnitt der *for*-Anweisung der Wert 1 zu *$i* hinzugefügt werden.

Der Codeblock beginnt mit der geschweiften Klammer und verwendet die Variable *$strQuery* (die Zeichenfolge, in der die WMI-Abfrage gespeichert ist). Eine separate Variable vereinfacht die Verwendung von *$intNetwork* zusammen mit der Indikatorvariablen *$i*, um eine gültige IP-Adresse für die WMI-Abfrage zu erstellen.

In der Variablen *$wmi* werden die Objekte gespeichert, die das Cmdlet **Get-WmiObject** zurückgibt. Mit dem Argument **optional query** des Cmdlets **Get-WmiObject** können Sie eine WMI-Abfrage ausführen. Die Eigenschaft *StatusCode* enthält das Ergebnis der Ping-Aktion. Der Wert 0 zeigt die erfolgreiche Ausführung an. Alle anderen Werte geben an, dass die Ping-Aktion fehlgeschlagen ist. Mit einer *if … else-Anweisung* können Sie die Informationen übersichtlich darstellen, um die Eigenschaft *StatusCode* auszuwerten.

PingARange.ps1

```
[int]$intPing = 10
[string]$intNetwork = "127.0.0."

for ($i=1;$i -le $intPing; $i++)
{
$strQuery = "select * from win32_pingstatus where address = '" +
$intNetwork + $i + "'"
   $wmi = get-wmiobject -query $strQuery
   "Pingen von $intNetwork$i ... "
   if ($wmi.statuscode -eq 0)
      {"Erfolgreich"}
   else
      {"Fehler: " + $wmi.statuscode + " ist aufgetreten."}
}
```

Verwenden von Anweisungen für die Entscheidungsfindung

Das Treffen von Entscheidungen zum Steuern der Verzweigung in einem Skript ist eine fundamentale Methode. Tatsächlich handelt es sich hierbei um die Basis der Automatisierung. Eine Bedingung wird erkannt und ausgewertet, um die weitere Vorgehensweise festzulegen. Wenn Sie Ihre Logik in ein Skript übertragen können, sind Sie auf dem besten Wege, Ihre Server vollautomatisch zu überwachen. Welchen Vorgang führen Sie beispielsweise als Erstes aus, wenn Sie den Task-Manager manuell auf einem Server öffnen? Ich sortiere die Prozessliste oft nach der Speicherbelegung. Dieser Vorgang wird vom Skript *GetTopMemory.ps1* automatisiert.

GetTopMemory.ps1

```
Get-Process |
Sort-Object workingset -Descending |
Select-Object -First 5
```

Das Skript *GetTopMemory.ps1* spart Zeit beim Sortieren einer Liste. Aber was machen Sie als Nächstes? Brechen Sie einfach den ersten Prozess ab, der Speicher belegt? In diesem Fall muss keine Entscheidung getroffen werden. Möglicherweise ist es jedoch konstruktiver, ausschließlich die Benutzerprozesse abzubrechen, die mehr als 100 MB an Arbeitsspeicher belegen. Dies erfordert jedoch eine gewisse Entscheidungsfähigkeit. Untersuchen Sie zuerst die klassische Entscheidungsstruktur *if … elseif … else*.

Verwenden von *If … Elseif … Else*

Die grundlegendste Entscheidungsanweisung entspricht der Struktur *if … elseif … else*. Diese Struktur ist einfach zu verwenden, da sie völlig natürlich ist und in normale Konversationen einbezogen wird. Nehmen Sie die folgende Konversation zwischen zwei Touristen in Kopenhagen als Beispiel:

```
If ( sonnig und warm )
  { Fahrt nach NyHavn }
Elseif ( bewölkt und kalt )
  { Fahrt nach Tivoli }
Else
  {  Mit dem S-Tog nach Malmö }
```

Auch wenn Sie nicht Dänisch sprechen, können Sie dem Gespräch folgen. Wenn es sonnig und warm ist, fahren die Touristen nach NyHavn. Die erste Bedingungsauswertung ist, ob das Wetter sonnig und warm ist. Die Bedingung wird immer in Klammern eingeschlossen. Der Skriptblock, der ausgeführt wird, wenn die Bedingung zutrifft, ist in geschweiften Klammern eingeschlossen. Bei sonnigem und warmem Wetter reisen die Touristen nach NyHavn (eine schöne Hafenstadt mit vielen Straßencafes). Sollte das Wetter jedoch bewölkt und kalt sein, fahren sie nach Tivoli (ein Vergnügungspark in Kopenhagen). Wenn keine dieser Bedingungen zutrifft (es regnet oder schneit), fahren die Touristen zum Einkaufen mit dem Zug nach Malmö (eine Stadt in Schweden).

Um das Skript *GetServiceStatus.ps1* auszuführen, müssen Sie zuerst mit dem Cmdlet **Get-Service** alle Dienste auf dem Computer auflisten. Anschließend sortieren Sie die Liste mit dem Cmdlet **Sort-Object** basierend auf dem Dienststatus. Als Nächstes verwenden Sie eine *foreach*-Schleife, um die Dienste zu durchlaufen. Verwenden Sie beim Durchlaufen der Dienste *if … elseif … else*, um den Status auszuwerten. Wenn ein Dienst beendet wurde, zeigen Sie dessen Namen und den Status in Rot an. Wird der Dienst hingegen ausgeführt, zeigen Sie den Namen und den Status in Grün an. Für jeden anderen Status (beispielsweise Pause) ist der Name und der Status Gelb. Mittels einer solchen Entscheidungsmatrix

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

können Sie eine lange Dienstliste schnell überprüfen. Das Skript *GetServiceStatus.ps1* ist im Folgenden dargestellt. Die Farbwerte, die mit dem Cmdlet **Write-Host** verwendet werden können, sind in der nachfolgenden Tabelle aufgeführt.

GetServiceStatus.ps1

```
Get-Service |
Sort-Object status -descending |
foreach {
  if ( $_.status -eq "stopped")
   {Write-Host $_.name $_.status -ForegroundColor red}
  elseif ( $_.status -eq "running" )
   {Write-Host $_.name $_.status -ForegroundColor green}
  else
   {Write-Host $_.name $_.status -ForegroundColor yellow}
}
```

Black (Schwarz)	DarkBlue (Dunkelblau)	DarkGreen (Dunkelgrün)	DarkCyan (Dunkelblaugrün)
DarkRed (Dunkelrot)	DarkMagenta (Dunkelviolett)	DarkYellow (Dunkelgelb)	Gray (Grau)
DarkGray (Dunkelgrau)	Blue (Blau)	Green (Grün)	Cyan (Blaugrün)
Red (Rot)	Magenta (Violett)	Yellow (Gelb)	White (Weiß)

Verwenden von *Switch*

In anderen Programmiersprachen wird *switch* als *Select Case*-Anweisung bezeichnet. Die *switch*-Anweisung wird verwendet, um eine Bedingung gegen potenzielle Übereinstimmungen auszuwerten. Tatsächlich handelt es sich um eine vereinfachte *if … elseif*-Anweisung. Wenn Sie die *switch*-Anweisung verwenden, ist die auszuwertende Bedingung in Klammern eingeschlossen. Anschließend werden die Bedingungen im Codeblock in geschweifte Klammern gesetzt. Dies ist im folgenden Befehl dargestellt:

```
$a=5;switch ($a) { 4{"Vier erkannt."} 5{"Fünf erkannt."} }
```

Das folgende Skript *DisplayComputerRoles.ps1* beginnt mit der Variablen *$wmi*, die das vom Cmdlet **Get-WmiObject** zurückgegebene Objekt enthält. Die Eigenschaft *DomainRole* der Klasse *Win32_computersystem* wird als codierter Wert zurückgegeben. Damit die Ausgabe übersichtlicher ist, passt die *switch*-Anweisung den Wert der Eigenschaft *DomainRole* an den entsprechenden Textwert an.

DisplayComputerRoles.ps1

```
$wmi = get-wmiobject win32_computersystem
"Der Computer " + $wmi.name + " ist ein: "
switch ($wmi.domainrole)
   {
   0 {"`t Alleinstehende Arbeitsstation"}
   1 {"`t Arbeitsstation in einer Domäne"}
   2 {"`t Alleinstehender Server"}
   3 {"`t Mitgliedsserver in einer Domäne"}
   4 {"`t Sicherungsdomänencontroller"}
   5 {"`t Primärer Domänencontroller"}
   default {"`t Die Rolle kann nicht bestimmt werden."}
   }
```

Auswerten von Befehlszeilenargumenten

Switch eignet sich bestens zum Auswerten von Befehlszeilenargumenten. Im folgenden Skript *GetDriveArgs.ps1* können Sie mit der Funktion *funArg* den Wert der Variablen *$args* automatisch auswerten. Diese automatische Variable umfasst Argumente, die beim Ausführen des Skripts an die Befehlszeile übergeben werden. Die Variable ist für die Arbeit mit Befehlszeilenargumenten praktisch. *Switch* wertet den Wert von *$args* aus. In diesem Skript sind vier Parameterargumente zugelassen. Das Argument *all* führt eine WMI-Abfrage aus, um die Basisinformationen über die logischen Datenträger des Computers abzurufen. Das Argument **c** bewirkt nur die Ausgabe von Informationen über das Laufwerk C. Das Diskettenlaufwerk wird normalerweise zuerst aufgelistet und das zweite Element im Array ist Laufwerk C. Wenn dies auf Ihr System nicht zutrifft, können Sie die Reihenfolge ändern. Der Zweck des Skripts ist das Verdeutlichen der Verwendung von *switch*-Anweisungen, um Befehlszeilenargumente zu analysieren. Die Arrayelementnummer bietet eine praktische Methode zum Abrufen von WMI-Informationen in Windows PowerShell. Das Argument **free** gibt den freien Speicherplatz auf Laufwerk C zurück.

Das Argument **help** gibt eine Hilfeanweisung aus. Die Eingabe der Hilfemeldung wird durch **here-string** vereinfacht. Die Hilfemeldung gibt den Zweck des Skripts und mehrere Befehlszeilenbeispiele an.

GetDriveArgs.ps1

```
Function funArg()
{
switch ($args)
{
 "all" { gwmi win32_logicalDisk }
 "c"   { (gwmi win32_logicaldisk)[1] }
 "free" { (gwmi win32_logicaldisk)[1].freespace }
 "help" { $help = "
Das Skript gibt die Laufwerkinformationen für alle Laufwerke,
das Laufwerk C oder den freien Speicherplatz auf Laufwerk C aus.
Das Skript kann außerdem Hilfeinformationen anzeigen.
BEISPIEL:
 >GetDriveArgs.ps1 all
   Gibt Informationen zu allen Laufwerken aus.
 >GetDriveArgs.ps1 c
   Gibt nur Informationen zu Laufwerk C aus.
 >GetDriveArgs.ps1 free
   Gibt den freien Speicherplatz auf Laufwerk C aus.
" ; Write-Host $help }
}
}

#$args = "help"
funArg($args)
```

Verwenden von *Switch*-Platzhaltern

Einer der interessanteren Aspekte der *switch*-Anweisung ist die Verwendung von Platzhaltern. Dies ermöglicht das Schreiben von übersichtlichem und kompaktem Code, der leistungsfähig und einfach zu implementieren ist. Das Skript *SwitchIPConfig.ps1* enthält die Ergebnisse des Befehls **ipconfig /all** in der Variablen *$a*. Verwenden Sie *switch* mit dem Argument **-wildcard** und fügen Sie den zu analysierenden Text in Klammern ein. Öffnen Sie den Skriptblock mit den geschweiften Klammern und

geben Sie das anzupassende Schema ein. In diesem Fall handelt es sich um den einfachen Ausdruck **DHCP-Server**. Verwenden Sie im Skriptblock beim Feststellen einer Übereinstimmung das Cmdlet **Write-Host**, um die aktuelle Zeile im *switch*-Block auszugeben. Die automatische Variable *$switch* wird als der Enumerator verwendet. Geben Sie die aktuelle Eigenschaft an und rufen Sie die aktuell verarbeitete Zeile ab, um die zu überprüfende Zeile auszugeben. Das Skript *SwitchIPConfig.ps1* ist wie folgt aufgebaut.

SwitchIPConfig.ps1

```
$a = ipconfig /all

switch -wildCard ($a)
 {
   "*DHCP-Server*" { Write-Host $switch.current }
 }
```

Verwenden von *Switch* mit regulären Ausdrücken

Im Gegensatz zu einer normalen *Select Case*-Anweisung kann die *switch*-Anweisung reguläre Ausdrücke verarbeiten. Wenn Sie nach wichtigen Informationen suchen, können Sie mit der *switch*-Anweisung eine Textdatei öffnen, die Datei in den Arbeitsspeicher einlesen und anschließend unter Verwendung von regulären Ausdrücken analysieren. Reguläre Ausdrücke können sowohl zum Suchen von Wörtern und Ausdrücken als auch zum Überprüfen einer E-Mailadresse verwendet werden. Das Skript *SwitchRegEx.ps1* sucht in einer Textdatei nach den beiden Wörtern *Test* und *passend*. Wenn eines der Wörter gefunden wird, wird die Zeile ausgegeben, die das Wort enthält.

Nach der Anweisung *switch* können Sie den Parameter **-regex** angeben, um einen regulären Ausdruck zu verwenden. Der Wert in den Klammern ist ein Unterausdruck, mit dem eine Textdatei geöffnet und gelesen wird. Das *$* vor dem Dateipfad in geschweiften Klammern entspricht dem Befehl zum Öffnen und Lesen der Textdatei. Der Codeblock, der ausgeführt wird, wenn der reguläre Ausdruck gefunden wird, ist ebenfalls in geschweifte Klammern eingeschlossen (in diesem Beispiel das CmdLet **Write-Host**). Verwenden Sie den *$switch*-Enumerator, um die aktuelle Zeile mit der Übereinstimmung abzurufen.

SwitchRegEx.ps1

```
switch -regex (${c:\testa.txt})
{
 'Test' {Write-Host $switch.current}
 'passend' {Write-Host $switch.current}
}
```

Sie können die folgende Datei *TestA.txt* als Beispiel verwenden, um die Ausgabe des Skripts zu testen.

TestA.txt

```
Dies ist eine Testdatei.
Dies ist eine passende Datei.
Dies ist eine passende Testdatei.
```

Das Skript *VersionOfVista.ps1* ist möglicherweise ein besseres Beispiel für die Verwendung regulärer Ausdrücke in einer *switch*-Anweisung. Weisen Sie die Zeichenfolge *Version* der Variablen *$strPattern* zu und speichern Sie die Ausgabe des Befehls **net config workstation** in der Variablen *$text*. Geben Sie anschließend den Parameter **-regex** in der *switch*-Anweisung an, übertragen Sie diese in den in der Variablen *$text* gespeicherten Inhalt und suchen Sie das in der Variablen *$strPattern* gespeicherte

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Textmuster. Nachdem Sie das Textmuster gefunden haben, geben Sie die entsprechende Zeile mit der aktuellen Eigenschaft der automatischen Variablen *$switch* aus. Dieses Skript zeigt an, welche Version von Windows Vista ausgeführt wird. Die Ausgabe des Befehls **net config workstation** ist 19 Zeilen lang. Vergleichen Sie die Ergebnisse mit der Ausgabe des Skripts **VersionOfVista.ps1**:

```
Softwareversion                     Windows Vista (TM) Enterprise
```

VersionOfVista.ps1

```
$strPattern = "Version"
$text = net config workstation

switch -regex ($text)
{
  $strPattern { Write-Host $switch.current }
}
```

Arbeiten mit Datentypen

Windows PowerShell basiert auf einer typenorientierten Sprache, obwohl diese wie eine typenlose Sprache verwendet wird. Windows PowerShell erkennt die Datentypen und verarbeitet diese entsprechend. Eine Zeichenfolge wird beispielsweise von Windows PowerShell automatisch als Zeichenfolge behandelt. Im Folgenden sind drei Beispielanweisungen dargestellt.

```
PS C:\> 1 + 1
2
PS C:\> 12:00 + :30
Unerwartetes Token ":00" im Ausdruck oder in der Anweisung.
Bei Zeile:1 Zeichen:6
+ 12:00  <<<< + :30
PS C:\> a + b
Die Benennung "a+b" wurde nicht als Cmdlet, Funktion, ausführbares Programm oder Skriptdatei erkannt.
Überprüfen Sie die Benennung, und versuchen Sie es erneut.
Bei Zeile:1 Zeichen:2
+ a  <<<< + b
PS C:\>
```

Beachten Sie, dass nur die Anweisung **1 + 1** ohne Fehler ausgeführt wurde. Windows PowerShell erkennt die Zahlen korrekt und läßt die Addition zu. Buchstaben oder Zeitangaben können jedoch nicht addiert werden.

Wenn Sie die Buchstaben *a* und *b* jedoch in doppelte Anführungszeichen setzen, werden diese aneinandergefügt. Beispiel:

```
PS C:\> "a" + "b"
ab
```

Dieses Verhalten ist nicht überraschend, sondern erwartet. Die doppelten Anführungszeichen wandeln die Buchstaben *a* und *b* in Zeichenfolgenwerte um und verbinden die beiden Buchstaben. Um dies zu veranschaulichen, fügen Sie den Buchstaben *a* in das Cmdlet **Get-Member** ein. Beachten Sie, dass die erste Zeile der Ausgabe anzeigt, dass der Buchstabe *a* ein Objekt des Typs *system.string* ist. Objekte vom Typ *system.string* bieten zahlreiche Eigenschaften und Methoden.

```
PS C:\> "a" | get-member


   TypeName: System.String
```

```
Name             MemberType           Definition
----             ----------           ----------
Clone            Method               System.Object Clone()
System.Int32 CompareTo(String strB)
Contains         Method               System.Boolean Contains(String value)
CopyTo           Method               System.Void CopyTo(Int32 sourceIndex, Char[]
destination, Int32 destinationIn
EndsWith         Method               System.Boolean EndsWith(String value),
System.Boolean EndsWith(String value,
Equals           Method               System.Boolean Equals(Object obj),
System.Boolean Equals(String value), Syste...
GetEnumerator    Method               System.CharEnumerator GetEnumerator()
GetHashCode      Method               System.Int32 GetHashCode()
GetType          Method               System.Type GetType()
GetTypeCode      Method               System.TypeCode GetTypeCode()
get_Chars        Method               System.Char get_Chars(Int32 index)
get_Length       Method               System.Int32 get_Length()
IndexOf          Method               System.Int32 IndexOf(Char value, Int32
startIndex, Int32 count), System.Int32...
IndexOfAny       Method               System.Int32 IndexOfAny(Char[] anyOf, Int32
startIndex, Int32 count), System....
Insert           Method               System.String Insert(Int32 startIndex, String
value)
IsNormalized     Method               System.Boolean IsNormalized(), System.Boolean
IsNormalized(NormalizationForm
LastIndexOf      Method               System.Int32 LastIndexOf(Char value, Int32
startIndex, Int32 count), System.I...
LastIndexOfAny   Method               System.Int32 LastIndexOfAny(Char[] anyOf, Int32 startIndex, Int32 count), Sys
...
Normalize        Method               System.String Normalize(), System.String
Normalize(NormalizationForm normaliz...
PadLeft          Method               System.String PadLeft(Int32 totalWidth),
System.String PadLeft(Int32 totalWid...
PadRight         Method               System.String PadRight(Int32 totalWidth),
System.String PadRight(Int32 totalW...
Remove           Method               System.String Remove(Int32 startIndex, Int32
count), System.String Remove(Int...
Replace          Method               System.String Replace(Char oldChar, Char
newChar), System.String Replace(Stri...
Split            Method               System.String[] Split(Params Char[]
separator), System.String[] Split(Char[] ...
StartsWith       Method               System.Boolean StartsWith(String value),
System.Boolean StartsWith(String val...
Substring        Method               System.String Substring(Int32 startIndex),
System.String Substring(Int32 star...
ToCharArray      Method               System.Char[] ToCharArray(), System.Char[]
ToCharArray(Int32 startIndex, Int3...
ToLower          Method               System.String ToLower(), System.String
ToLower(CultureInfo culture)
ToLowerInvariant Method               System.String ToLowerInvariant()
ToString         Method               System.String ToString(), System.String
ToString(IFormatProvider provider)
ToUpper          Method               System.String ToUpper(), System.String
ToUpper(CultureInfo culture)
ToUpperInvariant Method               System.String ToUpperInvariant()
```

```
Trim            Method               System.String Trim(Params Char[] trimChars),
System.String Trim()
TrimEnd         Method               System.String TrimEnd(Params Char[]
trimChars)
TrimStart       Method               System.String TrimStart(Params Char[]
trimChars)
Chars           ParameterizedProperty System.Char Chars(Int32 index) {get
```

Wenn Sie die Zahl 1 in das Cmdlet **Get-Member** einfügen, wird angezeigt, dass es sich um ein *system.int32*-Objekt handelt. Die Objektklasse *system.int32* bietet weniger Methoden als die Objektklasse *system.string*:

```
PS C:\> 1 | get-member


   TypeName: System.Int32

Name        MemberType Definition
----        ---------- ----------
CompareTo   Method     System.Int32 CompareTo(Int32 value), System.Int32
CompareTo(Object value)
Equals      Method     System.Boolean Equals(Object obj), System.Boolean
Equals(Int32 obj)
GetHashCode Method     System.Int32 GetHashCode()
GetType     Method     System.Type GetType()
GetTypeCode Method     System.TypeCode GetTypeCode()
ToString    Method     System.String ToString(), System.String
ToString(IFormatProvider provider), System.String ToS...
```

Wenn Sie mit **Get-Member** die Ursache des Objektverhaltens überprüfen, können Sie den Datentyp eines Objekts unter Verwendung einer Typeinschränkung unmittelbar festlegen. Um zum Beispiel 12:00 als ein *datetime*-Objekt zu interpretieren, wandeln Sie die Zeichenfolge *12:00* mit der Typeinschränkung *[datetime]* in ein *datetime*-Objekt um. Beispiel:

```
PS C:\> [datetime]"12:00" | get-member


   TypeName: System.DateTime

Name                 MemberType     Definition
----                 ----------     ----------
Add                  Method         System.DateTime Add(TimeSpan value)
AddDays              Method         System.DateTime AddDays(Double value)
AddHours             Method         System.DateTime AddHours(Double value)
AddMilliseconds      Method         System.DateTime AddMilliseconds(Double value)
AddMinutes           Method         System.DateTime AddMinutes(Double value)
AddMonths            Method         System.DateTime AddMonths(Int32 months)
AddSeconds           Method         System.DateTime AddSeconds(Double value)
AddTicks             Method         System.DateTime AddTicks(Int64 value)
AddYears             Method         System.DateTime AddYears(Int32 value)
CompareTo            Method         System.Int32 CompareTo(Object value),
System.Int32 CompareTo(DateTime value)
Equals               Method         System.Boolean Equals(Object value),
System.Boolean Equals(DateTime value)
GetDateTimeFormats   Method         System.String[] GetDateTimeFormats(),
System.String[] GetDateTimeFormats(IFormat...
GetHashCode          Method         System.Int32 GetHashCode()
```

```
GetType              Method         System.Type GetType()
GetTypeCode          Method         System.TypeCode GetTypeCode()
get_Date             Method         System.DateTime get_Date()
get_Day              Method         System.Int32 get_Day()
get_DayOfWeek        Method         System.DayOfWeek get_DayOfWeek()
get_DayOfYear        Method         System.Int32 get_DayOfYear()
get_Hour             Method         System.Int32 get_Hour()
get_Kind             Method         System.DateTimeKind get_Kind()
get_Millisecond      Method         System.Int32 get_Millisecond()
get_Minute           Method         System.Int32 get_Minute()
get_Month            Method         System.Int32 get_Month()
get_Second           Method         System.Int32 get_Second()
get_Ticks            Method         System.Int64 get_Ticks()
get_TimeOfDay        Method         System.TimeSpan get_TimeOfDay()
get_Year             Method         System.Int32 get_Year()
IsDaylightSavingTime Method         System.Boolean IsDaylightSavingTime()
Subtract             Method         System.TimeSpan Subtract(DateTime value),
System.DateTime Subtract(TimeSpan value)
ToBinary             Method         System.Int64 ToBinary()
ToFileTime           Method         System.Int64 ToFileTime()
ToFileTimeUtc        Method         System.Int64 ToFileTimeUtc()
ToLocalTime          Method         System.DateTime ToLocalTime()
ToLongDateString     Method         System.String ToLongDateString()
ToLongTimeString     Method         System.String ToLongTimeString()
ToOADate             Method         System.Double ToOADate()
ToShortDateString    Method         System.String ToShortDateString()
ToShortTimeString    Method         System.String ToShortTimeString()
ToString             Method         System.String ToString(), System.String
ToString(String format), System.String T...
ToUniversalTime      Method         System.DateTime ToUniversalTime()
Date                 Property       System.DateTime Date {get;}
Day                  Property       System.Int32 Day {get;}
DayOfWeek            Property       System.DayOfWeek DayOfWeek {get;}
DayOfYear            Property       System.Int32 DayOfYear {get;}
Hour                 Property       System.Int32 Hour {get;}
Kind                 Property       System.DateTimeKind Kind {get;}
Millisecond          Property       System.Int32 Millisecond {get;}Property
 System.Int32 Minute {get;}
Month                Property       System.Int32 Month {get;}
Second               Property       System.Int32 Second {get;}
Ticks                Property       System.Int64 Ticks {get;}
TimeOfDay            Property       System.TimeSpan TimeOfDay {get;}
Year                 Property       System.Int32 Year {get;}
DateTime             ScriptProperty System.Object DateTime {get=if
($this.DisplayHint -ieq  "Date")...
```

Sie müssen **Get-Member** nicht verwenden, um den Datentyp eines bestimmten Objekts zu bestimmen, wenn Sie nur den Namen des Objekts anzeigen möchten. Verwenden Sie hierzu die Methode *getType()*. Im ersten Fall bestätigen Sie, dass *12:00* eine Zeichenfolge ist. Im zweiten Fall wandeln Sie die Zeichenfolge in den Datentyp *datetime* um und bestätigen diesen mit der Methode *getType()*:

```
PS C:\> "12:00".getType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     String                                   System.Object
```

```
PS C:\> ([dateTime]"12:00").getType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     DateTime                                 System.ValueType
```

Sie finden diese Befehle in der Datei *DataTypes.txt* im Ordner *Chapter02* auf der Begleit-CD. In der Tabelle 2.3 sind weitere Bezeichner für Datentypen aufgeführt.

Tabelle 2.3 Aliasnamen für Datentypen

Alias	Typ
[int]	32 Bit signierte ganze Zahl
[long]	64 Bit signierte ganze Zahl
[string]	Zeichenfolge fester Länge aus Unicode-Zeichen
[char]	Ein 16 Bit Unicode-Zeichen
[bool]	True/False-Wert
[byte]	8 Bit nicht signierte ganze Zahl
[double]	64 Bit Gleitkommazahl mit doppelter Genauigkeit
[datetime]	DateTime-Datentyp
[decimal]	Ein dezimaler 128 Bit Wert
[single]	32 Bit Gleitkommazahl mit einfacher Genauigkeit
[array]	Ein Array aus Werten
[xml]	*Xml*-Objekte
[hashtable]	Ein *hashtable*-Objekt (einem *dictionary*-Objekt ähnlich)

Die Leistungsfähigkeit regulärer Ausdrücke

Eines der interessanten Features von Windows PowerShell ist der Einsatz regulärer Ausdrücke. Reguläre Ausdrücke sind insbesondere für die Textverarbeitung ausgelegt. Sie können mit dem Parameter **-wildcard** genauso wie mit einem regulären Ausdruck unter Verwendung einer *switch*-Anweisung nach einem bestimmten Wort suchen. Im Folgenden sind einige fortgeschrittenere Aufgaben erklärt, die Sie mit regulären Ausdrücken ausführen können. In Tabelle 2.4 sind die Escape-Zeichenfolgen aufgeführt, die mit regulären Ausdrücken verwendet werden können.

Tabelle 2.4 Escape-Zeichenfolgen

Zeichen	Beschreibung
Normale Zeichen	Diese Zeichen, außer . $ ^ { [(|) * + ? \, entsprechen sich selbst
\a	Entspricht einer Warnung \u0007
\b	Entspricht einem Rückschritt \u0008 in einer []-Zeichenklasse in einem regulären Ausdruck; \b ist eine Wortgrenze
\t	Entspricht einem Tabzeichen \u0009

Tabelle 2.4 Escape-Zeichenfolgen *(Fortsetzung)*

Zeichen	Beschreibung
\r	Entspricht einer Zeilenumschaltung \u000D
\v	Entspricht einem vertikalen Tabzeichen \u000B
\f	Entspricht einem Seitenvorschub \u000C
\n	Entspricht einer neuen Zeile \u000A
\e	Entspricht einem Escape-Zeichen \u001B
\040	Entspricht einem ASCII-Zeichen als Oktal (bis zu drei Ziffern); Zahlen ohne führende Null sind Rückverweise, wenn sie aus nur einer Ziffer bestehen oder einer Gruppenzahl entsprechen. Beispielsweise repräsentiert das Zeichen \040 ein Leerzeichen.
\x20	Entspricht einem ASCII-Zeichen unter Verwendung der hexadezimalen Darstellung (zwei Ziffern)
\cC	Entspricht einem ASCII-Steuerzeichen, beispielsweise ist \cC Control-C
\u0020	Entspricht einem Unicode-Zeichen unter Verwendung der hexadezimalen Darstellung (zwei Ziffern)

Das Skript *RegExTab.ps1* verwendet eine Escape-Zeichenfolge in einem regulären Ausdruck. Dieses Skript öffnet eine Textdatei und sucht nach Tabzeichen. Die einfachste Methode bei der Arbeit mit regulären Ausdrücken ist das Speichern des Textmusters in einer Variablen. Sie können das Skript ändern und ausprobieren (kommentieren Sie die Zeile mit dem #-Zeichen aus und erstellen Sie eine neue Zeile mit dem gleichen Namen und einem anderen Wert).

Das Skript *RegExTab.ps1* gibt \t als Textmuster an. Gemäß Tabelle 2.4 wird nach Tabzeichen gesucht. Weisen Sie das Textmusters der Variablen *$strPattern* zu und geben Sie dabei die Typeneinschränkung *[regex]* an:

```
$regex = [regex]$strPattern
```

Speichern Sie unter Verwendung der folgenden Syntax den Inhalt der Datei *TabLine.txt* in der Variablen *$text*:

```
$text = ${C:\Chapter02\Tabline.txt}
```

Analysieren Sie die Textdatei mit der *matches*-Methode und suchen Sie nach den Übereinstimmungen, die im Textmuster *$strPattern* angegeben sind. Beachten Sie, dass Sie das im *regular expression*-Objekt verwendete Textmuster bereits der Variablen *$regex* zugeordnet haben. Zählen Sie die gefundenen Übereinstimmungen. Das Skript *RegExTab.ps1* ist wie folgt aufgebaut:

RegExTab.ps1

```
$strPattern = "\t"
$regex = [regex]$strPattern

$text = ${C:\Chapter02\Tabline.txt}

$mc = $regex.matches($text)
$mc.count
```

In Tabelle 2.5 sind die Zeichen aufgeführt, die mit regulären Ausdrücken für erweiterte Textmusterübereinstimmungen verwendet werden können.

Tabelle 2.5 Zeichenschemas

Zeichen	Beschreibung
[Zeichengruppe]	Entspricht den Zeichen in der angegebenen Zeichengruppe. *[aeiou]* gibt alle Vokale an. *[\p{P}\d]* gibt alle Interpunktionszeichen und dezimalen Ziffern an.
[^Zeichengruppe]	Entspricht den Zeichen in der angegebenen Zeichengruppe. [^aeiou] gibt alle Konsonanten an. [^\p{P}\d] gibt alle Zeichen an, außer Interpunktionszeichen und dezimale Ziffern.
[erstesZeichen-letztesZeichen]	Entspricht allen Zeichen in einem Zeichenbereich. *[0-9a-fA-F]* gibt dezimale Ziffern (0 bis 9), Kleinbuchstaben (a bis f) und Großbuchstaben (A bis F) an.
.	Entspricht allen Zeichen außer \n. Bei Änderung der *Singleline*-Option entspricht der Punkt allen Zeichen.
\p{Name}	Entspricht allen Zeichen in der allgemeinen Unicode-Kategorie oder einem benannten Block (beispielsweise *Ll*, *Nd*, *Z*, *IsGreek* und *IsBoxDrawing*).
\P{Name}	Entspricht allen Zeichen außerhalb der allgemeinen Unicode-Kategorie oder einem benannten Block.
\w	Entspricht allen Wortzeichen. Gleichbedeutend mit den allgemeinen Unicode-Kategorien *[\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}\p{Lm}]*. Wenn das ECMAScript-kompatible Verhalten mit der *ECMAScript*-Option festgelegt wird, entspricht \w dem Schema *[a-zA-Z_0-9]*.
\W	Entspricht allen nicht-Wortzeichen. Gleichbedeutend mit den allgemeinen Unicode-Kategorien *[^\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}\p{Lm}]*. Wenn das ECMAScript-kompatible Verhalten mit der *ECMAScript*-Option festgelegt wird, entspricht \W dem Schema *[^a-zA-Z_0-9]*.
\s	Entspricht allen Leerzeichen. Gleichbedeutend mit den Escape-Zeichenfolgen und allgemeinen Unicode-Kategorien *[\f\n\r\t\v\x85\p{Z}]*. Wenn das ECMAScript-kompatible Verhalten mit der *ECMAScript*-Option festgelegt wird, entspricht \s dem Schema *[\f\n\r\t\v]*.
\S	Entspricht allen nicht-Leerzeichen. Gleichbedeutend mit den Escape-Zeichenfolgen und allgemeinen Unicode-Kategorien *[^\f\n\r\t\v\x85\p{Z}]*. Wenn das ECMAScript-kompatible Verhalten mit der *ECMAScript*-Option festgelegt wird, entspricht \S dem Schema *[^ \f\n\r\t\v]*.
\d	Entspricht allen dezimalen Ziffern. Gleichbedeutend mit *\p{Nd}* für Unicode und *[0-9]* nicht-Unicode, ECMAScript-Verhalten.
\D	Entspricht allen nicht-Ziffernzeichen. Gleichbedeutend mit *\P{Nd}* für Unicode und *[^0-9]* nicht-Unicode, ECMAScript-Verhalten.

Um beispielsweise die Leerzeichen in einer Datei zu identifizieren, verwenden Sie das Textmuster *\s*, das in Tabelle 2.5 als Zeichenschema aufgeführt ist. Das Suchen von Leerzeichen in einer Textdatei ist hilfreich, da das Zeilenende häufig nur durch ein Leerzeichen markiert ist. Das folgende Skript *RegWhiteSpace.ps1* veranschaulicht die Verwendung von Leerzeichen.

Die erste Zeile des Skripts enthält Text. Das Textmuster gibt ein einfaches *\s* an, das laut Tabelle 2.5 einem Leerzeichen entspricht. Verwenden Sie die Variable *$matches* zum Speichern des *match*-Objekts, das von der statischen Methode *match* zurückgegeben wird.

Nach der Ausgabe der Ergebnisse setzen Sie die Verarbeitung unter Verwendung des gleichen Textmusters fort. Verwenden Sie die *replace*-Methode, um das Textmuster im Text von *$strText* durch einen Unterstrich (_) zu ersetzen. Geben Sie danach den Wert von *$strReplace* aus, der nun das geänderte Objekt enthält.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

RegWhiteSpace.ps1

```
$strText = "Eine nette kleine Textzeile, die wir nach einem Ausdruck durchsuchen können."
$Pattern = "\s"
$matches = [regex]::match($strText, $pattern)

"Als Ergebnis der match-Methode erhalten wir folgendes Resultat:"
$matches

$strReplace = [regex]::replace($strText, $pattern, "_")
"Nun führen wir die replace-Methode unter Verwendung des gleichen Textmusters aus.
Dabei verwenden wir einen Unterstrich, um die Leerzeichen zwischen den Worten zu ersetzen:"

$strReplace
```

Verwenden von Befehlszeilenargumenten

Das Ändern von Skripts zur Laufzeit spart Zeit und Arbeit. In vielen Unternehmen ist der technische Support jedoch nicht für die Erstellung oder Anpassung von Skripts zuständig. Die Supportmitarbeiter haben keinen Zugriff auf Skript-Editoren und können Skripts nicht zur Entwurfszeit ändern. Die Lösung besteht in der Verwendung von Befehlszeilenargumenten, die das Verhalten des Skripts ändern. Im Prinzip verhalten sich Skripts wie Dienstprogramme, die jedoch von einem Benutzer angepasst werden können, anstatt von Komponenten, die mit Parametern und Befehlen geändert werden. Das Beispielskript *ArgsShare.ps1* veranschaulicht diese Tatsache.

Das Skript *ArgsShare.ps1* definiert eine einfache Funktion zum Ausführen einer WMI-Abfrage. Die Funktion bestimmt beim Ausführen des Skripts anhand des Befehlszeilenarguments, welcher Freigabetyp zurückgegeben werden soll.

In einer *if … else*-Anweisung wird das Vorhandensein eines Befehlszeilenarguments überprüft. Kann kein Argument gefunden werden, wird eine Hilfemeldung angezeigt. Jede Eingabe, die nicht als gültiges Argument erkannt wird, resultiert in einer Hilfemeldung. Die Hilfemeldung kann aber auch mit einem Fragezeichen als Parameter direkt angezeigt werden.

Nachdem bestätigt wurde, dass ein gültiges Befehlszeilenargument existiert, ordnet die *switch*-Anweisung der Variablen *$strShare* einen entsprechenden Wert zu und ruft die WMI-Funktion auf. Dieses Verfahren ermöglicht dem Benutzer die Eingabe eines einfachen Wortes, beispielsweise *admin*, *drucker, datei, ipc* oder *alle*, um die entsprechende WMI-Abfrage auszuführen. WMI erwartet jedoch eine gültige ganze Zahl für den Freigabetyp. Mit *switch* wird die entsprechende WMI-Abfrage basierend auf der Befehlseingabe generiert. Bei Eingabe eines unerwarteten Befehlszeilenarguments wird der Standardparameter verwendet und die Hilfemeldung ausgegeben. Sie können auch eine *all*-Abfrage oder eine andere WMI-Standardabfrage ausführen. Es ist möglich, die WMI-Standardabfrage in die *if(!args)*-Abfrage einzufügen und die Standardabfrage auszuführen, wenn kein Argument vorhanden ist. Dieser Vorgang simuliert das Verhalten einiger Windows-Befehlszeilenprogramme. Das Skript *ArgsShare.ps1* ist wie folgt aufgebaut:

ArgsShare.ps1

```
Function FunWMI($strShare)
 {
  Get-WmiObject win32_share -Filter "type = $strShare"
 }
```

```
if(!$args)
{ "Sie müssen ein Befehlszeilenargument angeben. Beispiel: ArgsShare.ps1 ?"}
ELSE
{
$strShare = $args
switch ($strShare)
  {
   "admin" { $strShare = 2147483648 ; funwmi($strShare) }
   "drucker" { $strShare = 2147483649 ; funwmi($strShare) }
   "datei"  { $strShare = 0 ; funwmi($strShare) }
   "ipc"   { $strShare = 2147483651 ; funwmi($strShare) }
   "alle"   { Get-WmiObject win32_share }
   Default { Write-Host "Folgende Argumente werden unterstützt: admin, drucker, datei, ipc, alle `n
             Beispiel: > ArgsShare.ps1 admin" }
  }
}
```

Zusammenfassung

In diesem Kapitel wurden Skriptrichtlinien für Windows PowerShell, die Konfiguration von Windows PowerShell für die Verwendung von Skripts, die Flusskontrollanweisungen und deren Verwendung in Skripts erklärt. Außerdem wurde die Implementierung der Verarbeitungsverzweigung anhand von Entscheidungen in Windows PowerShell und die Vereinfachung von Netzwerkverwaltungsaufgaben sowie die Erledigung von Routineaufgaben unter Verwendung von Skripts beschrieben. Am Ende des Kapitels haben Sie sich mit regulären Ausdrücken für erweiterte Übereinstimmungsfunktionen in Skripts und Cmdlets vertraut gemacht.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

KAPITEL 3

Verwalten von Protokollen

Nach Abschluss dieses Kapitels können Sie:

- Das Ereignisprotokoll lesen
- Allgemeine Protokolldateien auswerten
- Das Ereignisprotokoll verwalten und durchsuchen
- Die WMI-Ereignisprotokolle überprüfen
- Ereignisse in den Ereignisprotokollen aufzeichnen
- Benutzerdefinierte Ereignisprotokolle erstellen

Auf der Begleit-CD Alle Skripts in diesem Kapitel befinden sich auf der Begleit-CD im Ordner *\Scripts\Chapter03*.

In der Vergangenheit waren Benutzer hauptsächlich auf die drei wichtigsten Protokolle der zahlreichen Windows-Ereignisprotokolle beschränkt: *Anwendung*, *System* und *Sicherheit*. In Windows Vista and Windows Server 2008 stehen jedoch viele neue Protokolle zur Verfügung, die umfangreiche Informationen enthalten. Das Überprüfen der Ereignisprotokolle kann mit Ausgleichssport verglichen werden: Sie haben einfach nicht immer die Zeit dafür.

Identifizieren der Ereignisprotokolle

Mit Windows Vista und Windows Server 2008 hat es die Ereignisprotokollierung endlich in das 21. Jahrhundert geschafft. Die drei wichtigsten Protokolle umfassen nun neue Optionen und das Cmdlet **Get-EventLog** zeigt an, welche Ereignisprotokolle aktiviert sind.

GetEventLogs.ps1

```
Get-EventLog -List
```

Das Skript **GetEventLog** listet die Ereignisprotokolle auf dem Bildschirm auf. Die Liste bietet eine hervorragende Übersicht der maximalen Größe der Ereignisprotokolle, der Anzahl der Protokolleinträge sowie der Beibehaltungs- und Überschreibungsrichtlinien. Beispiel:

```
Max(K) Retain OverflowAction       Entries Name
------ ------ --------------       ------- ----
15,168      0 OverwriteAsNeeded      7,318 Anwendung
15,168      0 OverwriteAsNeeded          0 DFS-Replikation
20,480      0 OverwriteAsNeeded          0 Hardware-Ereignisse
   512      7 OverwriteOlder             0 Internet Explorer
   512      7 OverwriteOlder             0 Key Management Service
16,384      0 OverwriteAsNeeded          0 Microsoft Office Diagnostics
```

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

```
16,384      0 OverwriteAsNeeded         495 Microsoft Office Sessions
30,016      0 OverwriteAsNeeded      48,462 Sicherheit
15,168      0 OverwriteAsNeeded      23,109 System
15,360      0 OverwriteAsNeeded       1,919 Windows PowerShell
```

Lesen der Ereignisprotokolle

Nachdem Sie den Befehl **Get-EventLog -list** ausgeführt haben, um die auf dem Computer installierten Ereignisprotokolle zu ermitteln, können Sie **Get-EventLog** verwenden, um die Ereignisprotokolle zu lesen. Meist ist es am einfachsten, den englischen Namen des Ereignisprotokolls an das CmdLet **Get-EventLog** zu übergeben. Das Skript *GetApplicationEventLog.ps1* führt diesen Vorgang aus:

GetApplicationEventLog.ps1

```
Get-EventLog application
```

Dieser Befehl zeigt den gesamten Inhalt des Ereignisprotokolls auf dem Bildschirm an. Das Skript *GetApplicationEventLog.ps1* besteht aus einer einzigen Zeile. Das Speichern des Befehls als Skript ermöglicht das spätere Hinzufügen weiterer Befehle. Wenn Sie das Skript mit diesem Befehl in der Windows PowerShell-Konsole ausführen, wird folgende Ausgabe angezeigt:

```
PS C:\> Get-EventLog application

Index Time          Type Source                Eventid Message
----- ----          ---- ------                ------- -------
 7705 Mai 25 08:42  Info Software Licensin...      8196 Der Lizenzaktivierungsplaner (SLUINotify.dll) ...
 7704 Mai 25 08:42  Info Software Licensin...       902 Der Softwarelizenzierungsdienst wurde ges...
 7703 Mai 25 08:40  Winlogon                      4104 Auf Windows wird im Benachrichtigungszeit...
 7701 Mai 25 08:37  Iprofsvc                      1531 Der Benutzerprofildienst wurde erfolgreic...
 7702 Mai 25 08:37  Info Desktop Window Ma...      9009 Der Desktopfenster-Manager wurde mit dem ...
 7700 Mai 25 08:36  Info Info Software Licensin... 1003 Softwarelizenzierungsdienst hat die Überp ...
```

Den Bildlauf mit so umfangreichem Text in Windows PowerShell auszuführen ist mühsam. Obwohl die Ausgabe beeindruckend ist, ist sie für die meisten Benutzer nahezu nutzlos. Damit die Informationen brauchbar sind, müssen Sie die Ausgabe anpassen. Geeignete Textverarbeitungsmethoden werden später in diesem Kapitel erklärt.

Exportieren in eine Textdatei

Sie können die Ausgabe der vielen auf dem Bildschirm angezeigten Protokolleinträge in eine Textdatei umleiten. Sie führen diesen Vorgang mit dem Skript *WriteAppLogToText.ps1* aus:

WriteAppLogToText.ps1

```
Get-EventLog application > c:\fso\applog.txt
```

Die Textdatei ist in Abbildung 3.1 dargestellt. Nachdem Sie die Ausgabe in einer Textdatei gespeichert haben, können Sie nach bestimmten Elementen in der Protokolldatei suchen.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

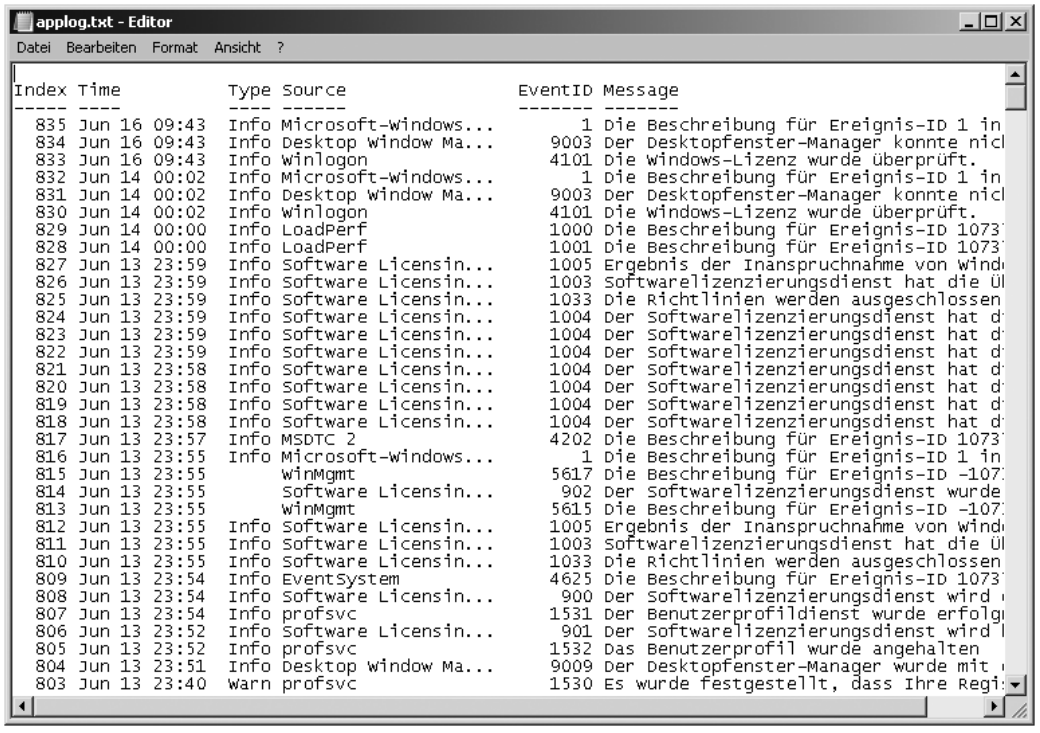


Abbildung 3.1 Ein exportiertes Anwendungsprotokoll in *Notepad.exe*

Diese Methode ist zwar in beschränktem Umfang hilfreich, aber die Textverarbeitungsfunktionen der *switch*-Anweisung stellen eine interessantere Lösung dar. Dies ist mit dem folgenden Skript *ParseAppTextLog.ps1* veranschaulicht. Sie können beispielsweise die Eintragstypen in einem Ereignisprotokoll zählen oder ein komplexeres Skript ausführen, das reguläre Ausdrücke verwendet.

ParseAppTextLog.ps1 initialisiert die Variable *$strLog*, die den Pfad zum exportierten Ereignisprotokoll enthält. Initialisieren Sie anschließend die Indikatorvariablen *$e, $i* und *$w.* Syntax:

```
$e=$i=$w=0
```

Nachdem Sie die Indikatorvariablen auf 0 festgelegt haben, verwenden Sie die *switch*-Anweisung. Mit den erweiterten Features von *switch* können Sie beispielsweise die Ausgabe mit dem Argument *-file* in eine Textdatei umleiten oder mit dem Argument *-wildcard* eine Platzhaltersuche ausführen. *Switch* durchsucht den Inhalt der Textdatei nach Zeichenfolgen, die das Wort *error* enthalten. Wenn das Wort *error* gefunden wird, wird der Wert von *$e* um 1 erhöht. Wenn *info* in einer Zeichenfolge gefunden wird, wird der Wert von *$i* ebenfalls um 1 erhöht. Außerdem wird nach allen Übereinstimmungen des Wortes *warn* gesucht. Wenn eine Übereinstimmung gefunden wird, wird der Wert von *$w* um 1 erhöht.

Nachdem Sie die Datei *Applog.txt* durchsucht haben, können Sie mit einem Skript unter Verwendung des Cmdlets **Write-Output** eine Zusammenfassung ausgeben. Das Skript gibt den in der Variablen *$strLog* angegebenen Pfad sowie Fehler-, Informations- und Warnungsmeldungen aus.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

ParseAppTextLog.ps1

```
$strLog = "c:\FSO\Applog.txt"
$e=$i=$w=0

switch -wildcard -file $strLog {
"*error*" { $e++ }
"*info*"  { $i++ }
"*warn*"  { $w++ }
}
Write-Output "
$strLog umfasst folgende Einträge:
      Fehler      $e
      Warnungen    $w
      Informationen $i
"
```

Exportieren in eine XML-Datei

Ein interessante Methode zum Bearbeiten langer Textpassagen ist das Exportieren des Ereignisprotokolls mit dem Cmdlet **Export-Clixml** in eine XML-Datei (Extensible Markup Language). Diese Methode ist im Beispielskript *WriteAppLogToXML.ps1* veranschaulicht. Um das Skript auszuführen, rufen Sie zuerst das Objekt für das aktuelle Anwendungsprotokoll ab. Verwenden Sie hierzu das Cmdlet **Get-EventLog** und geben Sie den Namen des Ereignisprotokolls an. In diesem Beispiel wird das Anwendungsprotokoll abgerufen. Übergeben Sie dann die Ausgabe des Cmdlets **Get-EventLog** an das Cmdlet **Export-Clixml**. Geben Sie unter Verwendung des Arguments **-path** des Cmdlets **Export-Clixml** einen Ordner und Dateinamen für die XML-Ausgabe an. Der Ordner muss vorhanden sein, da ansonsten die in Abbildung 3.2 dargestellte Fehlermeldung angezeigt wird.

Abbildung 3.2 Fehlermeldung, wenn der Zielordner nicht vorhanden ist

Beachten Sie, dass die Fehlermeldung irreführend ist. Die Meldung gibt an, dass eine Datei nicht geöffnet werden kann, obwohl Sie in eine XML-Datei exportieren und den Ordner nicht öffnen können. Die Fehlermeldung wird angezeigt, da ein Ordner nicht vorhanden ist. Das Cmdlet kann den Ordner nicht erstellen und öffnen. Die Ausgabedatei selbst muss jedoch nicht unbedingt vorhanden sein, bevor Sie das Cmdlet ausführen.

Benutzerrechte für den Zugriff auf Ereignisprotokolle

In Windows Vista und Windows Server 2008 verfügen Benutzer ohne erhöhte Rechte im Stammverzeichnis des Laufwerks nicht über Schreibberechtigungen. Sie müssen über Rechte für den entsprechenden Ordner verfügen, um Ereignisprotokolle zu speichern. Für den Zugriff auf das Anwendungsprotokoll benötigen Sie jedoch keine erhöhten Berechtigungen. Beim Starten des Dienstprogramms **Eventvwr.exe** werden Sie aufgrund des Sicherheitsprotokolls von der Benutzerkontensteuerung (User Account Control, UAC) zur Eingabe Ihrer Anmeldeinformationen aufgefordert. Für den Zugriff auf das Sicherheitsprotokoll muss Ihrem Sicherheitstoken die *seSecurityPrivilege*-Berechtigung zugewiesen sein. Diese Berechtigung wird einem normalen Benutzer nicht standardmäßig gewährt. Da den Mitgliedern der Administratorgruppe die Berechtigung zugewiesen ist, müssen Sie Ihr Skript mit erhöhten Berechtigungen ausführen. Eine einfache Methode zum Ausführen von Skripts mit erhöhten Berechtigungen ist das Erstellen einer erhöhten Windows PowerShell-Konsole. Klicken Sie mit der rechten Maustaste auf die Verknüpfung, wählen Sie **Eigenschaften**, klicken Sie dann auf der Registerkarte **Verknüpfung** auf **Erweitert** und aktivieren Sie die Option **Als Administrator ausführen**.

WriteAppLogToXML.ps1

```
Get-EventLog application |
Export-Clixml -Path c:\fso\applog.xml -Depth 2
```

Nachdem das Ereignisprotokoll in eine XML-Datei exportiert wurde, können Sie die Datei in Microsoft Excel öffnen. Klicken Sie auf **Daten**, dann auf **Externe Daten abrufen**, wählen Sie **Aus anderen Quellen** aus und klicken Sie auf **Vom XML-Datenimport**. Die Umwandlung dauert einige Minuten. Möglicherweise wird eine Meldung angezeigt, dass das Schema nicht gefunden werden kann, aber die Daten werden trotzdem in ein Excel-Arbeitsblatt importiert. Die Spaltennamen sind nicht mit den Feldnamen im Ereignisprotokoll identisch, sondern werden beispielsweise als *n* oder *ns:1* angezeigt. Mittels der Daten in den Spalten können Sie die Namen jedoch mit den in der Protokolldatei gespeicherten Daten vergleichen. Sie können die Daten filtern, indem Sie auf den Dropdown-Pfeil am oberen Spaltenrand klicken (siehe Abbildung 3.3).

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

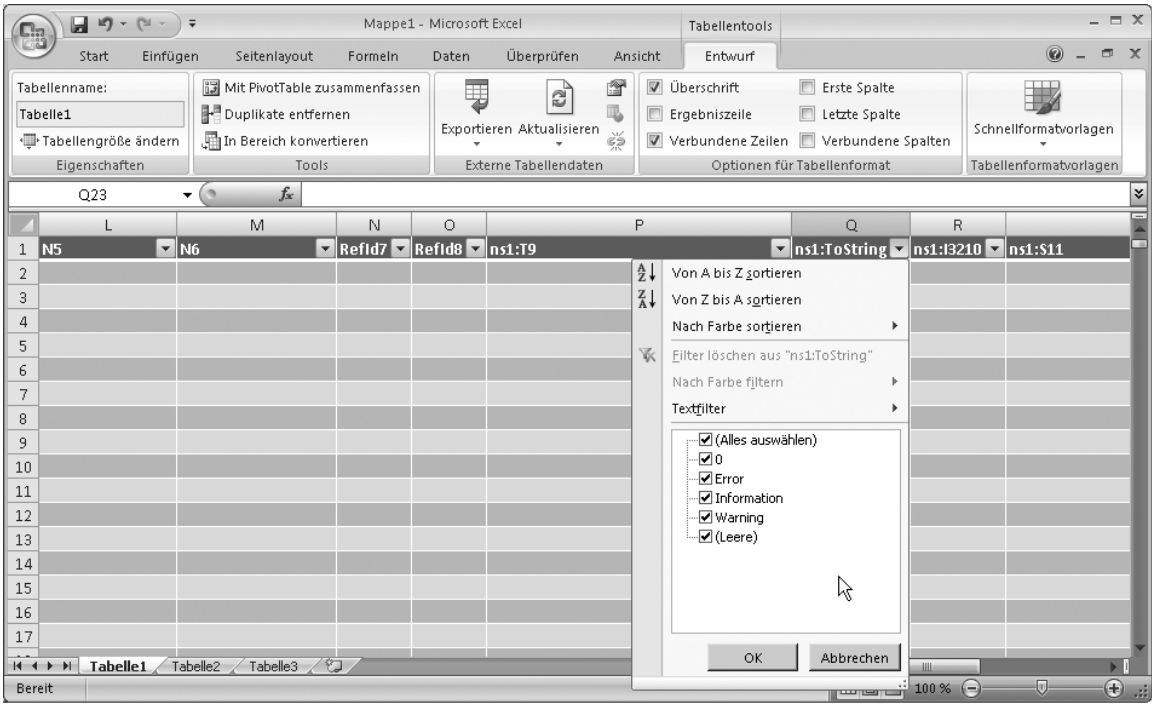


Abbildung 3.3 Exportieren des Ereignisprotokolls in Excel, um wichtige Meldungen anzuzeigen und zu sortieren

Überprüfen allgemeiner Protokolldateien

Um nur eine Übersicht der Fehlertypen im Ereignisprotokoll zusammenzustellen, können Sie das Cmdlet **Get-EventLog** verwenden. Dies ist im folgenden Skript *GetNewestLogEntries.ps1* veranschaulicht. Mit dem Argument **-newest** ruft das Skript eine bestimmte Anzahl an Protokolleinträgen ab.

Tipp Wenn Sie mehrere Argumente für ein Cmdlet verwenden, sollten Sie die Namen aller Parameter angeben. Auch wenn Sie den Standardparameter ohne Parameternamen verwenden, müssen Sie die optionalen Parameter festlegen. Um diesen Vorgang zu vereinfachen, geben Sie einen Bindestrich ein, drücken Sie die Tabtaste, drücken Sie die Eingabetaste, geben Sie einen weiteren Bindestrich ein und drücken Sie erneut die Tabtaste. Diese Methode ist schnell und zuverlässig.

Geben Sie im Skript *GetNewestLogEntries.ps1* in der Variablen *$strLog* die Zeichenfolge an, die das gewünschte Ereignisprotokoll repräsentiert. Mit der Variablen *$intNew* können Sie die Anzahl der abzurufenden Protokolleinträge festlegen. Nachdem Sie die Variablen initialisiert haben, können Sie mit dem Cmdlet **Get-EventLog** die letzten 50 Einträge aus dem Anwendungsprotokoll anzeigen. Die Auswahl von lediglich 50 Einträgen stellt das Gleichgewicht zwischen Geschwindigkeit und Funktionalität sicher. In vielen Fällen reicht diese Methode aus, um eine schnelle Übersicht über aufgetretene Fehler auf einem Server oder einer Arbeitsstation anzuzeigen. Diese Methode ist jedoch nicht sehr genau, da nicht ersichtlich ist, wie viele Einträge am Vortag oder in der vorherigen Stunde

protokolliert wurden. Die 50 Einträge können, abhängig von der Serverauslastung, in einer Woche, einem Tag oder einer Stunde erstellt worden sein. Das Skript *GetNewestLogEntries.ps1* umfasst folgende Anweisungen:

GetNewestLogEntries.ps1

```
$strLog = "application"
$intNew = 50
Get-EventLog -LogName $strLog -newest $intNew
```

Überprüfen mehrerer Protokolle

Das Überprüfen der neuesten Einträge in einer Protokolldatei kann zwar Ihre Neugier befriedigen, eignet sich aber nicht für die umfassende Fehlerbehandlung. Sie können die neuesten Einträge aus dem Ereignisprotokoll abrufen, indem Sie das Skript *GetNewestLogEntries.ps1* entsprechend ändern.

Wichtig Verwenden Sie Variablen anstelle von festen Werten, um den Code wiederverwenden zu können.

Erstellen Sie im Skript *GetNewestLogEntriesAllLogs.ps1* die Variable *$aryLogs* für die Ereignisprotokollobjekte, die vom Cmdlet **Get-EventLog** mit dem Argument **-list** zurückgegeben werden. Nachdem die Ereignisprotokollobjekte zurückgegeben wurden, fügen Sie diese in eine *foreach*-Anweisung ein. Verwenden Sie für das Ereignisprotokoll den gleichen Variablennamen, der im Skript *GetNewestLogEntries.ps1* angegeben ist. Verwenden Sie die Variable *$strLog*, um jeweils ein Ereignisprotokollobjekt aus der Variablen *$aryLogs* zuzuweisen.

Verwenden Sie **Write-Host** im *foreach*-Anweisungscode, um für jedes Protokollergebnis einen Header auszugeben. Legen Sie mit dem Argument **-foregroundcolor** des Cmdlets **Write-Host** die Textfarbe Grün fest, um den Text von den anderen Zeilen auf dem Bildschirm abzuheben. Verwenden Sie das Graviszeichen, um den Befehl in der nächsten Zeile fortzusetzen und die Fragezeichen sowie die Textausgabe auszurichten. Wenn Sie kein weiteres *$* vor dem Befehl *$strLog.log* eingeben, wird nur der Name des Objekts anstatt des Wertes der angegebenen Eigenschaft zurückgegeben. Da Sie doppelte Anführungszeichen eingegeben haben, müssen Sie den übrigen Text, einschließlich der Variablen *$intNew*, nicht weiter bearbeiten.

Rufen Sie anschließend mit dem Cmdlet **Get-EventLog** die Protokolle anhand der Namen unter Verwendung des Eigenschaftenwerts *Log* aus dem Objekt *$strLog* ab. Rufen Sie mit dem Argument **-newest** die Anzahl der Ereignisprotokollobjekte aus der Variablen *$intNew* ab. Das Skript *GetNewestLogEntriesAllLogs.ps1* hat folgenden Aufbau:

GetNewestLogEntriesAllLogs.ps1

```
$aryLogs = Get-EventLog -List
$intNew = 5
foreach ($strLog in $aryLogs)
{
  Write-Host -ForegroundColor green `
        "
         Die $intNew neuesten Einträge aus dem $($strLog.log)-Ereignisprotokoll:
        "
  Get-EventLog -LogName $strLog.log -newest $intNew

}
```

Abrufen eines Protokolleintrags

Um nur den letzten Eintrag aus einem Ereignisprotokoll anzuzeigen (wenn Sie beispielsweise bei einem Anwendungsabsturz weitere Informationen benötigen), führen Sie das Cmdlet **Get-EventLog** aus und geben den Parameter **-newest 1** an. Dieser Vorgang ist im Skript *GetSingleEventEntry.ps1* veranschaulicht. Das Skript verwendet den Standardparameter **-newest**, um den neuesten Eintrag abzurufen. Das Skript *GetSingleEventEntry.ps1* umfasst folgende Anweisung:

GetSingleEventEntry.ps1

```
Get-EventLog -LogName application -Newest 1
```

Das Skript *GetSingleEventEntry.ps1* gibt eine einzige Zeile zurück, die möglicherweise genügend Informationen enthält, um das Problem zu beheben. Beispiel:

```
Index Time          Type Source                EventID Message
----- ----          ---- ------                ------- -------
 7929 Mai 26 09:15  Erro Dhcp                  1001 Diesem Computer konnte keine Netzwerkadresse durch den DHCP-Server
für die Netzwerkkarte mit der Netzwerkadresse 000C29EC70CC zugeteilt werden...
```

Möglicherweise benötigen Sie jedoch mehr Informationen, als die Standardausgabe anzeigt. Die einfachste Art weitere Informationen abzurufen besteht im Einfügen der Skriptausgabe in das Cmdlet **Format-List**. Sie müssen das Skript *GetSingleEventEntry.ps1* nicht ändern. Sie können das zurückgegebene Objekt direkt in ein anderes Cmdlet einfügen (siehe Abbildung 3.4).

Eine interessantere Methode zum Abrufen des letzten Ereignisprotokolleintrags basiert auf einer Eigenschaft des Cmdlets **Get-EventLog**. Dieses Cmdlet ruft eine Sammlung der Ereignisprotokolleinträge ab, die im Wesentlichen ein auf Null basierendes Array darstellt. Das heißt, Sie können die Ausgabe des Cmdlets **Get-EventLog** verwenden, um mit einem Indexwert in eckigen Klammern einen Eintrag wie bei einem Array abzurufen. Schließen Sie hierzu den Systembefehl **Get-EventLog** in runde Klammern ein, wie im folgenden Skript *Get32ndEventLogEntry.ps1* dargestellt. Fügen Sie [31] hinzu, um den 32. Eintrag aus dem Ereignisprotokoll abzurufen.

Get32ndEventLogEntry.ps1

```
(get-eventlog system)[31]
```

Wenn Ihnen die Gesamtanzahl der Einträge nicht bekannt ist, schließen Sie das Cmdlet **Get-EventLog** in runde und eckige Klammern ein. Ermitteln Sie die Länge des Anwendungsprotokolls, subtrahieren Sie 1 (da es sich um ein auf Null basierendes Array handelt) und rufen Sie mit diesem Wert den ersten Eintrag aus dem Ereignisprotokoll ab. Dies wird im Skript *GetFirstEntry.ps1* demonstriert.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

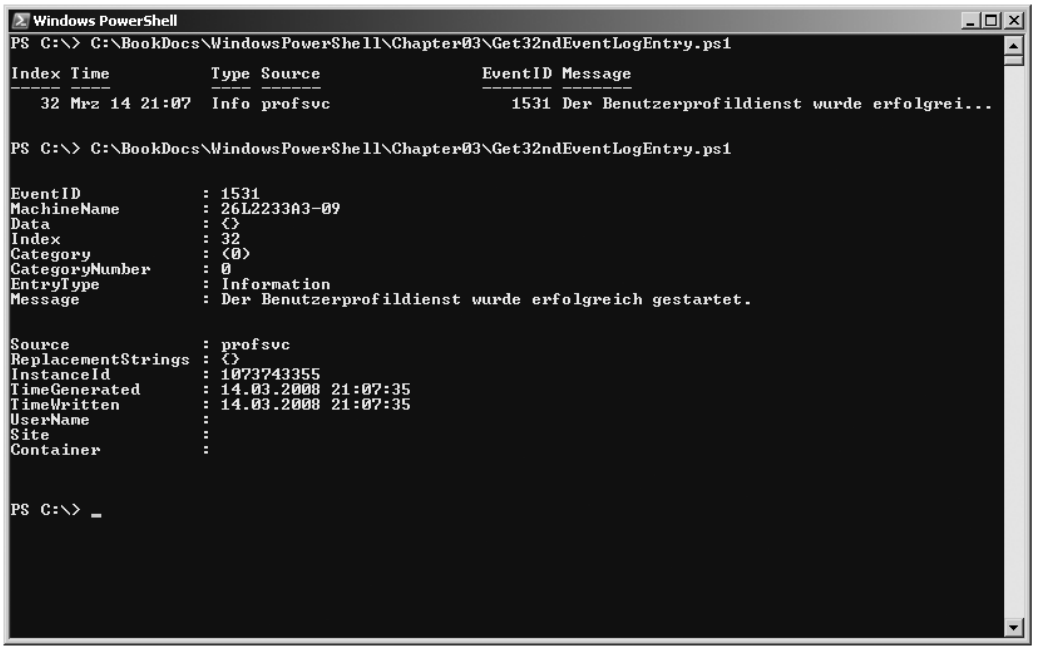


Abbildung 3.4 Die Ausgabe eines Skripts kann für die weitere Verarbeitung in ein Cmdlet eingefügt werden

GetFirstEntry.ps1

```
(Get-EventLog application)[(Get-eventlog application).length-1] |
Format-list *
```

Wenn Sie nur am letzten Eintrag interessiert sind, geben Sie [0] ein. Dies ist im Skript *GetLastEvent.ps1* dargestellt.

Tipp Beachten Sie bei der Arbeit mit Ereignisprotokollen, dass diese für den Zeilenumbruch ausgelegt sind. Das heißt, der erste Eintrag hat die höchste Indexnummer, da sich der letzte Eintrag immer an der Indexposition [0] befindet. Da Ihnen die höchste Indexnummer wahrscheinlich nicht bekannt ist, können Sie die Länge des Ereignisprotokolls verwenden (siehe *GetFirstEntry.ps1*).

Wie bereits erwähnt, hängt das Skript *GetFirstEntry.ps1* davon ab, dass die Ereignisprotokollobjekte vom Cmdlet **Get-EventLog** als indizierte Sammlung zurückgegeben werden. Dies ermöglicht das Abrufen der Objekte mittels der Indexnummer.

GetLastEvent.ps1

```
Write-Host "Die folgenden Informationen stammen aus dem letzten Ereignisprotokolleintrag:"
(Get-EventLog application)[0] | format-list *
```

Um alle Informationen zu einem bestimmten Eintrag anzuzeigen, fügen Sie das zurückgegebene Ereignisprotokollobjekt in das Cmdlet **Format-List** ein. Beispiel:

```
EventID           : 1531
MachineName       : M5-18.nwtraders.com
Data              : {}
Index             : 544
Category          : (0)
```

```
CategoryNumber     : 0
EntryType          : Information
Message            : Der Benutzerprofildienst wurde erfolgreich gestartet.
Source             : profsvc
ReplacementStrings : {}
InstanceId         : 1073743355
TimeGenerated      : 27.5.2007 4:47:53
TimeWritten        : 27.5.2007 4:47:53
UserName           :
Site               :
Container          :
```

Durchsuchen des Ereignisprotokolls

Da das Exportieren der Ereignisprotokolle in eine Text- oder XML-Datei einen zusätzlichen Schritt erfordert, ist die Online-Analyse für Produktionssysteme besser geeignet. Aus diesem Grund sollten Sie Ihre Kenntisse bezüglich der Suchverfahren vertiefen. Die einfachste Methode zum Durchsuchen des Ereignisprotokolls stützt sich auf das Cmdlet **Get-EventLog**. Anstatt die Daten jedoch in einem Zwischenformat zu speichern, fügen Sie die Ergebnisse in ein anderes Cmdlet ein, um die Suche auszuführen. In den folgenden Abschnitten werden mehrere Methoden erklärt. Eine dieser Methoden verwendet das Skript *SearchByEventID.ps1*.

SearchByEventID.ps1

```
Get-EventLog -LogName system |
Where-Object { $_.eventID -eq 1129 }
```

Um das Ereignisprotokoll zu durchsuchen, müssen Ihnen die Mitgliedsvariablen des *EventLogEntry*-Objekts bekannt sein. Dieses Objekt wird als das *System.Diagnostics.EventLogEntry*-Objekt bezeichnet und entspricht einer Standardklasse im Microsoft .NET Framework. Mit dem Cmdlet **Get-Member** können Sie die Eigenschaften des *System.Diagnostics.EventLogEntry*-Objekts abrufen. Übergeben Sie hierzu mit folgendem Befehl das Objekt an das Cmdlet **Get-Member**. Die zurückgegebenen Eigenschaften sind in Tabelle 3.1 aufgeführt.

```
(Get-EventLog application)[0] | Get-Member -MemberType property
```

Tabelle 3.1 *System.Diagnostics.EventLogEntry*-Eigenschaften

Name	Definition
Category	System.String Category {get;}
CategoryNumber	System.Int16 CategoryNumber {get;}
Container	System.ComponentModel.IContainer Container {get;}
Data	System.Byte[] Data {get;}
EntryType	System.Diagnostics.EventLogEntryType EntryType {get;}
Index	System.Int32 Index {get;}
InstanceID	System.Int64 InstanceId {get;}
MachineName	System.String MachineName {get;}
Message	System.String Message {get;}
ReplacementStrings	System.String[] ReplacementStrings {get;}

Tabelle 3.1 *System.Diagnostics.EventLogEntry*-Eigenschaften *(Fortsetzung)*

Name	Definition
Site	System.ComponentModel.ISite Site {get;set;}
Source	System.String Source {get;}
TimeGenerated	System.DateTime TimeGenerated {get;}
TimeWritten	System.DateTime TimeWritten {get;}
UserName	System.String UserName {get;}
EventID	System.Object EventID {get=$this.get_EventID() -band 0xFFFF;}

Filtern von Eigenschaften

Um die Menge der vom Cmdlet **Get-EventLog** zurückgegebenen Informationen zu reduzieren, verwenden Sie das Cmdlet **Where-Object**. Die Eigenschaften, die ich hauptsächlich zum Filtern der Einträge verwende, sind *Source, Severity, Event ID,* und *Message Text.* Das Filtern des Ereignisprotokolls basierend auf der *Ereignis-ID* wurde bereits in diesem Kapitel erklärt. Im Folgenden werden weitere Optionen beschrieben.

Auswählen der Quelle

Wenn beispielsweise mit Microsoft Outlook Probleme auftreten, suchen Sie im Ereignisprotokoll nach der Quelle *Outlook*. Das Skript *FindUSBEvents.ps1* filtert die Ergebnisse basierend auf der *Source*-Eigenschaft. Die *Source*-Eigenschaft eines Ereignisprotokollobjekts gibt die Quelle des Ereignisses an. Das Ereignis kann beispielsweise von einer Anwendung oder einem Controller ausgelöst worden sein. Dieses Beispielskript sucht nach einer Fehlerquelle, die die Buchstaben *usb* enthält, um die Ereignisse anzuzeigen, die sich auf USB-Geräte beziehen.

Führen Sie das Cmdlet **Where-Object** aus. Geben Sie dabei im Codeblock von **Where-Object** die Variable *$_* an, um auf das aktuelle Pipelineobjekt zu verweisen. Suchen Sie nach der *Source*-Eigenschaft des Ereignisprotokollobjekts und geben Sie die Standardinformationen bei Übereinstimmung mit dem Textmuster aus, beispielsweise *usb*. Das Skript *FindUSBEvents.ps1* hat folgenden Aufbau:

FindUSBEvents.ps1

```
Get-EventLog application |
Where-Object { $_.source -like "*usb*" }
```

Auswählen des Schweregrads

In vielen Fällen ist es hilfreich nur die Fehler aus dem Ereignisprotokoll anzuzeigen. Früher musste der Netzwerkadministrator hierfür mit der rechten Maustaste auf das Ereignisprotokoll klicken, dann auf **Aktuelles Protokoll filtern** klicken und das Kontrollkästchen **Fehler** auswählen. Diese Methode hat einige Vorteile, aber verbirgt möglicherweise potenzielle Probleme, die als Warnungen oder Informationen im Ereignisprotokoll aufgezeichnet sind.

Beachten Sie dies beim Ausführen des Skripts *GetSystemLogErrors.ps1*. Geben Sie den Namen des zu überprüfenden Ereignisprotokolls in der Variablen *$strLog* an. Speichern Sie den Namen des

gewünschten Eintragstyps in der Variablen *$strType*. Führen Sie das Cmdlet **Get-EventLog** aus, um das Systemereignisprotokoll abzurufen und die Ereignisprotokollobjekte anzuzeigen. Fügen Sie die zurückgegebenen Objekte in das Cmdlet **Where-Object** ein.

Verwenden Sie im Cmdlet **Where-Object** nun die automatische Variable *$_* und wählen Sie die Eigenschaft *EntryType* des eingefügten Objekts aus. Geben Sie die Standardansicht des Objekts aus, wenn die *EntryType*-Eigenschaft dem Wert in der Variablen *$strType* entspricht (in diesem Beispiel *error*). Das Skript *GetSystemLogErrors.ps1* hat folgenden Aufbau:

GetSystemLogErrors.ps1

```
$strLog ="system"
$strType="error"

Get-EventLog $strLog |
Where-Object { $_.entryType -eq $strType }
```

Auswählen der Meldung

Eine leistungsfähige Methode zum Durchsuchen des Ereignisprotokolls ist die Verwendung von regulären Ausdrücken, um den Meldungstext zu analysieren. Das Cmdlet **Where-Object** kann reguläre Ausdrücke verwenden, wenn Sie das Argument **-match** angeben. Reguläre Ausdrücke sind in Kapitel 2 erklärt. Um die Onlinehilfe anzuzeigen, führen Sie folgenden Befehl aus:

```
get-help about_Regular_Expression
```

Ein Vorteil ist, dass Sie die obskure Sprache für reguläre Ausdrücke nicht beherrschen müssen. Beispielsweise können Sie im Skript *GetHalfDuplex.ps1* den Ausdruck *halbduplex* in der *Message*-Eigenschaft der Protokolleinträge suchen. Dieses Skript ist ausgesprochen hilfreich, da es anzeigt, wie oft die Arbeitsstation oder der Server eine Halbduplex-Verbindung anstatt einer Vollduplex-Verbindung herstellt.

Wenn Sie das Skript *GetHalfDuplex.ps1* verwenden, geben Sie die Zeichenfolge *system* in der Variablen *$strLog an.* Die Variable *$strLog* enthält den Namen des zu durchsuchenden Protokolls. Geben Sie den regulären Ausdruck für die Suche in der Variablen *$strText* an. Geben Sie für dieses Beispiel die Zeichenfolge *half duplex* ein. Nachdem Sie die Variablen initialisiert haben, führen Sie das Cmdlet **Get-EventLog** mit dem Argument **-logname** aus, um die Suche auf ein bestimmtes Ereignisprotokoll zu beschränken.

Fügen Sie die vom Cmdlet **Get-EventLog** zurückgegebenen Objekte in das Cmdlet **Where-Object** ein. Rufen Sie mit der automatischen Variablen *$_*, welche einem Ereignisprotokollobjekt zugewiesen ist, die Eigenschaft *Message* ab. Führen Sie das Cmdlet **Where-Object** mit dem Argument **-match** aus, um nach der Zeichenfolge in der Variablen *$strText* zu suchen. Das Skript *GetHalfDuplex.psi* hat folgenden Aufbau:

GetHalfDuplex.ps1

```
$strLog = "system"
$strText = "half duplex"
Get-EventLog -LogName $strLog |
Where-Object { $_.message -match $strText }
```

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Verwalten des Ereignisprotokolls

Bei der Arbeit mit Ereignisprotokollen müssen zahlreiche Konfigurationseinstellungen verwaltet werden. Die wichtigste Konfigurationseinstellung betrifft die Größe der Protokolldatei. Die Protokolldatei muss groß genug sein, um den Verlauf eines bestimmten Systemereignisses zu umfassen, aber nicht so groß, dass die Ansammlung der Ereignisse unübersichtlich wird.

Identifizieren der Quellen

Es ist wichtig zu wissen, welches Ereignisprotokoll für Protokollierungszwecke verwendet wird. Um diese Informationen zu identifizieren, müssen Sie die registrierten Quellen für das Ereignisprotokoll ermitteln. Eine einfache Methode die Quellen zu ermitteln bietet die WMI-Klasse *Win32_NtEventLogFile*. Dieser Vorgang wird vom Skript *GetLogSources.ps1* ausgeführt. Geben Sie den Namen des Ereignisprotokolls in der Variablen *$strLog* an. In diesem Beispiel wird das Anwendungsprotokoll verwendet.

Verwenden Sie das Cmdlet **Write-Host**, um eine Headerzeichenfolge auszugeben. Führen Sie anschließend das Cmdlet **Get-WmiObject** aus, um die WMI-Klasse *Win32_NtEventLogFile* abzufragen. Definieren Sie einen Filter, der ausschließlich die Quellen abruft, die den Namen des Ereignisprotokolls enthalten. Verwenden Sie das Cmdlet **ForEach-Object**, um die Quellnamen auszugeben. Das vollständige Skript *GetLogSources.ps1* umfasst folgende Anweisungen:

GetLogSources.ps1

```
$strLog = "application"
Write-Host "Die folgenden Quellen sind für das Ereignisprotokoll $strLog registriert: `n"
Get-WmiObject win32_nteventlogfile -Filter "logfilename like '%$strLog%'" |
foreach { $_.sources }
```

Ändern der Ereignisprotokolleinstellungen

In der Vergangenheit war das Konfigurieren bestimmter Einstellungen für die Verwaltung von Windows-Servern oft schwierig. Es war zwar möglich die Standardgröße eines Ereignisprotokolls festzulegen, aber die Aufbewahrungsrichtlinie konnte nicht mit einem Skript geändert werden. Sie können nun auf Windows Vista- und Windows Server 2008-Computern die Aufbewahrungsrichtlinie mit der .NET Framework-Klasse *System.Diagnostics.EventLog* ändern.

Außerdem können Sie die Aufbewahrungsrichtlinie abrufen. Das Skript *GetEventLogRetentionPolicy.ps1* gibt die maximale Größe des Ereignisprotokolls in KB, die Mindestaufbewahrungszeit für die Protokolle in Tagen und die Überlaufrichtlinie zurück. In Windows Vista und Windows Server 2008 können die folgenden drei Überlaufrichtlinien konfiguriert werden:

- **DoNotOverwrite** Wenn das Ereignisprotokoll voll ist, werden die vorhandenen Einträge beibehalten, aber neue Einträge werden verworfen.
- **OverwriteAsNeeded** Wenn das Ereignisprotokoll voll ist, werden die alten Einträge mit neuen Einträgen überschrieben.
- **OverwriteOlder** Wenn das Ereignisprotokoll voll ist, werden Einträge, die älter sind, als in der *MinimumRetentionDays*-Eigenschaft festgelegt, mit neuen Einträgen überschrieben. Neue Ereignisse werden verworfen, wenn das Ereignisprotokoll voll ist. Ereignisse können außerdem nicht älter werden, als in der Eigenschaft *MinimumRetentionDays* festgelegt.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Das Skript *GetEventLogRetentionPolicy.ps1* verwendet das Cmdlet **New-Object**, um eine Instanz der Klasse *System.Diagnostics.EventLog* zu erstellen. Dieses Verfahren ist insofern ungewöhnlich, da Sie ein Argument mit dem Namen des Ereignisprotokolls angeben. Nachdem Sie ein Objekt erstellt haben, das das Anwendungsprotokoll repräsentiert, zeigen Sie mit dem Cmdlet **Write-Host** die Eigenschaft *LogDisplayName* im Header der Ausgabe an. Rufen Sie anschließend die Eigenschaften *MaximumKiloBytes*, *MinimumRetentionDays* und *OverFlowAction* ab. Damit nicht nur der Objektname mit dem zugehörigen Eigenschaftennamen ausgegeben wird, stellen Sie allen Variablen das $-Zeichen voran und schließen Sie den Namen in runde Klammern ein:

```
$($objLog.maximumKiloBytes)
```

Das $-Zeichen und die runden Klammern stellen sicher, dass der Eigenschaftenwert ausgegeben wird. Um die Ausgabe auszurichten und den Code übersichtlicher darzustellen, verschieben Sie die Anführungszeichen für das Cmdlet **Write-Host** in separate Zeilen. Da sich das öffnende Anführungszeichen in der gleichen Zeile wie **Write-Host** befinden muss, geben Sie am Ende der **Write-Host**-Anweisung ein Graviszeichen ein. Das Graviszeichen zeigt an, dass der Windows PowerShell-Befehl nicht beendet ist und verhindert eine Fehlermeldung. Das Skript *GetEventLogRetentionPolicy.ps1* hat folgenden Aufbau:

GetEventLogRetentionPolicy.ps1

```
$strLog = "application"
$objLog = New-Object system.diagnostics.eventlog("$strLog")

Write-Host `
 "
 Die gegenwärtigen Einstellungen für das Ereignisprotokoll $($objlog.logDisplayName) lauten:
 max. Kilobytes: $($objLog.maximumKiloBytes)
 min. Aufbewahrung (Tage): $($objLog.minimumRetentionDays)
 aktuelle Überlaufrichtlinie: $($objLog.overFlowAction)
"
```

Um die Aufbewahrungsrichtlinie für Ereignisprotokolle zu ändern, verwenden Sie die .NET Framework-Klasse *System.Diagnostics. EventLog*. Sie müssen angeben, welche Aufbewahrungsrichtlinie Sie ändern möchten. Die Methode *ModifyOverFlowPolicy* erfordert zwei Parameter: Den Namen der Richtlinie und die Anzahl der Tage für die Aufbewahrung. Wenn Sie DoNotOverwrite oder OverwriteAsNeeded angeben, wird der zweite an die Methode übergebene Parameter ignoriert.

Das Skript *SetEventLogRetentionPolicy.ps1* verwendet die .NET Framework-Klasse *System.Diagnostics.EventLog*, um zwei Vorgänge auszuführen. Das Skript gibt die aktuellen Einstellungen für das angegebene Ereignisprotokoll zurück und legt anschließend die Aufbewahrungsrichtlinie auf den Wert fest, der in der Befehlszeile als Argument angegeben wurde.

Bewährte Vorgehensweise Die Befehlszeilenargumente im Skript *SetEventLogRetentionPolicy.ps1* stellen eine hohe Flexibilität sicher. Sie können das Skript wie ein herkömmliches Skript verwenden und die gewünschte Richtlinieneinstellung so festlegen, dass die Funktion *ChangeLogSettings* aufgerufen wird. Sie können das Skript über ein Anmeldeskript aufrufen und das Argument im Skript übergeben. Sie können den gleichen Vorgang mit einer Batchdatei ausführen. Sie können das Skript auch wie ein Befehlszeilenprogramm verwenden und das Argument in der gleichen Zeile wie das Argument eingeben, das das Skript startet.

Die beiden Funktionen werden am Anfang des Skripts *SetEventLogRetentionPolicy.ps1* angezeigt, da Windows PowerShell von oben nach unten liest. Nach der Definition der beiden Funktionen müssen Sie

mehrere Variablen initialisieren. Die erste Variable ist *$strLog,* die den Namen des zu ändernden Ereignisprotokolls enthält. In diesen Beispiel wurde *application* für das Anwendungsprotokoll angegeben.

Wichtig Stellen Sie sicher, dass Sie das Skript *SetEventLogRetentionPolicy.ps1* mit erhöhten Rechten ausführen, um zu verhindern, dass eine Fehlermeldung angezeigt wird. Sie müssen über Administratorberechtigungen verfügen, um an der Aufbewahrungsrichtlinie für Ereignisprotokolle Änderungen vorzunehmen.

Die nächste Variable ist *$intRetention.* Diese Variable ist standardmäßig auf 30 festgelegt. Dieser Wert ist nur bei Verwendung der Einstellung *OverwriteOlder* gültig.

In der Variablen *$objLog* ist die Instanz der Klasse *System.Diagnostics.EventLog* und der Name des Ereignisprotokolls gespeichert. Code:

```
$objLog = New-Object system.diagnostics.eventlog("$strLog")
```

Nachdem Sie die Variablen erstellt und initialisiert haben, müssen Sie die Funktion *DisplayLogSettings* aufrufen. Diese Funktion gibt die maximale Größe des Ereignisprotokolls in KB, die Mindestaufbewahrungszeit in Tagen und die Überlaufaktion an. Geben Sie die Werte mit dem Cmdlet **Write-Host** aus. Um die Anführungszeichen für das Cmdlet **Write-Host** übersichtlich auszurichten, geben Sie am Ende der Zeile das Graviszeichen ein. Um nun die Werte der Variablen auszugeben (anstatt nur den Variablennamen oder Objektnamen), müssen Sie den Namen der Variablen oder Eigenschaft in runde Klammern setzen und ein $-Zeichen voranstellen. Syntax:

```
overflow policy: $($objLog.overFlowAction)
```

Nachdem Sie die aktuellen Ereignisprotokolleinstellungen angezeigt haben, beenden Sie die Funktion und fahren mit der nächsten Codezeile des Skripts fort. Wenn im Skript keine Argumente angegeben wurden, rufen Sie die Funktion *ChangeLogSettings* mit dem Argument *help* auf. Das Skript gibt daraufhin detaillierte Hilfeinformationen aus und wird beendet. Codezeile:

```
if (!$args) { ChangeLogSettings("help") }
```

Die Funktion *ChangeLogSettings* umfasst den wichtigsten Codeblock im Skript. Die Funktion verwendet eine *switch*-Anweisung und ermöglicht das Festlegen von drei separaten Aufbewahrungsrichtlinien, ohne dass Sie sich die komplizierte Syntax der .NET Framework-Klasse *System.Diagnostics.EventLog* merken müssen. Der Eingabeparameter für die *switch*-Anweisung ist *$profile*, der einem der drei folgenden drei Argumente entspricht: *-donotow, -owasneeded* und *-owolder.* Der als Argument angegebene Wert legt fest, welche Aufbewahrungsrichtlinie angewendet wird. Code:

```
"doNotOW"    { $objlog.modifyoverflowpolicy("DoNotOverwrite",-1) }
"owAsNeeded" { $objlog.modifyoverflowpolicy("OverwriteAsNeeded",-1) }
"owOlder"    { $objlog.modifyoverflowpolicy("Overwriteolder",$intRetention) }
```

Die *switch*-Anweisung gibt standardmäßig eine detaillierte Hilfemeldung in Rot aus, um diese hervorzuheben. Beenden Sie das Skript, nachdem Sie die Hilfemeldung gelesen haben. Das Skript *SetEventLogRetentionPolicy.ps1* hat folgenden Aufbau:

SetEventLogRetentionPolicy.ps1

```
function DisplayLogSettings()
{
 Write-Host `
 "
 Die gegenwärtigen Einstellungen für das Ereignisprotokoll $($objlog.LogDisplayName) lauten:
 max. Kilobytes: $($objLog.maximumKiloBYtes)
 min. Aufbewahrung (Tage): $($objLog.minimumRetentionDays)
```

```
 aktuelle Beibehaltungsrichtlinie: $($objLog.overFlowAction)
"
 if (!$args) { ChangeLogSettings("help") }
}

function ChangeLogSettings($policy)
{ if($policy -ne "help")
    {
     Write-Host -ForegroundColor green "Ändern der Protokollrichtlinie ..."
    }
 switch($policy)
  {
   "doNotOW"    { $objlog.modifyoverflowpolicy("DoNotOverwrite",-1) }
   "owAsNeeded" { $objlog.modifyoverflowpolicy("OverwriteAsNeeded",-1) }
   "owOlder"    { $objlog.modifyoverflowpolicy("Overwriteolder",$intRetention) }
   DEFAULT      {
                 Write-Host -ForegroundColor red `
                  "
                 Sie müssen einen der folgenden Parameter angeben: `n
                 doNotOW - Protokolleinträge nicht überschreiben.
                 owAsNeeded - Protokolleinträge falls erforderlich überschreiben.
                 owOlder - Protokolleinträge überschreiben, wenn diese älter als $intRetention Tage sind. `n
                 Beispiel: > SetEventLogRetentionPolicy.ps1 doNotOW
                             Beibehaltungsrichtlinie auf 'Protokolleinträge nicht überschreiben' setzen

                 Beispiel: > SetEventLogRetentionPolicy.ps1 owAsNeeded
                             Beibehaltungsrichtlinie auf 'Protokolleinträge falls erforderlich überschreiben' setzen.

                 Beispiel: > SetEventLogRetentionPolicy.ps1 owOlder
                             Beibehaltungsrichtlinie auf 'Protokolleinträge älter als 30 Tage überschreiben' setzen.

                 Beispiel: > SetEventLogRetentionPolicy.ps1 help
                             Diese Hilfeinformationen anzeigen.

                  "
    exit            }
 }
}

$strLog = "application" #Ändern Sie diesen Wert, um ein anderes Protokoll anzugeben.
$intRetention = 30      #Ändern Sie diesen Wert, um eine andere Beibehaltungszeitspanne in Tagen anzugeben.
$objLog = New-Object system.diagnostics.eventlog("$strLog")

DisplayLogSettings($args)
ChangeLogSettings($args)
DisplayLogSettings($args)
```

Überprüfen der WMI-Ereignisprotokolle

WMI (Windows Management Instrumentation) ist eine wichtige Komponente von Windows Vista und Windows Server 2008. Um diese Komponente zu überwachen, müssen Sie die WMI-Protokollierungsebene anpassen. Sie können die folgenden drei Protokollierungsebenen festlegen: *Keine, Nur Fehler* und *Ausführlich*. Diese Ebenen sind mit 0, 1 und 2 nummeriert. Diese Protokollierungsebenen sind

jedoch veraltet und werden nur für die WMI-Basisprotokollierung und ältere Anwendungen verwendet. Neuere WMI-Anwendungen verwenden die Ereignisnachverfolgung für Windows-Protokolle (Event Tracing for Windows, ETW). Mit folgendem Skript können Sie die Protokollierungsebene anzeigen.

GetWMILogLevel.ps1

```
Write-Host "Die WMI-Protokollierungsebene ist:
$((Get-WmiObject win32_wmisetting).logginglevel)"
```

Ändern der WMI-Protokollierungsebene

Mit dem Skript *SetWMILogLevel.ps1* können Sie Änderungen an der WMI-Protokollierungsebene vornehmen. Das Skript verwendet die gleiche WMI-Klasse und WMI-Eigenschaft, aber die Methode *put()*, um Änderungen an WMI zurückzugeben. Beachten Sie, dass Sie zum Ausführen des Skripts erhöhte Berechtigungen benötigen. Wenn Sie nicht über erhöhte Rechte verfügen, wird zwar kein Fehler generiert, aber die gewünschte Änderung wird nicht vorgenommen. Das Skript *SetWMILogLevel.ps1* umfasst folgende Anweisungen:

SetWMILogLevel.ps1

```
$wmiLog = Get-WmiObject win32_WMISetting
$wmiLog.logginglevel = 2
$wmiLog.put()
```

Das Befehlszeilenprogramm Wevtutil.exe

In Windows Vista und Windows Server 2008 wurden viele der veralteten ASCII-basierten Textprotokolle durch ETW-Ablaufverfolgungsprotokolle ersetzt. Die Ablaufverfolgungsprotokolle können in der MMC-Konsole (Microsoft Management Console), aber nicht mit den normalen Dienstprogrammen, angezeigt werden. Die Anzeige in der MMC beruht nicht auf Verwendungsanforderungen, sondern auf der praktischen Positionierung. Sie können das Befehlszeilenprogramm *Wevtutil.exe* (Windows Event Command-Line Utility) für die Arbeit mit Ablaufverfolgungsprotokollen verwenden. Sie können die von diesem Programm zurückgegebenen Daten in Windows PowerShell manipulieren. Rufen Sie mit dem Programm *Wevtutil.exe* im Skript *CheckStatusWMILog.ps1* Informationen über das WMI-Diagnoseprotokoll ab.

Wenn Sie das Skript *CheckStatusWMILog.ps1* verwenden, weisen Sie den Namen des WMI-Protokolls als eine Zeichenfolge der Variablen *$strLog* zu. Verwenden Sie anschließend eine *switch*-Anweisung mit dem Argument **-wildcard**, um die Befehlsausgabe zu durchsuchen, ohne eine genaue Übereinstimmung angeben zu müssen. Sie können beispielsweise den Platzhalter * oder ? angeben, um in der Befehlsausgabe nach Übereinstimmungen zu suchen. Geben Sie als Nächstes den Text an. Führen Sie den Befehl **Wevtutil** mit dem Argument **gl** aus, um eine bestimmtes Protokoll abzurufen. Der Name des Protokolls ist in der Variablen *$strLog* angegeben.

Sie können nun im nächsten Codeblock nach einer Zeichenfolge suchen, die das Wort *enabled* enthält. Nachdem das Wort gefunden wurde, rufen Sie mit der Variablen *$switch* die aktuelle Zeile in der Ausgabe ab. Das Skript *CheckStatusWMILog.ps1* hat folgenden Aufbau:

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

CheckStatusWMILog.ps1

```
$strLog = "Microsoft-Windows-EventLog-WMIProvider/Debug"
switch -wildcard (wevtutil gl $strLog)
 {
  "*enabled*" { $switch.Current }
 }
```

Schreiben in Ereignisprotokolle

Das Schreiben in Ereignisprotokolle ist ein ausgesprochen nützliches Feature, da die Windows Vista- und Windows Server 2008-Ereignisprotokolle die zentralisierte Verwaltung von Systemereignissen unterstützen. Mit der .NET Framework-Klasse *System.Diagnostics.EventLog* können Sie Ereignisse in allen Ereignisprotokollen aufzeichnen: *Anwendung*, *System* und *Sicherheit*. Außerdem können Sie zusätzliche Ereignisprotokolle erstellen und in diesen Ereignisprotokollen Informationen erfassen.

Erstellen einer Quelle

Um in ein Ereignisprotokoll zu schreiben, müssen Sie zuerst eine Quelle erstellen. Die Quelle wird vom Ereignisprotokoll verwendet, um den Ursprung des Ereignisses zu bestimmen. Diese Eigenschaft ist für Abfragen hilfreich, wenn Sie in ein Ereignisprotokoll mit einer freigegebenen Quelle schreiben. Nachdem Sie die Quelle erstellt haben, können Sie eine neue Instanz des Ereignisprotokollobjekts erstellen, die Quelle dem Ereignisprotokoll zuordnen und die Meldung angeben. Dies ist im Skript *WriteToAppLog.ps1* veranschaulicht.

Überprüfen Sie zunächst wie im Skript *WriteToAppLog.ps1*, ob die Quelle, die Sie verwenden möchten, bereits definiert und einem Ereignisprotokoll zugeordnet ist. Verwenden Sie hierzu den Operator *not* (d.h. ein Ausrufezeichen). Die Klasse *System.Diagnostics.EventLog* umfasst die Methode *SourceExists*. Beachten Sie, dass die Methode statischer Natur ist und dem Methodennamen deshalb zwei Doppelpunkte vorangehen müssen. Syntax:

```
if(![system.diagnostics.eventlog]::sourceExists("ps_script"))
```

Wenn die Quelle nicht vorhanden ist, müssen Sie diese mit der Methode *CreateEventSource* der Klasse *System.Diagnostics.EventLog* erstellen. Beim Aufrufen der Methode *CreateEventSource* müssen Sie den Quellnamen und den Namen des Ereignisprotokolls angeben, dem die Quelle zugeordnet wird.

Wichtig Eine Ereignisquelle kann nur einem Ereignisprotokoll zugeordnet werden. Um in mehrere Ereignisprotokolle zu schreiben, müssen Sie mehrere Ereignisquellen erstellen.

Nachdem Sie die Ereignisquelle definiert haben, können Sie mit dem Cmdlet **New-Object** eine Instanz des Ereignisprotokolls erstellen. Geben Sie im Skript *WriteToAppLog.ps1* die Variable *$strLog* an, die das vom Cmdlet **New-Object** zurückgegebene Ereignisobjekt enthält. Wenn Sie dieses Cmdlet ausführen, müssen Sie den Namen des Protokolls und den Namen des Computers angeben, auf dem das Protokoll gespeichert ist. Im folgenden Beispiel ist das Anwendungsprotokoll auf einem lokalen Computer gespeichert. Der Computer wird durch einen Punkt (**.**) angegeben.

Wenn die Variable *$strLog* einen Verweis auf das Anwendungsprotokoll enthält, weisen Sie mit der Eigenschaft *Source* die Quelle *ps_script* dem Objekt zu. Verwenden Sie anschließend die Methode *WriteEntry*, um den Text in das Anwendungsprotokoll zu schreiben. Geben Sie für dieses Beispiel *Test mit Script* ein. Das Skript *WriteToAppLog.ps1* hat folgenden Aufbau:

WriteToAppLog.ps1

```
if(![system.diagnostics.eventlog]::sourceExists("ps_script"))
{
 $strLog = [system.diagnostics.eventlog]::CreateEventSource("ps_script","Application")
}
$strLog = new-object system.diagnostics.eventlog("application",".")
$strLog.source = "ps_script"
$strLog.writeEntry("Test mit Script")
```

Einfügen der Ausgabe in das Protokoll

Der Text *Test mit Script* ist zwar einfach, aber kann sich als hilfreich erweisen. Beispielsweise können Sie mit *WriteToAppLog.ps1* aufzeichnen, zu welchem Zeitpunkt das Skript ausgeführt wurde und ob die Ausführung erfolgreich war. Dies kann für die Problembehandlung hilfreich sein, um beispielsweise herauszufinden, warum eine bestimmte Einstellung nicht verfügbar ist.

Außerdem können Sie die Ausgabe von Windows PowerShell-Befehlen aufzeichnen. Mit *WriteProcessesToAppLog.ps1* können Sie die Ergebnisse einer WMI-Abfrage in das Anwendungsprotokoll schreiben. Die Dokumentation der zu einem bestimmten Zeitpunkt auf einem Computer ausgeführten Prozesse bietet wertvolle Informationen für Leistungs- und Sicherheitseinstellungen.

Das Skript *WriteProcessesToAppLog.ps1* entspricht bis auf wenige Unterschiede dem Skript *WriteToAppLog.ps1*. Beachten Sie, dass Windows PowerShell-Objekte und -Pipelines verwendet sowie Objektinformationen ausgegeben werden. Wenn Sie versuchen, die Ergebnisse einer WMI-Abfrage in einer Variablen zu speichern, um diese in das Ereignisprotokoll zu schreiben, sind die Ergebnisse nicht sehr spannend (siehe Abbildung 3.5).

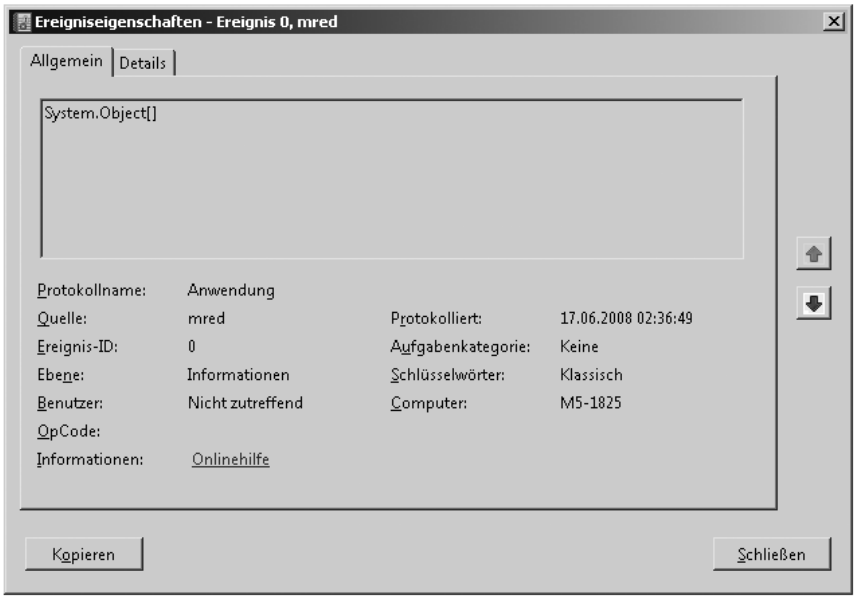


Abbildung 3.5 Die Ausgabe von Cmdlets muss in Zeichenfolgen konvertiert werden

Das Skript *WriteProcessesToAppLog.ps1* verwendet *$strProcess*, um die von den Windows PowerShell-Cmdlets zurückgegebenen Informationen zu erfassen. Führen Sie mit dem Cmdlet

Get-WmiObject eine WMI-Standardabfrage der Klasse *Win32_process* aus. Dies resultiert in einer Sammlung von Objekten, die alle Eigenschaften und Methoden der Prozesse auf dem Computer enthalten. Mit **Select-Object** können Sie die Informationen reduzieren und nur den Namen auswählen. Das Ergebnis ist ein benutzerdefiniertes Windows PowerShell-Objekt. Um die Informationen in eine Zeichenfolge zu konvertieren, führen Sie das Cmdlet **Out-String** aus. Übertragen Sie mit der Methode *WriteEntry* der .NET Framework-Klasse *System.Diagnostics.EventLog* die Prozessnamen aus der Variablen *$strProcess* in eine Zeichenfolge. Das Skript *WriteProcessesToAppLog.ps1* verdeutlicht diese Verarbeitungsschritte:

WriteProcessesToAppLog.ps1

```
$strProcess = get-WmiObject win32_process |
select-object name | out-string

if(![system.diagnostics.eventlog]::sourceExists("ps_script","."))
{
 $strLog = [system.diagnostics.eventlog]::CreateEventSource("ps_script","Application")
}
$strLog = new-object system.diagnostics.eventlog("application",".")
$strLog.source = "ps_script"
$strLog.writeEntry($strProcess)
```

Erstellen von Ereignisprotokollen

Sie können ein separates Ereignisprotokoll erstellen, um die Ereignisse übersichtlicher aufzulisten und die Suche zu vereinfachen. Um ein neues Ereignisprotokoll zu erstellen, verwenden Sie die Methode *CreateEventSource* der Klasse *System.Diagnostics.EventLog* und geben sowohl den Namen der Quelle als auch den Namen des Protokolls an. Eine Ereignisquelle kann nur einem einzigen Ereignisprotokoll zugeordnet werden. Ein Protokoll kann jedoch viele Quellen aufweisen. Um Fehler zu vermeiden, verwenden Sie die Methode *SourceExists* und geben Sie den Namen der gesuchten Quelle an. Wenn die Quelle nicht vorhanden ist, erstellen Sie diese und das Ereignisprotokoll gleichzeitig. Sollte die Quelle jedoch vorhanden sein, können Sie eine Fehlermeldung ausgeben und das Skript beenden. Das Skript CreateEventLog.ps1:

CreateEventLog.ps1

```
$strProcess = get-WmiObject win32_process |
  select-object name | out-string
$source = "ps_script"
$log = "PS_Script_Log"

if(![system.diagnostics.eventlog]::sourceExists($source,"."))
 {
  [system.diagnostics.eventlog]::CreateEventSource($source,$log)
 }
ELSE
 {
  write-host "$source ist bereits für ein anderes Ereignisprotokoll registriert."
  EXIT
 }


$strLog = new-object system.diagnostics.eventlog($log,".")
```

```
$strLog.source = $source
$strLog.writeEntry($strProcess)
```

Wenn die Ereignisquelle bereits mit einem anderen Ereignisprotokoll registriert ist und Sie den Namen der Quelle beibehalten, aber ein benutzerdefiniertes Ereignisprotokoll verwenden möchten, müssen Sie die Ereignisquelle löschen. Sie können hierzu die Methode *DeleteEventSource* der Klasse *System.Diagnostics.EventLog* verwenden. Die Methode *SourceExists* im Skript *DeleteEventSource.ps1* zeigt an, ob die Ereignisquelle bereits registriert ist. Ist dies der Fall, können Sie die Methode *LogNameFromSourceName* verwenden, um den Namen des entsprechenden Ereignisprotokolls auszugeben. Es wird eine Meldung angezeigt, dass die Quelle gelöscht wird. Bestätigen Sie den Löschvorgang. Sollte die Quelle noch nicht registriert sein, wird eine entsprechende Meldung angezeigt. Das Skript *DeleteEventSource.ps1* hat folgenden Aufbau:

DeleteEventSource.ps1

```
$source = "ps_script"

if([system.diagnostics.eventlog]::sourceExists($source,"."))
 {
  $log = [system.diagnostics.eventlog]::LogNameFromSourceName($source,".")
  Write-Host "$source ist bereits für das Ereignisprotokoll $log registriert."
  Write-Host -ForegroundColor red "$source wird gelöscht."
 [system.diagnostics.eventlog]::DeleteEventSource($source)
 }
 ELSE
  { Write-Host -ForegroundColor green "$source ist nicht registriert." }
```

Zusammenfassung

In diesem Kapitel wurden die Ereignisprotokolle in Windows Vista und Windows Server 2008 erklärt. Es wurde beschrieben, wie Sie mit dem Cmdlet **Get-EventLog** eine Liste der verfügbaren Ereignisprotokolle erstellen und die Protokolle auslesen können.

Außerdem wurde die Analyse der Informationen aus den Ereignisprotokollen beschrieben. Sie haben die Ausgabe des Cmdlets **Get-Cmdlet** in das Cmdlet **Where-Object** eingefügt, um die Befehlsausgabe nach bestimmten Protokolleinträgen zu filtern. Nachdem Sie sich mit den Suchfunktionen vertraut gemacht haben, haben Sie Informationen in den Ereignisprotokollen aufgezeichnet und benutzerdefinierte Protokolldateien erstellt.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

KAPITEL 4

Verwalten von Diensten

Nach Abschluss dieses Kapitels können Sie:

- Die vorhandene Dienstkonfiguration dokumentieren
- Informationen in Textdateien aufzeichnen
- Daten in einer zentralisierten Datenbank erfassen
- Eine Liste der erforderlichen Dienste erstellen
- Gewünschte Konfigurationen erstellen
- Einen Einhaltungsbericht generieren

Auf der Begleit-CD Alle Skripts in diesem Kapitel befinden sich auf der Begleit-CD im Ordner *\Scripts\Chapter04*.

Dokumentieren der vorhandenen Dienstkonfiguration

Aus Leistungs- und Sicherheitsgründen ist es wichtig zu wissen, welche Dienste auf einem Server oder einer Arbeitsstation ausgeführt werden. Sie können diese Informationen mit den Cmdlets **Get-Service** und **Get-WmiObject** zusammenstellen.

Das Cmdlet **Get-WmiObject** verwendet die Klasse *Win32_Service* und umfasst mehr Funktionen als das Cmdlet **Get-Service**, einschließlich der Möglichkeit die Konfiguration eines Dienstes zu ändern. Die zusätzliche Funktionalität hat jedoch einen Preis: Dieses Cmdlet ist schwieriger zu verwenden.

Das Cmdlet **Get-Service** gibt standardmäßig eine Liste aller Dienste auf dem Computer zurück, unabhängig davon, ob diese ausgeführt werden oder nicht. Es werden nur die drei folgenden Werte ausgegeben: *Status, Name* und *DisplayName*. Die Liste ist alphabetisch nach Dienstnamen sortiert. Die gekürzte Standardausgabe von Get-Service ist:

```
PS C:\> Get-Service

Status   Name               DisplayName
------   ----               -----------
Running  AeLookupSvc        Anwendungserfahrung
Stopped  ALG                Gatewaydienst auf Anwendungsebene
Running  Appinfo            Anwendungsinformationen
Stopped  AppMgmt            Anwendungsverwaltung
Running  AudioEndpointBu... Windows-Audio-Endpunkterstellung
Running  Audiosrv           Windows-Audio
Running  BFE                Basisfiltermodul
Running  BITS               Intelligenter Hintergrundübertragun...
```

Wenn Sie wissen möchten, wie viele Dienste auf dem Computer definiert sind, verwenden Sie das Skript *CountServices.ps1*. Auf meinem Windows Vista-Computer sind 139 Dienste registriert. Dies wird offensichtlich zu einem Verwaltungsproblem. Die Anzahl der auf einem Computer registrierten

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Dienste ist ein einfacher Hinweise auf den allgemeinen Zustand des Computers. Dieser Indikator bietet jedoch keinen Gesamtüberblick, da Sie einen Dienst deinstallieren und einen anderen Dienst installieren können. Insgesamt sind dann immer noch 139 Dienste vorhanden. Um das Skript *CountServices.ps1* zu verwenden, rufen Sie das Cmdlet **Get-Service** auf, schließen Sie den Namen des Cmdlets in Klammern ein und fragen Sie die Eigenschaft *Length* ab. Die Eigenschaft *Length* listet die Anzahl der Dienste auf dem Computer auf. Die Klammern weisen Windows PowerShell an, den Code in den Klammern zuerst und anschließend die Aktion außerhalb der Klammern auszuführen. In diesem Beispiel besteht die Aktion aus dem Ausführen einer Zählung. Das einzeilige Skript *CountServices.ps1* verwendet folgende Anweisung:

CountServices.ps1

```
(Get-Service).length
```

Arbeiten mit ausgeführten Diensten

Wenn Sie wissen möchten, welche Dienste auf Ihrem Computer ausgeführt werden, können Sie das Cmdlet **Get-Service** verwenden. Sie benötigen jedoch auch das Cmdlet **Where-Object**. Um eine Liste der ausgeführten Dienste anzuzeigen, müssen Sie zuerst mit dem Cmdlet **Get-Service** eine Liste aller Dienste abrufen und das resultierende Objekt anschließend in das Cmdlet **Where-Object** einfügen. Verwenden Sie im Cmdlet **Where-Object** einen Skriptblock, um den Status der Dienste im Objekt zu überprüfen. Referenzieren Sie das aktuelle Objekt mit der automatischen Variablen *$_* und verwenden Sie den Operator *-eq*, um zu überprüfen, ob der Status *running* ist. Wenn dies der Fall ist, fügen Sie das Ergebnis ist das neue Windows PowerShell-Objekt ein, das vom Cmdlet **Where-Object** erstellt wird. Schließen Sie das Pipelineobjekt in Klammern ein und fragen Sie die Eigenschaft *Length* ab. Die Informationen werden in der Konsole angezeigt. Die Klammern erzwingen das Ausführen des Codes bevor die Länge abgerufen wird. Das Skript *CountRunningServices.ps1* umfasst folgende Anweisung:

CountRunningServices.ps1

```
(Get-Service | where-object { $_.status -eq "running" }).length
```

Das Überwachen der Anzahl der ausgeführten Dienste ist sinnvoll. Wie die Anzahl der installierten Dienste insgesamt, bietet die Anzahl der ausgeführten Dienste eine schnelle Übersicht, mittels der Sie feststellen können, ob das System geändert wurde. Diese Methode ist einfacher als eine Überprüfung in der Konsole **Dienste** (siehe Abbildung 4.1). Die Benutzeroberfläche der Konsole **Dienste** ist unübersichtlich und deshalb nicht für die schnelle Überprüfung des Status der Dienste geeignet. Windows PowerShell bietet eine umfassendere Übersicht der Dienste auf dem Computer.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

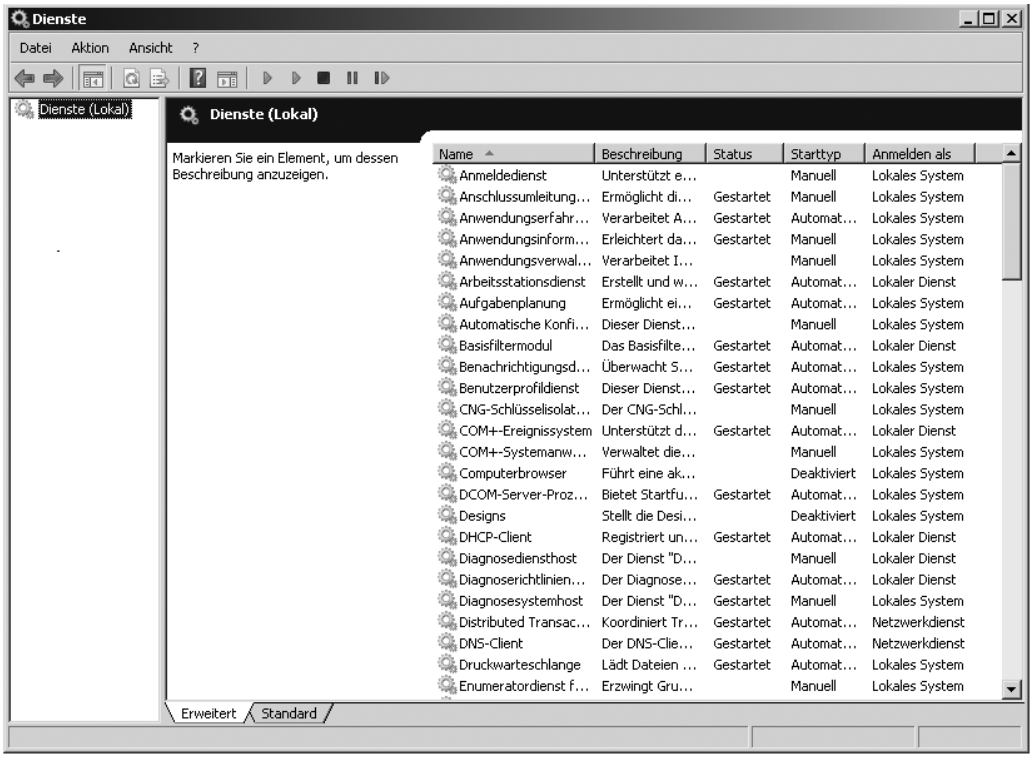


Abbildung 4.1 Die Konsole *Dienste* enthält Informationen zum Dienststatus

Das Gleiche gilt natürlich auch für die Anzahl der definierten Dienste. Diese Angabe ist kein Ersatz für die Überwachung und bietet keine erweiterte Sicherheit, sondern ist ein unkomplizierter visueller Indikator in Hinsicht auf Systemänderungen.

Schreiben einer Textdatei

Obwohl das Überprüfen der ausgeführten Dienste für die tägliche Verwaltung nützlich ist, ist das Dokumentieren der Namen der ausgeführten Dienste noch wichtiger. Es gibt mehrere Gründe für das Erfassen von Dienstinformationen in einer Textdatei. Das Erfassen der Dienstinformationen ist einfach und eine praktische Methode zur Überwachung des Serverstatus. Außerdem ist das Festhalten der Informationen in einer Textdatei hilfreich, um die Basiskonfiguration zu dokumentieren.

Wenn Sie beispielsweise einen Server durch Deaktivieren aller nicht erforderlichen Dienste optimieren möchten, ist es sinnvoll, die vorhandene Konfiguration zu dokumentieren, bevor Sie umfassende Änderungen vornehmen, die zu einer Katastrophe führen können. Die Dokumentation der funktionierenden Konfiguration kann das Wiederherstellen eines betriebsbereiten Zustandes wesentlich vereinfachen.

Das Aufzeichnen der ausgeführten Dienste in einer Textdatei ist im Skript *WriteRunningServices-ToTxt.ps1* veranschaulicht. Die vom Skript *WriteRunningServicesToTxt.ps1* erstellte Textdatei ist in Abbildung 4.2 dargestellt. Die automatischen Spaltenheader vereinfachen das Lesen der Datei, können aber ein Problem verursachen, wenn sie als Eingabe verwendet werden, außer Sie überspringen die Header- und Trennlinien.

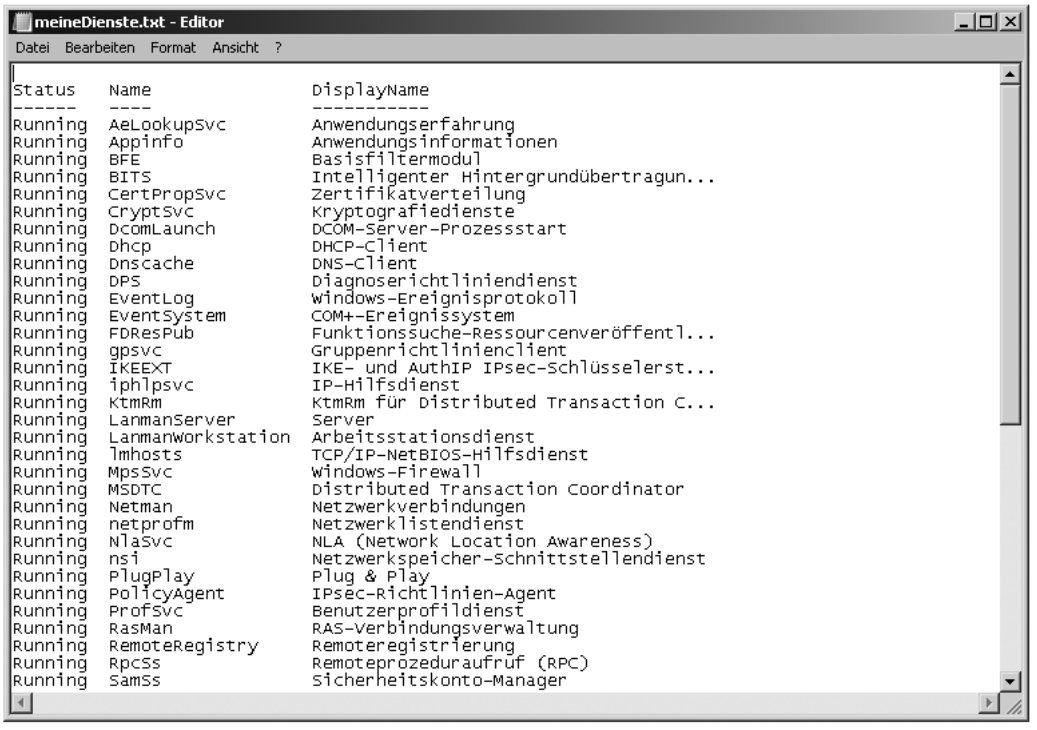


Abbildung 4.2 Eine Liste der ausgeführten Dienste in einer Textdatei

Um das Skript *WriteRunningServicesToTxt.ps1* auszuführen, erstellen Sie zuerst die Variable *$strState* und weisen dieser die Zeichenfolge *running* zu. Erstellen Sie anschließend die Variable *$strPath* und weisen Sie dieser einen Pfad im lokalen System zu. In diesem Beispiel verwenden Sie den Pfad *c:\FSO\meineDienste.txt*. Beachten Sie, dass der Pfad den Dateinamen enthalten muss. Führen Sie anschließend das Cmdlet **Get-Service** aus und fügen Sie die resultierenden Objekte in das Cmdlet **Where-Object** ein. Das Cmdlet **Where-Object** filtert alle Dienste aus, die nicht ausgeführt werden. Nachdem Sie die Objekte gefiltert haben, fügen Sie die Ergebnisse in das Cmdlet **Out-File** ein und die Zeichenfolge in der Variablen *$strPath* in den Parameter **-filepath**. Das vollständige Skript *WriteRunningServicesToTxt.ps1* hat folgenden Inhalt:

WriteRunningServicesToTxt.ps1

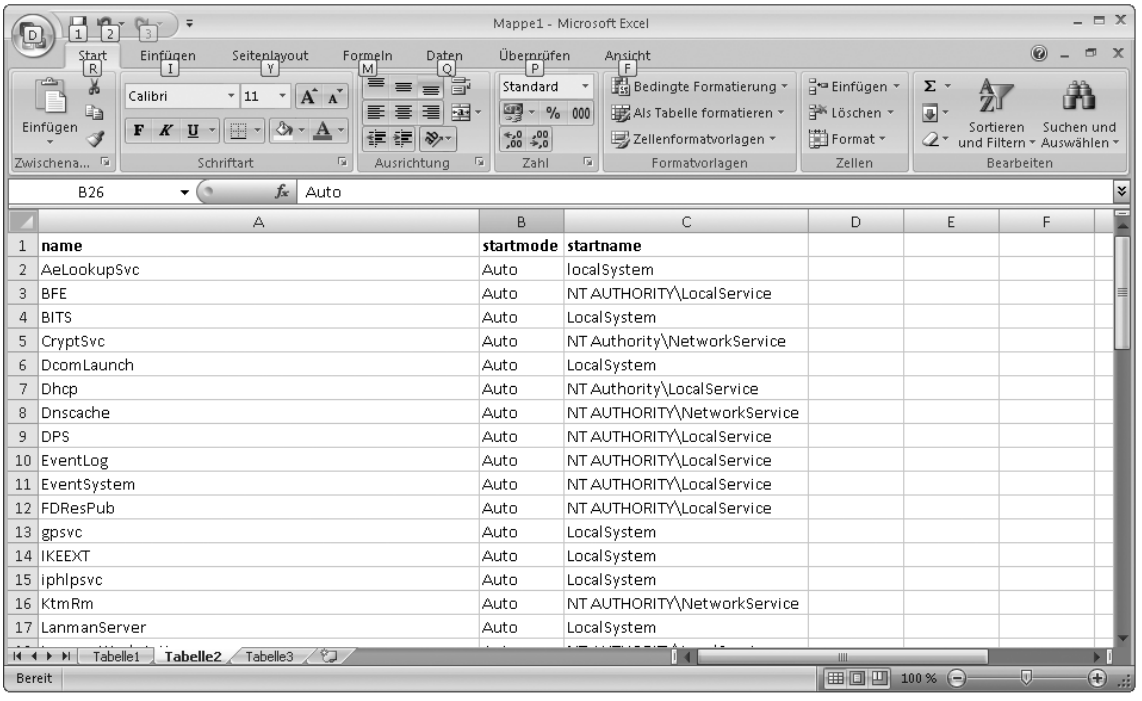
```
$strState = "running"
$strPath = "C:\FSO\meineDienste.txt"
Get-Service |
Where-Object { $_.status -eq $strState } |
Out-File -FilePath $strPath
```

Zusätzlich zum Schreiben der Dienstinformationen in eine Textdatei können Sie die Ergebnisse auch in einer csv-Datei speichern. Eine csv-Datei kann sowohl in Microsoft Excel und Microsoft Access als auch in Microsoft Word als Tabelle geöffnet werden. Außerdem können csv-Dateien von Microsoft SQL Server importiert werden.

Stellen Sie sich vor, wie Ihre Anwendung aussehen soll und von wem sie verwendet wird. Sie können die csv-Datei mit Windows PowerShell in einem für den Benutzer geeigneten Format bereitstellen.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Wenn ein Arbeitsblatt beispielsweise zwei Spalten umfasst und die ersten Spalte einen Dienstnamen und die zweite Spalte den Status enthält, können Sie die csv-Datei bereinigen, um alle nicht erforderlichen Eigenschaften zu entfernen, bevor Sie die Daten in Excel importieren. Verwenden Sie hierzu das Skript *ExportRunningServices.ps1*, das alle Eigenschaften entfernt (außer *Name, StartMode* und *StartName*), bevor Sie die Daten in die csv-Datei exportieren. Die entsprechende Ausgabe ist in Abbildung 4.3 dargestellt.



	A	B	C	D	E	F
1	**name**	**startmode**	**startname**			
2	AeLookupSvc	Auto	localSystem			
3	BFE	Auto	NT AUTHORITY\LocalService			
4	BITS	Auto	LocalSystem			
5	CryptSvc	Auto	NT Authority\NetworkService			
6	DcomLaunch	Auto	LocalSystem			
7	Dhcp	Auto	NT Authority\LocalService			
8	Dnscache	Auto	NT AUTHORITY\NetworkService			
9	DPS	Auto	NT AUTHORITY\LocalService			
10	EventLog	Auto	NT AUTHORITY\LocalService			
11	EventSystem	Auto	NT AUTHORITY\LocalService			
12	FDResPub	Auto	NT AUTHORITY\LocalService			
13	gpsvc	Auto	LocalSystem			
14	IKEEXT	Auto	LocalSystem			
15	iphlpsvc	Auto	LocalSystem			
16	KtmRm	Auto	NT AUTHORITY\NetworkService			
17	LanmanServer	Auto	LocalSystem			

Abbildung 4.3 Das Vereinfachen einer csv-Datei ist mit dem Skript *ExportRunningServices.ps1* möglich, das die Daten in Excel exportiert

Erstellen Sie im Skript *ExportRunningServices.ps1* zuerst die Variable *$strState* und weisen Sie dieser die Zeichenfolge *running* zu. Erstellen Sie anschließend die Variable *$strPath* mit dem Pfad zur exportierten Datei. Führen Sie das Cmdlet **Get-WmiObject** aus, um die WMI-Klasse *Win32_Service* abzufragen. Fügen Sie die Zeichenfolge in der Variablen *$strState* in den Parameter **-filter** des Cmdlets **Get-WmiObject** ein. Fügen Sie das resultierende Objekt in das Cmdlet **Select-Object** ein. Sie können die Eigenschaften *Name, StartMode* und *StartName* in der WMI-Klassse *Win32_Service* auswählen. Exportieren Sie das Objekt mit dem Cmdlet **Export-Csv** in eine csv-Datei und fügen Sie die Zeichenfolge in der Variablen *$strPath* in den Parameter **-path** ein. Das Skript *ExportRunningServices.ps1* umfasst folgende Anweisungen:

ExportRunningServices.ps1

```
$strState = "running"
$strPath = "C:\FSO\meineDienste.txt"
Get-WmiObject win32_service -Filter "state='$strState'" |
select-object name, startmode, startname |
Export-Csv -Path $strPath
```

Erfassen in einer Datenbank

Die in eine Datenbank geschriebenen Daten werden permanent gespeichert und können zur Erstellung von Berichten mit relevanten Informationen herangezogen werden, die übersichtlich und einfach zu lesen sind. Datenbanken können außerdem von mehreren Benutzern gleichzeitig verwendet werden und sind eine stabilere Lösung zum Speichern von Daten als Textdateien, auf die nur jeweils ein Benutzer zugreifen kann. Das Erstellen eines Berichts ist mit dem Report Writer in Access so einfach wie mit einem Assistenten. Nachdem die Datei geschrieben wurde, wird der Bericht automatisch gruppiert und sortiert, um die Analyse der Informationen zu vereinfachen. Ein mit dieser Methode generierter Bericht sieht professionell aus und enthält alle wichtigen Informationen. In Abbildung 4.4 ist ein mit Access generierter Bericht dargestellt.

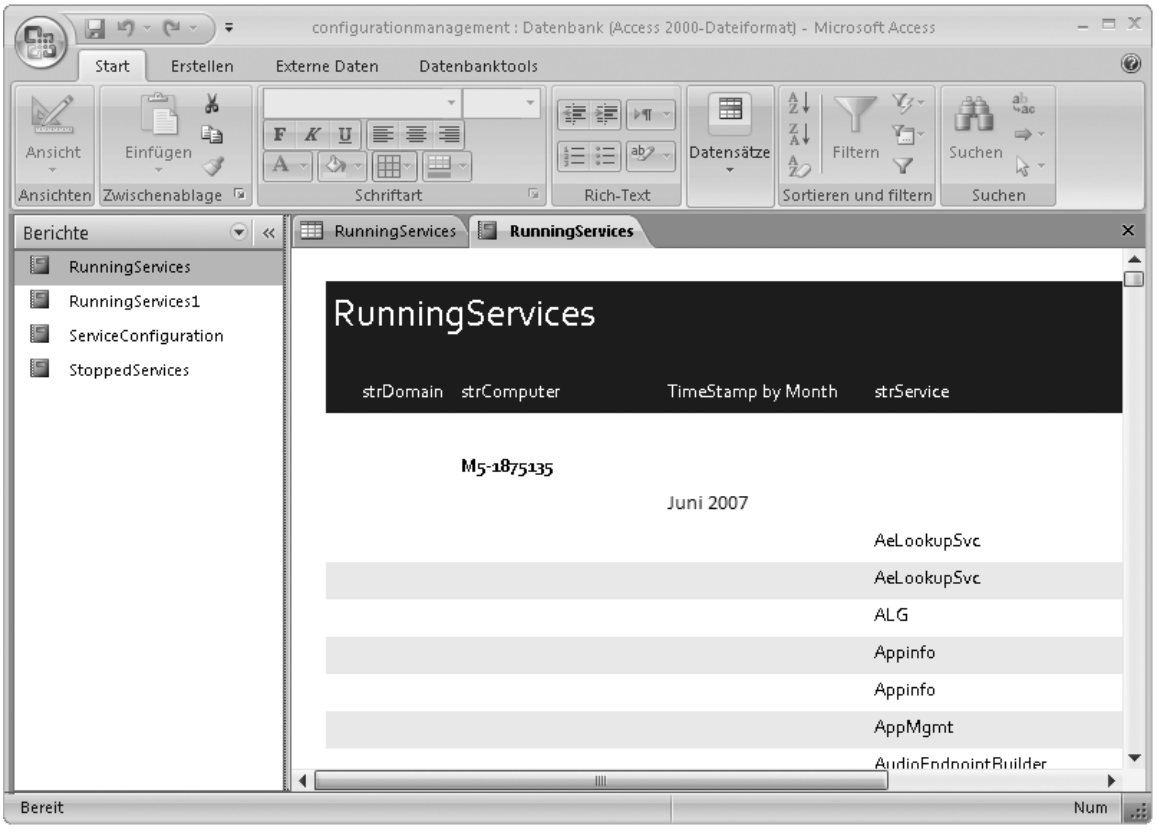


Abbildung 4.4 Der Berichts-Assistent in Access erstellt professionelle Berichte

Das Skript *WriteRunningServicesToAccess.ps1* veranschaulicht den Prozess zum Übermitteln von Daten an eine Access-Datenbank unter Verwendung der ADO-Technologie (Active X Data Objects). In der ersten Skriptzeile rufen Sie mit dem *wshNetwork*-Objekt den aktuellen Computernamen ab. Das Objekt wird vom Cmdlet **New-Object** durch Angabe des Parameters **-comobject** und der Programm-ID *wscript.network* erstellt. Setzen Sie die gesamte Anweisung in Klammern und wählen Sie nur die Eigenschaft *ComputerName* aus dem Objekt aus. Weisen Sie den Computernamen der Variablen *$strComputer* zu.

Verwenden Sie für die zweite Skriptzeile das gleiche Objekt und das gleiche Verfahren mit einer Ausnahme: Wählen Sie die Eigenschaft *Domain* anstatt der Eigenschaft *ComputerName* aus. Die folgenden beiden Codezeilen verdeutlichen die Arbeit mit dem *wshNetwork*-Objekt:

```
$StrComputer = (New-Object -ComObject WScript.Network).computername
$StrDomain = (New-Object -ComObject WScript.Network).Domain
```

Tipp Um die Eigenschaften *ComputerName* und *Domain* aus dem *wshNetwork*-Objekt abzurufen, erstellen Sie zweimal das gleiche Objekt. Sie können auch folgende Methode verwenden:

```
$wshNetwork = (New-Object -ComObject WScript.Network)
$StrComputer = $wshNetwork.computername
$strDomain = $wshNetwork.domain
```

Definieren Sie in der dritten Skriptzeile die WMI-Abfrage anhand einer WQL-Anweisung (WMI Query Language) *"Select * from Win32_Service"*. Wenn Sie diese Abfrage mit dem Cmdlet **Get-WmiObject** ausführen, werden alle Eigenschaften der auf dem Computer definierten Dienste abgerufen. Speichern Sie die WQL-Anweisung in der Variablen *$strQuery.*

Hinweis WQL ist SQL ähnlich, da WQL eine Teilmenge von Transact-SQL darstellt.

Rufen Sie den WMI-Dienst auf dem Computer ab, indem Sie das Cmdlet **Get-WmiObject** mit dem Parameter **-query** ausführen. Die der Variablen *$strWMIQuery* zugewiesene Zeichenfolge wird als Abfragezeichenfolge übergeben und das resultierende Objekt in der Variablen *$objService* erfasst. Die folgenden beiden Codezeilen definieren die WMI-Abfrage:

```
$strWMIQuery = "Select * from win32_Service"
$objservice = get-wmiobject -query $strWMIQuery
```

Nachdem Sie die Informationen abgerufen haben, geben Sie mit dem Cmdlet **Write-Host** eine Statusmeldung aus. Verwenden Sie **-foregroundcolor**, um die Meldung in Gelb auszugeben. Die Zeichenfolge *Ermitteln der Dienstinformationen …* ist für das Cmdlet **Write-Host** hartcodiert. Die **Write-Host**-Codezeile hat folgendes Aussehen:

```
write-host -foreGroundColor yellow "Ermitteln der Dienstinformationen …"
```

Das Cmdlet **Get-WmiObject** gibt mehrere WMI-Objekte zurück, die jeweils einen Dienst auf dem Computer repräsentieren. Verwenden Sie die Anweisung *foreach*, um die Daten zu durchlaufen. *$strService* ist eine Variable, die alle in der Variablen *$objService* gespeicherten Dienste erfasst.

Beginnen Sie den Skriptblock für das Cmdlet **ForEach** unter Verwendung von geschweiften Klammern. Die erste Anweisung im *ForEach*-Codeblock ist eine *if*-Anweisung, um zu bestimmten, ob der jeweilige Dienst ausgeführt wird. Überprüfen Sie, ob die Eigenschaft *$Service.State* den Status *running* aufweist. Der *ForEach*-Codeblock und die *if*-Anweisung beginnen wir folgt:

```
foreach ($service in $objService)
 {
  if ($service.state -eq "running")
  {
```

Wenn der Dienst ausgeführt wird, verzweigen Sie in einen anderen Codeblock und speichern den Wert der Eigenschaft *Service.Name* in der Variablen *$strServiceName,*. Rufen Sie anschließend den Dienststatus ab und weisen Sie diesen in der Variablen *$strStatus* zu. Die beiden WMI-Wertzuordnungen sind unkompliziert:

```
$strServiceName = $service.name
$strStatus = $service.State
```

Erstellen Sie in der nächsten Zeile die Variable *$adOpenStatic* und weisen Sie dieser den Wert 3 zu. Dieser Wert wird beim Öffnen der Datenbankverbindung verwendet. Erstellen Sie die Variable *$adLockOptimistic* und legen Sie diese auf 3 fest. Dieser Wert wird ebenfalls beim Öffnen der Datenbankverbindung verwendet.

Der vollständige Pfad zur Datenbank ist in der Variablen *$strDB* gespeichert. Die Variable *$strTable* enthält den Namen der Zugriffstabelle. Mit diesem Skript greifen Sie auf die Tabelle *runningservices* zu. Weisen Sie also der Variablen *$strTable* diese Zeichenfolge zu. Die folgenden vier Variablen werden von ADO verwendet:

```
$adOpenStatic = 3
$adLockOptimistic = 3
$strDB = "c:\FSO\Services.mdb"
$strTable = "runningServices"
```

Nachdem Sie diese vorbereitenden Maßnahmen getroffen haben, können Sie sich mit den wesentlichen ADO-Vorgängen befassen. Sie müssen zwei Objekte erstellen: Ein *Connection*-Objekt und ein *RecordSet*-Objekt. Um das *Connection*-Objekt zu erstellen, verwenden Sie das Cmdlet **New-Object** mit dem Parameter **-comobject** und der Programm-ID *ADODB.Connection*. Speichern Sie das *Connection*-Objekt in der Variablen *$objConnection*.

Erstellen Sie als Nächstes das *RecordSet*-Objekt. Verwenden Sie hierzu ebenfalls das Cmdlet **New-Object** mit dem Parameter **-comobject**. Verwenden Sie die COM-Objekt-ID *ADODB.Recordset* und speichern Sie das resultierende *Recordset*-Objekt in der Variablen *$objRecordSet*. Die beiden ADODB-Objekte werden mit folgendem Code erstellt:

```
$objConnection = New-Object -ComObject ADODB.Connection
$objRecordSet = new-object -ComObject ADODB.Recordset
```

Nachdem Sie die beiden ADODB-Objekte erstellt haben, können Sie die ADO-Verbindung mit der Datenbank *services.mdb* herstellen. Öffnen Sie die Datenbankverbindung mit dem *Connection*-Objekt in der Variablen *$objConnection*. Verwenden Sie die *Open*-Methode des *Connection*-Objekts und geben Sie den Microsoft.Jet.OLEDB.4.0-Anbieter an. Halten Sie den Anbieter von der Datenquelle getrennt. Die Datenquelle ist als Datenbank mit einem in der Variablen *$strDB* gespeicherten Pfad angegeben. Beachten Sie, dass der Befehl aus einer einzigen logischen Zeile besteht. Das Graviszeichen nach dem Semikolon zeigt die Fortsetzung der Zeile an:

```
$objConnection.Open("Provider = Microsoft.Jet.OLEDB.4.0; `
Data Source= $strDB")
```

Nachdem die Datenbankverbindung geöffnet wurde, verwenden Sie die *Open*-Methode des *RecordSet*-Objekts. Geben Sie hierzu eine SQL-Abfrage ein, listen Sie die Verbindung auf und geben Sie an, wie die Datenbank geöffnet werden soll. Diese Parameter sind in folgendem Code enthalten:

```
$objRecordSet.Open("SELECT * FROM runningServices", `
$objConnection, $adOpenStatic, $adLockOptimistic)
```

Nachdem das *RecordSet* geöffnet wurde, können Sie neue Datensätze zum *RecordSet* hinzufügen. Verwenden Sie hierzu die *Addnew*-Methode des *RecordSet*-Objekts. Mit der Eigenschaft *Fields.Item* des *RecordSet*-Objekts können Sie Daten zur Datenbank hinzufügen. Die Feldnamen aus der Access-Datenbank können Sie für die Datenbanktabelle in der Design-Ansicht anzeigen (siehe Abbildung 4.5).

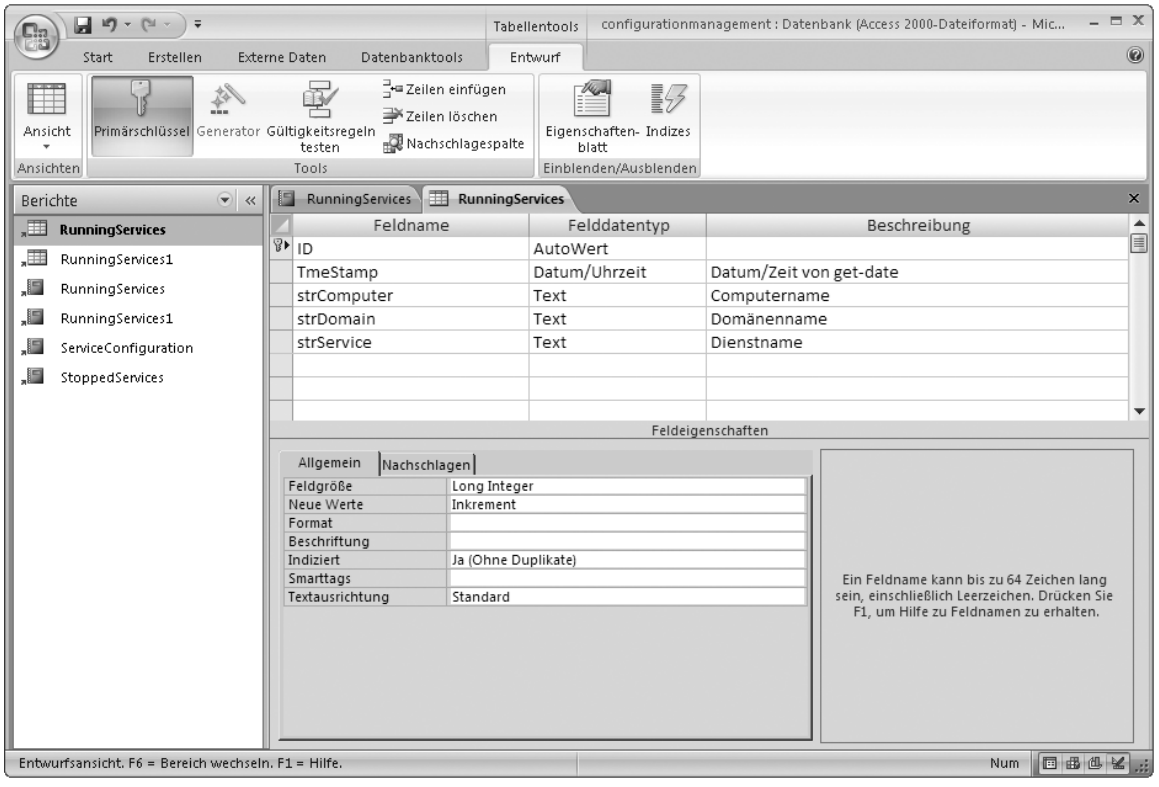


Abbildung 4.5 Die Feldnamen für ein Windows PowerShell-Skript in der Design-Ansicht der Datenbanktabelle

Die Feldnamen in Anführungszeichen stammen aus der Datenbank. Verwenden Sie die Variablen, die Sie zuvor zugewiesen haben, um den Feldern Werte zuzuweisen. Eine Ausnahme ist die Verwendung des Cmdlets **Get-Date** zum Abrufen des aktuellen Datum-/Zeitstempels. Der entsprechende Codeblock umfasst folgende Anweisungen:

```
$objRecordSet.AddNew()
$objRecordSet.Fields.item("TimeStamp") = Get-Date
$objRecordSet.Fields.item("strComputer") = $strComputer
$objRecordSet.Fields.item("strDomain") = $strDomain
$objRecordSet.Fields.item("strService") = $strServiceName
$objRecordSet.Fields.item("strStatus") = $strStatus
```

Um die Daten zurück in die Datenbank zu schreiben, verwenden Sie die *Update*-Methode des *RecordSet*-Objekts. Beispiel:

```
$objRecordSet.Update()
```

Um den Status beim Übermitteln der Daten an die Datenbank anzuzeigen, verwenden Sie das Cmdlet **Write-Host** und geben Sie Schrägstriche und umgekehrte Schrägstriche (/\) aus, die jeweils einen Dienst repräsentieren. Dies ist in folgender Codezeile dargestellt: Um das neue Zeichen an das bereits ausgegebene Zeichen anzuhängen und auf diese Weise das Fortschreiten der Verarbeitung anzuzeigen, verwenden Sie den Parameter **-nonewline**:

```
write-host -foregroundColor yellow "/\" -noNewLine
```

Die Ausgabe des Cmdlets **Write-Host** in der Konsole ist zwar nicht beeindruckend, aber zeigt die Ausführung des Skripts und den Status an. Die vollständige Ausgabe ist in Abbildung 4.6 dargestellt.

Abbildung 4.6 Visuelle Darstellung des Status in der Konsole

Nachdem alle Datensätze in die Datenbank geschrieben wurden, müssen Sie die *Connection*- und *RecordSet*-Objekte schließen. Die folgenden beiden Codezeilen führen diesen Schritt aus:

```
$objRecordSet.Close()
$objConnection.Close()
```

Das folgende Listing spiegelt den vollständigen Text des Skripts *WriteRunningServicesToAccess.ps1* wider.

WriteRunningServicesToAccess.ps1

```
$StrComputer = (New-Object -ComObject WScript.Network).computername
$StrDomain = (New-Object -ComObject WScript.Network).Domain
$strWMIQuery = "Select * from win32_Service"
$objservice = get-wmiobject -query $strWMIQuery

write-host -foreGroundColor yellow "Ermitteln der Dienstinformationen …"

foreach ($service in $objService)
 {
  if ($service.state -eq "running")
  {
   $strServiceName = $service.name
   $strStatus = $service.State
   $adOpenStatic = 3
   $adLockOptimistic = 3
   $strDB = "c:\FSO\Services.mdb"
   $strTable = "runningServices"
   $objConnection = New-Object -ComObject ADODB.Connection
   $objRecordSet = new-object -ComObject ADODB.Recordset
   $objConnection.Open("Provider = Microsoft.Jet.OLEDB.4.0; `
     Data Source= $strDB")
   $objRecordSet.Open("SELECT * FROM runningServices", `

   $objConnection, $adOpenStatic, $adLockOptimistic)
   $objRecordSet.AddNew()
   $objRecordSet.Fields.item("TimeStamp") = Get-Date
   $objRecordSet.Fields.item("strComputer") = $strComputer
   $objRecordSet.Fields.item("strDomain") = $strDomain
   $objRecordSet.Fields.item("strService") = $strServiceName
   $objRecordSet.Fields.item("strStatus") = $strStatus
```

```
   $objRecordSet.Update()
   write-host -foregroundColor yellow "/\" -noNewLine
  }
 }

$objRecordSet.Close()
$objConnection.Close()
```

Erfassen beendeter Dienste

Nachdem Sie sich mit dem Erfassen ausgeführter Dienste in einer Access-Datenbank vertraut gemacht haben, ist das Ändern des Skripts zum Erfassen beendeter Dienste einfach. Dieser Vorgang umfasst das Hinzufügen einer zusätzlichen Tabelle und weiterer Felder zur Access-Datenbank sowie das Erstellen eines geeigneten Berichts. Sie müssen bei Verwendung von Windows PowerShell-Skripting lediglich sicherstellen, dass die im Skript angegebenen Felder mit dem Datenbankdesign übereinstimmen.

Da die meiste Arbeit bereits im Skript *WriteRunningServicesToAccess.ps1* erledigt wurde, können Sie dieses als Ausgangspunkt und Vorlage verwenden. Ändern Sie die *if*-Anweisung, damit nach beendeten Diensten anstatt nach ausgeführten Diensten gesucht wird. Die geänderte Codezeile hat folgendes Aussehen:

```
if ($service.state -eq "stopped")
```

Nachdem Sie die *if*-Anweisung geändert haben, benennen Sie die in der Variablen *$strTable* gespeicherte Datenbanktabelle in *StoppedServices* um. Sie müssen außerdem die im Aufruf der *Open*-Methode hardcodierte Zugriffsabfrage für das *RecordSet*-Objekt ändern. Der geänderte *Open*-Methodenaufruf lautet wie folgt:

```
$objRecordSet.Open("SELECT * FROM StoppedServices", `
$objConnection, $adOpenStatic, $adLockOptimistic)
```

Aufgrund des Datenbankdesigns und da die gleichen Informationen zusammengestellt werden, erfordert das Schreiben in die Datenbank keine Änderungen. Sie verwenden für die Tabelle *StoppedServices* die gleichen Feldnamen wie für die Tabelle *RunningServices* in der Dienstdatenbank. Das vollständige Skript *WriteStoppedServicesToAccess.ps1* hat folgenden Aufbau:

WriteStoppedServicesToAccess.ps1

```
$StrComputer = (New-Object -ComObject WScript.Network).computername
$StrDomain = (New-Object -ComObject WScript.Network).Domain
$strWMIQuery = "Select * from win32_Service"
$objservice = get-wmiobject -query $strWMIQuery

write-host -foreGroundColor yellow "Ermitteln der Dienstinformationen …"


foreach ($service in $objService)
 {
  if ($service.state -eq "stopped")
  {
   $strServiceName = $service.name
   $strStatus = $service.State
   $adOpenStatic = 3
   $adLockOptimistic = 3
   $strDB = "c:\fso\services.mdb"
   $strTable = "StoppedServices"
   $objConnection = New-Object -ComObject ADODB.Connection
```

```
   $objRecordSet = new-object -ComObject ADODB.Recordset
   $objConnection.Open("Provider = Microsoft.Jet.OLEDB.4.0; `
     Data Source= $strDB")
   $objRecordSet.Open("SELECT * FROM StoppedServices", `
   $objConnection, $adOpenStatic, $adLockOptimistic)

   $objRecordSet.AddNew()
   $objRecordSet.Fields.item("TimeStamp") = Get-Date
   $objRecordSet.Fields.item("strComputer") = $strComputer
   $objRecordSet.Fields.item("strDomain") = $strDomain
   $objRecordSet.Fields.item("strService") = $strServiceName
   $objRecordSet.Fields.item("strStatus") = $strStatus
   $objRecordSet.Update()
   write-host -foregroundColor yellow "/\" -noNewLine
  }
 }

$objRecordSet.Close()
$objConnection.Close()
```

Dokumentieren der Dienstkonfiguration

Das Skript *WriteServiceConfigToAccess.ps1* umfasst einige Änderungen und zusätzliche Felder. Die erste Änderung besteht im Entfernen der *if*-Anweisung, da Sie nun die Konfiguration aller Dienste abfragen müssen, unabhängig davon, ob diese ausgeführt werden. Sie müssen nicht nur die Codezeile mit der *if*-Anweisung löschen, sondern auch zwei geschweifte Klammern entfernen.

Nachdem Sie die *if*-Anweisung gelöscht haben, müssen Sie einige neue Variablen hinzufügen und neue Informationen vom WMI-Dienstobjekt abrufen. Speichern Sie den Kontonamen, mit dem der Dienst gestartet wird, in der Variablen *$strStartName*. Der Kontoname ist über die WMI-Eigenschaft *StartName* der Klasse *Win32_Service* ermittelbar. Die neue Codezeile lautet:

```
$strStartName = $service.StartName
```

Als Nächstes müssen Sie den Startmodus der Dienste abfragen. Die Eigenschaft *StartMode* gibt die Konfiguration für den Dienststart an. Die von der WMI-Eigenschaft *StartMode* der Klasse *Win32_Service* zurückgegebenen Werte sind in Tabelle 4.1 aufgeführt.

Tabelle 4.1 Dienststartmodus

Startmodus	Bedeutung
Boot	Der Gerätetreiber wird vom Ladeprogramm des Betriebssystems gestartet
System	Der Gerätetreiber wird vom Initialisierungsprozess des Betriebssystems gestartet
Auto	Der Dienst wird automatisch vom SCM-Manager (Service Control Manager) beim Systemstart gestartet
Manual	Der Dienst wird von SCM gestartet, wenn ein Prozess die *StartService*-Methode aufruft
Disabled	Der Dienst kann nicht gestartet werden

Die neue Codezeile, um die *Startmodusinformationen* abzurufen, lautet:

```
$strStartMode = $service.StartMode
```

Die nächsten abzurufenden Informationen beziehen sich darauf, ob ein Dienst beendet oder angehalten werden kann.

Anhalten von Diensten

Obwohl es nicht ungewöhnlich ist, dass ein Dienst beendet werden kann, kann das Beenden verschiedener Prozesse zu Systeminstabilitäten führen. Deshalb akzeptieren diese Dienste keinen *Stop*-Befehl. Es ist außerdem äußerst ungewöhnlich, dass ein Dienst einen *Pause*-Befehl akzeptiert. Auf meinem Windows Vista-Laptop können nur acht Dienste angehalten werden. Das folgende Skript ruft die entsprechenden Informationen ab.

AcceptPause.ps1

```
Get-WmiObject -Class win32_service |
Where-Object { $_.acceptpause -eq "true" } |
Select-Object name
```

Die folgenden Dienste auf meinem Windows Vista Professional-Laptop akzeptieren *Pause*-Anweisungen. (Beachten Sie, dass das Ergebnis von den ausgewählten Optionen und der installierten Version von Windows Vista abhängt.)

```
name
----
LanmanServer
LanmanWorkstation
Netlogon
seclogon
stisvc
TapiSrv
WerSvc
Winmgmt
```

Die beiden Eigenschaften *AcceptPause* und *AcceptStop* der WMI-Klasse *Win32_Service* zeigen an, ob Sie den jeweiligen Dienst beenden oder anhalten können. Das vorangehende Skriptbeispiel bestimmt mittels der Eigenschaft *AcceptPause*, welche Dienste angehalten werden können. Die Eigenschaften geben einen booleschen Wert (*true* oder *false*) zurück und die Variablen sind *$blnAcceptPause* und *$blnAcceptStop*. Folgender Code ruft diese Werte ab und speichert diese in entsprechenden Variablen:

```
$blnAcceptPause = $service.AcceptPause
$blnAcceptStop = $service.AcceptStop
```

Das Übernehmen dieser Werte in die Datenbank ist relativ einfach. Verwenden Sie einfach das bereits erläuterte Datenbankverfahren. Geben Sie ähnliche Namen für die Variablen und Datenbankfelder an, um Verwirrung zu vermeiden. Codebeispiel:

```
$objRecordSet.Fields.item("strStartMode") = $strStartMode
$objRecordSet.Fields.item("blnAcceptPause") = $blnAcceptPause
$objRecordSet.Fields.item("blnAcceptStop") = $blnAcceptStop
```

Wenn Sie die erforderlichen Änderungen am Skript vornehmen, erhalten Sie den Code für das Skript *WriteServiceConfigToAccess.ps1*. Das vollständige Skript hat folgenden Aufbau:

WriteServiceConfigToAccess.ps1

```
$StrComputer = (New-Object -ComObject WScript.Network).computername
$StrDomain = (New-Object -ComObject WScript.Network).Domain
$strWMIQuery = "Select * from win32_Service"
$objservice = get-wmiobject -query $strWMIQuery

write-host -foreGroundColor yellow "Ermitteln der Dienstinformationen …"
```

```
foreach ($service in $objService)
{
   $strServiceName = $service.name
   $strStartName = $service.StartName
   $strStartMode = $service.StartMode
   $blnAcceptPause = $service.AcceptPause
   $blnAcceptStop = $service.AcceptStop
   $adOpenStatic = 3
   $adLockOptimistic = 3
   $strDB = "c:\FSO\Services.mdb"
   $strTable = "ServiceConfiguration"
   $strAccessQuery = "Select * from $strTable"
   $objConnection = New-Object -ComObject ADODB.Connection
   $objRecordSet = new-object -ComObject ADODB.Recordset
   $objConnection.Open("Provider = Microsoft.Jet.OLEDB.4.0; `
     Data Source= $strDB")
   $objRecordSet.Open($strAccessQuery, `
   $objConnection, $adOpenStatic, $adLockOptimistic)

   $objRecordSet.AddNew()
   $objRecordSet.Fields.item("TimeStamp") = Get-Date
   $objRecordSet.Fields.item("strComputer") = $strComputer
   $objRecordSet.Fields.item("strDomain") = $strDomain
   $objRecordSet.Fields.item("strService") = $strServiceName
   $objRecordSet.Fields.item("strStartName") = $strStartName
   $objRecordSet.Fields.item("strStartMode") = $strStartMode
   $objRecordSet.Fields.item("blnAcceptPause") = $blnAcceptPause
   $objRecordSet.Fields.item("blnAcceptStop") = $blnAcceptStop
   $objRecordSet.Update()
   write-host -foregroundColor yellow "/\" -noNewLine
}

$objRecordSet.Close()
$objConnection.Close()
```

Festlegen der Dienstkonfiguration

Um die Dienstkonfiguration auf einem Server festzulegen, müssen Sie wissen, welche Dienste ausgeführt werden, welche Dienste automatisch oder manuell gestartet werden und welche Dienste beendet oder deaktiviert sind. Diese Informationen sind in den Microsoft Resource Kits, in TechNet und zahlreichen Whitepapers dokumentiert. Sie können jedoch viele dieser Informationen auch über Windows PowerShell abrufen. Mit dem Skript *GetSpecificService.ps1* können Sie die Informationen ausgeben, die von **Get-Service** über einen bestimmten Dienst abgerufen werden. Weisen Sie hierzu zuerst der Variablen *$strService* den Namen des Dienstes zu. Führen Sie anschließend das Cmdlet **Get-Service** mit dem Parameter **-name** aus und verwenden Sie den Wert in der Variablen *$strService*, um den Namen anzugeben. Fügen Sie die Ergebnisse in das Cmdlet **Format-List** ein und geben Sie den Platzhalter * an, um alle Eigenschaften der Klasse auszuwählen. Das Skript *GetSpecificService.ps1* führt diese Schritte aus:

GetSpecificService.ps1

```
$strService = "bits"
Get-Service -Name $strService |
Format-list *
```

Das Skript *GetSpecificService.ps1* gibt die Informationen zurück, die Sie zum Festlegen der gewünschten Konfiguration benötigen. Folgendes Listing zeigt die Ausgabe einer Abfrage des BITS-Dienstes:

```
Name                : BITS
CanPauseAndContinue : False
CanShutdown         : True
CanStop             : True
DisplayName         : Intelligenter Hintergrundübertragungsdienst
DependentServices   : {}
MachineName         : .
ServiceName         : BITS
ServicesDependedOn  : {EventSystem, RpcSs}
ServiceHandle       :
Status              : Running
ServiceType         : Win32ShareProcess
Site                :
Container           :
```

Für die Verwaltung sind die folgenden vier Informationsabschnitte wesentlich:

- CanPauseAndContinue
- CanStop
- DependentServices
- ServicesDependedOn

Sie sollten nicht versuchen, einen Dienst zu beenden, wenn *CanStop* den Status *False* hat, da die Ausführung des Skripts bei unzulässigen Aktionen verzögert oder abgebrochen wird. Aus Sicherheits- und Stabilitätsgründen, sollte ein Dienst außerdem nicht beendet werden, wenn dies Probleme mit anderen abhängigen Diensten verursachen kann. Die Abhängigkeiten sind in der Konsole **Dienste** aufgelistet, allerdings ist es schwieriger, *CanStop* in der Konsole **Dienste** zu bestimmen (siehe Abbildung 4.7).

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

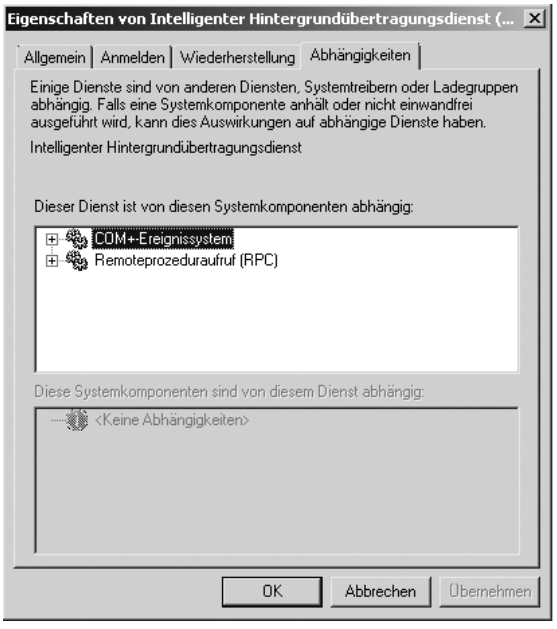


Abbildung 4.7 Bevor Sie einen Dienst beenden, sollten Sie die Abhängigkeitsinformationen überprüfen

Im nächsten Abschnitt wird die Arbeit mit diesen Informationen in Skripts näher beschrieben.

Sie können das Skript *GetSpecificService.ps1* ändern, um mehrere Dienste abzufragen. Wenn Sie *GetSpecificService.ps1* beispielsweise für den BITS-Dienst ausführen, werden zwei abhängige Dienste angezeigt: *EventSystem* und *RpcSs*. Diese Dienstabhängigkeiten werden im Folgenden erklärt.

Übergeben Sie den Namen der beiden mit dem Skript *GetSpecificService.ps1* identifizierten Dienste an die Variable *$aryService*. Wenn Sie der Variablen *$aryService* mehrere Werte zuweisen, erstellt die Windows PowerShell automatisch ein Array. Um das Array anzulegen, verwenden Sie eine *foreach*-Anweisung und *$strService* als Enumeratoren. Bearbeiten Sie den Skriptblock mit den geschweiften Klammern, geben Sie den Namen des Dienstes mit **Write-Host** aus und vervollständigen Sie den restlichen Code in *GetSpecificService.ps1*. Rufen Sie mit dem Cmdlet **Get-Service** und dem Argument **-name** den Dienst ab, dessen Name mit dem Namen in der Variablen *$strService* übereinstimmt. Um alle Eigenschaften des Dienstes und die dazugehörigen Werte abzurufen, fügen Sie die Ausgabe in das Cmdlet **Format-List** ein und geben den Platzhalter * an. Das Skript *GetMultipleServices.ps1* hat folgenden Aufbau:

GetMultipleServices.ps1

```
$aryService = "EventSystem","RpcSs"
foreach($strService in $aryService)
{
 Write-Host "Dienstinfo für: $strService"
 Get-Service -Name $strService |
 Format-list *
}
```

Befehlszeilenargumente

Das Abrufen detaillierter Informationen zu mehreren Diensten über Windows PowerShell ist ausgesprochen nützlich, insbesondere wenn Sie nicht auf das Internet, Microsoft TechNet und die MSDN-Websites (Microsoft Developer Network) zugreifen können. Über Windows PowerShell können Sie genügend Informationen zusammenstellen, um informierte Entscheidungen bezüglich der Serverinfrastruktur zu treffen.

Vorschlag: Um die Verwendbarkeit des Skripts *GetMultipleServices.ps1* zu verbessern, nehmen Sie eine geringfügige Änderung vor. Anstatt die Namen der abzufragenden Dienste hart zu codieren und in die Variable *$aryService* einzugeben, weisen Sie die automatische Variable *$args* der Variablen *$aryService* zu. Auf diese Weise können Sie die Ausführung des Skripts zur Laufzeit steuern, ohne weitere Änderungen am Skript vornehmen zu müssen.

Um das Angeben von Befehlszeilenargumenten im Skript zu unterstützen, ändern Sie die erste Zeile. Anstatt die Dienstnamen einzugeben, ändern Sie den Code wie folgt:

```
$aryService = $args
```

Wenn Sie das Skript *ArgGetMultipleServices.ps1* ausführen, können Sie die Dienstnamen direkt in der Befehlszeile angeben. Beispiel:

```
C:\FSO\ArgGetMultipleServices.ps1 bits lanmanserver
```

Dieser Befehl setzt voraus, dass das Skript im Ordner *FSO* im Stammverzeichnis von Laufwerk C gespeichert ist. Geben Sie zwei Befehlszeilenargumente für das Skript an, um das Dienstarray zu erstellen. Die beiden Dienstnamen sind *bits* und *lanmanserver*. Die Argumente müssen nicht durch Kommas getrennt werden. Das vollständige Skript *ArgGetMultipleServices.ps1* verwendet folgenden Code:

ArgGetMultipleServices.ps1

```
$aryService = $args

foreach($strService in $aryService)
{
 Write-Host "Dienstinfo für: $strService"
 Get-Service -Name $strService |
 Format-list *
}
```

Beenden von Diensten

Sie können Dienste mit Hilfe von Windows PowerShell auf zwei Arten beenden. Im folgenden Beispiel wird der BITS-Dienst beendet:

- **Stop-Service -name BITS**
- **(Get-WmiObject -class win32_service -filter "name = 'bits'").stopService()**

Das Cmdlet **Stop-Service** verkörpert offensichtlich die einfachere Methode. Das Skript *Stop-Service.ps1* verwendet das Cmdlet **Stop-Service**, um den BITS-Dienst auf dem Computer zu beenden. Ein Dienst kann auch mit dem Cmdlet **Stop-Service** auf zwei Arten beendet werden: Basierend auf dem Dienstnamen oder dem Anzeigenamen. Das Skript *StopService.ps1* beendet den BITS-Dienst unter Verwendung des Dienstnamens BITS. Wenn Sie die Eigenschaft *DisplayName* verwenden, um den BITS-Dienst zu beenden, geben Sie *Intelligenter Hintergrundübertragungsdienst* ein. Wenn Ihnen der Dienstname bekannt ist, verwenden Sie diesen anstatt der Eigenschaft *DisplayName*, um den Dienst zu

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

steuern. In der Variablen *$strService* wird der Name des Dienstes gespeichert, der beendet werden soll. Nachdem Sie den Namen des Dienstes ermittelt haben, beenden Sie den Dienst mit dem Cmdlet **Stop-Service**. Geben Sie den Parameter **-name** mit dem Namen des Dienstes anhand der Variablen *$strService* an. Das Skript *StopService.ps1* umfasst folgende Anweisungen:

StopService.ps1

```
$strService = "BITS"
Stop-Service -Name $strService
```

Wichtig Wenn Sie einen Dienst mit dem Cmdlet **Stop-Service** beenden, stellen Sie sicher, dass Sie das Skript mit Administratorrechten ausführen. Sollten Sie nicht über Administratorrechte verfügen, wird die in Abbildung 4.8 dargestellte Fehlermeldung angezeigt.

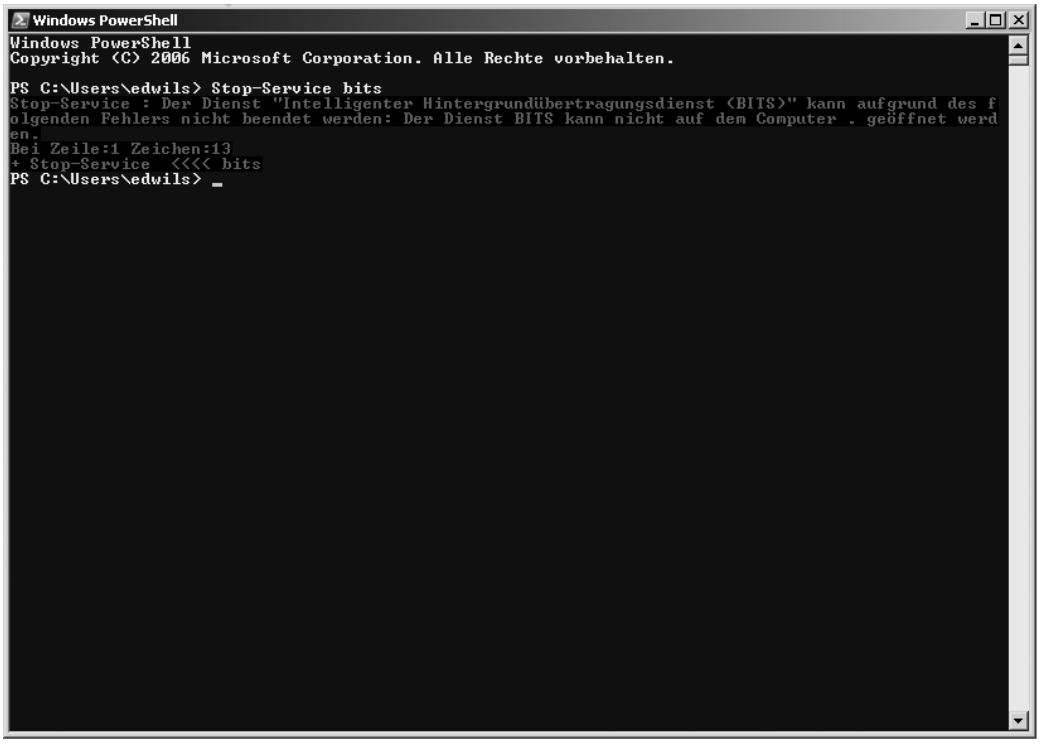


Abbildung 4.8 Sie benötigen Administratorrechte, um einen Dienst über Windows PowerShell starten, beenden oder ändern zu können

Um mehrere Dienste zu beenden, können Sie das Skript *StopService.ps1* entsprechend anpassen. Das Ändern des Skripts umfasst das Erstellen eines Arrays mit Dienstnamen sowie das Durchlaufen des Arrays mit einer *foreach*-Anweisung. Das restliche Skripts wird nicht geändert.

Erstellen Sie im Skript *StopMultipleServices.ps1* ein Array mit den Dienstnamen. Weisen Sie in der ersten Zeile des Skripts die Namen der Dienste der Variablen *$aryServices* zu. Verwenden Sie anschließend eine *foreach*-Anweisung, um das Dienstearray zu durchlaufen. Verwenden Sie die Variable *$strService* als Enumerator für das Array. Geben Sie mit dem Cmdlet **Write-Host** eine

Meldung aus, dass Sie einen bestimmten Dienst beenden. Rufen Sie das Cmdlet **Stop-Service** auf und übergeben Sie den Namen des Dienstes, der beendet werden soll. Das Skript *StopMultipleServices.ps1* hat folgenden Aufbau:

StopMultipleServices.ps1

```
$aryServices = "BITS", "WUAUSERV", "CcmExec"
foreach ($strService in $aryServices)
{
 Write-Host "Der Dienst $strService wird beendet..."
 Stop-Service -Name $strService
}
```

Ordnungsgemäßes Beenden

Es ist sinnvoll die Eigenschaft *AcceptStop* der WMI-Klasse *Win32_Service* abzufragen, bevor Sie den Dienst beenden.

Problembehandlung Wenn mit einem Skript, das das Cmdlet **Get-WmiObject** und die *Win32_Service*-Klasse verwendet, ein Problem auftritt, denken Sie daran, dass die Eigenschaftennamen nicht mit den vom Cmdlet **Get-Service** verwendeten Namen identisch sind. Beispielsweise verwendet **Get-Service** *CanStop* für die Eigenschaft, um anzuzeigen, ob ein Dienst beendet werden kann. Das Cmdlet **Get-WmiObject** und die WMI-Klasse *Win32_Service* verwenden hierzu *AcceptStop*.

Das Skript wird nicht nur schneller und effizienter ausgeführt, sondern es wird auch der Skriptabbruch vermieden. Erstellen Sie im Skript *CheckServiceThenStop.ps1* die Variable *$strService*, um den Namen des zu beendenden Dienstes anzugeben. Sie können den Dienstnamen hartcodieren oder das Skript zum Akzeptieren von Befehlsargumenten anpassen. Die einfachste Methode ist das Ändern von *$strService* für die Verwendung von *$args*. Der Name des Computers, auf dem sich der Dienst befindet, ist in der Variablen *$strComputer* gespeichert. In diesem Beispiel ist localhost der Name des lokalen Computers. Geben Sie den Namen der abzufragenden WMI-Klasse in der Variablen *$strClass* an. In diesem Beispiel *Win32_Service*.

Geben Sie im Cmdlet **Get-WmiObject** drei Argumente an: Die Klasse, den Computer und den Filter. Übergeben Sie die in den Variablen *$strService* und *$strComputer* gespeicherten Werte an die Argumente **-class** und **-computer**. Das Argument **-filter** ersetzt eine *Where*-Klausel in einer WQL-Abfrage. Mit dem Argument **-filter** ist der Code etwas übersichtlicher, als mit einer WQL-Abfrage:

```
"Select * from win32_service where name = 'bits'"
```

Die Syntax ist zwar nicht kompliziert, aber länger als die Windows PowerShell-Anweisung.

Tipp Beachten Sie, dass Sie bei Angabe des Arguments **-filter** im Cmdlet **Get-WmiObject** das Wort ***Where*** im Filter nicht eingeben müssen. Der Filter schlägt vielmehr fehl, wenn Sie das Wort ***Where*** einbeziehen. Diese Methode spart Zeit bei der Eingabe und der Problembehandlung.

Verwenden Sie eine *if*-Anweisung, um zu bestimmen, ob ein Dienst angehalten werden soll. Da die Eigenschaft *AcceptStop* ein boolescher Wert (*true/false*) ist, können Sie die Syntax vereinfachen und das *if*-Format (*true*) verwenden. Dieses Format ist einfacher als beispielsweise folgender Befehl:

```
If ( $objWMIService.acceptStop -eq "true")
```

Der Code ist der gleiche, aber die Eingabe ist kürzer und der Code ist besser lesbar. Die *if*-Anweisung lautet:

```
if( $objWMIService.Acceptstop )
```

Öffnen Sie anschließend eine geschweifte Klammer und geben Sie mit dem Cmdlet **Write-Host** eine Meldung aus, die besagt, dass Sie versuchen, den in der Variablen *$strService* angegebenen Dienst zu beenden. Rufen Sie die *stopService*-Methode aus der *Win32_Service*-Klasse auf und beenden Sie den angegebenen Dienst. Die Variable *$rtn* erfasst die Statusinformationen für den Methodenaufruf. Das Ergebnis 0 zeigt an, dass keine Fehler aufgetreten sind. Alle anderen Werte müssen überprüft werden.

Überprüfen Sie den vom *stopService*-Methodenaufruf zurückgegebenen Code mit einer *switch*-Anweisung. Wenn die Eigenschaft *ReturnValue* den Wert 0 hat, geben Sie mit **Write-Host** eine Meldung aus, dass keine Fehler aufgetreten sind und die Methode erfolgreich abgeschlossen wurde. Überprüfen Sie die allgemeineren Fehler und geben Sie die entsprechende Meldung aus. Wenn ein unerwarteter Fehler aufgetreten ist, geben Sie mit dem Standardparameter die Fehlernummer aus. Die *switch*-Anweisung:

```
Switch ($rtn.returnvalue)
  {
   0 { Write-Host -foregroundcolor green "Der Dienst $strService wurde beendet." }
   2 { Write-Host -foregroundcolor red "Der Dienst $strService berichtet," `
       " dass der Zugriff verweigert wurde." }
   5 { Write-Host -ForegroundColor red "Der Dienst $strService kann" `
       " zu diesem Zeitpunkt keine Steuerungsanforderungen verarbeiten." }
   10 { Write-Host -ForegroundColor red "Der Dienst $strService ist bereits" `
         " beendet." }
   DEFAULT { Write-Host -ForegroundColor red "Der Dienst $strService berichtet" `
             " ERROR $($rtn.returnValue)" }
  }
```

Wenn der Dienst den *Stop*-Befehl nicht akzeptiert, geben Sie mit der *else*-Anweisung und dem Cmdlet **Write-Host** den Namen des Dienstes zusammen mit einer entsprechenden Meldung aus. Dieses Problem sollte nur auftreten, wenn der Dienst den *Stop*-Befehl nicht akzeptieren kann. Beachten Sie, dass der Dienst für den *Stop*-Befehl konfiguriert sein kann, aber diesen möglicherweise zu diesem Zeitpunkt nicht akzeptiert. Dies kann auftreten, wenn der Dienstcontroller von einem anderen Dienst ausgelastet wird. In diesem Fall sollte der Fehlercode 5 zurückgegeben werden, der von der *switch*-Anweisung ausgewertet wird. Das vollständige Skript *CheckServiceThenStop.ps1* hat folgenden Aufbau:

CheckServiceThenStop.ps1

```
$strService = "bits"
$strComputer = "localhost"
$strClass = "win32_service"
$objWmiService = Get-Wmiobject -Class $strClass -computer $strComputer `
  -filter "name = '$strService'"
if( $objWMIService.Acceptstop )
 {
  Write-Host "Der Dienst $strService wird jetzt beendet ..."
  $rtn = $objWMIService.stopService()
  Switch ($rtn.returnvalue)
  {
   0 { Write-Host -foregroundcolor green "Der Dienst $strService wurde beendet." }
   2 { Write-Host -foregroundcolor red "Der Dienst $strService berichtet," `
       " dass der Zugriff verweigert wurde." }
```

```
   5 { Write-Host -ForegroundColor red "Der Dienst $strService kann" `
       " zu diesem Zeitpunkt keine Steuerungsanforderungen verarbeiten." }
   10 { Write-Host -ForegroundColor red "Der Dienst $strService ist bereits" `
         " beendet." }
   DEFAULT { Write-Host -ForegroundColor red "Der Dienst $strService berichtet" `
             " ERROR $($rtn.returnValue)" }
  }
 }
ELSE
 {
  Write-Host "Der Dienst $strService akzeptiert gegenwärtig keine Stop-Anforderungen."
 }
```

Starten von Diensten

Wie zum Beenden von Diensten sind in Windows PowerShell auch zum Starten von Diensten zwei Methoden verfügbar. Die einfachste Methode zum Starten eines Dienstes in Windows PowerShell bietet das Cmdlet **Start-Service**. Geben Sie hierzu entweder den Dienstnamen oder den Anzeigenamen ein. Wie beim Cmdlet **Stop-Service** ist die Eingabe des Dienstnamens normalerweise am einfachsten.

StartService.ps1 verwendet die Variable *$strService* zum Speichern des Namens des Dienstes, der gestartet werden soll. In diesem Beispiel wird der BITS-Dienst gestartet. Nachdem Sie den Dienst der Variablen zugewiesen haben, starten Sie den Dienst mit dem Cmdlet **Start-Service**. Geben Sie den Parameter **-name** und die Variable *$strService* an. Das Skript *StartService.ps1* besteht aus folgenden Codezeilen:

StartService.ps1

```
$strService = "bits"
Start-Service -Name $strService
```

Um mehrere Dienste zu starten, können Sie ein Array mit Dienstnamen erstellen, das Array mit *foreach* durchlaufen und die Dienste mit dem Cmdlet **Start-Service** starten. Dieser Vorgang wird vom Skript *StartMultipleServices.ps1* ausgeführt.

Beispielsweise können Sie im Skript *StartMultipleServices.ps1* die Namen von drei Diensten der Variablen *$aryServices* zuweisen. Sie können dann anschließend die *foreach*-Anweisung verwenden, um das Array mit der Variablen *$strService* als Enumerator zu durchlaufen. Verwenden Sie im Skriptblock für die *foreach*-Anweisung das Cmdlet **Write-Host**, um eine Meldung mit dem Namen des Dienstes auszugeben, der gestartet wird. Starten Sie den Dienst unter Angabe des Namens mit dem Cmdlet **Start-Service**. Das Skript *StartMultipleServices.ps1* verwendet folgende Anweisungen:

StartMultipleServices.ps1

```
$aryServices = "bits", "wuauserv", "CcmExec"
foreach ($strService in $aryServices)
{
 Write-Host "Der Dienst $strService wird gestartet ..."
 Start-Service -Name $strService
}
```

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Ordnungsgemäßes Starten

Wie beim Überprüfen, ob ein Dienst den *Stop*-Befehl akzeptiert, ist es wichtig, sicherzustellen, dass der Dienst den *Start*-Befehl annehmen kann. Sie müssen die folgenden beiden Bedingungen überprüfen: Ob der Dienst bereits ausgeführt wird und ob der Dienst deaktiviert ist. Diese beiden Bedingungen generieren eine Fehlermeldung.

Ein Problem mit dem Skript *StartMultipleServices.ps1* besteht darin, dass dieses ohne weitere Überprüfung des Dienstestatus versucht, den Dienst zu starten. Dieses Problem ist zwar nicht schwerwiegend, kann aber die Ausführung des Skripts verzögern. Um dieses Problem zu beheben, sollten Sie den Dienststatus mit **Get-Service** überprüfen. Wenn der Dienst bereits ausgeführt wird, geben Sie den Status in einer Meldung aus. Wird der Dienst hingegen nicht ausgeführt, können Sie diesen starten.

Um das Skript *CheckServiceThenStart.ps1* auszuführen, speichern Sie den Dienstnamen in der Variablen *$strService* und rufen Sie die Informationen über den gewünschten Dienst mit dem Cmdlet **Get-Service** ab. Geben Sie hierzu den Namen des Dienstes an. Nachdem Sie die Dienstinformationen abgerufen haben, fügen Sie das Objekt in das Cmdlet **ForEach-Object** ein. Sie müssen diesen Vorgang auch dann ausführen, wenn die Pipeline nur ein Objekt umfasst. Wenn Sie das Cmdlet **ForEach-Object** nicht ausführen, wird die in Abbildung 4.9 dargestellte Fehlermeldung angezeigt.

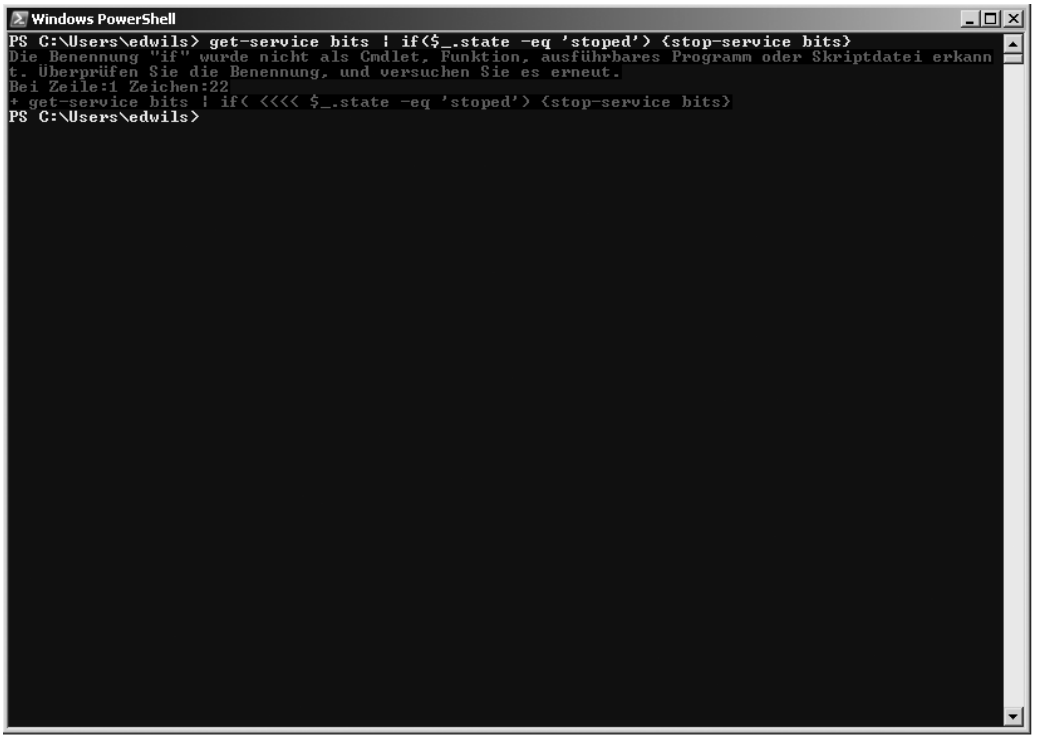


Abbildung 4.9 Sie können in die *if*-Anweisung kein Objekt einfügen

Verwenden Sie im Skriptblock des Cmdlets **ForEach-Object** die *if*-Anweisung, um die Eigenschaft *Status* des aktuellen Pipelineobjekts auszuwerten. Bei diesem Objekt handelt es sich im gegebenen

Beispiel um den BITS-Dienst. Wenn dieser Dienst nicht ausgeführt wird, geben Sie mit dem Cmdlet **Write-Host** eine Meldung aus, dass der Dienst gestartet wird. Der Dienst wird mit dem Cmdlet **Start-Service** gestartet. Geben Sie mit dem Parameter **-name** den zu startenden Dienst an.

Wenn der Dienst ausgeführt wird, geben Sie diese Informationen mit dem Cmdlet **Write-Host** aus. Das Skript *CheckServiceThenStart.ps1* hat folgenden Aufbau:

CheckServiceThenStart.ps1

```
$strService = "bits"

Get-Service -name $strService |
Foreach-object { if ($_.status -ne "running")
{
 Write-Host "Der Dienst $strService wird gestartet ..."
 Start-Service -Name $strService
}
 ELSE
{
 Write-Host "Der Dienst $strService ist bereits gestartet."
}
}
```

Zwei Methoden zum Arbeiten mit Diensten

Eine der Schwierigkeiten, die Anfänger mit der Windows PowerShell haben, ist, dass Ergebnisse auf mehrere Arten erreicht werden können. Ich wähle das Verfahren zum Ausführen eines Prozesses basierend auf zwei Kriterien aus: Die einfachste Methode und die Methode, mit der ich am besten vertraut bin. Diese Entscheidung ist jedoch manchmal nicht so einfach. Um beispielsweise den Status eines Dienstes zu überprüfen, kann ich folgenden Befehl ausführen:

```
Get-service bits
```

Dieser Befehl zeigt den Status des BITS-Dienstes und den Anzeigenamen an. Wenn ich jedoch herausfinden möchte, ob der Dienst automatisch gestartet wird, übergebe ich das Objekt an das Cmdlet **Format-List**. Beispiel:

```
Get-Service bits | Format-List *

Name                : BITS
CanPauseAndContinue : False
CanShutdown         : True
CanStop             : True
DisplayName         : Intelligenter Hintergrundübertragungsdienst
DependentServices   : {}
MachineName         : .
ServiceName         : BITS
ServicesDependedOn  : {EventSystem, RpcSs}
ServiceHandle       :
Status              : Running
ServiceType         : Win32ShareProcess
Site                :
Container           :
```

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Wie Sie möglicherweise bemerkt haben, wird der Startmodus des Dienstes nicht aufgeführt. Zum Abrufen dieser Informationen müssen Sie WMI verwenden. WMI ist oft leistungsfähiger als die in Windows PowerShell integrierten Cmdlets. Die Cmdlets sind für einfachere Verwendungsszenarien und allgemeine Verwaltungsanforderungen ausgelegt. Bisweilen hat dies jedoch die Einbuße anspruchsvollerer Methoden zur Folge.

Windows PowerShell kann WMI jedoch in vollem Umfang nutzen. Mit folgendem WMI-Befehl können Sie alle Eigenschaften des BITS-Dienstes abrufen:

```
Get-WmiObject win32_service -Filter "name = 'bits'" | fl [a-z]*

Name                    : BITS
Status                  : OK
ExitCode                : 0
DesktopInteract         : False
ErrorControl            : Normal
PathName                : C:\Windows\System32\svchost.exe -k netsvcs
ServiceType             : Share Process
StartMode               : Auto
AcceptPause             : False
AcceptStop              : True
Caption                 : Intelligenter Hintergrundübertragungsdienst
CheckPoint              : 0
CreationClassName       : Win32_Service
Description             : Überträgt Dateien im Hintergrund unter Verwendung von sich in Leerlauf
befindender Netzwerkbandbreite. Falls dieser Dienst deaktiviert wird,
können von BITS abhängige Anwendungen wie Windows Update oder MSN Explorer
Programme und andere Informationen nicht automatisch heruntergeladen werden.
DisplayName             : Intelligenter Hintergrundübertragungsdienst
InstallDate             :
ProcessId               : 1096
ServiceSpecificExitCode : 0
Started                 : True
StartName               : LocalSystem
State                   : Running
SystemCreationClassName : Win32_ComputerSystem
SystemName              : M5-1875135
TagId                   : 0
WaitHint                : 0
```

Verzweifeln Sie also nicht, wenn Sie kein geeignetes Windows PowerShell-Cmdlet finden. Sie können stattdessen möglicherweise WMI oder ein andere Technologie verwenden. Wenn ein Prozess zur Verfügung steht, können Sie diesen auch in Windows PowerShell ausführen.

Mit dem Cmdlet **Get-WmiObject** und der WMI-Klasse *Win32_Service* können Sie herausfinden, ob ein Dienst deaktiviert ist, den Startmodus ändern und den Dienst starten. Diese Vorgänge werden vom Skript *ChangeModeThenStart.ps1* ausgeführt.

Das Skript *ChangeModeThenStart.ps1* beginnt mit einer benutzerdefinierten Funktion. Diese Funktion wertet den von den Methoden *changeStartMode()* und *startService()* zurückgegebenen Code aus. Beide Methoden verwenden die gleichen Rückgabecodewerte. Rufen Sie die Funktion *FunEvalRTN* auf und übergeben Sie den zurückgegebenen Wert an die Variable *$rtn*. Werten Sie den zurückgegebenen Wert dann mit einer *switch*-Anweisung aus. Der Wert 0 zeigt an, dass keine Fehler aufgetreten sind. Alle anderen Werte weisen auf einen Fehler im Methodenaufruf hin. Wenn keine Fehler aufgetreten sind, wird die

Meldung in Grün ausgegeben. Ein interessantes Feature dieser Funktion ist die Verwendung der Variablen *$strCall*. Nachdem die Methoden aufgerufen wurden, können Sie der Variablen *$strCall* eine Zeichenfolge zuweisen, die die aufgerufene Methode angibt.

Der Funktionsaufruf ist allerdings nicht die erste Codeanweisung, sondern die Zuweisung des Werts *BITS* zur Variablen *$strService*. (Dieser Code ist im Abschnitt „Starten von Diensten“ beschrieben). Der WMI-Abschnitt des Skripts ruft die WMI-Informationen über *Win32_Service* unter Verwendung der Variablen *$strService* ab. Wenn der BITS-Dienst nicht ausgeführt wird und deaktiviert ist, müssen Sie den Startmodus in *Manuell* ändern, bevor Sie den Dienst starten können. Die Codezeile zum Überprüfen des Dienstestatus lautet:

```
if( $objWMIService.state -ne 'running' -AND $objWMIService.startMode -eq 'Disabled')
```

Hinweis Beachten Sie beim Ausführen von *if*-Anweisungen, dass der Parameter *-AND* ähnlich wie die Parameter *-ne* (ungleich) und *-eq* (gleich) verwendet wird. Deshalb muss diesem Parameter ein Bindestrich vorangestellt sein.

Wenn der Dienst nicht ausgeführt wird und deaktiviert ist, müssen Sie den Startmodus ändern. Verwenden Sie hierzu die *changeStartMode()*-Methode der WMI-Klasse *Win32_Service*. Nach dem Aufruf dieser Methode, gibt diese ein Objekt mit einem Codewert zurück. Der vom Methodenaufruf zurückgegebene Code ist in der Eigenschaft *ReturnValue* gespeichert. Systemeigenschaften, die Informationen zum WMI-Aufruf anzeigen, sind am doppelten Unterstrich vor dem zurückgegebenen Objekt erkennbar. Das *Management*-Objekt wird wie folgt verwendet:

```
$a = (Get-WmiObject win32_service -filter "name = 'bits'").changeStartMode()
$a | get-member

   TypeName: System.Management.ManagementBaseObject#\__PARAMETERS

Name             MemberType Definition
----             ---------- ----------
ReturnValue      Property   System.UInt32 ReturnValue {get;set;}
__CLASS          Property   System.String __CLASS {get;set;}
__DERIVATION     Property   System.String[] __DERIVATION {get;set;}
__DYNASTY        Property   System.String __DYNASTY {get;set;}
__GENUS          Property   System.Int32 __GENUS {get;set;}
__NAMESPACE      Property   System.String __NAMESPACE {get;set;}
__PATH           Property   System.String __PATH {get;set;}
__PROPERTY_COUNT Property   System.Int32 __PROPERTY_COUNT {get;set;}
__RELPATH        Property   System.String __RELPATH {get;set;}
__SERVER         Property   System.String __SERVER {get;set;}
__SUPERCLASS     Property   System.String __SUPERCLASS {get;set;}
```

Nachdem Sie die *changeStartMode()*-Methode ausgeführt haben, weisen Sie dem Parameter *$strCall* eine Zeichenfolge zu, um die ausgeführte Prozedur anzugeben. Rufen Sie die *FunEvalRTN* auf und übergeben Sie das Objekt in der Variablen *$rtn*. Diese Funktion wandelt zahlreiche der allgemeinen Rückgabecodes aus dem Methodenaufruf um.

Nachdem die Funktion beendet ist, fahren Sie mit dem nächsten Skriptabschnitt fort. Wenn der Wert 0 zurückgegeben wird, starten Sie den Dienst mit der *startService()*-Methode und rufen die Funktion *FunEvalRTN* auf, um die Ergebnisse des Methodenaufrufs auszuwerten.

Wenn der Startmodus nicht auf deaktiviert festgelegt ist, rufen Sie die *startService()*-Methode und anschließend die Funktion auf, um den Rückgabecode zu überprüfen. Das vollständige Skript *ChangeModeThenStart.ps1* lautet:

ChangeModeThenStart.ps1

```
function FunEvalRTN($rtn)
{
Switch ($rtn.returnvalue)
  {
   0 { Write-Host -foregroundcolor green "Keine Fehler beim $strCall." }
   2 { Write-Host -foregroundcolor red "Der Dienst $strService berichtet," `
       " dass der Zugriff verweigert wurde." }
   5 { Write-Host -ForegroundColor red "Der Dienst $strService kann" `
       " zum gegenwärtigen Zeitpunkt keine Steuerungsanweisungen verarbeiten." }
   10 { Write-Host -ForegroundColor red "Der Dienst $strService ist bereits" `
         " gestartet." }
   14 { Write-Host -ForegroundColor red "Der Dienst $strService ist deaktiviert." }
   DEFAULT { Write-Host -ForegroundColor red "Der Dienst $strService berichtet: " `
         " ERROR $($rtn.returnValue)" }
  }
  $rtn=$strCall=$null
}


$strService = "BITS"
$strComputer = "localhost"
$strClass = "win32_service"
$objWmiService = Get-Wmiobject -Class $strClass -computer $strComputer `
  -filter "name = '$strService'"

if( $objWMIService.state -ne 'running' -AND $objWMIService.startMode -eq 'Disabled')
  {
   Write-Host "Der Dienst $strService ist deaktiviert. Ändern des Startmodus in Manuell ..."
   $rtn = $objWmiService.ChangeStartMode("Manual")
   $strCall = "Ändern der Dienstkonfiguration auf Manuell."

   FunEvalRTN($rtn)

   if($rtn.returnValue -eq 0)
     {
      Write-Host "Der Dienst $strService wird nicht ausgeführt. Versuche den Dienst zu starten ..."
      $rtn = $objWMIService.StartService()
      $strCall = "Starten des Dienstes."

   FunEvalRTN($rtn)

      }

   }
ELSEIF($objWMIService.state -ne 'running')
  {
   Write-Host "Der Dienst $strService wird nicht ausgeführt. Versuche den Dienst zu starten ..."
   $rtn = $objWMIService.StartService()
   $strCall = "Starten des Dienstes."

   FunEvalRTN($rtn)

  }
ELSEIF($objWMIService.state -eq 'running')
```

```
  {
   Write-Host "Der Dienst $strService wird bereits ausgeführt."
  }
ELSE
  {
   Write-Host "Der Dienst $strService befindet sich im Übergangszustand."
  }
```

Verwalten der Konfiguration

Um die Dienstkonfiguration auf einem Server festzulegen, muss Ihnen der Status aller Dienste bekannt sein. Es ist wichtig, zu wissen, welche Dienste ausgeführt, beendet oder deaktiviert sind. Sie müssen jedoch auch die Abhängigkeiten und die jeweiligen Dienstkonten kennen. Außerdem ist es sinnvoll, die aktuelle Konfiguration mit der dokumentierten Konfiguration zu vergleichen. Eine einfache Methode besteht darin, ein Skript auf einem Server auszuführen, auf dem die zu dokumentierenden Konfigurationseinstellungen vorgenommen wurden. Hierbei kann es sich um eine Arbeitsstation, den zu verwaltenden Server oder einen anderen Server mit einer ähnlichen Konfiguration handeln. Wählen Sie den Namen und den Status der Dienste auf dem Computer aus und zeichnen Sie die Informationen in einer Textdatei auf. Das Skript *WriteServiceStatus.ps1* führt diese Schritte aus:

WriteServiceStatus.ps1

```
$strPath = "c:\FSO\Dcm1.txt"
Get-Service |
format-table name, status -autosize |
Out-File -FilePath $strPath
```

Wie können Sie die Datei jedoch verwenden, nachdem Sie die Dienste auf dem Computer aufgelistet haben? Analysieren Sie die Datei, suchen Sie den Dienstnamen und vergleichen Sie die dazugehörigen Informationen mit dem Status auf dem vorherigen Computer. Am leichtesten läßt sich dieses mit dem Cmdlet **Compare-Object** erledigen. Wenn Sie das Skript *WriteServiceStatus.ps1* zum Erstellen einer Textdatei mit einer Liste der Dienste und des jeweiligen Dienststatus auf dem Computer verwenden, können Sie die beiden Textdateien mit dem Skript *CompareServicesTxt.ps1* vergleichen. Sie können die beiden Computer vergleichen oder nach Änderungen auf einem Computer suchen. Die Methode eignet sich für die Problembehandlung und die Überwachung.

Weisen Sie im Skript *CompareServicesTxt.ps1* den Pfad zu den Konfigurationsdateien den Variablen zu. Speichern Sie den Pfad zur Referenzkonfiguration auf dem primären oder sekundären Computer in der Variablen *$strReference* und die Liste der zu überprüfenden Dienste in der Variablen *$strDifference*. Nachdem Sie den Variablen die Werte zugewiesen haben, vergleichen Sie mit dem Cmdlet **Compare-Object** den Inhalt der beiden Dateien, indem Sie mit dem Parameter **-referenceobject** auf die Textdatei für die Basiskonfiguration verweisen. Der Parameter **-differenceobject** verweist auf die Datei für die aktuelle Konfiguration. Wenn die Parameter **-referenceobject** und **-differenceobject** auf die Variablen mit dem Pfad zu den Konfigurationsdateien verweisen, ist der Vergleich nicht besonders aussagekräftig. Da das Cmdlet jedoch zum Vergleichen von Objekten ausgelegt ist, müssen Sie Objekte für das Cmdlet erstellen. Verwenden Sie hierzu das Cmdlet **Get-Content**, um die Referenzdatei und die Vergleichsdatei zu öffnen und zu lesen. Das Graviszeichen im Skript *CompareServicesTxt.ps1* verbessert die Lesbarkeit. Das Skript *CompareServicesTxt.ps1* verwendet folgende Anweisungen:

CompareServicesTxt.ps1

```
$strReference =  "c:\FSO\Dcm.txt"
$strDifference = "c:\FSO\Dcm1.txt"

Compare-Object `
-referenceobject $(get-content $strReference) `
-differenceobject $(get-content $strDifference)
```

Überprüfen, ob die Dienste beendet wurden

Um zu überprüfen, ob die gewünschten Dienste beendet wurden, können Sie eine Liste der entsprechenden Dienste kompilieren. Erfassen Sie hierzu unter Verwendung des Skripts *WriteStoppedServices.ps1* die derzeit beendeten Dienste in einer Textdatei, die Sie gegebenenfalls bearbeiten können.

Weisen Sie mit dem Skript *WriteStoppedServices.ps1* die Zeichenfolge *stopped* der Variablen *$strState* zu. Geben Sie die Zeichenfolge mit dem Pfad, einschließlich dem Dateinamen, in der Variablen *$strPath* an, um die Datei zu speichern, die die Liste der beendeten Dienste enthält. Rufen Sie mit dem Cmdlet **Get-WmiObject** die WMI-Klasse *Win32_Service* ab. Rufen Sie mit dem Filter **-filter** nur die Dienste ab, deren Status mit dem in der Variablen *$strState* definierten Status identisch ist. Nachdem Sie alle zu beendenden Dienste abgerufen haben, fügen Sie das Objekt in das Cmdlet **Select-Object** ein und rufen Sie nur die Namen der Dienste ab. Fügen Sie das Ergebnis in das Cmdlet **Out-File** ein und verweisen Sie mit dem Parameter **-filepath** auf den in der Variablen *$strpath* angegebenen Pfad. Das Skript *WriteStoppedServices.ps1* hat folgenden Aufbau:

WriteStoppedServices.ps1

```
$strState = "stopped"
$strPath = "C:\FSO\StoppedServices.txt"
Get-WmiObject win32_service -Filter "state='$strState'" |
select-object name |
Out-File -FilePath $strPath
```

Sie können die Liste mit den beendeten Diensten in ein Skript eingeben, dann auf die Dienstinformationen zugreifen und den Status überprüfen, um sicherzustellen, dass die Dienste beendet wurden. Diese Vorgänge werden mit Skript *CheckStoppedServices.ps1* ausgeführt, das im nächsten Abschnitt beschrieben ist.

Lesen einer Datei, um den Dienststatus zu überprüfen

Weisen Sie mit dem Skript *CheckStoppedServices.ps1* die Zeichenfolge mit dem Pfad zur Datei, die die Liste der zu beendenden Dienste enthält, der Variablen *$strFile* zu. Rufen Sie mit dem Cmdlet **Get-Content** den Inhalt der Datei ab, der von *$strFile* repräsentiert wird. Führen Sie anschließend das Cmdlet **ForEach-Object** aus, um die von **Get-Content** zurückgegebenen Objekte zu überprüfen. Verwenden Sie beim Durchlaufen der Dienstnamen die Methode *trimend()*, um die Leerzeichen am Ende der Zeilen in der Textdatei zu entfernen. Dies ist erforderlich, da das Cmdlet **Out-File** zahlreiche Leerzeichen in die Textzeilen einfügt. Speichern Sie den Text der WMI-Abfrage in der Variablen *$strQuery* und führen Sie die Abfrage mit dem Cmdlet **Get-WmiObject** aus. Um die Abfrage für das Cmdlet **Get-WmiObject** festzulegen, geben Sie den Parameter **-query** an und fügen Sie die in der Variablen *$strQuery* gespeicherte Zeichenfolge ein.

Nachdem die WMI-Abfrage abgeschlossen ist, fügen Sie die Ergebnisse in das Cmdlet **ForEach-Object** ein und überprüfen Sie mit der *if*-Anweisung, ob der Dienst den Status *stopped* hat. Geben Sie in diesem Fall mit dem Cmdlet **Write-Host** eine Meldung aus, dass der Dienst beendet wurde. Um die

Meldung etwas interessanter zu machen, rufen Sie mit folgendem Code die Eigenschaft *Name* aus dem aktuellen Pipeline-Objekt ab:

```
{ Write-Host $_.name "ist noch immer beendet." }
```

Wenn der Status des Dienstes nicht *stopped* ist, wird der Dienst ausgeführt oder ist angehalten. Geben Sie mit dem Cmdlet **Write-Host** den Namen des Dienstes und den entsprechenden Status aus. Mit dem Parameter **-foregroundcolor** können Sie die Meldung in Rot anzeigen.

CheckStoppedServices.ps1

```
$strFile = "c:\FSO\StoppedServices.txt"
Get-Content  $strFile |
foreach-object { $strService = $_.trimend()
$strQuery = "Select * from win32_service where name ='$strService'"
get-wmiobject -query $strQuery |
foreach-object `
{
 if ($_.state -eq "stopped" )
  { Write-Host $_.name "ist noch immer beendet." }
 ELSE
  { Write-Host -foregroundcolor RED $_.name `
    " ist nicht mehr beendet. Der aktuelle Status ist: $($_.state)" }
}
}
```

Überprüfen, ob die Dienste ausgeführt werden

Um den Status der Dienste zu überprüfen, die ausgeführt werden sollten, listen Sie als Erstes die entsprechenden Dienste auf. Sie müssen die Textdatei überprüfen, in der die Dienste aufgeführt sind, und eine WMI-Abfrage ausführen, die den Status dieser Dienste zurückgibt. Stellen Sie anschließend sicher, dass die Dienste ausgeführt werden.

Diese Vorgänge werden vom Skript *CompareRunningServices.ps1* ausgeführt. Weisen Sie den Pfad zur Textdatei der Variablen *$strFile* zu. Fügen Sie den in der Variablen *$strFile* gespeicherten Pfad in das Cmdlet **Get-Content** ein. Übergeben Sie das vom Cmdlet **Get-Content** zurückgegebene Objekt an das Cmdlet **ForEach-Object**. Entfernen Sie im Codeblock für das Cmdlet **ForEach-Object** die Leerzeichen nach den Dienstnamen. Verwenden Sie hierzu die *trimend()*-Methode. Rufen Sie die Methode in der Variablen *$_* auf, die das aktuelle Pipeline-Objekt repräsentiert. Das Skript CompareRunningServices.ps1 hat folgenden Aufbau:

CompareRunningServices.ps1

```
$strFile = "c:\FSO\RunningServices.txt"
Get-Content  $strFile |
Foreach-object { $strService = $_.trimend()
$strQuery = "Select * from win32_service where name ='$strService'"
get-wmiobject -query $strQuery |
foreach-object `
{
 if ($_.state -eq "running" )
  { Write-Host $_.name "wird noch immer ausgeführt." }
 ELSE
  { Write-Host -foregroundcolor RED $_.name `
    " wird nicht mehr länger ausgeführt. Der aktuelle Status ist: $($_.state)" }
}
}
```

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Bestätigen der Konfiguration

Die Dienstkonfiguration ist ein äußerst wichtiger Sicherheitsaspekt. Eine entscheidende Sicherheitsmaßnahme ist das Verringern der *Angriffsfläche*. Windows Server 2008 Core Edition (Server Core) ist unter anderem aufgrund seiner reduzierten Angriffsfläche so beliebt. Da die Dienstkonfiguration zum Verringern der Angriffsfläche ausgesprochen wichtig ist, müssen Sie sich folgende drei Fragen zur Konfiguration stellen:

- Wie wird der Dienst gestartet (automatisch, manuell, deaktiviert)?
- Unter welchem Konto wird der Dienst ausgeführt (lokales System, Netzwerkdienst, lokaler Dienst, benutzerdefiniert)?
- Welches Kennwort wird für den Dienst verwendet (automatisch, benutzerdefiniert)?

Erstellen eines Ausnahmeberichts

Um einen Zusammenfassungsbericht mit den Konfigurationsinformationen der ermittelten Berichte zu erstellen, müssen Sie jeden Dienst auflisten und dessen Startmodus überprüfen. Wenn der Dienst mit einem benutzerdefinierten Konto gestartet wird, müssen Sie diese Informationen ebenfalls erfassen. Verwenden Sie beispielsweise das Skript *EvaluateServicesAndCount.ps1*, welches das Cmdlet **Get-WmiObject** verwendet, um die *Win32_Service*-Klasse abzurufen, und speichern Sie das resultierende Objekt in der Variablen *$objWMIService*.

Verwenden Sie eine *foreach*-Anweisung und durchlaufen Sie die Dienste. Analysieren Sie das jeweilige Objekt mit zwei *switch*-Anweisungen. Suchen Sie in der ersten *switch*-Anweisung nach *startmode*. Wenn *startmode* auf auto festgelegt ist, erhöhen Sie die *$a*-Indikatorvariable (auto) und fügen Sie den Namen zur Variablen *$auto* hinzu, die eine Liste der automatisch gestarteteten Dienste enthält. Um die Dienstnamen in jeweils einer Zeile auszugeben, verwenden Sie die Zeichenkombination Gravis+n (`n).

Verwenden Sie die gleiche Methode für manuell gestartete und deaktivierte Dienste. Die *switch*-Anweisung hat folgendes Format:

```
switch ($i.startmode)
{
 "auto"     { $a++ ; $auto+="$($i.name)`n"}
 "manual"   { $m++ ; $manual+="$($i.name)`n"}
 "disabled" { $d++ ; $disabled+="$($i.name)`n"}
 DEFAULT { }
}
```

Überprüfen Sie mit der zweiten *switch*-Anweisung das Benutzerkonto, unter dem der Dienst ausgeführt wird. Suchen Sie mit einem regulären Ausdruck nach den Dienstkontonamen. Wenn eine Übereinstimmung gefunden wird, erhöhen Sie eine Indikatorvariable. Wenn das Konto nicht *localsystem*, *localservice* oder *networkservice* ist, handelt es sich um ein benutzerdefiniertes Konto, das für die allgemeine Sicherheitskonfiguration und insbesondere für die Kennwortverwaltung eingehend überprüft werden sollte. Die zweite *switch*-Anweisung hat folgendes Format:

```
switch -regex ($i.startName)
{
 "localsystem"    { $lsys++ }
 "localservice"   { $lsvc++ }
 "NetworkService" { $nsvc++ }
 DEFAULT           { $osn++ ; $otherServiceNames+="$($i.startName)`n"}
}
```

Der nächste Skriptabschnitt erstellt die Ausgabe. Um den Formatierungsaufwand für den Bericht zu verringern, speichern Sie die Ausgabe in einer *here*-Zeichenfolge, die Ihnen die Eingabe ohne spezielle Satzzeichen und Syntax ermöglicht.

Wenn keine benutzerdefinierten Dienstkonten vorhanden sind, müssen Sie keine Erinnerung zum Überprüfen der Kennwörter ausgeben. Sollten jedoch benutzerdefinierte Dienstkonten vorhanden sein, müssen Sie eine Erinnerung generieren. Verwenden Sie hierzu eine *if*-Anweisung sowie die Anweisung += am Ende der Variablen *$string*. Die Warnung befindet sich in einer separaten *here*-Zeichenfolge:

```
if($osn -ne 0)
{
$string+= @"

Die anderen verwendeten IDs werden hier aufgeführt:
$otherServiceNames

Sie sollten die von den folgenden Diensten verwendeten Kennworte überprüfen:
$otherServiceNames
"@
}
```

Das vollständige Skript *EvaluateServicesAndCount.ps1* ist in folgendem Listing dargestellt. Das Skript sollte in WordPad geöffnet werden, da das Zeichen für eine neue Zeile (`n) in Notepad nicht richtig ausgegeben wird.

EvaluateServicesAndCount.ps1

```
$a=$m=$d=0
$lsvc=$lsys=$nsvc=$osn=0
$objWMIService = Get-WmiObject -Class win32_service -computer localhost

foreach ($i in $objWMIService)
{
switch ($i.startmode)
{
 "auto"     { $a++ ; $auto+="$($i.name)`n"}
 "manual"   { $m++ ; $manual+="$($i.name)`n"}
 "disabled" { $d++ ; $disabled+="$($i.name)`n"}
 DEFAULT { }
}
switch -regex ($i.startName)
{
 "localsystem"    { $lsys++ }
 "localservice"   { $lsvc++ }
 "NetworkService" { $nsvc++ }
 DEFAULT           { $osn++ ; $otherServiceNames+="$($i.startName)`n"}
}
}

$string =  @"
Es sind $($objWMIService.length) Dienste definiert.
Diese werden wie folgt gestartet:
Automatisch $a Manuell $m Deaktiviert $d

Die automatisch gestarteten Dienste:
--------------------------
$auto
```

```
Die manuell gestarteten Dienste:
------------------------
$manual

Die deaktivierten Dienste:
--------------------------
$disabled


Die Dienste verwenden die folgenden Systemkonten:
 localsystem: $lsys Dienste.
 localService: $lsvc Dienste.
 networkService: $nsvc Dienste.
 Andere Benutzer-IDs: $osn Dienste.
"@

if($osn -ne 0)
{
$string+= @"

Die anderen verwendeten IDs werden hier aufgeführt:
$otherServiceNames

Sie sollten die von den folgenden Diensten verwendeten Kennworte überprüfen:
$otherServiceNames
"@
}
Out-File -InputObject $string -FilePath c:\FSO\exceptopn.txt
```

Zusammenfassung

In diesem Kapitel wurden die verschiedenen Dienste beschrieben, die auf einem Server oder einer Arbeitsstation gestartet und ausgeführt werden. Es wurden sowohl die zum Dokumentieren der vorhandenen Konfiguration erforderlichen Schritte als auch die Startmodi, die Sicherheitsaspekte und die Anmeldeinformationen für die Dienste erklärt. Außerdem wurde das Ändern dieser Einstellungen mit Skripts und die Verwendung einer Datenbank zum Sicherstellen der Konsistenz in einem Windows-Unternehmensnetzwerk beschrieben.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

KAPITEL 5

Verwalten von Freigaben

Nach Abschluss dieses Kapitels können Sie:

- Die auf einem System vorhandenen Freigaben dokumentieren
- Die benutzerdefinierten Freigaben dokumentieren
- Feststellen, ob administrative Freigaben vorhanden sind
- Freigaben überwachen
- Freigaben ändern
- Neue Freigaben erstellen
- Vorhandene Freigaben löschen

Auf der Begleit-CD Alle Skripts in diesem Kapitel befinden sich auf der Begleit-CD im Ordner *\Scripts\Chapter05*.

Dokumentieren von Freigaben

Es gibt mehrere Gründe, warum ein Netzwerkadministrator die Freigaben auf einem Server oder einer Arbeitsstation dokumentieren sollte. Der Administrator muss wissen, wie viele Freigaben vorhanden sind und welche Laufwerke und Ordner auf einem Computer freigegeben wurden. Außerdem müssen Freigaben aus Sicherheitsgründen überprüft werden. Nach dem Dokumentieren der Freigaben stellen sich häufig folgende Fragen:

- Welche Freigaben sind erforderlich?
- Wer kann auf die Freigaben zugreifen?
- Welche Sicherheitseinstellungen sind für die Freigaben konfiguriert?
- Welche Arten von Dokumenten werden über die Freigaben bereitgestellt?

Um Informationen zu den Freigaben abzurufen, verwenden Sie die WMI-Klasse *Win32_Share*. Mit dem Cmdlet *Get-WmiObject* können Sie ein Objekt der Klasse *Win32_Share* für den Zugriff auf diese Informationen ermitteln. Beispielsweise können Sie ein Skript namens *ListShares.ps1* mit dem Cmdlet **Get-WmiObject** beginnen, um die WMI-Klasse *Win32_Share* abzufragen. Um das Skript für den lokalen Computer auszuführen, geben Sie *localhost* als Computernamen an. Das von dieser Abfrage zurückgegebene *Management*-Objekt kann dann in das Cmdlet **Sort-Object** eingefügt werden, um die Ausgabe basierend auf der Eigenschaft *Name* zu sortieren. Fügen Sie das Objekt anschließend in das Cmdlet **Format-Table** ein und wählen Sie die Eigenschaften *Name, Path* und *Description* aus.

Tipp Die Eigenschaften, die mit dem Cmdlet **Format-Table** in Spalten ausgegeben werden, werden in der Reihenfolge angezeigt, in der sie ausgewählt wurden.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Sie können darüber hinaus mit dem Parameter **-autosize** den Abstand zwischen den Tabellenspalten verkleinern. Das vollständige Skript *ListShares.ps1* verwendet folgende Anweisungen:

ListShares.ps1

```
Get-WmiObject -Class win32_share -ComputerName localhost |
Sort-Object name |
Format-Table name, path, description -AutoSize
```

Das Skript *ListShares.ps1* gibt folgende Liste aus:

```
name      path                          description
----      ----                          -----------
ADMIN$    C:\Windows                    Remoteverwaltung
C$        C:\                           Standardfreigabe
CCMLogs$  C:\Windows\system32\ccm\logs
CCMSetup$ C:\Windows\system32\ccmsetup
IPC$                                    Remote-IPC
Musik     C:\Musik                      keine
VPCache$  C:\Windows\system32\VPCache
WMILogs$  C:\Windows\system32\wbem\logs
```

Wenn Sie detailliertere Informationen zu den Freigaben benötigen, führen Sie das Skript *ListShares-Detailed.ps1* aus. Dieses Skript ermöglicht den Zugriff auf die in Abbildung 5.1 dargestellten Informationen.

Abbildung 5.1 Das Dialogfeld *Erweiterte Freigabe*

Geben Sie im Skript *ListSharesDetailed.ps1* die Variable *$class* an, um den Namen der abzufragenden WMI-Klasse zu speichern. Die erforderliche WMI-Klasse ist *Win32_Share*. Geben Sie außerdem den Namen des Computers an, auf dem die Abfrage ausgeführt werden soll. In diesem Beispiel lautet der Computername *localhost*, aber Sie können einen beliebigen Computer im Netzwerk angeben, für den Sie die entsprechenden Berechtigungen besitzen. Wählen Sie die gewünschten Eigenschaften aus der WMI-Klasse *Win32_Share* aus.

Identifizieren der Eigenschaften von WMI-Klassen

Eine der schwierigeren Aufgaben bei der Arbeit mit WMI-Klassen ist das Identifizieren der verfügbaren Klasseneigenschaften. Eine einfache Methode ist die Verwendung des Cmdlets **Get-Member**. Geben Sie im Cmdlet **Get-WmiObject** den Namen der WMI-Klasse an und fügen Sie das Ergebnis in das Cmdlet **Get-Member** ein. Es werden alle Methoden und Eigenschaften angezeigt, die für die WMI-Klasse *Win32_Share* definiert sind. Die beiden Befehle und die Ausgabe lauten:

```
PS C:\> Get-WmiObject win32_share | get-member

   TypeName: System.Management.ManagementObject#root\cimv2\Win32_Share

Name                MemberType   Definition
----                ----------   ----------
GetAccessMask       Method       System.Management.ManagementBaseObject
SetShareInfo        Method       System.Management.ManagementBaseObject
AccessMask          Property     System.UInt32 AccessMask {get;set;}
AllowMaximum        Property     System.Boolean AllowMaximum {get;set;}
Caption             Property     System.String Caption {get;set;}
Description         Property     System.String Description {get;set;}
InstallDate         Property     System.String InstallDate {get;set;}
MaximumAllowed      Property     System.UInt32 MaximumAllowed {get;set;}
Name                Property     System.String Name {get;set;}
Path                Property     System.String Path {get;set;}
Status              Property     System.String Status {get;set;}
Type                Property     System.UInt32 Type {get;set;}
__CLASS             Property     System.String __CLASS {get;set;}
__DERIVATION        Property     System.String[] __DERIVATION {get;set;}
__DYNASTY           Property     System.String __DYNASTY {get;set;}
__GENUS             Property     System.Int32 __GENUS {get;set;}
__NAMESPACE         Property     System.String __NAMESPACE {get;set;}
__PATH              Property     System.String __PATH {get;set;}
__PROPERTY_COUNT    Property     System.Int32    __PROPERTY_COUNT    {get;set;}
__RELPATH           Property     System.String __RELPATH {get;set;}
__SERVER            Property     System.String __SERVER {get;set;}
__SUPERCLASS        Property     System.String __SUPERCLASS {get;set;}
PSStatus            PropertySet  PSStatus {Status, Type, Name}
ConvertFromDateTime ScriptMethod System.Object ConvertFromDateTime();
ConvertToDateTime   ScriptMethod System.Object ConvertToDateTime();
Delete              ScriptMethod System.Object Delete();
GetType             ScriptMethod System.Object GetType();
Put                 ScriptMethod System.Object Put();
```

Sie können diese Informationen auch mit dem Programm Windows Management Instrumentation Tester (*Wbemtest.exe*) abrufen, das in jeder Windows-Version mit WMI-Unterstützung verfügbar ist. In Abbildung 5.2 sind die von Wbemtest.exe angezeigten Eigenschaften und Methoden der WMI-Klassen dargestellt.

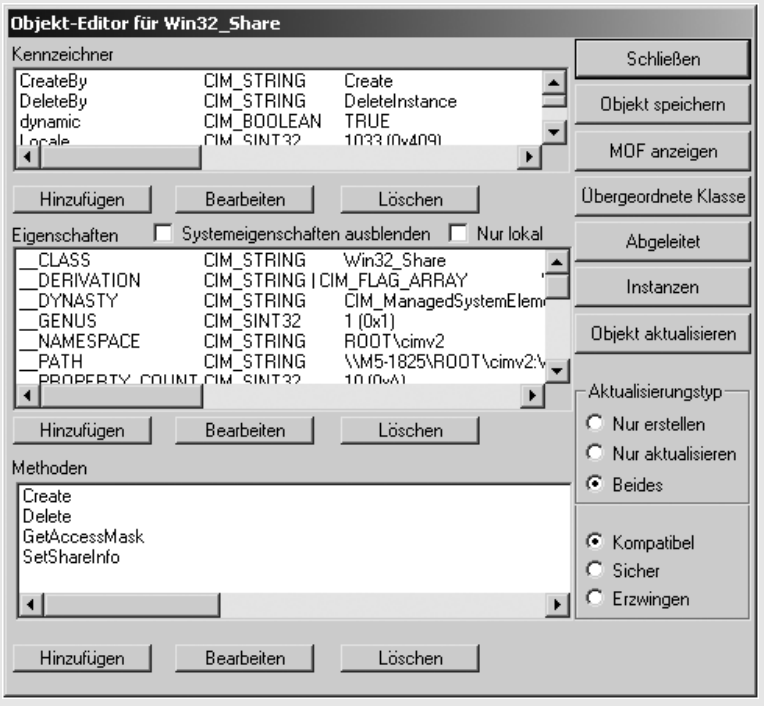


Abbildung 5.2 Das Dialogfeld *Objekt-Editor* der Windows Management Instrumentation für die Klasse *Win32_Share*

Die Eigenschaften, die Sie im Skript *ListSharesDetailed.ps1* auswählen, werden im Array *$aryProperty* gespeichert. Alle Eigenschaftsnamen stehen in Anführungszeichen und sind durch Kommas getrennt. Da zahlreiche Eigenschaften angezeigt werden, sollten Sie ein Gravis-Zeichen eingeben, um die Arraydefinition in der nächsten Zeile fortzusetzen.

Fragen Sie mit dem Cmdlet **Get-WmiObject** den in der Variablen *$computer* angegebenen Computer ab und weisen Sie das zurückgegebene Objekt der Variablen *$objWMI* zu. Das Objekt umfasst möglicherweise Informationen zu mehreren Freigaben. Verwenden Sie eine *foreach*-Anweisung und geben Sie mit dem Cmdlet **Write-Host** eine Kopfzeile mit dem Namen der jeweiligen Freigabe aus.

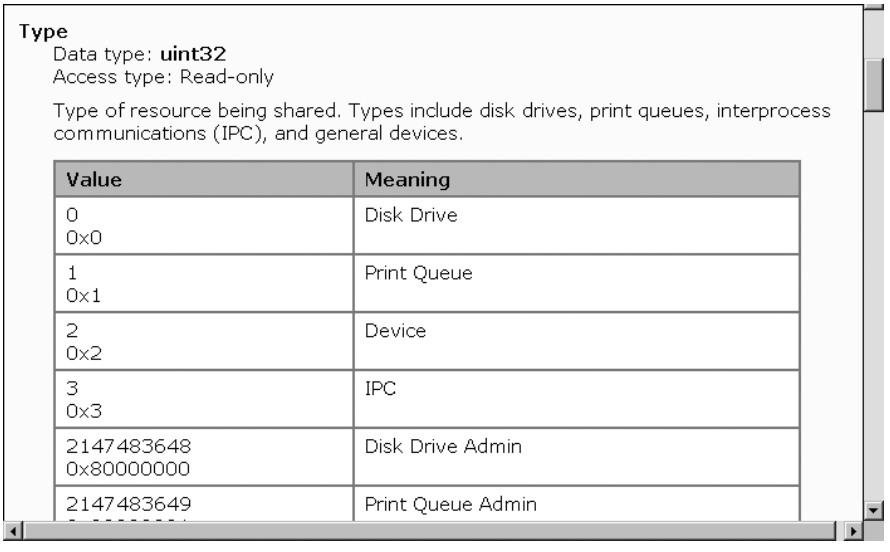
Verwenden Sie die *foreach*-Anweisung erneut, um auf die Eigenschaften der Freigabe zuzugreifen. Rufen Sie die Werte der Eigenschaften ab und geben Sie diese aus. Mit einer *if*-Anweisung können Sie dabei überprüfen, ob die jeweilige Eigenschaft Informationen enthält. Wenn die Eigenschaft leer ist, geben Sie diese nicht aus. Das Skript *ListSharesDetailed.ps1* hat folgenden Aufbau:

ListSharesDetailed.ps1

```
$class = "Win32_Share"
$computer = "localhost"
$aryProperty ="Type", "Name", "AllowMaximum", "Caption", `
  "Description", "MaximumAllowed", "Path"
$objWMI = Get-WmiObject -Class $class -computername $computer

foreach($share in $objWMI)
{
```

```
 Write-Host `
 "
 `nEigenschaften der Freigabe: $($share.name)
------------------------
 "


  foreach($property in $aryProperty)
   {
    if($share.$property -notlike "")
     {
      Write-Host $property : $share.$property
    }
   }
}
```

Das Skript *ListSharesDetailed.ps1* zeigt detaillierte Informationen zu den Freigaben an. Wie der folgende Code veranschaulicht, ist der Freigabetyp in der Ausgabe jedoch als Zahlenwert angegeben. Diese Werte sind zwar im Windows Software Development Kit (SDK) dokumentiert, es ist aber unpraktisch diese ständig nachzuschlagen. In Abbildung 5.3 sind die Freigabewerte aus dem Windows SDK dargestellt.



Type

Data type: **uint32**
Access type: Read-only

Type of resource being shared. Types include disk drives, print queues, interprocess communications (IPC), and general devices.

Value	Meaning
0 0x0	Disk Drive
1 0x1	Print Queue
2 0x2	Device
3 0x3	IPC
2147483648 0x80000000	Disk Drive Admin
2147483649	Print Queue Admin

Abbildung 5.3 Die Werte für Freigaben im Windows Software Development Kit

Folgendes Listing zeigt eine Beispielsausgabe des Skripts *ListSharesDetailed.ps1*:

```
Eigenschaften der Freigabe: ADMIN$
------------------------

Type : 2147483648
Name : ADMIN$
AllowMaximum : True
Caption : Remoteverwaltung
Description : Remoteverwaltung
Path : C:\Windows
```

```
Eigenschaften der Freigabe: C$
------------------------

Type : 2147483648
Name : C$
AllowMaximum : True
Caption : Standardfreigabe
Description : Standardfreigabe
Path : C:\
```

Um den Wert der Freigabe in eine Beschreibung umzuwandeln, erstellen Sie im Skript *ListSharesDetailedTranslateShareType.ps1* eine Funktion, die auf den Informationen im Windows SDK basiert.

Das Skript *ListSharesDetailedTranslateShareType.ps1* beginnt mit einer solchen Funktion. Deklarieren Sie die Funktion und geben Sie dieser den Namen *funlookup*. Wenn Sie das Skript aufrufen, müssen Sie einen Wert an diese Funktion übergeben. Der Name des Übergabeparameters lautet *$intIN*.

Überprüfen Sie den an die Funktion übergebenen Wert mit einer *switch*-Anweisung. Wenn der Wert *0* ist, weisen Sie der globalen Variablen *$strRTN* die Zeichenfolge "Laufwerk" zu. Führen Sie diesen Schritt für alle gültigen Freigabetypen aus.

Hinweis Wenn Sie eine Variable in einer Funktion verwenden, wird der Wert normalerweise in der Funktion gespeichert. Eine Variable mit dem gleichen Namen, die sich außerhalb der Funktion befindet, kann zu Verwechslungen führen. Da Funktionen keine Werte zurückgeben, müssen Sie eine Variable erstellen, die den Wert enthält. Um die gleiche Variable innerhalb und außerhalb einer Funktion zu verwenden, ist eine globale Variable erforderlich. Die Syntax für globale Variablen ist:

```
$global:strRTN="Laufwerk"
```

Deklarieren Sie die Variable *$strRTN* als eine globale Variable und weisen Sie dieser den Wert *$null* zu, um sicherzustellen, dass die Variable keine veralteten Daten enthält, die möglicherweise zu unerwarteten Ergebnissen führen. Befehlssyntax:

```
$global:strRTN = $null
```

Der restliche Code ist bis auf eine zusätzliche *if*-Anweisung zur Auswertung der Eigenschaft *Type* mit dem Code im Skript *ListSharesDetailed.ps1* identisch. Wenn die Eigenschaft *Type* gefunden wird, überprüfen Sie den Wert mit der Funktion *funlookup* und geben Sie den umgewandelten Wert anschließend aus.

```
if($property -eq "type")
     {
      funLookup($share.$property)
      Write-Host $property "Name:" $strRTN
     }
```

Nachdem das Skript die Funktion *funlookup* aufgerufen hat, legen Sie die Variable *$strRTN* wieder auf *$null* fest und setzen das Durchlaufen der Freigaben und der Eigenschaften fort. Das vollständige Skript *ListSharesDetailedTranslateShareType.ps1* ist wie folgt implementiert:

ListSharesDetailedTranslateShareType.ps1

```
Function funLookUp ($intIN)
{
 switch ($intIN)
 {
   0  { $global:strRTN="Laufwerk" }
   1  { $global:strRTN="Druckwarteschlange" }
   2  { $global:strRTN="Gerät" }
   3  { $global:strRTN="IPC " }
   2147483648 { $global:strRTN="Laufwerksverwaltung" }
   2147483649 { $global:strRTN="Druckwarteschlangenverwaltung"}
   2147483650 { $global:strRTN="Geräteverwaltung" }
   2147483651 { $global:strRTN="IPC-Verwaltung" }
 }
}


$global:strRTN = $null
$class = "Win32_Share"
$computer = "localhost"
$aryProperty ="Type", "Name", "AllowMaximum", "Caption", `
  "Description", "MaximumAllowed", "Path"
$objWMI = Get-WmiObject -Class $class -computername $computer

foreach($share in $objWMI)
{
 Write-Host `
 "
 `nEigenschaften der Freigabe: $($share.name)
-------------------------
 "

  foreach($property in $aryProperty)
   {
    if($share.$property -notlike "")
     {
      Write-Host $property : $share.$property
    }
   if($property -eq "type")
     {
      funLookup($share.$property)
      Write-Host $property "Name:" $strRTN
     }
   }
$Global:strRTN=$null
}
```

Dokumentieren von Benutzerfreigaben

Benutzerdefinierte Freigaben werden nicht als spezieller Freigabetyp angezeigt. Wenn eine Freigabe keine administrative Freigabe ist und nicht von einem IT-Administrator erstellt wurde, muss es sich um eine Benutzerfreigabe handeln. Auch wenn eine Freigabe benutzerdefiniert ist, hat der Benutzer möglicherweise keine Kenntnis von der Freigabe.

Bewährte Vorgehensweise Geben Sie beim Erstellen einer Freigabe den Parameter *Description* an, damit die von IT erstellten Freigaben von Benutzerfreigaben unterschieden werden können.

Nicht administrative Freigaben werden nicht automatisch vom Betriebssystem erstellt und umfassen die von der IT-Abteilung oder den Benutzern erstellten Freigaben. Das Skript *ListNonAdminShares.ps1* gibt die Standardeigenschaften aller Freigaben mit einem Freigabetyp kleiner als 10 aus.

ListNonAdminShares.ps1

```
Get-WmiObject win32_share -Filter "type < '10'"
```

Eine Beispielausgabe des Skripts *ListNonAdminShares.ps1* ist im folgenden Abschnitt dargestellt. Für die Freigaben wird keine Beschreibung angezeigt. In den Informationen ist lediglich angegeben, dass es sich um Datenträgerfreigaben handelt. Der Verwendungszweck der Freigaben auf dem Computer ist nicht offensichtlich. Dieses Problem kann vermieden werden, wenn der Benutzer, der die Freigabe erstellt, eine Beschreibung eingibt. Die Freigaben ohne Beschreibung:

```
Name      Path                          Description
----      ----                          -----------
CCMLogs$  C:\Windows\system32\ccm\logs
CCMSetup$ C:\Windows\system32\ccmsetup
VPCache$  C:\Windows\system32\VPCache
WMILogs$  C:\Windows\system32\wbem\logs
```

In Abbildung 5.4 ist das Textfeld **Kommentar** im Dialogfeld **Erweiterte Freigabe** dargestellt. In diesem Feld können Sie die Freigabebeschreibung eingeben.

Die von der IT-Abteilung und vom Benutzer erstellten Freigaben sind nicht einfach zu unterscheiden. Die automatisch erstellten administrativen Freigaben können jedoch sofort erkannt werden. Beispielsweise können Sie wie im Skript *WriteUserSharesToExcel.ps1* demonstriert das Microsoft Excel-Objektmodell verwenden und die Freigabeinformationen in einem Excel-Arbeitsblatt erfassen.

Erstellen Sie als Erstes die Variable *$strPath*, um den Pfad und den Namen des Arbeitsblatts anzugeben, und anschließend eine Instanz des COM-Objekts *Excel.Application*. Dieses Objekt automatisiert Excel. Erstellen Sie das Objekt mit dem Cmdlet **New-Object** und dem Parameter **-comobject**. Das neue *Excel.Application*-Objekt wird in der Variablen *$objExcel* gespeichert.

Abbildung 5.4 Im Dialogfeld *Erweiterte Freigabe* können Kommentare eingegeben werden

Legen Sie die Eigenschaft *Visible* auf -1 (*true*) fest. Die Eigenschaft entspricht der automatischen Variablen *$true*. Fügen Sie anschließend mit der *Add*-Methode eine Arbeitsmappe hinzu:

```
$WorkBook=$objExcel.Workbooks.Add()
```

Sie müssen nun auf ein bestimmtes Arbeitsblatt zugreifen. Verwenden Sie hierzu die Methode *Item*:

```
$sheet=$workbook.worksheets.item(1)
```

WriteUserSharesToExcel.ps1

```
$strPath="c:\FSO\Arbeitsblatt.xls"
$objExcel=New-Object -ComObject Excel.Application
$objExcel.Visible=-1
$WorkBook=$objExcel.Workbooks.Add()
$sheet=$workbook.worksheets.item(1)

$x=2

$strComputer = "."
$objWMIService = Get-WmiObject win32_Share

$sheet.Cells.item(1,1)=("Name der Freigabe")
$sheet.Cells.item(1,2)=("Beschreibung")
$sheet.Cells.item(1,3)=("Art der Freigabe")


ForEach ($objShare in $objWMIService)
{
    $sheet.Cells.item($x, 1)=($objShare.Name)
    $sheet.Cells.item($x, 2)=($objShare.Description)
    $sheet.Cells.item($x, 3)=($objShare.Type)

    If($objShare.type -ne 0)
    {
    $sheet.Cells.item($x,3).font.colorIndex=3
    $sheet.Cells.item($x,3).font.bold=$true
   }
    $x++
}
 $range = $sheet.usedRange
 $range.EntireColumn.AutoFit()

IF(Test-Path $strPath)
  {
   Remove-Item $strPath
   $objExcel.ActiveWorkbook.SaveAs($strPath)
  }
ELSE
  {
   $objExcel.ActiveWorkbook.SaveAs($strPath)
  }
```

In Abbildung 5.5 ist ein Beispiel eines vollständigen Excel-Arbeitsblatts dargestellt.

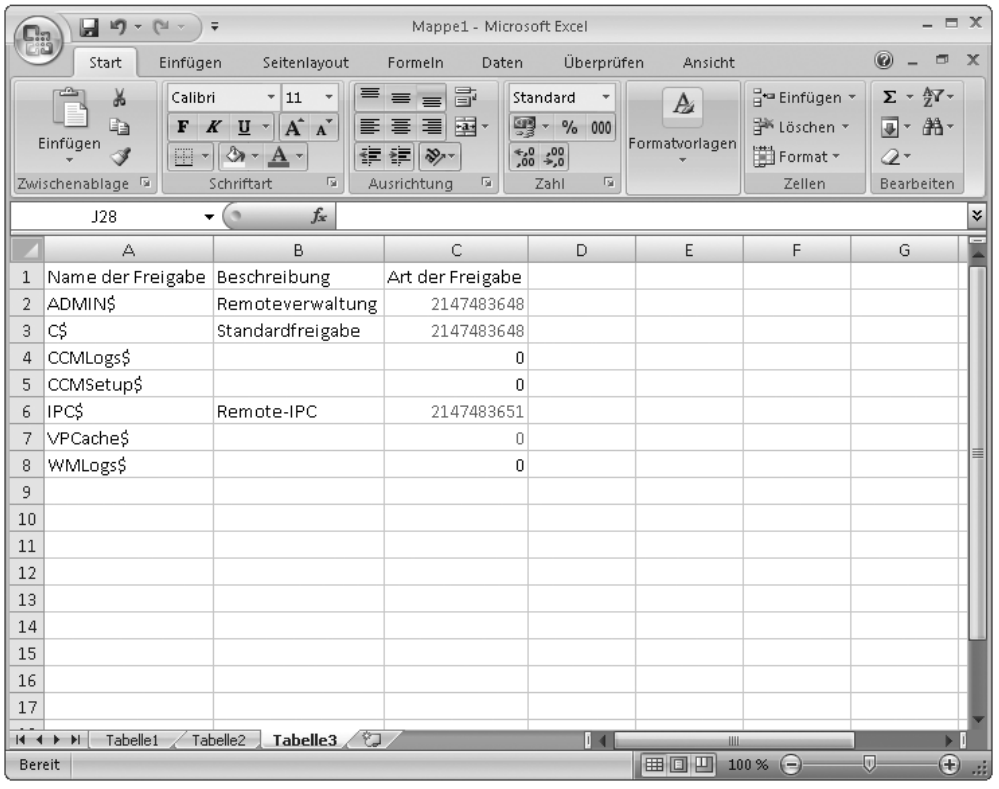


Abbildung 5.5 Nachdem die Freigabeinformationen in einem Excel-Arbeitsblatt gespeichert wurden, können Sie die generierten Daten analysieren

Bewährte Vorgehensweise Zahlreiche Softwarepakete erstellen Freigaben auf einem Computer. Bei diesen Softwarepaketen handelt es sich nicht um Malware. Kommerzielle Software erstellt Freigaben aus mehreren Gründen. Eine Freigabe kann beispielsweise von einem unbedeutenden Feature der Software erstellt werden. Deshalb ist das Überwachen von Freigaben ausgesprochen wichtig.

Erfassen von Freigaben in Textdateien

Obwohl ein Excel-Arbeitsblatt für die Analyse zahlreicher Daten nützlich ist, ist manchmal nur eine einfache ASCII-Textdatei erforderlich. Mit dem Skript *WriteSharesToFile.ps1* können Sie eine Textdatei erstellen.

Deklarieren Sie als Erstes die Variable *$class*, um den Namen der abgefragten WMI-Klasse zu speichern. In diesem Skript fragen Sie die Klasse *Win32_Share* ab. Speichern Sie anschließend eine Zeichenfolge, die den Pfad zur Textdatei angibt, in der Variablen *$filePath*. Die Pfadangabe muss den Namen der Textdatei umfassen.

Fragen Sie WMI unter Verwendung des Cmdlets **Get-WmiObject** ab. Da der Standardparameter für das Cmdlet **-class** ist, ist die Parameterangabe optional. Um das Skript übersichtlicher zu gestalten, sollten Sie diesen Parameter jedoch angeben. Fügen Sie das Ergebnis des Cmdlets **Get-WmiObject** in das Cmdlet **Format-Table** ein, um die WMI-Klasse *Win32_Share* abzufragen. Das Cmdlet **Format-**

Table entfernt überflüssige Headerinformationen und gibt den Namen des WMI-Objekts aus, das von der Abfrage zurückgegeben wird. Das Objekt wird anschließend an das Cmdlet **Out-File** übergeben. Dieses Cmdlet erfordert mindestens einen Dateipfad. Stellen Sie mit dem Parameter **-encoding** sicher, dass die Ausgabedatei eine ASCII-Datei ist. Das vollständige Skript *WriteSharesToFile.ps1* umfasst folgende Anweisungen:

WriteSharesToFile.ps1

```
$class = "win32_share"
$filePath = "c:\FSO\Shares.txt"
Get-WmiObject -class $class |
Format-Table -property name -hidetableheader |
Out-File -FilePath $filePath -encoding ASCII
```

In Abbildung 5.6 ist eine mit dem Skript *WriteSharesToFile.ps1* erstellte Datei *Shares.txt* dargestellt.

Abbildung 5.6 Die Ergebnisse des Skripts *WriteSharesToFile.ps1*

Dokumentieren administrativer Freigaben

Administrative Freigaben auf Windows Vista- und Windows Server 2008-Plattformen werden automatisch vom Betriebssystem erstellt. Diese Freigaben unterstützen zahlreiche Funktionen verschiedener Anwendungen.

- Da diese administrativen Freigaben in Hochsicherheitsumgebungen ein nicht akzeptables Risiko darstellen, müssen diese gelöscht werden. Diese Aufgabe wird normalerweise von IT-Beratern, z.B. Microsoft Consulting Services, übernommen. Das Verfahren ist umfassend dokumentiert und getestet, um die Kompatibilität mit Branchenanwendungen sicherzustellen.
- In Umgebungen mit geringeren Sicherheitsanforderungen werden diese Freigaben manchmal von Benutzern gelöscht, die ihre Computer absichern möchten. Dies kann zur Folge haben, dass der Netzwerkadministrator viele frustierende Stunden mit der Problembehandlung verbringen muss, um die Ursache unerwarteten Verhaltens zu ermitteln.

Um eine Liste der administrativen Freigaben auf einem Computer abzurufen, führen Sie das Skript *ListAdminShares.ps1* aus. Dieses Skript fragt die WMI-Klasse *Win32_Share* mit dem Cmdlet **Get-WmiObject** ab. Verwenden Sie einen Filter, um die Freigaben mit einem Typwert von größer als 10 abzurufen. Auf diese Art werden nur die automatisch erstellten administrativen Freigaben angezeigt. Das Skript *ListAdminShares.ps1* hat folgenden Inhalt:

ListAdminShares.ps1

```
Get-WmiObject win32_share -Filter "type > '10'"
```

Eine Beispielausgabe des Skripts *ListAdminShares.ps1* ist im folgenden Abschnitt dargestellt. In der Standardansicht werden der Name der Freigabe, der Freigabepfad und eine Beschreibung angezeigt. Da für alle administrativen Freigaben eine Beschreibung angegeben ist, sind diese Freigaben einfacher zu verwalten.

```
Name   Path       Description
----   ----       -----------
ADMIN$ C:\Windows Remoteverwaltung
C$     C:\        Standardfreigabe
IPC$              Remote-IPC
```

Erfassen von Freigabeinformationen in einer Microsoft Access-Datenbank

Sie können die Konfigurationsinformationen auch in einer Access-Datenbank speichern. In diesem Abschnitt fügen Sie weitere Daten zur Konfigurationsdatenbank hinzu. Indem Sie die Informationen mit einem Skript protokollieren, können Sie die an einer Freigabe vorgenommenen Änderungen nachverfolgen, Berichte erstellen und die Konfigurationseinstellungen überprüfen.

Beginnen Sie das Skript *WriteSharesToAccess.ps1*, indem Sie mehrere Variablen deklarieren, in denen der Computername und die Domäne, in der sich der Computer befindet, gespeichert werden. Erstellen Sie zuerst eine Instanz des *wshNetwork*-Objekts. Verwenden Sie hierzu das Cmdlet **New-Object** mit dem Parameter **-comobject** und geben Sie die Programm-ID *WScript.Network* an. Die beiden Codezeilen lauten:

```
$StrComputer = (New-Object -ComObject WScript.Network).computername
$StrDomain = (New-Object -ComObject WScript.Network).userDomain
```

Deklarieren Sie die Variable *$strWmiQuery*, um die WMI-Abfrage zu speichern, und wählen Sie alle Eigenschaften der WMI-Klasse *Win32_Share* aus. Führen Sie die Abfrage mit dem Cmdlet **Get-WmiObject** und dem Parameter **-query** aus. Das resultierende Objekt wird der Variablen *$obService* zugewiesen. Die beiden Codezeilen lauten:

```
$strWMIQuery = "Select * from win32_Share"
$objService = get-wmiobject -query $strWMIQuery
```

Deklarieren Sie anschließend mehrere Variablen, um die Datenbank zu öffnen. Die Variable *$adOpenStatic* hat den Wert 3 und gibt an, dass ein statischer Datensatz geöffnet wird. Die Variable *$adLockOptimistic* hat ebenfalls den Wert 3 und gibt an, dass das vollständige Sperren (Optimistic Locking) verwendet werden soll. Der Pfad zur Datenbank ist in der Variablen *$strDB* gespeichert. Die Variable *$strTable* enthält den Namen der Tabelle, in der Sie die Daten speichern möchten. Die letzte Variable in diesem Codeabschnitt ist *$strAccessQuery*, mit der die Zeichenfolge "*Select * from $strTable*" angegeben wird. Führen Sie diese Abfrage aus, um auf die Tabelle zuzugreifen. Das Ergebnis der Abfrage ist in diesem Fall jedoch nicht von Interesse. Codeabschnitt:

```
$adOpenStatic = 3
$adLockOptimistic = 3
$strDB = "c:\FSO\ConfigurationMaintenance.mdb"
$strTable = "Shares"
$strAccessQuery = "Select * from $strTable"
```

Sie müssen nun ein *Connection*-Objekt und ein *RecordSet*-Objekt erstellen. Erstellen Sie diese beiden COM-Objekte mit dem Cmdlet **New-Object**. Der Code zum Erstellen der beiden Objekte ist:

```
$objConnection = new-Object -ComObject ADODB.Connection
$objRecordSet = new-object -ComObject ADODB.Recordset
```

Nachdem Sie das *Connection*-Objekt und das *RecordSet*-Objekt erstellt haben, können Sie die Datenbankverbindung öffnen. Sie müssen den Namen des Anbieters und die Datenquelle angeben. Die Datenquelle ist die Datenbank, mit der Sie arbeiten möchten. Die Datenquelle umfasst den Datenbanknamen und den Datenbankpfad. Der Anbieter ist datenbankspezifisch. Da Sie mit einer Access-Datenbank arbeiten, müssen Sie den Microsoft.Jet.OLEDB.4.0-Anbieter angeben. Die Codezeile zum Öffnen der Datenbankverbindung lautet:

```
$objConnection.Open("Provider = Microsoft.Jet.OLEDB.4.0; Data Source= $strDB")
```

Nachdem die Datenbankverbindung hergestellt wurde, können Sie den Datensatz öffnen. Um den Datensatz zu öffnen, geben Sie vier Parameter an: Die Abfrage, die Verbindung, die Methode zum Öffnen der Datenbank und die Methode zum Handhaben gleichzeitiger Verbindungen. Codezeile:

```
$objRecordSet.Open($strAccessQuery, `
  $objConnection, $adOpenStatic, $adLockOptimistic)
```

Nachdem die Verbindung und das *RecordSet*-Objekt geöffnet wurden, geben Sie eine Meldung aus, dass das Skript ausgeführt wird. Verwenden Sie hierzu das Cmdlet **Write-Host** und legen Sie Gelb als Schriftfarbe fest. Beispiel:

```
write-host -foreGroundColor yellow "Abfragen der Freigabeinformationen ..."
```

Da die WMI-Abfrage die Informationen zu mehreren Freigaben zurückgibt, müssen Sie die Informationen für jede Freigabe einzeln durchlaufen. Verwenden Sie hierzu eine *foreach*-Anweisung. Die Freigabeinformationen sind in der Variablen *$objService* verfügbar. Der Enumerator ist die Variable *$service*, die beim Durchlaufen der Informationen auf jeweils eine einzelne Freigabe verweist. Rufen Sie mit der Variablen *$service* die Eigenschaften der Freigaben ab:

```
foreach ($service in $objService)
```

Um die Informationen zusammenzustellen, die in der Datenbank gespeichert werden sollen, fragen Sie mit der Variablen *$service* die gewünschten Werte ab. Geben Sie den Variablen Namen, die den Eigenschaftsnamen ähnlich sind, damit Sie die Eigenschaften einfach auseinanderhalten können. Codebeispiel:

```
$blnAllowMaximum = $service.AllowMaximum
$strCaption = $service.Caption
$strDescription = $service.Description
$intMaximumAllowed = $service.MaximumAllowed
$strName = $service.Name
$strPath = $service.Path
$intType = $service.Type
```

Nachdem Sie die Informationen mit Hilfe von WMI abgerufen haben, fügen Sie mit der *addNew()*-Methode des *RecordSet*-Objekts einen neuen Datensatz zur Datenbank hinzu. Beispiel:

```
$objRecordSet.AddNew()
```

Erfassen Sie mit dem Cmdlet **Get-Date** das Datum und die Uhrzeit der Datenabfrage. Alle anderen Daten werden aus den einzelnen Variablen bezogen. Nachdem die Daten den entsprechenden Feldern in der Tabelle zugeordnet wurden, rufen Sie die *Update*-Methode des *RecordSet*-Objekts auf. Codeabschnitt:

```
$objRecordSet.Fields.item("TimeStamp") = Get-Date
$objRecordSet.Fields.item("strComputer") = $strComputer
$objRecordSet.Fields.item("strDomain") = $strDomain
$objRecordSet.Fields.item("blnAllowMaximum") = $blnAllowMaximum
```

```
$objRecordSet.Fields.item("strCaption") = $strCaption
$objRecordSet.Fields.item("strDescription") = $strDescription
$objRecordSet.Fields.item("intMaximumAllowed") = $intMaximumAllowed
$objRecordSet.Fields.item("strName") = $strName
$objRecordSet.Fields.item("strPath") = $strPath
$objRecordSet.Fields.item("intType") = $intType
$objRecordSet.Update()
```

Jede mit der Methode *Update()* aktualisierte Eigenschaft entspricht einem Feld in der Access-Datenbank. In Abbildung 5.7 ist die entsprechende Datenbanktabelle dargestellt.

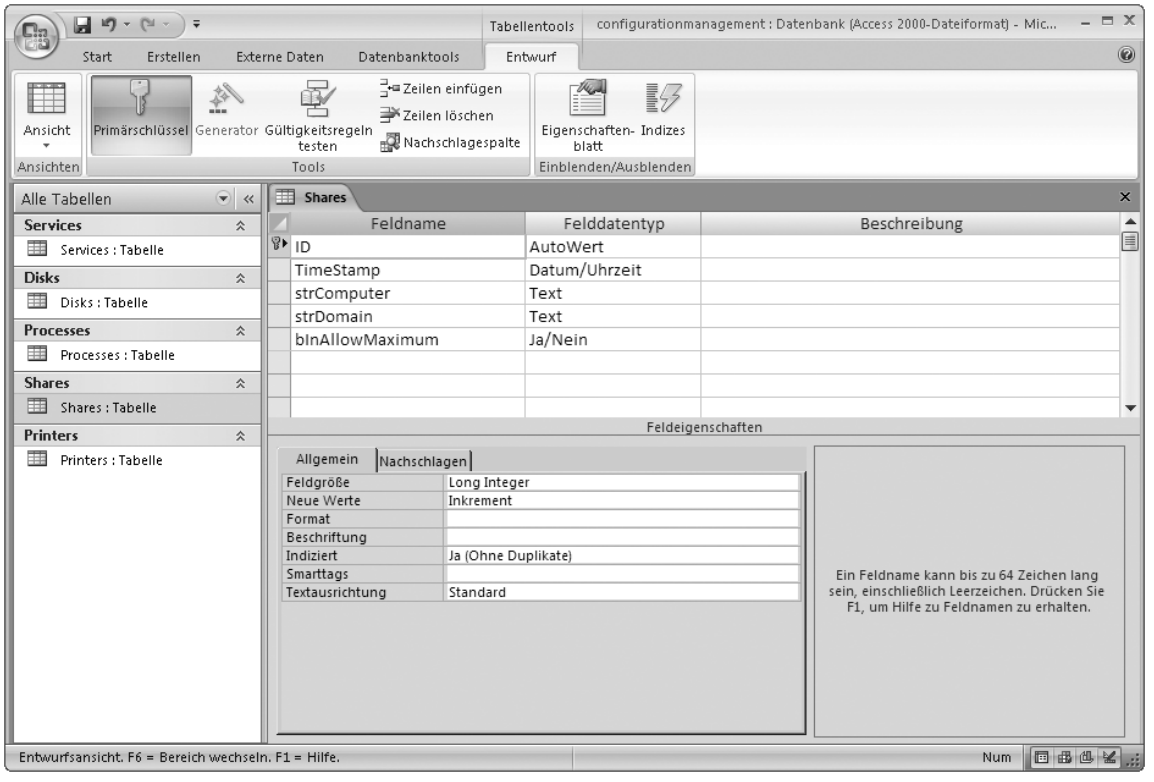


Abbildung 5.7 Die Tabelle mit den Freigaben

Geben Sie mit dem Cmdlet **Write-Host** einen Statusindikator aus. Geben Sie für jedes Element, das in die Datenbank geschrieben wird, einen Schrägstrich und einen Backslash (/\) ein und verwenden Sie den Parameter **-nonewline**, um die Schrägstriche in einer fortlaufenden Zeile auszugeben. Codezeile:

```
write-host -foregroundColor yellow "/\" -noNewLine
```

Nachdem alle Daten in die Datenbank geschrieben wurden, schließen Sie die Verbindung und den Datensatz. Codebeispiel:

```
$objRecordSet.Close()
$objConnection.Close()
```

Das vollständige Skript *WriteSharesToAccess.ps1* hat folgenden Aufbau:

WriteSharesToAccess.ps1

```
$StrComputer = (New-Object -ComObject WScript.Network).computername
$StrDomain = (New-Object -ComObject WScript.Network).userDomain
$strWMIQuery = "Select * from win32_Share"
$objService = get-wmiobject -query $strWMIQuery

$adOpenStatic = 3
$adLockOptimistic = 3
$strDB = "c:\FSO\ConfigurationMaintenance.mdb"
$strTable = "Shares"
$strAccessQuery = "Select * from $strTable"

$objConnection = new-Object -ComObject ADODB.Connection
$objRecordSet = new-object -ComObject ADODB.Recordset
$objConnection.Open("Provider = Microsoft.Jet.OLEDB.4.0; `
  Data Source= $strDB")
$objRecordSet.Open($strAccessQuery, `
  $objConnection, $adOpenStatic, $adLockOptimistic)

write-host -foreGroundColor yellow "Abfragen der Freigabeinformationen ..."

foreach ($service in $objService)
{
   $blnAllowMaximum = $service.AllowMaximum
   $strCaption = $service.Caption
   $strDescription = $service.Description
   $intMaximumAllowed = $service.MaximumAllowed
   $strName = $service.Name
   $strPath = $service.Path
   $intType = $service.Type

   $objRecordSet.AddNew()
   $objRecordSet.Fields.item("TimeStamp") = Get-Date
   $objRecordSet.Fields.item("strComputer") = $strComputer
   $objRecordSet.Fields.item("strDomain") = $strDomain
   $objRecordSet.Fields.item("blnAllowMaximum") = $blnAllowMaximum
   $objRecordSet.Fields.item("strCaption") = $strCaption
   $objRecordSet.Fields.item("strDescription") = $strDescription
   $objRecordSet.Fields.item("intMaximumAllowed") = $intMaximumAllowed
   $objRecordSet.Fields.item("strName") = $strName
   $objRecordSet.Fields.item("strPath") = $strPath
   $objRecordSet.Fields.item("intType") = $intType
   $objRecordSet.Update()
   write-host -foregroundColor yellow "/\" -noNewLine
}

$objRecordSet.Close()
$objConnection.Close()
```

Überwachen von Freigaben

Freigaben, insbesondere Benutzerfreigaben, können ein Sicherheitsrisiko darstellen. Deshalb müssen Netzwerkadministratoren die Freigaben auf Arbeitsstationen und Servern überwachen, um sicherzustellen, dass alle vorhandenen Freigaben autorisiert und ordnungsgemäß konfiguriert sind.

Sie können die Freigaben überprüfen, indem Sie in der Access-Datenbank einen Bericht erstellen. Dieser Bericht ist in Abbildung 5.8 dargestellt.

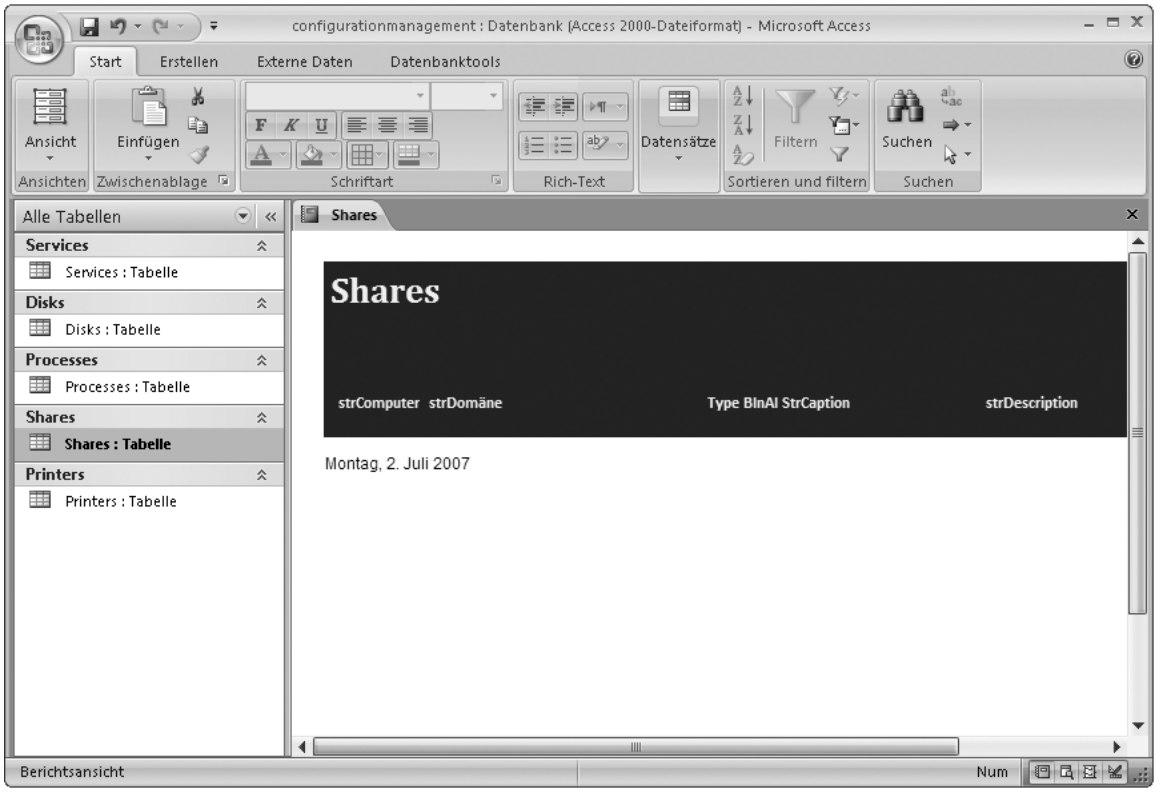


Abbildung 5.8 Ein mit Access erstellter Freigabebericht

Sie können die Freigaben auf einem Computer auch mittels einer Textdatei überwachen. Die Textdatei kann beispielsweise mit dem Skript *WriteSharesToFile.ps1* erstellt werden. In Abbildung 5.6 ist eine Beispieltextdatei dargestellt. Die Textdatei kann nun als Grundlage für Vergleiche mit dem aktuellen Computerstatus dienen.

Mit den in Windows PowerShell integrierten Textverarbeitungsfunktionen können Sie die aktuellen und gewünschten Freigaben vergleichen. Die Überwachung umfasst die beiden folgenden Aufgaben: Sie müssen sicherstellen, dass die vorhandenen Freigaben autorisiert und keine nicht autorisierten Freigaben vorhanden sind.

Mit dem Skript *CompareShares.ps1* können Sie die vorhandenen Freigaben mit den in einer Textdatei aufgeführten Freigaben vergleichen. Das Skript überprüft, dass die erforderlichen Freigaben vorhanden sind, erkennt aber nicht, ob die Freigaben autorisiert sind. Erstellen Sie für das Skript *CompareShares.ps1* die Variable *$strFile*, um den Pfad zur Datei mit den zu überwachenden Freigaben zu speichern. Geben Sie anschließend den Inhalt der Datei mit dem Cmdlet **Get-Content** aus. Übergeben Sie das resultierende Objekt an das Cmdlet **ForEach-Object** und verwenden Sie die Methode *trimend()* aus der .NET Framework-Klasse *System.String* , um sicherzustellen, dass der in der Datei *Shares.txt* angegebene Freigabenname keine ungültigen Zeichen enthält.

Geben Sie den bereinigten Freigabenamen im Cmdlet **Get-WmiObject** für die WMI-Abfrage an. Überprüfen Sie mit dem Cmdlet **ForEach-Object**, ob die in der Datei *Shares.txt* aufgeführte Freigabe noch vorhanden ist. Ist die Freigabe vorhanden, geben Sie die entsprechenden Informationen aus. Geben Sie ansonsten eine Meldung aus, dass die Freigabe nicht mehr verfügbar ist.

Das Skript *CompareShares.ps1* ist wie folgt aufgebaut:

CompareShares.ps1

```
$strFile = "c:\FSO\Shares.txt"
Get-Content  $strFile |
foreach-object { $strShare = $_.trimend()
$strQuery = "Select * from win32_share where name ='$strShare'"
get-wmiobject -query $strQuery |
foreach-object `
{
 if ($_.name )
  { Write-Host $_.name "ist noch vorhanden." }
 ELSE
  { Write-Host -foregroundcolor RED $_.name `
    " ist nicht mehr vorhanden." }
}
}
```

Das Skript *CompareShares.ps1* stellt sicher, dass die erforderlichen Freigaben vorhanden sind. Sie müssen jedoch auch nicht autorisierte Freigaben ermitteln. Diese scheinbar einfache Aufgabe ist jedoch schwierig auszuführen, wenn der Freigabename mit einem Dollarzeichen endet und mit einem regulären Ausdruck nach Übereinstimmungen gesucht wird. Um dieses Problem zu umgehen, verwenden Sie die Methode *substring* aus der .NET Framework-Klasse *System.String*. Die Methode *substring* akzeptiert zwei Parameter: Ein Parameter gibt die Startposition an und der andere Parameter legt die Anzahl der Zeichen fest. Da die Freigabenamen unterschiedlich lang sind, subtrahieren Sie 1 vom Wert der Eigenschaft *Length* und verwenden den gekürzten Freigabenamen.

Löschen Sie im Skript *AuditUnauthorizedShares.ps1* mit dem Cmdlet **Clear-Host** die Ausgabe und rufen Sie dann mit dem Cmdlet **Get-Content** den Inhalt der Textdatei ab, die die Namen der autorisierten Freigaben enthält. Speichern Sie den Inhalt der Datei in der Variablen *$strFile*. Speichern Sie anschließend die WMI-Abfrage, die alle Freigaben auf dem Computer abruft, in der Variablen *$strQuery*. Führen Sie die WMI-Abfrage mit dem Cmdlet **Get-WmiObject** aus und weisen Sie das zurückgegebene *Management*-Objekt der Variablen *$shares* zu.

Verwenden Sie eine *foreach*-Anweisung, um die Freigaben zu überprüfen. Geben Sie in der Variablen *$share* eine Freigabe an, rufen Sie mit *$shareName* den Freigabenamen aus dem *share*-Objekt ab und wandeln Sie den Wert in eine Zeichenfolge um. Rufen Sie anschließend mit der Methode *substring()* aus der .NET Framework-Klasse *System.String* alle Zeichen des Freigabenamens mit Ausnahme des letzten Zeichens ab. Die beiden Codezeilen lauten:

```
$shareName = $($share.name).tostring()
$shareName = $shareName.substring(0,$shareName.length-1)
```

Geben Sie mit dem Cmdlet **Write-Host** eine Statusmeldung aus, dass nach einer bestimmten Freigabe gesucht wird. Da der Freigabename gekürzt wurde, verwenden Sie die Eigenschaft *Name* des *Share*-Objekts. Wenn der Freigabename in der Liste der autorisierten Freigaben gefunden wird, geben Sie die Meldung in Grün aus. Wenn der Freigabename nicht in der Liste der autorisierten Freigaben gefunden wird, geben Sie die Meldung in Rot aus. Das vollständige Skript *AuditUnauthorizedShares.ps1* umfasst folgende Anweisungen:

AuditUnauthorizedShares.ps1

```
Clear-Host
$strFile = Get-Content "c:\FSO\Shares.txt"

$strQuery = "Select * from win32_share"
$shares = get-wmiobject -query $strQuery

foreach ( $share in $shares)
 {
   $shareName = $($share.name).tostring()
   $shareName = $shareName.substring(0,$shareName.length-1)

  Write-Host "Abfragen der Freigabe $($share.Name) ..." -ForegroundColor yellow

  if ( $strFile -match $shareName )
   { Write-Host "`t$($share.name) gefunden." -foregroundcolor Green}
  ELSE
   { Write-Host "`t$($share.Name) nicht gefunden." -foregroundcolor red}
 }
```

Ändern von Freigaben

Sie können drei Einstellungen für Freigaben ändern: Die maximale Anzahl der zugelassenen Benutzer, die Beschreibung der Freigabe und die Sicherheitseinstellungen. Das Ändern der Beschreibung und der Benutzeranzahl ist einfach. Das Ändern der Sicherheitseinstellungen ist jedoch etwas komplizierter.

Erstellen Sie z.B. mit dem Skript *SetShareInfo.ps1* vier Variablen, um die Informationen für das Skript zu speichern. In der Variablen *$shareName* wird der Name der zu ändernden Freigabe angegeben. Da WMI erwartet, dass der Name der Freigabe in einfachen Anführungszeichen eingeschlossen ist, geben Sie diese in den doppelten Anführungszeichen ein. Erstellen Sie anschließend die Variable *$wmiClass*, in der der Name der abzufragenden WMI-Klasse gespeichert wird. Da Sie mit der Klasse *Win32_Share* arbeiten, muss dieser Name der Variablen *$wmiClass* zugewiesen werden.

Weisen Sie außerdem die Werte für die zu ändernden Eigenschaften zu. Die erste Eigenschaft ist *MaximumAllowed*. Dieser Wert legt die Anzahl der Benutzer fest, die gleichzeitig auf die Freigabe zugreifen können.

Die nächste Eigenschaft betrifft die Beschreibung der Freigabe. Diese Eigenschaft entspricht einer Zeichenfolge, in der Sie den Verwendungszweck der Freigabe, die Anwendungen, die die Freigabe verwenden, oder die Abteilung, die die Freigabe benötigt, angeben können. Beachten Sie jedoch, dass die in der Eigenschaft *Description* angegebenen Informationen im Netzwerk sichtbar sind. Die Informationen werden in der **Netzwerkumgebung**, vom Befehl **Get-WmiObject *Win32_Share*** und vom Befehl **net share** angezeigt.

Nachdem Sie die vier Variablen erstellt haben, erstellen Sie mit dem Cmdlet **Get-WmiObject** eine Instanz der Klasse *Win32_Share*. Der Parameter **-filter** gibt den Namen der gewünschten Freigabe an. Der Name der Freigabe ist der Variablen *$shareName* zugewiesen. Nachdem Sie eine Freigabe abgerufen haben, rufen Sie die Methode *setShareInfo* auf, um den Eigenschaften *MaxAllowed* und *Description* Werte zuzuweisen. Verwenden Sie die Methode *setShareInfo*, um die neuen Werte zu übernehmen oder die vorhandenen Werte zu ändern.

Wenn Sie die Methode *setShareInfo* aufrufen, speichern Sie den Rückgabecode in der Variablen *$errRTN*. Sie können dann den Wert der Eigenschaft *ReturnValue* des zurückgegebenen Objekts

auswerten. Der Wert *0* zeigt an, dass keine Fehler aufgetreten sind und die Methode erfolgreich aufgerufen wurde. Dieser Wert wird in der nächsten Codezeile ausgegeben.

Das vollständige Skript *SetShareInfo.ps1* ist wie folgt aufgebaut:

SetShareInfo.ps1

```
$shareName="'FSO'"
$maxAllowed="5"
$description="Test"
$wmiClass="Win32_share"
$objService=Get-WmiObject -Class $wmiClass -filter "name=$shareName"
$errRTN=$objService.setShareInfo($maxAllowed,$description)

"Die Methode SetShareInfo wurde mit folgendem Rückgabecode beendet: $($errRTN.returnvalue)"
```

Angeben von Parametern im Skript

Das vorherige Skript ist nützlich und veranschaulicht das Verfahren zum Ändern der Freigabebeschreibung und der maximalen Benutzeranzahl. Sie müssen das Skript jedoch manuell bearbeiten, um Änderungen vorzunehmen. Sie können das Skript allerdings auch so implementieren, dass das Skriptverhalten in Windows PowerShell gesteuert werden kann. Konfigurieren Sie das Skript so, dass dieses benannte Argumente akzeptiert. Benannte Argumente werden in der Windows PowerShell als *Parameter* bezeichnet.

Bewährte Vorgehensweise Da die wichtigsten Werte für das Skript *SetShareInfo.ps1* bereits in Variablen gespeichert sind und nicht im Methodenaufruf hart codiert wurden, ist das Hinzufügen der Parameterfunktionalität zum Skript einfach. Sie sollten alle Werte in Variablen speichern und die Variablen in den Methoden- und Funktionsaufrufen verwenden. Dies bedeutet zwar mehr Arbeit, erhöht aber die Flexibilität des Skripts und vereinfacht das Erstellen komplexerer Skripts unter Verwendung des gleichen Codes. Weitere Informationen zum Strukturieren von Skripts und bewährte Vorgehensweisen bei der Skriptentwicklung finden Sie im Buch *Microsoft VBScript Step by Step* (Microsoft Press, 2006).

Um das Skript *SetShareInfo.ps1* so zu konfigurieren, dass dieses Befehlszeilenparameter akzeptiert, geben Sie die *param*-Anweisung an. Schließen Sie die Variablen *$shareName*, *$maxAllowed* und *$description* in der *param*-Anweisung in runde Klammern ein. Behalten Sie die Werte bei, die den Variablen bereits zugewiesen wurden, da diese als Standardwerte für das Skript verwendet werden. Wenn beim Ausführen des Skripts kein Wert für den Parameter angegeben wird, wird der entsprechende Standardwert verwendet. Wenn Sie also beim Aufrufen des Skripts für keinen der Parameter einen Wert angeben, wird das Skript mit den gleichen Funktionen wie das Skript *SetShareInfo.ps1* ausgeführt. Die geänderte Codezeile lautet:

```
param($shareName="'FSO'", $maxAllowed=5, $description="Test")
```

Das vollständige Skript *SetShareInfoWithParameters.ps1* umfasst folgende Anweisungen:

SetShareInfoWithParameters.ps1

```
param($shareName="'FSO'", $maxAllowed=5, $description="Test")

$wmiClass="Win32_share"
$objService=Get-WmiObject -Class $wmiClass -filter "name=$shareName"
$errRTN=$objService.setShareInfo($maxAllowed,$description)

"Die Methode SetShareInfo wurde mit folgendem Rückgabecode beendet: $($errRTN.returnvalue)"
```

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Umwandeln des Rückgabecodes

Der letzte Schritt, den Sie beim Festlegen der Freigabeinformationen ausführen müssen, ist das Umwandeln des Rückgabecodes. Der umgewandelte Code macht es einfacher, zu erkennen, ob ein Problem bei der Skriptausführung aufgetreten ist. Platzieren Sie den Code für die Umwandlung in einer Funktion, damit das Skript übersichtlich bleibt.

Weiterführende Informationen Die Werte, die von der Methode *SetShareInfo* für die WMI-Klasse *Win32_Share* zurückgegeben werden, sind im Windows SDK aufgeführt. Das Windows SDK ist unter *http://www.microsoft.com/downloads* und *http://msdn2.microsoft.com/en-us/default.aspx* verfügbar.

Das Skript *SetShareInfoWithParametersTranslateRtnValue.ps1* beginnt mit der *param*-Anweisung. Die *param*-Anweisung ermöglicht die Eingabe der Befehlszeilenparameter. Jede Variable beginnt mit einem Dollarzeichen und erhält einen Standardwert zugewiesen. Wenn der Parameter in der Befehlszeile angegeben wird, überschreibt dieser Wert den Standardwert aus der *param*-Anweisung. Das Skript verwendet den Wert aus der *param*-Anweisung für den Parameter, wenn ein Parameter nicht angegeben wurde. Ein Vorteil der *param*-Anweisung ist deren Flexibilität. Sie können keinen, alle oder eine beliebige Anzahl von Befehlszeilenparametern angeben.

Die Funktion *funlookup* aus dem Skript *SetShareInfoWithParametersTranslateRtnValue.ps1* akzeptiert eine ganze Zahl. Wenn die Funktion *funlookup* aufgerufen wird, übergeben Sie den Rückgabewert der Funktion *ShareInfo* als Eingabeparameter an *funlookup*:

```
funlookup($errRTN.returnValue)
```

Die Funktion *funlookup* umfasst eine *switch*-Anweisung, die den an die Funktion in der Variablen *$intIN* übergebenen Wert auswertet. Wenn für den Rückgabecode keine Übereinstimmung gefunden wird, wird die Standardzeichenfolge angezeigt, einschließlich des Fehlercodes und einer entsprechenden Meldung. Die Funktion *funlookup* ist wie folgt implementiert:

```
Function funlookup($intIN)
{
 Switch($intIN)
 {
  0  { "Erfolg" }
  2  { "Zugriff verweigert" }
  8  { "Unbekannter Fehler" }
  9  { "Ungültiger Name" }
  10 { "Ungültige Ebene" }
  21 { "Ungültiger Parameter" }
  22 { "Doppelte Freigabe" }
  23 { "Umgeleiteter Pfad" }
  24 { "Unbekanntes Gerät oder Verzeichnis" }
  25 { "Netzwerkname nicht gefunden" }
  DEFAULT { "$intIN ist ein unbekannter Wert." }

 }
}
```

Das vollständige Skript *SetShareInfoWithParametersTranslateRtnValue.ps1* hat folgenden Aufbau:

SetShareInfoWithParametersTranslateRtnValue.ps1

```
param($shareName="'FSO'", $maxAllowed=5, $description="Test script")

Function funlookup($intIN)
{
 Switch($intIN)
 {
  0  { "Erfolg" }
  2  { "Zugriff verweigert" }
  8  { "Unbekannter Fehler" }
  9  { "Ungültiger Name" }
  10 { "Ungültige Ebene" }
  21 { "Ungültiger Parameter" }
  22 { "Doppelte Freigabe" }
  23 { "Umgeleiteter Pfad" }
  24 { "Unbekanntes Gerät oder Verzeichnis" }
  25 { "Netzwerkname nicht gefunden" }
  DEFAULT { "$intIN ist ein unbekannter Wert." }

 }
}

$wmiClass="Win32_share"
$objService=Get-WmiObject -Class $wmiClass -filter "name=$shareName"
$errRTN=$objService.setShareInfo($maxAllowed,$description)

#"Die Methode SetShareInfo wurde mit folgendem Rückgabecode beendet: $($errRTN.returnvalue)"

funlookup($errRTN.returnValue)
```

Erstellen neuer Freigaben

Um neue Freigaben zu erstellen, verwenden Sie die Methode *Create* aus der WMI-Klasse *Win32_Share*. Um diese Methode, anstatt das Cmdlet **Get-WmiObject** zu verwenden, erstellen Sie mit dem *[wmiclass]*-Accelerator eine neue Instanz der Klasse *Win32_Share*. Der Accelerator *[wmiclass]* erstellt eine Instanz der .NET Framework-Klasse *System.Management.ManagementObject*.

Geben Sie im Skript *CreateShare.ps1* eine *param*-Anweisung an, um die Eingabe von Befehlszeilenparametern zu ermöglichen. Für das Skript sind vier Parameter erforderlich. *Folderpath*, *Sharename, Maxallowed* und *Description*. Geben Sie in der *param*-Anweisung für die Parameter *Maxallowed* und *Description* mit folgender Codezeile Standardwerte an:

```
param($folderPath, $shareName, $maxAllowed=5, $description="Mit PowerShell erstellt.")
```

Wichtig Wenn Sie Argumente mit einer *param*-Anweisung an ein Skript übergeben, arbeiten Sie mit Parametern. Parameter sind keine Argumente, auch wenn sie manchmal als benannte Argumente bezeichnet werden. Beispielsweise umfasst die automatische Variable *$args* keine Parameter. In der automatischen Variablen *$args* werden die beim Skriptaufruf angegebenen Argumente gespeichert. Wenn Sie im Skript *CreateShare.ps1* die *param*-Anweisung angeben, ist der Wert von *$args* immer 0. Wenn Parameter mit Argumenten identisch wären, würde *$args* anzeigen, wie viele Parameter im Skript angegeben wurden.

Um sicherzustellen, dass die beiden erforderlichen Parameter an das Skript übergeben werden, verwenden Sie zwei *if*-Anweisungen. Diese beiden Codezeilen werden nach den Funktionsdefinitionen eingegeben. Wird ein Parameter nicht angegeben, ist dessen Variable, in der der benannte Parameter gespeichert ist, nicht vorhanden. Geben Sie in diesem Fall eine Fehlermeldung mit dem fehlenden Parameter und mit der Funktion *funhelp* die Hilfe für das Skript aus. Die beiden Codezeilen lauten:

```
if(!($folderpath)) { "Sie müssen einen Pfad angeben." ; funHelp }
if(!($sharename))  { "Sie müssen einen Namen angeben." ; funHelp }
```

Die Funktion *funhelp* kann drei Aktionen ausführen. Sie können einen langen Hilfetext in einer *here*-Zeichenfolge eingeben (beispielsweise eine Beschreibung aller Parameter und Syntaxbeispiele). Die zweite Aktion, die Sie mit der Funktion *funhelp* ausführen können, ist die Ausgabe des Textes der *here*-Zeichenfolge, die der Variablen *helpText* zugewiesen ist. Die dritte Aktion der Funktion *funhelp* beendet das Skript. Die Funktion *funhelp* ist wie folgt implementiert:

```
$helpText=@"

NAME: CreateShare.ps1
Erstellt eine Freigabe auf dem lokalen Computer unter Verwendung der Standardberechtigungen.
Der freizugebende Ordner muss nicht existieren, da das Skript das Vorhandensein des Ordners
überprüft und diesen falls erforderlich erstellt.

PARAMETER:
-folderPath  Gibt den Pfad zu dem Ordner an, der freigegeben werden soll.
-shareName   Gibt den Namen an, der der Freigabe zugewiesen werden soll.
-maxAllowed  [optional] Legt die maximale Anzahl gleichzeitiger Verbindungen fest.
-description [optional] Gibt eine Beschreibung der Freigaben an (Hinweise, Grund der Freigabe,  etc.).

SYNTAX:
CreateShare.ps1 -folderPath "C:\FSO" -shareName "FSO"

Erstellt eine Freigabe für den Ordner C:\FSO und weist dieser den Namen FSO zu.
5 Benutzer können auf die Freigabe gleichzeitig zugreifen und
die Beschreibung lautet: Mit PowerShell erstellt.

CreateShare.ps1 -folderPath "C:\FSO" -shareName "FSO" -maxAllowed 1

Erstellt eine Freigabe für den Ordner C:\FSO und weist dieser den Namen FSO zu.
Nur 1 Benutzer kann auf die Freigabe gleichzeitig zugreifen und
die Beschreibung lautet: Mit PowerShell erstellt.

CreateShare.ps1 -folderPath "C:\FSO" -shareName "FSO" -maxAllowed 3
-description "FSO Freigabe"

Erstellt eine Freigabe für den Ordner C:\FSO und weist dieser den Namen FSO zu.
3 Benutzer können auf die Freigabe gleichzeitig zugreifen und
die Beschreibung lautet: FSO Freigabe.

"@
$helpText
exit
}
```

Wenn der freizugebende Ordner nicht bereits auf dem Computer vorhanden ist, müssen Sie den Ordner erstellen. Überprüfen Sie als Erstes mit dem Cmdlet **Test-Path**, ob der Pfad existiert. Wenn der Ordner nicht vorhanden ist, geben Sie eine Meldung aus, dass der Ordner erstellt wird. Erstellen Sie

anschließend den Ordner mit dem Cmdlet **New-Item**. Der Code für die Funktion hat folgendes Aussehen:

```
if(!(Test-Path $folderPath))
 {
  "Erstellen von $folderPath ..."
  New-Item -Path $folderPath -type directory
 }
```

Nachdem Sie sichergestellt haben, dass die erforderlichen Parameter und der freizugebende Ordner vorhanden sind, können Sie die Freigabe erstellen. Erstellen Sie hierzu eine Instanz der Klasse *Win32_Share* und verwenden Sie anschließend die Methode *Create* dieser WMI-Klasse. Mit dem *[wmiclass]*-Accelerator lässt sich die Klasse einfach erstellen und einer Variablen zuweisen. Rufen Sie anschließend die *Create*-Methode mit den erforderlichen Parametern auf. Geben Sie alle Parameter (außer für die Sicherheitseinstellungen) in Variablen an, um den Prozess zu vereinfachen. Die beiden Codezeilen lauten:

```
$objWMI = [wmiClass]$class
$errRTN=$objWMI.create($folderPath, $shareName, $Type, $MaxAllowed, $description)
```

Da beim Aufrufen von Methoden Probleme auftreten können, sollten Sie das von der Methode zurückgegebene *error*-Objekt auswerten. Weisen Sie das *error*-Objekt der Variablen *$errRTN* zu und übergeben Sie den Rückgabewert an die Funktion *error*. Diese Funktion wandelt den Rückgabewert und den codierten Wert in eine verständlichere Zeichenfolge um. Die Funktion ist wie folgt implementiert:

```
Function funlookup($intIN)
{
 Switch($intIN)
 {
  0  { "Erfolg" }
  2  { "Zugriff verweigert" }
  8  { "Unbekannter Fehler" }
  9  { "Ungültiger Name" }
  10 { "Ungültige Ebene" }
  21 { "Ungültiger Parameter" }
  22 { "Doppelte Freigabe" }
  23 { "Umgeleiteter Pfad" }
  24 { "Unbekanntes Gerät oder Verzeichnis" }
  25 { "Netzwerkname nicht gefunden" }
  DEFAULT { "$intIN ist ein unbekannter Wert." }

 }
}
```

Das vollständige Skript *CreateShare.ps1* ist im folgenden Abschnitt dargestellt. Um das Skript auszuführen, müssen Sie den Ordnerpfad und den Namen der zu erstellenden Freigabe angeben. Sie können außerdem einen Wert für *maxallowed* und eine Beschreibung für die Freigabe eingeben. Wenn Sie das Skript ohne Parameter ausführen, werden die Hilfeinformationen einschließlich mehrerer Syntaxbeispiele angezeigt.

CreateShare.ps1

```
param($folderPath, $shareName, $maxAllowed=5, $description="Mit PowerShell erstellt.")

function funHelp()
{
```

```
$helpText=@"

NAME: CreateShare.ps1
Erstellt eine Freigabe auf dem lokalen Computer unter Verwendung der Standardberechtigungen.
Der freizugebende Ordner muss nicht existieren, da das Skript das Vorhandensein des Ordners
überprüft und diesen falls erforderlich erstellt.

PARAMETER:
-folderPath  Gibt den Pfad zu dem Ordner an, der freigegeben werden soll.
-shareName   Gibt den Namen an, der der Freigabe zugewiesen werden soll.
-maxAllowed  [optional] Legt die maximale Anzahl gleichzeitiger Verbindungen fest.
-description [optional] Gibt eine Beschreibung der Freigaben an (Hinweise, Grund der Freigabe, etc.).

SYNTAX:
CreateShare.ps1 -folderPath "C:\FSO" -shareName "FSO"

Erstellt eine Freigabe für den Ordner C:\FSO und weist dieser den Namen FSO zu.
5 Benutzer können auf die Freigabe gleichzeitig zugreifen und
die Beschreibung lautet: Mit PowerShell erstellt.

CreateShare.ps1 -folderPath "C:\FSO" -shareName "FSO" -maxAllowed 1

Erstellt eine Freigabe für den Ordner C:\FSO und weist dieser den Namen FSO zu.
Nur 1 Benutzer kann auf die Freigabe gleichzeitig zugreifen und
die Beschreibung lautet: Mit PowerShell erstellt.

CreateShare.ps1 -folderPath "C:\FSO" -shareName "FSO" -maxAllowed 3
-description "FSO Freigabe"

Erstellt eine Freigabe für den Ordner C:\FSO und weist dieser den Namen FSO zu.
3 Benutzer können auf die Freigabe gleichzeitig zugreifen und
die Beschreibung lautet: FSO Freigabe.

"@
$helpText
exit
}

Function funlookup($intIN)
{
 Switch($intIN)
 {
  0  { "Erfolg" }
  2  { "Zugriff verweigert" }
  8  { "Unbekannter Fehler" }
  9  { "Ungültiger Name" }
  10 { "Ungültige Ebene" }
  21 { "Ungültiger Parameter" }
  22 { "Doppelte Freigabe" }
  23 { "Umgeleiteter Pfad" }
  24 { "Unbekanntes Gerät oder Verzeichnis" }
  25 { "Netzwerkname nicht gefunden" }
  DEFAULT { "$intIN ist ein unbekannter Wert." }

 }
}
```

```
if(!($folderpath)) { "Sie müssen einen Pfad angeben." ; funHelp }
if(!($sharename))  { "Sie müssen einen Namen angeben." ; funHelp }

$class = "Win32_share"
$Type = 0
if(!(Test-Path $folderPath))
 {
  "Erstellen von $folderPath ..."
  New-Item -Path $folderPath -type directory
 }
$objWMI = [wmiClass]$class
$errRTN=$objWMI.create($folderPath, $shareName, $Type, $MaxAllowed, $description)
funLookup($errRTN.returnValue)
```

In Abbildung 5.9 ist eine Beispielfreigabe dargestellt.

Abbildung 5.9 Eine mit dem Skript *CreateShare.ps1* erstellte Freigabe

Erstellen mehrerer Freigaben

Um mehrere Freigaben gleichzeitig zu erstellen, die über die Befehlszeile eingegeben werden können, ändern Sie das Skript *CreateShare.ps1*, damit dieses mehrere Freigabe- und Ordnernamen akzeptiert.

CreateMultipleShares.ps1

```
param($folderPath, $shareName, $maxAllowed=5, $description="Mit PowerShell erstellt.")

function funHelp()
{
$helpText=@"

NAME: CreateMultipleShares.ps1
Erstellt Freigaben auf dem lokalen Computer unter Verwendung der Standardberechtigungen.
Die freizugebenden Ordner müssen nicht existieren, da das Skript das Vorhandensein der Ordner
```

```
überprüft und diese falls erforderlich erstellt.

PARAMETER:
-folderPath  Gibt den Pfad zu einem Ordner an, der freigegeben werden soll.
-shareName   Gibt den Namen an, der der Freigabe zugewiesen werden soll.
-maxAllowed  [optional] Legt die maximale Anzahl gleichzeitiger Verbindungen fest.
-description [optional] Gibt eine Beschreibung der Freigaben an (Hinweise, Grund der Freigabe,  etc.).

SYNTAX:
CreateMultipleShares.ps1 -folderPath "C:\FSO", "C:\FSO1" `
-shareName "FSO", "FSO1"

Erstellt zwei Freigaben für die Ordner C:\FSO und  C:\FSO1 und weist diesen die Namen FSO und FSO1 zu.
5 Benutzer können auf die Freigaben gleichzeitig zugreifen und
die Beschreibung lautet: Mit PowerShell erstellt.

CreateMultipleShares.ps1 -folderPath "C:\FSO", "C:\FSO1" `
-shareName "FSO", "FSO1" -maxAllowed 1

Erstellt zwei Freigaben für die Ordner C:\FSO und  C:\FSO1 und weist diesen die Namen FSO und FSO1 zu.
Nur 1 Benutzer kann auf die Freigaben gleichzeitig zugreifen und
die Beschreibung lautet: Mit PowerShell erstellt.

CreateMultipleShares.ps1 -folderPath "C:\FSO", "C:\FSO1", "C:\FSO2" `
-shareName "FSO", "FSO1", "FSO2" -maxAllowed 3 -description "fso share"

Erstellt drei Freigaben für die Ordner C:\FSO, C:\FSO1 und C:\FSO2
und weist diesen die Namen FSO, FSO1 und FSO2 zu.
3 Benutzer können auf die Freigaben gleichzeitig zugreifen und
die Beschreibung lautet: FSO Freigabe.

"@
$helpText
exit
}

Function funlookup($intIN)
{
 Switch($intIN)
 {
  0  { "Erfolg" }
  2  { "Zugriff verweigert" }
  8  { "Unbekannter Fehler" }
  9  { "Ungültiger Name" }
  10 { "Ungültige Ebene" }
  21 { "Ungültiger Parameter" }
  22 { "Doppelte Freigabe" }
  23 { "Umgeleiteter Pfad" }
  24 { "Unbekanntes Gerät oder Verzeichnis" }
  25 { "Netzwerkname nicht gefunden" }
  DEFAULT { "$intIN ist ein unbekannter Wert." }
 }
}
```

```
if(!($folderpath)) { "Sie müssen einen Pfad angeben." ; funHelp }
if(!($sharename))  { "Sie müssen einen Namen angeben." ; funHelp }

$class = "Win32_share"
$Type = 0
$iLength = $folderPath.length-1

for($i=0;$i -le $iLength;$i++)
{
if(!(Test-Path $folderPath[$i]))
 {
  "Erstellen von $folderPath ..."
  New-Item -Path $folderPath[$i] -type directory
 }
$objWMI = [wmiClass]$class

$folder= $folderPath[$i]
$share= $shareName[$i]
$errRTN=$objWMI.create($folder, $share, $Type, $MaxAllowed, $description)
funLookup($errRTN.returnValue)
}
```

Entfernen von Freigaben

Um eine Freigabe mit einem Windows PowerShell-Skript zu löschen, verwenden Sie die WMI-Klasse *Win32_Share* und die Methode *Delete*.

Das Skript *DeleteShare.ps1* beginnt mit einer *param*-Anweisung, die das Eingeben von Befehlszeilenparametern ermöglicht. Der Parameter, der den Namen der zu entfernenden Eingabe angibt, ist erforderlich. Wird der Parameter **-computername** ausgelassen, verwendet das Skript den Standardwert *localhost*, um eine lokale Freigabe zu löschen. Codezeile:

```
Param($shareName, $computerName="localhost")
```

Das Skript wertet die Befehlszeilenparameter aus und überprüft, ob der Parameter *shareName* vorhanden ist. Wurde der Parameter *shareName* nicht angegeben, wird eine Meldung ausgegeben, dass der Parameter nicht vorhanden ist. Das Skript ruft entsprechend die Funktion *funhelp* auf. Codezeile:

```
if(!($ShareName)) { "you must supply a shareName" ; funHelp }
```

Die Funktion *funhelp* verwendet eine *here*-Zeichenfolge, um die Definition des Hilfetextes zu vereinfachen. Die *here*-Zeichenfolge wird der Variablen *$helpText* zugewiesen, die vor dem Beenden des Skripts ausgegeben wird. Die Funktion wird nur dann aufgerufen, wenn der Parameter *shareName* nicht vorhanden ist:

```
function funHelp()
{
$helpText=@"

NAME: DeleteShare.ps1
Entfernt eine Freigabe auf der lokalen Arbeitsstation oder einem Remotecomputer
 unter Verwendung der Berechtigungen des gegenwärtig angemeldeten Benutzers.

PARAMETER:
-shareName     Gibt den Namen der Freigabe an.
-computerName  [optional] Gibt den Namen des Computers an, auf dem die Freigaben entfernt werden soll.
```

```
SYNTAX:
DeleteShare.ps1 -shareName "fso"

Entfernt die Freigabe namens FSO vom lokalen Computer.

DeleteShare.ps1 -shareName "fso" -computerName "london"

Entfernt die Freigabe namens FSO vom Remotecomputer London.

"@
$helpText
exit
}
```

Um die Freigabe zu löschen, rufen Sie die Methode *Delete* aus der WMI-Klasse *Win32_Share* auf. Codeabschnitt:

```
$objWMI= Get-WmiObject -Class $wmiClass -computername $computerName `
-filter "Name = '$shareName'"
$objWMI.delete()
```

Das Skript *DeleteShare.ps1* hat folgenden Aufbau:

DeleteShare.ps1

```
Param($shareName, $computerName="localhost")

function funHelp()
{
$helpText=@"

NAME: DeleteShare.ps1
Entfernt eine Freigabe auf der lokalen Arbeitsstation oder einem Remotecomputer
unter Verwendung der Berechtigungen des gegenwärtig angemeldeten Benutzers.

PARAMETER:
-shareName     Gibt den Namen der Freigabe an.
-computerName  [optional] Gibt den Namen des Computers an, auf dem die Freigaben entfernt werden soll.

SYNTAX:
DeleteShare.ps1 -shareName "fso"

Entfernt die Freigabe namens FSO vom lokalen Computer.

DeleteShare.ps1 -shareName "fso" -computerName "london"

Entfernt die Freigabe namens FSO vom Remotecomputer London.

"@
$helpText
exit
}

if(!($ShareName)) { "Sie müssen den Parameter shareName angeben." ; funHelp }
$wmiClass = "Win32_Share"
$objWMI= Get-WmiObject -Class $wmiClass -computername $computerName `
-filter "Name = '$shareName'"
$objWMI.delete()
```

Entfernen nicht autorisierter Freigaben

Für die Konfigurationsverwaltung (Desired Configuration Maintenance, DCM) ist es wichtig, dass die Freigaben auf einem Server oder einer Arbeitsstation überwacht werden. Alle Freigaben sollten genehmigt und mit Standardeinstellungen konfiguriert sein. Nicht autorisierte Freigaben sollten entfernt werden. Sie haben mit dem Skript *WriteSharesToFile.ps1* bereits die Freigaben auf einem Computer in einer Textdatei erfasst. Anschließend haben Sie mit dem Skript *AuditUnauthorizedShares.ps1* den Inhalt dieser Textdatei mit der aktuellen Freigabekonfiguration vergleichen. Um die gewünschte Konfiguration eines Servers zu unterstützen, müssen Sie nun alle nicht autorisierten Freigaben entfernen. Ändern Sie hierzu das Skript *AuditUnauthorizedShares.ps1* so ab, dass dieses die nicht autorisierten Freigaben löscht.

Die einzige Änderung, die Sie vornehmen müssen, betrifft das Hinzufügen einer Anweisung zur vorhandenen *else*-Klausel der *if ... else*-Anweisung, um den Löschvorgang auszuführen. Beachten Sie, dass der Code beinahe mit dem Code im Skript *DeleteShare.ps1* identisch ist.

```
$wmiClass = "Win32_Share"
$objWMI= Get-WmiObject -Class $wmiClass -filter "Name = '$($share.Name)'"
$objWMI.delete()
```

Das vollständige Skript *DeleteUnauthorizedShares.ps1* hat folgenden Aufbau:

DeleteUnauthorizedShares.ps1

```
Clear-Host
$strFile = Get-Content "c:\FSO\Shares.txt"

$strQuery = "Select * from win32_share"
$shares = get-wmiobject -query $strQuery

foreach ( $share in $shares)
 {
   $shareName = $($share.name).tostring()
   $shareName = $shareName.substring(0,$shareName.length-1)

  Write-Host "Erfassen der Freigabe $($share.Name) ..." -ForegroundColor yellow

  if ( $strFile -match $shareName )
   { Write-Host "`t$($share.name) wurde gefunden." -foregroundcolor Green}
  ELSE
   {
    Write-Host "`t$($share.Name) ist nicht autorisiert und wird entfernt ..."
-foregroundcolor red
   $wmiClass = "Win32_Share"
    $objWMI= Get-WmiObject -Class $wmiClass -filter "Name = '$($share.Name)'"
    $objWMI.delete()

   }
 }
```

Zusammenfassung

In diesem Kapitel wurde die Verwaltung von Freigaben behandelt. Sie haben die aktuellen benutzerdefinierten und administrativen Freigaben auf einem Computer dokumentiert. Anschließend wurden die zum Erstellen neuer Freigaben und Festlegen bestimmter Servereinstellungen erforderlichen Schritte erklärt. Das Kapitel wurde mit dem Überwachen von Freigaben und dem Entfernen nicht autorisierter Freigaben abgeschlossen.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

KAPITEL 6

Verwalten von Druckern

Nach Abschluss dieses Kapitels können Sie:

- Die Druckeranzahl ermitteln
- Druckertreiber installieren und verwalten
- Drucker freigeben

Auf der Begleit-CD Alle Skripts in diesem Kapitel befinden sich auf der Begleit-CD im Ordner *\Scripts\Chapter06*.

Ermitteln der Druckeranzahl

Wer weiß, wie viele Drucker in einem Netzwerk installiert sind? Wer kann mit den freigegebenen Druckern in Arbeitsgruppen oder kleinen Niederlassungen Schritt halten? Das Verwalten von Druckern stellt für die meisten Netzwerkadministratoren ein großes Problem dar.

Für viele Administratoren ist diese scheinbar einfache Aufgabe ausgesprochen mühsam. Durch die umsichtige Verwendung von Windows PowerShell und Windows Management Instrumentation (WMI) können Sie jedoch schnell Ordnung ins Chaos bringen. Dies ist im Skript *ListPrinters.ps1* veranschaulicht. Im Skript *ListPrinters.ps1* speichern Sie die Zeichenfolge *Win32_Printer* in der Variablen *$class*, die dann in der WMI-Abfrage verwendet wird. Geben Sie den Namen des Computers für die Abfrage in der Variablen *$computer* an. Die Variable *$wmi* enthält die Objekte, die vom Cmdlet **Get-WmiObject** zurückgegeben werden, das die Informationen zu den Druckern auf dem Computer abruft, der in der Variablen *$computer* angegeben ist. Die vom Skript *ListPrinters.ps1* abgerufenen Drucker sind mit den Druckern identisch, die im Applet **Drucker** in der Systemsteuerung angezeigt werden (siehe Abbildung 6.1).

Nachdem das Cmdlet **Get-WmiObject** die Objekte zurückgegeben hat, können Sie die Ausgabe mit dem Cmdlet **Format-Table** formatieren. Geben Sie im Cmdlet **Format-Table** das Argument **-property** an, um die Eigenschaften auszuwählen, die in die Ausgabe einbezogen werden sollen. Wählen Sie für dieses Beispiel die Eigenschaften *Name*, *SystemName* und *ShareName* aus. Verwenden Sie das Argument **-groupby**, um die Ausgabe anhand des Treibers zu formatieren. Das Argument **-inputobject** ermöglicht die Eingabe für das Cmdlet. Verwenden Sie das Objekt, das vom Cmdlet **Get-WmiObject** zurückgegeben wurde. Das Objekt ist der Variablen *$wmi* zugewiesen. Das vollständige Skript *ListPrinters.ps1* hat folgenden Aufbau:

ListPrinters.ps1

```
$class = "win32_printer"
$computer = "localhost"
$wmi = Get-WmiObject -Class $class -computername $computer
format-table -Property name, systemName, shareName -groupby driverName `
-inputobject $wmi -autosize
```

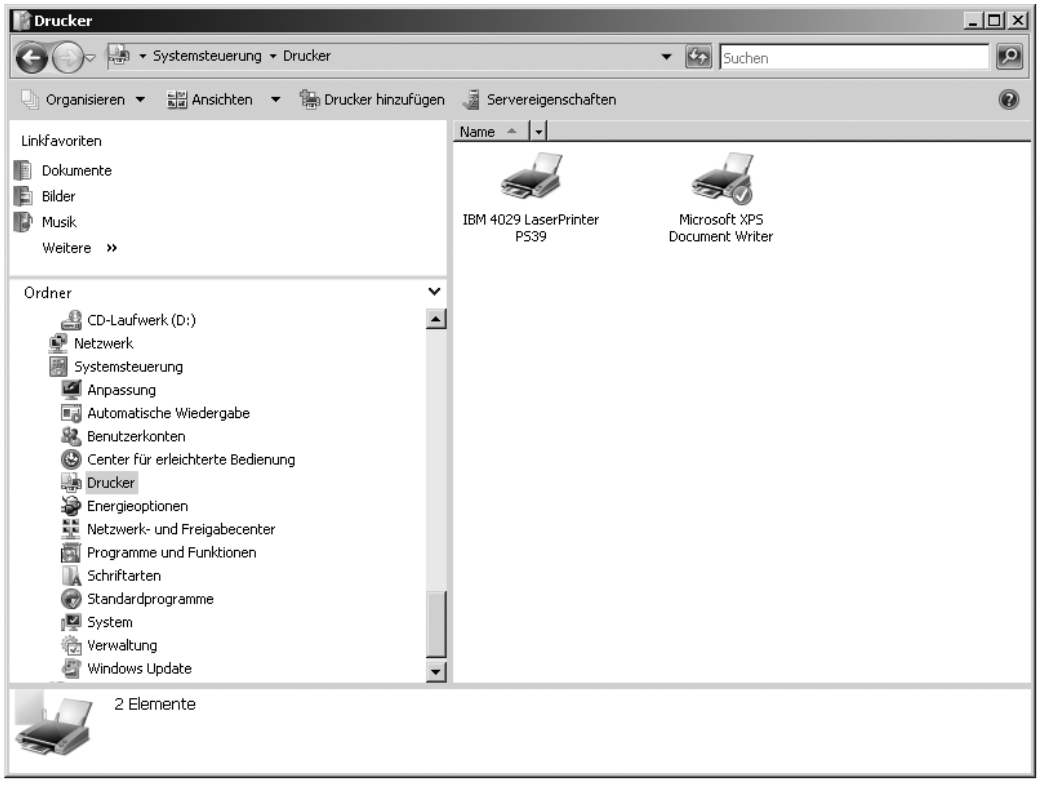


Abbildung 6.1 Die in der Systemsteuerung angezeigten Drucker auf einem Windows Server 2008-Computer

Eine Beispielausgabe des Skripts *ListPrinters.ps1* ist in folgendem Listing dargestellt. Beachten Sie, dass jeder Eintrag mit dem Treibernamen beginnt, da Sie das Argument **-groupby** angegeben haben. Diese Methode ist vorteilhaft, wenn das Skript zur Abfrage eines Druckservers verwendet wird, der zahlreiche Drucker umfasst.

```
   driverName: Microsoft XPS Document Writer

name                          systemName shareName
----                          ---------- ---------
Microsoft XPS Document Writer M5-1875135


   driverName: IBM 4029 LaserPrinter PS39

name                       systemName shareName
----                       ---------- ---------
IBM 4029 LaserPrinter PS39 M5-1875135
```

Abfragen mehrerer Computer

Eine einfache Methode zum gleichzeitigen Abfragen mehrerer Computer oder Server eröffnet sich durch Anpassung des Skripts *ListPrinters.ps1*, um mehrere Computernamen einzubeziehen. Ändern Sie hierzu die Variable *$computer*. Um die Computernamen zu durchlaufen, verwenden Sie eine

foreach-Anweisung und *$computer* als Enumerator. Damit Sie das Skript nicht ständig ändern müssen, können Sie die Variable *$arycomputer* erstellen und die Computernamen für die Abfrage in dieser Variablen speichern. Fügen Sie eine *foreach*-Anweisung hinzu, um die Computernamen in der Variablen *$arycomputer* zu überprüfen. Das vollständige Skript *ListPrintersFromMultipleComputers.ps1* umfasst folgende Anweisungen:

ListPrintersFromMultipleComputers.ps1

```
$class = "win32_printer"
$arycomputer = "localhost", "loopback"
foreach( $computer in $aryComputer)
{
 Write-Host "Ermitteln der Durcker auf $computer ..."
 $wmi = Get-WmiObject -Class $class -computername $computer
 format-table -Property name, systemName, shareName -groupby driverName `
 -inputobject $wmi -autosize
}
```

Das Skript *ListPrintersFromMultipleComputers.ps1* listet alle Computer auf, die in der Variablen *$arycomputer* angegeben sind. Wenn zahlreiche Computer vorhanden sind, müssen Sie verdeutlichen, welche Drucker welchen Computern zugeordnet sind. Geben Sie hierzu mit dem Cmdlet **Write-Host** den Wert von *$computer* aus, bevor Sie die Drucker abrufen. (In Abbildung 6.2 sind einige Druckereigenschaften dargestellt.) Die Ausgabe des Cmdlets entspricht folgendem Listing:

```
Ermitteln der Durcker auf localhost ...


   driverName: Microsoft XPS Document Writer

name                          systemName shareName
----                          ---------- ---------
Microsoft XPS Document Writer M5-1875135


   driverName: IBM 4029 LaserPrinter PS39

name                       systemName shareName
----                       ---------- ---------
IBM 4029 LaserPrinter PS39 M5-1875135


Ermitteln der Durcker auf loopback ...


   driverName: Microsoft XPS Document Writer

name                          systemName shareName
----                          ---------- ---------
Microsoft XPS Document Writer M5-1875135


   driverName: IBM 4029 LaserPrinter PS39

name                       systemName shareName
----                       ---------- ---------
IBM 4029 LaserPrinter PS39 M5-1875135
```

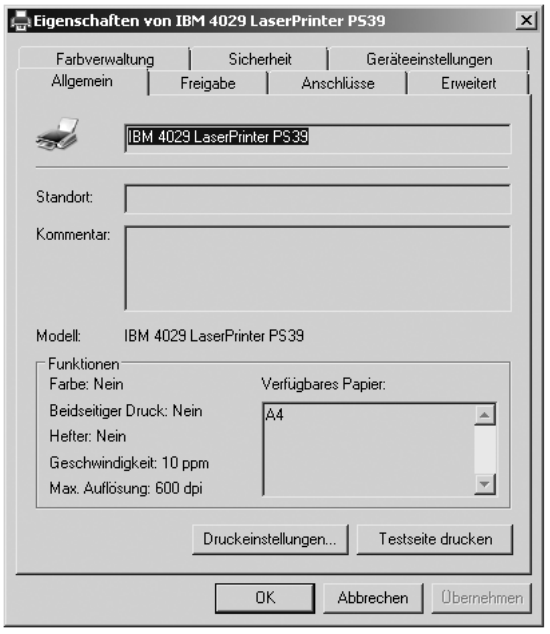


Abbildung 6.2 Beispiel der Druckereigenschaften

Protokollieren in eine Datei

Sie können die mit der WMI-Abfrage zusammengestellten Informationen permanent in einer Textdatei speichern. Textdateien haben den Vorteil, dass diese einfach zu verwenden sind, wenig Speicherplatz belegen und in anderen Anwendungen, beispielsweise Microsoft Office-Anwendungen, verarbeitet werden können.

Arbeiten mit Dateien

Windows PowerShell umfasst zahlreiche Cmdlets zur Verarbeitung von Textdateien. Beispielsweise liest **Get-Content** den Inhalt von Dateien aus und mit **Out-File** können Sie Textdateien erstellen. Textdateien können einfach erstellt, geändert und gelöscht werden. Die Windows PowerShell-Cmdlets machen die Arbeit mit Textdateien sogar noch einfacher. In Tabelle 6.1 sind die Cmdlets für die Arbeit mit Textdateien aufgeführt.

Tabelle 6.1 Cmdlets für Textdateien

Cmdlet	Verwendung
Out-File	Erstellt Dateien und verarbeitet verschiedene Codierungsschemas: Unicode, UTF 7,8,32, BigEndianUnicode und ASCII. Das Standardschema ist Unicode.
Get-Content	Gibt die Daten aus einer Datei zurück. Liest die Datei zeilenweise und gibt für jede Zeile ein anderes Objekt zurück. Dieses Cmdlet kann auch Anmeldeinformationen und die Codierung angeben.
Add-Content	Fügt Text zu einer Datei hinzu

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Tabelle 6.1 Cmdlets für Textdateien *(Fortsetzung)*

Cmdlet	Verwendung
Set-Content	Überschreibt den Text in einer Datei. Set-Content kann den ursprünglichen Inhalt zu einer Datei hinzufügen.
Clear-Content	Löscht Daten aus einer Datei, aber nicht die Datei

Deklarieren Sie im Skript *ListPrintersFromMultipleComputersWriteToFile.ps1* als Erstes die Variable *$filePath*, um den Pfad zur Datei anzugeben, die Sie mit dem Cmdlet **Out-File** erstellen möchten. Speichern Sie die WMI-Klasse, mit der Sie die Druckerinformationen abrufen, in der Variablen *$class*. In diesem Skript wird die WMI-Klasse *Win32_Printer* verwendet. Erstellen Sie dann ein Array mit den Namen der abgefragten Computer und rufen Sie die Drucker ab. In diesem Beispiel geben Sie die folgenden zwei Namen für den lokalen Computer an: *localhost* und *loopback*. Verwenden Sie diese Computernamen, um das Skript für mehrere Computer auszuführen.

Um das Array zu durchlaufen, verwenden Sie wie üblich eine *foreach*-Anweisung. Erstellen Sie die Variable *$computer* und verwenden Sie diese als Enumerator zum Abfragen der einzelnen Computer, die in der Variablen *$aryComputer* festgelegt sind. Geben Sie anschließend mit dem Cmdlet **Write-Host** eine Statusmeldung aus, die den Namen des aktuellen Computers und einen Hinweis enthält, dass die Druckerinformationen abgerufen werden.

Greifen Sie mit dem Cmdlet **Get-WmiObject** auf den WMI-Dienst auf dem Computer zu und rufen Sie die Druckerinformationen ab. Geben Sie im Cmdlet **Get-WmiObject** den Namen der WMI-Klasse und des Computer an, auf den Sie zugreifen. Speichern Sie das zurückgegebene WMI-Verwaltungsobjekt in der Variablen *$wmi*.

Das WMI-Verwaltungsobjekt wird mit dem Parameter **-inputobject** an das Cmdlet **Format-Table** übergeben. Wählen Sie die Eigenschaften *Name, SystemName* und *ShareName* des Verwaltungsobjekts aus. Gruppieren Sie die Liste nach den Treibernamen und geben Sie den Parameter **-autosize** an, um die Tabelle passend zu formatieren. Fügen Sie das resultierende Objekt in das Cmdlet **Out-File** ein und geben Sie den in der Variable *$filePath* gespeicherten Pfad im Parameter **-filepath** an. Geben Sie den Parameter **-encoding** an, um die Datei im ASCII-Format zu erstellen. Das vollständige Skript *ListPrintersFromMultipleComputersWriteToFile.ps1* hat folgenden Aufbau:

ListPrintersFromMultipleComputersWriteToFile.ps1

```
$filePath = "c:\FSO\Printers.txt"
$class = "win32_printer"
$arycomputer = "localhost", "loopback"
foreach( $computer in $aryComputer)
{
 Write-Host "Ermitteln der Durcker auf $computer ..."
 $wmi = Get-WmiObject -Class $class -computername $computer
 format-table -Property name, systemName, shareName -groupby driverName `
 -inputobject $wmi -autosize | Out-File -FilePath $filePath -encoding ASCII
}
```

Erfassen in einer Microsoft Access-Datenbank

Sie können die Druckerinformationen auch in einer Access-Datenbank speichern. In Abbildung 6.3 ist das Datenbankformat in Access dargestellt.

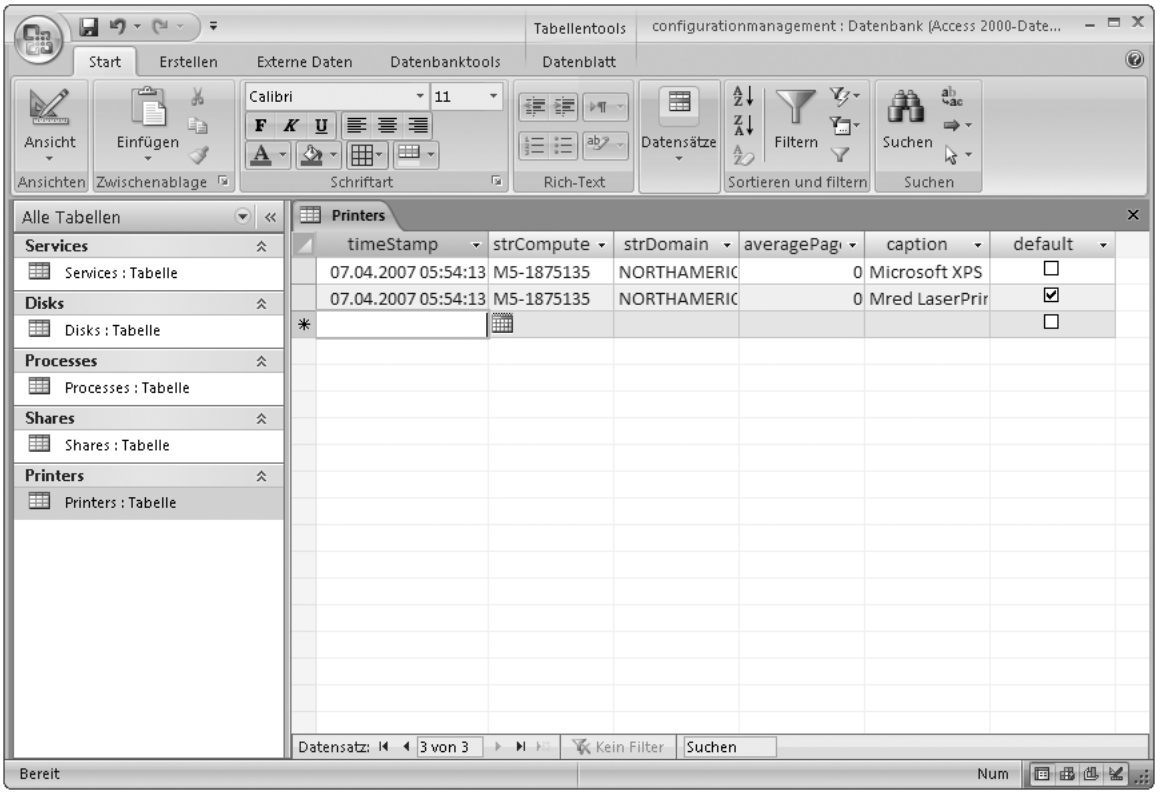


Abbildung 6.3 Access-Datenbank in der Tabellenentwurfsansicht

Überprüfen Sie auch das Berichtslayout im Berichts-Designer, bevor Sie die Informationen in der Datenbank speichern. In Abbildung 6.4 ist der Berichts-Designer in Access dargestellt.

Erstellen Sie im Skript *WritePrinterInfoToAccess.ps1* die Variable *$strComputer*, um den Computernamen zu speichern. Um den Computernamen abzurufen, erstellen Sie mit dem Cmdlet **New-Object** eine Instanz des *wshNetwork*-Objekts. Das *wshNetwork*-Objekt ist ein COM-Objekt mit der Programm-ID *wscript.network*. Geben Sie zum Erstellen des COM-Objekts runde Klammern ein und wählen Sie die Eigenschaft *ComputerName* aus. Der Computername wird in der Variablen *$strComputer* gespeichert.

Rufen Sie den Domänennamen unter Verwendung des gleichen Objekts und der gleichen Methode ab. Der Domänenname ist in der Eigenschaft *UserDomain* des *wshNetwork*-Objekts zu finden. Nachdem Sie die Daten abgerufen haben, speichern Sie den Wert der Eigenschaft *UserDomain* in der Variablen *$strDomain*. Erstellen Sie die Variable *$strWMIQuery*, um den Text der WMI-Abfrage anzugeben. In der WMI-Abfrage werden alle Eigenschaften der Druckerobjekte ausgewählt.

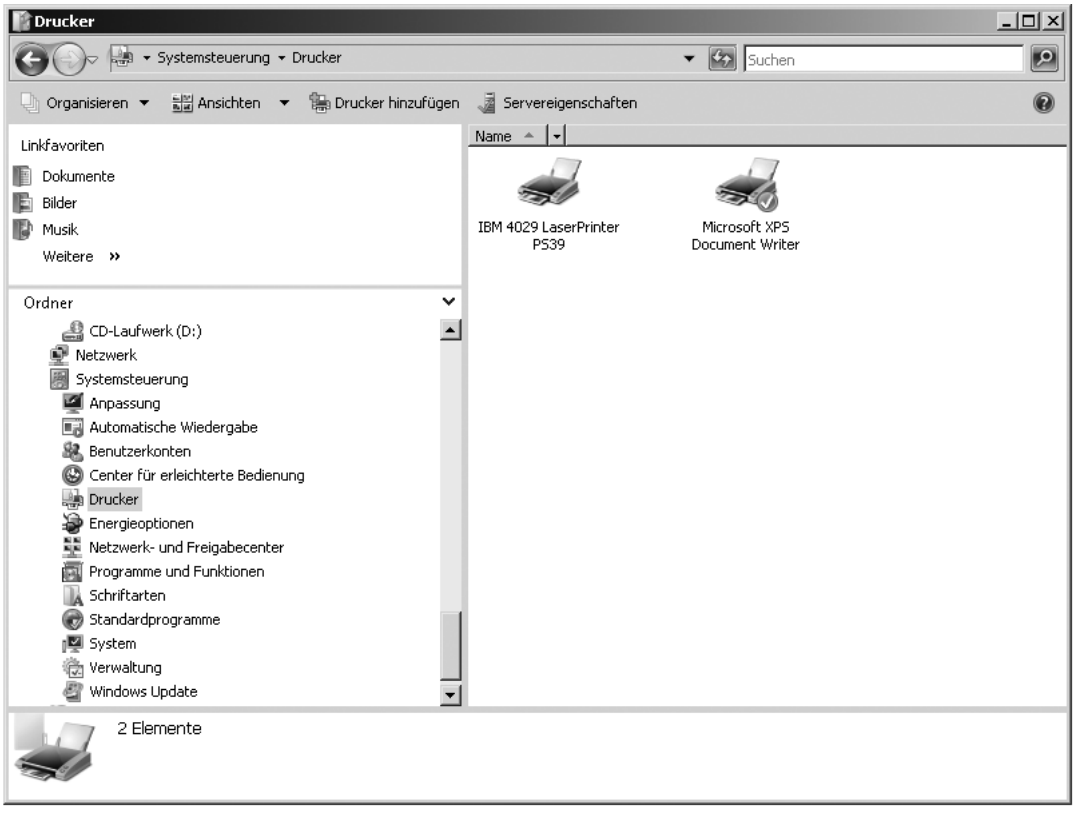


Abbildung 6.4 Das Layout der Berichtsfelder im Berichts-Designer von Access

Erstellen Sie die Variablen *$adOpenStatic* und *$adLockOptimistic*, um die Methode zum Herstellen der Datenbankverbindung festzulegen. Speichern Sie den Pfad zur Datenbank in der Variablen *$strDB*. Da Sie mit einer Access-Datenbank arbeiten, geben Sie den Pfad zur .mdb-Datei an. In der Variablen *$strTable* ist der Name der Tabelle gespeichert, in die Sie die Daten schreiben.

Um eine Datenbankverbindung herzustellen und Daten in der Datenbank zu speichern, müssen Sie zwei COM-Objekte erstellen. Das erste Objekt ist ein *Connection*-Objekt, um die Datenbank zu öffnen. Geben Sie die Programm-ID *ADODB.Connection* im Cmdlet **New-Object** an, um eine Instanz des *Connection*-Objekts zu instanziieren. Das vom Cmdlet **New-Object** zurückgegebene Objekt wird in der Variablen *$objConnection* gespeichert.

Das nächste COM-Objekt ist ein *RecordSet*-Objekt. Geben Sie die Programm-ID *ADODB.RecordSet* im Cmdlet **New-Object** an, um eine Instanz des *RecordSet*-Objekts zu erstellen. Das vom Cmdlet **New-Object** zurückgegebene Objekt wird in der Variablen *$obj*Recordset gespeichert.

Nachdem Sie die COM-Objekte erstellt haben, starten Sie den Verbindungsprozess. Öffnen Sie als Erstes die Datenbankverbindung, indem Sie den Anbieter und den Pfad zur Datenbankdatei angeben. Verwenden Sie zum Öffnen der Access-Datenbankverbindung den Jet OLEDB-Anbieter. Führen Sie hierzu folgenden Befehl aus:

```
$objConnection.Open("Provider = Microsoft.Jet.OLEDB.4.0; `
  Data Source= $strDB")
```

Nachdem die Datenbankverbindung hergestellt wurde, können Sie die *Open*-Methode des *RecordSet*-Objekts verwenden. Die *Open*-Methode erfordert vier Parameter, *query*, *connection*, *means* und *locking*, die im Folgenden dargestellt sind:

```
$objRecordSet.Open("SELECT * FROM $strTable", `
  $objConnection, $adOpenStatic, $adLockOptimistic)
```

Nachdem die Datenbankverbindung geöffnet wurde, sollten Sie mit dem Cmdlet **Write-Host** eine Statusmeldung ausgeben, dass die Druckerinformationen abgerufen werden. Zeigen Sie die Meldung mit dem Parameter **-foregroundcolor** in Gelb an.

Farbparameter für Write-Host

Für das Cmdlet Write-Host sind 16 Farbwerte verfügbar. Sie können die Farbwerte für die Parameter *-foreground* und *-background* festlegen. Das Anzeigen von Meldungen in verschiedenen Farben kann die Ausgabe verbessern. Sie sollten Farben jedoch vorsichtig verwenden, da die Konsolenfarbe möglicherweise von einem anderen Benutzer geändert wurde. Auch wenn die Ausgabe von Fehlermeldungen in Rot sinnvoll scheint, kann ein Benutzer einen roten Hintergrund verwenden. In diesem Fall, sind rote Fehlermeldungen für diesen Benutzer nicht sichtbar. Sie können dieses Problem auf zwei Arten umgehen. Legen Sie für Meldungen sowohl die Vordergrund- als auch die Hintergrundfarbe fest. Dies ist möglicherweise unansehnlich, aber die Meldungen können gelesen werden. Eine anspruchsvollere Methode ist das Ermitteln der Hintergrundfarbe und das Auswählen einer Kontrastfarbe. Sie können für das Cmdlet Write-Host die folgenden Farbkonstanten angeben:

Black (Schwarz)	DarkBlue (Dunkelblau)	DarkGreen (Dunkelgrün)	DarkCyan (Dunkelblaugrün)
DarkRed (Dunkelrot)	DarkMagenta (Dunkelviolett)	DarkYellow (Dunkelgelb)	Gray (Grau)
DarkGray (Dunkelgrau)	Blue (Blau)	Green (Grün)	Cyan (Blaugrün)
Red (Rot)	Magenta (Violett)	Yellow (Gelb)	White (Weiß)

Um die Farben mit der Hintergrundfarbe zu vergleichen, verwenden Sie das in Abbildung 6.5 dargestellte Skript. Sie finden das Skript *DemoWriteHostColors.ps1* im Ordner *Extras* auf der Begleit-CD.

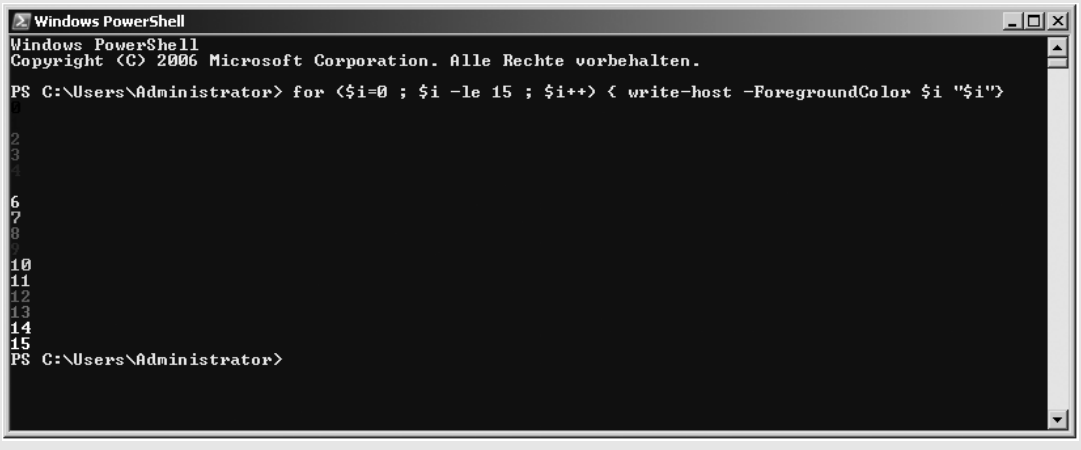


Abbildung 6.5 Das Skript *DemoWriteHostColors.ps1* zeigt alle für das Cmdlet *Write-Host* verfügbaren Farben an

Nachdem Sie die Statusmeldung ausgegeben haben, durchlaufen Sie die Druckerobjekte mit einer *foreach*-Anweisung. Verwenden Sie *$printer* als Enumerator. Erstellen Sie einen Codeblock und verwenden Sie die Methode *addnew()* des *RecordSet*-Objekts, das Sie zuvor erstellt haben. Ermöglichen Sie mit der *item()*-Methode den Zugriff auf alle Felder in der Access-Tabelle, die in der Abfragezeichenfolge angegeben ist. Sie müssen die Datenquelle der WMI-Abfrage dem entsprechenden Feld in der Datenbanktabelle zuweisen. Dieser Codeabschnitt ist ziemlich lang und kann leicht durcheinander gebracht werden. Nachdem Sie alle aus WMI abgerufenen Eigenschaften mit den in der Datenbanktabelle definierten Feldern abgeglichen haben, übertragen Sie die Informationen mit der Methode *update()* zurück in die Access-Datenbank. Dieser Codeabschnitt ist in folgendem Listing dargestellt:

```
$objRecordSet.AddNew()
   $objRecordSet.Fields.item("TimeStamp") = Get-Date
   $objRecordSet.Fields.item("strComputer") = $strComputer
   $objRecordSet.Fields.item("strDomain") = $strDomain
   $objRecordSet.Fields.item("averagePagesPerMinute") =  $printer.averagePagesPerMinute
   $objRecordSet.Fields.item("caption") = $printer.caption
   $objRecordSet.Fields.item("default") = $printer.default
   $objRecordSet.Fields.item("comment") = $printer.comment
   $objRecordSet.Fields.item("averagePagesPerMinute") = $printer.averagePagesPerMinute
   $objRecordSet.Fields.item("description") = $printer.description
   $objRecordSet.Fields.item("deviceID") = $printer.deviceID
   $objRecordSet.Fields.item("direct") = $printer.direct
   $objRecordSet.Fields.item("doCompleteFirst") = $printer.doCompleteFirst
   $objRecordSet.Fields.item("driverName") = $printer.driverName
   $objRecordSet.Fields.item("enableBIDI") = $printer.enableBIDI
   $objRecordSet.Fields.item("enableDevQueryPrint") = $printer.enableDevQueryPrint
   $objRecordSet.Fields.item("extendedPrinterStatus") = $printer.extendedPrinterStatus
   $objRecordSet.Fields.item("hidden") = $printer.hidden
   $objRecordSet.Fields.item("horizontalresolution") = $printer.horizontalresolution
   $objRecordSet.Fields.item("verticalresolution") = $printer.verticalresolution
   $objRecordSet.Fields.item("local") = $printer.local
   $objRecordSet.Fields.item("keepprintedjobs") = $printer.keepprintedjobs
   $objRecordSet.Fields.item("network") = $printer.network
   $objRecordSet.Fields.item("printerstate") = $printer.printerstate
   $objRecordSet.Fields.item("printerstatus") = $printer.printerstatus
   $objRecordSet.Fields.item("printjobdatatype") = $printer.printjobdatatype
   $objRecordSet.Fields.item("printprocessor") = $printer.printprocessor
   $objRecordSet.Fields.item("priority") = $printer.priority
   $objRecordSet.Fields.item("published") = $printer.published
   $objRecordSet.Fields.item("queued") = $printer.queued
   $objRecordSet.Fields.item("spoolenabled") = $printer.spoolenabled
   $objRecordSet.Fields.item("systemname") = $printer.systemname
   $objRecordSet.Fields.item("workoffline") = $printer.workoffline
   $objRecordSet.Update()
```

Nachdem die Informationen in der Datenbank aktualisiert wurden, fahren Sie mit dem nächsten WMI-Objekt fort. Fügen Sie einen neuen Datensatz zur Datenbank hinzu und aktualisieren Sie die Informationen. Setzen Sie diesen Vorgang fort, bis das Ende der WMI-Informationen erreicht wird. Um den Status anzuzeigen, geben Sie mit dem Cmdlet **Write-Host** eine Statuszeile mit /\-Zeichen aus. Jedes /\ repräsentiert ein Druckerobjekt. Codezeile:

```
write-host -foregroundColor yellow "/\" -noNewLine
```

Nachdem die Datenbank aktualisiert wurde, müssen Sie die Objekte *Connection* und *RecordSet* schließen. Die erforderlichen Codezeilen lauten:

```
$objRecordSet.Close()
$objConnection.Close()
```

Das vollständige Skript *WritePrinterInfoToAccess.ps1* hat folgenden Aufbau:

WritePrinterInfoToAccess.ps1

```
$StrComputer = (New-Object -ComObject WScript.Network).computername
$StrDomain = (New-Object -ComObject WScript.Network).userDomain
$strWMIQuery = "Select * from win32_printer"
$objprinters = get-wmiobject -query $strWMIQuery

$adOpenStatic = 3
$adLockOptimistic = 3
$strDB = "c:\FSO\ConfigurationMaintenance.mdb"
$strTable = "printers"
$objConnection = New-Object -ComObject ADODB.Connection
$objRecordSet = new-object -ComObject ADODB.Recordset
$objConnection.Open("Provider = Microsoft.Jet.OLEDB.4.0; `
  Data Source= $strDB")
$objRecordSet.Open("SELECT * FROM $strTable", `
  $objConnection, $adOpenStatic, $adLockOptimistic)

write-host -foreGroundColor yellow "Ermitteln der Durcker auf ..."

foreach ($printer in $objprinters)
 {
   $objRecordSet.AddNew()
   $objRecordSet.Fields.item("TimeStamp") = Get-Date
   $objRecordSet.Fields.item("strComputer") = $strComputer
   $objRecordSet.Fields.item("strDomain") = $strDomain
   $objRecordSet.Fields.item("averagePagesPerMinute") = $printer.averagePagesPerMinute
   $objRecordSet.Fields.item("caption") = $printer.caption
   $objRecordSet.Fields.item("default") = $printer.default
   $objRecordSet.Fields.item("comment") = $printer.comment
   $objRecordSet.Fields.item("averagePagesPerMinute") = $printer.averagePagesPerMinute
   $objRecordSet.Fields.item("description") = $printer.description
   $objRecordSet.Fields.item("deviceID") = $printer.deviceID
   $objRecordSet.Fields.item("direct") = $printer.direct
   $objRecordSet.Fields.item("doCompleteFirst") = $printer.doCompleteFirst
   $objRecordSet.Fields.item("driverName") = $printer.driverName
   $objRecordSet.Fields.item("enableBIDI") = $printer.enableBIDI
   $objRecordSet.Fields.item("enableDevQueryPrint") = $printer.enableDevQueryPrint
   $objRecordSet.Fields.item("extendedPrinterStatus") = $printer.extendedPrinterStatus
   $objRecordSet.Fields.item("hidden") = $printer.hidden
   $objRecordSet.Fields.item("horizontalresolution") = $printer.horizontalresolution
   $objRecordSet.Fields.item("verticalresolution") = $printer.verticalresolution
   $objRecordSet.Fields.item("local") = $printer.local
   $objRecordSet.Fields.item("keepprintedjobs") = $printer.keepprintedjobs
   $objRecordSet.Fields.item("network") = $printer.network
   $objRecordSet.Fields.item("printerstate") = $printer.printerstate
   $objRecordSet.Fields.item("printerstatus") = $printer.printerstatus
   $objRecordSet.Fields.item("printjobdatatype") = $printer.printjobdatatype
   $objRecordSet.Fields.item("printprocessor") = $printer.printprocessor
```

```
   $objRecordSet.Fields.item("priority") = $printer.priority
   $objRecordSet.Fields.item("published") = $printer.published
   $objRecordSet.Fields.item("queued") = $printer.queued
   $objRecordSet.Fields.item("spoolenabled") = $printer.spoolenabled
   $objRecordSet.Fields.item("systemname") = $printer.systemname
   $objRecordSet.Fields.item("workoffline") = $printer.workoffline
   $objRecordSet.Update()
   write-host -foregroundColor yellow "/\" -noNewLine
 }

$objRecordSet.Close()
$objConnection.Close()
```

Überprüfen der Druckerports

Obwohl Druckerports unerlässlich sind, wissen viele Benutzer nicht, was Druckerports eigentlich sind. Allgemein wird angenommen, dass Druckerports mit IP-Adressen zu tun haben. Die falsche Konfiguration der Druckerports kann jedoch dazu führen, dass Druckaufträge im Cyberspace verschwinden.

In Abbildung 6.6 sind die auf einem Server konfigurierten Druckerports dargestellt.

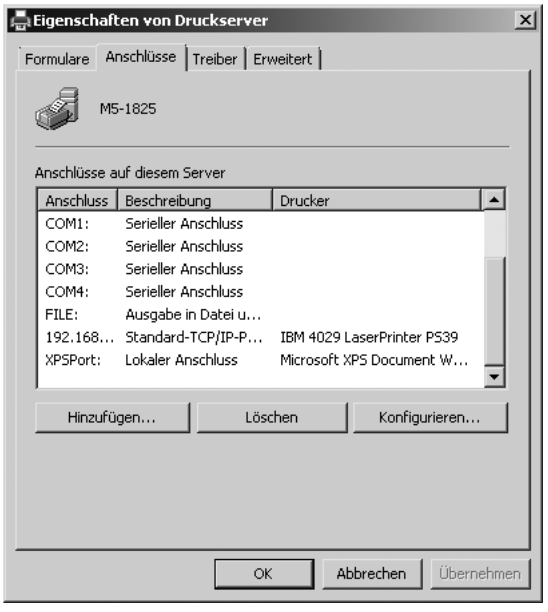


Abbildung 6.6 Beispiel konfigurierter Druckerports

Das Skript *ListPrinterPorts.ps1* definiert die beiden Variablen *$strComputer* und *$help*, um die Skriptausführung anhand von Befehlszeilenparametern zu steuern. Wenn keine Parameter angegeben werden, listet das Skript die auf dem lokalen Computer konfigurierten Druckerports auf. Sie können mit diesem Skript jedoch auch die Druckerports auf einem Remotecomputer abrufen.

Beginnen Sie das Skript *ListPrinterPorts.ps1* mit einer *param*-Anweisung und definieren Sie zwei Parameter. Der Parameter *$strComputer* ist auf den Standardwert *localhost* für den lokalen Computer festgelegt. Der zweite Parameter lautet *$help* und generiert die Hilfedatei.

Wichtig Beachten Sie bei der Angabe von Parametern mit der *param*-Anweisung, dass *param* die erste nicht kommentierte Zeile des Skripts sein muss.

Nachdem Sie die Argumente für das Skript definiert haben, erstellen Sie die Funktion *funhelp*. Diese Funktion wird aufgerufen, wenn das Skript mit dem Parameter *-help* ausgeführt wird. Obwohl in allen Beispielen des Parameters *-help* in einer Hilfedatei *-help ?* angegeben ist, kann das Fragezeichen durch einen beliebigen Wert ersetzt werden, da in der *if*-Anweisung nur überprüft wird, ob die Variable *$help* vorhanden ist:

```
if($help) { "Die Hilfeinformationen werden ausgegeben..." ; funHelp }
```

Der Skriptabschnitt für die WMI-Abfrage besteht aus nur drei Codezeilen. Die erste Zeile gibt die WMI-Klasse an, mit der die Abfrage ausgeführt wird. Dieses Skript verwendet die WMI-Klasse *Win32_TcpIpPrinterPort*. Die zweite Codezeile verwendet das Cmdlet **Get-WmiObject**, um die WMI-Informationen zu den Druckerports auf dem Computer abzurufen, der in der Variablen *$strComputer* angegeben ist. Die zurückgegebenen WMI-Verwaltungsobjekte werden bereinigt, um alle nicht alphabetischen Zeichen zu entfernen. Dieser Codeabschnitt ist in folgendem Listing dargestellt:

```
$class = "Win32_TcpIpPrinterPort"
Get-WmiObject -Class $class -computername $strcomputer |
format-list [a-z]*
```

Das vollständige Skript *ListPrinterPorts.ps1* hat folgenden Aufbau:

ListPrinterPorts.ps1

```
param($strComputer="localhost", $help)

function funHelp()
{
$helpText="
NAME: ListPrinterPorts.ps1
Erstellt eine Liste der Druckerports auf der lokalen Arbeitsstation oder einem Remotecomputer.

PARAMETER:
-computerName Gibt den Namen des Computers an, der mit diesem Skript überprüft werden soll.
-help         Druckt die Hilfedatei aus.

SYNTAX:
ListPrinterPorts.ps1 -comptuerName MunichServer
Erstellt eine Liste der Druckerports auf einem Computer namens MunichServer.

FindPrinterPorts.ps1 -help ?
Gibt die Hilfeinformationen aus, die der Variablen $helpText zugewiesen wurden.

"
$helpText
exit
}

if($help) { "Die Hilfeinformationen werden ausgegeben..." ; funHelp }

$class = "Win32_TcpIpPrinterPort"
Get-WmiObject -Class $class -computername $strcomputer |
format-list [a-z]*
```

Erstellen einer Hilfefunktion für das Skript

Um die Benutzerfreundlichkeit Ihrer Skripts sicherzustellen, sollten Sie eine Hilfefunktion implementieren. Das Skript *ListPrinterPorts.ps1* zeigt ein Beispiel einer Hilfefunktion. Zum Erstellen von Hilfefunktionen gibt es mehrere Ansätze. Beispielsweise definiert das Skript *ListPrinterPorts.ps1* den Parameter *$help*. Die Hilfefunktion wird angezeigt, wenn das Skript mit dem Parameter **-help ?** ausgeführt wird. Sie sollten mindestens drei Features in die Hilfefunktion einbeziehen: Eine Beschreibung, die Parameter und ein Syntaxbeispiel. Erstellen Sie zu diesem Zweck eine Vorlage, um die Syntax zu standardisieren und das Erstellen der Hilfeinformationen zu vereinfachen.

Eine zweite Methode zum Erstellen einer Hilfefunktion besteht im Angeben eines nicht benannten Arguments und der Verwendung einer *switch*-Anweisung, um den Wert von *$args* auszuwerten. Diese Methode ist anhand eines Beispiels im Skript *ArgsShare.ps1* veranschaulicht. In Abbildung 6.7 ist die Ausgabe des Skripts *ArgsShare.ps1* dargestellt.

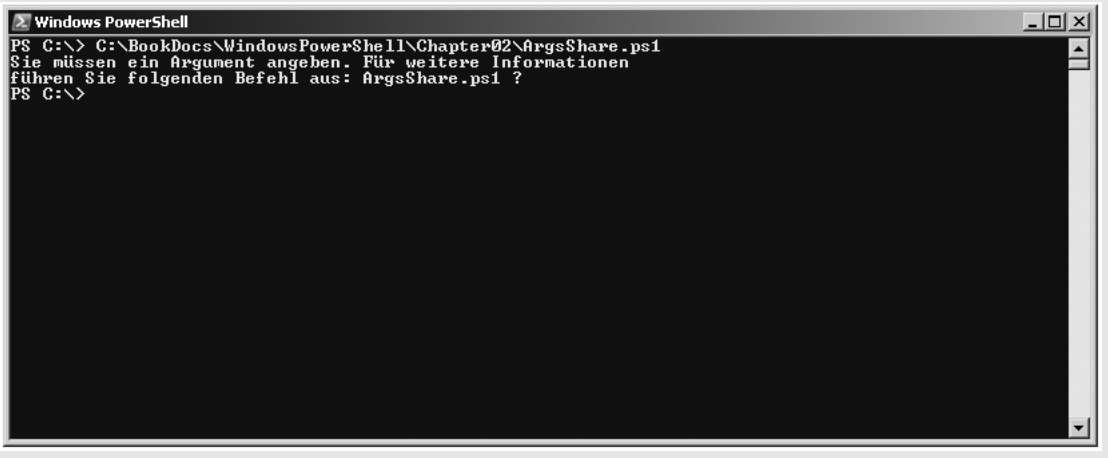


Abbildung 6.7 Mit einer Hilfemeldung bezüglich fehlender Argumente wird die Verwendung des Skripts vereinfacht

Eine weitere Methode zum Implementieren der Hilfe ist, diese in Verbindung mit einem fehlenden Parameter aufzurufen. Wenn das Skript beispielsweise jeweils einen Parameter für einen Freigabenamen und einen Pfad erfordert, können Sie keine Freigabe erstellen, ohne einen Namen und den Pfad anzugeben. Wenn einer der Parameter fehlt, können Sie eine entsprechende Hilfemeldung anzeigen. Diese Methode ist anhand eines Beispiels im Skript *CreateShare.ps1* veranschaulicht.

Unabhängig von der Methode, mit der Sie die Hilfe in Ihren Skripts implementieren, ist die Konsistenz der Parameter für die Lesbarkeit und Verwendbarkeit der Skripts wichtig. Wenn ein Skript nur für einen bestimmten Zweck verwendet wird, müssen Sie keinen Hilfetext erstellen. Es ist jedoch sinnvoll sowohl Hilfetext als auch entsprechende Kommentare zum Skript hinzuzufügen, wenn dieses von Help Desk-Mitarbeitern oder Administratoren verwendet wird. Für ein Skript, das mehrere Parameter umfasst, sollten Sie die erste und dritte beschriebene Methode verwenden.

Da Druckerserver oft mehrfach vernetzt sind und Druckerports in mehreren Netzwerken verwenden, müssen Sie in der Lage sein, ausschließlich die in einem bestimmten Netzwerk konfigurierten Druckerports abzurufen. Dies ist sowohl für die Netzwerkverwaltung als auch für die Problembehandlung nützlich.

Ändern Sie das Skript *ListPrinterPorts.ps1*, um das Skript *FindPrinterPorts.ps1* zu erstellen, das einen Befehlszeilenparameter namens **-network** unterstützt. Der Parameter **-network** gibt die Netzwerk-ID des Druckerports an. Legen Sie den Parameter auf den Standardwert 192.168 (eine interne Netzwerkadresse) fest. Der Wert kann im Skript geändert oder über die Befehlszeile überschrieben werden, indem das Skript mit dem Parameter **-network** ausgeführt wird.

Der Parameter **-help** führt die gleichen Vorgänge wie im Skript *ListPrinterPorts.ps1* aus.

Um nur die Druckerports für die im Parameter **-network** angegebene Netzwerkadresse anzuzeigen, geben Sie im Cmdlet **Where-Object** einen regulären Ausdruck an und suchen Sie nach der Netzwerkadresse. Filtern Sie die Ergebnisse mit dem Parameter **-match**.

Nachdem Sie die lokalen Drucker abgerufen haben, geben Sie mit dem Cmdlet **Write-Host** eine Statusmeldung aus. Rufen Sie mit dem Cmdlet **Get-WmiObject** die Instanzen der Klasse *Win32_TcpIpPrinterPort* auf dem angegebenen Computer ab und fügen Sie die Ergebnisse in das Cmdlet **Where-Object** ein. Der Codeblock für **Where-Object** verwendet die automatische Variable *$_*, die das aktuelle Objekt in der Pipeline repräsentiert und die für den Parameter **-network** angegebene Zeichenfolge mit einem regulären Ausdruck sucht. Dieser Codeabschnitt ist in folgendem Listing dargestellt:

```
Write-Host -foregroundColor Yellow "Die folgenden Druckerports wurden im Netzwerkbereich $network ermittelt:`n"
Get-WmiObject -class $class -computername $strcomputer |
Where-object { $_.name -match $network }
```

Hinweis Sie können die Informationen zu Druckgeräten auch über SNMP (Simple Network Management Protocol) abrufen. SNMP ist ein Industriestandard, der auf dem Senden von Nachrichten an zentralisierte Systeme basiert. Das Kennwort in diesen Systemen wird als *Communityzeichenfolge* bezeichnet. In vielen Netzwerken entspricht SNMP nicht den Sicherheitsstandards, da die Nachrichten im Klartext übermittelt werden.

Wenn der Drucker für SNMP konfiguriert ist und SNMP-Meldungen unterstützt, ist das SNMP-Protokoll aktiviert. Überprüfen Sie den Wert von *SNMPEnabled* und geben Sie die Informationen auf dem entsprechenden Gerät aus, damit die Ausgabe einfacher zu lesen ist. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
if($($_.SNMPEnabled))
 {
  Write-Host -foregroundColor yellow "`tFolgende Drucker sind für SNMP aktiviert:`n"
  Write-Host "`t$($_.name), $($_.portNumber), $($_.SNMPCommunity, $($_.SNMPDevIndex)`n"
 }
 ELSE
 {
  Write-Host -foregroundColor yellow "`tFolgende Drucker sind nicht für SNMP aktiviert:`n"
  write-host "`t$($_.name), $($_.portNumber)"
 }
}
```

Tipp Bei der Auswertung von *SNMPEnabled* im *if … else*-Codeblock musste ich feststellen, dass die Eigenschaft nicht richtig erweitert wurde. Wenn der Wert direkt ausgegeben wurde, konnte dieser allerdings korrekt angezeigt werden. Um den Wert zu überprüfen, bevor dieser ausgegeben wird, können Sie ein zusätzliches Dollarzeichen einfügen: *$($_.SNMPEnabled).* Verwenden Sie diese Methode, wenn der Wert nicht wie erwartet erweitert wird.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Nachdem Sie die geschweiften Klammern geschlossen und die Informationen über den entsprechenden Druckerport ausgegeben haben, kann das Skript beendet werden. Das vollständige Skript *FindPrinterPorts.ps1* hat folgenden Aufbau:

FindPrinterPorts.ps1

```
param( $strcomputer="localhost", $network="192.168", $help)
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: FindPrinterPorts.ps1
Ermöglicht die Verwaltung von Durckerports auf der lokalen Arbeitsstation oder einem Remotecomputer.

PARAMETER:
-computerName Gibt den Namen des Computers an, der mit diesem Skript überprüft werden soll.
-help         Druckt die Hilfedatei aus.
-network      Eine IP-Adresse mit einem, zwei, der drei Oktetten.


SYNTAX:
FindPrinterPorts.ps1 -comptuerName MunichServer
  Erstellt eine Liste der Druckerports auf einem Computer namens MunichServer.

FindPrinterPorts.ps1 -help ?
  Gibt die Hilfeinformationen zu diesem Skript aus.

FindPrinterPorts.ps1 -computername MunichServer -network "10"
 Verwendet für das Netzwerk eine Adresse der Klasse A aus dem Bereich 10. Nur die
 Druckerports des Remoteservers MunichServer aus dem Bereich 10.x.x.x werden ausgegeben.

FindPrinterPorts.ps1
 Zeigt die Druckerports aus dem Netzwerkbereich 192.168.x.x auf dem lokalen Computer an.

"@
$helpText
exit
}

if($help) { "Die Hilfeinformationen werden ausgegeben..." ; funHelp }
$class = "Win32_TcpIpPrinterPort"

Write-Host -foregroundColor Yellow "Die folgenden Druckerports wurden im Netzwerkbereich $network ermittelt:`n"
Get-WmiObject -class $class -computername $strcomputer |
Where-object { $_.name -match $network } | foreach($_){

 if($($_.SNMPEnabled))
 {
  Write-Host -foregroundColor yellow "`tFolgende Drucker sind für SNMP aktiviert:`n"
  Write-Host "`t$($_.name), $($_.portNumber), $($_.SNMPCommunity, $($_.SNMPDevIndex)`n"
 }
 ELSE
 {
  Write-Host -foregroundColor yellow "`tFolgende Drucker sind für SNMP nicht aktiviert:`n"
  write-host "`t$($_.name), $($_.portNumber)"
 }
}
```

Identifizieren der Druckertreiber

Zur Netzwerkverwaltung gehört auch die Arbeit mit Druckertreibern. In der Tat ist das Überprüfen von Druckertreibern eine unerlässliche Aufgabe. Auf einem Windows Vista-System sind zahlreiche Standarddruckertreiber installiert. Das Hinzufügen eines Standarddruckertreibers für einen neuen Drucker ist unkompliziert. Wenn der Treiber jedoch nicht standardmäßig vorhanden ist, ist diese Aufgabe etwas schwieriger.

Rufen Sie beispielsweise mit einem Skript names *FindPrinterDrivers.ps1* mit dem Cmdlet **Get-ChildItem** die auf dem System installierten .inf-Dateien ab, deren Namen die Buchstaben *prn* enthalten. Ermitteln Sie hierfür unter Verwendung von *env:\ psdrive* den Wert der Umgebungsvariablen *%SYSTEMROOT%*. Geben Sie den Parameter **-exclude** an, um Dateien mit der Erweiterung .pnf auszuschließen.

In Abbildung 6.8 ist das inf-Verzeichnis auf einem Windows Server 2008-Computer dargestellt. Die zahlreichen Dateien in diesem Verzeichnis sind der Grund, warum Sie den Parameter **-exclude** im Skript angeben sollten.

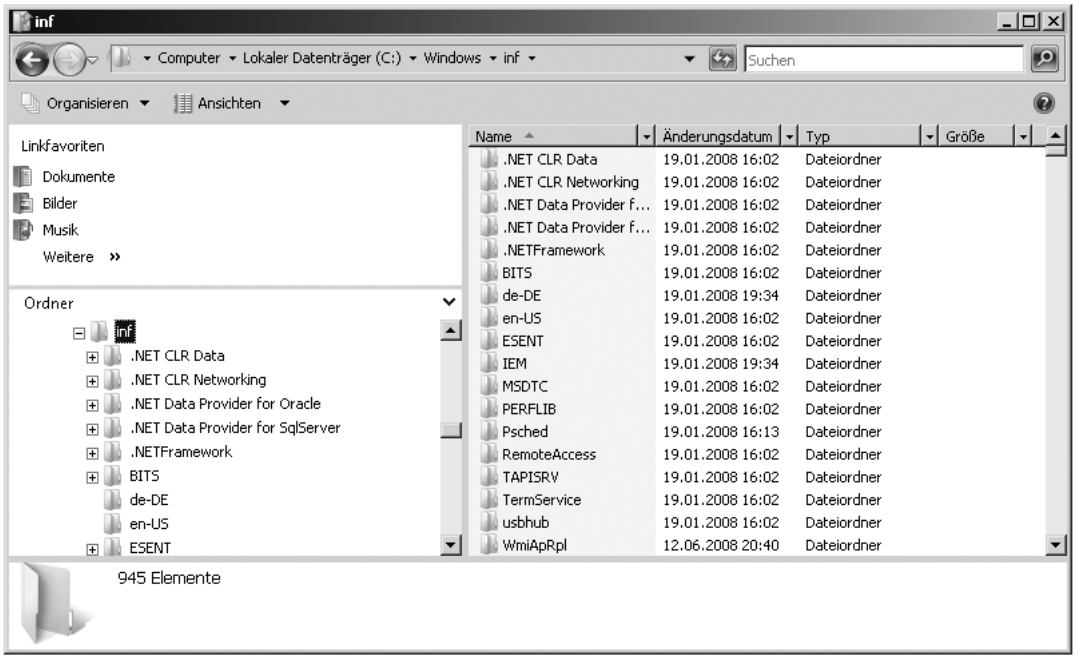


Abbildung 6.8 Das inf-Verzeichnis auf einem Windows Vista- oder Windows Server 2008-Computer enthält Treiberinformationen

Nachdem die Informationen abgerufen wurden, fügen Sie die Ergebnisse in das Cmdlet **Where-Object** ein und suchen Sie mit einem regulären Ausdruck nach den Buchstaben *prn* in Dateinamen. Die zurückgegebenen Dateien werden mit dem Cmdlet **Sort-Object** sortiert und in das Cmdlet **Format-Table** eingefügt. Wählen Sie die Eigenschaften *Name, Length, Creation Time* und *LastWriteTime* aus. Das vollständige Skript *FindPrinterDrivers.ps1* lautet:

FindPrinterDrivers.ps1

```
Get-ChildItem ((Get-Item Env:\systemroot).value+"\inf") -Exclude *.pnf |
Where-Object { $_.name -match "prn" } |
Sort-Object -Property name |
format-table -Property name, length, creationTime, lastWriteTime
```

Mit dem Skript *FindPrinterDrivers.ps1* können Sie die .inf-Dateien für Druckertreiber auflisten, aber nicht feststellen, welche Druckertreiber verfügbar sind. Das Skript *ReportAvailableDrivers.ps1* analysiert alle .inf-Dateien für Druckertreiber und sucht nach bestimmten Druckermodellen. Sie können die Anzahl der jeweiligen Druckertreibertypen anzeigen, die auf dem System verfügbar sind.

Das Skript *ReportAvailableDrivers.ps1* initialisiert als Erstes sieben Variablen und legt den ursprünglichen Wert dieser Variablen auf 0 fest. In den Variablen wird die aktuelle Anzahl der verfügbaren Druckertreiber gespeichert, die das Skript ausgibt. Geben Sie im Cmdlet **Get-ChildItem** den Pfad zum inf-Verzeichnis an. Wenn der Name einer .inf-Datei die Buchstaben *prn* enthält, öffnen Sie die Datei und suchen Sie mit einem regulären Ausdruck in einem *switch*-Block nach den gewünschten Druckertypen. Erhöhen Sie den Wert der Variablen entsprechend der gefundenen Übereinstimmung und wiederholen Sie den Vorgang für die nächste .inf-Datei.

Nachdem Sie alle .inf-Dateien durchlaufen haben, geben Sie die Ergebnisse mit dem Cmdlet **Write-Host** aus. Das vollständige Skript *ReportAvailableDrivers.ps1* hat folgenden Aufbau:

ReportAvailableDrivers.ps1

```
$hp=$ibm=$lexmark=$star=$text=$ps=$generic=0
Get-ChildItem ((Get-Item Env:\systemroot).value+"\inf") -Exclude *.pnf |
Where-Object { $_.name -match "prn" } |
foreach-object($_){
  switch -regex -file $_.fullname
   {
    'hp'      { $hp++ }
    'ibm'     { $ibm++ }
    'lexmark' { $lexmark++ }
    'star'    { $star++ }
    'text'    { $text++ }
    'ps'      { $ps++ }
    'generic' { $generic++ }
   }

}
"
Die folgenden Details sind zu den Druckertreibern auf dem Computer gegenwärtig verfügbar:
 HP-Treiber:      $hp
 IBM-Treiber:     $ibm
 Lexmark-Treiber: $lexmark
 Star-Treiber:    $Star
 Text-Treiber:    $text
 PS-Treiber:      $ps
 Allgemeine Treiber: $generic
"
```

Installieren von Druckertreibern

Nach der Installation von Windows Vista oder Windows Server 2008 müssen die Drucker konfiguriert werden. In Windows wird der physische Drucker als *Druckgerät* und die Druckerwarteschlange auf dem Computer als *Drucker* bezeichnet. Der Druckertreiber stellt sicher, dass die Informationen auf dem Bildschirm mit dem Ausdruck auf dem Papier identisch sind. Dieser Vorgang wird als WYSIWYG (What You See Is What You Get) bezeichnet. Ein ordnungsgemäß funktionierender Druckertreiber verhindert nicht nur einen BSOD (Blue Screen Of Death), sondern ist für Microsoft Office-Anwendungen für die korrekte Formatierung auf dem Bildschirm erforderlich. Ohne korrekten Druckertreiber können Microsoft Word und Microsoft Excel den Text möglicherweise nicht wie erforderlich umbrechen.

Installieren von Druckertreibern, die auf dem Computer vorhanden sind

Windows Vista und Windows Server 2008 umfassen zahlreiche Hardwaretreiber unter anderem für Drucker. Die Druckertreiber sind im Verzeichnis *%SYSTEMROOT%\inf* gespeichert. Druckertreiberdateien haben die Erweiterung .inf und die Dateinamen enthalten die Buchstaben *prn*. Mit diesen Informationen sind Sie in der Lage die .inf-Dateien für die Druckertreiber mit dem Skript *FindPrinterDrivers.ps1* abzurufen. Nachdem Sie alle Informationen zusammengestellt haben, können Sie die Druckertreiber auf Windows Vista oder Windows Server 2008 installieren.

Der Installationsprozess für Druckertreiber bietet den Vorteil, dass ein Benutzer einen Drucker zu seinem Profil hinzufügen kann, auch wenn dieser nicht über die Berechtigung *seLoadDriverPrivilege* verfügt. In Abbildung 6.9 ist die Auswahl des vorhandenen Treibers dargestellt.

Das Skript *InstallPrinterDriver.ps1* fügt einen Druckertreiber zu einem Benutzerprofil hinzu, das sich bereits auf dem Windows-Computer befindet. Der Treiber ist im Laufwerkscache gespeichert, aber noch nicht geladen. Um den Treiber zu laden, müssen Sie ein Verbindung mit der WMI-Klasse *Win32_PrinterDriver* herstellen. Verwenden Sie hierzu den *[wmiclass]*-Accelerator. Codezeile:

```
$objWMI = [wmiclass]"Win32_PrinterDriver"
```

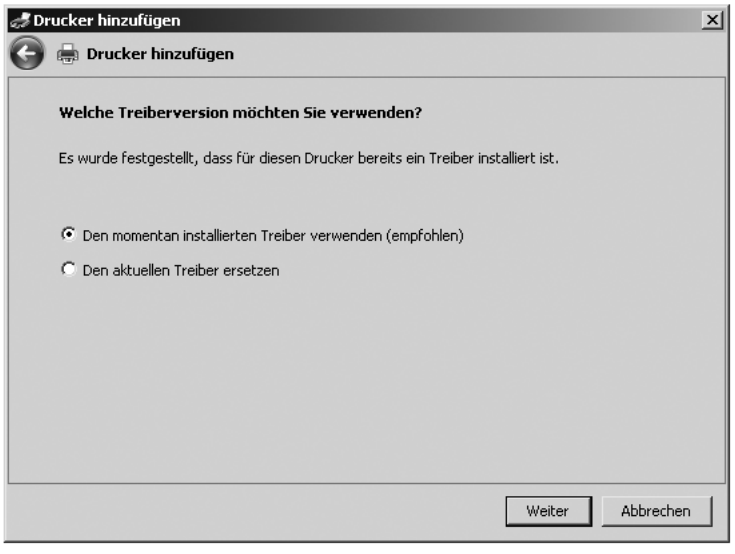


Abbildung 6.9 Windows Vista und Windows Server 2008 erkennen, ob ein Treiber bereits installiert ist

Wenn in der Variablen *$objWMI* eine Instanz des *System.Management.ManagementClass*-Objekts gespeichert ist, erstellen Sie mit der Methode *CreateInstance()* eine neue Instanz der Klasse *Win32_PrinterDriver*. Diese Instanz gibt die Informationen für das *ManagementClass*-Objekt in der Variablen *$objWMI* an, wenn Sie die Methode *AddPrinterDriver()* aufrufen.

Hinweis Das Verhalten von WMI ist manchmal ungewöhnlich. Um die Methode *AddPrinterDriver()* aus WMI zu verwenden, müssen Sie zuerst eine neue Instanz der WMI-Klasse *Win32_PrinterDriver* erstellen, da die Methode *AddPrinterDriver()* erfordert, dass der Druckertreiber als Instanz der Klasse *Win32_PrinterDriver* übergeben wird. Sie müssen hierzu *Win32_PrinterDriver* verwenden, um eine neue Instanz der Klasse *Win32_PrinterDriver* zu erstellen, bevor Sie die Methode *AddPrinterDriver()* der Klasse *Win32_PrinterDriver* aufrufen können.

Nachdem Sie das *ManagementClass*-Objekt der Variablen *$objWMI* zugewiesen haben, können Sie mit der Methode *CreateInstance()* eine leere Instanz der Klasse *Win32_PrinterDriver* erstellen. Die neue Instanz der Klasse *Win32_PrinterDriver* wird von folgender Codezeile erstellt:

```
$objDriver=$objWMI.CreateInstance()
```

Eine leere Instanz der Klasse ermöglicht Ihnen gegebenenfalls Werte für alle Eigenschaften der WMI-Klasse festzulegen. Im Skript *InstallPrinterDriver.ps1* müssen Sie nur den Namen des Drucker-treibers angeben, da die lokal installierten .inf-Dateien auf alle anderen erforderlichen Daten verweisen. Das Zuweisen des Treibernamens zur Eigenschaft *Name* ist einfach:

```
$objDriver.name = "Generic / Text Only"
```

Nachdem Sie alle zum Erstellen des Druckertreibers erforderlichen Werte angegeben haben, rufen Sie die Methode *AddPrinterDriver()* auf. Diese Methode verwendet ein Objekt, das eine Instanz der Klasse *Win32_PrinterDriver* sein muss. Im Skript *InstallPrinterDriver.ps1* verweist die Variable *$objDriver* auf dieses Objekt. Codezeile:

```
$rtnCode = $objwmi.addPrinterDriver($objDriver)
```

Um zu überprüfen, ob die Methode *AddPrinterDriver()* erfolgreich aufgerufen wurde, geben Sie den Rückgabecode aus. Das Skript *InstallPrinterDriver.ps1* speichert das *error*-Objekt in der Variablen *$rtnCode*. Das *error*-Objekt umfasst die Eigenschaft *ReturnValue*. Der Wert 0 zeigt an, dass der Befehl erfolgreich ausgeführt wurde. Die Eigenschaft *ReturnValue* des *error*-Objekts wird von folgender Codezeile ausgegeben:

```
$rtncode.returnValue
```

Das vollständige Skript *InstallPrinterDriver.ps1* umfasst folgende Anweisungen:

InstallPrinterDriver.ps1

```
$objWMI = [wmiclass]"Win32_PrinterDriver"
$objDriver=$objWMI.CreateInstance()

$objDriver.name = "Generic / Text Only"
$rtnCode = $objwmi.addPrinterDriver($objDriver)
$rtncode.returnValue
```

Installieren von Druckertreibern, die nicht auf dem Computer vorhanden sind

Die Installation eines Druckertreibers, der nicht im System vorhanden ist, ist etwas komplizierter. In Tabelle 6.2 sind die für die Klasse *Win32_PrinterDriver* definierten Eigenschaften aufgeführt. Für einen Druckertreiber müssen nicht alle Eigenschaften festgelegt, aber andere Aufgaben ausgeführt werden.

Tabelle 6.2 *Win32_PrinterDriver*-Eigenschaften

Eigenschaft	Definition
Caption	System.String Caption {get;set;}
ConfigFile	System.String ConfigFile {get;set;}
CreationClassName	System.String CreationClassName {get;set;}
DataFile	System.String DataFile {get;set;}
DefaultDataType	System.String DefaultDataType {get;set;}
DependentFiles	System.String DependentFiles {get;set;}
Description	System.String Description {get;set;}
DriverPath	System.String DriverPath {get;set;}
FilePath	System.String FilePath {get;set;}
HelpFile	System.String HelpFile {get;set;}
InfName	System.String InfName {get;set;}
InstallDate	System.String InstallDate {get;set;}
MonitorName	System.String MonitorName {get;set;}
Name	System.String Name {get;set;}
OEMUrl	System.String OEMUrl {get;set;}
Started	System.Boolean Started {get;set;}
StartMode	System.String StartMode {get;set;}
Status	System.String Status {get;set;}
SupportedPlatform	System.String SupportedPlatform {get;set;}
SystemCreationClassName	System.String SystemCreationClassName {get;set;}
SystemName	System.String SystemName {get;set;}
Version	System.UInt16 Version {get;set;}

Über das Applet **Drucker** in der Windows Vista- oder Windows Server 2008-Systemsteuerung können Sie die Treibereigenschaften anzeigen. In Abbildung 6.10 sind die Druckereigenschaften dargestellt.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

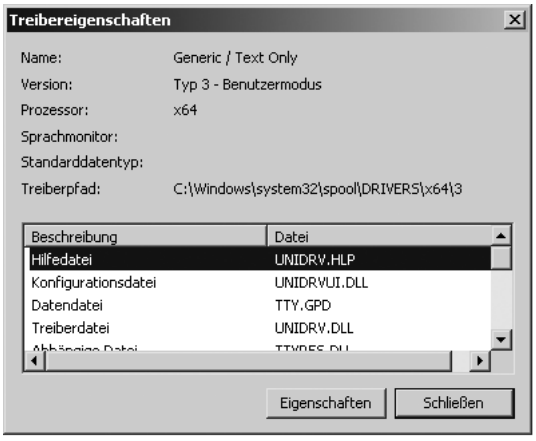


Abbildung 6.10 Eigenschaften der Druckertreiber

Um das Skript *InstallPrinterDriverFull.ps1* auszuführen, verwenden Sie den *[wmiclass]*-Accelerator und geben Sie die WMI-Klasse *Win32_PrinterDriver* an, um ein Verwaltungsobjekt zurückzugeben, das die Verwendung der Methoden der Klasse *Win32_PrinterDriver* ermöglicht. Speichern Sie dieses Objekt in der Variablen *$objWMI*.

Das *System.Management.ManagementObject* für *Win32_PrinterDriver* enthält die Methode *CreateInstance()*. Nachdem Sie eine neue Instanz der Klasse *Win32_PrinterDriver* erstellt haben, weisen Sie das Objekt der Variablen *$objDriver* zu. Die neue Instanz der Klasse *Win32_PrinterDriver* wird mit der Methode *AddPrinterDriver()* des ursprünglichen Verwaltungsobjekts verwendet, das in der Variablen *$objWMI* gespeichert ist.

Die neue Instanz der Druckertreiberklasse wird der Variablen *$objDriver* zugewiesen, die alle Eigenschaftenwerte erfordert. Geben Sie für die neue Instanz den Pfad zu den Dateien an. Das Skript *InstallPrinterDriverFull.ps1* verwendet nur die Basiseigenschaften. Das vollständige Skript *InstallPrinterDriverFull.ps1* hat folgenden Aufbau:

InstallPrinterDriverFull.ps1

```
$objWMI = [wmiclass]"Win32_PrinterDriver"
$objDriver=$objWMI.CreateInstance()

$objDriver.name = "Generic / Text Only"
$objDriver.DriverPath = "C:\WINDOWS\System32\spool\DRIVERS\W32X86\3\UNIDRV.DLL"
$objDriver.ConfigFile = "C:\WINDOWS\System32\spool\DRIVERS\W32X86\3\UNIDRVUI.DLL"
$objDriver.DataFile = "C:\WINDOWS\System32\spool\DRIVERS\W32X86\3\TTY.GPD"
$objDriver.DependentFiles ="C:\WINDOWS\System32\spool\DRIVERS\W32X86\3\TTYRES.DLL", `
"C:\WINDOWS\System32\spool\DRIVERS\W32X86\3\TTY.INI", `
"C:\WINDOWS\System32\spool\DRIVERS\W32X86\3\TTY.DLL", `
"C:\WINDOWS\System32\spool\DRIVERS\W32X86\3\TTYUI.DLL", `
"C:\WINDOWS\System32\spool\DRIVERS\W32X86\3\UNIRES.DLL", `
"C:\WINDOWS\System32\spool\DRIVERS\W32X86\3\TTYUI.HLP", `
"C:\WINDOWS\System32\spool\DRIVERS\W32X86\3\STDNAMES.GPD"
$objDriver.HelpFile = "C:\WINDOWS\System32\spool\DRIVERS\W32X86\3\UNIDRV.HLP"

$rtnCode = $objwmi.addPrinterDriver($objDriver)
$rtncode.returnValue
```

Zusammenfassung

In diesem Kapitel wurden die mit Druckertreibern verbundenen Aufgaben erklärt. Die Arbeit mit Druckern beginnt mit den Druckertreibern. Sie können nicht drucken, wenn der Treiber für den Drucker nicht installiert ist. Druckertreiber sind entweder in Windows Vista und Windows Server 2008 integriert oder über die Website des Hardwareherstellers bzw. auf einer CD verfügbar, die mit dem Druckgerät geliefert wurde. Der Treiber muss unter Windows installiert werden. Nach der Bereitstellung der Druckertreiber wurde das Freigeben von Druckgeräten erklärt. Außerdem wurde beschrieben, wie Sie die vorhandenen Einstellungen sowie die Eigenschaften von Druckertreibern konfigurieren können.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

KAPITEL 7

Desktopverwaltung

Nach Abschluss dieses Kapitels können Sie:

- Laufwerkkonfigurationen überprüfen
- Datenträgerinformationen in einer Microsoft Access-Datenbank erfassen
- Die logischen Laufwerkkonfigurationen überprüfen
- Die Speicherplatzbelegung überwachen
- Die Leistungsindikatorklassen verwenden

Auf der Begleit-CD Alle Skripts in diesem Kapitel befinden sich auf der Begleit-CD im Ordner *\Scripts\Chapter07*.

Verwalten der Desktopcomputer

Windows Vista und Windows Server 2008 sind ausgesprochen zuverlässig, selbstregulierend und beinahe wartungsfrei. Die Standardeinstellungen erfüllen jedoch in Umgebungen großer Unternehmen möglicherweise nicht alle Leistungs- und Sicherheitsanforderungen. Sie müssen die Systemleistung überwachen und gegebenenfalls bestimmte Einstellungen ändern oder anpassen.

Überprüfen der Laufwerke

Überprüfen Sie als Erstes die Laufwerkkonfiguration Ihres Windows Vista- oder Windows Server 2008-Systems. Ein Benutzer kann möglicherweise nicht erkennen, dass nur ein oder zwei Laufwerke ver-fügbar sind, da Windows das physische Layout der Laufwerke von den tatsächlich vorhandenen Laufwerken ableitet.

Viele Hardwarehersteller erstellen aus verschiedenen Gründen „verborgene“ Partitionen. Beispielsweise kann ein Ersatzlaufwerk eine Partition in der gleichen Größe wie das ursprüngliche Laufwerk umfassen, da die Datenträgerkapazität ausreichend ist. Unter Verwendung von Windows PowerShell und WMI können Sie die Laufwerkkonfiguration eines Computers ermitteln und dokumentieren. Die Standardansicht im Windows Explorer von Windows Vista und Windows Server 2008 zeigt keine Partitionsinformationen an (siehe Abbildung 7.1).

Das Skript *ReportDiskDriveConfiguration.ps1* ruft alle Eigenschaften eines physichen Datenträgers über WMI ab. Verwenden Sie das Cmdlet **Get-WmiObject** und die WMI-Klasse *Win32_DiskDrive*, um diese Informationen anzuzeigen. Das Skript akzeptiert ein Argument, bei dem es sich um den Namen eines Computers, von dem die Konfigurationsinformationen abgerufen werden sollen, oder um ein Fragezeichen (?) handeln kann, um die Hilfe anzuzeigen.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

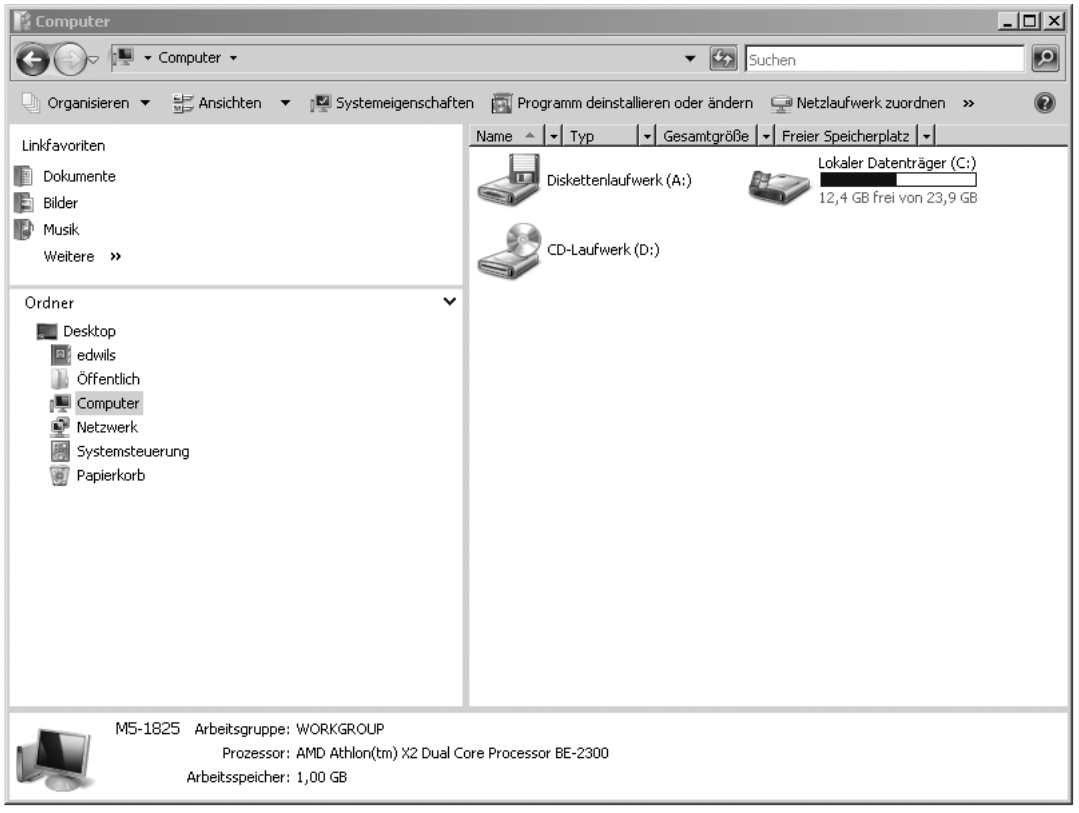


Abbildung 7.1 Windows Vista-Standardansicht der Laufwerke in Windows Explorer

Verwenden der Standardargumente

Wenn Sie ein Skript mit einem Befehlszeilenargument ausführen, wird die automatische Variable *$args* erstellt. Um zu überprüfen, ob ein Argument angegeben wurde, suchen Sie nach der Variablen *$args*. Wurde kein Befehlszeilenargument angegeben, können Sie einen der folgenden Schritte ausführen:

- Sie können *$args* einen Wert zuweisen, um eine Standardaktion auszuführen.
- Sie können eine Hilfemeldung anzeigen und das Skript beenden.
- Sie können zur Eingabe des Werts auffordern.

Im Folgenden werden diese Optionen näher beschrieben. Beispielsweise können Sie *$args* wie folgt einen Wert zuweisen, um eine Standardaktion auszuführen:

```
if(!$args) { $args = "Meine Standardaktion" }
```

Sie müssen bei Verwendung dieser Methode sicherstellen, dass die meisten Benutzer diese Aktion ausführen möchten. Außerdem müssen Sie den Benutzern mitteilen, dass die Standardaktion ausgeführt wird. Sie können vorschlagen, dass der Benutzer die Hilfe für weitere Optionen anzeigt.

Die zweite Methode betrifft das Anzeigen einer Hilfemeldung und das Beenden des Skripts. Codebeispiel:

```
if(!$args) { "Dieses Skript erfordert ein Argument. Zeigen Sie weitere Informationen mit folgendem Befehl
an …" ; exit }
```

Beachten Sie, dass das Anzeigen der Hilfe und Beenden des Skripts die Nützlichkeit des Skripts einschränkt. Sie können den Benutzer auffordern, einen Wert in der Befehlszeile einzugeben, um das Skript auszuführen.

Bei der dritten Methode wird der Benutzer zur Eingabe eines Werts aufgefordert:

```
if(!$args) { $args = Read-Host -Prompt "Bitte geben Sie den fehlenden Parameter an." }
```

Ein Problem mit dieser Methode ist, dass das Skript möglicherweise fehlschlägt, wenn der Benutzer keinen Wert eingibt.

Überprüfen Sie, ob die Variable *$args* vorhanden ist. Ohne diese Variable führt das Skript die Standardaktion aus und fragt den lokalen Computer ab. Geben Sie in diesem Fall ein Ausrufezeichen (*not*-Operator) vor der Variablen *$args* ein. Die *if*-Anweisung überprüft, ob *$args* vorhanden ist. Codezeile:

```
if(!$args)
```

Beginnen Sie einen neuen Codeblock und geben Sie mit dem Cmdlet **Write-Host** eine Meldung aus, dass der lokale Computer abgefragt wird. Legen Sie den Wert von *$args* auf *localhost* fest und geben Sie die Meldung in Grün aus:

```
{
  Write-Host -foregroundcolor green `
  'Abfragen von localhost ...'
  $args = 'localhost'
 }
```

Wenn der Wert der Variablen *$args* ein Fragezeichen (?) ist, geben Sie eine Hilfemeldung aus. Die Hilfemeldung sollte den Namen des Skripts, eine Beschreibung und Syntaxbeispiele umfassen:

```
if($args -eq "?")
 { "
    ReportDiskDriveConfiguration.ps1

   BESCHREIBUNG:
   Dieses Skript unterstützt als einziges Argument einen Computernamen.
   Das Skript zeigt entweder die Laufwerkskonfiguration
   des lokalen Computers oder eines Remotecomputers an.
   Sie können ein Fragezeichen (?) oder den Namen eines Computers angeben.

   BEISPIEL:
   ReportDiskDriveConfiguration.ps1 remoteComputerName

   Ermittelt die Laufwerkskonfiguration auf einem  Computer namens remoteComputerName.

   Das Skript kann außerdem diese Hilfeinformationen anzeigen.
   Dies wird mit dem Fragezeichen (?) als Parameter wie in folgendem Beispiel angegeben.
   ReportDiskDriveConfiguration.ps1 ?
   "
 }
```

Die WMI-Standardabfrage verwendet das Cmdlet **Get-WmiObject**. Dieses Cmdlet verwendet sowohl den Parameter **-class**, der das *Win32_DiskDrive* angibt, als auch den Parameter **-computer**, der von der automatischen Variablen *$args* festgelegt wird. Wenn das Skript ohne Befehlszeilenargument für den Computernamen ausgeführt wird, wird ein Fehler generiert:

```
Get-WmiObject -Class Win32_DiskDrive `
-computer $args
```

Das vollständige Skript *ReportDiskDriveConfiguration.ps1* umfasst folgende Anweisungen:

ReportDiskDriveConfiguration.ps1

```
if(!$args)
 {
  Write-Host -foregroundcolor green `
  'Abfragen von localhost ...'
  $args = 'localhost'
 }
if($args -eq "?")
 { "
    ReportDiskDriveConfiguration.ps1

   BESCHREIBUNG:
   Dieses Skript unterstützt als einziges Argument einen Computernamen.
   Das Skript zeigt entweder die Laufwerkskonfiguration
   des lokalen Computers oder eines Remotecomputers an.
   Sie können ein Fragezeichen (?) oder den Namen eines Computers angeben.

   BEISPIEL:
   ReportDiskDriveConfiguration.ps1 remoteComputerName

   Ermittelt die Laufwerkskonfiguration auf einem  Computer namens  remoteComputerName.

   Das Skript kann außerdem diese Hilfeinformationen anzeigen.
   Dies wird mit dem Fragezeichen (?) als Parameter wie in folgendem Beispiel angegeben.
   ReportDiskDriveConfiguration.ps1 ?
   "
 }
Get-WmiObject -Class Win32_DiskDrive `
-computer $args
```

Erfassen der Datenträgerinformationen in Microsoft Access

Um die Konfigurationsinformationen besser verwalten zu können, speichern Sie diese in einer Access-Datenbank. Die Datenbank ermöglicht das Erstellen eines Berichts über die Laufwerkkonfiguration mehrerer Computer auf relativ einfache Art.

Rufen Sie mit dem Skript *WritePhysicalDiskInfoToAccess.ps1* den Computernamen und die Domäne des Benutzers mit Hilfe der *WshNetwork*-Klasse ab. Diese Klasse hat die Programm-ID *Wscript.Network* und wird vom Cmdlet **New-Object** instantiiert. Codebeispiel:

```
$SystemName = (New-Object -ComObject WScript.Network).computername
$DomainName = (New-Object -ComObject WScript.Network).userDomain
```

Speichern Sie die Zeichenfolge für die WMI-Abfrage in der Variablen *$strWMIQuery*. Codezeile:

```
$strWMIQuery = "Select * from win32_diskdrive"
```

Rufen Sie mit dem Cmdlet **Get-WmiObject** die Informationen aus WMI ab und geben Sie den Parameter **-query** an. Die Zeichenfolge in der Variablen *$strWMIQuery* wird an den Parameter **-query** übergeben. Speichern Sie das vom Cmdlet **Get-WmiObject** zurückgegebene *Management*-Objekt in der Variablen *$objdisks* wie folgt:

```
$objdisks = get-wmiobject -query $strWMIQuery
```

Sie können das Öffnen der Datenbank und den mehrfachen Zugriff mit zwei Variablen steuern. Diese Variablen werden mit dem Wert *3* initialisiert und in der *Open*-Methode des *Connection*-Objekts verwendet. Der Code ist übersichtlicher, wenn Sie diesen in einer Zeile angeben:

```
$adOpenStatic = $adLockOptimistic = 3
```

Die nächsten beiden Variablen dienen der Verbindungsherstellung mit der Datenbank und dem Zugriff auf die gewünschte Tabelle in der Datenbank. Die Variable *$strDB* enthält eine Zeichenfolge, die auf die Access-Datenbank verweist. In der Variablen *$strTable* ist der Name der Tabelle gespeichert, in der die Informationen erfasst werden sollen. In Abbildung 7.1 ist die Tabelle *PhyDisk* dargestellt.

Die Variablen, um auf die Datenbank und die Tabelle *PhyDisk* zu verweisen, lauten:

```
$strDB = "c:\FSO\ConfigurationMaintenance.mdb"
$strTable = "phydisk"
```

Erstellen Sie als Nächstes ein *Connection*-Objekt und ein *RecordSet*-Objekt:

```
$objConnection = New-Object -ComObject ADODB.Connection
$objRecordSet = new-object -ComObject ADODB.Recordset
```

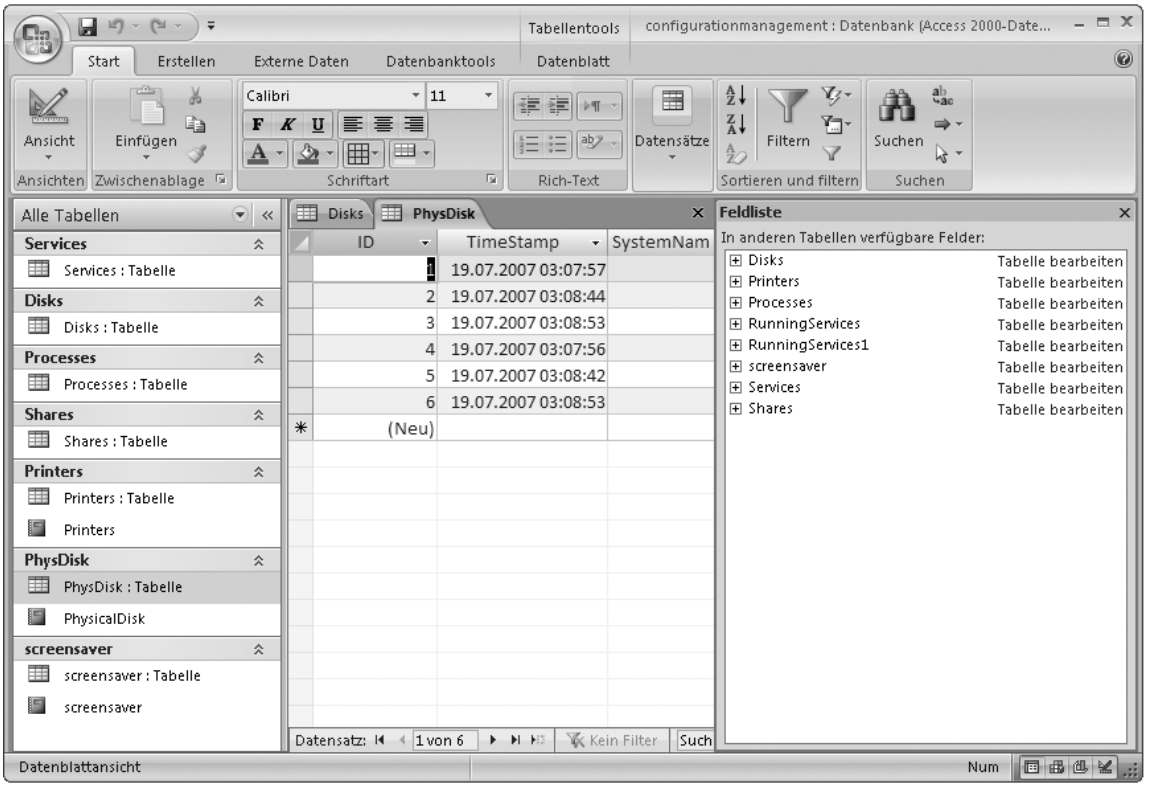


Abbildung 7.2 Das Layout der Tabelle *PhyDisk* in der Access-Datenbank

Nachdem Sie die beiden Objekte erstellt haben, rufen Sie die Methode *Open* des *Connection*-Objekts auf. Für die *Open*-Methode müssen Sie den Anbieter und den Namen des zu öffnenden Datensatzes angeben. Codezeile:

```
$objConnection.Open("Provider = Microsoft.Jet.OLEDB.4.0; `
  Data Source= $strDB")
```

Nachdem die Datenbankverbindung geöffnet wurde, können Sie einen Datensatz öffnen. Geben Sie in der Methode *Open* des *RecordSet*-Objekts die folgenden vier Parameter an: Eine Structured Query Language (SQL)-Abfrage, den Verweis auf das *Connection*-Objekt, die Methode zum Öffnen der Datenbank und die Sperrmethode. Beispielcode:

```
$objRecordSet.Open("SELECT * FROM $strTable", `
  $objConnection, $adOpenStatic, $adLockOptimistic)
```

Geben Sie nun mit dem Cmdlet **Write-Host** eine Statusmeldung aus:

```
write-host -foreGroundColor yellow "Ermitteln der Laufwerksinformationen ..."
```

Da das Cmdlet **Get-WmiObject** möglicherweise mehrere Laufwerke zurückgibt, durchlaufen Sie die *Management*-Objekte mit einer *foreach*-Anweisung. Schreiben Sie unter Verwendung der Methode *AddNew()* aus dem *RecordSet*-Objekt einen neuen Eintrag in die Datenbank. Beispiel:

```
foreach ($disk in $objdisks)
 {
   $objRecordSet.AddNew()
```

Nachdem ein neuer Datensatz zur Tabelle hinzugefügt wurde, fügen Sie die von WMI zurückgegebenen Eigenschaften in die entsprechenden Felder der Datenbank ein. Um diesen Vorgang zu vereinfachen, verwenden Sie die gleichen Namen:

```
$objRecordSet.Fields.item("TimeStamp") = Get-Date
$objRecordSet.Fields.item("systemName") = $systemName
$objRecordSet.Fields.item("DomainName") = $DomainName
$objRecordSet.Fields.item("DeviceID") = $disk.DeviceID
$objRecordSet.Fields.item("Partitions") = $disk.Partitions
$objRecordSet.Fields.item("Index") = $disk.Index
$objRecordSet.Fields.item("SectorsPerTrack") = $disk.SectorsPerTrack
$objRecordSet.Fields.item("Size") = $disk.Size
$objRecordSet.Fields.item("TotalCylinders") = $disk.TotalCylinders
$objRecordSet.Fields.item("TotalHeads") = $disk.TotalHeads
$objRecordSet.Fields.item("TotalSectors") = $disk.TotalSectors
$objRecordSet.Fields.item("TotalTracks") = $disk.TotalTracks
$objRecordSet.Fields.item("TracksPerCylinder") = $disk.TracksPerCylinder
$objRecordSet.Fields.item("FirmWareRevision") = $disk.FirmWareRevision
$objRecordSet.Fields.item("Caption") = $disk.Caption
$objRecordSet.Fields.item("Model") = $disk.Model
$objRecordSet.Fields.item("SerialNumber") = $disk.SerialNumber
```

Um die Informationen in der Datenbank zu speichern, verwenden Sie die *Update*-Methode des *RecordSet*-Objekts. Beispiel:

```
$objRecordSet.Update()
```

Geben Sie mit dem Cmdlet **Write-Host** eine Statusmeldung aus und verwenden Sie dabei für jedes abgerufene Element ein /\-Zeichen:

```
write-host -foregroundColor yellow "/\" -noNewLine
```

Der letzte Schritt umfasst die Bereinigung der angelegten Objekte. Verwenden Sie hierzu die Methode *Close()* des *RecordSet*-Objekts und des *Connection*-Objekts:

```
$objRecordSet.Close()
$objConnection.Close()
```

Nachdem das Skript *WritePhysicalDiskInfoToAccess.ps1* die Daten in die Datenbank übernommen hat, können Sie die Ergebnisse im Datenträgerbericht anzeigen. Dieser Bericht ist in Abbildung 7.3 dargestellt.

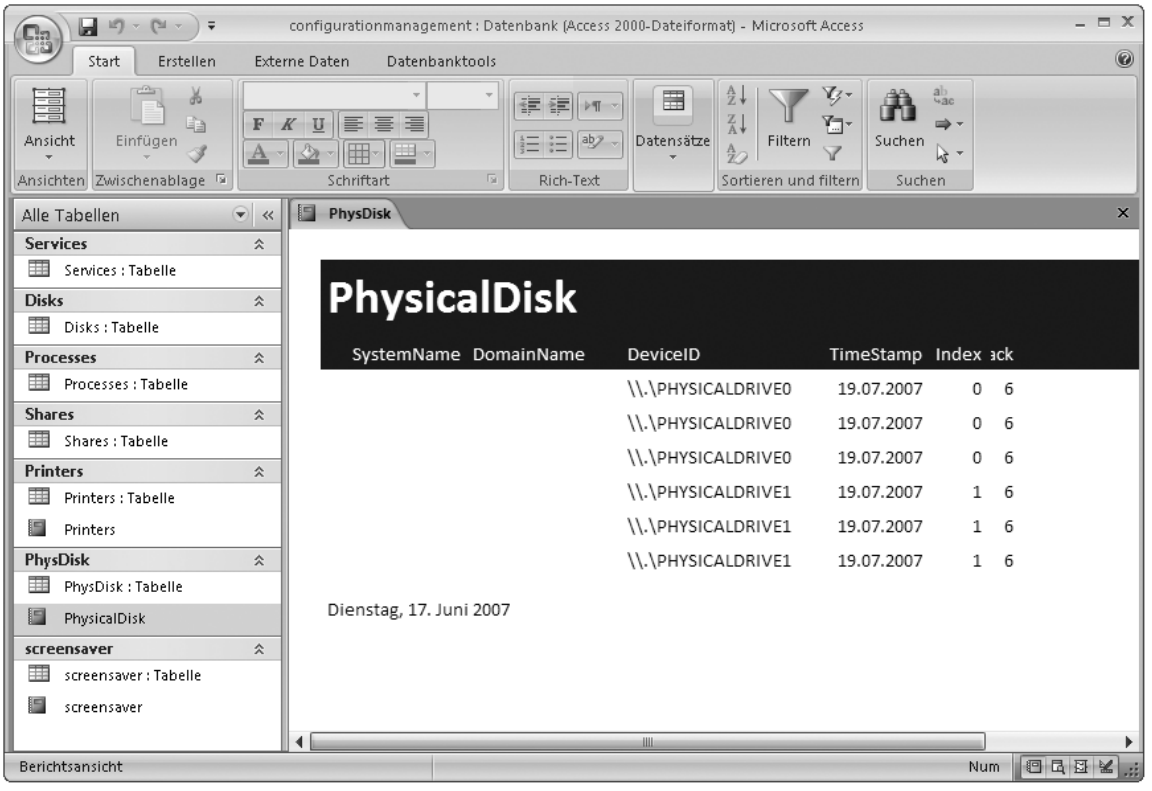


Abbildung 7.3 Der Bericht über die physischen Datenträger in Access vereinfacht das Überprüfen der Konfigurationsinformationen

Das vollständige Skript *WritePhysicalDiskInfoToAccess.ps1* hat folgenden Aufbau:

WritePhysicalDiskInfoToAccess.ps1

```
$SystemName = (New-Object -ComObject WScript.Network).computername
$DomainName = (New-Object -ComObject WScript.Network).userDomain
$strWMIQuery = "Select * from win32_diskdrive"
$objdisks = get-wmiobject -query $strWMIQuery

$adOpenStatic = $adLockOptimistic = 3
$strDB = "C:\FSO\ConfigurationMaintenance.mdb"
$strTable = "phydisk"
$objConnection = New-Object -ComObject ADODB.Connection
$objRecordSet = new-object -ComObject ADODB.Recordset
```

```
$objConnection.Open("Provider = Microsoft.Jet.OLEDB.4.0; `
  Data Source= $strDB")
$objRecordSet.Open("SELECT * FROM $strTable", `
  $objConnection, $adOpenStatic, $adLockOptimistic)

write-host -foreGroundColor yellow "Ermitteln der Laufwerksinformationen ..."

foreach ($disk in $objdisks)
 {
   $objRecordSet.AddNew()
   $objRecordSet.Fields.item("TimeStamp") = Get-Date
   $objRecordSet.Fields.item("systemName") = $systemName
   $objRecordSet.Fields.item("DomainName") = $DomainName
   $objRecordSet.Fields.item("DeviceID") = $disk.DeviceID
   $objRecordSet.Fields.item("Partitions") = $disk.Partitions
   $objRecordSet.Fields.item("Index") = $disk.Index
   $objRecordSet.Fields.item("SectorsPerTrack") = $disk.SectorsPerTrack
   $objRecordSet.Fields.item("Size") = $disk.Size
   $objRecordSet.Fields.item("TotalCylinders") = $disk.TotalCylinders
   $objRecordSet.Fields.item("TotalHeads") = $disk.TotalHeads
   $objRecordSet.Fields.item("TotalSectors") = $disk.TotalSectors
   $objRecordSet.Fields.item("TotalTracks") = $disk.TotalTracks
   $objRecordSet.Fields.item("TracksPerCylinder") = $disk.TracksPerCylinder
   $objRecordSet.Fields.item("FirmWareRevision") = $disk.FirmWareRevision
   $objRecordSet.Fields.item("Caption") = $disk.Caption
   $objRecordSet.Fields.item("Model") = $disk.Model
   $objRecordSet.Fields.item("SerialNumber") = $disk.SerialNumber

   $objRecordSet.Update()
   write-host -foregroundColor yellow "/\" -noNewLine
 }

$objRecordSet.Close()
$objConnection.Close()
```

Arbeiten mit Partitionen

Auf Servern und Arbeitsstationen ist es manchmal schwierig physische Laufwerke und Datenträgerpartitionen auseinanderzuhalten. Einer der Gründe hierfür ist, dass Windows die physische Hardware vom Betriebssystem ableitet. Möglicherweise werden Laufwerk C, D und E angezeigt, obwohl tatsächlich nur ein physischer Datenträger mit drei Partitionen vorhanden ist. Die physischen Informationen sind insbesondere dann wichtig, wenn auf einem Computer nicht genügend Speicherplatz verfügbar ist und Sie die aktuelle Partition erweitern oder ein neues Laufwerk bzw. eine neue Partition erstellen müssen. Sie können die Partitionsinformationen im Dienstprogramm **Datenträgerverwaltung** überprüfen (siehe Abbildung 7.4). Sie können die Partitionsinformationen aber auch mit dem Skript *ReportDiskPartition.ps1* überprüfen.

Im Skript *ReportDiskPartition.ps1* sollten Sie zuerst überprüfen, ob die automatische Variable *$args* einen Wert enthält. Ist die Variable *$args* nicht vorhanden, wurde das Skript ohne Argumente ausgeführt. In diesem Fall schlägt das Skript fehl. Um dieses Problem zu vermeiden, weisen Sie die Zeichen-

folge *localhost* der Variablen *$args* zu, um die WMI-Informationen vom lokalen Computer abzurufen. Teilen Sie dem Benutzer mit, dass ein Standardwert verwendet wird. Der folgende Codeabschnitt verdeutlicht diese Vorgehensweise:

```
if(!$args)
 {
  Write-Host -foregroundcolor green `
  'Abfragen von localhost ...'
  $args = 'localhost'
 }
```

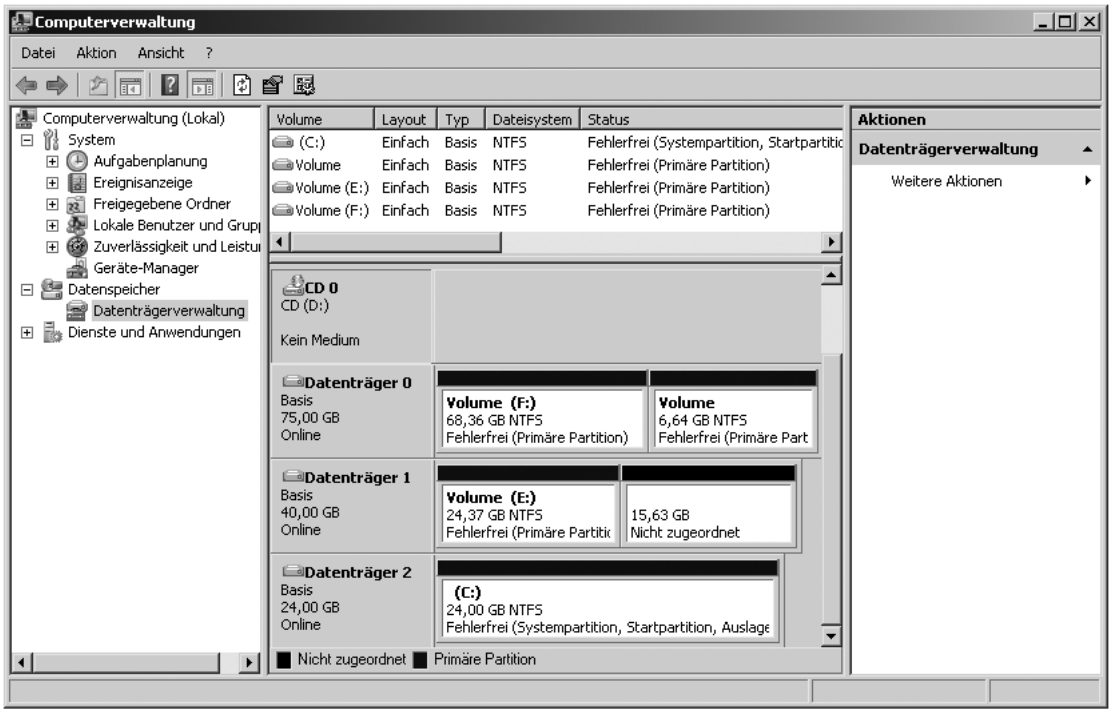


Abbildung 7.4 Die Partitionsinformationen im Dienstprogramm **Datenträgerverwaltung**

Wenn das Skript mit dem Parameter *?* ausgeführt wird, sollte eine Hilfemeldung ausgegeben werden. Sie können auch die Eingabe anderer Werte zulassen (beispielsweise **help** oder **h**). Codebeispiel:

```
if($args -eq "?")
 { "
    ReportDiskPartition.ps1

   BESCHREIBUNG:
   Dieses Skript unterstützt als einziges Argument einen Computernamen
   und zeigt entweder die Laufwerkskonfiguration
   des lokalen Computers oder eines Remotecomputers an.
   Sie können ein Fragezeichen (?) oder den Namen eines Computers angeben.

   BEISPIEL:
   ReportDiskPartition.ps1 remoteComputerName
```

```
   Ermittelt die Festplattenpartitionierung auf einem Computer namens  remoteComputerName.

   Das Skript kann außerdem diese Hilfeinformationen anzeigen.
   Dies wird mit dem Fragezeichen (?) als Parameter wie in folgendem Beispiel angegeben.
   ReportDiskPartition.ps1 ?
   "
 }
```

Das Skript verwendet das Cmdlet **Get-WmiObject** mit dem Parameter **-class**, um die Zuweisungen und Werte der Datenträgerpartition aus der WMI-Klasse *Win32_DiskPartition* abzurufen. Wenn Sie mit *$args* einen anderen Computernamen für die Abfrage angeben, wird dieser Name an den Parameter **-computer** des Cmdlets **Get-WmiObject** übergeben. Codebeispiel:

```
Get-WmiObject -Class Win32_DiskPartition `
-computer $args
```

Das vollständige Skript *ReportDiskPartition.ps1* ist wie folgt aufgebaut:

ReportDiskPartition.ps1

```
if(!$args)
 {
  Write-Host -foregroundcolor green `
  'Abfragen von localhost ...'
  $args = 'localhost'
 }
if($args -eq "?")
 { "
    ReportDiskPartition.ps1

   BESCHREIBUNG:
   Dieses Skript unterstützt als einziges Argument einen Computernamen
   und zeigt entweder die Laufwerkskonfiguration
   des lokalen Computers oder eines Remotecomputers an.
   Sie können ein Fragezeichen (?) oder den Namen eines Computers angeben.

   BEISPIEL:
   ReportDiskPartition.ps1 remoteComputerName

   Ermittelt die Festplattenpartitionierung auf einem Computer namens  remoteComputerName.

   Das Skript kann außerdem diese Hilfeinformationen anzeigen.
   Dies wird mit dem Fragezeichen (?) als Parameter wie in folgendem Beispiel angegeben.
   ReportDiskPartition.ps1 ?
   "
 }

Get-WmiObject -Class Win32_DiskPartition `
-computer $args
```

Vergleichen von Datenträgern und Partitionen

Zusätzlich zum Skript *ReportDiskPartition.ps1*, mit dem Sie Laufwerke und Partitionen ermitteln können, benötigen Sie unter Umständen ein weiteres Skript, um die Partitionsinformationen eines bestimmten Laufwerks abzufragen. Das Skript *ReportSpecificDiskPartition.ps1* zeigt die Partitionsinformationen eines einzelnen Laufwerks an. Das Skript kann *lokal* oder *remote* ausgeführt werden.

Um die Ausführung des Skripts *ReportSpecificDiskPartitioin.ps1* zu steuern, beginnen Sie mit einer *param*-Anweisung, die beim Starten des Skripts die Eingabe benannter Argumente zulässt. Definieren Sie die drei Argumente **-computer**, **-disk** und **-help**. Da der Parameter **-computer** mit dem Wert *localhost* initialisiert wird, wird das Skript für den lokalen Computer ausgeführt, wenn dieser Parameter nicht angegeben ist. Der Parameter **-disk** wird mit dem Wert *disk #0* initialisiert. Das heißt, das Skript wird ohne diesen Parameter standardmäßig für den ersten Datenträger des Computers ausgeführt. Der Parameter **-help** benötigt keinen Standardwert. Es hat keine Auswirkungen auf das Skript, wenn dieser Parameter nicht angegeben wird. Codezeile:

```
param($computer="localhost",$disk="Datenträger Nr. 0",$help)
```

Wichtig Beachten Sie bei der Angabe von Argumenten mit der *param*-Anweisung, dass *param* die erste nicht kommentierte Zeile des Skripts sein muss.

Der nächste Abschnitt des Skripts wertet die Befehlszeilenparameter aus. Überprüfen Sie mittels der entsprechenden Variablen, ob alle Argumente vorhanden sind. Wenn die Variable vorhanden ist, können Sie bestimmte Aktionen ausführen, einschließlich der Überprüfung des angegebenen Werts. Wenn der Parameter **-computer** einen Wert enthält, geben Sie eine Meldung aus, dass der entsprechende Computer abgefragt wird. Code:

```
if($computer)
 {
  Write-Host -foregroundcolor green `
  "Abfragen von $computer ..."

 }
```

Ist der Parameter **-disk** vorhanden, geben Sie eine Statusmeldung aus, dass die angegebene Partitionskonfiguration für das Laufwerk abgerufen wird. Code:

```
if($disk)
 {
  Write-Host -foregroundcolor green `
  "Abfragen der Partitionsinformationen für $disk ..."

 }
```

Der einzige Befehlszeilenparameter, der in der Parameterdefinition nicht initialisiert wird, ist der Parameter **-help**. Wenn der Parameter **-help** angegeben wurde, geben Sie den Namen des Skripts, eine Beschreibung und ein Syntaxbeispiel aus. Beenden Sie das Skript anschließend mit der *exit*-Anweisung. Codeabschnitt:

```
if($help)
 { "
    ReportSpecificDiskPartition.ps1

   BESCHREIBUNG:
   Dieses Skript unterstützt mehrere Parameter für Computername,
   Festplattennummer und Hilfeinformationen.
   und zeigt entweder die Laufwerkskonfiguration des lokalen Computers oder eines Remotecomputers an.
   Sie können den Parameter -help, den Parameter -drive,
   und den Parameter -computer für den Namen eines Computers angeben.

   BEISPIEL:
   ReportSpecificDiskPartition.ps1 -computer remoteComputername
```

```
   Ermittelt die Partitionsinformationen von Festplatte 0 auf einem Computer namens remoteComputerName.

   ReportSpecificDiskPartition.ps1 -computer remoteComputername -disk 'Datenträger Nr. 1'

   Ermittelt die Partitionsinformationen von Festplatte 1 auf einem Computer namens remoteComputerName.


   ReportSpecificDiskPartition.ps1 -help y
   Zeigt diese Hilfeinformationen an.

   "
   Exit
 }
```

Nachdem Sie den Code für die Befehlszeilenargumente geschrieben haben, rufen Sie mit dem Cmdlet **Get-WmiObject** die Partitionsinformationen des Laufwerks ab, das mit dem Parameter **-disk** festgelegt wird. Um die Eigenschaften aus der WMI-Klasse *Win32_DiskPartition* abzurufen, geben Sie den Parameter **-class** an. Geben Sie den Computer mit dem Parameter **-computer** an. Fügen Sie das zurückgegebene *Management*-Objekt in das Cmdlet **Where-Object** ein. Überprüfen Sie im Code, bei der Erstellung des Filters für das Cmdlet **Where-Object** die *Name*-Eigenschaft des aktuellen Pipelineobjekts. Wenn dieser Wert mit dem Wert in der Variablen *$disk* übereinstimmt, setzen Sie die Pipeline mit dem Cmdlet **Format-List** fort, um eine Liste der Eigenschaften anzuzeigen, die mit einem Buchstaben zwischen *a* und *z* beginnen. Auf diese Weise werden die Systemeigenschaften unterdrückt. Codeabschnitt:

```
Get-WmiObject -Class Win32_DiskPartition `
-computer $computer | Where-Object { $_.name -match $Disk } |
format-list [a-z]*
```

Das vollständige Skript *ReportSpecificDiskPartition.ps1* hat folgenden Aufbau:

ReportSpecificDiskPartition.ps1

```
param($computer="localhost",$disk="Datenträger Nr. 0",$help)

if($computer)
 {
  Write-Host -foregroundcolor green `
  "Abfragen von $computer ..."

 }
if($disk)
 {
  Write-Host -foregroundcolor green `
  "Abfragen der Partitionsinformationen für $disk ..."

 }
if($help)
 { "
    ReportSpecificDiskPartition.ps1

   BESCHREIBUNG:
   Dieses Skript unterstützt mehrere Parameter für Computername, Festplattennummer und Hilfeinformationen
   und zeigt entweder die Laufwerkskonfiguration des lokalen Computers oder eines Remotecomputers an.
   Sie können den Parameter -help, den Parameter -drive,
   und den Parameter -computer für den Namen eines Computers angeben.
```

```
   BEISPIEL:
   ReportSpecificDiskPartition.ps1 -computer remoteComputername

   Ermittelt die Partitionsinformationen von Festplatte 0 auf einem Computer namens remoteComputerName.

   ReportSpecificDiskPartition.ps1 -computer remoteComputername -disk 'Datenträger Nr. 1'

   Ermittelt die Partitionsinformationen von Festplatte 1 auf einem Computer namens remoteComputerName.


   ReportSpecificDiskPartition.ps1 -help y
   Zeigt diese Hilfeinformationen an.

   "
   Exit
 }

Get-WmiObject -Class Win32_DiskPartition `
-computer $computer | Where-Object { $_.name -match $Disk } |
format-list [a-z]*
```

Arbeiten mit logischen Datenträgern

Nachdem Sie die Datenträgerpartitionen ermittelt haben, können Sie die Konfiguration der logischen Datenträger des Computers überprüfen. Diese Aufgabe lässt sich mit dem Skript *ReportLogicalDiskConfiguration.ps1* schnell bewerkstelligen.

Überprüfen Sie zunächst die Befehlszeilenargumente im Skript *ReportLogicalDiskConfiguration.ps1*. Die automatische Variable *$args* sollte vorhanden sein. Ist dies nicht der Fall, wurde das Skript ohne Befehlszeilenargumente ausgeführt. Geben Sie mit dem Cmdlet **Write-Host** eine entsprechende Meldung aus, dass der lokale Computer abgefragt wird und die Standardwerte verwendet werden. Codebeispiel:

```
if(!$args)
 {
  Write-Host -foregroundcolor green `
  'Abfragen von localhost ...'
  $args = 'localhost'
 }
```

Wenn die automatische Variable *$args* vorhanden ist und den Wert *?,* hat, geben Sie eine Hilfemeldung aus. Zeigen Sie den Namen des Skripts, eine Beschreibung und Syntaxbeispiele an. Codeabschnitt:

```
if($args -eq "?")
 { "
    ReportLogicalDiskConfiguration.ps1

   BESCHREIBUNG:
   Dieses Skript unterstützt als einziges Argument einen Computernamen
   und zeigt entweder die Laufwerkskonfiguration
   des lokalen Computers oder eines Remotecomputers an.
   Sie können ein Fragezeichen (?) oder den Namen eines Computers angeben.

   BEISPIEL:
   ReportLogicalDiskConfiguration.ps1 remoteComputerName

   Ermittelt die logische Datenträgerkonfiguration auf einem Computer namens remoteComputerName.
```

```
   Das Skript kann außerdem diese Hilfeinformationen anzeigen.
   Dies wird mit dem Fragezeichen (?) als Parameter wie in folgendem Beispiel angegeben.
   ReportLogicalDiskConfiguration.ps1 ?
   "
 }
```

Um die Konfigurationsinformationen der logischen Datenträger des Computers abzurufen, erstellen Sie mit dem Cmdlet **Get-WmiObject** und dem Parameter **-class** ein Objekt der WMI-Klasse *Win32_LogicalDisk*. Fragen Sie dabei mit dem Parameter **-computer** den in der Variablen *$args* angegebenen Computer ab. Codebeispiel:

```
Get-WmiObject -Class Win32_LogicalDisk `
-computer $args
```

Das vollständige Skript *ReportLogicalDiskConfiguration.ps1* hat folgenden Aufbau:

ReportLogicalDiskConfiguration.ps1

```
if(!$args)
 {
  Write-Host -foregroundcolor green `
  'Abfragen von localhost ...'
  $args = 'localhost'
 }
if($args -eq "?")
 { "
    ReportLogicalDiskConfiguration.ps1

   BESCHREIBUNG:
   Dieses Skript unterstützt als einziges Argument einen Computernamen
   und zeigt entweder die Laufwerkskonfiguration
   des lokalen Computers oder eines Remotecomputers an.
   Sie können ein Fragezeichen (?) oder den Namen eines Computers angeben.

   BEISPIEL:
   ReportLogicalDiskConfiguration.ps1 remoteComputerName

   Ermittelt die Festplattenpartitionierung auf einem Computer namens remoteComputerName.

   Das Skript kann außerdem diese Hilfeinformationen anzeigen.
   Dies wird mit dem Fragezeichen (?) als Parameter wie in folgendem Beispiel angegeben.
   ReportLogicalDiskConfiguration.ps1 ?
   "
 }

Get-WmiObject -Class Win32_LogicalDisk `
-computer $args
```

Mit dem Skript *ReportSpecificLogicalDisk.ps1* können Sie die Konfigurationsinformationen eines logischen Datenträgers abrufen. Verwenden Sie hierzu die *param*-Anweisung, um das Skript mit benannten Argumenten auszuführen. Das Skript *ReportSpecificLogicalDisk.ps1* akzeptiert die drei Parameter **-computer**, **-disk** und **-help**. Der Parameter **-computer** wird mit dem Wert *localhost* initialisiert. Wenn für den Parameter **-computer** kein Wert in der Befehlszeile eingegeben wird, wird der Standardwert *localhost* für die Variable *$computer* verwendet. Der Parameter **-disk** wird mit dem

Wert *C:* initialisiert, um das Skript für Laufwerk C auszuführen, wenn kein anderer Wert angegeben wird. Der Parameter **-help** hat einen festen Wert und wird ignoriert, wenn dieser nicht in der Befehlszeile angegeben wurde. Codezeile:

```
param($computer="localhost",$disk="C:",$help)
```

Wenn die Variable *$computer* vorhanden ist, geben Sie mit dem Cmdlet **Write-Host** eine Meldung mit dem Namen des abgefragten Computers aus. Wenn für den Parameter **-computer** kein Wert eingegeben wird, hat die Variable *$computer* den Standardwert *localhost*. Codebeispiel:

```
if($computer)
 {
  Write-Host -foregroundcolor green `
  "Abfragen von $computer ..."

 }
```

Wenn die Variable *$disk* vorhanden ist, geben Sie mit dem Cmdlet **Write-Host** eine Meldung mit dem Namen des abgefragten Laufwerks aus. Wenn mit dem Parameter **-disk** kein Laufwerk angegeben wurde, wird wie für den Parameter *$disk* der Standardwert *C:* verwendet. Codebeispiel:

```
if($disk)
 {
  Write-Host -foregroundcolor green `
  "Abfragen der logischen Datenträgerinformationen für $disk ..."

 }
```

Wenn die Variable *$help* vorhanden ist, wurde der Parameter **-help** beim Ausführen des Skripts in der Befehlszeile angegeben. Entsprechend wird eine Meldung mit dem Namen des Skripts, einer Beschreibung und einem Syntaxbeispiel angezeigt. Nach Anzeigen der Hilfemeldung wird das Skript mit einer *exit*-Anweisung beendet. Codebeispiel:

```
if($help)
 { "
    ReportSpecificLogicalDisk.ps1

   BESCHREIBUNG:
   Dieses Skript unterstützt mehrere Parameter für Computername, Festplattennummer und Hilfeinformationen
   und zeigt entweder die logische Datenträgerkonfiguration des lokalen Computers oder eines Remotecomputers an.
   Sie können den Parameter -help, den Parameter -drive,
   und den Parameter -computer für den Namen eines Computers angeben.

   BEISPIEL:
   ReportSpecificLogicalDisk.ps1 -computer remoteComputername

   Ermittelt die logische Datenträgerkonfiguration von Laufwerk C: auf einem Computer namens remoteComputerName.

   ReportSpecificLogicalDisk.ps1 -computer remoteComputername -disk 'D:'

   Ermittelt die logische Datenträgerkonfiguration von Laufwerk D: auf einem Computer namens remoteComputerName.


   ReportSpecificLogicalDiskn.ps1 -help y
   Zeigt diese Hilfeinformationen an.

   "
   Exit
 }
```

Um die logische Datenträgerkonfiguration abzurufen, verwenden Sie das Cmdlet **Get-WmiObject** und geben die WMI-Klasse *Win32_LogicalDisk* im Parameter **-class** an. Geben Sie das Gravis-Zeichen ein, um den Code in der nächsten Zeile fortzusetzen. Geben Sie im Parameter **-computer** den Wert an, der in der Variablen *$computer* verwendet werden soll. Fügen Sie das resultierende Verwaltungsobjekt in das Cmdlet **Where-Object** ein und filtern Sie die Ergebnisse anhand der Eigenschaft *DeviceID* des aktuellen Pipelineobjekts. Wenn der Wert der Eigenschaft *DeviceID* mit dem Wert in der Variablen *$disk* übereinstimmt, fügen Sie das Objekt in das Cmdlet **Format-List** ein. Verwenden Sie ausschließlich die Eigenschaften, die mit einem Buchstaben zwischen *a* und *z* beginnen, um auf diese Weise die Systemeigenschaften, die mit einem doppelten Unterstrich beginnen, zu unterdrücken. Folgender Code demonstriert diese Vorgehensweise:

```
Get-WmiObject -Class Win32_LogicalDisk `
-computer $computer | Where-Object { $_.deviceID -match $Disk } |
format-list [a-z]*
```

Das vollständige Skript *ReportSpecificLogicalDisk.ps1* hat folgenden Aufbau:

ReportSpecificLogicalDisk.ps1

```
param($computer="localhost",$disk="c:",$help)

if($computer)
 {
  Write-Host -foregroundcolor green `
  "Abfragen von $computer ..."

 }
if($disk)
 {
  Write-Host -foregroundcolor green `
  "Abfragen der logischen Datenträgerkonfiguration für $disk ..."

 }
if($help)
 { "
    ReportSpecificLogicalDisk.ps1

   BESCHREIBUNG:
   Dieses Skript unterstützt mehrere Parameter für Computername, Festplattennummer und Hilfeinformationen
   und zeigt entweder die logische Datenträgerkonfiguration des lokalen Computers oder eines Remotecomputers an.
   Sie können den Parameter -help, den Parameter -drive,
   und den Parameter -computer für den Namen eines Computers angeben.

   BEISPIEL:
   ReportSpecificLogicalDisk.ps1 -computer remoteComputername

   Ermittelt die logische Datenträgerkonfiguration von Laufwerk C: auf einem Computer namens remoteComputerName.

   ReportSpecificLogicalDisk.ps1 -computer remoteComputername -disk 'D:'

   Ermittelt die logische Datenträgerkonfiguration von Laufwerk D: auf einem Computer namens remoteComputerName.


   ReportSpecificLogicalDiskn.ps1 -help y
   Zeigt diese Hilfeinformationen an.
```

```
   "
   Exit
 }

Get-WmiObject -Class Win32_LogicalDisk `
-computer $computer | Where-Object { $_.deviceID -match $Disk } |
format-list [a-z]*
```

Überwachen der Speicherplatzbelegung

Auch bei großen Festplatten kann eine typische Windows Vista-Installation schnell den gesamten verfügbaren Speicherplatz belegen. Obwohl dieses Problem möglicherweise nicht sofort erkennbar ist, wird es mit der Zeit offensichtlich. Wenn nicht genügend Speicherplatz zur Verfügung steht, sollten Sie zunächst überprüfen, dass tatsächlich der gesamte Speicherplatz belegt ist. Sie können die Speicherplatzinformationen im Dialogfeld **Eigenschaften** für ein Laufwerk anzeigen (siehe Abbildung 7.5). Es ist jedoch oft schwierig, sich an die vorherigen Werte zu erinnern.

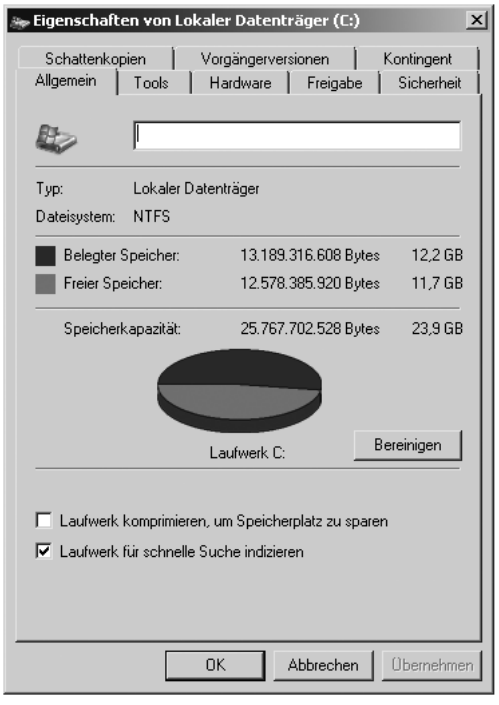


Abbildung 7.5 Das Dialogfeld mit den Laufwerkeigenschaften enthält eine Übersicht der Speicherplatzbelegung

Um die Werte nachzuverfolgen, benötigen Sie eine Trendanalyse der Speicherplatzbelegung im Zeitablauf. Sie können die Speicherplatzstatistik in einer Datei, einem Microsoft Excel-Arbeitsblatt oder einer Microsoft Access-Datenbank speichern.

Beginnen Sie das Skript *MonitorVolumeSpace.ps1* mit der Funktion *funline*. Die Funktion *funline* unterstreicht die Ausgabe, um die Ausgabedaten für die einzelnen Laufwerke übersichtlich zu trennen, und akzeptiert eine Zeichenfolge, die in der Variablen *$strIN* übergeben werden kann. Die Funktion

fragt die Eigenschaft *Length* der Eingabezeichenfolge ab und speichert diesen Wert in der Variablen *$num*. Danach verwendet die Funktion eine *for*-Schleife von 1 bis zur Länge der Eingabezeichenfolge. Die *for*-Schleife verwendet die Variable *$i* als Enumerator, verkettet das Gleichheitszeichen (=) und speichert die Ergebnisse in der Variablen *$funline*. Schließlich wird die Zeichenfolge mit dem Cmdlet **Write-Host** in Gelb ausgegeben. Geben Sie mit dem Cmdlet **Write-Host** die Zeile mit dem Gleichheitszeichen, die in der Variablen *$funline* gespeichert ist, in Dunkelgelb aus. Die Funktion *funline* ist wie folgt implementiert:

```
 function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline = $funline + "=" }
   Write-Host -ForegroundColor yellow $strIN
   Write-Host -ForegroundColor darkYellow $funline
}
```

Erstellen Sie im Skript *MonitorVolumeSpace.ps1* ein Array der Computernamen. Beispielsweise enthält das Skript die beiden hart codierten Werte *localhost* und *loopback*, die sich auf den lokalen Computer beziehen. Das heißt, dass das Skript den Speicherplatz des lokalen Computers zweimal ausgibt. Um die Werte von anderen Computern abzurufen, ändern Sie entsprechend die Werte in der Variablen *$aryComputer*. Sie müssen die Codezeile *$aryComputer* ändern, um das Skript für einen anderen Computer auszuführen:

```
$arycomputer = "localhost", "loopback"
```

Problembehandlung Skripts geben oft unerwartete Ergebnisse zurück. Ich habe beispielsweise kürzlich einen Computer mit dem Namen *localhost* in einem Netzwerk gefunden. Das Beheben dieser Art von Problemen kann mühsam sein. Das Skript *MonitorVolumeSpace.ps1* würde in diesem Netzwerk zwei unterschiedliche Werte ausgeben.

Die nächste Codezeile im Skript *MonitorVolumeSpace.ps1* beginnt eine *foreach*-Schleife. Die *foreach*-Anweisung durchläuft die Computernamen in der Variablen *$aryComputer*. Das Skript *MonitorVolumeSpace.ps1* verwendet die Variable *$computer* als Enumerator für das Array *$aryComputer*. Codezeile:

```
foreach($computer in $arycomputer)
```

Die Variable *$volumeSet* enthält das *Management*-Objekt, das vom Cmdlet **Get-WmiObject** bei der Abfrage der WMI-Klasse *Win32_Volume* zurückgegeben wird. Der Parameter **-computer** gibt dabei den Computer an, von dem die WMI-Klasseninformationen abgerufen werden sollen. Der Wert in der Variablen *$computer* wird an den Parameter **-computer** des Cmdlets **Get-WmiObject** übergeben. Um das Ergebnis auf lokale Festplatten zu beschränken, filtern Sie die Ergebnisse mit dem Parameter **-filter** nach drivetype = 3. Codebeispiel:

```
 $volumeSet = Get-WmiObject -Class win32_volume -computer $computer `
-filter "drivetype = 3"
```

Da die Abfrage möglicherweise mehrere Laufwerke zurückgibt, müssen Sie das *$volumeSet*-Objekt mit einer *foreach*-Anweisung durchlaufen. Geben Sie die Variable *$volume* als Enumerator an:

```
foreach($volume in $volumeSet)
```

Um mit einem einzelnen Laufwerk zu arbeiten, rufen Sie als Erstes den entsprechenden Laufwerkbuchstaben ab. Speichern Sie diesen Wert in der Variablen *$drive*. Codezeile:

```
$drive=$volume.driveLetter
```

Rufen Sie anschließend den freien Speicherplatz auf dem Laufwerk ab. Diese Informationen sind in der Eigenschaft *FreeSpace* der WMI-Klasse *Win32_Volume* verfügbar und werden in Bytes angezeigt. In Windows PowerShell können Sie diese Werte einfach umwandeln. Zum Umwandeln des Werts in Gigabytes benötigen Sie eine Instanz des Laufwerkobjekts und die Windows PowerShell-Konstante GB. Da eine ganze Zahl zurückgegeben werden soll, verwenden Sie die *[int]*-Einschränkung für die Variable *$free*, in der die Ergebnisse zum Ausgeben des Statusberichts gespeichert sind. Codezeile:

```
[int]$free=$volume.freespace/1GB
```

Als Nächstes müssen Sie die Kapazität des Laufwerks ermitteln. Rufen Sie diese Informationen aus der Eigenschaft *Capacity* der WMI-Klasse *Win32_Volume* ab. Sie können die Kapazität in Gigabytes umwandeln. Verwenden Sie hierzu wieder die GB-Konstante und dividieren Sie mit dieser den Kapazitätswert. Speichern Sie das Ergebnis als ganze Zahl in der Variablen *$capacity*. Beispiel:

```
[int]$capacity=$volume.capacity/1GB
```

Um einen Header auszugeben, verwenden Sie die Funktion *funline* und geben eine Zeichenfolge mit dem Namen des Computers und den Laufwerkinformationen an. Codezeile:

```
funline("Laufwerke auf dem: $computer")
```

Geben Sie für jedes Laufwerk des Computers eine Statusmeldung aus. Die Statusmeldung zeigt an, dass das Laufwerk analysiert wird und gibt den Laufwerkbuchstaben sowie den Servernamen aus. Hier ist ein Vorschlag: Anstatt ein weiteres Objekt zum Abrufen des Computernamens zu erstellen, verwenden Sie die Systemeigenschaft *__Server*. Codebeispiel:

```
"Analysieren von Laufwerk $drive $($volume.label) auf $($volume.__server)"
```

Tipp Beinahe jede WMI-Klasse umfasst die Systemeigenschaft *__Server*. Sie können mit dieser Eigenschaft den Namen des abgefragten Computers abrufen.

Geben Sie eine weitere Meldung mit dem Prozentsatz des freien Speicherplatzes auf dem Laufwerk aus. Geben Sie das **`t**-Zeichen ein, um zur nächsten Position zu wechseln. Geben Sie zwei **`t`t**-Zeichen ein, um zwei Positionen zu überspringen, und zeigen Sie die Meldung sowie den Laufwerkbuchstaben an. Verknüpfen Sie die Zeichenfolge mit einem Pluszeichen. Geben Sie die Formatbezeichnung *{0:N2}* an, um den Wert mit zwei Dezimalzahlen auszugeben. Berechnen Sie den Prozentsatz des freien Speicherplatzes mit der Formel *($free/$capacity)*100*. Codebeispiel:

```
"`t`t Prozentualer Anteil des freien Speicherplatzes auf Laufwerk $drive " +  "{0:N2}" -f `
   (($free/$capacity)*100)
```

Das vollständige Skript *MonitorVolumeSpace.ps1* lautet wie folgt:

MonitorVolumeSpace.ps1

```
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline = $funline + "=" }
   Write-Host -ForegroundColor yellow $strIN
   Write-Host -ForegroundColor darkYellow $funline
}
```

```
$arycomputer = "localhost", "loopback"

foreach($computer in $arycomputer)
{
 $volumeSet = Get-WmiObject -Class win32_volume -computer $computer `
-filter "drivetype = 3"
 foreach($volume in $volumeSet)
  {
   $drive=$volume.driveLetter
   [int]$free=$volume.freespace/1GB
   [int]$capacity=$volume.capacity/1GB
   funline("Drives on $computer computer:")
   "Analysieren von Laufwerk $drive $($volume.label) auf $($volume.__server)"
   "`t`t Prozentualer Anteil des freien Speicherplatzes auf Laufwerk $drive " +  "{0:N2}" -f `
   (($free/$capacity)*100)
   }
}
```

Protokollieren des Speicherplatzes in einer Datenbank

Um die Speicherplatzbelegung nachzuverfolgen, müssen Sie die entsprechenden Informationen an einer zentralen Stelle speichern, beispielsweise in einer Access-Datenbank. Mit dem Skript *WriteDiskSpaceInfoToAccess.ps1* können Sie die Kapazität der Laufwerke eines Computers abrufen und die Informationen beispielsweise in einer Variablen namens *$capacity* speichern. Rufen Sie anschließend den freien Speicherplatz auf den Laufwerken ab und speichern Sie diese Informationen in der Variablen *$freespace*. Nachdem der freie Speicherplatz in Prozent berechnet wurde, können Sie die Informationen in der Datenbank abspeichern. Dieser Vorgang wird in folgendem Abschnitt erklärt.

Beginnen Sie das Skript *WriteDiskSpaceInfoToAccess.ps1* mit einer Zeichenfolge für die WMI-Abfrage. Wählen Sie aus der WMI-Klasse *Win32_Volume* alle Werte aus, die mit dem Wert *3* der Eigenschaft *DriveType* übereinstimmen, um die Ergebnisse auf die lokalen Festplatten zu beschränken. Weisen Sie das *Management*-Objekt, das vom Cmdlet **Get-WmiObject** zurückgegeben wird, der Variablen *$objdisks* zu. Geben Sie mit dem Parameter **-query** die WQL-Syntaxabfrage anhand der Variablen *$strWMIQuery an.* Die beiden Codezeilen lauten:

```
$strWMIQuery = "Select * from win32_volume where drivetype=3"
$objdisks = get-wmiobject -query $strWMIQuery
```

Erstellen und initialisieren Sie in den nächsten vier Codezeilen die Variablen. Die ersten drei initialisierten Variablen erfassen den freien Speicherplatz, die Datenträgerkapazität und den berechneten Prozentsatz des freien Speicherplatzes. Um sicherzustellen, dass in den Variablen keine veralteten Informationen gespeichert sind, legen Sie den Wert der Variablen auf *$null* fest. Sie können den erforderlichen Code in einer Zeile zusammenfassen. Die nächsten beiden Variablen legen fest, wie auf die Datenbank zugegriffen werden soll. Diese Variablen, *$adOpenStatic* und *$adLockOptimistic*, werden in der gleichen Zeile angegeben und auf den Wert *3* festgelegt. Die nächsten beiden Variablen geben den Pfad zur Datenbank und zur Tabelle an. Die vier erforderlichen Codezeilen lauten entsprechend:

```
$percentFree=$free=$capacity=$null
$adOpenStatic = $adLockOptimistic = 3
$strDB = "C:\FSO\ConfigurationMaintenance.mdb"
$strTable = "diskSpace"
```

Nachdem die ursprünglichen Variablen erstellt und initialisiert wurden, müssen Sie mehrere Objekte erstellen. Im Skript *WriteDiskSpaceInfoToAccess.ps1* werden dementsprechend vier **New-Object**-Befehle angegeben. Die ersten zwei Zeilen erstellen das gleiche Objekt mit unterschiedlichen Eigenschaften. Das *wshNetwork*-Objekt gibt den Namen des Computers und die Domäne an, in der sich der Computer befindet. Die nächten zwei **New-Object**-Zeilen erstellen das *ADODB Connection*-Objekt und das *ADODB RecordSet*-Objekt. Die vier erforderlichen Codezeilen lauten:

```
$SystemName = (New-Object -ComObject WScript.Network).computername
$DomainName = (New-Object -ComObject WScript.Network).userDomain
$objConnection = New-Object -ComObject ADODB.Connection
$objRecordSet = new-object -ComObject ADODB.Recordset
```

Nachdem die **New-Object**-Befehle ausgeführt wurden, können Sie die Datenbankverbindung öffnen. Um das *Connection*-Objekt zu öffnen, müssen Sie den Anbieter und die Datenquelle angeben. Für das Skript *WriteDiskSpaceInfoToAccess.ps1* müssen Sie beispielsweise den Microsoft.Jet.OLEDB.4.0-Anbieter verwenden. Mit folgender Codezeile wird die Datenbankverbindung geöffnet:

```
$objConnection.Open("Provider = Microsoft.Jet.OLEDB.4.0; `
  Data Source= $strDB")
```

Sobald die Datenbankverbindung geöffnet wurde, können Sie einen Datensatz öffnen. Verwenden Sie hierzu die *Open*-Methode des *RecordSet*-Objekts. Diese Methode ruft die Einträge ab, die den Datensatz bilden. Wählen Sie alle Daten aus der Tabelle aus und geben Sie eine Verbindung für die Abfrage an. Legen Sie die Methode zum Öffnen der Datenbank und den Tabellentyp fest. Codebeispiel:

```
$objRecordSet.Open("SELECT * FROM $strTable", `
$objConnection, $adOpenStatic, $adLockOptimistic)
```

Geben Sie anschließend mit dem Cmdlet **Write-Host** eine Statusmeldung aus. Beispiel:

```
write-host -foreGroundColor yellow "Ermitteln der Speicherplatzinformationen ..."
```

Durchlaufen Sie die vom Cmdlet **Get-WmiObject** zurückgegebenen Laufwerke. Rufen Sie den freien Speicherplatz ab, konvertieren Sie diesen in MB und weisen Sie den Wert der Variablen *$free* zu. Verwenden Sie die *[int]*-Einschränkung, um sicherzustellen, dass die Daten als ganze Zahl gespeichert werden. Rufen Sie die Eigenschaft *Capacity* ab und konvertieren Sie den Wert in Megabytes. Speichern Sie das Ergebnis in der Variablen *$capacity*. Verwenden Sie wieder die *[int]*-Einschränkung, um sicherzustellen, dass die Kapazitätsinformationen als ganze Zahl in der Variablen gespeichert werden. Die beiden Codezeilen lauten:

```
[int]$free =$disk.freespace/1MB
[int]$capacity = $disk.capacity/1MB
```

Um die Datenbankabfrage zu vereinfachen, berechnen Sie den Prozentsatz des freien Speicherplatzes auf dem Laufwerk. Auf diese Art müssen Sie die Felder im Bericht nicht einzeln berechnen. Codezeile:

```
$percentFree = ($free/$capacity)*100
```

Sie müssen nun nur noch einen Datensatz zur Datenbank hinzufügen, um die Informationen zu speichern. Verwenden Sie hierzu die Methode *AddNew()* und rufen anschließend mit der Methode *item* des *RecordSet*-Objekts die Felder aus der Datenbank ab. Die Design-Ansicht der Datenbanktabelle vereinfacht das Erstellen der Eigenschaftsnamen (siehe Abbildung 7.6).

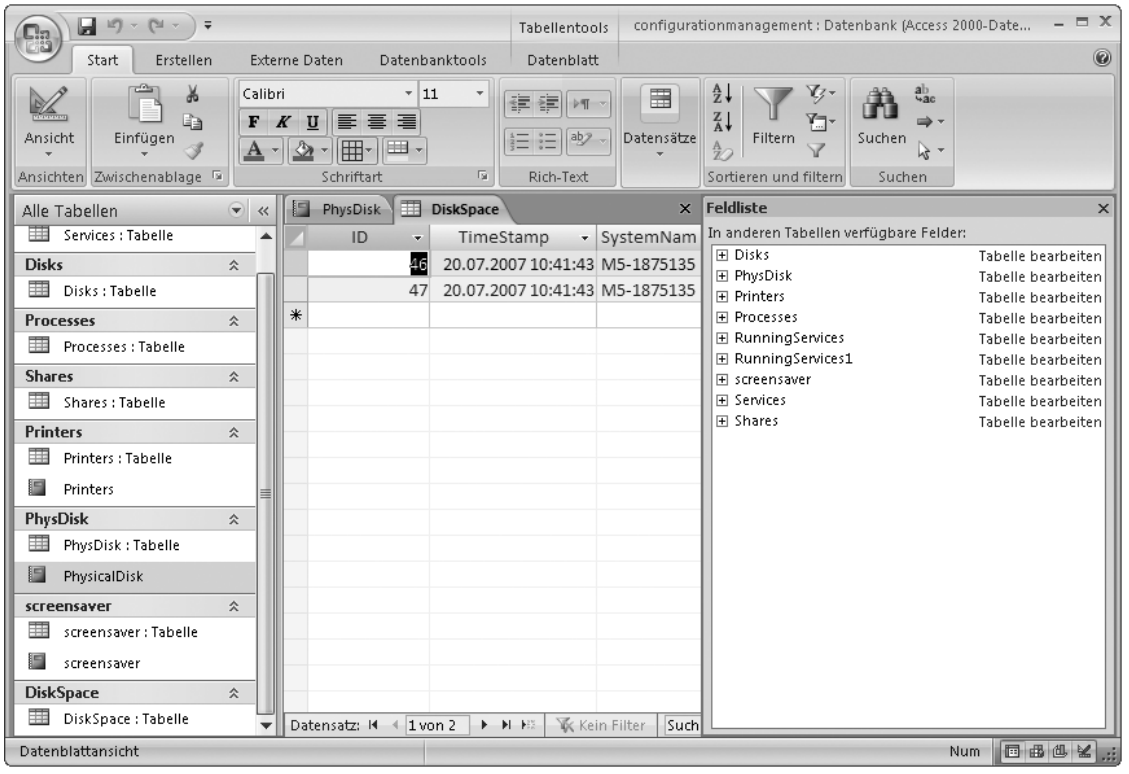


Abbildung 7.6 Die Datenbanktabelle **DiskSpace** in der Designansicht

Weisen Sie jedem Feld einen Wert zu. Nachdem Sie die Informationen zum neuen Datensatz hinzugefügt haben, übernehmen Sie die Änderungen mit der Methode *Update()*. Beispiel:

```
$objRecordSet.AddNew()
$objRecordSet.Fields.item("TimeStamp") = Get-Date
$objRecordSet.Fields.item("systemName") = $systemName
$objRecordSet.Fields.item("DomainName") = $DomainName
$objRecordSet.Fields.item("DriveLetter") = $disk.DriveLetter
$objRecordSet.Fields.item("FreeSpace") = $free
$objRecordSet.Fields.item("Capacity") = $capacity
$objRecordSet.Fields.item("PercentFree") = $percentFree
$objRecordSet.Update()
```

Nachdem Sie die Methode *Update()* des *RecordSet*-Objekts aufgerufen haben, können Sie alle zurückgegebenen Laufwerke durchlaufen und für jedes abgerufene Laufwerk einen neuen Datensatz erstellen. Geben Sie mit dem Cmdlet **Write-Host** wie üblich eine Statusmeldung aus. Um die Ausgabe übersichtlicher zu formatieren, geben Sie das ***`r***-Zeichen für eine neue Zeile ein. Diese Methode ist effizienter als das **-nonewline**-Argument des Cmdlets **Write-Host**. Vergessen Sie nicht, den Datensatz und das *Connection*-Objekt zu schließen. Codebeispiel:

```
write-host -foregroundColor yellow "/\" -noNewLine
 }
   "`r"
$objRecordSet.Close()
$objConnection.Close()
```

Das Skript *WriteDiskSpaceInfoToAccess.ps1* übermittelt die Speicherplatzinformationen an die Datenbank. In Abbildung 7.7 ist der Bericht **DiskSpace** in der Access-Datenbank dargestellt.

Das vollständige Skript *WriteDiskSpaceInfoToAccess.ps1* hat folgenden Aufbau:

WriteDiskSpaceInfoToAccess.ps1

```
$strWMIQuery = "Select * from win32_volume where drivetype=3"
$objdisks = get-wmiobject -query $strWMIQuery
$percentFree=$free=$capacity=$null
$adOpenStatic = $adLockOptimistic = 3
$strDB = "C:\FSO\ConfigurationMaintenance.mdb"
$strTable = "diskSpace"

$SystemName = (New-Object -ComObject WScript.Network).computername
$DomainName = (New-Object -ComObject WScript.Network).userDomain
$objConnection = New-Object -ComObject ADODB.Connection
$objRecordSet = new-object -ComObject ADODB.Recordset
$objConnection.Open("Provider = Microsoft.Jet.OLEDB.4.0; `
  Data Source= $strDB")
$objRecordSet.Open("SELECT * FROM $strTable", `
$objConnection, $adOpenStatic, $adLockOptimistic)

write-host -foreGroundColor yellow "Ermitteln der Speicherplatzinformationen  ..."

foreach ($disk in $objdisks)
 {
   [int]$free =$disk.freespace/1MB
   [int]$capacity = $disk.capacity/1MB
   $percentFree = ($free/$capacity)*100
   $objRecordSet.AddNew()
   $objRecordSet.Fields.item("TimeStamp") = Get-Date
   $objRecordSet.Fields.item("systemName") = $systemName
   $objRecordSet.Fields.item("DomainName") = $DomainName
   $objRecordSet.Fields.item("DriveLetter") = $disk.DriveLetter
   $objRecordSet.Fields.item("FreeSpace") = $free
   $objRecordSet.Fields.item("Capacity") = $capacity
   $objRecordSet.Fields.item("PercentFree") = $percentFree
   $objRecordSet.Update()
   write-host -foregroundColor yellow "/\" -noNewLine
 }
   "`r"
$objRecordSet.Close()
$objConnection.Close()
```

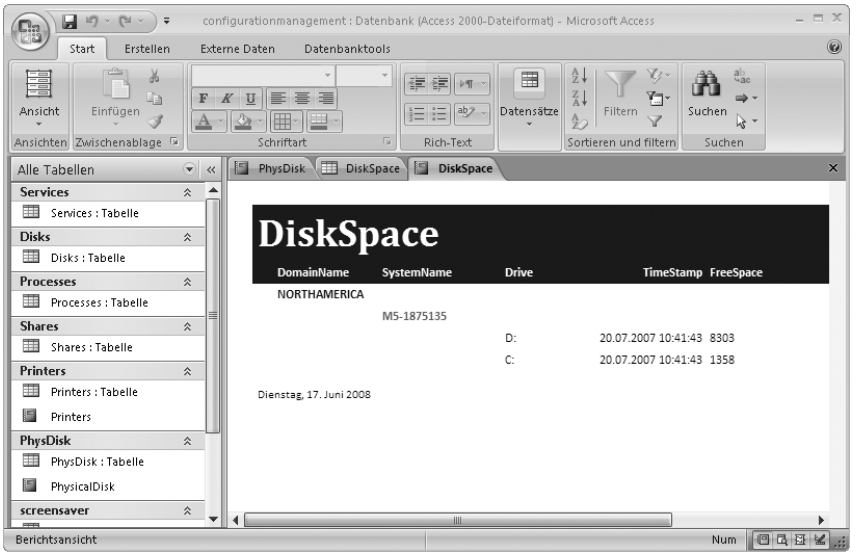


Abbildung 7.7 Der aus der Tabelle *DiskSpace* erstellte Bericht bietet eine Übersicht der Datenträgerauslastung und Tendenzen

Überwachen gespeicherter Dateien

Es ist eine Binsenweisheit von Netzwerkadministratoren, dass Benutzer nie etwas löschen. Eine nicht überwachte Dateifreigabe auf einem Server ist ein Freibrief für die Benutzer beliebig viel Speicherplatz in einer SAN-Umgebung (Storage Area Network) zu belegen. Einige Netzwerkadministratoren überwachen Dateifreigaben und entfernen veraltete Dateien manuell. Dies ist jedoch keine langfristige Lösung (siehe Abbildung 7.8).

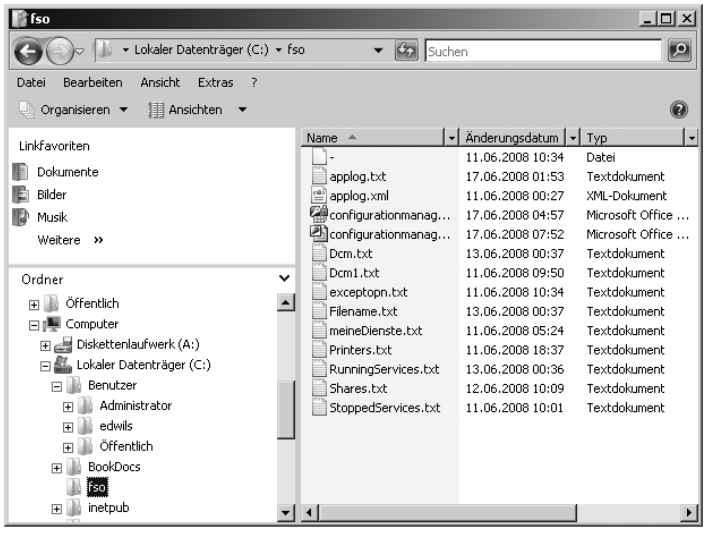


Abbildung 7.8 Manuelle Verwaltung veralteter Dateien im Windows Explorer mit nach Datum sortierter Anzeige der Dateien

Mit dem Skript *QueryOldFiles.ps1* können Sie auf einen Ordner zugreifen und die Dateien auflisten, die mehr als 30 Tage nicht verwendet wurden. Was Sie mit diesen Dateien machen, bleibt Ihnen überlassen.

Beginnen Sie das Skript *QueryOldFiles.ps1* mit der Funktion *funline*. Diese Funktion verwendet das Cmdlet **Write-Host**, um eine Kopfzeile in Dunkelgelb auszugeben und zu unterstreichen. Code:

```
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline = $funline + "=" }
    Write-Host -ForegroundColor yellow $strIN
    Write-Host -ForegroundColor darkYellow $funline
}
```

Erstellen und initialisieren Sie drei Variablen. Die Variable *$folder* gibt den Ordner an, der nach alten Dateien durchsucht werden soll. Die Variable *$date* verweist hingegen auf ein *DateTime*-Objekt mit dem aktuellen Datum. Das aktuelle Datum wird vom Cmdlet **Get-Date** abgerufen. Der Wert in der Variablen *$limit* bestimmt das maximale Alter einer Datei. Die drei erforderlichen Codezeilen lauten:

```
$folder = "C:\FSO"
$date = Get-Date
$limit = 30
```

Nachdem Sie die Variablen initialisiert haben, können Sie die Dateien mit dem Cmdlet **Get-ChildItem** abrufen. Übergeben Sie die Zeichenfolge in der Variablen *$folder* an den Parameter **-path** des Cmdlets **Get-ChildItem** und geben Sie mit dem Parameter **-force** alle ausgeblendeten Dateien zurück. Übergeben Sie das zurückgegebene *fileinfo*-Objekt anschließend an das Cmdlet **ForEach-Object**. Die beiden Codezeilen lauten:

```
Get-ChildItem -Path $folder -force |
foreach-object `
```

Erstellen Sie im Codeblock des Cmdlets **ForEach-Object** ein neues *DateTime*-Objekt und speichern Sie dieses in der Variablen *$newDate*. Um das neue *DateTime*-Objekt zu erstellen, verwenden Sie die Methode *addDays()* aus dem vorhandenen *DateTime*-Objekt, das von der Abfrage der Eigenschaft *LastAccessTime* des aktuellen *fileinfo*-Objekts zurückgegeben wird. Fügen Sie den in der Variablen *$limit* gespeicherten Wert zum *DateTime*-Objekt hinzu. Codezeile:

```
$newDate=($_.LastAccessTime).adddays($limit)
```

Erstellen Sie nun ein *TimeSpan*-Objekt, das den Stichtag für die Dateien angibt, auf die im angegebenen Zeitraum nicht zugegriffen wurde. Erstellen Sie das *TimeSpan*-Objekt mit dem Cmdlet **New-TimeSpan** und übergeben Sie das *DateTime*-Objekt, das das aktuelle Datum angibt, im Parameter **-start**. Der Parameter **-end** des Cmdlets **New-TimeSpan** ruft das *DateTime*-Objekt ab, das das Datum, zu dem zuletzt auf die Datei zugriffen wurde, und das in der Variablen *$limit* gespeicherte Zeitlimit angibt. Das resultierende *TimeSpan*-Objekt wird der Variablen *$limitdate* zugewiesen. Codezeile:

```
$limitDate = New-TimeSpan -start $date -end $newDate
```

Wenn das *TimeSpan*-Objekt in der Variablen *$limitdate* kleiner oder gleich *0* ist, wurde das in der Variablen *$limit* angegebene Zeitlimit für die Datei überschritten. Verwenden Sie in diesem Fall das *fileinfo*-Objekt in der aktuellen Pipeline, wählen Sie den Namen sowie die *LastAccessTime*-Eigenschaften aus und schreiben Sie diese in eine Hashtabelle. Codebeispiel:

```
if ($limitDate -le 0)
  {
   $xfiles += @{ $_.name = $_.lastAccessTime }
  }
```

Nachdem Sie die Dateien ausgewertet und eine Hashtabelle mit den abgelaufenen Dateien erstellt haben, müssen Sie einen Bericht generieren. Verwenden Sie hierzu das Cmdlet **Write-Host** und die Eigenschaft *Count* des *hashtable*-Objekts, um die abgelaufenen Dateien aufzulisten. Beziehen Sie den Pfad zum Ordner und den Zeitlimitwert in die Zusammenfassung ein. Markieren Sie mit der Funktion *funline* die Liste der abgelaufenen Dateien und geben Sie den Inhalt der Hashtabelle *$xfiles* aus. Codebeispiel:

```
Write-Host "Im Ordner $folder gibt es $($xfiles.count) Dateien, die älter als $limit Tage sind."
FunLine("Die veralteten Dateien sind in folgender Liste aufgeführt:")

$xfiles
```

Das vollständige Skript *QueryOldFiles.ps1* hat folgenden Inhalt:

QueryOldFiles.ps1

```
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline = $funline + "=" }
    Write-Host -ForegroundColor yellow $strIN
    Write-Host -ForegroundColor darkYellow $funline
}

$folder = "C:\FSO"
$date = Get-Date
$limit = 30

Get-ChildItem -Path $folder -force |
foreach-object `
{
 $newDate=($_.LastAccessTime).adddays($limit)
 $limitDate = New-TimeSpan -start $date -end $newDate

 if ($limitDate -le 0)
  {
   $xfiles += @{ $_.name = $_.lastAccessTime }
  }

}
Write-Host "Im Ordner $folder gibt es $($xfiles.count) Dateien, die älter als $limit Tage sind."
FunLine("Die veralteten Dateien sind in folgender Liste aufgeführt:")

$xfiles
```

Überwachen der Leistung

Mit den WMI-Leistungsklassen können Sie umfassende und genaue Leistungsinformationen abrufen. Die WMI-Klasse *Win32_perfrawdata_perfdisk_logicaldisk* ermöglicht den Zugriff auf die gleichen Leistungsinformationen, die auch über den Systemmonitor verfügbar sind. Sie können auf alle im Systemmonitor angezeigten Klassen mit einem Skript zugreifen (Abbildung 7.9). Der Vorteil des Zugriffs auf die Informationen über ein Skript anstatt der Microsoft-Verwaltungskonsole ist, dass Sie bestimmte Eigenschaften einfacher finden und die entsprechenden Aktionen ausführen können.

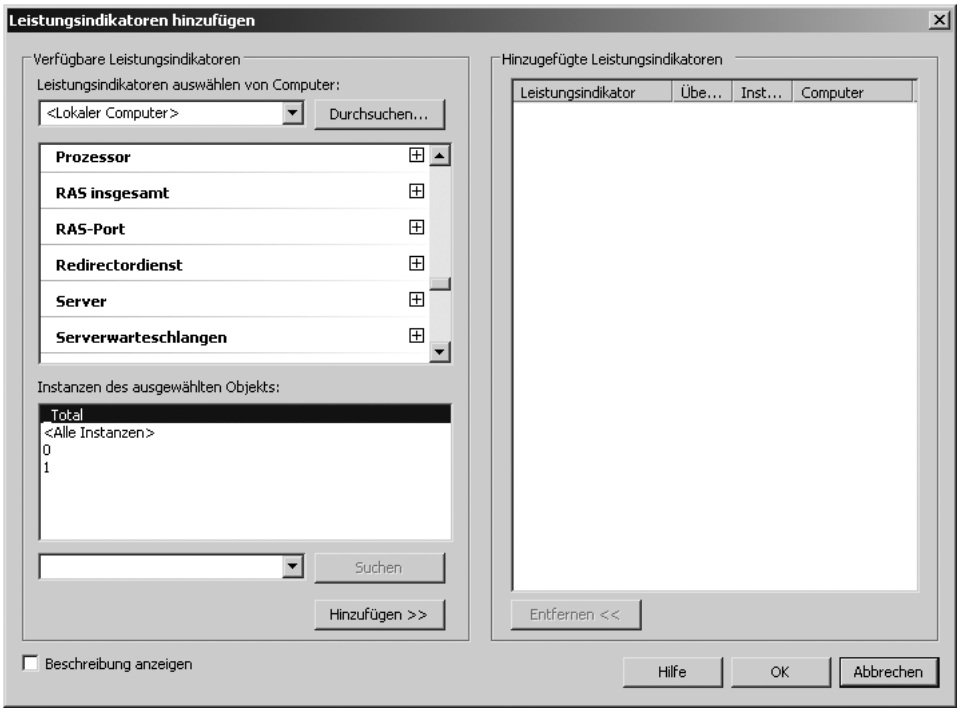


Abbildung 7.9 Die Leistungsindikatorklassen im Systemmonitor

Verwenden von Leistungsindikatorklassen

Auf einem Windows Vista-Computer sind 86 nicht berechnete sowie 87 berechnete Leistungsindikatorklassen verfügbar. Der Unterschied zwischen berechneten und nicht berechneten Klassen besteht darin, dass die Durchschnittsbestimmung in die berechneten Klassen bereits integriert ist, wohingegen die nicht berechneten Klassen einen aktuellen Snapshot bereitstellen. Die Arbeit mit den nicht berechneten Klassen ist deshalb etwas schwieriger. Um aussagekräftige Daten zusammenzustellen, benötigen Sie mehrere Datenpunkte.

Da mehr als 170 WMI-Leistungsklassen verfügbar sind, ist es schwierig, die für ein Skript erforderlichen Klassen auszuwählen. Um diese Aufgabe zu vereinfachen, können Sie das Skript *ListPerformanceCounterClasses.ps1* verwenden, um eine Liste mit den berechneten und nicht berechneten Indikatoren anzuzeigen. Das Skript kann *lokal* und *remote* ausgeführt werden und Sie können gegebenenfalls den Namespace ändern. Das Skript *ListPerformanceCounterClasses.ps1* ist wie folgt implementiert:

ListPerformanceCounterClasses.ps1

```
Param($computer="localhost", $namespace="root\cimv2")

"Abfragen von $computer..."
"Durchsuchen von $namespace nach Leistungsklassen."
$aryClasses = "perfformatteddata","perfrawdata"
foreach($class in $aryClasses)
{
 "Aufführen der WMI-Klassen vom Typ $class ...`n"
 Get-WmiObject -List -namespace $namespace `
   -computer $computer |
 Where-Object { $_.name -match $class }
}
```

Sie können zusätzliche Parameter zum Skript *ListPerformanceCounterClasses.ps1* hinzufügen, um beispielsweise nach Klassen zu suchen, die sich auf Datenträger, das Netzwerk, TCP oder Laufwerke beziehen. Das überarbeitete Skript mit dem Namen *SearchTypePerformanceCounterClasses.ps1* ist wie folgt aufgebaut.

SearchTypePerformanceCounterClasses.ps1

```
Param($computer="localhost", $namespace="root\cimv2", $type="disk")

"Abfragen von ..."
"Durchsuchen von $namespace nach Leistungsklassen."
"Die folgenden Leistungsklassen entsprechen dem Typ $type."
$aryClasses = "perfformatteddata","perfrawdata"
foreach($class in $aryClasses)
{
 "Aufführen der WMI-Klassen vom Typ $class ...`n"
 Get-WmiObject -List -namespace $namespace `
   -computer $computer |
 Where-Object { $_.name -match $class -and $_.name -match $type}
}
```

Problembehandlung Wenn Sie das Skript *GetDiskPerformance.ps1* auf Ihrem Computer ausführen und keine Daten zurückgegeben werden, verfügen Sie möglicherweise nicht über Administratorrechte. Starten Sie Windows PowerShell und aktivieren Sie die Option **Als Administrator ausführen**. Das Gleiche gilt für alle Leistungsindikatorklassen in Windows Vista und Windows Server 2008.

Das Skript *GetDiskPerformance.ps1* veranschaulicht die Arbeit mit nicht berechneten Leistungsindikatorklassen über ein Windows PowerShell-Skript. Deklarieren Sie als Erstes mehrere Variablen. Die Variable *$numrep* legt die Anzahl der Schleifen fest, die zum Erfassen der Daten erforderlich sind. Die Variable *$sleep* legt fest, wie lange das Skript zwischen den Schleifen angehalten wird. In den übrigen Variablen, die auf *$null* festgelegt sind, werden die tatsächlichen Indikatorwerte und der Zeitstempel gespeichert. Die drei erforderlichen Codezeilen lauten:

```
$numRep = 3
$sleep = 2
$n1=$d1=$n2=$d2=$r1=$r2=$w1=$w2=$null
```

Der nächste Schritt umfasst das Abrufen mehrerer Instanzen der Leistungsindikatoren mit einer *for*-Schleife. Durchlaufen Sie die *for*-Anweisung bis der Durchlaufzähler den Wert der Variablen *$numRep* erreicht. Der Grund für die Schleife ist, dass Leistungsdaten nur im Zeitverlauf aussagekräftig sind. Codezeile:

```
for ($i=1 ; $i -le $numRep ; $i++)
```

Bewährte Vorgehensweise Viele Benutzer fragen eine WMI-Leistungsklasse ab und geben die Werte der Indikatoren aus. Diese Informationen sind ziemlich nichtssagend, da sie keinen Trend anzeigen. Die Daten reflektieren häufig einen Anstieg bestimmter Prozesse aufgrund des Skripts. Leistungsdaten sollten im Zeitverlauf überprüft werden. Sie sollten deshalb die Leistungsdaten in einer Datenbank speichern.

Die nächste Codezeile fragt mit dem Cmdlet **Get-WmiObject** die WMI-Klasse *Win32_perfrawdata_perfdisk_logicaldisk* ab. Das zurückgegebene *Management*-Objekt wird der Variablen *$wmiPerf* zugewiesen. Wählen Sie die Indikatorinstanz *_Total* unter Verwendung eines Filters aus. Diese Instanz enthält die Daten für alle logischen Datenträger. Codebeispiel:

```
$wmiPerf=Get-WmiObject -class win32_perfrawdata_perfdisk_logicaldisk `
  -Filter "name = '_Total'"
```

Stellen Sie als Nächstes den ersten Datensatz zusammen. Fragen Sie hierzu die Eigenschaften des *Management*-Objekts ab, auf das die Variable *$wmiPerf* verweist. Diese Datenpunkte sind vom WMI-Datentyp *unit64*. Sie müssen die *[double]*-Einschränkung verwenden, um diesen Wert zu speichern. Wählen Sie die drei Datenpunkte *percentIdleTime*, *percentDiskTime* und *TimeStamp_Sys100NS* aus. Die Eigenschaft *TimeStamp_Sys100NS* ist ein vom System generierter Zeitstempel. Diese Eigenschaft ist nützlich, um die Datenpunkte mit dem gleichen Zeitstempel zu synchronisieren. Der Zeitstempel wird in Einheiten von 100 Nanosekunden generiert. Codebeispiel:

```
[double]$n1 = $wmiperf.percentIdleTime
[double]$r1 = $wmiperf.percentDiskTime
[double]$d1 = $wmiperf.TimeStamp_Sys100NS
```

Als Nächstes müssen Sie die Skriptausführung für kurze Zeit anhalten, um einen weiteren Snapshot zu erstellen. Abhängig von Ihren Anforderungen können Sie die Sleep-Zeit auf einen anderen Wert festlegen. Durch das Verringern des Zeitinvalls zwischen den Zyklen wird möglicherweise ein periodisch auftretendes Problem erkannt. Durch das Erhöhen des Zeitinvalls zwischen den Zyklen erhalten Sie eine gute Trendübersicht. Die Codezeile verwendet das Cmdlet **Start-Sleep** und den in der Variablen *$sleep* gespeicherten Wert. Codebeispiel:

```
Start-Sleep -Seconds $sleep
```

Nachdem die Ausführung des Skripts angehalten wurde, wird der bereits zuvor verwendete Code ausgeführt, da die WMI-Leistungsindikatorinformationen aktualisiert werden sollen. Dieser Codeabschnitt ist bis auf die Variablen, in denen die Leistungsindikatordaten gespeichert sind, mit dem anderen Code identisch. Sie können einem neuen Datensatz den neuen Zeitstempel zuweisen.

```
$wmiPerf=Get-WmiObject -class win32_perfrawdata_perfdisk_logicaldisk `
  -Filter "name = '_Total'"
[double]$n2 = $wmiperf.percentIdleTime
[double]$r2 = $wmiperf.percentDiskTime
[double]$d2 = $wmiperf.TimeStamp_Sys100NS
```

Geben Sie eine Statusmeldung aus, die mit der Variablen *$i* die Wiederholungen nachverfolgt. Codezeile:

```
"Durchlauf $i . Verarbeitung fortsetzen bis Durchlauf $numrep ..."
```

Der letzte Schritt umfasst das Berechnen der Prozentsätze basierend auf den erfassten Daten. Verwenden Sie hierzu folgende Formeln:

```
$PercentIdleTime = (1 - (($N2 - $N1)/($D2-$D1)))*100
  "`tLeerlaufzeit des Datenträgers in Prozent: " + "{0:N2}" -f $PercentIdleTime
$PercentDiskTime = (1 - (($r2 - $r1)/($D2-$D1)))*100
  "`tAuslastungszeit des Datenträgers in Prozent:      " + "{0:N2}" -f $PercentDiskTime
```

Das vollständige Skript *GetDiskPerformance.ps1* ist im folgenden Abschnitt dargestellt. Das Skript zeigt die gleichen Informationen wie der Systemmonitor an (siehe Abbildung 7.10).

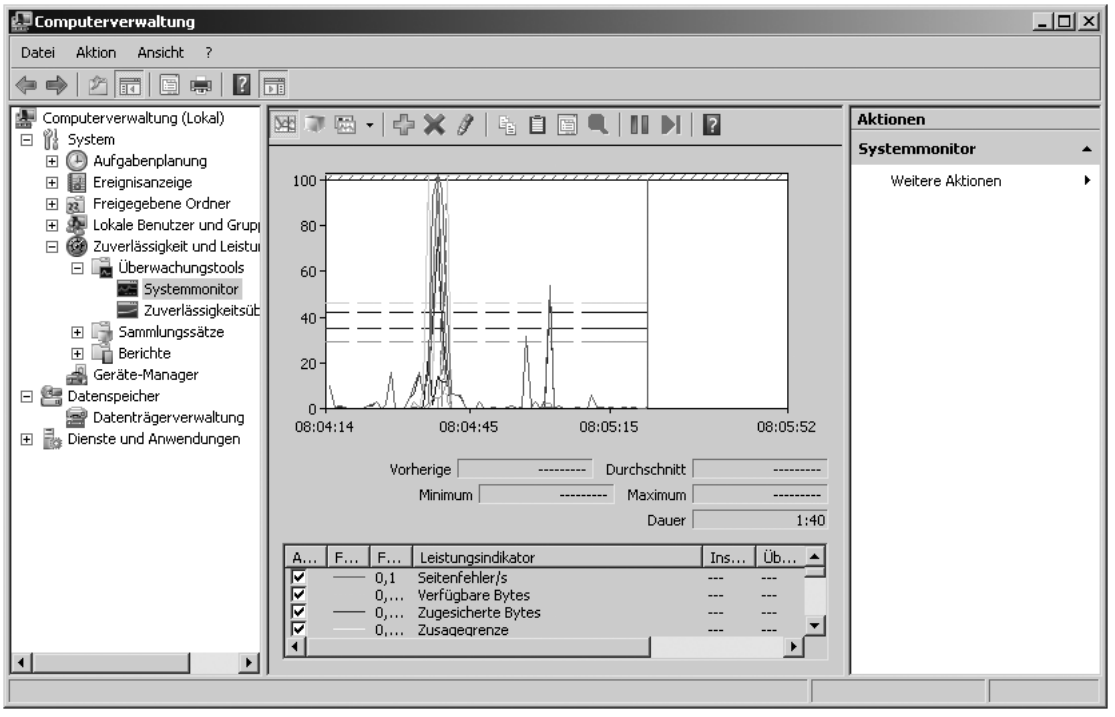


Abbildung 7.10 Datenträger-Leistungsindikatoren im Systemmonitor

GetDiskPerformance.ps1

```
$numRep = 3
$sleep = 2
$n1=$d1=$n2=$d2=$r1=$r2=$w1=$w2=$null

for ($i=1 ; $i -le $numRep ; $i++)
{
$wmiPerf=Get-WmiObject -class win32_perfrawdata_perfdisk_logicaldisk `
  -Filter "name = '_Total'"
[double]$n1 = $wmiperf.percentIdleTime
[double]$r1 = $wmiperf.percentDiskTime
[double]$d1 = $wmiperf.TimeStamp_Sys100NS

Start-Sleep -Seconds $sleep

$wmiPerf=Get-WmiObject -class win32_perfrawdata_perfdisk_logicaldisk `
```

```
  -Filter "name = '_Total'"
[double]$n2 = $wmiperf.percentIdleTime
[double]$r2 = $wmiperf.percentDiskTime
[double]$d2 = $wmiperf.TimeStamp_Sys100NS

"Durchlauf $i . Verarbeitung fortsetzen bis Durchlauf $numrep ..."

$PercentIdleTime = (1 - (($N2 - $N1)/($D2-$D1)))*100
  "`tLeerlaufzeit des Datenträgers in Prozent: " + "{0:N2}" -f $PercentIdleTime
$PercentDiskTime = (1 - (($r2 - $r1)/($D2-$D1)))*100
  "`tAuslastungszeit  des Datenträgers in Prozent:      " + "{0:N2}" -f $PercentDiskTime
}
```

Ermitteln der Ursachen von Seitenfehlern

Mit dem Skript *GetDiskPerformance.ps1* können Sie sowohl die Leerlaufzeit als auch die Auslastungszeit des Datenträgers ermitteln. Eine Ursache für Datenträgeraktivitäten sind Seitenfehler, die von verschiedenen Anwendungen ausgelöst werden. Das vorherige Skript zeigt die Trends der Datenträgeraktivitäten an, aber nicht die Ursache von Seitenfehlern. Rufen Sie im Skript *FindMaxPageFaults.ps1* mit dem Cmdlet **Get-WmiObject** alle Instanzen der WMI-Klasse *Win32_Process* ab. Fügen Sie das resultierende Objekt in das Cmdlet **Sort-Object** ein, um die Liste nach der Eigenschaft *PageFaults* zu sortieren. Setzen Sie die Pipeline im Cmdlet **Select-Object** fort. Der Vorteil dieses Cmdlets ist, dass Sie nur die fünf Prozesse abzurufen brauchen, die die meisten Seitenfehler verursachen. Sie können das Skript jedoch erweitern, um weitere Informationen zum verursachenden Prozess anzuzeigen. Das vollständige Skript *FindMaxPageFaults.ps1* basiert auf folgenden Anweisungen:

FindMaxPageFaults.ps1

```
Get-WmiObject -Class win32_process |
Sort-Object -property pagefaults|
Select-Object name, pagefaults -last 5
```

Zusammenfassung

In diesem Kapitel wurden wichtige Verwaltungsaufgaben, einschließlich der Überprüfung des Speicherplatzes, für typische Desktopcomputer in einer Unternehmensumgebung beschrieben. Es wurde sowohl das Überwachen der Speicherplatzbelegung als auch das Dokumentieren der Laufwerkkonfiguration und das Erfassen der Laufwerksinformationen in einer Access-Datenbank erklärt. Außerdem wurden Partitionen und ein Skript zum Ermitteln der Partitionsinformationen behandelt.

Sie haben logische Datenträger überprüft und die Konfigurationsinformationen von Datenträgern ausgegeben. Darüber hinaus haben Sie ein Skript verwendet, um die Konfigurationsinformationen eines Laufwerks abzufragen, und haben die Informationen zur Speicherplatzbelegung mit einem anderen Skript in einer Access-Datenbank gespeichert. Außerdem haben Sie veraltete Daten ermittelt und die entsprechenden Informationen angezeigt. Das Kapitel wurde mit dem Überprüfen der WMI-Leistungsindikatorklassen und dem Abrufen von Leistungsinformationen für Laufwerke abgeschlossen.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

KAPITEL 8

Netzwerkverwaltung

Nach Abschluss dieses Kapitels können Sie:

- Die Netzwerkeinstellungen konfigurieren
- Eine statische IP-Adresse zuweisen
- DHCP aktivieren
- Die aktuellen Windows Firewall-Einstellungen überprüfen
- Die Windows Firewall-Einstellungen konfigurieren

Auf der Begleit-CD Alle Skripts in diesem Kapitel befinden sich auf der Begleit-CD im Ordner *\Scripts\Chapter08*.

Arbeiten mit Netzwerkeinstellungen

Windows Vista und Windows Server 2008 umfassen bedeutende Änderungen in Bezug auf die Netzwerkfunktionalität, einschließlich neuer Firewalldienste und der Unterstützung von IPv6 (Internet Protocol Version 6). Diese neuen Funktionen stellen Netzwerkadministratoren jedoch auch vor neue Herausforderungen. Die Windows Management Instrumentation (WMI) umfasst viele neue Leistungsindikatorklassen für IPv6 und modifizierte Netzwerkklassen, um IPv6-Adressen zu unterstützen. Außerdem sind neue Methoden für die Arbeit mit Netzwerkkarten verfügbar. Die neuen Netzwerkfunktionen ziehen jedoch auch neue Verwaltungsaufgaben nach sich.

Überprüfen der Netzwerkeinstellungen

Die in Windows Vista und Windows Server 2008 integrierten Netzwerkfunktionen sind für die einfache Verwendung ausgelegt. Obwohl diese Funktionen die Arbeit für viele Benutzer vereinfachen, sind diese für erfahrene Netzwerkadministratoren etwas komplizierter. Sie können jedoch mit Windows PowerShell die zahlreichen Netzwerkkarten auf einem Computer einfacher verwalten. Die Konfiguration von Netzwerkkarten ist an einem Beispiel in Abbildung 8.1 dargestellt.

Das Skript *GetNetAdapterStatus.ps1* ermittelt den Status der Netzwerkkarten auf einem Computer.

Das Skript *GetNetAdapterStatus.ps1* beginnt mit einer *param*-Anweisung, die das Ausführen des Skripts für Remotecomputer ermöglicht. Wenn der Parameter **-computer** beim Ausführen des Skripts nicht angegeben wird, greift das Skript auf den lokalen Computer zu. Der Parameter **-help** zeigt die Hilfeinformationen zum Skript zusammen mit Syntaxbeispielen an. Codezeile:

```
param($computer="localhost",$help)
```

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

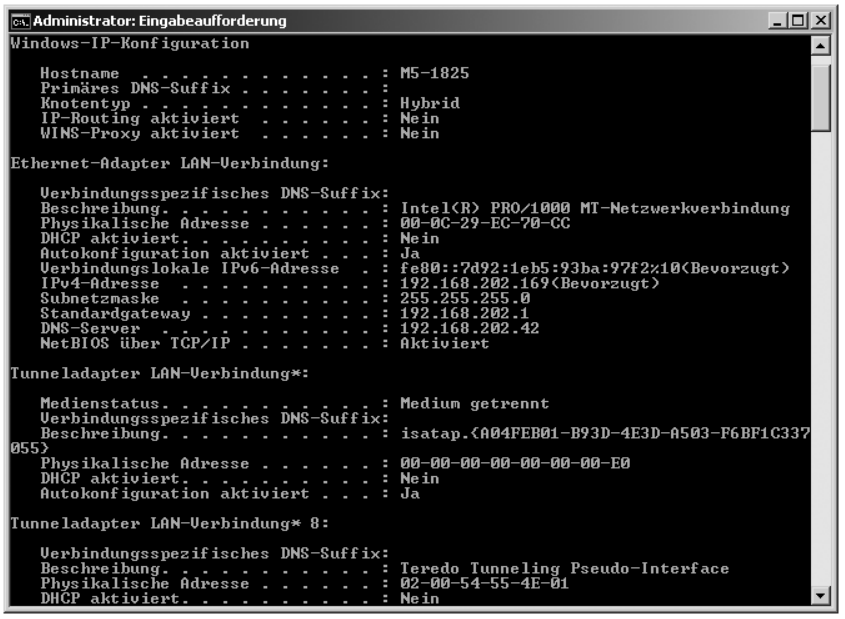


Abbildung 8.1 Die große Anzahl neuer Netzwerkkarten kompliziert die herkömmlichen Verwaltungsmethoden

Der nächste Codeabschnitt im Skript *GetNetAdapterStatus.ps1* wandelt den von der WMI-Klasse *Win32_NetworkAdapter* zurückgegebenen Statuscode in eine verständlichere Zeichenfolge um. Erstellen Sie hierzu die Funktion *funstatus*. Die Funktion *funstatus* akzeptiert einen Parameter, der angegeben wird, wenn die Funktion aus dem Skript aufgerufen wird. Der Statuscode wird von der WMI-Klasse *Win32_NetworkAdapter* zurückgegeben. Die *switch*-Anweisung in der Funktion *funstatus* wertet den Wert der Variablen *$status* aus. Der Skriptblock für die *switch*-Anweisung enthält alle Statuscodes, die für die WMI-Klasse *Win32_NetworkAdapter* definiert sind. Diese Statusmeldungen sind mit den im Netzwerk- und Freigabecenter angezeigten Statusmeldungen identisch (siehe Abbildung 8.2).

Diese Werte und Beschreibungen sind im Windows Software Development Kit (SDK) dokumentiert. Die Funktion *funstatus* ist wie folgt definiert:

```
function funStatus($status)
{
 switch($status)
  {
   0 { " Getrennt" }
   1 { " Verbinden" }
   2 { " Verbunden" }
   3 { " Trennen" }
   4 { " Hardware nicht vorhanden" }
   5 { " Hardware deaktiviert" }
   6 { " Hardwarefehler" }
   7 { " Medium getrennt" }
   8 { " Authentifizieren" }
   9 { " Authentifizierung erfolgreich" }
   10 { " Authentifizierung fehlgeschlagen" }
  }

}
```

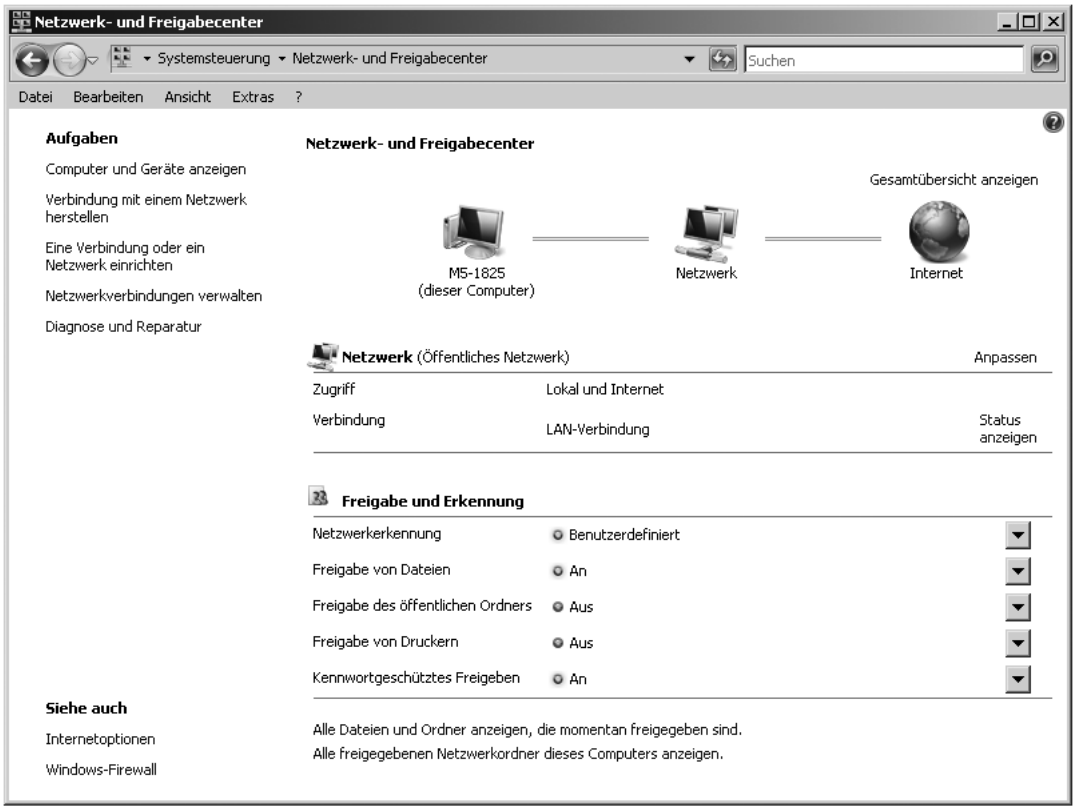


Abbildung 8.2 Netzwerkstatusmeldungen im Netzwerk- und Freigabecenter

Definieren Sie im Anschluss an die Funktion *funstatus* die Funktion *funhelp*, um eine Hilfemeldung anzuzeigen, wenn der Benutzer den Parameter **-help** angibt. Die Funktion *funhelp* verwendet für den anzuzeigenden Text eine *here*-Zeichenfolge. Die *here*-Zeichenfolge beginnt mit @" und endet mit "@. Der Vorteil von *here*-Zeichenfolgen ist, dass Sie die Formatierungsregeln ignorieren können. Sie müssen also weder das Tabzeichen **`t** noch das Zeichenumbruchzeichen **`r** oder Anführungszeichen eingeben. Die *here*-Zeichenfolge bietet keine neuen Formatierungen, aber vereinfacht die Eingabe. Weisen Sie die *here*-Zeichenfolge der Variablen *$helpText* zu. Die Zeichenfolge wird am Ende der Funktion vor dem Aufruf der *exit*-Anweisung ausgegeben. Die vollständige Funktion *funhelp* lautet:

```
function funHelp()
{
$helpText=@"v
BESCHREIBUNG:
NAME: GetNetAdapterStatus.ps1
Generiert eine Liste mit Statusinformationen für die Netzwerkkarten auf dem lokalen Computer oder einem Remotecomputer.

PARAMETER:
-computerName Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-help         Zeigt dieses Hilfethema an.

SYNTAX:
GetNetAdapterStatus.ps1 -computer MunichServer
```

```
Ermittelt die Statusinformationen für alle Netzwerkkarten auf einem Computer namens MunichServer.


GetNetAdapterStatus.ps1

Ermittelt die Statusinformationen für alle Netzwerkadapter auf dem lokalen Computer.


GetNetAdapterStatus.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Nachdem die Funktion *funhelp* abgeschlossen ist, fahren Sie mit der Funktion *funline* fort, um die Überschrift für die Netzwerkkarteneinstellungen zu unterstreichen. Die Funktion *funline* akzeptiert die Variable *$strIN*, die die Zeichenfolge enthält, die unterstrichen werden soll.

Die Funktion *funline* ermittelt die Länge der Zeichenfolge anhand der Eigenschaft *Length* und speichert den Wert in der Variablen *$num*. Verwenden Sie anschließend eine *for*-Anweisung, um in der Schleife das Gleichheitszeichen (=) wiederholt zur Variablen *$funline* hinzuzufügen, bis der Schleifenzähler den Wert der Variablen *$num* erreicht hat. Geben Sie danach die Zeichenfolge aus der Variablen *$strIN* in Gelb und die Gleichheitszeichen unter der Textzeile in Dunkelgelb aus. Die Funktion *funline* ist wie folgt implementiert:

```
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline = $funline + "=" }
    Write-Host -ForegroundColor yellow $strIN
    Write-Host -ForegroundColor darkYellow $funline
}
```

Die erste ausgeführte Codezeile überprüft, ob die Variable *$help* vorhanden ist. Die Variable *$help* ist nur vorhanden, wenn beim Starten des Skripts der entsprechende Parameter angegeben wurde. Ist dies der Fall, geben Sie eine entsprechende Meldung aus und rufen die Funktion *funhelp* auf. Die Codezeile zum Überprüfen des Parameters *$help* lautet:

```
if($help) { "Ausgeben der Hilfeinformationen ..." ; funHelp }
```

Sollte die Variable *$help* nicht vorhanden sein, führen Sie folgende Anweisungen im Code aus:

```
$objWMI=Get-WmiObject -Class win32_networkadapter -computer $computer
funline   ("Status der Netzwerkkarten auf $computer")
foreach($net in $objWMI)
{
 Write-Host "$($net.name)"
  funstatus($net.netconnectionstatus)
}
```

Das vollständige Skript *GetNetAdapterStatus.ps1* ist wie folgt aufgebaut:

GetNetAdapterStatus.ps1

```
param($computer="localhost",$help)
function funStatus($status)
{
 switch($status)
  {
   0 { " Getrennt" }
   1 { " Verbinden" }
   2 { " Verbunden" }
   3 { " Trennen" }
   4 { " Hardware nicht vorhanden" }
   5 { " Hardware deaktiviert" }
   6 { " Hardwarefehler" }
   7 { " Medium getrennt" }
   8 { " Authentifizieren" }
   9 { " Authentifizierung erfolgreich" }
   10 { " Authentifizierung fehlgeschlagen" }
  }

}

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GetNetAdapterStatus.ps1
Generiert eine Liste von Statusinformationen für die Netzwerkkarten auf dem lokalen Computer oder einem Remotecomputer.

PARAMETER:
-computerName Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-help         Zeigt dieses Hilfethema an.

SYNTAX:
GetNetAdapterStatus.ps1 -computer MunichServer

Ermittelt die Statusinformationen für alle Netzwerkkarten auf einem Computer namens MunichServer.

GetNetAdapterStatus.ps1

Ermittelt die Statusinformationen für alle Netzwerkkarten auf dem lokalen Computer.

GetNetAdapterStatus.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline = $funline + "=" }
```

```
    Write-Host -ForegroundColor yellow $strIN
    Write-Host -ForegroundColor darkYellow $funline
}

if($help) { "Ausgeben der Hilfeinformationen ..." ; funHelp }

$objWMI=Get-WmiObject -Class win32_networkadapter -computer $computer

funline   ("Statusinformationen für die Netzwerkkarten von $computer")
$objWMI
```

Arbeiten mit den Netzwerkeinstellungen

Nachdem die Statusliste der Netzwerkkarten abgerufen wurde, können Sie die Konfiguration der Netzwerkkarten abfragen. Die Netzwerkkarten sind mit den unter **Netzwerkverbindungen** in der Systemsteuerung angezeigten Netzwerkkarten identisch (siehe Abbildung 8.3).

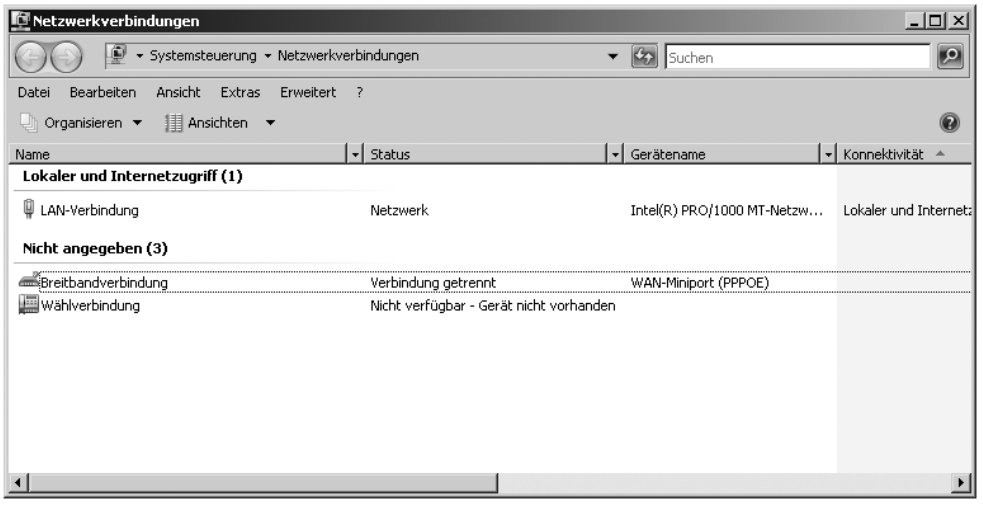


Abbildung 8.3 In der Systemsteuerung angezeigte Netzwerkkarten

Fragen Sie die Netzwerkkonfiguration mit Hilfe der WMI-Klasse *Win32_NetworkAdapterConfiguration* ab. Mit dem Skript *GetNetAdapterConfig.ps1* können Sie umfassende Informationen zu Netzwerkkarten für die Problembehandlung abrufen. Lassen Sie die Eingabe von mehreren Schlüsselwörtern zu, um jeweils bestimmte Konfigurationsinformationen zurückzugeben. Im Folgenden wird das Skript *GetNetAdapterConfig.ps1* beschrieben.

Beginnen Sie das Skript *GetNetAdapterConfig.ps1* mit einer *param*-Anweisung. In diesem Skript definieren Sie drei Parameter: **-computer**, **-query** und **-help**. Der Parameter **-computer** ist auf den Standardwert *localhost* festgelegt. Das heißt, das Skript wird für den lokalen Computer ausgeführt, wenn kein Parameter angegeben wird. Die *param*-Anweisung lautet:

```
param($computer="localhost",$query,$help)
```

Erstellen Sie nach der *param*-Anweisung die Funktion *funhelp*. Die Syntax ist der Funktion im Skript *GetNetAdapterStatus.ps1* ähnlich. Erstellen Sie eine *here*-Zeichenfolge und weisen Sie diese der Variablen *$helpText* zu. Geben Sie die *here*-Zeichenfolge in der Variablen *$helpText* am Ende der Funktion aus und beenden Sie das Skript. Beispiel:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GetNetAdapterConfig.ps1
Generiert eine Liste mit Konfigurationsinformationen für die Netzwerkkarten
auf dem lokalen Computer oder einem Remotecomputer.

PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-help    Zeigt dieses Hilfethema an.
-query    Gibt den Abfragetyp an < ip, dns, dhcp, all >.
SYNTAX:
GetNetAdapterConfig.ps1 -computerName MunichServer

Ermittelt die Standardkonfigurationsinformationen der Netzwerkkarten auf einem Computer namens MunichServer.

GetNetAdapterConfig.ps1 -computerName MunichServer -query IP

Ermittelt die Parameter IPaddress, IPsubnet, DefaultIPgateway und MACAddress
auf einem Computer namens MunichServer.

GetNetAdapterConfig.ps1 -computerName MunichServer -query DNS

Ermittelt die Parameter DNSDomain, DNSDomainSuffixSearchOrder, DNSServerSearchOrder und
DomainDNSRegistrationEnabled auf einem Computer namens MunichServer.

GetNetAdapterConfig.ps1 -computerName MunichServer -query DHCP

Ermittelt die Parameter Index, DHCPEnabled, DHCPLeaseExpires, DHCPLeaseObtained und
DHCPServer auf einem Computer namens MunichServer.

GetNetAdapterConfig.ps1 -computerName MunichServer -query ALL

Ermittelt alle Konfigurationsinformationen der Netzwerkkarten
 auf einem Computer namens MunichServer.

GetNetAdapterConfig.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Die nächste Codezeile bestimmt, ob die Hilfe ausgegeben werden soll. Verwenden Sie eine *if*-Anweisung und überprüfen Sie, ob die Variable *$help* vorhanden ist. Die Variable *$help* ist nur dann vorhanden, wenn der entsprechende Parameter in der Befehlszeile angegeben wurde. Wird die Variable *$help* gefunden, hat die *if*-Anweisung den Wert *true* und aktiviert den Codeblock, der eine Statusmeldung ausgibt und die Funktion *funHelp* aufruft. Beispiel:

```
if($help) { "Ausgeben der Hilfeinformationen..." ; funHelp }
```

Weisen Sie als Nächstes den Variablen Werte zu. Diese Variablen werden für die WMI-Abfrage verwendet. Weisen Sie außerdem der Variablen *$class* einen Wert zu, um die WMI-Abfrage auszuführen. Weisen Sie den Variablen mehrere Eigenschaftsnamen zu. Die Eigenschaften ermöglichen eine flexible Eingabe über die Befehlszeile. Die Variablen steuern die Funktionalität der Abfrage. Die Variablenzuweisungen sind im Folgenden dargestellt:

```
$class="win32_networkadapterconfiguration"
$IPproperty="IPaddress, IPsubnet, DefaultIPgateway, MACAddress"
$dnsProperty="DNSDomain, DNSDomainSuffixSearchOrder, `
 DNSServerSearchOrder, DomainDNSRegistrationEnabled"
$dhcpProperty="Index,DHCPEnabled, DHCPLeaseExpires, `
 DHCPLeaseObtained, DHCPServer"
```

Bestimmen Sie mit der *if*-Anweisung, ob der Parameter **-query** verarbeitet werden muss. Diese Anweisung ist relativ einfach. Wenn die Variable *$query* vorhanden ist, muss diese verarbeitet werden. Verwenden Sie folgendes Codesegment:

```
if($query)
```

Wenn die Variable *$query* vorhanden ist, können Sie in einer *switch*-Anweisung den zur Laufzeit zugewiesenen Wert der Variablen *$query* überprüfen. Schließen Sie für die *switch*-Anweisung den Wert der Variablen *$query* in runde Klammern ein. Öffnen Sie einen Codeblock mit geschweiften Klammern und listen Sie alle Bedingungen auf, die Sie auswerten möchten. Im vorliegenden Skript werden die Eigenschaften der WMI-Klasse in verschiedenen Variablen basierend auf der Zeichenfolge des Parameters *-query* angegeben.

Mit der *switch*-Anweisung wird letztendlich eine *select*-Anweisung für die WMI-Klasse erstellt. Beachten Sie, dass ein *DEFAULT*-Parameter vorhanden ist. Dieser Codeblock wird ausgeführt, wenn die Variable *$query* mit einem Wert initialisiert wurde, der keiner der vordefinierten Bedingungen entspricht. Dies kann beispielsweise auftreten, wenn ein Benutzer ein ungültiges Format für die Variable *$query* angibt.

Codeabschnitt:

```
switch($query)
 {
  "ip"    { $query="Select $IPproperty from $class" }
  "dns"   { $query="Select $dnsProperty from $class" }
  "dhcp"  { $query="Select $dhcpProperty from $class" }
  "all"   {
            $query = "Select * from $class" ; `
             Get-WmiObject -Query $query | format-list * ;
             exit
           }
  DEFAULT {
            $query = "Select * from $class" ; `
            Get-WmiObject -Query $query ; exit
           }
 }
}
```

Wenn Fehler auftreten, verwenden Sie die *else*-Klausel der *if*-Anweisung. Wählen Sie in der *else*-Klausel alle Eigenschaften des WMI-Objekts aus, führen Sie die Abfrage mit dem Cmdlet **Get-WmiObject** aus und senden Sie die Anweisung an die WMI-Datenbank:

```
ELSE
 {
  $query = "Select * from $class" ; `
  Get-WmiObject -Query $query ; exit
 }
```

Die letzte Anweisung im Skript *GetNetAdapterConfig.ps1* führt die WMI-Abfrage aus. Verwenden Sie hierzu das Cmdlet **Get-WmiObject** und geben Sie die Abfrage im Parameter **-query** an. Formatieren Sie die Ausgabe mit dem Cmdlet **Format-Table**. Geben Sie im Cmdlet **Format-Table** an, dass Sie ausschließlich die Parameter verwenden möchten, die dem Code im Cmdlet **Get-WmiObject** entsprechen. Codebeispiel:

```
Get-WmiObject -query $query | format-table [a-z]* -AutoSize
```

Das vollständige Skript *GetNetAdapterConfig.ps1* hat folgenden Inhalt:

GetNetAdapterConfig.ps1

```
param($computer="localhost",$query,$help)
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GetNetAdapterConfig.ps1
Generiert eine Liste mit Konfigurationsinformationen für die Netzwerkkarten
auf dem lokalen Computer oder einem Remotecomputer.

PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-help     Zeigt dieses Hilfethema an.
-query    Gibt den Abfragetyp an < ip, dns, dhcp, all >.

SYNTAX:
GetNetAdapterConfig.ps1 -computerName MunichServer

Ermittelt die Standardkonfigurationsinformationen der Netzwerkkarten
auf einem Computer namens MunichServer.

GetNetAdapterConfig.ps1 -computerName MunichServer -query IP

Ermittelt die Parameter IPaddress, IPsubnet, DefaultIPgateway und MACAddress
auf einem Computer namens MunichServer.

GetNetAdapterConfig.ps1 -computerName MunichServer -query DNS

Ermittelt die Parameter DNSDomain, DNSDomainSuffixSearchOrder, DNSServerSearchOrder und
DomainDNSRegistrationEnabled auf einem Computer namens MunichServer.

GetNetAdapterConfig.ps1 -computerName MunichServer -query DHCP

Ermittelt die Parameter Index, DHCPEnabled, DHCPLeaseExpires, DHCPLeaseObtained und
DHCPServer auf einem Computer namens MunichServer.

GetNetAdapterConfig.ps1 -computerName MunichServer -query ALL

Ermittelt alle Konfigurationsinformationen der Netzwerkkarten
auf einem Computer namens MunichServer.
```

```
GetNetAdapterConfig.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help) { "Ausgeben der Hilfeinformationen ..." ; funHelp }

$class="win32_networkadapterconfiguration"
$IPproperty="IPaddress, IPsubnet, DefaultIPgateway, MACAddress"
$dnsProperty="DNSDomain, DNSDomainSuffixSearchOrder, `
 DNSServerSearchOrder, DomainDNSRegistrationEnabled"
$dhcpProperty="Index,DHCPEnabled, DHCPLeaseExpires, `
 DHCPLeaseObtained, DHCPServer"

if($query)
{
 switch($query)
 {
  "ip"    { $query="Select $IPproperty from $class" }
  "dns"   { $query="Select $dnsProperty from $class" }
  "dhcp"  { $query="Select $dhcpProperty from $class" }
  "all"   {
            $query = "Select * from $class" ; `
             Get-WmiObject -Query $query | format-list * ;
             exit
          }
  DEFAULT {
           $query = "Select * from $class" ; `
            Get-WmiObject -Query $query ; exit
           }
 }
}
ELSE
 {
  $query = "Select * from $class" ; `
  Get-WmiObject -Query $query ; exit
 }

Get-WmiObject -query $query | format-table [a-z]* -AutoSize
```

Filtern von initialisierten Eigenschaften

Die Anzeige der Konfigurationsinformationen lässt sich verbessern, wenn nur die Eigenschaften mit tatsächlichen Werten angezeigt werden. Fragen Sie beispielsweise im Skript *NetworkAdapterConfigFiltered.ps1* die WMI-Klasse *Win32_NetworkAdapterConfiguration* ab und geben Sie nur die Eigenschaften mit einem Wert aus. Wie Sie sicher festgestellt haben, enthalten die angezeigten Eigenschaften oft keinen Wert. Im Dienstprogramm Windows Management Instrumentation Tester werden viele Eigenschaften ohne einen Wert angezeigt (siehe Abbildung 8.4).

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

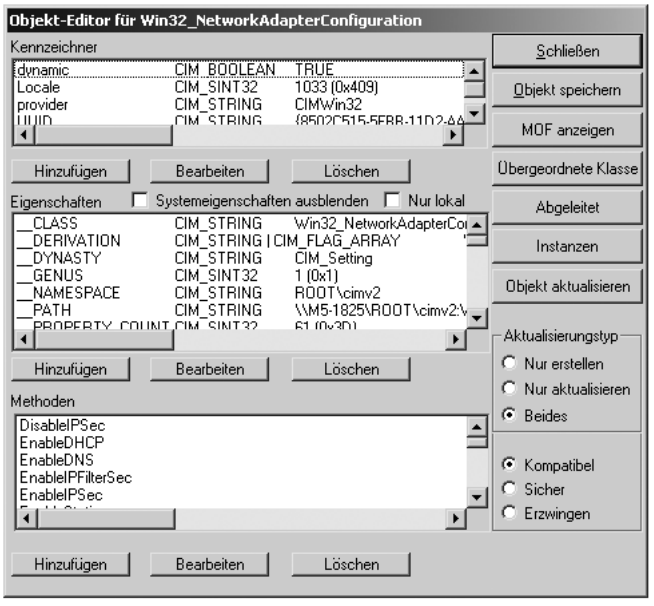


Abbildung 8.4 Der *Objekt-Editor* des Dienstprogramms Windows Management Instrumentation Tester kann Eigenschaften und ihre Werte anzeigen

Eigenschaften ohne Werte machen die Ausgabe unübersichtlich. Sie können dieses Problem jedoch beheben.

Die erste Zeile im Skript *NetworkAdapterConfigFIltered.ps1* deklariert die Funktion *funline*, die eine Zeichenfolge als Eingabe akzeptiert. Codezeile:

```
function funline ($strIN)
```

Die Funktion *funline* aus dem Skript *GetNetAdapterStatus.ps1* wurde bereits erklärt. Die Funktion *funline* ermittelt die Länge der übergebenen Zeichenfolge und verwendet anschließend eine *for*-Schleife und hängt bei jedem Schleifendurchlauf ein Gleichheitszeichen an. Die erstellte Zeichenfolge für die Zeilentrennung hat die gleiche Länge wie die an die Funktion übergebene Zeichenfolge. Geben Sie mit dem Cmdlet **Write-Host** die Zeichenfolge in Gelb und die Zeilentrennung in Dunkelgelb aus. Die Funktion *funline* ist wie folgt implementiert:

```
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline = $funline + "=" }
    Write-Host -ForegroundColor yellow `n$strIN
    Write-Host -ForegroundColor darkYellow $funline
}
```

Im nächsten Abschnitt des Skripts *NetworkAdapterConfigFiltered.ps1* werden mit dem Cmdlet **Get-WmiObject** die Informationen der WMI-Klasse *Win32_NetworkAdapterConfiguration* abgerufen. Fügen Sie das zurückgegebene Verwaltungsobjekt in das Cmdlet **ForEach-Object** ein und durchlaufen Sie die Netzwerkadapterobjekte. Geben Sie am Ende des Cmdlets **ForEach-Object** das Graviszeichen ein, um die Zeile fortzusetzen.

```
Get-WmiObject win32_networkadapterconfiguration |
  foreach-object `
```

Geben Sie im Codeblock des Cmdlets **ForEach-Object** einen Header für die WMI-Informationen aus. Der Header vereinfacht das Unterscheiden zwischen den auf dem Computer definierten Netzwerkkarten. Verwenden Sie die Eigenschaft *Caption* der WMI-Klasse *Win32_NetworkAdapterConfiguration* und rufen Sie die Funktion *funline* auf. Geben Sie eine Meldung aus, dass die Netzwerkkarte abgefragt wird. Codezeile:

```
funline ("Abfragen von: $($_.caption)")
```

Arbeiten mit Variablen und Anführungszeichen

Wenn Sie den Wert einer Variablen in Windows PowerShell anzeigen möchten, brauchen Sie nur den Namen der Variablen anzugeben:

```
PS C:\> $a = 5
PS C:\> $a
5
PS C:\>
```

Selbst wenn Sie die Variable in doppelte Anführungszeichen einschließen, wird der Wert der Variablen angezeigt:

```
PS C:\> write-host "Der Wert ist $a."
Der Wert ist 5.
PS C:\>
```

Wenn Sie den Befehl ausführen, wird der Variablenname nicht angezeigt. Sie können dieses Problem auf zwei Arten beheben. Fügen Sie einen weiteren Aufruf von *$a* zum aktuellen Code hinzu oder beenden Sie *$a* mit einem Graviszeichen. Beispiel:

```
PS C:\> write-host "Der Wert von `$a ist: $a."
Der Wert von $a ist: 5.
PS C:\>
```

Wenn Sie den Wert und den Namen der Variablen in doppelten Anführungszeichen ausgeben möchten, geben Sie ein Graviszeichen ein.

Verwenden Sie keine einfachen Anführungszeichen, da diese den umgekehrten Effekt haben. In folgendem Beispiel wird der Wert von *$a* in einfachen Anführungszeichen ausgegeben:

```
PS C:\> write-host 'Dies ist der Wert von $a.'
Dies ist der Wert von $a.
PS C:\>
```

Wenn eine Variable in einfachen Anführungszeichen ausgegeben wird, wird nur der Name der Variablen, aber nicht ihr Wert, angezeigt. Die einfachste Methode dieses Verhalten zu umgehen, ist den Code beizubehalten und *$a* außerhalb der einfachen Anführungszeichen anzugeben:

```
PS C:\> write-host 'Dies ist der Wert von $a:' $a'.'
Dies ist der Wert von $a: 5.
PS C:\>
```

Um ein weiteres Feature von Windows PowerShell bezüglich Variablen und Anführungszeichen (das automatische Auflösen) zu veranschaulichen, speichern Sie die Ergebnisse einer WMI-Abfrage in der Variablen *$a*. Beispiel:

```
PS C:\> $a=get-wmiobject -class win32_bios
PS C:\>
```

Geben Sie die BIOS-Version direkt in der Befehlszeile aus:

```
PS C:\> $a.Version
TOSHIB - 20060821
PS C:\>
```

Fügen Sie das Ergebnis in das Cmdlet **Write-Host** ein und geben Sie weitere Informationen an:

```
PS C:\> Write-Host "Dieser Computer verwendet die Biosversion: $a.Version."
Dieser Computer verwendet die Biosversion: \\M5-1875135\root\cimv2:Win32_BIOS.Name
="v3.20  ",SoftwareElementID="v3.20  ",SoftwareElementState=3,
TargetOperatingSystem=0,Version="TOSHIB - 20060821".Version
PS C:\>
```

Wie Sie sehen können, ist das Ergebnis überwältigend. Dies ist das Ergebnis des automatischen Auflösungsfeatures von Windows PowerShell. Diese Informationen sind wahrscheinlich umfassender als Sie erwartet haben. Wie können Sie die gleichen Informationen wie in Windows PowerShell anzeigen? Geben Sie hierzu ein weiteres *$* ein und schließen Sie den Befehl in Klammern ein. Beispiel:

```
PS C:\> Write-Host "Dieser Computer verwendet die Biosversion: $($a.Version)"
Dieser Computer verwendet die Biosversion: TOSHIB - 20060821
PS C:\>
```

Nachdem die Funktion *funline* ausgeführt wurde, um eine Kopfzeile für die Eigenschaftenliste auszugeben, verwenden Sie das Objekt *psobject*, um ein *System.Management.Automation.PSMemberSet*-Objekt zurückzugeben. Das Objekt *psobject* verfügt über folgende Methoden und Eigenschaften:

```
PS C:\> (get-wmiobject win32_bus).psobject | get-member |
Format-Table name, membertype -AutoSize

Name                    MemberType
----                    ----------
CompareTo                   Method
Copy                        Method
Equals                      Method
GetHashCode                 Method
GetType                     Method
get_BaseObject              Method
get_ImmediateBaseObject     Method
get_Members                 Method
get_Methods                 Method
get_Properties              Method
get_TypeNames               Method
ToString                    Method
BaseObject                Property
ImmediateBaseObject       Property
Members                   Property
Methods                   Property
Properties                Property
TypeNames                 Property
```

Sie können eine Liste der Eigenschaften, Methoden und Eigenschaften abrufen. Sie benötigen nur die Liste der Eigenschaften, da Sie den Wert überprüfen möchten, bevor Sie die Ergebnisse ausgeben. Ohne diesen zusätzlichen Schritt ist das Abfragen und Filtern von Werten schwierig.

Im Skript *NetworkAdapterConfigFiltered.ps1* wird beispielsweise überprüft, ob die Eigenschaft *Value* vorhanden ist. Hat die Eigenschaft einen Wert, ist die Eigenschaft *Value* vorhanden. Sollte die Eigenschaft *Value* jedoch nicht vorhanden sein, hat die Eigenschaft auch keinen Wert und muss nicht angezeigt werden. Codezeile:

```
If($_.value)
```

Wenn die Eigenschaft einen Wert hat, überprüfen Sie, ob der Name der Eigenschaft mit einem doppelten Unterstrich (__) beginnt. Überspringen Sie die Eigenschaft in diesem Fall, da Sie nicht an den Systemeigenschaften der WMI-Klasse interessiert sind. Um den Eigenschaftsnamen zu überprüfen, geben Sie einen regulären Ausdruck ein:

```
if ($_.name -match "__"){}
```

Tipp Um nach Übereinstimmungen von Eigenschaftsnamen zu suchen, können Sie Platzhalter oder reguläre Ausdrücke verwenden. Um Platzhalter zu verwenden, geben Sie beispielsweise den Befehl *$_.name -like "*__*"* ein. Wie Sie sehen können, ist diese Syntax etwas kompliziert. Sie können jedoch auch *not*-Operatoren verwenden und den Befehl *$_.name -notmatch "__"* eingeben. Diese Methode ist wesentlich einfacher.

Wenn die Eigenschaft keine Systemeigenschaft ist, geben Sie den Namen der Eigenschaft aus, zusammen mit zwei Tabulator-Zeichen und dem eigentlichen Wert der Eigenschaft. Codezeile:

```
Write-Host "$($_.name)`t`t $($_.value)"
```

Das vollständige Skript *NetworkAdapterConfigFiltered.ps1* hat folgenden Aufbau:

NetworkAdapterConfigFiltered.ps1

```
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline = $funline + "=" }
    Write-Host -ForegroundColor yellow `n$strIN
    Write-Host -ForegroundColor darkYellow $funline
}

Get-WmiObject win32_networkadapterconfiguration |
  foreach-object `
    {
     funline ("Abfragen von: $($_.caption)")
      $_.psobject.properties |
      foreach-object `
        {
         If($_.value)
         {
           if ($_.name -match "__"){}
           ELSE
          {
            Write-Host "$($_.name)`t`t $($_.value)"
            }
      }
   }
}
```

Konfigurieren der Netzwerkeinstellungen

Wenn auf einem Computer mehrere Netzwerkkarten installiert sind, ist die Konfiguration komplizierter. Sie müssen sicherstellen, dass Sie die richtige Netzwerkkarte konfigurieren und nicht die Netzwerkkarte deaktivieren, mit der Sie verbunden sind. In diesem Abschnitt wird die Arbeit mit mehreren Netzwerkkarten erklärt.

Erkennen mehrerer Netzwerkkarten

Ein Problem mit Windows Vista ist, dass der Drahtlosadapter die höchste Priorität hat. Dies ist für Benutzer praktisch, die ein Kabelmodem und eine Drahtlosverbindung verwenden, kann aber viele Probleme für Netzwerkadministratoren verursachen. Dieses Feature kann sogar ein Sicherheitsproblem darstellen. Wenn beispielsweise ein Benutzer in seinem Hotelzimmer nicht auf das Internet zugreifen kann, schlägt Windows Vista vor, den Drahtlosadapter zu aktivieren, um die Verbindung mit einem ungeschützten Netzwerk herzustellen (siehe Abbildung 8.5).

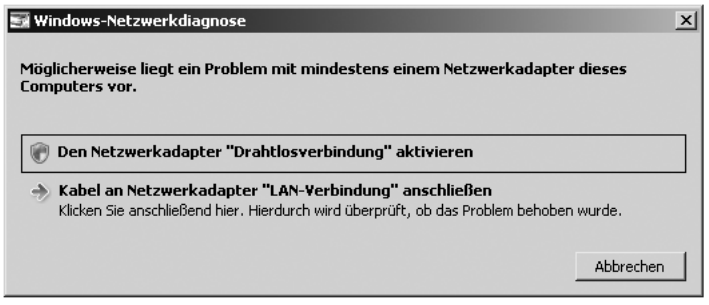


Abbildung 8.5 Beim Auftreten eines Verbindungsproblems schlägt Windows Vista vor, den Drahtlosadapter zu aktivieren

Das Skript *GetNetID.ps1* gibt den Netzwerkkartenamen, die Schnittstellennummer, den Adaptertyp und die MAC-Adresse des lokalen Computers aus. Diese Eigenschaften sind zum Überprüfen der Netzwerkkarten nützlich.

Das Skript *GetNetID.ps1* ruft mit dem Cmdlet **Get-WmiObject** die Informationen der WMI-Klasse *Win32_NetworkAdapter* ab. Formatieren Sie die Ausgabe mit dem Cmdlet **Format-Table**. Geben Sie nicht alle Eigenschaften aus, sondern nur die Eigenschaften *Name, InterfaceIndex, AdapterType* und *MacAddress*. Der Parameter **-autosize** des Cmdlets **Format-Table** verbessert die Ausgabe. Das Skript *GetNetID.ps1* ist wie folgt aufgebaut:

GetNetID.ps1

```
Get-WmiObject -Class win32_networkadapter |
format-table -Property name, interfaceIndex, `
adapterType, macAddress -autosize
```

Erfassen der Netzwerkinformationen in einem Microsoft Excel-Arbeitsblatt

Das Skript *WriteNetworkAdapterInfoToExcel.ps1* ruft die Konfigurationsinformationen der auf dem Computer installierten Netzwerkkarten ab und schreibt die Informationen in ein Excel-Arbeitsblatt, um die Informationen analysieren und permanent speichern zu können.

Geben Sie im Skript *WriteNetworkAdapterInfoToExcel.ps1* den Pfad zum Excel-Arbeitsblatt in der Variablen *$strPath* an. Erstellen Sie eine Instanz des COM-Objekts *Excel.Application*, um das Excel-Arbeitsblatt zu erstellen und zu ändern. Die folgenden beiden Codezeilen demonstrieren diese Vorgehensweise:

```
$strPath="C:\FSO\netAdapter.xls"
$objExcel=New-Object -ComObject Excel.Application
```

Legen Sie die Eigenschaft *Visible* des *Excel.Application*-Objekts, dem die Variable *$objExcel* zugewiesen ist, auf -1 (*true*) fest. Fügen Sie anschließend eine neue Arbeitsmappe zu Excel hinzu. Die folgenden beiden Codezeilen verdeutlichen diese Schritte:

```
$objExcel.Visible=-1
$WorkBook=$objExcel.Workbooks.Add()
```

Greifen Sie auf das erste Arbeitsblatt zu und halten Sie die Referenz in der Variablen *$sheet* fest. Referenzieren Sie die neue Arbeitsmappe über die Variable *$workbook* und verwenden Sie die Objektmethode *Worksheets.Item*, um das erste Arbeitsblatt zurückzugeben. Codezeile:

```
$sheet=$workbook.worksheets.item(1)
```

Deklarieren Sie in der nächsten Zeile die Variable *$x* und weisen Sie dieser den Wert 2 zu. Die Variable bewirkt, dass die Werte in die zweite Zeile des Excel-Arbeitsblattes geschrieben werden. Rufen Sie als Nächstes den Namen des Computers ab. Ermitteln Sie hierzu auf dem PS-Laufwerk den Wert der Umge-bungsvariablen *%COMPUTERNAME%*. Der zurückgegebene Wert wird in der Variablen *$computer* gespeichert. Die beiden Codezeilen lauten:

```
$x=2

$Computer = $env:computerName
```

Fragen Sie mit dem Cmdlet **Get-WmiObject** die WMI-Klasse *Win32_NetworkAdapter* ab. Das resultierende WMI-Objekt wird in der Variablen *$objWMIService* gespeichert. Codezeile:

```
$objWMIService = Get-WmiObject -class win32_NetworkAdapter `
   -computer $Computer
```

Der nächste Codeabschnitt erstellt die Spaltenüberschriften für die aus WMI abgerufenen Eigenschaften. Verwenden Sie eine *for*-Schleife, um die fett formatierten Spalten anzugeben. Um die Spaltenüberschriften fett zu formatieren, setzen Sie die Eigenschaft *Bold* der Schriftart auf *true* fest. Sie können hierzu die automatische Variable *$true* verwenden. Codebeispiel:

```
for($b=1 ; $b -le 10 ; $b++)
{$sheet.Cells.item(1,$b).font.bold=$true}
$sheet.Cells.item(1,1)=("Name der Netzwerkkarte")
$sheet.Cells.item(1,2)=("Schnittstellenindex")
$sheet.Cells.item(1,3)=("Index")
$sheet.Cells.item(1,4)=("Geräte-ID")
$sheet.Cells.item(1,5)=("Adaptertyp")
$sheet.Cells.item(1,6)=("MAC-Adresse")
$sheet.Cells.item(1,7)=("ID für Netzwerkverbindungen")
```

```
$sheet.Cells.item(1,8)=("Status für Netzwerkverbindungen")
$sheet.Cells.item(1,9)=("Netzwerkadressen")
$sheet.Cells.item(1,10)=("Permanente Adresse")
```

Verwenden Sie anschließend eine *foreach*-Anweisung, um die WMI-Objekte zu durchlaufen, rufen Sie die gewünschten Eigenschaften ab und fügen Sie diese in die entsprechenden Spalten ein. Referenzieren Sie die Zellen mit der *item*-Methode. Die *item*-Methode benötigt die *x*- und *y*-Koordinaten, um die gewünschte Zelle zu finden. Um den Prozess zu vereinfachen, geben Sie mit *$x* die aktuelle Zeile an und verweisen Sie mit der *y*-Koordinate auf die entsprechende Spalte. Codesegment:

```
ForEach ($objNet in $objWMIService)
{
    $sheet.Cells.item($x, 1)=($objNet.Name)
    $sheet.Cells.item($x, 2)=($objNet.InterfaceIndex)
    $sheet.Cells.item($x, 3)=($objNet.index)
    $sheet.Cells.item($x, 4)=($objNet.DeviceID)
    $sheet.Cells.item($x, 5)=($objNet.adapterType)
    $sheet.Cells.item($x, 6)=($objNet.MacAddress)
    $sheet.Cells.item($x,7)=($objNet.netconnectionid)
    $sheet.Cells.item($x,8)=($objNet.NetConnectionStatus)
    $sheet.Cells.item($x,9)=($objNet.NetworkAddresses)
    $sheet.Cells.item($x,10)=($objNet.PermanentAddress)
```

Überprüfen Sie, ob die Netzwerkkarte ein Ethernet-Adapter ist. Die einfachste Methode zum Bestimmen des Adaptertyps ist das Auswerten der Eigenschaft *AdapterType* der WMI-Klasse *Win32_NetworkAdapter*. Da Sie nach Ethernet-Adaptern suchen, geben Sie den *-notmatch*-Operator an:

```
    If($objNet.AdapterType -notMatch 'ethernet')
```

Wenn die Netzwerkkarte kein Ethernet-Adapter ist, ändern Sie die Schriftfarbe und formatieren die Ausgabe fett. Die beiden Codezeilen lauten:

```
$sheet.Cells.item($x,5).font.colorIndex=3
$sheet.Cells.item($x,5).font.bold=$true
```

Die letzte Funktion in diesem Abschnitt erhöht den Wert *$x* inkrementell, um die folgenden Daten in die nächste Zeile des Arbeitsblatts zu schreiben. Verwenden Sie hierzu die ++-Methode:

```
$x++
```

Nachdem das Arbeitsblatt erstellt und geöffnet wurde, müssen Sie möglicherweise die Spaltenbreite ändern, um alle Informationen anzuzeigen. Sie können zu diesem Zweck die Methode *autofit()* aufrufen, die für Spaltenobjekte in einem festgelegten Arbeitsblattbereich zur Verfügung steht. Mit der Eigenschaft *UsedRange* des Arbeitsblattobjekts können Sie den Bereich definieren. Die beiden Codezeilen lauten:

```
 $range = $sheet.usedRange
 $range.EntireColumn.AutoFit()
```

Sie müssen darüber hinaus das Arbeitsblatt speichern. Wenn die Excel-Arbeitsmappe vorhanden ist, können Sie diese löschen und als neues Arbeitsblatt anlegen. Sollte jedoch keine Excel-Arbeitsmappe vorhanden sein, können Sie diese als neue Arbeitsmappe erstellen. Hierzu ist folgender Code erforderlich:

```
IF(Test-Path $strPath)
  {
   Remove-Item $strPath
   $objExcel.ActiveWorkbook.SaveAs($strPath)
  }
```

```
ELSE
  {
   $objExcel.ActiveWorkbook.SaveAs($strPath)
  }
```

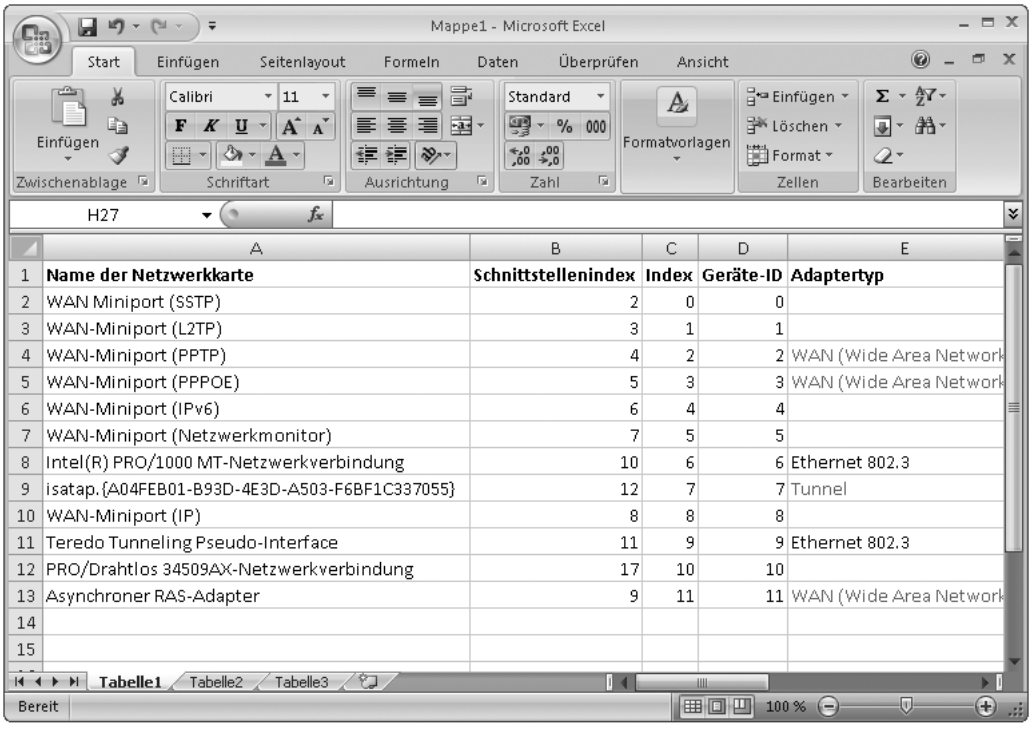
In Abbildung 8.6 ist ein Excel-Arbeitsblatt mit den Konfigurationsinformationen einer Netzwerkkarte, beispielsweise Name, Schnittstelle und MAC-Adresse, dargestellt.

Das vollständige Skript *WriteNetworkAdapterInfoToExcel.ps1* hat folgenden Aufbau:

WriteNetworkAdapterInfoToExcel.ps1

```
$strPath="C:\FSO\netAdapter.xls"
$objExcel=New-Object -ComObject Excel.Application
$objExcel.Visible=-1
$workbook=$objExcel.workbooks.Add()
$sheet=$workbook.worksheets.item(1)

$x=2

$Computer = $env:computerName
$objWMIService = Get-WmiObject -class win32_NetworkAdapter `
   -computer $Computer
for($b=1 ; $b -le 10 ; $b++)
{$sheet.Cells.item(1,$b).font.bold=$true}
$sheet.Cells.item(1,1)=("Name der Netzwerkkarte")
$sheet.Cells.item(1,2)=("Schnittstellenindex")
$sheet.Cells.item(1,3)=("Index")
$sheet.Cells.item(1,4)=("Geräte-ID")
$sheet.Cells.item(1,5)=("Adaptertyp")
$sheet.Cells.item(1,6)=("MAC-Adresse")
$sheet.Cells.item(1,7)=("ID für Netzwerkverbindungen")
$sheet.Cells.item(1,8)=("Status für Netzwerkverbindungen")
$sheet.Cells.item(1,9)=("Netzwerkadressen")
$sheet.Cells.item(1,10)=("Permanente Adresse")

ForEach ($objNet in $objWMIService)
{
    $sheet.Cells.item($x, 1)=($objNet.Name)
    $sheet.Cells.item($x, 2)=($objNet.InterfaceIndex)
    $sheet.Cells.item($x, 3)=($objNet.index)
    $sheet.Cells.item($x, 4)=($objNet.DeviceID)
    $sheet.Cells.item($x, 5)=($objNet.adapterType)
    $sheet.Cells.item($x, 6)=($objNet.MacAddress)
    $sheet.Cells.item($x,7)=($objNet.netconnectionid)
    $sheet.Cells.item($x,8)=($objNet.NetConnectionStatus)
    $sheet.Cells.item($x,9)=($objNet.NetworkAddresses)
    $sheet.Cells.item($x,10)=($objNet.PermanentAddress)

    If($objNet.AdapterType -notMatch 'ethernet')
    {
     $sheet.Cells.item($x,5).font.colorIndex=3
     $sheet.Cells.item($x,5).font.bold=$true
    }
   $x++
}
 $range = $sheet.usedRange
```

```
 $range.EntireColumn.AutoFit()

IF(Test-Path $strPath)
  {
   Remove-Item $strPath
   $objExcel.ActiveWorkbook.SaveAs($strPath)
  }
ELSE
  {
   $objExcel.ActiveWorkbook.SaveAs($strPath)
  }
```



	A	B	C	D	E
1	**Name der Netzwerkkarte**	**Schnittstellenindex**	**Index**	**Geräte-ID**	**Adaptertyp**
2	WAN Miniport (SSTP)	2	0	0	
3	WAN-Miniport (L2TP)	3	1	1	
4	WAN-Miniport (PPTP)	4	2	2	WAN (Wide Area Network
5	WAN-Miniport (PPPOE)	5	3	3	WAN (Wide Area Network
6	WAN-Miniport (IPv6)	6	4	4	
7	WAN-Miniport (Netzwerkmonitor)	7	5	5	
8	Intel(R) PRO/1000 MT-Netzwerkverbindung	10	6	6	Ethernet 802.3
9	isatap.{A04FEB01-B93D-4E3D-A503-F6BF1C337055}	12	7	7	Tunnel
10	WAN-Miniport (IP)	8	8	8	
11	Teredo Tunneling Pseudo-Interface	11	9	9	Ethernet 802.3
12	PRO/Drahtlos 3945ABG-Netzwerkverbindung	17	10	10	
13	Asynchroner RAS-Adapter	9	11	11	WAN (Wide Area Network
14					
15					

Abbildung 8.6 Das Excel-Arbeitsblatt mit den Netzwerkkarteninformationen

Erkennen verbundener Netzwerkkarten

Ein Sicherheitsrisiko in Netzwerken sind Computer, die mit mehreren Netzwerken verbunden sind. Diese mehrfach vernetzten Computer stellen eine Gefährdung dar, wenn sie ein sicheres Netzwerk mit einem unsicheren Netzwerk verbinden. Eine unter **Netzwerkverbindungen** in der Systemsteuerung angezeigte Mehrfachverbindung ist für den Netzwerkadministrator eine unwillkommene Überraschung (siehe Abbildung 8.7).

Das Skript *FindConfigurationOfConnectedAdapters.ps1* identifiziert Computer mit mehreren verbundenen Netzwerkkarten. Außerdem gibt das Skript nur Informationen zu den verbundenen Netzwerkkarten zurück. Wenn keine aktiven Verbindungen vorhanden sind, gibt das Skript keine Daten aus.

Das Skript *FindConfigurationOfConnectedAdapters.ps1* verwendet zwei WMI-Klassen. Die WMI-Klasse *Win32_NetworkAdapter* umfasst im Gegensatz zur WMI-Klasse *Win32_NetworkAdapterConfiguration* die Eigenschaft *Connected*.

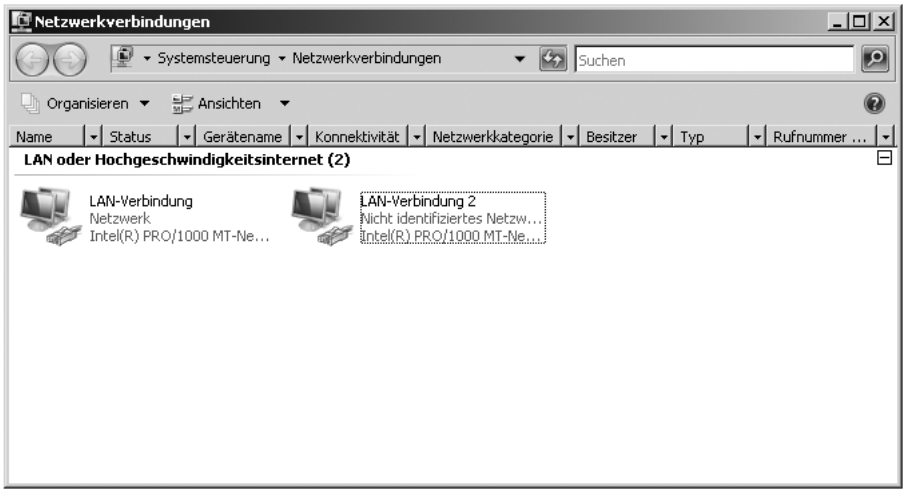


Abbildung 8.7 Unter *Netzwerkverbindungen* wird angezeigt, dass zwei Netzwerkkarten verbunden sind

Definieren Sie als Erstes zwei Variablen: Die Variable *$computer* steuert die Ausführung der WMI-Abfrage und die Variable *$connected* gibt den Wert für die Eigenschaft *NetConnectionStatus* an. Die beiden Codezeilen:

```
$computer="localhost"
$connected=2
```

Erstellen Sie mit Hilfe der Klasse *Win32_NetworkAdapter* ein Verwaltungsobjekt, das die verbundenen Netzwerkkarten repräsentiert. Um nur die verbundenen Netzwerkkarten abzurufen, geben Sie mit dem Parameter **-filter** den gewünschten Verbindungsstatus in der Variablen *$connected* an. Fügen Sie die Informationen in das Cmdlet **ForEach-Object** ein. Codebeispiel:

```
Get-WmiObject -Class win32_networkadapter -computername $computer `
-filter "netconnectionstatus = $connected" |
foreach-object `
```

Führen Sie mit dem Cmdlet **ForEach-Object** eine weitere WMI-Abfrage aus. Fragen Sie dieses Mal die WMI-Klasse *Win32_NetworkAdapterConfiguration* ab und filtern Sie nach den in der vorherigen Abfrage identifizierten Netzwerkkarten, indem Sie die *DeviceID* aus dem aktuellen Pipelineobjekt abrufen. Codeabschnitt:

```
Get-WmiObject -Class win32_networkadapterconfiguration `
   -computername $computer -filter "Index = $($_.deviceID)"
```

Das vollständige Skript *FindConfigurationOfConnectedAdapters.ps1* hat folgendes Aussehen:

FindConfigurationOfConnectedAdapters.ps1

```
$computer="localhost"
$connected=2
Get-WmiObject -Class win32_networkadapter -computername $computer `
-filter "netconnectionstatus = $connected" |
foreach-object `
```

```
{
  Get-WmiObject -Class win32_networkadapterconfiguration `
  -computername $computer -filter "Index = $($_.deviceID)"
}
```

Festlegen einer statischen IP-Adresse

Ein Netzwerkadministrator muss für Netzwerkgeräte, Arbeitsstationen oder Server oft eine statische IP-Adresse konfigurieren. Obwohl die statische IP-Adresse über die IPv4-Eigenschaftenseite einfach festgelegt werden kann, ist dies keine Lösung zum Festlegen der IP-Adresskonfiguration für zahlreiche Server (siehe Abbildung 8.8).

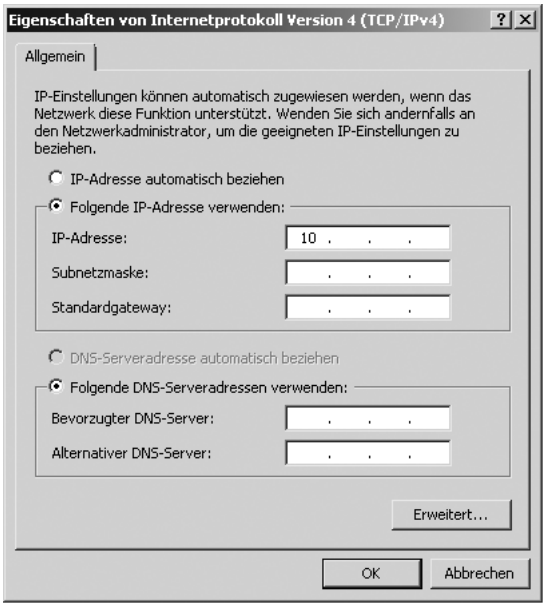


Abbildung 8.8 Festlegen der IP-Adresse über die Eigenschaften von IPv4

Die WMI-Klasse *Win32_NetworkAdapterConfiguration* umfasst 14 Methoden, die auch in Windows PowerShell verfügbar sind. Das Skript *SetStaticIP.ps1* ruft drei dieser Methoden auf.

Die erste Zeile des Skripts SetStaticIP.ps1 enthält die *param*-Anweisung, um die Eingabe von Parametern zuzulassen. Es sind mehrere Parameter definiert, aber nur einem Paramter, *($computer)*, ist ein Standardwert zugewiesen. Codezeile:

```
param($computer="localhost",$q,$ip,$sm,$dg,$dns,$help)
```

Definieren Sie die Funktion *funhelp*, da die Hilfe für ein Skript, das zahlreiche Befehlszeilenparameter umfasst, wichtig ist. Die Funktion *funhelp* verwendet, wie die anderen *funhelp*-Funktionen, eine lange *here*-Zeichenfolge. Am Ende der *here*-Zeichenfolge gibt die Funktion *funhelp* den Wert in der Variablen *$helpText* aus und beendet das Skript. Die Funktion *funhelp* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: SetStaticIP.ps1
```

```
Weist dem lokalen Computer oder einem Remotecomputer eine statische IP-Adresse zu.

PARAMETER:
-computerName Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-q            Fragt alle für IP konfigurierten Netzwerkkarten ab.
-ip           Die zu verwendende IP-Adresse.
-sm           Die zu verwendende Subnetzmaske
-dg           Das zu verwendende Standardgateway.
-dns          Der zu verwendende Dns-Server.
-help         Zeigt dieses Hilfethema an.

SYNTAX:
SetStaticIP.ps1 -q "yes" -computer MunichServer

Ermittelt alle für IP konfigurierten Netzwerkkarten auf einem Computer namens MunichServer.

SetStaticIP.ps1

Ermittelt alle für IP konfigurierten Netzwerkkarten auf dem lokalen Computer.

SetStaticIP.ps1 -ip "10.0.0.1" -sm "255.0.0.0" -dg "10.0.0.5" -dns "10.0.0.2"

Setzt die IP-Adresse auf dem lokalen Computer auf 10.0.0.1, die Subnetzmaske auf 255.0.0.0
und das Standardgateway auf 10.0.0.5 mit dem DNS-Server 10.0.0.2.

SetStaticIP.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Definieren Sie anschließend die Funktion *FunEvalRTN*. Diese Funktion wertet den von den WMI-Methoden zur Konfiguration der IP-Adressinformationen zurückgegebenen Codewert aus. Verwenden Sie in der Funktion *FunEvalRTN* eine *switch*-Anweisung, um die Eigenschaft namens *ReturnValue* des Rückgabecodes auszuwerten. Wenn *ReturnValue* den Wert 0 hat, sind keine Fehler aufgetreten. Jeder andere Wert zeigt an, dass der Befehl nicht erfolgreich ausgeführt wurde. Um die zurückgegebene Zeichenfolge informativer zu gestalten, können Sie die Variable *$strCall* mit dem Namen der aufgerufenen Methode, die den Rückgabewert generierte, in die Ausgabe einbeziehen. Die Funktion *FunEvalRTN* ist wie folgt implementiert:

```
function FunEvalRTN($rtn)
{
Switch ($rtn.returnvalue)
  {
   0 { Write-Host -foregroundcolor green "Keine Fehler für $strCall." }
   66 { Write-Host -foregroundcolor red "$strCall berichtet" `
       " ungültige Subnetzmaske." }
   70 { Write-Host -ForegroundColor red "$strCall berichtet" `
       " ungültige IP-Adresse." }
   71 { Write-Host -ForegroundColor red "$strCall berichtet" `
         " ungültiges Gateway." }
   91 { Write-Host -ForegroundColor red "$strCall berichtet" `
         " Zugriff verweigert."}
```

```
   96 { Write-Host -ForegroundColor red "$strCall berichtet" `
         " DNS-Server nicht verfügbar."}
   DEFAULT { Write-Host -ForegroundColor red "$strCall berichtet" `
             " ERROR $($rtn.returnValue)" }
  }
  $rtn=$strCall=$null
}
```

Überprüfen Sie anschließend, ob der Parameter **-help** angegeben wurde. Wenn der Parameter in der Befehlszeile angegeben wurde, ist die Variable *$help* vorhanden. Rufen Sie in diesem Fall die Funktion *funhelp* auf und beenden Sie die Skriptausführung. Fragen Sie danach den Parameter **-q** ab. Wenn der Parameter **-q** in der Befehlszeile eingegeben wurde, ist die Variable $q gesetzt. Führen Sie in diesem Fall eine WMI-Abfrage aus, die die Netzwerkkarten ermittelt, die für IP konfiguriert sind. Wenn die übrigen Befehlszeilenparameter nicht vorhanden sind, können Sie die IP-Einstellungen nicht konfigurieren und müssen die Funktion *funhelp* aufrufen. Codebeispiel:

```
if($help) { funhelp }

if($q)
{
 Get-WmiObject -Class win32_networkadapterconfiguration `
 -computer $computer -filter "ipenabled = 'true'"
exit
}

if(!$ip)  { funhelp }
if(!$sm)  { funhelp }
if(!$dg)  { funhelp }
if(!$dns) { funhelp }
```

Deklarieren Sie unter Verwendung des Tags *$global* eine globale Variable. Weisen Sie der Variablen den Wert *$null* zu. Codezeile:

```
$global:RTN = $null
```

Sie müssen nun ein Array aus einer einzigen Zahl erstellen, da die Metrik für das Gateway als Array angegeben werden muss. Sie definieren jedoch nur ein Standardgateway. Der Zeichenfolgenwert wird zwar vom Methodenaufruf akzeptiert, aber die Metrik muss ein Array sein. Stellen Sie mit der *[int32]*-Einschränkung sicher, dass die Zahl dem *int32*-Datentyp entspricht und geben Sie in der Einschränkung zwei eckige Klammern ein. Weisen Sie das Array der Variablen *$metric* zu, die Sie im Methodenaufruf an WMI übergeben. Codezeile:

```
$metric = [int32[]]1
```

Führen Sie die WMI-Abfrage aus, um die mit einer IP-Adresse konfigurierten Netzwerkkarten abzurufen, und speichern Sie das Abfrageergebnis in der Variablen *$objWMI*. Codebeispiel:

```
$objWMI = Get-WmiObject -Class win32_networkadapterconfiguration `
 -computer $computer -filter "ipenabled = 'true'"
```

Der nächste Codeabschnitt ist unkompliziert. Rufen Sie die Methoden nacheinander auf und übergeben Sie die erforderlichen Werte. Vermerken Sie die ausgeführte Aktion in einer Zeichenfolge und weisen Sie diese der Variablen *$strCall* zu. Werten Sie in der Funktion *FunEvalRTN* den von den Methodenaufrufen zurückgegebenen Code aus:

```
$RTN=$objwmi.EnableStatic($ip, $sm)
$strCall="Konfigurieren einer statischen IP-Adresse und Subnetzmaske."

FunEvalRTN($rtn)
```

```
$RTN=$objwmi.SetGateways($dg, $metric)
$strCall="Konfigurieren von Standardgateway und Metrik."
FunEvalRTN($rtn)
$RTN=$objwmi.SetDNSServerSearchOrder($dns)
$strCall="Konfigurieren der Suchreihenfolge für DNS-Server."
FunEvalRTN($rtn)
```

Das vollständige Skript *SetStaticIP.ps1* hat folgenden Inhalt:

SetStaticIP.ps1

```
param($computer="localhost",$q,$ip,$sm,$dg,$dns,$help)

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: SetStaticIP.ps1
Konfiguriert eine statische IP-Adresse auf dem lokalen Computer oder einem Remotecomputer.

PARAMETER:
-computerName Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-q            Ermittelt alle für IP konfigurierten Netzwerkkarten.
-ip           Die zu verwendende IP-Adresse.
-sm           Die zu verwendende Subnetzmaske.
-dg           Das zu verwendende Standardgateway.
-dns          Der zu verwendende DNS-Server.
-help         Zeigt dieses Hilfethema an.

SYNTAX:
SetStaticIP.ps1 -q "yes" -computer MunichServer

Ermittelt alle für IP konfigurierten Netzwerkkarten auf einem Computer namens MunichServer.

SetStaticIP.ps1

Ermittelt alle für IP konfigurierten Netzwerkkarten auf dem lokalen Computer.

SetStaticIP.ps1 -ip "10.0.0.1" -sm "255.0.0.0" -dg "10.0.0.5" -dns "10.0.0.2"

Setzt die IP-Adresse auf dem lokalen Computer auf 10.0.0.1, die Subnetzmaske auf 255.0.0.0
und das Standardgateway auf 10.0.0.5 mit dem DNS-Server 10.0.0.2.

SetStaticIP.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

function FunEvalRTN($rtn)
{
Switch ($rtn.returnvalue)
  {
   0 { Write-Host -foregroundcolor green "Keine Fehler für  $strCall." }
```

```
   66 { Write-Host -foregroundcolor red "$strCall berichtet" `
       " ungültige Subnetzmaske." }
   70 { Write-Host -ForegroundColor red "$strCall berichtet" `
       " ungültige IP-Adresse." }
   71 { Write-Host -ForegroundColor red "$strCall berichtet" `
         " ungültiges Gateway." }
   91 { Write-Host -ForegroundColor red "$strCall berichtet" `
         " Zugriff verweigert."}
   96 { Write-Host -ForegroundColor red "$strCall berichtet" `
         " DNS-Server nicht verfügbar."}
   DEFAULT { Write-Host -ForegroundColor red "$strCall berichtet" `
             " ERROR $($rtn.returnValue)" }
  }
  $rtn=$strCall=$null
}


if($help) { funhelp }

if($q)
{

 Get-WmiObject -Class win32_networkadapterconfiguration `
 -computer $computer -filter "ipenabled = 'true'"
exit
}

if(!$ip)  { funhelp }
if(!$sm)  { funhelp }
if(!$dg)  { funhelp }
if(!$dns) { funhelp }

$global:RTN = $null
$metric = [int32[]]1
$objWMI = Get-WmiObject -Class win32_networkadapterconfiguration `
 -computer $computer -filter "ipenabled = 'true'"

$RTN=$objwmi.EnableStatic($ip, $sm)
$strCall="Konfigurieren einer statischen IP-Adresse und Subnetzmaske."

FunEvalRTN($rtn)
$RTN=$objwmi.SetGateways($dg, $metric)
$strCall="Konfigurieren von Standardgateway und Metrik."
FunEvalRTN($rtn)
$RTN=$objwmi.SetDNSServerSearchOrder($dns)
$strCall="Konfigurieren der Suchreihenfolge für DNS-Server."
FunEvalRTN($rtn)
```

Aktivieren von DHCP

Das Gegenteil zum Festlegen einer statischen IP-Adresse ist das Aktivieren von DHCP (Dynamic Host Configuration Protocol). DHCP kann in den Eigenschaften von IPv4 mit nur einem Mausklick aktiviert werden (siehe Abbildung 8.9).

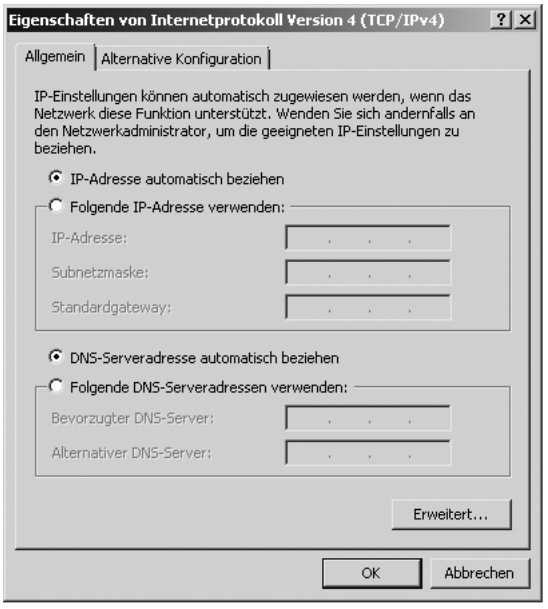


Abbildung 8.9 DHCP kann mit nur einem Mausklick aktiviert werden

Wie gehen Sie jedoch vor, um DHCP für 1000 Arbeitsstationen zu aktivieren? Das Aktivieren von DHCP auf zahlreichen Arbeitsstationen kann lange Zeit in Anspruch nehmen. DHCP ist die am häufigsten verwendete Methode zum Konfigurieren der IP-Adressinformationen. Nur wenige Unternehmen verwenden statische IP-Adressen für Arbeitsstationen. Viele große Unternehmen implementieren DHCP mit statischen Reservierungen für Serverfarmen. Sie können die DHCP-Aktivierung mit einem Skript automatisieren. Das Skript *WorkWithDHCP.ps1* zeigt den DHCP-Status an, aktiviert DHCP, gibt die von DHCP zugewiesene IP-Adresse frei und erneuert die IP-Adresse. Da das Skript dem Skript *SetStaticIP.ps1* ähnlich ist, wird es nicht ausführlich beschrieben. Die wichtigsten Abschnitte des Skripts sind jedoch im Folgenden erklärt.

Beginnen Sie das Skript mit einer *param*-Anweisung und drei Parametern. Der Parameter **-computer** ist auf den Standardwert *localhost* festgelegt. Codezeile:

```
param($computer="localhost",$action,$help)
```

Da die Funktion *funhelp* beinahe mit der Funktion im Skript *SetStaticIP.ps1* identisch ist, wird diese Funktion hier nicht ausführlich erklärt. Die Funktion *FunEvalRTN* muss ebenfalls nicht erklärt werden, da sie mit der Funktion im Skript *SetStaticIP.ps1* identisch ist.

Wenn die Variable *$help* vorhanden ist, wurde der Parameter **-help** angegeben, und Sie müssen die Funktion *funhelp* aufrufen. Deklarieren Sie die globale Variable *RTN* und legen Sie den Wert auf *$null* fest. Da das Skript die DHCP-Konfigurationsinformationen anzeigen soll, wenn keine Parameter angegeben werden, überprüfen Sie, ob die Variable *$action* existiert. Wenn die Variable nicht vorhanden ist, erstellen Sie diese und weisen Sie den Wert *q* zu, um die Abfrage auszuführen. Führen Sie die gleiche WMI-Abfrage wie im vorherigen Skript aus. Codebeispiel:

```
if($help) { funhelp }
$global:RTN = $null
if(!$action) { $action="q" }
$objWMI = Get-WmiObject -Class win32_networkadapterconfiguration `
 -computer $computer -filter "ipenabled = 'true'"
```

Verwenden Sie eine *switch*-Anweisung, um den Wert der Variablen *$action* auszuwerten. Wenn der Wert *e* ist, aktivieren Sie DHCP auf dem Zielcomputer. Nachdem Sie die Methode *enableDHCP()* aufgerufen haben, weisen Sie der Variablen *$strCall* eine Zeichenfolge zu, die an die Funktion *FunEvalRTN* übergeben wird, um zu bestimmen, ob die Methode *enableDHCP()* erfolgreich aufgerufen wurde. Codesegment:

```
Switch($action)
{
 "e" {
      $rtn = $objWMI.EnableDHCP() ;
     $strCall = "Aktivieren von DHCP" ;
     FunEvalRTN($rtn)
     }
```

Der nächste Abschnitt der *switch*-Anweisung ist dem Buchstaben *r* zugeordnet. Wenn die *switch*-Anweisung den Buchstaben *r* findet, wird die DHCP-Adresse freigegeben. Die Anweisung ruft hierzu die Methode *releaseDHCPLease()* auf. Weisen Sie der Variablen *$strCall* eine Zeichenfolge zu, um anzuzeigen, dass die Adresse freigegeben wurde, und werten Sie den zurückgegebenen Code aus, indem Sie diesen an die Funktion *FunEvalRTN* übergeben. Codeabschnitt:

```
"r" {
     $rtn = $objWMI.ReleaseDHCPLease() ;
     $strCall = "Freigeben der DHCP-Adresse" ;
     FunEvalRTN($rtn)
     }
```

Der nächste Abschnitt der *switch*-Anweisung ist den Buchstaben *rr* zugeordnet. Wenn die *switch*-Anweisung die Buchstaben *re* findet, wird die DHCP-Adresse erneuert. Weisen Sie der Variablen *$strCall* eine Zeichenfolge zu und werten Sie den zurückgegebenen Code aus:

```
 "rr" {
     $rtn = $objWMI.RenewDHCPLease() ;
     $strCall = "Freigeben und Erneuern der DHCP-Adresse" ;
     FunEvalRTN($rtn)
     }
```

Der letzte Abschnitt der *switch*-Anweisung ist am schwierigsten. Zeigen Sie den DHCP-Server an, der die IP-Adresse zugewiesen hat, und stellen Sie fest, wann die Lease ausgestellt wurde und wann diese abläuft. Das Abrufen der IP-Adresse vom DHCP-Server ist einfach, aber das Konvertieren des UTC-Datenobjekts in eine Zeitangabe ist problematisch. Um die Konvertierung auszuführen, verwenden Sie die .NET Framework-Klasse *Management.ManagementDateTimeConverter* und rufen Sie die statische Methode *toDateTime* auf. Diese .NET Framework-Klasse ist immer verfügbar. Übergeben Sie das UTC-formatierte *DateTime* -Objekt an die Methode. Codeabschnitt:

```
q" {
      "DHCP Server: $($objWMI.dhcpserver)"
      "Lease zugewiesen: " + [Management.ManagementDatetimeConverter]::`
       todatetime($objWMI.DHCPleaseObtained)
      "Lease läuft ab: " + [Management.ManagementDatetimeConverter]::`
      todatetime($objWMI.DHCPleaseExpires)
    }
```

Das vollständige Skript *WorkWithDHCP.ps1* hat folgenden Inhalt:

WorkWithDHCP.ps1

```
param($computer="localhost",$action,$help)

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: WorkWithDHCP.ps1
Konfiguriert die DHCP-Einstellungen auf dem lokalen Computer oder auf einem Remotecomputer.

PARAMETER:
-computerName Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-action <q (Abfrage) e (Aktivieren)  r (Freigeben) rr (Freigeben/Erneuern)  Die auszuführende Aktion.
-help   Zeigt dieses Hilfethema an.

SYNTAX:
WorkWithDHCP.ps1 -q "yes" -computer MunichServer

Ermittelt die DHCP-Einstellungen auf einem Computer namens MunichServer.

WorkWithDHCP.ps1 -action e

Aktiviert DHCP auf dem lokalen Computer.

WorkWithDHCP.ps1 -action r

Gibt die DHCP-Adresse auf dem lokalen Computer frei.

WorkWithDHCP.ps1 -action rr

Gibt die DHCP-Adresse auf dem lokalen Computer frei und erneuert diese.

WorkWithDHCP.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

function FunEvalRTN($rtn)
{
Switch ($rtn.returnvalue)
  {
   0 { Write-Host -foregroundcolor green "Keine Fehler für $strCall" }
   82 { Write-Host -foregroundcolor red "$strCall berichtet" `
       " Die DHCP-Lease konnte nicht erneuert werden." }
   83 { Write-Host -ForegroundColor red "$strCall berichtet" `
       " Die DHCP-Lease konnte nicht freigegeben werden." }
   91 { Write-Host -ForegroundColor red "$strCall berichtet" `
         " Zugriff verweigert"}
   DEFAULT { Write-Host -ForegroundColor red "$strCall berichtet" `
             " ERROR $($rtn.returnValue)" }
  }
```

```
  $rtn=$strCall=$null
}


if($help) { funhelp }
$global:RTN = $null
if(!$action) { $action="q" }
$objWMI = Get-WmiObject -Class win32_networkadapterconfiguration `
 -computer $computer -filter "ipenabled = 'true'"

Switch($action)
{
 "e" {
      $rtn = $objWMI.EnableDHCP() ;

     $strCall = "Aktivieren von DHCP" ;
     FunEvalRTN($rtn)
     }
  "r" {
     $rtn = $objWMI.ReleaseDHCPLease() ;
     $strCall = "Freigeben der DHCP-Adresse" ;
     FunEvalRTN($rtn)
     }
 "rr"
 {

     $rtn = $objWMI.RenewDHCPLease() ;
     $strCall = "Freigeben und Erneuern der DHCP-Adresse" ;
     FunEvalRTN($rtn)
     }
 "q" {
      "DHCP Server: $($objWMI.dhcpserver)"
      "Lease zugewiesen: " + [Management.ManagementDatetimeConverter]::`
       todatetime($objWMI.DHCPleaseObtained)
      "Lease läuft ab: " + [Management.ManagementDatetimeConverter]::`
      todatetime($objWMI.DHCPleaseExpires)
    }

}
```

Konfigurieren der Windows-Firewall

Eines der neuen Sicherheitsfeatures von Windows Vista und Windows Server 2008 ist eine wesentlich verbesserte Windows-Firewall. Die verbesserte Benutzeroberfläche der Windows-Firewall vereinfacht das Überprüfen der aktivierten bzw. deaktivierten Einstellungen.

Zum Verwalten der Windows-Firewall stehen Gruppenrichtlinien und **netsh**-Befehle zur Verfügung. Sie können also **netsh**-Befehle ausführen und Skripts erstellen, um die Verwaltung der Firewall zu vereinfachen.

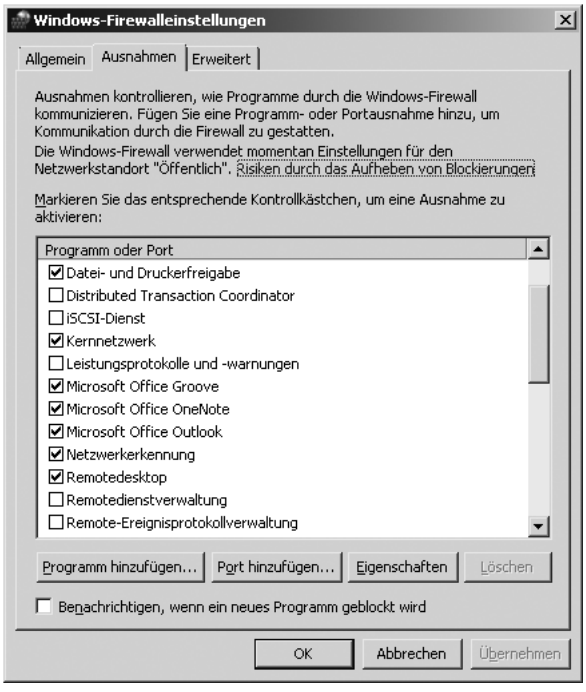


Abbildung 8.10 Die verbesserte Firewall von Windows Vista und Windows Server 2008

Überprüfen der Firewalleinstellungen

Eine Firewall ist nur nützlich, wenn Ihnen die Konfigurationseinstellungen bekannt sind. Viele Programme öffnen während der Installation für unbekannte und oft unerwünschte Dienste Ports auf der Firewall. Auch wenn die Programme nicht anzeigen, dass Firewallports geöffnet werden, können Sie feststellen, welche Ports geöffnet wurden und das Problem mit einem Skript beheben.

Beginnen Sie das Skript *ParseFWConfig.ps1* mit dem Dienstprogramm **netsh**, um die Konfigurationseinstellungen der Windows-Firewall anzuzeigen, und speichern Sie die Informationen in der Variablen *$fwCfg*. Initialisieren Sie die Variablen *$enable* und *$disable* und weisen Sie diesen den Wert *$null* zu. Die beiden Codezeilen lauten:

```
$fwCfg = netsh firewall show config
$enable=$disable=$null
```

Verwenden Sie eine *switch*-Anweisung und einen regulären Ausdruck, um das der Variablen *$fwCfg* zugewiesene Objekt abzufragen. Die erste Codezeile der *switch*-Anweisung lautet:

```
switch -regex ($fwCfg)
```

Überprüfen Sie die Daten in der Variablen *$fwCfg* und suchen Sie nach Übereinstimmungen für *Aktiv*. Wenn eine Übereinstimmung gefunden wird, fügen Sie die aktuelle Zeile zur Variablen *$enable* hinzu und danach eine neue Zeile ein. Der folgende Codeabschnitt demonstriert diese Technik:

```
"Aktiv"
    {
    $enable+=$switch.current+"`n"
```

Suchen Sie in den Daten der Variablen *$fwCfg* außerdem nach *Inaktiv*. Wenn eine Übereinstimmung gefunden wird, fügen Sie die aktuelle Zeile zur Variablen *$disable* hinzu und danach eine neue Zeile ein. Der folgende Codeabschnitt veranschaulicht diesen Schritt:

```
"Inaktiv"
    {
    $disable+=$switch.current+"`n"
```

Nachdem die Variablen zum Speichern der Konfigurationsinformationen der Firewall erstellt wurden, geben Sie die Informationen mit mehreren **Write-Host**-Anweisungen aus. Rufen Sie den Wert der Umgebungsvariablen *%COMPUTERNAME%* aus dem *PSDrive* ab und geben Sie diesen in der Kopfzeile des Berichts aus. Listen Sie sowohl die aktivierten als auch die deaktivierten Einstellungen der Firewallkonfiguration auf. Das folgende Codesegment demonstriert diese Vorgehensweise:

```
Write-Host -ForegroundColor cyan `
  "Firewallkonfiguration von $env:computername"
Write-Host -ForegroundColor green `
  "Die folgenden Ports sind aktiviert:`n"
  $enable
Write-Host -ForegroundColor red `
  "Die folgenden Ports sind deaktiviert:`n"
  $disable
```

Das vollständige Skript *ParseFWConfig.ps1* hat folgenden Aufbau:

ParseFWConfig.ps1

```
$fwCfg = netsh firewall show config
$enable=$disable=$null

switch -regex ($fwCfg)
 {
  "Aktiv"
    {
    $enable+=$switch.current+"`n"
   }
  "Inaktiv"
    {
    $disable+=$switch.current+"`n"
   }
 }

Write-Host -ForegroundColor cyan `
  "Firewallkonfiguration von $env:computername"
Write-Host -ForegroundColor green `
  "Die folgenden Features sind aktiviert:`n"
  $enable
Write-Host -ForegroundColor red `
  "Die folgenden Features sind deaktiviert:`n"
  $disable
```

Konfigurieren der Firewalleinstellungen

Nachdem Sie die aktuellen Einstellungen abgerufen haben, können Sie die Windows-Firewalleinstellungen konfigurieren. Sie müssen auf Windows Vista- und Windows Server 2008-Computern möglicherweise zwei Einstellungen aktivieren. Die erste Einstellung ist die Remoteverwaltung. Die Remoteverwaltung ist zum Ausführen von WMI-Remoteabfragen erforderlich. Die zweite Einstellung aktiviert freigegebene Ordner. Diese Konfigurationsschritte können mit den folgenden beiden Skripts ausgeführt werden.

Das Skript *EnableRemoteAdmin.ps1* öffnet einen Port auf der Windows-Firewall, um die Remoteverwaltung von Windows Vista- und Windows Server 2008-Computern zu ermöglichen. Verwenden Sie hierzu das Dienstprogramm **netsh** und dessen Dienstfunktionen. Bestätigen Sie, dass die Remoteverwaltung aktiviert werden soll. Speichern Sie die Ausgabe und suchen Sie nach *ok*. Wenn Sie das Wort finden, geben Sie die Informationen in Grün aus. Der folgende Code führt diese Schritte aus:

```
$errRTN=netsh firewall set service remoteAdmin enable
if($errRTN -match 'ok')
   { Write-Host -ForegroundColor green "Remoteverwaltung aktiviert." }
```

Der Befehl schlägt fehl, wenn Sie nicht über die erforderlichen Berechtigungen verfügen. Um Ports auf der Windows-Firewall zu öffnen, benötigen Sie Administratorrechte. Überprüfen Sie, ob die zurückgegebenen Informationen die Zeichenfolge *erfordert erhöhte Rechte* enthalten. Wenn Sie diese Zeichenfolge finden, müssen Sie die Rechte erhöhen. Der folgende Code veranschaulicht diese Schritte:

```
ELSEIF($errRTN -match 'erfordert erhöhte Rechte')
   { Write-Host -ForegroundColor red "Remoteverwaltung nicht aktiviert." `
```

Das Fehlschlagen des Skripts kann jedoch auch andere Ursachen haben. Geben Sie in diesem Fall den gesamten Fehlerbericht in der Variablen *$errRTN* aus. Codeabschnitt:

```
ELSE
    { Write-Host -ForegroundColor red "Remoteverwaltung nicht aktiviert." `
      "Die vollständige Fehlermeldung lautet: $errRTN" }
```

Das vollständige Skript *EnableRemoteAdmin.ps1* hat folgenden Inhalt:

EnableRemoteAdmin.ps1

```
$errRTN=netsh firewall set service remoteAdmin enable
if($errRTN -match 'ok')
   { Write-Host -ForegroundColor green "Remoteverwaltung aktiviert." }
   ELSEIF($errRTN -match 'erfordert erhöhte Rechte')
   { Write-Host -ForegroundColor red "Remoteverwaltung nicht aktiviert." `
     "Dieser Vorgang erfordert Administratorberechtigungen."}
   ELSE
    { Write-Host -ForegroundColor red "Remoteverwaltung nicht aktiviert." `
      "Die vollständige Fehlermeldung lautet:  $errRTN" }
```

Um freigegebene Ordner zu aktivieren, müssen Sie die den Befehl bzw. die übergebenen Parameter im Skript *EnableRemoteAdmin.ps1* ändern.

Der Befehl zum Aktivieren freigegebener Ordner umfasst einen **netsh**-Befehl, der angibt, dass der Dienst *FileAndPrint* aktiviert werden muss. Speichern Sie die vom Befehl zurückgegebenen Daten in der Variablen *$errRTN*:

```
$errRTN=netsh firewall set service FileAndPrint enable
```

Das restliche Skript ist bis auf die mit den Write-Host-Anweisungen ausgegebenen Texte mit dem Skript *EnableRemoteAdmin.ps1* identisch.

Das vollständige Skript *EnableSharedFolders.ps1* umfasst folgende Anweisungen:

EnableSharedFolders.ps1

```
$errRTN=netsh firewall set service fileAndPrint enable
if($errRTN -match 'ok')
   { Write-Host -ForegroundColor green "Freigegebene Ordner aktiviert." }
   ELSEIF($errRTN -match 'erfordert erhöhte Rechte')
   { Write-Host -ForegroundColor red "Freigegebene Ordner nicht aktiviert." `
     "Dieser Vorgang erfordert Administratorberechtigungen."}
   ELSE
    { Write-Host -ForegroundColor red "Freigegebene Ordner nicht aktiviert." `
      "Die vollständige Fehlermeldung lautet: $errRTN" }
```

Zusammenfassung

In diesem Kapitel wurden die verschiedenen Aufgaben bei der Arbeit mit Windows Vista- und Windows Server 2008-Netzwerken beschrieben. Sie haben die Einstellungen für die Elemente des Windows TCP/IP-Stacks überprüft und den Status der Netzwerkkarten mit verschiedenen Skripts abgerufen. Beispielsweise haben Sie die IDs der Netzwerkkarten ermittelt und die Informationen in einem Excel-Arbeitsblatt zusammengefasst. Im Weiteren wurde das Festlegen einer statischen IP-Adresse, das Aktivieren von DHCP und das Konfigurieren von DNS für die Namensauflösung erklärt. Das Kapitel wurde mit dem Abrufen der Windows-Firewalleinstellungen und dem Konfigurieren der Firewall für die Remoteverwaltung abgeschlossen.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

KAPITEL 9

Konfigurieren der Desktopeinstellungen

Nach Abschluss dieses Kapitels können Sie:

- Überprüfen der Desktopeinstellungen
- Konfigurieren der Bildschirmschonereinstellungen
- Verwalten von Energieeinstellungen

Auf der Begleit-CD Alle Skripts in diesem Kapitel befinden sich auf der Begleit-CD im Ordner *\Scripts\Chapter09*.

Desktopkonfiguration

Wenn Sie Windows Vista oder Windows Server 2008 bereitstellen, möchten Sie möglicherweise einige Desktopeinstellungen anpassen (vorausgesetzt Sie installieren nicht Windows Server 2008 Server Core, das keine Benutzeroberfläche hat). Diese Einstellungen umfassen Bildschirmschoner und Desktopenergieeinstellungen. Obwohl die meisten Unternehmen diese Einstellungen über Gruppenrichtlinien konfigurieren, wird die Active Directory-Funktionalität von einigen Unternehmen nur teilweise oder überhaupt nicht genutzt. Außerdem verwenden viele Arbeitsgruppen und kleinere Unternehmen nur die Standardgruppenrichtlinie. Windows PowerShell kann jedoch Ordnung ins Chaos bringen.

Konfigurieren des Bildschirmschoners

In den meisten Unternehmen haben die Mitarbeiter keine Büros, sondern Arbeitsnischen. Arbeitsnischen tragen zwar zu einer offenen Atmosphäre bei und fördern die Zusammenarbeit, sind aber andererseits ein Sicherheitsalbtraum. Wenn jeder Mitarbeiter ein eigenes Büro hat, können Sicherheitsmaßnahmen einfacher durchgesetzt werden. Beispielsweise muss ein Mitarbeiter, der sein Büro verlässt, nur die selbstschließende Tür ins Schloss fallen lassen, um die Sicherheit zu gewährleisten. Die meisten Arbeitsnischen haben jedoch keine Türen. Einer der anderen Mitarbeiter kann die Arbeitsnische betreten, um auf das System des abwesenden Kollegen zuzugreifen. Auf diese Art kann auch ein Besucher auf das System zugreifen. Aufgrund der fehlenden physischen Sicherheit ist ein gesicherter Bildschirmschoner so wichtig wie eine Maus.

Überprüfen des Bildschirmschoners

Einer der ersten Schritte, den Sie bei der Arbeit mit Desktopeinstellungen ausführen sollten, ist das Überprüfen des auf dem Computer konfigurierten Bildschirmschoners. Überprüfen Sie den Bildschirmschoner aus der Leistungsperspektive. Auf einem Server sind weder Dias von Strandszenen noch rotierende dreidimensionale Würfel mit schimmernden Oberflächen erforderlich. Das Dialogfeld zum Auswählen des Bildschirmschoners ist in Abbildung 9.1 dargestellt.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Abbildung 9.1 Auswahl des Bildschirmschoners im Dialogfeld *Bildschirmschonereinstellungen*

Berücksichtigen Sie die Sicherheitsaspekte bezüglich des Bildschirmschoners. Einige Bildschirmschoner kommunizieren mit externen Servern, um Konfigurationen zu aktualisieren oder andere Informationen zu übertragen. In einigen Unternehmen ist die Verwendung eines bestimmten Bildschirmschoners sogar vorgeschrieben. Wenn der Computer öffentlich zugänglich ist, sind Bildschirmschoner mit einer passenden Meldung erforderlich. Die IT-Abteilung muss die Bildschirmschoner bestimmter Computer häufig überprüfen. Um diese Aufgabe zu vereinfachen, können Sie das Skript *AuditScreenSaver.ps1* ausführen. Das Skript überprüft, ob der angemeldete Benutzer einen Bildschirmschoner aktiviert hat. Außerdem ermittelt das Skript den Namen des Bildschirmschoners, den Zeitüberschreitungswert und ob der Bildschirmschoner gesichert ist.

Beginnen Sie das Skript *AuditScreensaver.ps1* mit einer *param*-Anweisung und definierten Sie zwei Eingabeparameter: *$computer* und *$help*. Der Parameter *$computer* legt den Computer fest, für den das Skript ausgeführt wird. Der Parameter *-computer* hat den Standardwert *localhost*. Das bedeutet, dass das Skript standardmäßig auf dem lokalen Computer ausgeführt wird. Der Parameter *$help* signalisiert, ob die Hilfe angezeigt werden soll. Codezeile:

```
param($computer="localhost", $help)
```

Definieren Sie als Nächstes die Funktion *funline*, um für eine übergebene Zeichenfolge eine Art Unterstrich zu erzeugen. Die Funktion *funline* dient der Formatierung der ausgegebenen Informationen sowohl auf dem Bildschirm als auch wenn diese in einer Textdatei gespeichert werden. Die Funktion *funline* ermittelt zuerst die Länge der übergebenen Zeichenfolge. Die Anzahl der Gleichheitszeichen (=), die als Unterstrich dienen, wird basierend auf der Länge der Zeichenfolge festgelegt. Die Zeile mit den Gleichheitszeichen ist genauso lang wie die Zeichenfolge aus der Variablen *$strIN*. Die Trennungs-

zeichenfolge (*$funline*) wird mit einer *for*-Schleife erstellt. Wenn Sie das Gleichheitszeichen in der Variablen *$funline* verknüpfen, geben Sie die gekürzte Syntax += ein, um mit den Werten links zu beginnen und die Werte rechts hinzuzufügen. Die gekürzte Syntax hat die gleiche Bedeutung wie das Hinzufügen der Variablen zu sich selbst:

```
$funline = $funline + "="
```

Die gekürzte Syntax für diese Anweisung in der Funktion *funline* lautet:

```
$funline += "="
```

Der letzte Schritt in der Funktion *funline* besteht darin, die Zeichenfolge und den erzeugten Unterstrich mit dem Cmdlet **Write-Host** auszugeben. Die Zeichenfolge wird in Gelb angezeigt und der Unterstrich in Dunkelgelb. Die Funktion *funline* umfasst folgende Anweisungen:

```
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline += "=" }
    Write-Host -ForegroundColor yellow `n$strIN
    Write-Host -ForegroundColor darkYellow $funline
}
```

Definieren Sie als Nächstes die Funktion *funhelp*, die aufgerufen wird, wenn der Parameter *-help* im Skript angegeben wird. In der Variablen *$helpText* wird der Inhalt einer *here*-Zeichenfolge gespeichert. Die *here*-Zeichenfolge beginnt mit @" und endet mit "@. Der Vorteil von *here*-Zeichenfolgen ist, dass Sie die Regeln für Anführungszeichen ignorieren können, da der gesamte Text zwischen @" und "@ als Zeichenfolge behandelt wird. Verwenden Sie die Funktion *funhelp*, um eine Beschreibung und die Syntax des Skripts anzuzeigen. Nachdem Sie den Inhalt der *here*-Zeichenfolge mit *$helpText* ausgegeben haben, beenden Sie das Skript mit dem Befehl *exit*. Die Funktion *funhelp* hat folgendes Aussehen:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: AuditScreenSaver.ps1
Ermittelt die Bildschirmschonereinstellungen der lokalen Arbeitsstation oder eines Remotecomputers.

PARAMETER:
-computerName Gibt den Namen des Computers an, der mit diesem Skript überprüft werden soll.
-help         Gibt die Hilfeinformationen aus.

SYNTAX:
AuditScreenSaver.ps1 -computer MunichServer

 Ermittelt die Bildschirmschonereinstellungen auf einem Computer namens MunichServer.

AuditScreenSaver.ps1

 Ermittelt die Bildschirmschonereinstellungen auf dem lokalen Computer.

AuditScreenSaver.ps1 -help ?
```

```
Zeigt die Hilfethemen für dieses Skript an.

"@
$helpText
exit
}
```

Da der Parameter **-computer** einen Standardwert ausweist, müssen Sie nicht überprüfen, ob die Variable *$computer* vorhanden ist. Der Parameter *-help* verhält sich jedoch anders. Die Hilfe wird nur angezeigt, wenn das Skript mit dem Parameter *-help* aufgerufen wird. Wenn *$help* vorhanden ist, rufen Sie die Funktion *funhelp* auf. Codezeile:

```
if($help){funline("Die Hilfeinformationen werden ausgegeben ...") ; funhelp }
```

Um den Namen des angemeldeten Benutzers zu ermitteln, rufen Sie mit dem Cmdlet **Get-WmiObject** die Eigenschaft *UserName* aus der WMI-Klasse *Win32_ComputerSystem* ab. Da das Skript remote ausgeführt werden kann, müssen Sie mit dem Parameter **-computername** den Wert in der Variablen *$computer* angeben. Beispiel:

```
$username = (get-wmiobject -class win32_computersystem `
  -computername $computer).username
```

Nachdem Sie den Benutzernamen mittels WMI abgerufen haben, müssen Sie möglicherweise den Backslash (“\”) aus dem Namen entfernen. Da der Wert in der Variablen *$username* gespeichert ist, können Sie den Backslash mit Zeichenfolgenmethoden suchen.

Zeichenfolgenmethoden

Zeichenfolgenmethoden werden von der Microsoft .NET Framework-Klasse *system.string* bereitgestellt und sind bei der Textverarbeitung hilfreich. Wenn Sie beispielsweise eine Zeichenfolge in einer Variablen speichern, können Sie auf die Zeichenfolgenmethoden zugreifen, um die Textinformationen zu bearbeiten.

Beginnen Sie das Skript *StringMethods.ps1* mit einer Zeichenfolge. Konvertieren Sie mit der Methode *toUpper()* alle Zeichen in Großbuchstaben. Geben Sie den Wert aus und rufen Sie die Methode *toLower()* auf. Ersetzen Sie mit der Methode *replace()* die Zahl 1 durch das Wort *eine*. Geben Sie das Ergebnis aus. Das Skript *StringMethods.ps1*:

StringMethods.ps1

```
$a="Dies ist 1 Zeichenfolge."
$a=$a.toUpper()
$a
$a=$a.ToLower()
$a
$a=$a.replace("1","eine")
$a
```

Wenn Sie das Cmdlet **Get-Member** für die Variable *$a* im Skript *StringMethods.ps1* ausführen, werden Sie feststellen, dass für die .NET Framework-Klasse *system.string* 35 Methoden verfügbar sind:

Clone	CompareTo	Contains
CopyTo	EndsWith	Equals
get_Chars	get_Length	GetEnumerator
GetHashCode	GetType	GetTypeCode
IndexOf	IndexOfAny	Insert
IsNormalized	LastIndexOf	LastIndexOfAny
Normalize	PadLeft	PadRight
Remove	Replace	Split
StartsWith	Substring	ToCharArray
ToLower	ToLowerInvariant	ToString
ToUpper	ToUpperInvariant	Trim
TrimEnd	TrimStart	

Die erste im Skript verwendete Methode ist *indexof()*. Die Methode *index()* durchsucht eine Zeichenfolge und gibt eine Zahl zurück, die die Stelle angibt, an der eine Übereinstimmung gefunden wurde. Rufen Sie anschließend mit der Methode *substring()* einen bestimmten Textabschnitt aus der Zeichenfolge ab. Das Skript *AuditScreenSaver.ps1* gibt den Text nach dem Backslash zurück. Die beiden Codezeilen lauten:

```
$index=$username.indexof("\")
$username=$username.substring($index+1)
```

Danach können Sie mit dem Cmdlet **Get-WmiObject** die WMI-Klasse *Win32_Desktop* abfragen. Geben Sie mit dem Parameter **-computername** einen Remotecomputer an. Der Wert für den Parameter **-computername** entspricht dem Wert im Parameter *$computer*, der beim Start des Skripts angegeben wird. Geben Sie den Parameter **-filter** an, um die Anzahl der zurückgegebenen Objekte auf den angemeldeten Benutzer zu beschränken. Sie müssen den in der Variablen *$username* gespeicherten Benutzernamen in Anführungszeichen an WMI übergeben. Geben Sie hierzu vor den Anführungszeichen ein Gravis-Zeichen ein. Um die korrekte Verarbeitung der Variablen *$username* sicherzustellen, sollten Sie ein weiteres Dollar-Zeichen in Klammern eingeben. Übergeben Sie danach das resultierende *psobject*-Objekt an das Cmdlet **Select-Object** ein. Der folgende Code führt diese Schritte aus:

```
$screensaver = Get-WmiObject -Class win32_desktop `
  -computername $computer -filter "name like `"%$($username)`"" |
```

Wählen Sie anschließend mit dem Cmdlet **Select-Object** alle Eigenschaften, die mit dem Wort *screen* beginnen, und die Eigenschaft *Name* aus. Schreiben Sie diese Informationen zurück in die Variable *$screensaver*. Codezeile:

```
Select-Object -Property screen*, name
```

Nachdem Sie das benutzerdefinierte Objekt erstellt und in der Variablen *$screensaver* gespeichert haben, geben Sie mit der Funktion *funline* einen Header für den Bericht aus. Wählen Sie die Eigenschaft *Name* im Objekt *$screensaver* aus, die den Namen des angegebenen Benutzers enthält. Codezeile:

```
funline("Bildschirmschonereinstellungen für $($screensaver.name)")
```

Verwenden Sie eine *if*-Anweisung, um die Eigenschaft *ScreenSaverActive* auszuwerten. Beachten Sie, dass die Eigenschaft für den Bildschirmschoner nicht aktualisiert wird, wenn der Bildschirmschoner deaktiviert ist und kein Neustart ausgeführt wird (siehe Abbildung 9.2). Dies ist auf die Methode zurückzuführen, mit der der aktuelle Konfigurationsregistrierungsschlüssel beim Start erstellt wird.

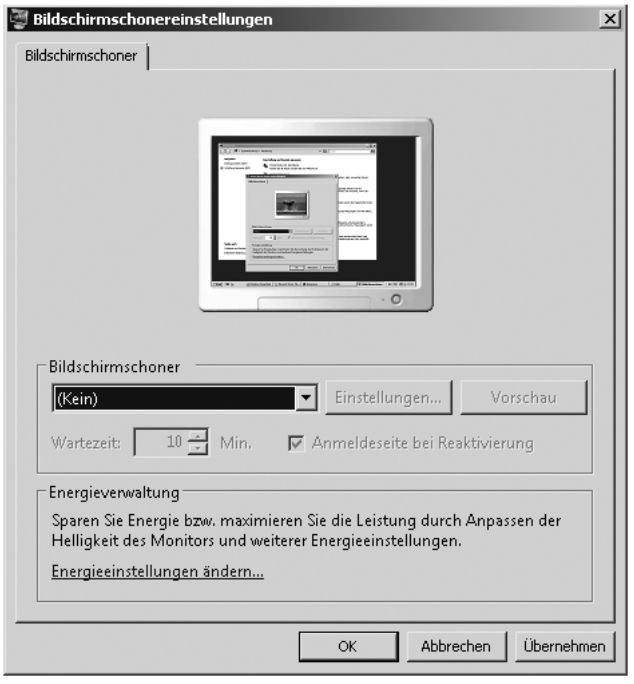


Abbildung 9.2 Kein Bildschirmschoner ausgewählt

Wenn die Eigenschaft *ScreenSaverActive* den Wert *true* hat, geben Sie die ausführbare Datei und den für den Bildschirmschoner konfigurierten Zeitüberschreitungswert aus. Der Zeitüberschreitungswert wird in Sekunden angegeben. Der Standardwert von 10 Minuten wird deshalb als Wert 600 angezeigt. Codesegment:

```
if($screensaver.ScreenSaverActive -eq "true")
 {
  Write-Host "Verwendeter Bildschirmschoner: $($screensaver.screensaverExecutable)"
  Write-Host "Bildschirmschoner gesichert: $($screensaver.ScreenSaverSecure)"
  Write-Host "Zeitüberschreitungswert: $($screensaver.ScreenSaverTimeout)"
 }
```

Wenn kein Bildschirmschoner konfiguriert ist, verwenden Sie die *else*-Klausel und geben Sie eine entsprechende Meldung aus. Verwenden Sie hierzu folgenden Code:

```
ELSE
 { Write-Host "$($screensaver.name) verfügt über keinen Bildschirmschoner."}
```

Das vollständige Skript *AuditScreenSaver.ps1* hat folgenden Aufbau:

AuditScreenSaver.ps1

```
param($computer="localhost", $help)
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline = $funline + "=" }
    Write-Host -ForegroundColor yellow `n$strIN
    Write-Host -ForegroundColor darkYellow $funline
}

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: AuditScreenSaver.ps1
Ermittelt die Bildschirmschonereinstellungen der lokalen Arbeitsstation oder eines Remotecomputers.

PARAMETERS:
-computerName Gibt den Namen des Computers an, der mit diesem Skript überprüft werden soll.
-help         Gibt die Hilfeinformationen aus.

SYNTAX:
AuditScreenSaver.ps1 -computer MunichServer

Ermittelt die Bildschirmschonereinstellungen auf einem Computer namens MunichServer.

AuditScreenSaver.ps1

Ermittelt die Bildschirmschonereinstellungen auf dem lokalen Computer.

AuditScreenSaver.ps1 -help ?

Zeigt die Hilfethemen für dieses Skript an.

"@
$helpText
exit
}

if($help){funline("Die Hilfeinformationen werden ausgegeben ...") ; funhelp }
```

```
$username = (get-wmiobject -class win32_computersystem `
  -computername $computer).username
$index=$username.indexof("\")
$username=$username.substring($index+1)

$screensaver = Get-WmiObject -Class win32_desktop `
  -computername $computer -filter "name like `"%$($username)`"" |
Select-Object -Property screen*, name

funline("Bildschirmschonereinstellungen für  $($screensaver.name)")
if($screensaver.ScreenSaverActive -eq "true")
 {
  Write-Host "Verwendeter Bildschirmschoner: $($screensaver.screensaverExecutable)"
  Write-Host "Bildschirmschoner gesichert: $($screensaver.ScreenSaverSecure)"
  Write-Host "Zeitüberschreitungswert: $($screensaver.ScreenSaverTimeout)"
 }
ELSE
 { Write-Host "$($screensaver.name) verwendet keinen Bildschirmschoner."}
```

Auflisten von Eigenschaften mit Werten

Ein Problem bei der Verwendung von WMI zum Erfassen von Informationen in Datenbanken, Arbeitsblättern, Berichten sowie bei der Ausgabe in der Konsole ist die große Anzahl der Eigenschaften, die keine Informationen zurückgeben. Diese leeren Zeilen machen die Ausgabe unübersichtlich (siehe Abbildung 9.3).

Sie können entweder jede Eigenschaft mit einem Wert explizit angeben oder die leeren Werte ignorieren. Es ist jedoch auch möglich, die leeren Werte mit dem Skript *ReportDesktopSettings.ps1* herauszufiltern.

Das Skript *ReportDesktopSettings.ps1* beginnt mit einer *param*-Anweisung, um zwei Eingabeparameter zu definieren: *-computer* und *-help.* Weisen Sie nur der Variablen *$computer* einen Standardwert zu:

```
param($computer="localhost", $help)
```

Definieren Sie die Funktion *funline* mit dem Funktionsparameter *$strIN*. Diese Funktion dient der Ausgabe einer Trennungszeile zwischen einer Textzeile und dem vom Skript zurückgegebenen Wert. Die Funktion bestimmt die Länge der Zeichenfolge und erstellt eine Zeile gleicher Länge mit Gleichheitszeichen (Sie können auch ein anderes Zeichen verwenden). Die Funktion *funline* hat folgendes Aussehen:

```
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline = $funline + "=" }
    Write-Host -ForegroundColor yellow `n$strIN
    Write-Host -ForegroundColor darkYellow $funline
}
```

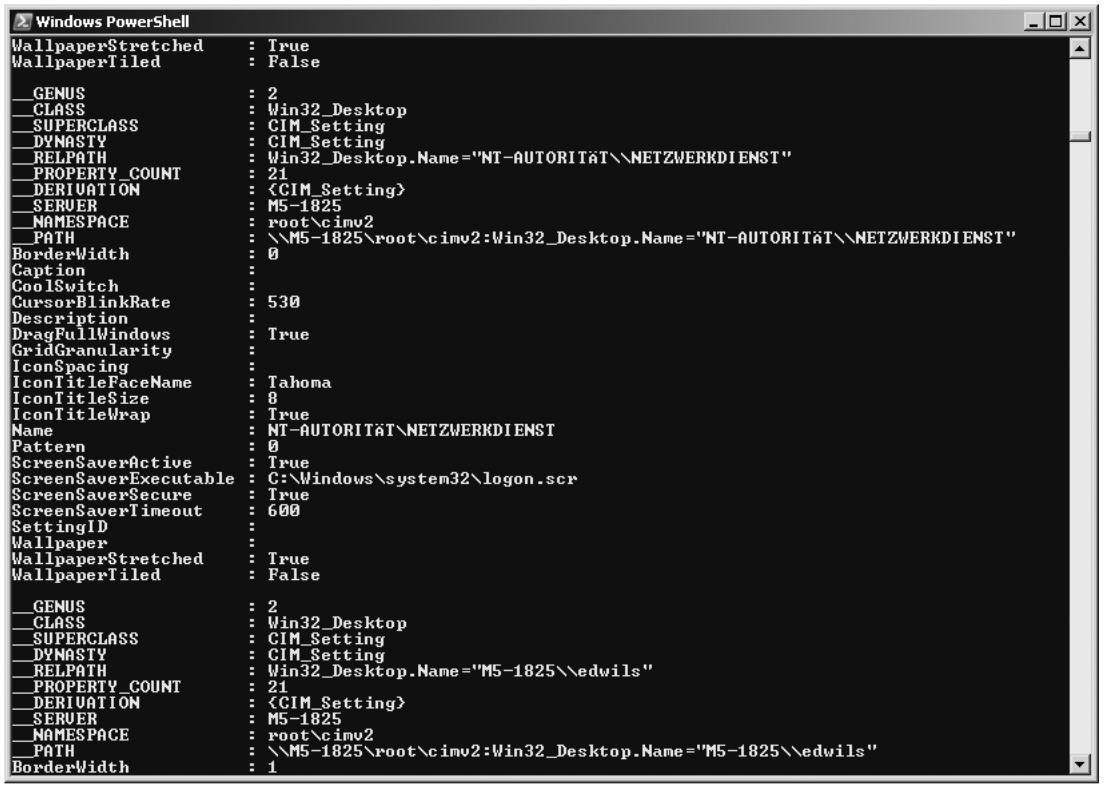


Abbildung 9.3 Leere Zeilen machen die Ausgabe unübersichtlich

Definieren Sie anschließend die Funktion *funhelp*. Die Funktion besteht aus einer langen *here*-Zeichenfolge, die nur angezeigt wird, wenn das Skript mit dem Parameter *-help* ausgeführt wird. Definieren Sie in der *here*-Zeichenfolge drei Abschnitte: *Beschreibung*, *Parameter* und *Syntax*. Nachdem Sie die *here*-Zeichenfolge erstellt haben, können Sie diese der Variablen *$helpText* zuweisen. Die Funktion *funhelp* zeigt lediglich den Inhalt der Variablen *$helpText* an und wird danach beendet. Die Funktion *funhelp* ist folgendermaßen implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ReportDesktopSettings.ps1
Ermittelt die Desktopeinstellungen der lokalen Arbeitsstation oder eines Remotecomputers.

PARAMETER:
-computerName Gibt den Namen des Computers an, der mit diesem Skript überprüft werden soll.
-help         Gibt die Hilfeinformationen aus.

SYNTAX:
ReportDesktopSettings.ps1-computer MunichServer

Ermittelt die Desktopeinstellungen auf einem Computer namens MunichServer.
```

```
ReportDesktopSettings.ps1

Ermittelt die Desktopeinstellungen auf dem lokalen Computer.

ReportDesktopSettings.ps1-help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Nachdem Sie die Funktionen erstellt haben, überprüfen Sie, ob die Variable *$help* vorhanden ist. Gibt es die Variable, wurde das Skript mit dem Parameter **-help** ausgeführt. Rufen Sie im Codeblock die Funktion *funline* auf und geben Sie eine Statuszeichenfolge ein. Beenden Sie den Befehl mit einem Semikolon und rufen Sie die Funktion *funhelp* auf. Codesegment:

```
if($help){ funline("Ausgeben der Hilfeinformationen ...") ; funhelp }
```

Rufen Sie mit dem Cmdlet **Get-WmiObject** den Namen des aktuellen Benutzers unter Verwendung der WMI-Klasse *Win32_ComputerSystem* ab. Stellen Sie die Verbindung mit dem Computer her, der im Parameter **-computer** angegeben und in der Variablen *$computer* gespeichert ist. Dieser Codeabschnitt ist wie folgt implementiert:

```
$currentUser = (Get-WmiObject -class win32_computersystem `
 -computername $computer).username
```

Fragen Sie mit dem Cmdlet **Get-WmiObject** die WMI-Klasse *Win32_Desktop* ab. Geben Sie im Parameter **-computername** des Cmdlets **Get-WmiObject** den Wert der Variablen *$computer* an. Übergeben Sie das resultierende Verwaltungsobjekt an das Cmdlet **Where-Object**. Überprüfen Sie danach im Codeblock, ob die Eigenschaft *Name* der Klasse *Win32_Desktop* mit dem Namen in der Variablen *$currentUser* übereinstimmt. Wenn die Namen identisch sind, verarbeiten Sie das Objekt. Dieser Codeabschnitt umfasst die folgenden Anweisungen:

```
Get-WmiObject -Class win32_desktop -computername $computer |
Where-Object { $_.name -Eq $currentUser } |
```

Verwenden Sie das Cmdlet **ForEach-Object**, um das von der Desktopabfrage zurückgegebene Objekt zu durchlaufen. Geben Sie mit der Funktion *funline* eine Überschrift für die Ausgabe aus. Rufen Sie mit *psobject* eine Liste der für das WMI-Objekt definierten Eigenschaften ab. (Sie können für diesen Zweck auch *psbase* verwenden.) Rufen Sie anschließend eine Liste mit allen Eigenschaften der WMI-Klasse über die Eigenschaft *Properties* ab. Fügen Sie die Eigenschaften in den nächsten Skriptabschnitt ein:

```
foreach-object `
    {  funline("Desktopeinstellungen für $($currentUser)")
      $_.psobject.properties |
```

Durchlaufen Sie die Eigenschaften mit dem Cmdlet **ForEach-Object**. Wenn die Eigenschaft in der aktuellen Pipeline einen Wert aufweist, sollten Sie überprüfen, ob der Name mit zwei Unterstrichen beginnt. Ignorieren Sie den Namen, wenn dies der Fall ist, denn derartige Namen weisen auf Systemeigenschaften hin. Wenn der Name nicht mit zwei Unterstrichen beginnt, handelt es sich nicht um eine Systemeigenschaft.

Geben Sie für alle Eigenschaften, die keine Systemeigenschaften sind, den jeweiligen Namen aus, geben Sie zweimal das Tabulatorzeichen aus und geben Sie dann den Wert aus. Der folgende Codeabschnitt führt diese Schritte aus:

```
foreach-object `
        {
         If($_.value)
             {
               if ($_.name -match "__"){}
             ELSE
           {
             Write-Host "$($_.name)`t`t $($_.value)"
           }
        }
       }
     }
```

Das vollständige Skript *ReportDesktopSettings.ps1* hat folgenden Aufbau:

ReportDesktopSettings.ps1

```
param($computer="localhost", $help)
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline = $funline + "=" }
    Write-Host -ForegroundColor yellow `n$strIN
    Write-Host -ForegroundColor darkYellow $funline
}

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ReportDesktopSettings.ps1
Ermittelt die Desktopeinstellungen der lokalen Arbeitsstation oder eines Remotecomputers.

PARAMETER:
-computerName Gibt den Namen des Computers an, der mit diesem Skript überprüft werden soll.
-help         Gibt die Hilfeinformationen aus.

SYNTAX:
ReportDesktopSettings.ps1-computer MunichServer

Ermittelt die Desktopeinstellungen auf einem Computer namens MunichServer.

ReportDesktopSettings.ps1

Ermittelt die Desktopeinstellungen auf dem lokalen Computer.

ReportDesktopSettings.ps1-help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help){ funline("Ausgeben der Hilfeinformationen ...") ; funhelp }
```

```
$currentUser = (Get-WmiObject -class win32_computersystem `
 -computername $computer).username

Get-WmiObject -Class win32_desktop -computername $computer |
Where-Object { $_.name -Eq $currentUser } |
foreach-object `
    {  funline("Desktopeinstellungen für $($currentUser)")
      $_.psobject.properties |
      foreach-object `
        {
         If($_.value)
         {
           if ($_.name -match "__"){}
             ELSE
          {
            Write-Host "$($_.name)`t`t $($_.value)"
            }
        }
       }
     }
```

Überprüfen der Sicherheitseinstellungen für Bildschirmschoner

In den meisten Fällen ist es unwichtig, welchen Bildschirmschoner ein Benutzer aktiviert hat. Sie müssen lediglich sicherstellen, dass der Bildschirmschoner gesichert ist. Ein gesicherter Bildschirmschoner sperrt den Computer nach einer bestimmten Inaktivitätsdauer. In Abbildung 9.4 ist ein Bildschirmschoner mit aktivierter Sicherheitseinstellung dargestellt.

Abbildung 9.4 Gesicherte Bildschirmschoner fordern zur Eingabe der Anmeldeinformationen auf

Die Inaktivitätsdauer, die oft von der Sicherheitsrichtlinie eines Unternehmens bestimmt wird, beträgt normalerweise 1 bis 5 Minuten. Um diese Einstellungen zu überprüfen, speichern Sie die Informationen in einer Datenbank. Dies ermöglicht Ihnen, die Informationen zu analysieren und Berichte mit der Prozentanzahl der Benutzer zu erstellen, die sich an die Richtlinie halten.

Hinweis Warum müssen Sie Einstellungen überprüfen, wenn der gesicherte Bildschirmschoner über ein Gruppenrichtlinienobjekt konfiguriert werden kann? Zwei mögliche Gründe sind beispielsweise ein Problem bei der Dateireplikation, das verhindert, dass das aktuelle Gruppenrichtlinienobjekt repliziert wird, und Benutzer, die sich für längere Zeit vom Netzwerk abmelden. Sie sollten die Einhaltung der Sicherheitsrichlinien überwachen, um sicherzustellen, dass diese angewendet werden.

Mit dem Skript *AuditScreenSaverWriteToAccess.ps1* können Sie alle Benutzer, für die Profile definiert sind, auf dem lokalen Computer oder einem Remotecomputer abfragen. Rufen Sie die Bildschirmschonerkonfiguration ab und speichern Sie die Informationen in eine Microsoft Access-Datenbank. Verwenden Sie für alle Beispiele in diesem Buch eine Datenbankdatei namens *ConfigurationMaintenance.mdb*. Die in der Datei *ConfigurationMaintenance.mdb* erstellte Bildschirmschonertabelle ist in Abbildung 9.5 dargestellt.

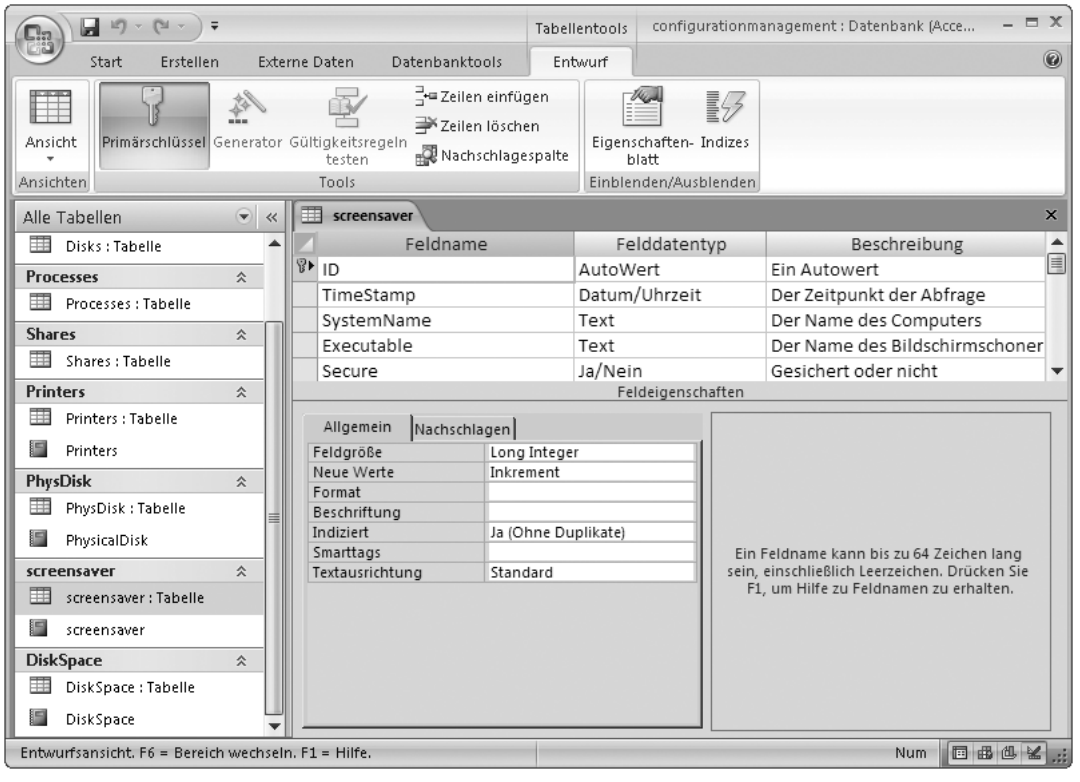


Abbildung 9.5 Die Bildschirmschonertabelle

Sie können auch den in Abbildung 9.6 dargestellten Bericht erstellen. Der Bericht listet alle Einträge aus der Bildschirmschonertabelle auf, berechnet die Prozentanzahl der Benutzer mit aktivierten Bildschirmschonern und zeigt an, ob die Sicherheitseinstellungen der Bildschirmschoner aktiviert sind.

Diese Daten werden mit dem Skript *AuditScreenSaverWriteToAccess.ps1* in die Datenbank übertragen. Beginnen Sie das Skript mit einer *param*-Anweisung. Definieren Sie zwei benannte Argumente: **-computer** und **-help**. Diese Parameter werden den Variablen *$computer* und *$help* zugewiesen. Die Variable *$computer* verwendet den Standardwert *localhost*, um auf den lokalen Computer zu verweisen. Codezeile:

```
param($computer="localhost", $help)
```

Definieren Sie die Funktion *funline*, um die Ausgabe übersichtlicher zu formatieren. Diese Funktion akzeptiert einen Zeichenfolgenwert, bestimmt die Länge der Zeichenfolge und erstellt Ausgabeinformationen, die aus Gleichheitszeichen (=) bestehen. Die Ausgabe ist zweifarbig. Die Funktion *funline* ist wie folgt implementiert:

```
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline += "=" }
    Write-Host -ForegroundColor yellow `n$strIN
    Write-Host -ForegroundColor darkYellow $funline
}
```

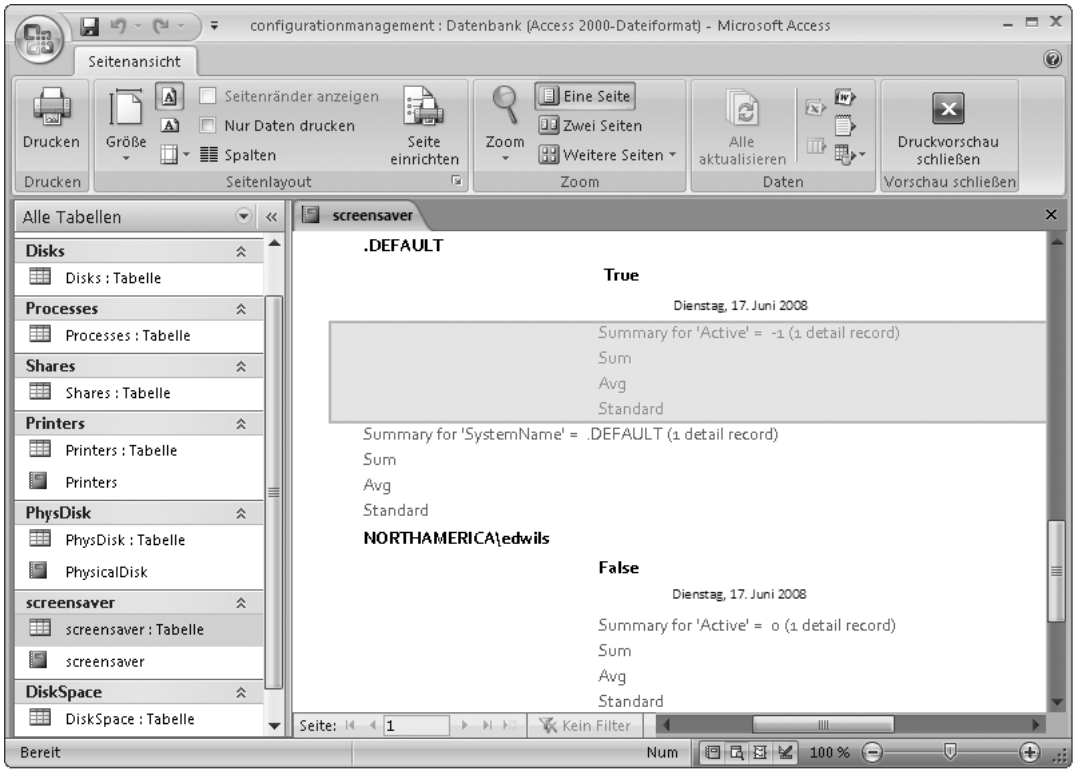


Abbildung 9.6 Ein Bildschirmschonerbericht mit einer Übersicht zu den Sicherheitseinstellungen für Netzwerkadministratoren

Um Hilfeinformationen auszugeben, verwenden Sie die Funktion *funhelp*. Diese Funktion wird nur angezeigt, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Die Funktion *funhelp* definiert eine *here*-Zeichenfolge, die der Variablen *$helpText* zugewiesen ist. Anschließend wird die in *$helpText* gespeicherte Zeichenfolge angezeigt und das Skript beendet. Die Funktion ist wie folgt aufgebaut:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: AuditScreenSaverWriteToAccess.ps1
Erfasst die Bildschirmschonereinstellungen der lokalen Arbeitsstation oder eines Remotecomputers
in einer Access-Datenbank.

PARAMETER:
-computerName Gibt den Namen des Computers an, der mit diesem Skript überprüft werden soll.
-help         Gibt die Hilfeinformationen aus.

SYNTAX:
AuditScreenSaverWriteToAccess.ps1 -computer MunichServer

Erfasst die Bildschirmschonereinstellungen eines Computers namens MunichServer
in einer Access-Datenbank.

AuditScreenSaverWriteToAccess.ps1

Erfasst die Bildschirmschonereinstellungen des lokalen Computers
in einer Access-Datenbank.

AuditScreenSaverWriteToAccess.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Bestimmen Sie mit einer *if*-Anweisung, ob die Funktion *funhelp* aufgerufen werden soll. Wenn die Variable *$help* vorhanden ist, führen Sie den Code aus, der die Funktion *funline* aufruft und die Zeichenfolge "*Ausgeben der Hilfeinformationen …"* übergibt. Die Funktion unterstreicht die Ausgabe. Das Semikolon ermöglicht das Ausführen eines weiteren Befehls, d.h. der Funktion *funhelp*. Dieser Codeabschnitt ist wie folgt implementiert:

```
if($help){ funline("Ausgeben der Hilfeinformationen ...") ; funhelp }
```

Deklarieren Sie die beiden Variablen *$adOpenStatic* und *$adLockOptimistic* mit dem Wert 3. Diese Variablen werden für den Verbindungsaufbau mit der Access-Datenbank benötigt. Die Werte sind im Windows Software Development Kit (SDK) definiert und auf der Microsoft-Website *http://www.microsoft.com* verfügbar. Weisen Sie den Pfad zur Datenbank *ConfigurationMaintenance.mdb* der Variablen *$strDB* zu. Speichern Sie den Namen der Bildschirmschonertabelle in der Variablen *$strTable*. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
$adOpenStatic = $adLockOptimistic = 3
$strDB = "c:\FSO\ConfigurationMaintenance.mdb"
$strTable = "screensaver"
```

Damit das Skript funktioniert, müssen Sie ein *Connection*-Objekt und ein *RecordSet*-Objekt erstellen. Diese Objekte ermöglichen den Zugriff auf die Datenbank und das Aktualisieren der Tabelle. Die beiden Codezeilen lauten:

```
$objConnection = New-Object -ComObject ADODB.Connection
$objRecordSet = new-object -ComObject ADODB.Recordset
```

Öffnen Sie als Nächstes die Verbindung zur Datenbank, indem Sie den Namen des Anbieters angeben. Geben Sie für Access-Datenbanken den Microsoft.Jet.OLEDB.4.0-Anbieter an. Eine Liste der Anbietern amen, die mit der *Open*-Methode des Objekts *ADODB.Connection* verwendet werden können, finden Sie in Anhang B. Der zweite Parameter für die *Open*-Methode bezeichnet den Pfad zur Datenquelle. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
objConnection.Open("Provider = Microsoft.Jet.OLEDB.4.0; `
  Data Source= $strDB")
```

Nachdem eine Verbindung mit der Datenbank hergestellt wurde, öffnen Sie den Datensatz. Wählen Sie hierzu die Datenbanktabelle, die zu verwendende Verbindung und die Methode zum Öffnen der Tabelle aus. Codebeispiel:

```
$objRecordSet.Open("SELECT * FROM $strTable", `
$objConnection, $adOpenStatic, $adLockOptimistic)
```

Geben Sie anschließend mit dem Cmdlet **Write-Host** eine Statusmeldung aus. Wählen Sie mit dem Parameter **-foregroundcolor** eine hervorstechende Farbe aus. Geben Sie einen Zeichenfolgenwert ein, der den Benutzer informiert, dass die Bildschirmschonerinformationen abgerufen werden. Codezeile:

```
write-host -foreGroundColor yellow "Bildschirmschonerinformationen werden abgerufen ..."
```

Sie müssen nun die WMI-Informationen abrufen. Verwenden Sie hierzu das Cmdlet **Get-WmiObject** und wählen Sie die WMI-Klasse *Win32_Desktop* aus. Geben Sie im Parameter **-computername** den Computernamen an, der in der Variablen *$computer* gespeichert wird. Geben Sie mit dem Parameter **-property** die gewünschten Eigenschaften und mit dem Gravis-Zeichen die Zeilenfortsetzung an. Dieser Codeabschnitt ist wie folgt implementiert:

```
$aryscreensaver = Get-WmiObject -Class win32_desktop `
  -computername $computer `
  -Property name, screensaversecure, screensavertimeout, `
  __server, ScreenSaverActive
```

Um die vom vorherigen Befehl zurückgegebenen Objekte zu durchlaufen, verwenden Sie eine *foreach*-Anweisung. Fügen Sie mit der Methode *AddNew()* in jeder Schleife einen neuen Datensatz zur Tabelle hinzu. Führen Sie die Methode *item()* aus, um weitere Elemente zu den angegebenen Eigenschaftsnamen hinzuzufügen. Nachdem Sie alle Informationen hinzugefügt haben, können Sie mit der Methode *Update()* die Änderungen in der Datenbank speichern. Während Sie die Eigenschaften durchlaufen, geben Sie mit dem Cmdlet **Write-Host** eine Statuszeile mit /\-Zeichen aus. Dieser Codeabschnitt ist wie folgt aufgebaut:

```
foreach( $screensaver in $aryScreensaver)
{
   $objRecordSet.AddNew()
   $objRecordSet.Fields.item("TimeStamp") = Get-Date
   $objRecordSet.Fields.item("SystemName") = $($screensaver.name)
   $objRecordSet.Fields.item("Executable") = $($screensaver.screensaverExecutable)
   $objRecordSet.Fields.item("Secure") = $($screensaver.ScreenSaverSecure)
   $objRecordSet.Fields.item("Active") = $($screensaver.ScreenSaverActive)
```

```
   $objRecordSet.Fields.item("Timeout") = $($screensaver.ScreenSaverTimeout)
   $objRecordSet.Update()
   write-host -foregroundColor yellow "/\" -noNewLine
}
```

Die letzten beiden Skriptabschnitte schließen den Datensatz und die Verbindungsobjekte, unter Verwendung der Methode *Close()*. Dieser Codeabschnitt verwendet die folgenden Anweisungen:

```
$objRecordSet.Close()
$objConnection.Close()
```

Das vollständige Skript *AuditScreenSaverWriteToAccess.ps1* hat folgenden Aufbau:

AuditScreenSaverWriteToAccess.ps1

```
param($computer="localhost", $help)
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline += "=" }
    Write-Host -ForegroundColor yellow `n$strIN
    Write-Host -ForegroundColor darkYellow $funline
}

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: AuditScreenSaverWriteToAccess.ps1
Erfasst die Bildschirmschonereinstellungen der lokalen Arbeitsstation oder eines Remotecomputers
in einer Access-Datenbank.

PARAMETER:
-computerName Gibt den Namen des Computers an, der mit diesem Skript überprüft werden soll.
-help         Gibt die Hilfeinformationen aus.

SYNTAX:
AuditScreenSaverWriteToAccess.ps1 -computer MunichServer

Erfasst die Bildschirmschonereinstellungen eines Computers namens MunichServer
in einer Access-Datenbank.

AuditScreenSaverWriteToAccess.ps1

Erfasst die Bildschirmschonereinstellungen des lokalen Computers
in einer Access-Datenbank.

AuditScreenSaverWriteToAccess.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help){ funline("Ausgeben der Hilfeinformationen ...") ; funhelp }
```

```
$adOpenStatic = $adLockOptimistic = 3
$strDB = "C:\FSO\ConfigurationMaintenance.mdb"
$strTable = "screensaver"
$objConnection = New-Object -ComObject ADODB.Connection
$objRecordSet = new-object -ComObject ADODB.Recordset
$objConnection.Open("Provider = Microsoft.Jet.OLEDB.4.0; `
  Data Source= $strDB")
$objRecordSet.Open("SELECT * FROM $strTable", `
$objConnection, $adOpenStatic, $adLockOptimistic)

write-host -foreGroundColor yellow "Ermitteln der Bildschirmschonerinformationen ..."


$aryscreensaver = Get-WmiObject -Class win32_desktop `
  -computername $computer `
  -Property name, screensaversecure, screensavertimeout, `
  __server, ScreenSaverActive

foreach( $screensaver in $aryScreensaver)
{
   $objRecordSet.AddNew()
   $objRecordSet.Fields.item("TimeStamp") = Get-Date
   $objRecordSet.Fields.item("SystemName") = $($screensaver.name)
   $objRecordSet.Fields.item("Executable") = $($screensaver.screensaverExecutable)
   $objRecordSet.Fields.item("Secure") = $($screensaver.ScreenSaverSecure)
   $objRecordSet.Fields.item("Active") = $($screensaver.ScreenSaverActive)
   $objRecordSet.Fields.item("Timeout") = $($screensaver.ScreenSaverTimeout)
   $objRecordSet.Update()
   write-host -foregroundColor yellow "/\" -noNewLine
}

$objRecordSet.Close()
$objConnection.Close()
```

Verwalten der Energieeinstellungen

In einer Energierichtlinie können zahlreiche Einstellungen festgelegt werden. In diesem Abschnitt werden die Energierichtlinien für einen Computer beschrieben. Die Energieeinstellungen sind in Abbildung 9.7 dargestellt.

Mit dem Skript *ReportPowerConfig.ps1* können Sie die vorhandenen Energiekonfigurationseinstellungen abrufen. Das Skript unterstützt mehrere Parameter und gibt folgende Informationen zurück:

- Alle Energiekonfigurationseinstellungen
- Die aktuelle Energiekonfigurationseinstellung
- Die verfügbaren Ruhezustände
- Das letzte Reaktivierungsereignis
- Alle Geräte auf dem aktuellen Computer
- Alle Geräte und die Gerätekonfiguration (einschließlich, ob diese den Ruhezustand unterstützen)
- Alle Geräte, die zum Reaktivieren des Computers konfiguriert sind
- Alle Geräte, die vom Benutzer zum Reaktivieren des Computers konfiguriert werden können

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

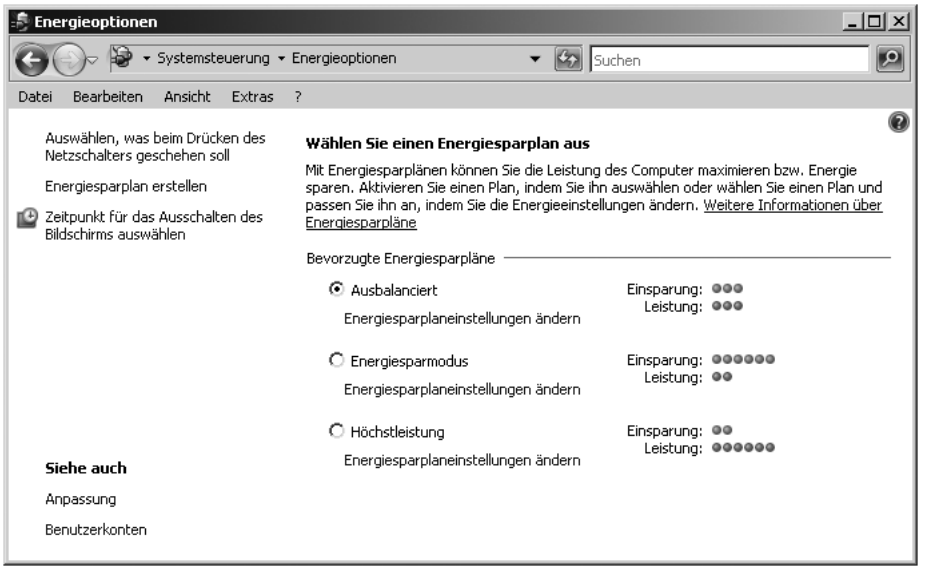


Abbildung 9.7 Energieeinstellungen für einen Windows Server 2008-Computer

Das Skript *GetSites.ps1* beginnt mit einer *param*-Anweisung und zwei Parametern. Der erste Parameter ist **-a** mit dem Standardwert *a*. Der Parameter **-a** gibt die vom Skript ausgeführte Aktion an. Wenn der Parameter **-a** den Wert *a* hat, listet das Skript das aktive Energieschema auf. Da dies der Standardwert ist, zeigt das Skript *ReportPowerConfig.ps1* das aktive Energieschema standardmäßig an, wenn Sie keine Argumente angeben. Für das Argument **-a** können zahlreiche Werte angegeben werden, die später beschrieben werden. Der Parameter **-help** zeigt die Hilfe an, einschließlich einer Beschreibung, der Parameter und der Syntax. Die Remoteausführung wird von diesem Skript nicht unterstützt. Die *param*-Anweisung lautet:

```
param($a="a", $help)
```

Der nächste Abschnitt des Skripts *ReportPowerConfig.ps1* implementiert die Funktion *funline*. Diese Funktion akzeptiert eine Eingabezeichenfolge, bestimmt die Länge der Zeichenfolge und gibt die Zeichenfolge mit einer Zeilentrennung aus, die genauso lang wie die Zeichenfolge ist. Diese Funktion bestimmt die Länge der Zeichenfolge und durchläuft eine *for*-Schleife, um die Variable *$funline* zu erstellen, die aus Gleichheitszeichen besteht. Das Cmdlet **Write-Host** gibt schließlich die Zeichenfolge und die Variable *$funline* aus. Die Funktion *funline* ist wie folgt implementiert:

```
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline += "=" }
    Write-Host -ForegroundColor yellow `n$strIN
    Write-Host -ForegroundColor darkYellow $funline
}
```

Der Funktion *funline* folgt die Funktion *funhelp*. Die Funktion *funhelp* verwendet im Grunde genommen nur eine *here*-Zeichenfolge, die in der Variablen *$helpText* gespeichert ist. Nachdem der Text formatiert und der Variablen *$helpText* zugewiesen wurde, geben Sie die Zeichenfolge aus und beenden Sie das Skript. Die wichtigsten Abschnitte des Hilfetextes sind die Parameterbeschreibungen und die Beispielbefehle. Die Funktion *funhelp* ist wie folgt aufgebaut:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ReportPowerConfig.ps1
Ermittelt die Energieeinstellungen des lokalen Computers.

PARAMETER:
-a(ktion) gibt die auszuführende Aktion an <a (aktives Schema), l (auflisten),
          q (abfragen), d (Gerät), dv (ausführliche Geräteinformationen),
          dwa (für Reaktivieren des Computers aktiviert),
          dwp (für Reaktivieren des Computers konfigurierbar)>
-help    Gibt die Hilfeinformationen aus.

SYNTAX:
ReportPowerConfig.ps1

Ermittelt die Energieeinstellungen des lokalen Computers.

ReportPowerConfig.ps1 -a a

Listet die gegenwärtig aktivierte Gerätekonfiguration des lokalen Computers auf.

ReportPowerConfig.ps1 -a l

Listet alle Gerätekonfigurationen des lokalen Computers auf.

ReportPowerConfig.ps1 -a q

Listet die verfügbaren Ruhezustände des lokalen Computers auf.

ReportPowerConfig.ps1 -a w

Listet das letzte Reaktivierungsereignis des lokalen Computers auf.

ReportPowerConfig.ps1 -a d

Listet alle Geräte auf dem lokalen Computer auf.

ReportPowerConfig.ps1 -a dv

Listet alle Geräte auf dem lokalen Computer mit ausführlichen Informationen auf.

ReportPowerConfig.ps1 -a dwa

Listet alle Geräte auf dem lokalen Computer auf, die den Computer reaktivieren können.

ReportPowerConfig.ps1 -a dwp

Listet alle Geräte auf dem lokalen Computer auf, die der Benutzer zum Reaktivieren des lokalen Computers konfigurieren
```

```
kann.

ReportPowerConfig.ps1 -help ?

Gibt die Hilfeinformationen aus.

"@
$helpText
exit
}
```

Nachdem Sie die Funktion *funhelp* erstellt haben, müssen Sie im Code bestimmen, ob der Hilfetext angezeigt werden soll. Überprüfen Sie zuerst, ob die Variable *$help* vorhanden ist. Wird die Variable gefunden, wurde das Skript mit dem Parameter **-help** ausgeführt. Ist die Variable *$help* vorhanden, rufen Sie die Funktion *funline* auf, geben eine Meldung aus und führen die Funktion *funhelp* aus. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
if($help){funline("Ausgeben der Hilfeinformationen ...") ; funhelp }
```

Um den Computernamen abzurufen, verwenden Sie das COM-Objekt *WScript.Network*. Erstellen Sie das Objekt mit dem Cmdlet **New-Object** und dem Parameter **-comobject**. Wählen Sie nur die Eigenschaft *ComputerName* aus. Der Computername ist in der Variablen *$computer* gespeichert. Codezeile:

```
$computer = (New-Object -ComObject WScript.Network).computername
```

Bevor Sie die Informationen zur Energiekonfiguration abrufen, geben Sie eine Überschrift aus. Verwenden Sie hierzu die Funktion *funline* und geben Sie eine Zeichenfolge an. Verwenden Sie für die Überschrift den Wert, der in der Variablen *$computer* gespeichert ist. Stellen Sie dem Variablennamen ein Dollar-Zeichen voran und geben Sie die Zeichenfolge in Klammern ein. Codezeile:

```
funline("Energieeinstellungen von: $($computer)")
```

Das wichtigste Codesegment ist die *switch*-Anweisung, die den Wert der Variablen *$a* auswertet, der in der Befehlszeile eingegeben wurde. Wenn der Wert *a* ist, zeigen Sie das aktive Energieschema, wie im Dienstprogramm **Powercfg.exe** konfiguriert, an. Geben Sie das Sonderzeichen *`r* ein, um mit der nächsten Zeile fortzufahren und die Ausgabe übersichtlicher zu gestalten. Wenn die Variable *$a* den Wert *l* hat, geben Sie eine Liste aller konfigurierten Energieschemas aus. Wenn der Wert *q* für den Parameter **-a** angegeben wurde, geben Sie alle verfügbaren Ruhezustände aus, die auf dem Computer konfiguriert sind. Wenn der Parameter **-a** den Wert *w* hat, geben Sie das letzte Reaktivierungsereignis aus. Der Wert *d* fragt alle auf dem Computer definierten Geräte ab. Wenn das Skript ausgeführt wird und *$a* den Wert *dv* aufweist, wird eine ausführlichere Abfrage aller Geräte ausgeführt, um alle Geräte und die unterstützten Energieverwaltungsfunktionen aufzulisten. Wenn Sie beim Ausführen des Skripts den Wert *dwa* angeben, gibt die *switch*-Anweisung eine Liste aller konfigurierten Geräte zurück, um den Computer aus dem Ruhezustand zu reaktivieren. Der letzte Wert ist *dwp,* mit dem Sie eine Liste aller Geräte abrufen, die zum Reaktivieren des Computers konfiguriert werden können. Die vollständige *switch*-Anweisung lautet:

```
switch($a)
{
 "a"   { powercfg -getactivescheme ; "`r"}
 "l"   { powercfg -list }
 "q"   { powercfg -availablesleepstates }
 "w"   { powercfg -lastwake }
 "d"   { powercfg -devicequery all_devices }
```

```
 "dv"  { powercfg -devicequery all_devices_verbose }
 "dwa" { powercfg -devicequery wake_armed }
 "dwp" { powercfg -devicequery wake_programmable }
 }
```

Das vollständige Skript *ReportPowerConfig.ps1* hat folgenden Inhalt:

ReportPowerConfig.ps1

```
param($a="a", $help)
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline += "=" }
    Write-Host -ForegroundColor yellow `n$strIN
    Write-Host -ForegroundColor darkYellow $funline
}

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ReportPowerConfig.ps1
Ermittelt die Energieeinstellungen des lokalen Computers.

PARAMETER:
-a(ktion) gibt die auszuführende Aktion an <a (aktives Schema), l (auflisten),
          q (abfragen), d (Gerät), dv (ausführliche Geräteinformationen),
          dwa (für Reaktivieren des Computers aktiviert),
          dwp (für Reaktivieren des Computers konfigurierbar)>
-help     Gibt die Hilfeinformationen aus.

SYNTAX:
ReportPowerConfig.ps1

Ermittelt die Energieeinstellungen des lokalen Computers.

ReportPowerConfig.ps1 -a a

Listet die gegenwärtig aktivierte Gerätekonfiguration des lokalen Computers auf.

ReportPowerConfig.ps1 -a l

Listet alle Gerätekonfigurationen des lokalen Computers auf.

ReportPowerConfig.ps1 -a q

Listet die verfügbaren Ruhezustände des lokalen Computers auf.

ReportPowerConfig.ps1 -a w

Listet das letzte Reaktivierungsereignis des lokalen Computers auf.

ReportPowerConfig.ps1 -a d

Listet alle Geräte auf dem lokalen Computer auf.
```

```
ReportPowerConfig.ps1 -a dv

Listet alle Geräte auf dem lokalen Computer mit ausführlichen Informationen auf.

ReportPowerConfig.ps1 -a dwa

Listet alle Geräte auf dem lokalen Computer auf, die den Computer reaktivieren können.

ReportPowerConfig.ps1 -a dwp

Listet alle Geräte auf dem lokalen Computer auf, die der Benutzer
zum Reaktivieren des lokalen Computers konfigurieren kann.

ReportPowerConfig.ps1 -help ?

Gibt die Hilfeinformationen aus.

"@
$helpText
exit
}

if($help){funline("Ausgeben der Hilfeinformationen ...") ; funhelp }
$computer = (New-Object -ComObject WScript.Network).computername

funline("Energieeinstellungen für: $($computer)")

switch($a)
{
 "a"   { POWERCFG -getactivescheme ; "`r"}

 "l"   { powercfg -list }
 "q"   { powercfg -availablesleepstates }
 "w"   { powercfg -lastwake }
 "d"   { powercfg -devicequery all_devices }
 "dv"  { powercfg -devicequery all_devices_verbose }
 "dwa" { powercfg -devicequery wake_armed }
 "dwp" { powercfg -devicequery wake_programmable }
 }
```

Ändern des Energieschemas

Sie können am Energieschema in Windows Vista und Windows Server 2008 zahlreiche Änderungen vornehmen. Diese Einstellungen berücksichtigen, ob der Computer mit Netzstrom oder Batteriestrom betrieben wird. Wenn der Computer mit Batterien betrieben wird, ist die Energieeinsparung oft das wichtigste Anliegen. Dies trifft jedoch nicht immer zu. Manchmal ist die Leistung des Computers ein wichtigerer Faktor (beispielsweise wenn die Batterien in einem bestimmten Zeitraum wieder aufgeladen werden können). Das Skript *SetPowerConfig.ps1* ermöglicht Ihnen, die Energieeinstellungen für den Monitor und die Festplatte sowie die Ruhezustands- und Standby-Funktionen auf batteriebetriebenen und mit Netzstrom versorgten Computern zu konfigurieren. Sie können über die Energieoptionen einen benutzerdefinierten Energieplan erstellen (siehe Abbildung 9.8).

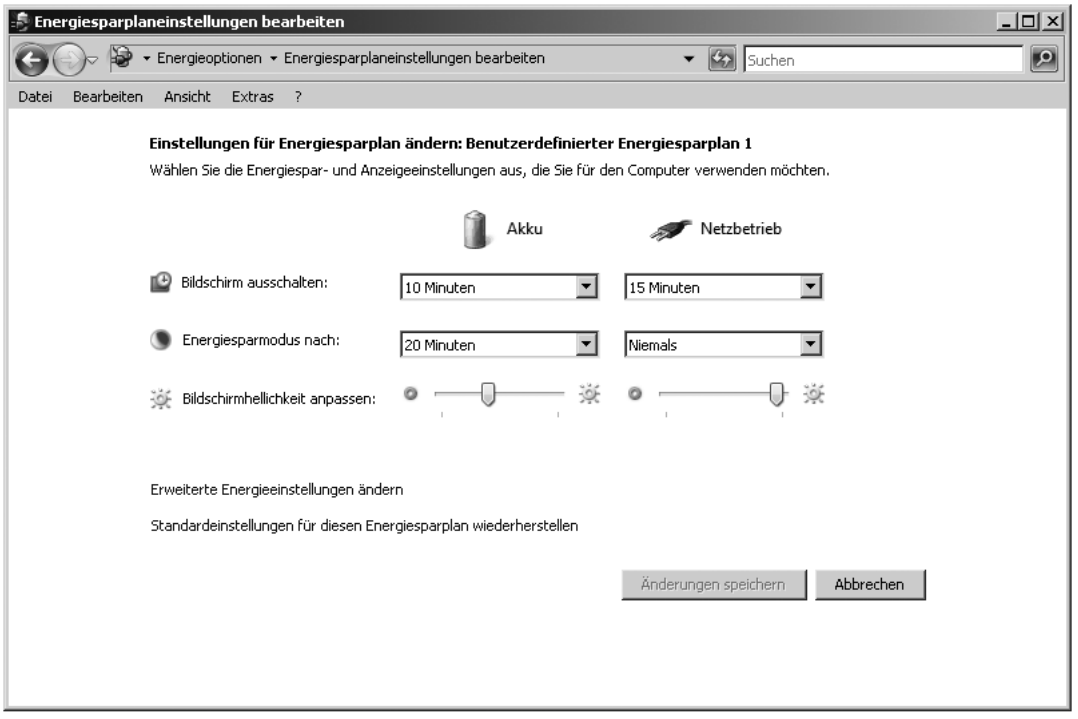


Abbildung 9.8 Benutzerdefinierter Energieplan für einen Windows Server 2008-Computer

Das Skript *SetPowerConfig.ps1* beginnt mit einer *param*-Anweisung. Geben Sie für dieses Skript vier Parameter an, die nicht auf Standardwerte festgelegt werden, da sich einige der Argumente gegenseitig ausschließen (beispielsweise **-q** und **-help**). Die Argumente **-c** und **-t** müssen zusammen angegeben werden, da der Wert von **-t** den Zeitüberschreitungswert beim Ändern des Energieschemas für den angegebenen Parameter festlegt. Wenn der Wert nicht vorhanden ist, wird ein Fehler generiert. Dies wird später erklärt. Die *param*-Anweisung lautet:

```
param($c, $t, $q, $help)
```

Als Nächstes folgt der Codeabschnitt für die Funktion *funline* im Skript *SetPowerConfig.ps1*. Diese Funktion versieht die Überschrift des Energieberichts auf dem lokalen Computer mit einer Trennlinie. Diese Funktion ist wie folgt implementiert:

```
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline += "=" }
    Write-Host -ForegroundColor yellow `n$strIN
    Write-Host -ForegroundColor darkYellow $funline
}
```

Die Funktion *funhelp* zeigt die Hilfe für das Skript an. Für ein Skript, das so viele verschiedene Parameter umfasst, ist eine umfassende Hilfe ausgesprochen wichtig. Die Funktion *funhelp* erstellt eine lange *here*-Zeichenfolge, weist diese der Variablen *$helpText* zu, gibt die Hilfe aus und beendet dann das Skript. Die Funktion *funhelp* umfasst folgende Anweisungen:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: SetPowerConfig.ps1
Ändert die Energieeinstellungen des lokalen Computers.

PARAMETER:
-c (ändern)   <mp,mb,dp,db,sp,sb,hp,hb>
-q (abfragen)   Detaillierte Abfrage des gegenwärtigen Energieplans.
-t (Zeitüberschreitung) Gibt einen neuen Zeitüberschreitungswert an. Dieser Wert muss angegeben werden,
            wenn der Parameter -c verwendet wird, um einen Wert zu ändern.
-help       Gibt die Hilfeinformationen aus.

SYNTAX:
SetPowerConfig.ps1

Zeigt eine Fehlermeldung an, denn es muss ein Parameter angegeben werden.

SetPowerConfig.ps1 -c mp -t 10

Setzt den Zeitüberschreitungswert im Netzbetrieb
auf 10 Minuten.

SetPowerConfig.ps1 -c mb -t 5

Setzt den Zeitüberschreitungswert im Batteriebetrieb
auf 5 Minuten.

SetPowerConfig.ps1 -c dp -t 15

Setzt den Zeitüberschreitungswert für die Festplatte im Netzbetrieb
auf 15 Minuten.

SetPowerConfig.ps1 -c db -t 7

Setzt den Zeitüberschreitungswert für die Festplatte im Batteriebetrieb
auf 7 Minuten.

SetPowerConfig.ps1 -c sp -t 30

Setzt den Zeitüberschreitungswert für den Standbymodus  im Netzbetrieb
auf 30 Minuten.

SetPowerConfig.ps1 -c sb -t 10

Setzt den Zeitüberschreitungswert für den Standbymodus  im Batteriebetrieb
auf 10 Minuten.

SetPowerConfig.ps1 -c hp -t 45

Setzt den Zeitüberschreitungswert für den Ruhezustand im Netzbetrieb
auf 45 Minuten.

SetPowerConfig.ps1 -c hb -t 15
```

```
Setzt den Zeitüberschreitungswert für den Ruhezustand im Batteriebetrieb
auf 15 Minuten.

SetPowerConfig.ps1 -q c

Listet die Konfigurationseinstellungen des gegenwärtigen
Energieplans detailliert auf.

SetPowerConfig.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Überprüfen Sie, ob die Variable *$help* vorhanden ist. Ist dies der Fall, wurde das Skript mit dem Argument **-help** ausgeführt. Wenn die Variable *$help* vorhanden ist, verwenden Sie die Funktion *funline*, um eine entsprechende Meldung auszugeben, und rufen Sie danach die Funktion *funhelp* auf. Codezeile:

```
if($help){funline("Ausgeben der Hilfeinformationen ...") ; funhelp }
```

Verwenden Sie das *WshNetwork*-Objekt, um den Namen des lokalen Computers abzufragen. Verwenden Sie hierzu das Cmdlet **New-Object** mit dem Parameter **-comobject** und der Programm-ID *WScript.Network*. Schließen Sie die Eingabe in Klammern ein, rufen Sie die Eigenschaft *ComputerName* ab und speichern Sie das Ergebnis in der Variablen *$computer*. Codezeile:

```
$computer = (New-Object -ComObject WScript.Network).computername
```

Wenn der Parameter **-q** in der Befehlszeile angegeben wird, ist die Variable *$q* vorhanden. Geben Sie mit der Funktion *funline* eine entsprechende Überschrift mit dem Computernamen aus und geben Sie für das Dienstprogramm **Powercfg** das Argument **-query** an. Mit diesem Vorgang können Sie eine detaillierte Liste des aktuellen Energieschemas auf dem Computer erstellen. Beenden Sie danach das Skript mit einer *exit*-Anweisung. Dieser Codeabschnitt ist wie folgt implementiert:

```
if($q)
{
 funline("Energieeinstellungen für: $($computer)")
 powercfg -query
 exit
}
```

Sie müssen sicherstellen, dass die Parameter **-c** und **-t** zusammen angegeben werden, da die Variable *$t* den Zeitüberschreitungswert enthält. Wenn die Variable *$c* vorhanden ist, aber nicht die Variable *$t*, verwenden Sie eine *throw*-Anweisung, um einen Fehler zu generieren. Die Meldung wird standardmäßig in Rot angezeigt und das Skript wird beendet. Codesegment:

```
if($c -and !$t)
      {
       $(Throw 'Für $t ist ein Wert erforderlich.
       Für weitere Informationen führen Sie folgenden Befehl aus: SetPowerConfig.ps1 -help ?')
      }
```

Überprüfen Sie anschließend den Wert des Parameters **-c** mit einer *switch*-Anweisung. Wenn der Wert *mp* ist, betrifft die Einstellung den Computerbetrieb mit Netzstrom. Legen Sie entsprechend den Zeitüberschreitungswert für den Monitor auf den Wert in der Variablen *$t* fest. Wenn der Wert *mb*

lautet, betrifft die Einstellung den Computerbetrieb mit Batteriestrom. Legen Sie in diesem Fall den Zeitüberschreitungswert für den Batteriebetrieb des Monitors auf den Wert *$t* fest. Der Wert *dp* deaktiviert die Datenträger bei Netzstrombetrieb nachdem der in *$t* angegebene Wert abgelaufen ist. Wenn der Wert *db* ist, konfigurieren Sie den aktuellen Energieplan so, dass die Laufwerke beim Erreichen des Werts *$t* deaktiviert werden. *Sp* betrifft das Aktivieren des Standbymodus, wenn der Computer mit Netzstrom versorgt und der Zeitüberschreitungswert *$t* erreicht wird. *Sb* ist der Zeitüberschreitungswert für Standby bei Batteriebetrieb. Um den Computer in den Ruhezustand zu versetzen, verwenden Sie *hp* und legen Sie den Zeitüberschreitungswert fest. *Hb* gibt den Zeitüberschreitungswert bei Batteriebetrieb an. Der Standardblock fängt ungültige Werte für *$c* ab. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
switch($c)
{
 "mp" { powercfg -CHANGE -monitor-timeout-ac $t }
 "mb" { powercfg -CHANGE -monitor-timeout-dc $t }
 "dp" { powercfg -CHANGE -disk-timeout-ac $t}
 "db" { powercfg -CHANGE -disk-timeout-dc $t }
 "sp" { powercfg -CHANGE -standby-timeout-ac $t }
 "sb" { powercfg -CHANGE -standby-timeout-dc $t }
 "hp" { powercfg -CHANGE -hibernate-timeout-ac $t }
 "hb" { powercfg -CHANGE -hibernate-timeout-dc $t }
 DEFAULT {
           "Der Wert $c ist nicht zulässig. Für weitere Informationen führen Sie folgenden Befehl aus:
           SetPowerConfig.ps1 -help ?"
         }
}
```

Das vollständige Skript *SetPowerConfig.ps1* hat folgenden Aufbau:

SetPowerConfig.ps1

```
param($c, $t, $q, $help)
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline += "=" }
    Write-Host -ForegroundColor yellow `n$strIN
    Write-Host -ForegroundColor darkYellow $funline
}

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: SetPowerConfig.ps1
Ändert die Energieeinstellungen des lokalen Computers.

PARAMETER:
c (ändern)   <mp,mb,dp,db,sp,sb,hp,hb>
-q (abfragen)   Detaillierte Abfrage des gegenwärtigen Energieplans.
-t (Zeitüberschreitung) Gibt einen neuen Zeitüberschreitungswert an. Dieser Wert muss angegeben werden,
            wenn der Parameter -c verwendet wird, um einen Wert zu ändern.
-help       Gibt die Hilfeinformationen aus.

SYNTAX:
```

```
SetPowerConfig.ps1

Zeigt eine Fehlermeldung an, denn es muss ein Parameter angegeben werden.

SetPowerConfig.ps1 -c mp -t 10

Setzt den Zeitüberschreitungswert im Netzbetrieb
auf 10 Minuten.

SetPowerConfig.ps1 -c mb -t 5

Setzt den Zeitüberschreitungswert im Batteriebetrieb
auf 5 Minuten.

SetPowerConfig.ps1 -c dp -t 15

Setzt den Zeitüberschreitungswert für die Festplatte im Netzbetrieb
auf 15 Minuten.

SetPowerConfig.ps1 -c db -t 7

Setzt den Zeitüberschreitungswert für die Festplatte im Batteriebetrieb
auf 7 Minuten.

SetPowerConfig.ps1 -c sp -t 30

Setzt den Zeitüberschreitungswert für den Standbymodus  im Netzbetrieb
auf 30 Minuten.

SetPowerConfig.ps1 -c sb -t 10

Setzt den Zeitüberschreitungswert für den Standbymodus  im Batteriebetrieb
auf 10 Minuten.

SetPowerConfig.ps1 -c hp -t 45

Setzt den Zeitüberschreitungswert für den Ruhezustand im Netzbetrieb
auf 45 Minuten.

SetPowerConfig.ps1 -c hb -t 15

Setzt den Zeitüberschreitungswert für den Ruhezustand im Batteriebetrieb
auf 15 Minuten.

SetPowerConfig.ps1 -q c

Listet die Konfigurationseinstellungen des gegenwärtigen
Energieplans detailliert auf.

SetPowerConfig.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
```

```
}

if($help){funline("Ausgeben der Hilfeinformationen ...") ; funhelp }
$computer = (New-Object -ComObject WScript.Network).computername

if($q)
{
 funline("Energieeinstellungen für: $($computer)")
 powercfg -query
 exit
}

if($c -and !$t)
      {
       $(Throw ''Für  $t ist ein Wert erforderlich.
       Für weitere Informationen führen Sie folgenden Befehl aus: SetPowerConfig.ps1 -help ?')
      }

switch($c)
{
 "mp" { powercfg -CHANGE -monitor-timeout-ac $t }
 "mb" { powercfg -CHANGE -monitor-timeout-dc $t }
 "dp" { powercfg -CHANGE -disk-timeout-ac $t}
 "db" { powercfg -CHANGE -disk-timeout-dc $t }
 "sp" { powercfg -CHANGE -standby-timeout-ac $t }
 "sb" { powercfg -CHANGE -standby-timeout-dc $t }
 "hp" { powercfg -CHANGE -hibernate-timeout-ac $t }
 "hb" { powercfg -CHANGE -hibernate-timeout-dc $t }
 DEFAULT {
           "Der Wert $c ist nicht zulässig. Für weitere Informationen führen Sie folgenden Befehl aus:
           SetPowerConfig.ps1 -help ?"
         }
}
```

Zusammenfassung

In diesem Kapitel wurde das Konfigurieren der Desktopeinstellungen von Windows Vista und Windows Server 2008 beschrieben, einschließlich des auf einem Computer konfigurierten Bildschirmschoners und den dazugehörigen Sicherheitseinstellungen. Sie haben überprüft, ob ein Bildschirmschoner gesichert ist und die Informationen in einer Datenbank gespeichert. Im Anschluss daran wurden die Energieeinstellungen erklärt, einschließlich das Überprüfen der aktuellen Energieeinstellungen. Diese nützliche Methode hilft Benutzern, die Lebensdauer der Batterien von tragbaren Computern zu verlängern. Das Kapitel wurde mit Informationen zum Konfigurieren der Energieverwaltungseinstellungen abgeschlossen.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

KAPITEL 10

Behandeln von Problemen nach der Bereitstellung

Nach Abschluss dieses Kapitels können Sie:

- Einen Computer umbenennen
- Die korrekte Zeit festlegen
- Die autorisierte Zeitquelle konfigurieren
- Ein lokales Benutzerkonto erstellen und ein Kennwort festlegen
- Ein Administratorkonto aktivieren
- Einen Remotecomputer oder einen Server herunterfahren und neu starten

 **Auf der Begleit-CD** Alle Skripts in diesem Kapitel befinden sich auf der Begleit-CD im Ordner *\Scripts\Chapter10*.

Obwohl die Installation von Windows weitgehend automatisiert ist, müssen nach der Installation des Betriebsystems mehrere Konfigurationsaufgaben ausgeführt werden. Diese Aufgaben umfassen beispielsweise das Festlegen der Systemzeit, das Erstellen lokaler Benutzer und das Konfigurieren von Firewalleinstellungen, um die Remoteverwaltung zu ermöglichen. Einige dieser Aufgaben lassen sich schnell erledigen, während andere Aufgaben etwas komplizierter sind. In diesem Kapitel werden die nach der Installation am häufigsten auszuführenden Aufgaben beschrieben.

Festlegen der Systemzeit

Die Geräteverwaltung beginnt mit dem Festlegen der Systemzeit. Auf Computern, die mit einer Domäne verbunden sind, wird die Zeit von den Domänencontrollern aktualisiert, aber nicht alle Computer, beispielsweise Laptops, sind ständig mit der Domäne verbunden. Es kann vorkommen, dass ein Computer aufgrund einer fehlerhaften Zeiteinstellung der Domäne nicht beitreten kann. In diesem Fall spart die Möglichkeit, die Zeit remote festzulegen, Zeit und Arbeit. Dies trifft insbesondere auf Windows Server 2008 Server Core zu. Das Dienstprogramm **Datum und Uhrzeit** von Windows Vista und Windows Server 2008 wurde zwar wesentlich verbessert, aber erfüllt die Anforderungen großer Unternehmen immer noch nicht.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

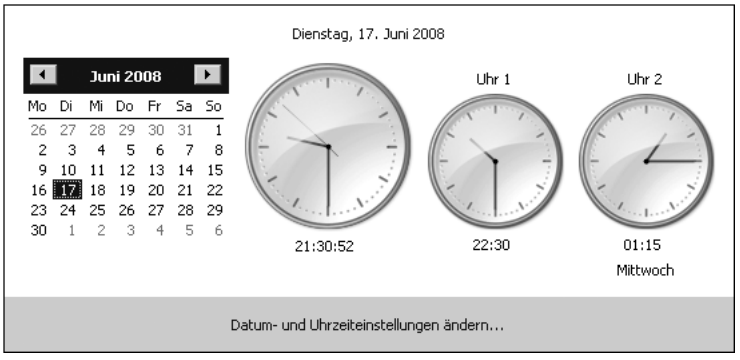


Abbildung 10.1 Das neue und verbesserte Dienstprogramm Datum und Uhrzeit von Windows Vista und Windows Server 2008

Festlegen der Systemzeit auf einem Remotecomputer

Mit dem Skript *GetSetTime.ps1* können Sie auf einem Remotecomputer über WMI (Windows Management Instrumentation) die Systemzeit abfragen und auf die Zeit des lokalen Computers festlegen. Da Sie die Systemzeit auf einem Remotecomputer mit WMI abfragen und festlegen können, bietet dieses Skript die Möglichkeit, Windows Server 2008 Server Core-Computer und andere Windows-Computer zu verwalten, auf denen WMI installiert ist.

Beginnen Sie das Skript *GetSetTime.ps1* mit der Definition von drei Parametern. Der erste Parameter ist der Name des Computers, mit dem Sie die Verbindung herstellen. Hierbei kann es sich um einen lokalen Computer oder einen Remotecomputer handeln. Der Wert der Variablen *$computer* ist standardmäßig auf *localhost* (den lokalen Computer) festgelegt. Der zweite Parameter ist **-a**. Weisen Sie der Variablen *$a* die Aktion zu, die Sie ausführen möchten. Die Aktion kann mit q (abfragen) oder s (setzen) angegeben werden. Der dritte Parameter ist **-help**. Wenn die Variable *$help* vorhanden ist, geben Sie eine Hilfemeldung aus. Die erste Codezeile lautet:

```
param($computer="localhost", $a, $help)
```

Die nächste Funktion ist *funline*, mit der ein Teil der Ausgabe unterstrichen werden kann, damit die ausgegebenen Informationen einfacher zu lesen sind. Die Funktion akzeptiert eine Zeichenfolge als Parameter, die angezeigt wird, wenn die Funktion aufgerufen wird. Die Funktion bestimmt die Länge der Zeichenfolge und Sie erstellen mit einer *for*-Anweisung eine Zeichenfolge aus Gleichheitszeichen als Unterstreichung bzw. Zeilentrennung. Geben Sie mit dem Cmdlet **Write-Host** zuerst die Eingabezeichenfolge und anschließend die generierte Zeilentrennung aus. Die Funktion *funline* ist wie folgt implementiert:

```
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline += "=" }
    Write-Host -ForegroundColor yellow `n$strIN
    Write-Host -ForegroundColor darkYellow $funline
}
```

Definieren Sie danach die Funktion *funhelp*, um eine Hilfemeldung anzuzeigen, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Erstellen Sie eine *here*-Zeichenfolge, die mit @" beginnt und mit "@ endet. Weisen Sie die *here*-Zeichenfolge der Variablen *$helpText* zu. Nachdem Sie die *here*-

Zeichenfolge definiert haben, geben Sie den Wert der Variablen *$helpText* aus und beenden Sie das Skript. Die Funktion *funhelp* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GetSetTime.ps1
Ermittelt oder aktualisiert die Systemzeit des lokalen Computers oder eines Remotecomputers.

PARAMETER:
-computer  Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-a(ktion)  Legt fest, ob die aktuelle Zeit angezeigt oder geändert werden soll.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
GetSetTime.ps1 -computer MunichServer

Zeigt die aktuelle Systemzeit eines Computers namens MunichServer an.

GetSetTime.ps1

Zeigt die aktuelle Systemzeit des Computers an.

GetSetTime.ps1 -a q

Zeigt die aktuelle Systemzeit des Computers an.

GetSetTime.ps1 -a q -computer MunichServer

Zeigt die aktuelle Systemzeit eines Computers namens MunichServer.

GetSetTime.ps1 -a s -computer MunichServer

Aktualisiert die Systemzeit eines Computers namens MunichServer.

GetSetTime.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Um zu bestimmen, ob die Hilfe angezeigt werden muss, überprüfen Sie, ob die Variable *$help* existiert. Da die Variable *$help* nicht von der *param*-Anweisung initialisiert wird, ist die Variable nur vorhanden, wenn das Skript mit dem Parameter *-help* ausgeführt wird. Ist die Variable *$help* vorhanden, geben Sie eine Meldung aus, dass die Hilfe abgerufen wird und rufen Sie die Funktion *funhelp* auf. Codezeile:

```
if($help){funline("Ausgeben der Hilfeinformationen ...") ; funhelp }
```

Sie müssen nun das aktuelle Datum und die Zeit auf dem lokalen Computer abrufen. Auch wenn Sie das Skript für einen Remotecomputer ausführen, müssen Sie das aktuelle Datum und die Zeit auf dem lokalen Computer ermitteln.

Hinweis Wenn Sie das Skript für einen Remotecomputer ausführen, greift dieses tatsächlich auf zwei Computer zu. Das Skript ruft das Datum und die Uhrzeit mit dem Cmdlet **Get-Date** vom lokalen Computer ab. Anschließend greift das Skript mit WMI auf das Datum und die Zeit des Remotecomputers zu. Sie müssen WMI verwenden, da das Cmdlet **Get-Date** in Version 1.0 von Windows PowerShell keine Remotevorgänge unterstützt.

Nachdem Sie mit dem Cmdlet **Get-Date** das Datum und die Uhrzeit abgerufen haben, müssen Sie die Informationen in ein Format konvertieren, das WMI verarbeiten kann. WMI erfordert Datum/Zeit-Werte im UTC-Format (Universal Time Coordinates) oder im DMTF-Format (Distributed Management Task Force). Die UTC-Zeit wird in Minuten der GMT-Zeit (Greenwich Mean Time) ausgedrückt und liegt im Bereich von +720 bis –720. Der entsprechende Wert wird gegebenenfalls für die Sommerzeit subtrahiert oder addiert. Die UTC-Zeit kann gelesen werden, aber ist etwas schwierig. Die ersten vier Ziffern sind das Jahr (2007), dem der Monat (08) und der Tag (10) folgen. Die nächsten Ziffern geben die Zeit an (12:07 und 19 Sekunden). Die letzten drei Ziffern sind der Offest. In diesem Beispiel ist der Offset -240 (-4 GMT). Auch wenn der Offset während der Sommerzeit tatsächlich -5 GMT ist, ist er eigentlich -4 GMT.

```
20070810120719.323553-240
```

Sie können einen normalen Datum/Zeit-Wert in UTC konvertieren, aber dieses Verfahren ist mühsam und fehleranfällig. Sie müssen diesen Vorgang jedoch nicht manuell ausführen, sondern können die Microsoft .NET Framework-Klasse *Management.ManagementDatetimeConverter* verwenden. Rufen Sie die statische Methode *ToDmtfDateTime() auf.* Um eine statische Methode aufzurufen, geben Sie zwei Doppelpunkte (::) an. Diese Notation wandelt ein Datum/Zeit-Objekt in das UTC-Format um. Weisen Sie den aktuellen Datum/Zeit-Wert der Variablen *$date* zu. Damit der Code übersichtlicher ist, brechen Sie die Zeile mit dem Graviszeichen um. Wenn Sie die Zeile nicht umbrechen möchten, geben Sie den Code in einer Zeile an und entfernen Sie das Graviszeichen:

```
$date = [Management.ManagementDatetimeConverter]::`
  ToDmtfDateTime($(get-date))
```

Fragen Sie nun mit dem Cmdlet **Get-WmiObject** die WMI-Klasse *Win32_OperatingSystem* ab und verwenden Sie den Computer, der in der Befehlszeile angegeben wurde. Die Variable *$computer* ist standardmäßig auf den Wert *localhost* festgelegt. Sie können jedoch einen anderen Wert im Parameter **-computer** angeben. Weisen Sie das resultierende Verwaltungsobjekt der Variablen *$objWMI* zu. Geben Sie den Code in zwei Zeilen ein. Codeabschnitt:

```
$objWMI = Get-WmiObject -ComputerName $computer `
  -Class win32_operatingsystem
```

In der WMI-Klasse *Win32_OperatingSystem* sind 75 Eigenschaften definiert, die mit der vorherigen Codezeile abgerufen werden. Die Variable *$objWMI* bietet Zugriff auf alle diese Eigenschaften. Sie benötigen jedoch nur die Eigenschaft *LocalDateTime*. Diese Eigenschaft gibt das Datum und die Zeit an, die vom in der Variablen *$computer* festgelegten Computer abgerufen wurde. Der Wert wird im UTC-Format zurückgegeben und anschließend in ein lesbareres Format konvertiert. Codezeile:

```
$localUTC=$objwmi.localDateTime
```

Um das Skript weiter auszubauen, analysieren Sie den Wert, der im Parameter **-a** angegeben wurde. Lautet der für **-a** angegebene Wert **q**, rufen Sie die Zeit des Computers ab. Hierbei kann es sich um den lokalen Computer oder einen Remotecomputer handeln, abhängig vom Wert des Parameters **-computer**. Geben Sie mit der Funktion *funline* eine Kopfzeile für den Bericht aus. Da das Skript lokal oder remote

ausgeführt werden kann, rufen Sie den Wert der Eigenschaft *Csname* ab. Die Eigenschaft *Csname* gibt den Namen des Computersystems und die Quelle des Zeitwerts an. Verwenden Sie die .NET Framework-Klasse *Management.ManagementDatetimeConverter* und die statische Methode *ToDateTime()*, um das UTC-Format in das normale Format zu konvertieren. Die *switch*-Anweisung lautet:

```
switch($a)
{
 "q"    {
          funline("Die Systemzeit auf dem Computer $($objWMI.csname) ist:")
          [Management.ManagementDatetimeConverter]::`
          ToDateTime($localUTC)
        }
```

Wenn der Wert **s** im Parameter **-a** angegeben wird, rufen Sie die Methode *SetDateTime()* aus der WMI-Klasse *Win32_OperatingSystem* auf. Für die Methode *SetDateTime()* muss die Zeit im UTC-Format angegeben werden. Da Sie die lokale Zeit bereits in das UTC-Format konvertiert und der Variablen *$date* zugewiesen haben, können Sie diese Variable direkt in den Methodenaufruf einbeziehen. Heben Sie die Aktion mit der Funktion *funline* auf dem lokalen Bildschirm hervor. Das zurückgegebene Fehlerobjekt zeigt den Status des Methodenaufrufs an. Der Wert 0 zeigt an, dass beim Festlegen der Zeit keine Fehler aufgetreten sind. Geben Sie das Objekt aus, ohne die Fehlermeldungen zu erklären. Die *switch*-Anweisung lautet:

```
 "s"     {
          funline("Aktualisieren der Systemzeit auf dem Computer $computer ...")
          $objWMI.SetDateTime($date)
        }
```

Wenn für den Parameter **-a** weder ein **s** noch ein **q** angegeben wurde, geben Sie die lokale Zeit im normalen Format aus. Verwenden Sie hierzu die *DEFAULT*-Klausel der *switch*-Anweisung. Geben Sie mit der Funktion *funline* eine Kopfzeile aus und verwenden Sie die .NET Framework-Klasse *Management.ManagementDatetimeConverter*, um die Methode *ToDateTime()* aufzurufen. Übergeben Sie den Datum/Zeit-Wert in der Variablen *$localUTC*. Die Zeit wird direkt aus WMI abgerufen. Die *switch*-Anweisung lautet:

```
 DEFAULT {
          funline("Die Systemzeit auf dem Computer $($objWMI.csname) ist:")
          [Management.ManagementDatetimeConverter]::`
          ToDateTime($localUTC)
          }
}
```

Das vollständige Skript *GetSetTime.ps1* hat folgenden Inhalt:

GetSetTime.ps1

```
param($computer="localhost", $a, $help)
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline += "=" }
    Write-Host -ForegroundColor yellow `n$strIN
    Write-Host -ForegroundColor darkYellow $funline
}

function funHelp()
{
```

```
$helpText=@"
BESCHREIBUNG:
NAME: GetSetTime.ps1
Ermittelt oder aktualisiert die Systemzeit des lokalen Computers oder eines Remotecomputers.

PARAMETER:
-computer  Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-a(ktion)  Legt fest, ob die aktuelle Zeit angezeigt oder geändert werden soll.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
GetSetTime.ps1 -computer MunichServer

Zeigt die aktuelle Systemzeit eines Computers namens MunichServer an.

GetSetTime.ps1

Zeigt die aktuelle Systemzeit des Computers an.

GetSetTime.ps1 -a q

Zeigt die aktuelle Systemzeit des Computers an.

GetSetTime.ps1 -a q -computer MunichServer

Zeigt die aktuelle Systemzeit eines Computers namens MunichServer.

GetSetTime.ps1 -a s -computer MunichServer

Aktualisiert die Systemzeit eines Computers namens MunichServer.

GetSetTime.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help){funline("Ausgeben der Hilfeinformationen ...") ; funhelp }

$date = [Management.ManagementDatetimeConverter]::`
  ToDmtfDateTime($(get-date))

$objWMI = Get-WmiObject -ComputerName $computer `
  -Class win32_operatingsystem
$localUTC=$objwmi.localDateTime

switch($a)
{
 "q"    {
```

```
          funline("Die Systemzeit auf dem Computer $($objWMI.csname) ist:")
          [Management.ManagementDatetimeConverter]::`
          ToDateTime($localUTC)
        }
 "s"     {
          funline("Aktualisieren der Systemzeit auf dem Computer $computer ...")
          $objWMI.SetDateTime($date)
        }
 DEFAULT {
          funline("Die Systemzeit auf dem Computer $($objWMI.csname) ist:")
          [Management.ManagementDatetimeConverter]::`
          ToDateTime($localUTC)
       }
}
```

Aufzeichnen der Ergebnisse im Ereignisprotokoll

Das Skript *GetSetTimeWriteToEventLog.ps1* fügt eine Funktion zum Skript *GetSetTime.ps1* hinzu, die Ergebnisse der Verarbeitungsschritte im Anwendungsprotokoll auf dem Computer aufzuzeichnen, auf dem das Skript ausgeführt wird.

Weiterführende Informationen Das Erfassen von Ereignissen im Ereignisprotokoll wurde in Kapitel 3 „Verwalten von Protokollen“ beschrieben.

In diesem Kapitel werden lediglich die neuen Abschnitte erklärt, die zum Skript hinzugefügt werden.

Die erste Änderung im Skript *GetSetTimeWriteToEventLog.ps1* betrifft den Einsatz der automatischen Variablen *$erroractionpreference*. Legen Sie den Wert der Variablen auf *SilentlyContinue* fest, damit das Skript auch beim Auftreten eines Fehlers bis zum Ende ausgeführt wird. Mit diesem Schritt wird sichergestellt, dass das Skript auch dann Informationen im Ereignisprotokoll aufzeichnen kann, wenn beim Festlegen der Uhrzeit ein Fehler auftritt.

Hinweis Für die Variable *$erroractionpreference* können vier Werte angegeben werden: *SilentlyContinue* (Fehler werden nicht angezeigt), *Continue* (Fehler werden angezeigt), *Inquire* (Fehler werden angezeigt und das Skript wird nach einer Benutzereingabe fortgesetzt) und *Stop* (Fehler werden angezeigt und das Skript wird abgebrochen). Die angegebene Aktion hängt von der Wichtigkeit des Vorgangs (Bearbeiten der Registrierung oder Lesen der BIOS-Konfiguration) und der Fehlerbehandlung (ein Fehler wird erkannt und die Änderungen werden zurückgesetzt) ab. Der Standardwert für *$erroractionpreference* ist *Continue*.

Die folgende Codezeile legt die Aktion beim Auftreten eines Fehlers fest:

```
$erroractionpreference = "SilentlyContinue"
```

Die nächste Funktion ist *funlog*. Verwenden Sie im Skript *GetSetTimeWriteToEventLog.ps1* die Funktion *funlog*, um Fehlerinformationen im Anwendungsprotokoll aufzuzeichnen. Deklarieren Sie die Funktion und definieren Sie die Eingabevariable *$strErr*. Überprüfen Sie mit einer *if*-Anweisung, ob die Datenquelle *ps_script* definiert ist.

Definieren von Ereignisquellen

Sie können auch eine neue Ereignisquelle erstellen oder ein neues Ereignisprotokoll anlegen. Ich bevorzuge eine benutzerdefinierte Ereignisquelle namens *ps_script* und verwende diese für alle meine Skripts. Dies vereinfacht die Abfrage des Ereignisprotokolls nach bestimmten Ereignissen. Mit dem Skript *EventLogSpecificSource.ps1* können Sie die Einträge aus Ihren Skripts zurückgeben. Das Skript *EventLogSpecificSource.ps1* hat folgenden Inhalt:

EventLogSpecificSource.ps1

```
Get-EventLog -LogName application |
Where-Object { $_.source -eq "ps_script" }
```

Die Ereignisquellen werden in der Ereignisanzeige und im Dialogfeld **Aktuelles Protokoll filtern** aufgeführt. Sie können die Ereignisse basierend auf der Ereignisquelle filtern (siehe Abbildung 10.2).

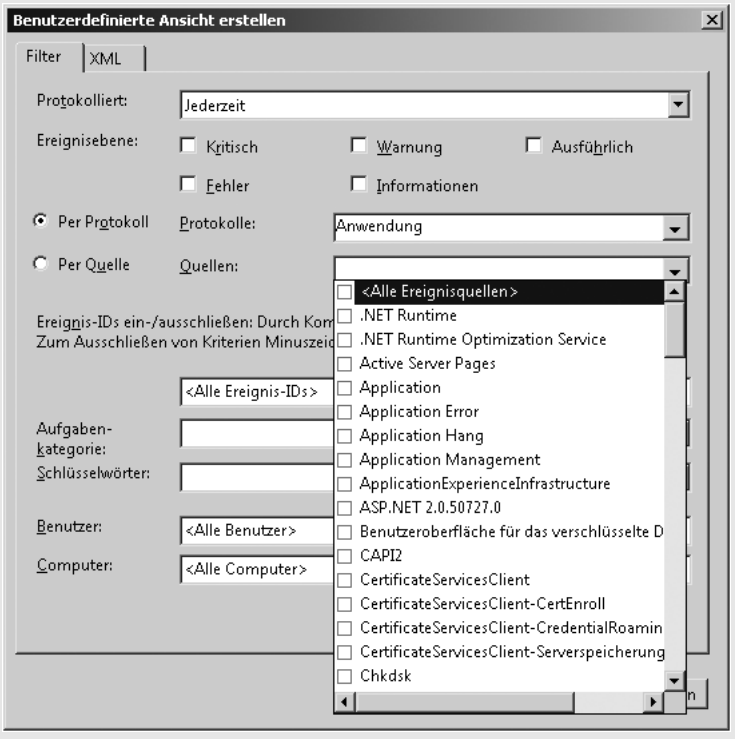


Abbildung 10.2 Vordefinierte Filter für Ereignisquellen

Da Sie die Ereignisse aus einer bestimmten Quelle abrufen können, sollten Sie die Ereignisquellen beschränken und möglichst nur eine Ereignisquelle verwenden.

Um zu überprüfen, ob die Datenquelle *ps_script* vorhanden ist, verwenden Sie die statische Methode *sourceExists()* aus der .NET Framework-Klasse *System.Diagnostics.Eventlog*. Wenn die Quelle *ps_script* existiert, fahren Sie mit dem nächsten Abschnitt der Funktion fort und zeichnen die über-

gebene Zeichenfolge als Ereignis im Anwendungsprotokoll auf. Wenn ein Ereignis in das Anwendungsprotokoll geschrieben wird, das auf der Quelle *ps_script* basiert, wird dieses im Ereignisprotokoll entsprechend angezeigt (siehe Abbildung 10.3).

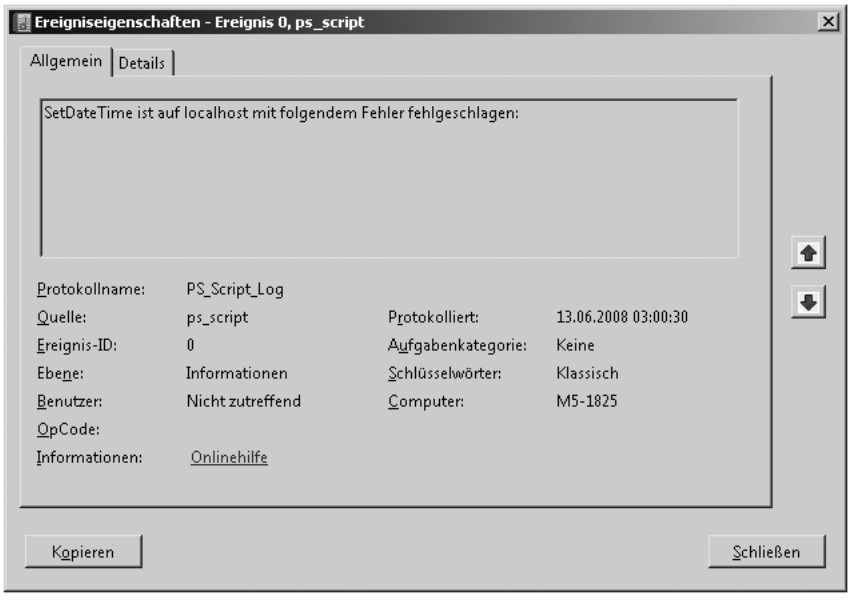


Abbildung 10.3 Ein Ereignis aus einer benutzerdefinierten Ereignisquelle im Anwendungsprotokoll

Wenn die Quelle nicht vorhanden ist, erstellen Sie mit der Methode *createEventSource()* der .NET Framework-Klasse *System.Diagnostics.Eventlog* eine neue Ereignisquelle.

Erstellen Sie anschließend mit dem Cmdlet **New-Object** eine neue Instanz der .NET Framework-Klasse *System.Diagnostics.Eventlog*. Geben Sie Application und einen Punkt in Anführungszeichen ein, um auf das Anwendungsprotokoll auf dem lokalen Computer zu verweisen. Weisen Sie das *eventlog*-Objekt der Variablen *$strLog* zu. Geben Sie als Quelle *ps_source* an und schreiben Sie mit der Methode *writeEntry()* den Inhalt der Variablen *$strErr* in das Anwendungsprotokoll. Die Funktion *funlog* ist wie folgt implementiert:

```
function funlog ($strErr)
{
 if(![system.diagnostics.eventlog]::sourceExists("ps_script","."))
  {
   $strLog = [system.diagnostics.eventlog]::CreateEventSource("ps_script",
                                                              "Application")
  }
 $strLog = new-object system.diagnostics.eventlog("application",".")
 $strLog.source = "ps_script"
 $strLog.writeEntry($strErr)
}
```

Legen Sie im *s*-Abschnitt der *switch*-Anweisung die Uhrzeit auf dem Zielcomputer fest. Rufen Sie hierzu die Funktion *funline* auf und geben Sie eine Meldung aus, dass die Systemzeit auf dem Computer aktualisiert wird. Rufen Sie die Methode *SetDateTime()* aus der WMI-Klasse *Win32_OperatingSystem* auf. Die Methode *SetDateTime()* erfordert einen Datum/Zeit-Wert im UTC-Format. Das vorherige

Skript enthält bereits den Code für die Umwandlung. Werten Sie den von der Methode zurückgegebenen Code aus. Ist der Wert 0, weisen Sie der Variablen *$strErr* eine Erfolgsmeldung zu und rufen Sie die Funktion *funlog* auf, um das Ergebnis in das Ereignisprotokoll zu schreiben. Sollte der zurückgegebene Code nicht 0 sein, geben Sie eine Fehlermeldung aus und rufen Sie die Funktion *funlog* auf, um die Informationen in das Ereignisprotokoll zu schreiben. Codeabschnitt:

```
"s"     {
          funline("Aktualisieren der Systemzeit auf dem Computer $computer ...")
          $strErr = $objWMI.SetDateTime($date)
          If($strErr.returnvalue -eq 0)
           {
            $strErr = "Aktualisieren der Systemzeit auf dem Computer $($computer) erfolgreich."
            }
            ELSE
            {
             $strErr = "Aktualisieren der Systemzeit auf dem Computer $($computer) fehlgeschlagen mit:`n" +
               $strErr.returnvalue
            }

          funlog($strErr)
         }
```

Das vollständige Skript *GetSetTimeWriteToEventLog.ps1* hat folgenden Inhalt:

GetSetTimeWriteToEventLog.ps1

```
param($computer="localhost", $a, $help)
$erroractionpreference = "SilentlyContinue"
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline += "=" }
    Write-Host -ForegroundColor yellow `n$strIN
    Write-Host -ForegroundColor darkYellow $funline
}

function funlog ($strErr)
{
 if(![system.diagnostics.eventlog]::sourceExists("ps_script","."))
  {
   $strLog = [system.diagnostics.eventlog]::CreateEventSource(
                                              "ps_script",
                                              "Application")
  }
 $strLog = new-object system.diagnostics.eventlog("application",".")
 $strLog.source = "ps_script"
 $strLog.writeEntry($strErr)
}

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GetSetTimewritetoeventlog.ps1
Ermittelt oder aktualisiert die Systemzeit des lokalen Computers oder eines Remotecomputers.
```

```
PARAMETER:
-computer     Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-a(ktion)     Legt fest, ob die aktuelle Zeit angezeigt oder geändert werden soll.
-help         Zeigt dieses Hilfethema an.

SYNTAX:
GetSetTimewritetoeventlog.ps1 -computer MunichServer

Zeigt die aktuelle Systemzeit eines Computers namens MunichServer an.

GetSetTimewritetoeventlog.ps1

Zeigt die aktuelle Systemzeit des Computers an.

GetSetTimewritetoeventlog.ps1 -a q

Zeigt die aktuelle Systemzeit des Computers an.

GetSetTimewritetoeventlog.ps1 -a q -computer MunichServer

Zeigt die aktuelle Systemzeit eines Computers namens MunichServer.

GetSetTimewritetoeventlog.ps1 -a s -computer MunichServer

Aktualisiert die Systemzeit eines Computers namens MunichServer.

GetSetTimewritetoeventlog.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help){funline("Ausgeben der Hilfeinformationen ...") ; funhelp }

$date = [Management.ManagementDatetimeConverter]::`
  ToDmtfDateTime($(get-date))

$objWMI = Get-WmiObject -ComputerName $computer `
  -Class win32_operatingsystem
$localUTC=$objwmi.localDateTime

switch($a)
{
 "q"    {
          funline("Die Systemzeit auf dem Computer ($objWMI.csname) ist:")
          [Management.ManagementDatetimeConverter]::`
          ToDateTime($localUTC)
        }
 "s"     {
          funline("Aktualisieren der Systemzeit auf dem Computer $computer ...")
          $strErr = $objWMI.SetDateTime($date)
           If($strErr.returnvalue -eq 0)
            {
```

```
            $strErr = "Aktualisieren der Systemzeit auf dem Computer $($computer) erfolgreich."
           }
          ELSE
           {
            $strErr = "Aktualisieren der Systemzeit auf dem Computer $($computer) fehlgeschlagen mit:`n" +
              $strErr.returnvalue
           }

         funlog($strErr)
        }
 DEFAULT {
         funline("Die Systemzeit auf dem Computer $($objWMI.csname) ist:")
         [Management.ManagementDatetimeConverter]::`
         ToDateTime($localUTC)
         }
}
```

Konfigurieren der Zeitquelle

Die Zeitquelle kann auf einem Computer entweder mit dem Befehl **Net Time** oder durch Bearbeiten der Registrierung konfiguriert werden. Die bevorzugte Methode ist der Befehl **Net Time**, da dieser remote ausgeführt werden kann. Sie sollten jedoch die Registrierung abfragen, um sicherzustellen, dass die Änderungen vorgenommen wurden. Die Registrierungsschlüssel sind in Abbildung 10.4 dargestellt.

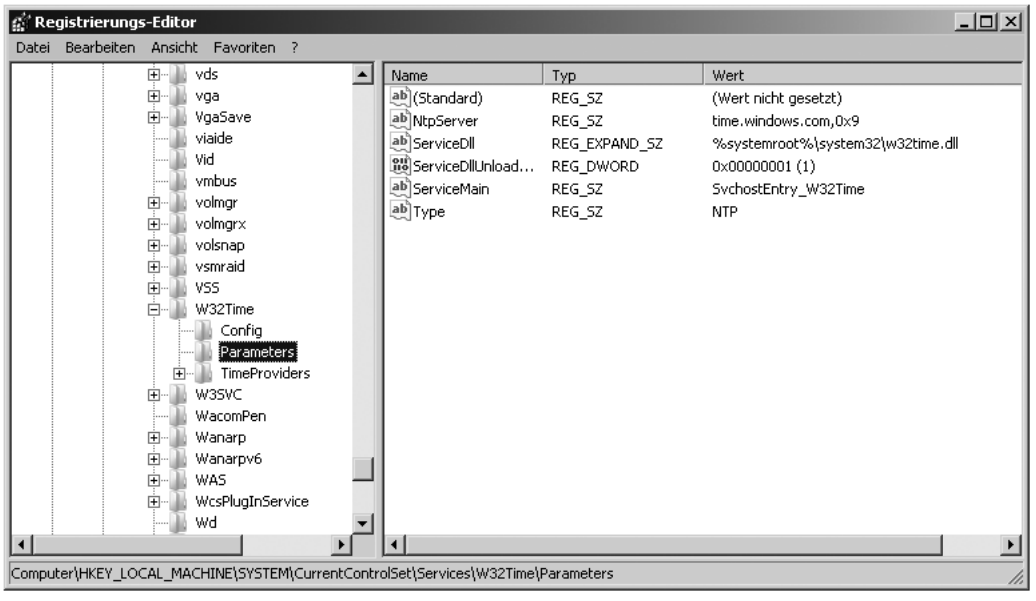


Abbildung 10.4 Die Zeitquelle in der Registrierung

Der Befehl Net Time

Definieren Sie im Skript *SetTimeSource.ps1* mit der *param*-Anweisung vier Befehlszeilenparameter. Der erste Parameter, **-computer**, legt fest, auf welchem Computer das Skript ausgeführt wird. Der zweite Parameter ist **-a**, um die auszuführende Aktion anzugeben. Der dritte Parameter ist **-timeserver**,

um den Namen des Zeitservers für den Computer festzulegen. Der vierte Parameter ist **-help**, um die Hilfe-informationen auszugeben. Codezeile:

```
param($computer="localhost",$a,$timeServer,$help)
```

Definieren Sie die Funktion *funhelp*, um eine Hilfemeldung anzuzeigen, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Erstellen Sie in der Funktion *funhelp* eine *here*-Zeichenfolge und weisen Sie diese der Variablen *$helpText* zu. Geben Sie in der *here*-Zeichenfolge die Beschreibung, Parameter und einige Syntaxbeispiele ein. Die Hilfe wird beim Ausführen des Skripts mit folgendem Befehl angezeigt:

```
PS C:\> .\SetTimeSource.ps1 -help ?
```

Nachdem Sie die *here*-Zeichenfolge erstellt haben, geben Sie den Text in der Variablen *$helpText* aus und beenden Sie das Skript. Die Funktion *funhelp* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: SetTimeSource.ps1
Ermittelt oder aktualisiert die aktuelle Zeitquelle des lokalen Computers oder eines Remotecomputers.

PARAMETER:
-computer   Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-a(ktion)   Gibt die auszuführende Aktion an < qt, qs, s >.
-timeServer Gibt den  zu verwendenden Zeitserver an.
-help       Zeigt dieses Hilfethema an.

SYNTAX:
SetTimeSource.ps1 -computer MunichServer

Zeigt die aktuelle Systemzeit eines Computers namens MunichServer an.

SetTimeSource.ps1

Zeigt die aktuelle Systemzeit des lokalen Computers an.

SetTimeSource.ps1 -computer MunichServer -a qs

Zeigt den aktuellen Zeitserver eines Computers namens MunichServer an.

SetTimeSource.ps1 -computer MunichServer -a qs -timeServer 192.168.2.5

Aktualisiert den Zeitserver eines Computers namens MunichServer
und registriert 192.168.2.5 als Zeitquelle.

SetTimeSource.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Sie müssen bestimmen, ob die Hilfe angezeigt werden soll. Überprüfen Sie zuerst, ob die Variable *$help* vorhanden ist. Wenn die Variable *$help* gefunden wird, wurde das Skript mit dem Parameter **-help** ausgeführt. Sollte die Variable nicht existieren, wurde das Skript ohne den Parameter ausgeführt. Ist die Variable *$help* vorhanden, rufen Sie entsprechend die Funktion *funhelp* auf. Codezeile:

```
if($help){funline("Ausgeben der Hilfeinformationen ...") ; funhelp }
```

Verwenden Sie eine *switch*-Anweisung, um den Wert der Variablen *$a* auszuwerten. Der Wert wird zugewiesen, wenn das Skript mit dem Parameter **-a** ausgeführt wird. In diesem Fall fragt das Skript die Systemzeit auf dem Computer ab, der vom Parameter **-computer** angegeben wird. Wenn für den Parameter **-computer** kein Wert angegeben wurde, wird das Skript auf dem lokalen Computer ausgeführt. Verwenden Sie den Wert **qs**, um festzulegen, dass der gegenwärtig verwendete SNTP-Server (Simple Network Time Protocol) abgefragt werden soll. Wenn das Skript mit dem Parameter **-a s** ausgeführt wird, legen Sie den Zeitserver fest. Befehl:

```
PS C:\> .\SetTimeSource.ps1 -computer Bonn -a qs -timeServer Bali
```

Sollte für den Parameter **-a** ein anderer Wert angegeben werden, führt die *switch*-Anweisung die Standardaktion aus und zeigt die aktuelle Zeit auf dem Computer an, der vom Parameter *$computer* festgelegt wird. Codesegment:

```
switch($a)
{
 "qt"    { net time \\$computer }
 "qs"    { net time \\$computer  /querySNTP}
 "s"     { net time \\$computer /setSNTP:$timeServer }
 DEFAULT { net time \\$computer }
}
```

Das vollständige Skript *SetTimeSource.ps1* hat folgenden Aufbau:

SetTimeSource.ps1

```
param($computer="localhost",$a,$timeServer,$help)

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: SetTimeSource.ps1
Ermittelt oder aktualisiert die aktuelle Zeitquelle des lokalen Computers oder eines Remotecomputers.

PARAMETER:
-computer   Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-a(ktion)   Gibt die auszuführende Aktion an < qt, qs, s >.
-timeServer Gibt den  zu verwendenden Zeitserver an.
-help       Zeigt dieses Hilfethema an.

SYNTAX:
SetTimeSource.ps1 -computer MunichServer

Zeigt die aktuelle Systemzeit eines Computers namens MunichServer an.

SetTimeSource.ps1

Zeigt die aktuelle Systemzeit des lokalen Computers an.

SetTimeSource.ps1 -computer MunichServer -a qs
```

```
Zeigt den aktuellen Zeitserver eines Computers namens MunichServer an.

SetTimeSource.ps1 -computer MunichServer -a qs -timeServer 192.168.2.5

Aktualisiert den Zeitserver eines Computers namens MunichServer
und registriert 192.168.2.5 als Zeitquelle.

SetTimeSource.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help){("Ausgeben der Hilfeinformationen ...") ; funhelp }

switch($a)
{
 "qt"    { net time \\$computer }
 "qs"    { net time \\$computer  /querySNTP}
 "s"     { net time \\$computer /setSNTP:$timeServer }
 DEFAULT { net time \\$computer }
}
```

Abfragen der Zeitquelle in der Registrierung

Im Skript *GetTimeSource.ps1* fragen Sie die Registrierung unter Verwendung der WMI-Klasse *StdRegProv* ab. Der Vorteil einer WMI-Abfrage ist, dass Sie diese remote ausführen können, wohingegen die *PSDrive*-Laufwerke *HLKM:* und *HKCU:* in Windows PowerShell Version 1.0 nur die lokale Abfrage unterstützen. Unter Verwendung der WMI-Klasse *StdRegProv* können Sie sowohl die Registrierung abfragen als auch Werte festlegen. Diese Vorgänge werden im Abschnitt „Konfigurieren des Bildschirmschoners“ näher beschrieben.

Definieren Sie im Skript *GetTimeSource.ps1* mit einer *param*-Anweisung zwei Befehlszeilenargumente. Der erste Argument ist der Parameter **-computer**. Dieses Argument legt den Computer fest, von dem die Zeitquelle abgerufen wird. Das zweite Argument ist der Parameter **-help**, um die Hilfe anzuzeigen. Codezeile:

```
param($computer="localhost", $help)
```

Definieren Sie anschließend die Funktion *funline*. Diese Funktion hat den Namen *funline2* und ist in der Datei *Funline2.ps1* gespeichert.

Wichtig Die Neuerung der Funktion *funline* im Skript *GetTimeSource.ps1* besteht darin, dass die Variable mit einer Referenz an die Funktion übergeben wird. Das heißt, dass der in der Funktion geänderte Wert der Variablen im Hauptskript zur Verfügung steht. Auf diese Art können Sie einen Wert aus der Funktion abrufen. Beachten Sie, dass Sie die Variable unter Verwendung der *[ref]*-Einschränkung in ein *PSReference*-Objekt konvertieren müssen.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Legen Sie in der Funktion *funline* die *[ref]*-Einschränkung für die Variable *$strIN* fest. Dies ermöglicht die Angabe einer Referenz anstatt eines Werts für die Variable. Der in der Funktion geänderte Wert der Variablen ist dann auch außerhalb der Funktion verfügbar. Die Codezeile zur Deklaration der Funktion *funline* lautet:

```
function funline ([ref]$strIN)
```

Sie müssen die Länge der an die Funktion übergebenen Zeichenfolge bestimmen. Überprüfen Sie hierzu den Wert der Variablen $strIN. Die *[ref]*-Einschränkung liefert ein *PSReference*-Objekt zurück. Um auf den Wert zuzugreifen, fragen Sie die Eigenschaft *Value* ab. Da *$strIN* ein Objekt enthält, sind Methoden und Eigenschaften vorhanden. Beispiel:

```
PS C:\> $strin = "hi"
PS C:\> [ref]$strin

Value
-----
hi


PS C:\> [ref]$strin | gm


   TypeName: System.Management.Automation.PSReference

Name        MemberType Definition
----        ---------- ----------
Equals      Method     System.Boolean Equals(Object obj)
GetHashCode Method     System.Int32 GetHashCode()
GetType     Method     System.Type GetType()
get_Value   Method     System.Object get_Value()
set_Value   Method     System.Void set_Value(Object value)
ToString    Method     System.String ToString()
Value       Property   System.Object Value {get;set;}
```

Überprüfen Sie die Codezeile, die die Länge der an die Funktion *funline* übergebenen Zeichenfolge bestimmt:

```
$num = $strIN.Value.length
```

Nachdem Sie die Länge der Zeichenfolge ermittelt haben, weisen Sie diesen Wert der Variablen *$num* zu. Erstellen Sie einen Unterstrich aus Gleichheitszeichen für die Zeilentrennung basierend auf diesem Wert. Weisen Sie die Zeichenfolge für die Zeilentrennung der Variablen *$funline* zu, fügen Sie *$funline* zur Eigenschaft *Value* von *$strIN* hinzu. Verknüpfen Sie also den bereits existierenden Wert mit der Zeilentrennung aus der Variablen *$funline*. Fügen Sie jedoch zuvor einen Zeilenumbruch hinzu, indem Sie das Sonderzeichen *`n* eingeben. Dieser Codeabschnitt aus der Funktion *funline* umfasst folgende Anweisungen:

```
{
 $num = $strIN.value.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline += "=" }
    $strIN.value = "$($strIN.value)`n" + $funline
}
```

Die nächste Funktion ist *funhelp*, die eine *here*-Zeichenfolge für die Hilfe umfasst. Weisen Sie die *here*-Zeichenfolge der Variablen *$helptext* zu. Geben Sie in der Zeichenfolge eine Beschreibung des Skripts, die vom Skript akzeptierten Parameter und Syntaxbeispiele an. Schließen Sie die *here*-Zeichenfolge,

geben Sie den Wert der Variablen *$helpText* aus und beenden Sie das Skript mit der *exit*-Anweisung. Die Funktion *funhelp* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GetTimeSource.ps1
Ermittelt die Zeitquelle des lokalen Computers oder eines Remotecomputers.

PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
GetTimeSource.ps1 -computer MunichServer

Zeigt die aktuelle Zeitquelle eines Computers namens MunichServer an.

GetTimeSource.ps1

Zeigt die aktuelle Zeitquelle des Computers an.

GetTimeSource.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Um zu bestimmen, ob die Hilfe angezeigt werden muss, verwenden Sie eine *if*-Anweisung und suchen Sie nach der Variablen *$help*. Wenn die Variable *$help* vorhanden ist, geben Sie eine Meldung aus, dass die Hilfe abgerufen wird, und führen Sie die Funktion *funhelp* aus. An die Funktion *funhelp* werden keine Argumente übergeben. Codeabschnitt:

```
if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
```

Deklarieren Sie vier Variablen, um die WMI-Klasse *StdRegProv* abzufragen. Die Variable *$hklm* hat den Wert *2147483650*. Dieser im WMI Software Development Kit (SDK) dokumentierte Wert wird vom *StdRegProv*-Anbieter verwendet, um auf die Registrierungsstruktur *HKEY_LOCAL_MACHINE* zu verweisen. Die Variable *$strKey* gibt den Registrierungsschlüssel an, der abgefragt wird. Für diese Variable muss ein Backslash angegeben werden. Die Variable *$strValue* gibt den abzufragenden Registrierungswert an. In diesem Beispiel überprüfen Sie den Registrierungswert *NtpServer* (siehe Abbildung 10.5).

Abbildung 10.5 Der Registrierungswert *NtpServer*

Die letzte Variable enthält das *System.Management.ManagementClass*-Objekt, das vom *[wmiclass]*-Accelerator zurückgegeben wird. Codeabschnitt:

```
$hklm = 2147483650
$strKey = "SYSTEM\CurrentControlSet\Services\W32Time\Parameters"
$strValue = "NtpServer"
$stdReg = [wmiclass]"\\$computer\root\default:stdregprov"
```

Da Sie nun über eine Kopie der WMI-Klasse *StdRegProv* verfügen, können Sie die Methode *GetStringValue()* verwenden. Die Methode *GetStringValue()* akzeptiert drei Parameter. Der erste Parameter gibt den numerischen Wert der Registrierungsstruktur aus dem WMI-SDK an. Der zweite Parameter verweist auf den Registrierungsschlüssel, der abgefragt wird. Der dritte Parameter identifiziert den Registrierungswert, der zurückgegeben werden soll. Weisen Sie den zurückgegebenen Wert der Variablen *$strTime* zu. Codezeile:

```
$strTime = $stdReg.GetStringValue($hklm,$strKey,$strValue)
```


Vorsicht Beachten Sie, dass *GetStringValue* im Gegensatz zur WMI-Klasse *StdRegProv* in VBScript nur drei Argumente akzeptiert. In VBScript geben Sie vier Variablen an, von denen die letzte den Rückgabewert enthält. Da sich die .NET-Verwaltungsklassen etwas vom VBSript-API unterscheiden, können Sie vorhandenen VBScript-Code nicht unmittelbar weiterverwenden.

Der abgefragte Registrierungswert wird der Variablen *sValue* zugewiesen. Wenn während der Abfrage ein Fehler auftritt, wird der Fehlercode in der Eigenschaft *ReturnValue* gespeichert. Analysieren Sie diese Informationen in einer *if*-Anweisung. Wenn die Eigenschaft *ReturnValue* den Wert 0 hat, geben Sie den Registrierungswert aus. Unterstreichen Sie den Wert mit der Funktion *funline*. Codeabschnitt:

```
if($strTime.returnvalue -eq 0)
 {
  $strOUT="$($strTime.sValue)"
  funline([ref]$strOut)
 }
```

Wenn ein Fehler auftritt, hat die Eigenschaft *ReturnValue* einen von 0 verschiedenen Wert, den Sie entsprechend ausgeben sollten. Codeabschnitt:

```
ELSE
 {
  $strOut="Ein Fehler ist aufgetreten: $($strTime.returnvalue)."
  funline([ref]$strOut)
 }
```

Unabhängig davon, ob das Skript erfolgreich ausgeführt wird, geben Sie das jeweilige Ergebnis mit dem Cmdlet **Write-Host** aus. Codeabschnitt:

```
Write-Host -foregroundcolor green "Zeitquelle auf dem Computer $computer:"
Write-Host -ForegroundColor cyan $strOut
```

Das vollständige Skript *GetTimeSource.ps1* hat folgenden Inhalt:

GetTimeSource.ps1

```
param($computer="localhost", $help)

function funline ([ref]$strIN)
{
 $num = $strIN.value.length
 for($i=1 ; $i -le $num ; $i++)
```

```
  { $funline += "=" }
    $strIN.value = "$($strIN.value)`n" + $funline
}

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GetTimeSource.ps1
Ermittelt die Zeitquelle des lokalen Computers oder eines Remotecomputers.

PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
GetTimeSource.ps1 -computer MunichServer

Zeigt die aktuelle Zeitquelle eines Computers namens MunichServer an.

GetTimeSource.ps1

Zeigt die aktuelle Zeitquelle des Computers an.

GetTimeSource.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }


$hklm = 2147483650
$strKey = "SYSTEM\CurrentControlSet\Services\W32Time\Parameters"
$strValue = "NtpServer"
$stdReg =  [wmiclass]"\\$computer\root\default:stdregprov"

$strTime = $stdReg.GetStringValue($hklm,$strKey,$strValue)

if($strTime.returnvalue -eq 0)
 {
  $strOUT="$($strTime.sValue)"
  funline([ref]$strOut)
 }
 ELSE
 {
  $strOut="Ein Fehler ist aufgetreten: $($strTime.returnvalue)."
  funline([ref]$strOut)
 }

Write-Host -foregroundcolor green "Zeitserver auf dem Computer $computer:"
Write-Host -ForegroundColor cyan $strout
```

Aktivieren von Benutzerkonten

Anders als Domänenkonten werden deaktivierte Benutzerkonten nicht oft erstellt. Lokale Benutzerkonten werden hauptsächlich für den Zugriff auf lokale Ressourcen oder für lokale Dienstkonten erstellt. Außer für Arbeitsgruppen werden lokale Benutzerkonten selten als Anmeldekonten verwendet. Das heißt jedoch nicht, dass lokale Benutzerkonten veraltet sind. Aufgrund der erweiterten Peer-zu-Peer-Funktionen in Windows Vista und der neuen Features von Windows Server 2008 sind lokale Benutzerkonten heute tatsächlich wichtiger als noch vor fünf Jahren.

Das lokale Administratorkonto ist nach der Installation von Windows Vista oder Windows Server 2008 standardmäßig deaktiviert (siehe Abbildung 10.6). Sie können dieses Konto aktivieren, um bestimmte Verwaltungsaufgaben auszuführen.

Abbildung 10.6 Das lokale Administratorkonto ist in Windows Vista standardmäßig deaktiviert

Mit dem Skript *EnableDisableUser.ps1* können Sie das Konto aktivieren, die erforderlichen Aufgaben ausführen und das lokale Administratorkonto anschließend wieder deaktivieren. Außerdem können Sie mit diesem Skript das Kennwort des lokalen Administrators ändern, indem Sie das Skript mit der Option zum Aktivieren von Benutzerkonten ausführen.

Das Skript *EnableDisableUser.ps1* beginnt mit einer *param*-Anweisung, die fünf Parameter definiert. Der erste Parameter ist **-computer**, um festzulegen, auf welchem Computer das Skript ausgeführt wird. Der Parameter **-computer** ist standardmäßig auf den lokalen Computer festgelegt. Der Parameter **-a** gibt die auszuführende Aktion an. Die Parameter **-user** und **-password** geben die Anmeldeinformationen des lokalen Benutzers an. Der Parameter **-help** dient der Anzeige von Hilfeinformationen. Codezeile:

```
param($computer="localhost", $a, $user, $password, $help)
```

Die Funktion *funhelp* zeigt die Hilfe an, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Die Funktion *funhelp* entspricht im Wesentlichen den anderen *funhelp*-Funktionen in diesem Kapitel. Die Funktion verwendet eine *here*-Zeichenfolge und weist die Informationen der Variablen *$helptext* zu. Nachdem der Inhalt der Variablen *$helptext* angezeigt wurde, wird das Skript beendet. Die Funktion *funhelp* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: EnableDisableUser.ps1
Aktiviert oder deaktiviert einen lokalen Benutzer auf dem lokalen Computer oder auf einem Remotecomputer.

PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-a(ktion) Gibt die auszuführende Aktion an < e (aktivieren) d (deaktivieren) >.
-user     Der Name des zu modifizierenden Benutzerkontos.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
EnableDisableUser.ps1
Generiert eine Fehlermeldung. Sie müssen einen Benutzernamen angeben.

EnableDisableUser.ps1 -computer MunichServer -user myUser
-password Passw0rd^&! -a e

Aktiviert den lokalen Benutzer namens myUser auf einem Computer namens MunichServer
mit dem Kennwort Passw0rd^&!

EnableDisableUser.ps1 -user myUser -a d
Deaktiviert einen lokalen Benutzer namens myUser auf dem lokalen Computer.

EnableDisableUser.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Deklarieren Sie als Nächstes zwei Variablen. Diese Variablen enthalten die *ADS_USER_FLAG_ENUM*-Enumerationswerte aus dem Windows SDK. Diese Werte aktivieren oder deaktivieren ein Benutzer-konto.

```
$EnableUser = 512
$DisableUser = 2
```

Hinweis Die *ADS_USER_FLAG_ENUM*-Enumerationswerte sind zwar im Windows SDK aufgeführt, aber ihre Verwendung ist nicht dokumentiert. Da Windows PowerShell die *IadsUser*-Schnittstelle nicht direkt unterstützt, können Sie nicht auf die boolesche Eigenschaft *AccountDisabled* zugreifen, die in VBScript verfügbar ist. Das Skript *EnableDisableUser.ps1* ist deshalb ein wichtiges Beispiel, da mit VBScript erstellte Skripts mit dem WinNT-Anbieter nicht funktionieren.

Nachdem Sie die beiden Variablen definiert haben, bestimmen Sie anhand der Variablen *$help*, ob die Hilfe angezeigt werden muss. (Sie können diese Aktion ausführen, bevor Sie die Variablen *$EnableUser* und *$DisableUser* festlegen.) Verwenden Sie die gleiche Codezeile wie im Skript *GetTimeSource.ps1*. Die Codezeile überprüft die Variable *$help* und gibt eine Zeichenfolge aus und ruft die Funktion *funhelp* auf, wenn die Variable *$help* vorhanden ist:

```
if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
```

Überprüfen Sie, ob die Variable *$user* vorhanden ist. Sollte die Variable nicht vorhanden sein, generieren Sie mit der *throw*-Anweisung eine Fehlermeldung.

Tipp Die *throw*-Anweisung ist in der Windows PowerShell-Dokumentation nicht beschrieben, sondern nur in einem Syntaxbeispiel angegeben. Die Anweisung ist einfacher als die Syntax if(xxx) { xxx ; exit }. Der Nachteil ist, dass die Ausgabe unübersichtlich ist.

Die Fehlermeldung besagt, dass ein Benutzername erforderlich ist. Geben Sie die Syntax zum Abrufen der Hilfe aus. Codebeispiel:

```
if(!$user)
      {
       $(Throw 'Sie müssen für $user einen Wert angeben.
       Für weitere Informationen führen Sie folgenden Befehl aus: EnableDisableUser.ps1 -help ?')
     }
```

Nachdem der Benutzername angegeben wurde, verwenden Sie den *[ADSI]*-Accelerator und den ADSI-Anbieter (Active Directory Services Interface) *WinNT*, um das Benutzerobjekt aus der SAM-Kontodatenbank des lokalen Computers abzurufen. Codezeile:

```
$ObjUser = [ADSI]"WinNT://$computer/$user"
```

Die *switch*-Anweisung wertet den Wert der Variablen *$a* aus, die die Aktion angibt, die das Skript ausführen soll. Beim Aktivieren des Benutzerkontos müssen Sie ein Kennwort festlegen. Geben Sie den Buchstaben *e* für den Parameter **-a** an, um das Benutzerkonto zu aktivieren. Suchen Sie mit der *if*-Anweisung nach einem Kennwort im Parameter *$password*. Sollte kein Kennwort vorhanden sein, lösen Sie eine Ausnahme aus und verweisen Sie den Benutzer an die Hilfe. Wenn das Kennwort angegeben ist, verwenden Sie die Methode *SetPassword*, um das Kennwort für das Benutzerobjekt festzulegen. Ändern Sie die Beschreibung in *"Aktiviertes Konto"* und geben Sie den entsprechenden Wert für die Eigenschaft *UserFlags* an. Rufen Sie anschließend die Methode *SetInfo()* auf, um die Änderungen in die SAM-Kontodatenbank zu übernehmen. Die *switch*-Anweisung lautet:

```
switch($a)
{
 "e" {
      if(!$password)
       {
           $(Throw 'Sie müssen für $password einen Wert angeben.
           Für weitere Informationen führen Sie folgenden Befehl aus: EnableDisableUser.ps1 -help ?')
       }
      $objUser.setpassword($password)
      $objUser.description = "Aktiviertes Konto"
      $objUser.userflags = $EnableUser
      $objUser.setinfo()
    }
```

Um ein Benutzerkonto zu deaktivieren, müssen Sie für *Userflags* den entsprechenden Wert festlegen und die Methode *SetInfo()* aufrufen. Ändern Sie die Eigenschaft *Description* in *"Deaktiviertes Konto"*. Führen Sie diese Aktion nur aus, wenn der Parameter **-a** den Wert **d** hat. Codeabschnitt:

```
 "d" {
      $objUser.description = "Deaktiviertes Konto"
      $objUser.userflags = $DisableUser
      $objUser.setinfo()
    }
```

Sollte der Parameter **-a** einen anderen Wert als **e** oder **d** haben, verwenden Sie die *DEFAULT*-Klausel der *switch*-Anweisung. Geben Sie eine Zeichenfolge aus, die den Benutzer an die Hilfe verweist. Codeabschnitt:

```
 DEFAULT
        {
          "Sie müssen einen Wert für die auszuführende Aktion angeben.
          Für weitere Informationen führen Sie folgenden Befehl aus: EnableDisableUser.ps1 -help ?"
         }
}
```

Das vollständige Skript *EnableDisableUser.ps1* hat folgenden Inhalt:

EnableDisableUser.ps1

```
param($computer="localhost", $a, $user, $password, $help)

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: EnableDisableUser.ps1
Aktiviert oder deaktiviert einen lokalen Benutzer auf dem lokalen Computer oder auf einem Remotecomputer.

PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-a(ktion) Gibt die auszuführende Aktion an < e (aktivieren) d (deaktivieren) >.
-user     Der Name des zu modifizierenden Benutzerkontos.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
EnableDisableUser.ps1
Generiert eine Fehlermeldung. Sie müssen einen Benutzernamen angeben.

EnableDisableUser.ps1 -computer MunichServer -user myUser
-password Passw0rd^&! -a e

Aktiviert den lokalen Benutzer namens myUser auf einem Computer namens MunichServer
mit dem Kennwort Passw0rd^&!

EnableDisableUser.ps1 -user myUser -a d
Deaktiviert einen lokalen Benutzer namens myUser auf dem lokalen Computer.

EnableDisableUser.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
```

```
$helpText
exit
}

$EnableUser = 512
$DisableUser = 2

if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }

if(!$user)
      {
       $(Throw 'Sie müssen für $user einen Wert angeben.
       Für weitere Informationen führen Sie folgenden Befehl aus: EnableDisableUser.ps1 -help ?')
     }

$ObjUser = [ADSI]"WinNT://$computer/$user"

switch($a)
{
 "e" {
      if(!$password)
       {
           $(Throw 'Sie müssen für $password einen Wert angeben.
           Für weitere Informationen führen Sie folgenden Befehl aus: EnableDisableUser.ps1 -help ?')
       }
      $objUser.setpassword($password)
      $objUser.description = "Aktiviertes Konto"
      $objUser.userflags = $EnableUser
      $objUser.setinfo()
    }
 "d" {
      $objUser.description = "Deaktiviertes Konto"
      $objUser.userflags = $DisableUser
      $objUser.setinfo()
    }
 DEFAULT
        {
          "Sie müssen einen Wert für die auszuführende Aktion angeben.
          Für weitere Informationen führen Sie folgenden Befehl aus: EnableDisableUser.ps1 -help ?"
         }
}
```

Erstellen eines lokalen Benutzerkontos

Sie können ein lokales Benutzerkonto mit dem Befehl **Net User** oder mit ADSI erstellen. Sie können auch die in Abbildung 10.7 dargestellten Tools verwenden.

Verwenden Sie den ADSI-Anbieter *WinNT* zum Erstellen lokaler Benutzer und Gruppen. Da lokale Benutzerkonten weniger Attribute als Domänenbenutzerkonten aufweisen, ist das Erstellen lokaler Konten relativ einfach.

Abbildung 10.7 Das neue Benutzertool in der Computerverwaltung

Erstellen eines lokalen Benutzers

Das Skript *CreateLocalUser.ps1* beginnt mit einer *param*-Anweisung, in der Sie vier Parameter definieren: **-computer**, **-user**, **-password** und **-help**. Codezeile:

```
param($computer="localhost", $user, $password, $help)
```

Der nächste Codeabschnitt definiert die Funktion *funhelp*, um die Hilfeinformationen auszugeben. Codeabschnitt:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: CreateLocalUser.ps1
Erstellt einen lokalen Benutzer auf dem lokalen Computer oder auf einem Remotecomputer.

PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-user     Der Name des zu erstellenden Benutzers.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
CreateLocalUser.ps1
Generiert eine Fehlermeldung. Sie müssen einen Benutzernamen angeben.

CreateLocalUser.ps1 -computer MunichServer -user myUser
 -password Passw0rd^&!

Erstellt einen lokalen Benutzer namens myUser auf einem Computer namens MunichServer
mit dem Kennwort Passw0rd^&!

CreateLocalUser.ps1 -user myUser -password Passw0rd^&!
with a password of Passw0rd^&!
```

```
Erstellt einen lokalen Benutzer namens myUser auf dem lokalen Computer
mit dem Kennwort Passw0rd^&!

CreateLocalUser.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Um zu bestimmen, ob die Hilfe angezeigt werden muss, suchen Sie nach der Variablen *$help*. Ist die Variable *$help* vorhanden, zeigen Sie die Hilfemeldung an, indem Sie die Funktion *funhelp* aufrufen. Codezeile:

```
if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
```

Sie müssen sicherstellen, dass die Parameter **-user** und **-password** Werte enthalten. Sie können auch die Kennwortlänge oder die Namenskonventionen überprüfen. In diesem Beispiel akzeptieren Sie jedoch die an das Skript übergebenen Benutzernamen und Kennworte ohne genauere Überprüfungen. Sollten diese Werte nicht angegeben sein, generieren Sie mit der *throw*-Anweisung eine Fehlermeldung und beenden Sie das Skript. Codeabschnitt:

```
if(!$user -or !$password)
      {
       $(Throw 'Für $user und $password sind Werte erforderlich.
       Für weitere Informationen führen Sie folgenden Befehl aus: CreateLocalUser.ps1 -help ?')
     }
```

Nachdem Sie sichergestellt haben, dass der Benutzername und das Kennwort an das Skript übergeben wurden, verwenden Sie den *[ADSI]*-Accelerator, um auf die Kontodatenbank des lokalen Computers zuzugreifen. Verwenden Sie die Methode *Create()*, um ein Benutzerkonto mit dem in der Variablen *$user* angegebenen Namen zu erstellen. Rufen Sie die Methode *SetPassword()* auf, um das Kennwort festzulegen. Übernehmen Sie die Änderungen anschließend mit der Methode *SetInfo()* in die Datenbank. Legen Sie die Eigenschaft *Description* fest und rufen Sie erneut die Methode *SetInfo()* auf. Codeabschnitt:

```
$objOu = [ADSI]"WinNT://$computer"
$objUser = $objOU.Create("User", $user)
$objUser.setpassword($password)
$objUser.SetInfo()
$objUser.description = "Testbenutzer"
$objUser.SetInfo()
```

Das vollständige Skript *CreateLocalUser.ps1* hat folgenden Inhalt:

CreateLocalUser.ps1

```
param($computer="localhost", $user, $password, $help)

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: CreateLocalUser.ps1
Erstellt einen lokalen Benutzer auf dem lokalen Computer oder auf einem Remotecomputer.
```

```
PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-user     Der Name des zu erstellenden Benutzers.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
CreateLocalUser.ps1
Generiert eine Fehlermeldung. Sie müssen einen Benutzernamen angeben.

CreateLocalUser.ps1 -computer MunichServer -user myUser
 -password Passw0rd^&!

Erstellt einen lokalen Benutzer namens myUser auf einem Computer namens MunichServer
mit dem Kennwort Passw0rd^&!

CreateLocalUser.ps1 -user myUser -password Passw0rd^&!
with a password of Passw0rd^&!

Erstellt einen lokalen Benutzer namens myUser auf dem lokalen Computer
mit dem Kennwort Passw0rd^&!

CreateLocalUser.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }

if(!$user -or !$password)
      {
       $(Throw 'Für $user und $password sind Werte erforderlich.
       Für weitere Informationen führen Sie folgenden Befehl aus: CreateLocalUser.ps1 -help ?')
     }

$objOu = [ADSI]"WinNT://$computer"
$objUser = $objOU.Create("User", $user)
$objUser.setpassword($password)
$objUser.SetInfo()
$objUser.description = "Testbenutzer"
$objUser.SetInfo()
```

Erstellen einer lokalen Benutzergruppe

Möglicherweise müssen Sie auf einem Windows Vista- oder Windows Server 2008-Computer lokale Gruppen erstellen, um auf lokale Ressourcen zuzugreifen, beispielsweise einen freigegebenen Drucker. In Abbildung 10.8 sind lokale Gruppen dargestellt. Lokale Gruppen werden auch für Arbeitsgruppen an Remotestandorten von Unternehmen verwendet. Das neue Gruppentool aus der **Computerverwaltung** ist in Abbildung 10.9 dargestellt.

Abbildung 10.8 Lokale Gruppen in der *Computerverwaltung*

Definieren Sie im Skript *CreateLocalGroup.ps1* mit einer *param*-Anweisung drei Parameter: **-computer**, **-group** und **-help**. Verwenden Sie den Parameter **-computer** mit dem Standardwert *localhost*. Codezeile:

```
param($computer="localhost", $group, $help)
```

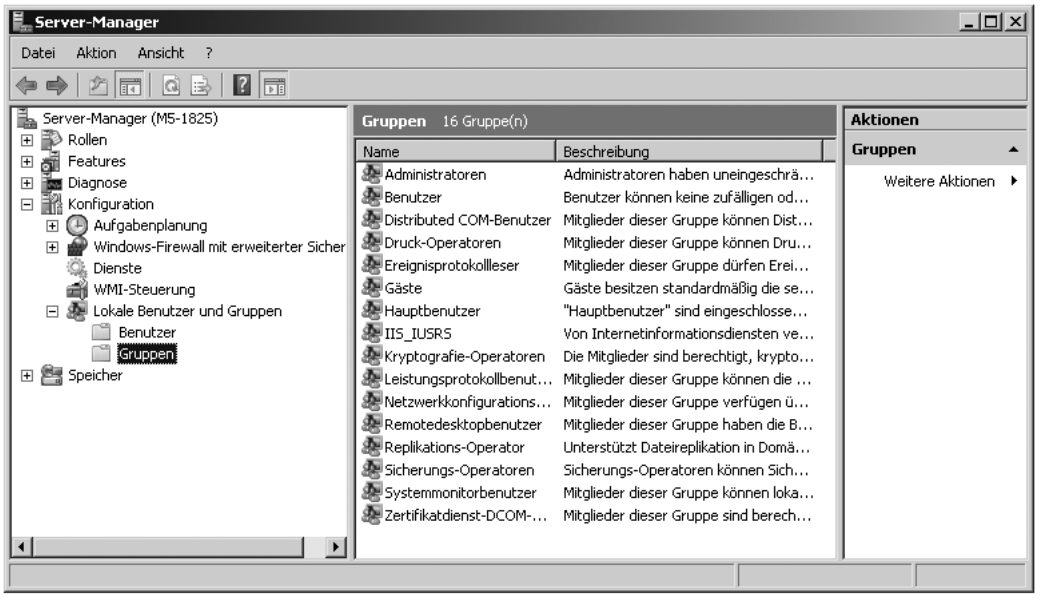


Abbildung 10.9 Das neue Gruppentool in der *Computerverwaltung*

Definieren Sie die Funktion *funhelp* und weisen Sie die *here*-Zeichenfolge der Variablen *$helpText* zu. Der Vorteil einer *here*-Zeichenfolge ist, dass Sie die Regeln für Anführungszeichen ignorieren und den Text so eingeben können, wie dieser angezeigt werden soll. Geben Sie eine Beschreibung des Skripts,

die Parameter und Syntaxbeispiele in der *here*-Zeichenfolge an. Geben Sie anschließend den Text der Variablen *$helpText* aus und beenden Sie das Skript. Die Funktion *funhelp* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: CreateLocalGroup.ps1
Erstellt eine lokale Gruppe auf dem lokalen Computer oder auf einem Remotecomputer.

PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-group    Der Name der zu erstellenden Gruppe.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
CreateLocalGroup.ps1
Generiert eine Fehlermeldung. Sie müssen einen Gruppennamen angeben.

CreateLocalGroup.ps1 -computer MunichServer -group MyGroup

Erstellt eine lokale Gruppe namens MyGroup auf einem Computer namens MunichServer.

CreateLocalGroup.ps1 -group Mygroup

Erstellt eine lokale Gruppe namens MyGroup auf dem lokalen Computer.

CreateLocalGroup.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Wenn die Variable *$help* vorhanden ist, geben Sie die Hilfe aus. Codezeile:

```
if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
```

Sie müssen sicherstellen, dass ein Gruppenname angegeben wird. Sollte die Variable *$group* nicht existieren, wurde der Gruppenname nicht angegeben. Generieren Sie in diesem Fall mit der *throw*-Anweisung eine Fehlermeldung. Codeabschnitt:

```
if(!$group)
      {
       $(Throw 'Sie müssen für $group einen Wert angeben.
       Für weitere Informationen führen Sie folgenden Befehl aus: CreateLocalGroup.ps1 -help ?')
     }
```

Der nächste Abschnitt des Skripts ist der *[ADSI]*-Abschnitt, der dem Code ähnlich ist, um ein lokales Benutzerkonto zu erstellen. Bis auf den Unterschied, dass für eine Gruppe kein Kennwort erforderlich ist, ist die Syntax beinahe identisch. Codeabschnitt:

```
$objOu = [ADSI]"WinNT://$computer"
$objUser = $objOU.Create("Group", $group)
$objUser.SetInfo()
$objUser.description = "Testgruppe"
$objUser.SetInfo()
```

Das vollständige Skript *CreateLocalGroup.ps1* hat folgenden Inhalt:

CreateLocalGroup.ps1

```
param($computer="localhost", $group, $help)

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: CreateLocalGroup.ps1
Erstellt eine lokale Gruppe auf dem lokalen Computer oder auf einem Remotecomputer.

PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-group    Der Name der zu erstellenden Gruppe.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
CreateLocalGroup.ps1
Generiert eine Fehlermeldung. Sie müssen einen Gruppennamen angeben.

CreateLocalGroup.ps1 -computer MunichServer -group MyGroup

Erstellt eine lokale Gruppe namens MyGroup auf einem Computer namens MunichServer.

CreateLocalGroup.ps1 -group Mygroup

Erstellt eine lokale Gruppe namens MyGroup auf dem lokalen Computer.

CreateLocalGroup.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }

if(!$group)
      {
       $(Throw 'Sie müssen für $group einen Wert angeben.
       Für weitere Informationen führen Sie folgenden Befehl aus: CreateLocalGroup.ps1 -help ?')
     }

$objOu = [ADSI]"WinNT://$computer"
$objUser = $objOU.Create("Group", $group)
$objUser.SetInfo()
$objUser.description = "Testgruppe"
$objUser.SetInfo()
```

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Konfigurieren des Bildschirmschoners

In Kapitel 9 „Konfigurieren der Desktopeinstellungen“ wurde das Abfragen der verschiedenen Informationen zu Bildschirmschonern erklärt. In diesem Abschnitt ändern Sie diese Informationen. Dies ist insbesondere für Windows Server 2008 Server Core wichtig. Das Bearbeiten der Registrierung ist neben Gruppenrichtlinien die einzige Methode, um einen Bildschirmschoner in Windows Server 2008 Server Core zu konfigurieren. Sie können auch das Bildschirmschonertool aus der Systemsteuerung verwenden, aber dieses Tool kann nicht remote ausgeführt werden (siehe Abbildung 10.10).

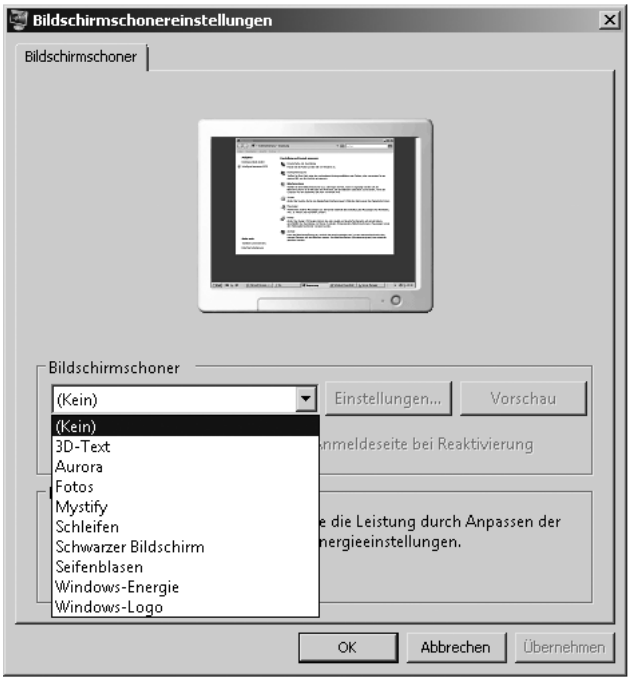


Abbildung 10.10 Das Dialogfeld *Bildschirmschoner* in der Systemsteuerung

In diesem Abschnitt wird das Auswählen und Aktivieren eines Bildschirmschoners sowie das Festlegen eines Timeoutwerts und eines Kennworts für den Bildschirmschoner erklärt. Sie erfahren außerdem, wie Sie diese Einstellungen wieder deaktivieren können.

Das Skript *ConfigureScreenSaver.ps1* akzeptiert mehrere Parameter: Den Namen des Computers, die auszuführende Aktion, den Wert für die Aktion und den Parameter **-help**. Codezeile:

```
param($computer="localhost", $a, $v, $help)
```

Die nächste Funktion ist *funline*, die Sie bereits zuvor verwendet haben. Diese Funktion ist eigentlich *funline2*, da diese eine Referenz akzeptiert. Sie können den Wert der Variablen ändern und die Variable anschließend an das Skript übergeben. Da die Variable *$strIN* referenziert ist, müssen Sie die Eigenschaft *Value* abfragen, um die Länge des Werts der Variablen zu bestimmen. Erstellen Sie anschließend die Unterstreichung aus Gleichheitszeichen für die Zeilentrennung und verknüpfen Sie die Eingabezeichenfolge mit der Zeilentrennung. Die Funktion *funline* ist wie folgt implementiert:

```
function funline ([ref]$strIN)
{
 $num = $strIN.value.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline += "=" }
    $strIN.value = "$($strIN.value)`n" + $funline
}
```

Die nächste Funktion ist *funhelp*, die den anderen Hilfefunktionen ähnlich ist. Erstellen Sie eine Hilfezeichenfolge und weisen Sie diese der Variablen *$helpText* zu. Geben Sie anschließend den Inhalt der Variablen aus und beenden Sie das Skript. Hilfetexte sind für Skripts mit mehreren möglichen Verarbeitungsoptionen wichtig. Die Funktion *funhelp* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ConfigureScreenSaver.ps1
Konfiguriert die Bildschirmschonereinstellungen auf dem lokalen Computer oder auf einem Remotecomputer.

PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-a(ktion) Gibt die auszuführende Aktion an: < q (abfragen), ex (ausführen), at (aktivieren),
          se (sichern), to (Timeoutwert festlegen) >.
-v(alue)  Der Wert für die angegebene Aktion (außer bei Abfragen).
-help     Zeigt dieses Hilfethema an.

SYNTAX:
ConfigureScreenSaver.ps1 -computer MunichServer -a ex -v bubbles.scr

Konfiguriert den Bildschirmschoner auf einem Computer namens MunichServer.
Die ausführbare Datei des Bildschirmschoners ist bubbles.scr.

ConfigureScreenSaver.ps1  -a se -v 1

Sichert den Bildschirmschoner auf dem lokalen Computer.
Dabei handelt es sich um den bereits konfigurierten Bildschirmschoner.

ConfigureScreenSaver.ps1  -a at -v 1

Aktiviert den Bildschirmschoner auf dem lokalen Computer.
Dabei handelt es sich um den bereits konfigurierten Bildschirmschoner.

ConfigureScreenSaver.ps1  -a to -v 300

Konfiguriert den Bildschirmschoner auf dem lokalen Computer mit einem Timeoutwert
von 5 Minuten. Dabei handelt es sich um den bereits konfigurierten Bildschirmschoner.

ConfigureScreenSaver.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Die Funktion *funeval* wertet die Ergebnisse der Änderungen aus. Anstatt den gleichen Code mehrmals zu kopieren, können Sie eine zentrale Funktion erstellen. Die Funktion *funeval* akzeptiert als Eingabeparameter die Variable *$strRTN*. Diese Variable enthält den von den WMI-Methoden zurückgegebenen Wert. Der Wert 0 zeigt an, dass kein Fehler aufgetreten ist. Geben Sie eine entsprechende Meldung in Grün aus. Geben Sie für jeden anderen Wert einen Fehlercode in Rot aus. Die Funktion *funeval* ist wie folgt implementiert:

```
function funeval ($strRTN)
{
 if($strRTN.returnvalue -eq 0)
  { Write-Host -ForegroundColor green "Erfolgreich abgeschlossen." }
 ELSE
  { Write-Host -ForegroundColor red "Fehlgeschlagen mit: $($strRTN.returnvalue)!" }
}
```

Überprüfen Sie, ob Sie die Hilfe anzeigen müssen, indem Sie die Variable *$help* analysieren. Wenn die Variable vorhanden ist, geben Sie eine Meldung aus, dass die Hilfe angezeigt wird, und rufen Sie die Funktion *funhelp* auf. Codezeile:

```
if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
```

Sie müssen nun einige Variablen definieren, um die Methodenaufrufe in der *switch*-Anweisung zu steuern. Die Variable *$hkcu* ist auf einen Wert festgelegt, der die *HKEY_CURRENT_USER*-Registrierungsstruktur repräsentiert. Definieren Sie den Registrierungsschlüssel C*ontrol Pane\Desktop* für die Abfrage. Weisen Sie diese Zeichenfolge der Variablen *$strKey* zu und definieren Sie die gewünschten Registrierungswerte. Weisen Sie jeder Variablen ein Array zu, damit Sie eine Variable für die Registrierungsabfrage und die Ausgabe einer Zeichenfolge verwenden können. Instantiieren Sie anschließend die WMI-Klasse *StdRegProv*. Codeabschnitt:

```
$hkcu = 2147483649  # Numerischer Referenzwert für HKCU aus dem WMI-SDK
$strKey = "Control Panel\Desktop"
$strExe = "SCRNSAVE.EXE", "Ausführbare Datei des Bildschirmschoners registriert"
$blnAct = "ScreenSaveActive", "Bildschirmschoner aktiviert"
$blnSec = "ScreenSaverIsSecure", "Bildschirmschoner gesichert"
$intTim = "ScreenSaveTimeOut", "TimeOut-Wert des Bildschirmschoners konfiguriert"
$stdReg =  [wmiclass]"\\$computer\root\default:stdregprov"
```

Der letzte Abschnitt des Skripts umfasst eine *switch*-Anweisung, mit der Sie mehrere Aktionen ausführen können. Werten Sie den Wert der Variablen *$a* aus. Wenn die Variable den Wert **q** hat, werden die im Referenzabschnitt des Skripts definierten vier Registrierungsschlüssel abgefragt. Erstellen Sie hierzu aus den vier Variablen ein Array und durchlaufen Sie das Array mit einer *foreach*-Anweisung. Rufen Sie den Zeichenfolgenwert aus dem Registrierungswert ab und geben Sie das Element *0* für die Abfrage an. Werten Sie den zurückgegebenen Wert aus und zeigen Sie diesen an. Wenn ein Fehler auftritt, geben Sie diesen ebenfalls aus. Die *switch*-Anweisung lautet:

```
switch($a)
{
 "q" {
      $aryValue = $strExe, $blnAct, $blnSec, $intTim
      foreach($strValue in $aryValue)
      {
       $strRTN = $stdReg.GetStringValue($hkcu,$strKey,$strValue[0])
        if($strRTN.returnvalue -eq 0)
         {
          $strOUT="$($strRTN.sValue)"
          funline([ref]$strOut)
```

```
        }
       ELSE
        {
         $strOut="Es ist ein Fehler aufgetreten: $($strRTN.returnvalue)."
         funline([ref]$strOut)
        }
        Write-Host -foregroundcolor green "$($strValue[1]) auf $computer:"
        Write-Host -ForegroundColor cyan $strout
     }
  }
```

Wenn Sie **ex** im Parameter **-a** angeben, wird der Bildschirmschoner als ausführbare Datei festgelegt. Die Bildschirmschoner für Windows Vista und Windows Server 2008 sind im hart codierten Pfad *C:\Windows\System32* gespeichert. Geben Sie den Namen des Bildschirmschoners an, beispielsweise *bubbles.scr*. Geben Sie eine Statusmeldung aus, dass der Wert im Element *$strexe* verwendet wird, und rufen Sie die Funktion *funeval* auf. Die *switch*-Anweisung lautet:

```
"ex" {
      $v = "C:\Windows\System32\$v"
      $strRTN = $stdReg.SetStringValue($hkcu,$strKey,$strExe[0],$v)
     "Registrieren von $($strExe[1]) ... "
     funeval($strRTN)
    }
```

Legen Sie als Nächstes fest, ob der Bildschirmschoner aktiviert werden soll. Da dies ein boolescher Wert ist, geben Sie 1 oder 0 an, wenn das Skript mit dem Parameter **-a** ausgeführt wird und dieser den Wert **at** hat. Rufen Sie die Methode *setStringValue()* aus der WMI-Klasse *StdRegProv* auf und schreiben Sie die Informationen in die Registrierung. Rufen Sie anschließend die Funktion *funeval* auf. Beispiel:

```
"at" {
     $strRTN = $stdReg.SetStringValue($hkcu,$strKey,$blnAct[0],$v)
     "Registrieren von $($blnAct[1]) ... "
     funeval($strRTN)
     }
```

Sie können auch festlegen, ob der Bildschirmschoner gesichert ist. Geben Sie hierzu **se** im Parameter **-a** und **1** oder **0** im Parameter **-v** an. Schreiben Sie die Informationen in die Registrierung und rufen Sie die Funktion *funeval* auf. Die *switch*-Anweisung lautet:

```
"se" {
     $strRTN = $stdReg.SetStringValue($hkcu,$strKey,$blnSec[0],$v)
     "Registrieren von $($blnSec[1]) ... "
     funeval($strRTN)
     }
```

Legen Sie anschließend einen Timeoutwert für den Bildschirmschoner fest. Dieser Wert wird in Sekunden angegeben. Geben Sie **se** im Parameter **-a** und die Anzahl der Sekunden im Parameter **-v** an. Codeabschnitt:

```
"to" {
     $strRTN = $stdReg.SetStringValue($hkcu,$strKey,$intTim[0],$v)
     "Registrieren von $($intTim[1]) ... "
     funeval($strRTN)
     }
}
```

Das vollständige Skript *ConfigureScreenSaver.ps1* hat folgenden Inhalt:

ConfigureScreenSaver.ps1

```
param($computer="localhost", $a, $v, $help)

function funline ([ref]$strIN)
{
 $num = $strIN.value.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline += "=" }
    $strIN.value = "$($strIN.value)`n" + $funline
}

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ConfigureScreenSaver.ps1
Konfiguriert die Bildschirmschonereinstellungen auf dem lokalen Computer oder auf einem Remotecomputer.

PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-a(ktion) Gibt die auszuführende Aktion an: < q (abfragen), ex (ausführen), at (aktivieren),
          se (sichern), to (Timeoutwert festlegen) >.
-v(alue)  Der Wert für die angegebene Aktion (außer bei Abfragen).
-help     Zeigt dieses Hilfethema an.

SYNTAX:
ConfigureScreenSaver.ps1 -computer MunichServer -a ex -v bubbles.scr

Konfiguriert den Bildschirmschoner auf einem Computer namens MunichServer.
Die ausführbare Datei des Bildschirmschoners ist bubbles.scr.

ConfigureScreenSaver.ps1  -a se -v 1

Sichert den Bildschirmschoner auf dem lokalen Computer.
Dabei handelt es sich um den bereits konfigurierten Bildschirmschoner.

ConfigureScreenSaver.ps1  -a at -v 1

Aktiviert den Bildschirmschoner auf dem lokalen Computer.
Dabei handelt es sich um den bereits konfigurierten Bildschirmschoner.

ConfigureScreenSaver.ps1  -a to -v 300

Konfiguriert den Bildschirmschoner auf dem lokalen Computer mit einem Timeoutwert
von 5 Minuten. Dabei handelt es sich um den bereits konfigurierten Bildschirmschoner.

ConfigureScreenSaver.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

```
function funeval ($strRTN)
{
 if($strRTN.returnvalue -eq 0)
  { Write-Host -ForegroundColor green "Erfolgreich abgeschlossen." }
 ELSE
  { Write-Host -ForegroundColor red "Fehlgeschlagen mit: $($strRTN.returnvalue)." }
}
if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }

$hkcu = 2147483649  # Numerischer Referenzwert für HKCU aus dem WMI-SDK
$strKey = "Control Panel\Desktop"
$strExe = "SCRNSAVE.EXE", "Ausführbare Datei des Bildschirmschoners registriert"
$blnAct = "ScreenSaveActive", "Bildschirmschoner aktiviert"
$blnSec = "ScreenSaverIsSecure", "Bildschirmschoner gesichert"
$intTim = "ScreenSaveTimeOut", "TimeOut-Wert des Bildschirmschoners konfiguriert"
$stdReg =  [wmiclass]"\\$computer\root\default:stdregprov"


switch($a)
{
 "q" {
      $aryValue = $strExe, $blnAct, $blnSec, $intTim
      foreach($strValue in $aryValue)
      {
       $strRTN = $stdReg.GetStringValue($hkcu,$strKey,$strValue[0])
        if($strRTN.returnvalue -eq 0)
         {
          $strOUT="$($strRTN.sValue)"
          funline([ref]$strOut)
         }
        ELSE
         {
          $strOut="Fehlgeschlagen mit: $($strRTN.returnvalue)."
          funline([ref]$strOut)
         }
        Write-Host -foregroundcolor green "$($strValue[1]) auf $computer"
        Write-Host -ForegroundColor cyan $strout
      }
   }
 "ex"
 {
       $v = "C:\Windows\System32\$v"
       $strRTN = $stdReg.SetStringValue($hkcu,$strKey,$strExe[0],$v)

      "Registrieren von $($strExe[1]) ... "
      funeval($strRTN)
   }
 "at"
 {
    $strRTN = $stdReg.SetStringValue($hkcu,$strKey,$blnAct[0],$v)
    "Registrieren von $($blnAct[1]) ... "
    funeval($strRTN)
   }
 "se"
 {
    $strRTN = $stdReg.SetStringValue($hkcu,$strKey,$blnSec[0],$v)
```

```
    "Registrieren von $($blnSec[1]) ... "
    funeval($strRTN)
  }
 "to"
 {
  $strRTN = $stdReg.SetStringValue($hkcu,$strKey,$intTim[0],$v)
  "Registrieren von $($intTim[1]) ... "
  funeval($strRTN)
  }
}
```

Umbenennen des Computers

Im Anschluss an die Installation von Windows Server 2008 oder Windows Vista müssen Sie den Computer möglicherweise umbenennen. Bei einer automatisierten Installation können Sie den Computernamen in der Antwortdatei angeben. Computer müssen jedoch dennoch gelegentlich umbenannt werden. Verwenden Sie hierzu das Skript *RenameComputer.ps1*.

Das Skript *RenameComputer.ps1* beginnt mit einer *param*-Anweisung. Die Anweisung unterscheidet sich hier etwas, da Sie mehrere Standardwerte verwenden. Legen Sie den Parameter **-computer** auf den loklaen Computer und den Parameter **-user** auf Administrator fest. Die Parameter **-password** und **-newname** werden nicht festgelegt. Außerdem ist der Parameter **-help** verfügbar. Beachten Sie, dass für eine lokale WMI-Verbindung alternative Anmeldeinformationen unzulässig sind. Die *param*-Anweisung lautet:

```
param(
      $computer="localhost",
      $newName,
      $user = "administrator",
      $password,
      $help
     )
```

Die Funktion *funhelp* zeigt die Hilfe an. Erstellen Sie eine *here*-Zeichenfolge mit der Beschreibung, den Parametern und Syntaxbeispielen. Weisen Sie die *here*-Zeichenfolge der Variablen *$helptext* zu. Geben Sie den Wert in der Variablen *$helptext* aus und beenden Sie das Skript. Die Funktion *funhelp* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: RenameComputer.ps1
Ändert den Namen des lokalen Computers oder eines Remotecomputers.

PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-newname  Gibt den neuen Namen des Computers an.
-user     Der Benutzername.
-password Das Kennwort des Benutzers.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
RenameComputer.ps1 -computer MunichServer -newname BerlinServer
```

```
Benennt einen Computer namens MunichServer in BerlinServer um.

RenameComputer.ps1 -computer MunichServer -newname BerlinServer
-user munich\admin -password MyPassword

Benennt einen Computer namens MunichServer in BerlinServer um. Verwendet für diesen
Vorgang das Konto admin aus der Domäne munich mit dem Kennwort MyPassword.

RenameComputer.ps1

Generiert eine Fehlermeldung. Der neue Computername muss angegeben werden.

RenameComputer.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Sie müssen bestimmen, ob die Hilfe angezeigt werden soll. Ist die Variable *$help* vorhanden, geben Sie eine Meldung aus und rufen Sie die Funktion *funhelp* auf. Codezeile:

```
if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
```

Wenn Sie das Skript remote ausführen, ist der Wert der Variablen *$computer* nicht *localhost*. Verwenden Sie in diesem Fall andere Anmeldeinformationen für das Skript. Geben Sie im Parameter **-credential** des Cmdlets **Get-WmiObject** den Benutzernamen anhand der Parametervariablen *$user* an und rufen Sie die Methode *Rename()* auf. Codeabschnitt:

```
if($computer -ne "localhost")
 {
  $objWMI = Get-WmiObject -Class Win32_Computersystem `
  -computername $computer -credential $user
  $objWMI.rename($newName)
 }
```

Wenn Sie das Skript lokal ausführen, hat die Variable *$computer* den Wert *localhost*. Rufen Sie in diesem Fall das Cmdlet **Get-WmiObject** ohne den Parameter **-credential** auf, um einen Fehler zu vermeiden, der bei der Angabe von anderen Anmeldeinformationen für lokale WMI-Verbindungen auftritt. Verwenden Sie das Cmdlet **Get-WmiObject** erneut und rufen Sie die Methode *Rename()* der WMI-Klasse *Win32_ComputerSystem* auf. Beispiel:

```
ELSE
 {
  $objWMI = Get-WmiObject -Class Win32_Computersystem `
  -computername $computer
  $objWMI.rename($newName)
 }
```

Das vollständige Skript *RenameComputer.ps1* hat folgenden Aufbau:

RenameComputer.ps1

```
param(
      $computer="localhost",
      $newName,
      $user = "administrator",
```

```
      $password,
      $help
     )

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: RenameComputer.ps1
Ändert den Namen des lokalen Computers oder eines Remotecomputers.

PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-newname  Gibt den neuen Namen des Computers an.
-user     Der Benutzername.
-password Das Kennwort des Benutzers.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
RenameComputer.ps1 -computer MunichServer -newname BerlinServer

Benennt einen Computer namens MunichServer in BerlinServer um.

RenameComputer.ps1 -computer MunichServer -newname BerlinServer
-user munich\admin -password MyPassword

Benennt einen Computer namens MunichServer in BerlinServer um. Verwendet für diesen
Vorgang das Konto admin aus der Domäne munich mit dem Kennwort MyPassword.

RenameComputer.ps1

Generiert eine Fehlermeldung. Der neue Computername muss angegeben werden.

RenameComputer.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help) { "Ausgeben der Hilfeinformationen ..." ; funhelp }

if($computer -ne "localhost")
 {
  $objWMI = Get-WmiObject -Class Win32_Computersystem `
  -computername $computer -credential $user
  $objWMI.rename($newName)
 }
ELSE
 {
  $objWMI = Get-WmiObject -Class Win32_Computersystem `
  -computername $computer
  $objWMI.rename($newName)
 }
```

Herunterfahren oder Neustarten eines Remotecomputers

Um einen Computer umzubenennen oder in eine Domäne aufzunehmen, müssen Sie in der Lage sein, den Computer herunterzufahren oder neu zu starten. Sie können diese Aufgaben mit WMI ausführen. Verwenden Sie beispielsweise das Skript *ShutdownRebootComputer.ps1* mit den Methoden *Shutdown()* und *Reboot()* aus der WMI-Klasse *Win32_OperatingSystem*. Um zu bestimmen, welche Methode aufgerufen wird, geben Sie die entsprechende Aktion mit dem Parameter **-a** an.

Das Skript *ShutdownRebootComputer.ps1* beginnt mit einer *param*-Anweisung. Geben Sie die folgenden Standardwerte an:

```
param(
      $computer="localhost",
      $user = "administrator",
      $password,
      $a,
      $help
    )
```

Geben Sie als Nächstes mit der Funktion *funhelp* eine Hilfemeldung aus. Die Funktion *funhelp* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ShutdownRebootComputer.ps1
Fährt den lokalen Computer oder einen Remotecomputer herunter oder startet diesen neu.

PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-user     Der Benutzername.
-password Das Kennwort des Benutzers.
-a(ktion) Die auszuführende Aktion: < s (herunterfahren), r (neu starten) >.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
ShutdownRebootComputer.ps1-computer MunichServer -a s

Fährt einen Remotecomputer namens MunichServer herunter.

ShutdownRebootComputer.ps1-computer MunichServer -a r
-user munich\admin -password MyPassword

Startet einen Computer namens MunichServer neu. Verwendet dabei
das Benutzerkonto admin aus der Domäne munich mit dem Kennwort MyPassword.

ShutdownRebootComputer.ps1

Zeigt eine Meldung mit Hinweisen zur Hilfe an.

ShutdownRebootComputer.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Überprüfen Sie, ob die Variable *$help* vorhanden ist. Gibt es die Variable, rufen Sie die Funktion *funhelp* auf. Codezeile:

```
if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
```

Definieren Sie anschließend eine *switch*-Anweisung. Der erste Wert ist **s** für das Herunterfahren des Computers. Wenn das Skript mit dem Argument **-a s** ausgeführt wird, rufen Sie die Methode *Shutdown()* des *Win32_OperatingSystem*-Objekts auf. Um den Server herunterzufahren, muss Ihr Konto über die entsprechende Berechtigung verfügen. Um Ihrem Konto diese Berechtigung zu gewähren, legen Sie die Eigenschaft *EnablePrivileges* auf *$true* fest. Hinweis: Suchen Sie auch nach *localhost* und definieren den **Get-WmiObject**-Abschnitt jeweils einmal mit und ohne Anmeldeinformationen.

```
switch($a)
{
 "s" {
      if($computer -ne "localhost")
      {
       $objWMI = Get-WmiObject -Class Win32_operatingsystem `
       -computername $computer -credential $user
       $objWMI.psbase.Scope.Options.EnablePrivileges = $true
       $objWMI.shutdown()
      }
      ELSE
      {
       $objWMI = Get-WmiObject -Class Win32_operatingsystem `
       -computername $computer
       $objWMI.psbase.Scope.Options.EnablePrivileges = $true
       $objWMI.shutdown()
      }
     }
```

Überprüfen Sie danach, ob *$a* den Wert **r** hat. Wenn der Wert **r** angegeben wurde, starten Sie den Server neu. Denken Sie daran, dass Sie bei Zugriff auf *localhost* keine anderen Anmeldeinformationen verwenden können. Sollte der Computer jedoch nicht der lokale Computer sein, können Sie andere Anmeldeinformationen angeben. Sie müssen die entsprechenden Berechtigungen explizit aktivieren. Die *switch*-Anweisung lautet:

```
 "r" {
      if($computer -ne "localhost")
      {
       $objWMI = Get-WmiObject -Class Win32_operatingsystem `
       -computername $computer -credential $user
       $objWMI.psbase.Scope.Options.EnablePrivileges = $true
       $objWMI.reboot()
      }
      ELSE
      {
       $objWMI = Get-WmiObject -Class Win32_operatingsystem `
       -computername $computer
       $objWMI.psbase.Scope.Options.EnablePrivileges = $true
       $objWMI.reboot()
      }
     }
```

Um zu verhindern, dass das Skript den Server unerwartet herunterfährt, legen Sie eine Standardaktion fest, um eine gekürzte Hilfemeldung auszugeben und den Benutzer aufzufordern, das Skript mit dem Parameter **-help** auszuführen. Die Standardaktion ist wie folgt implementiert:

```
 DEFAULT { "Sie müssen einen Wert für die auszuführende Aktion angeben."
           "Für weitere Informationen führen Sie folgenden Befehl aus: ShutdownRebootComputer.ps1 -help ?" }
}
```

Das vollständige Skript *ShutdownRebootComputer.ps1* hat folgenden Inhalt:

ShutdownRebootComputer.ps1

```
param(
      $computer="localhost",
      $user = "administrator",
      $password,
      $a,
      $help
    )

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ShutdownRebootComputer.ps1
Fährt den lokalen Computer oder einen Remotecomputer herunter oder startet diesen neu.

PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-user     Der Benutzername.
-password Das Kennwort des Benutzers.
-a(ktion) Die auszuführende Aktion: < s (herunterfahren), r (neu starten) >.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
ShutdownRebootComputer.ps1-computer MunichServer -a s

Fährt einen Remotecomputer namens MunichServer herunter.

ShutdownRebootComputer.ps1-computer MunichServer -a r
-user munich\admin -password MyPassword

Startet einen Computer namens MunichServer neu. Verwendet dabei
das Benutzerkonto admin aus der Domäne munich mit dem Kennwort MyPassword.

ShutdownRebootComputer.ps1

Zeigt eine Meldung mit Hinweisen zur Hilfe an.

ShutdownRebootComputer.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

```
if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }

switch($a)
{
 "s" {
      if($computer -ne "localhost")
      {
       $objWMI = Get-WmiObject -Class Win32_operatingsystem `
       -computername $computer -credential $user
       $objWMI.psbase.Scope.Options.EnablePrivileges = $true
       $objWMI.shutdown()
      }
      ELSE
      {
       $objWMI = Get-WmiObject -Class Win32_operatingsystem `
       -computername $computer
       $objWMI.psbase.Scope.Options.EnablePrivileges = $true
       $objWMI.shutdown()

      }
     }
 "r" {
      if($computer -ne "localhost")
      {
       $objWMI = Get-WmiObject -Class Win32_operatingsystem `
       -computername $computer -credential $user
       $objWMI.psbase.Scope.Options.EnablePrivileges = $true
       $objWMI.reboot()
      }
      ELSE
      {
       $objWMI = Get-WmiObject -Class Win32_operatingsystem `
       -computername $computer
       $objWMI.psbase.Scope.Options.EnablePrivileges = $true
       $objWMI.reboot()
      }
     }
 DEFAULT { "Sie müssen einen Wert für die auszuführende Aktion angeben."
           "Für weitere Informationen führen Sie folgenden Befehl aus: ShutdownRebootComputer.ps1 -help ?" }
}
```

Zusammenfassung

In diesem Kapitel wurden die Aufgaben beschrieben, die nach der Installation des Betriebssystems ausgeführt werden müssen. Diese Aufgaben umfassen das Festlegen der lokalen Systemzeit, das Konfigurieren der Zeitquelle für den Win32Time-Dienst und das Aktivieren bzw. Deaktivieren von Benutzerkonten. Außerdem wurde das Erstellen von lokalen Benutzern und Gruppen sowie das Konfigurieren der Bildschirmschonereinstellungen auf Remotecomputern erklärt. Das Kapitel wurde mit dem Umbenennen eines Computers und dem Neustarten des lokalen Computers und von Remotecomputern abgeschlossen.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

KAPITEL 11

Verwalten von Benutzerdaten

Nach Abschluss dieses Kapitels können Sie:

- Mit Datensicherungen arbeiten
- Offline-Dateien aktivieren
- Offline-Dateien konfigurieren
- Mit der Systemwiederherstellung arbeiten

Auf der Begleit-CD Alle Skripts in diesem Kapitel befinden sich auf der Begleit-CD im Ordner *\Scripts\Chapter11*.

Arbeiten mit Datensicherungen

In Windows Vista ist kein Sicherungsprogramm integriert. Es stehen also nur folgende Optionen zur Auswahl: Sie verlassen sich darauf, dass die Benutzer mit „Sichern und Wiederherstellen“ Sicherungen auf ihren Computern erstellen, Sie verwenden Systemwiederherstellungspunkte, Sie leiten die Benutzerdaten in eine Netzwerkfreigabe um, ordnen ein Laufwerk über ein Netzwerksicherungsprogramm zu, schreiben ein Skript zum Sichern der Dateien in einem Netzwerkverzeichnis oder Sie erwerben das Sicherungsprogramm eines Drittanbieters.

Windows PowerShell umfasst Cmdlets, mit denen Sie eine Lösung erstellen können, beispielsweise das Skript *BackupFolderToServer.ps1*. Dieses Skript kopiert mit dem Cmdlet **Copy-Item** die Dateien in ein zugeordnetes Verzeichnis auf einem Server oder auf einem Speichergerät. Das Skript kann die Daten auch auf ein tragbares Speichergerät, beispielsweise eine Flash-Speicherkarte oder ein USB-Laufwerk, oder in das Basisverzeichnis eines Benutzers kopieren.

Das Skript *BackupFolderToServer.ps1* beginnt mit der Anweisung *param*, die das Angeben von Befehlszeilenargumenten ermöglicht. Die Argumente steuern die Skriptausführung und stellen sicher, dass die Benutzer das Skript nicht direkt bearbeiten müssen. Ein Skript dieses Typs kann auf einer Batchdatei basieren, die Parameter umfasst, und aus anderen Windows PowerShell-Skripts aufgerufen werden. Das Skript definiert drei Parameter: **-source**, **-destination** und **-help**. Diese Parameter sind jeweils in der entsprechenden Variablen mit dem gleichen Namen gespeichert. Codezeile:

```
param($source, $destination, $help)
```

Implementieren Sie die Funktion *funhelp*, um eine Hilfemeldung anzuzeigen, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Die Funktion *funhelp* beginnt mit der Variablen *$helpText*, in der eine *here*-Zeichenfolge gespeichert ist. Die *here*-Zeichenfolge ermöglicht Ihnen das Festlegen des Textes, der auf dem Bildschirm angezeigt werden soll. Der Hilfetext besteht aus drei Abschnitten: Der Beschreibung des Skripts, den vom Skript akzeptierten Parametern und der Syntax. Nachdem Sie den Hilfetext erstellt haben, zeigen Sie den Inhalt der Variablen *$helpText* an und beenden das Skript. Die Funktion *funhelp*:

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: BackupFolderToServer.ps1
Sichert Dateien in einem Ordner unter Verwendung einer Netzwerkfreigabe. Der Zielordner
muss nicht existieren.

PARAMETER:
-source      Der Quellordner für die Dateien und Unterordner.
-destination Der Zielordner für die Dateien und Unterordner.
-help        Zeigt das Hilfethema für dieses Skript an.

SYNTAX:
BackupFolderToServer.ps1 -source c:\fso -destination h:\fso

Sichert alle Dateien und Unterordner aus dem Quellordner C:\FSO auf dem lokalen Computer
in einer Netzwerkfreigabe,  der der Laufwerksbuchstabe H. zugeordnet wurde. Der Ordner \FSO muss
auf dem Laufwerk H:\ nicht existieren.

BackupFolderToServer.ps1

Generiert eine Fehlermeldung, denn die Parameter -source und -destination
müssen angegeben werden.

BackupFolderToServer.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Stellen Sie sicher, dass die Variable *$help* vorhanden ist. Zeigen Sie anschließend eine Statusmeldung an und rufen Sie die Funktion *funhelp* auf. Um zwei Befehle in der gleichen Zeile einzugeben, trennen Sie diese durch ein Semikolon. Beispiel:

```
if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
```

Überprüfen Sie, ob die beiden erforderlichen Parameter vorhanden sind. Diese Parameter sind **-source** und **-destination**. Das *source*-Verzeichnis ist ein lokaler Pfad, der auf dem Computer vorhanden sein muss. Stellen Sie sicher, dass Sie über die Berechtigungen für den Ordner verfügen. Der Zielordner muss nicht bereits auf dem zugeordneten Laufwerk vorhanden sein, da er beim Ausführen des Skripts erstellt wird. Sollten diese beiden Variablen jedoch nicht vorhanden sein, zeigen Sie mit einer *throw*-Anweisung eine Fehlermeldung an und beenden das Skript. Weisen Sie den Benutzer auf die folgende Hilfetextsyntax hin:

```
if(!$source -or !$destination)
  {
    $(throw "Sie müssen sowohl den Parameter -source als auch den Parameter -destination angeben.
    Für weitere Informationen führen Sie folgenden Befehl aus: BackupFolderToServer.ps1 -help -?")
  }
```

Sie müssen nun die Dateien kopieren. Das Cmdlet **Copy-Item** akzeptiert den Parameter **-path**, dem die Variable *$source* zugewiesen ist. Übergeben Sie die Variable *$destination* an den Parameter **-destination** und kopieren Sie alle Unterordner mit dem Parameter **-recurse**. Codezeile:

```
Copy-Item -Path $source -destination $destination -recurse
```

Das vollständige Skript *BackupFolderToServer.ps1* hat folgenden Inhalt:

BackupFolderToServer.ps1

```
param($source, $destination, $help)

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: BackupFolderToServer.ps1
Sichert Dateien in einem Ordner unter Verwendung einer Netzwerkfreigabe. Der Zielordner
muss nicht existieren.

PARAMETER:
-source      Der Quellordner für die Dateien und Unterordner.
-destination Der Zielordner für die Dateien und Unterordner.
-help        Zeigt das Hilfethema für dieses Skript an.

SYNTAX:
BackupFolderToServer.ps1 -source c:\fso -destination h:\fso

Sichert alle Dateien und Unterordner aus dem Quellordner C:\FSO auf dem lokalen Computer
in einer Netzwerkfreigabe,  der der Laufwerksbuchstabe H. zugeordnet wurde. Der Ordner \FSO muss
auf dem Laufwerk H:\ nicht existieren.

BackupFolderToServer.ps1

Generiert eine Fehlermeldung, denn die Parameter -source und -destination
müssen angegeben werden.

BackupFolderToServer.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help){ "Obtaining help ..." ; funhelp }
if(!$source -or !$destination)
  {
    $(throw "Sie müssen sowohl den Parameter -source als auch den Parameter -destination angeben.
    Für weitere Informationen führen Sie folgenden Befehl aus: BackupFolderToServer.ps1 -help -?")
  }
Copy-Item -Path $source -destination $destination -recurse
```

Konfigurieren von Offline-Dateien

Offline-Dateien ermöglichen die Synchronisierung der auf einem Server oder auf einem tragbaren Computer gespeicherten Dateien. Offline-Dateien stellen die beste Stabilität für den Benutzer sicher und lösen das Problem, das das Sichern wichtiger Dateien darstellt. Es ist nur eine Instanz des Offline-Dateicaches verfügbar, auf die Sie über das Applet **Offline-Dateien** in der Systemsteuerung zugreifen können (siehe Abbildung 11.1). Der WMI-Anbieter unterstützt mehrere WMI-Klassen (*OfflineFilesWmiProvider*).

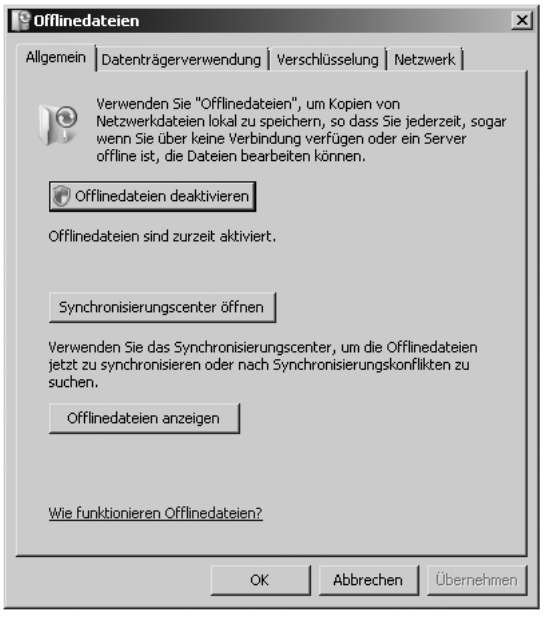


Abbildung 11.1 Der Offline-Cache auf der lokalen Festplatte

Verwenden Sie im ersten Skript für den Offline-Dateicache die WMI-Klasse *Win32_OfflineFilesCache*, um Informationen abzurufen. Sie müssen überprüfen, ob Offline-Dateien aktiviert sind. Wenn Offline-Dateien aktiviert sind, überprüfen Sie, wo die Dateien gespeichert sind.

Sie müssen im Skript *GetOffLineFiles.ps1* zwei Parameter angeben. Der erste Parameter lautet **-computer**. Legen Sie den Wert der Variablen *$computer* auf *localhost* fest, damit das Skript auf dem lokalen Computer ausgeführt wird. Der zweite Parameter ist **-help**. Legen Sie die Variable *$help* nicht auf einen Standardwert fest, da Sie diese nicht ständig verwenden möchten. Die *param*-Anweisung deklariert beide Befehlszeilenparameter. Codezeile:

```
param($computer="localhost", $help)
```

Anschließend müssen Sie zwei Funktionen definieren. Die erste Funktion ist *funline*, um eine Kopfzeile in der Skriptausgabe zu unterstreichen. Die Funktion *funline* akzeptiert eine Variable namens *$strIN*. Die Variable *$strIN* enthält den Zeichenfolgenwert, der an die Funktion als Ausgabetext übergeben wurde, und bestimmt mittels der Eigenschaft *Length* die Länge dieser Zeichenfolge. Speichern Sie diesen Wert in der Variablen *$num*. Verwenden Sie eine *for*-Anweisung, um die Variable *$funline* in einer Schleife zu erstellen. Die Variable enthält eine Reihe von Gleichheitszeichen (=), die der Länge der Zeichenfolge in der Variablen *$strIN* entsprechen. Nachdem Sie einen passenden Unterstrich

erstellt haben, geben Sie die Zeichenfolge in der Variablen *$strIN* und die Zeilentrennung in der Variablen *$funline* mit zwei **Write-Host**-Aufrufen aus. Wählen Sie Farben für die beiden Zeilen mit dem **-foregroundcolor**-Parameter aus. Die Funktion *funline* ist wie folgt implementiert:

```
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline += "=" }
    Write-Host -ForegroundColor yellow $strIN
    Write-Host -ForegroundColor darkYellow $funline
}
```

Nachdem Sie die Funktion *funline* definiert haben, erstellen Sie eine Funktion, um die Onlinehilfe anzuzeigen, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Beginnen Sie mit der Funktion *funhelp*, indem Sie die Variable *$helpText* deklarieren und dieser eine *here*-Zeichenfolge zuweisen. Legen Sie in der *here*-Zeichenfolge jeweils einen Textabschnitt für die Beschreibung, die Parameter und die Syntax fest. Zeigen Sie anschließend den Inhalt der Variablen *$helpText* an und beenden Sie das Skript. Die Funktion *funhelp* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GetOffLineFiles.ps1
Ermittelt die Konfiguration der Offline-Dateien auf der lokalen Arbeitsstation oder auf einem Remotecomputer.

PARAMETER:
-computer Gibt den Namen des Computers an, der mit diesem Skript überprüft werden soll.
-help     Zeigt das Hilfethema für dieses Skript an.

SYNTAX:
GetOffLineFiles.ps1 -computer MunichServer

Ermittelt die Konfiguration der Offline-Dateien auf einem Computer namens MunichServer.

GetOffLineFiles.ps1

Ermittelt die Konfiguration der Offline-Dateien auf dem lokalen Computer.

GetOffLineFiles.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Um zu bestimmen, ob der Inhalt der Hilfedatei angezeigt werden soll, überprüfen Sie mit einer *if*-Anweisung, ob die Variable *$help* vorhanden ist. Gibt es die Variable *$help*, wurde das Skript mit dem Parameter **-help** ausgeführt. Rufen Sie zuerst die Funktion *funline* auf, um die Statusmeldung auszugeben und zu unterstreichen, und führen Sie anschließend die Funktion *funhelp* aus, um die Hilfeinformationen auszugeben. Codezeile:

```
if($help){ funline("Ausgeben der Hilfeinformationen ...") ; funhelp }
```

Rufen Sie mit der WMI-Klasse *Win32_OfflineFilesCache* die für das Skript erforderlichen Informationen ab. Verwenden Sie hierzu das Cmdlet **Get-WmiObject**. Fragen Sie mit dem Parameter **-class** die Klasse *Win32_OfflineFilesCache* ab und stellen Sie mit dem Parameter **-computername** eine Verbindung zum gewünschten Computer her. Speichern Sie das resultierende WMI-Objekt gegebenenfalls in der Variablen *$outtxt*. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
$outtxt = Get-WmiObject -Class win32_OfflineFilesCache `
        -computername $computer
```

Nachdem Sie die Konfigurationsinformationen für die Offline-Dateien abgerufen haben, geben Sie mit der Funktion *funline* eine Statusmeldung für den Ordner *$offline* aus. Verwenden Sie die Umgebungsvariable *%COMPUTERNAME%*, um den Namen des aktuellen Computers aus *PS drive env:*\ abzurufen. Codezeile:

```
funline("Offlinedateikonfiguration $env:computername")
```

Formatieren Sie die Ausgabe mit dem Cmdlet **Format-Table**. Wählen Sie die Eigenschaften in der Reihenfolge aus, in der Sie die Daten anzeigen möchten. Verwenden Sie den Parameter **-inputobject** und geben Sie das *Management*-Objekt an, das in der Variablen *$outtxt* gespeichert ist. Geben Sie den Parameter **-autosize** für eine kompakte Ausgabe an. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
format-table -Property active, enabled,location -autosize `
       -inputobject $outtxt
```

Das vollständige Skript *GetOffLineFiles.ps1* hat folgenden Aufbau:

GetOffLineFiles.ps1

```
param($computer="localhost", $help)

function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline += "=" }
    Write-Host -ForegroundColor yellow $strIN
    Write-Host -ForegroundColor darkYellow $funline
}

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GetOffLineFiles.ps1
Ermittelt die Konfiguration der Offline-Dateien auf der lokalen Arbeitsstation oder auf einem Remotecomputer.

PARAMETER:
-computer Gibt den Namen des Computers an, der mit diesem Skript überprüft werden soll.
-help     Zeigt das Hilfethema für dieses Skript an.

SYNTAX:
GetOffLineFiles.ps1 -computer MunichServer

Ermittelt die Konfiguration der Offline-Dateien auf einem Computer namens MunichServer.

GetOffLineFiles.ps1

Ermittelt die Konfiguration der Offline-Dateien auf dem lokalen Computer.
```

```
GetOffLineFiles.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help){ funline("Ausgeben der Hilfeinformationen ...") ; funhelp }

$outtxt = Get-WmiObject -Class win32_OfflineFilesCache `
        -computername $computer
funline("Konfiguration der Offline-Dateien für $env:computername")

format-table -Property active, enabled,location -autosize `
       -inputobject $outtxt
```

Aktivieren von Offline-Dateien

Sie können einen Windows Vista- oder Windows Server 2008-Computer mit dem Verwaltungsprogramm für Offline-Dateien für die Verwendung von Offline-Dateien konfigurieren (siehe Abbildung 11.2). Dieses Tool ermöglicht das Aktivieren bzw. Deaktivieren von Offline-Dateien auf dem lokalen Computer. Um Offline-Dateien zu aktivieren, müssen Sie über Administratorberechtigungen verfügen und den Computer neu starten.

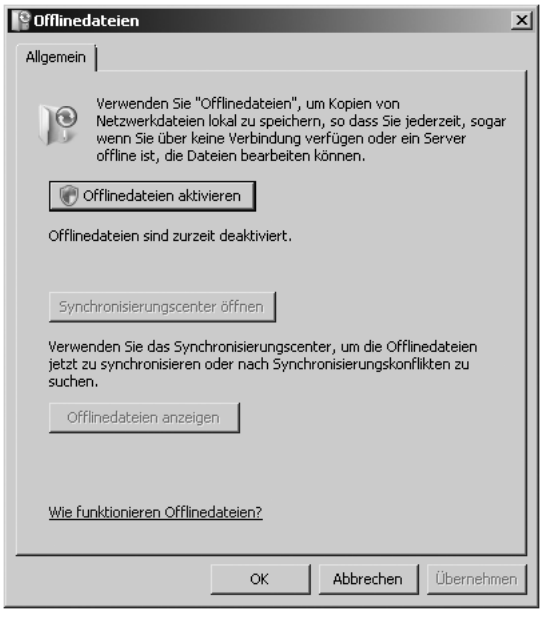


Abbildung 11.2 Aktivieren von Offline-Dateien

Um Offline-Dateien auf mehreren Computern oder als Bestandteil des Standardbereitstellungsprozesses zu aktivieren oder zu deaktivieren, können Sie das Skript *EnableDisableOfflineFiles.ps1* ausführen. Das Skript verwendet die WMI-Klasse *Win32_OfflineFilesCache*.

Das Skript *EnableDisableOfflineFiles.ps1* beginnt mit einer *param*-Anweisung, die die Angabe von benannten Argumenten ermöglicht. Mittels dieser Argumente können Sie die Skriptausführung steuern, ohne das Skript bearbeiten zu müssen. Der erste Parameter ist **-computer**. Die Variable *$computer* wird automatisch erstellt, um die bei der Skriptausführung übergebenen Daten zu speichern. In diesem Beispiel ist die Variable auf den Standardwert *localhost* festgelegt. Sie können das Skript deshalb ohne Angabe eines Werts auf dem lokalen Computer ausführen. Definieren Sie anschließend zwei weitere Parameter: **-a** und **-help**. Weisen Sie diesen Variablen keine Standardwerte zu. Der Parameter **-a** gibt die auszuführende Aktion an. Der Parameter **-help** bestimmt, ob der Hilfetext angezeigt werden soll. Codezeile:

```
param($computer="localhost", $a, $help)
```

Der nächste Schritt betrifft das Implementieren der Funktion *funline*, um die Skriptausgabe mit einem Unterstrich übersichtlicher zu gestalten. Deklarieren Sie in der Funktion den Übergabeparameter *$strIN*. Die auszugebende Zeichenfolge wird mit diesem Parameter an die Funktion *funline* übergeben. Bestimmen Sie mit der Eigenschaft *Length* die Länge der Zeile, die an die Funktion übergeben wurde. Speichern Sie die Länge der Zeichenfolge in der Variablen *$num* und verwenden Sie diese in einer *for*-Schleife. Beginnen Sie mit 1 und setzen Sie die Schleife fort, bis der Wert der Indikatorvariablen *$i* kleiner oder gleich der Variablen *$num* ist. Erhöhen Sie den Wert von *$i* jeweils um 1 ($i++). Der von der *for*-Schleife ausgeführte Code erstellt die Variable *$funline* mit einer Reihe von Gleichheitszeichen (=). Geben Sie die Zeichenfolge mit dem Cmdlet **Write-Host** aus. Legen Sie dabei fest, dass der Parameter **-foregroundcolor** die Ausgabe in Gelb und die Zeilentrennung in der Funktion *$funline* in Dunkelgelb anzeigt, um die Zeilentrennung hervorzuheben. Diese Funktion ist wie folgt implementiert:

```
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline += "=" }
    Write-Host -ForegroundColor yellow $strIN
    Write-Host -ForegroundColor darkYellow $funline
}
```

Der nächste Schritt betrifft die Funktion *funhelp*. Diese Funktion zeigt eine Hilfemeldung an, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Für diese Funktion sind keine Eingabeparameter definiert. Deklarieren Sie die Variable *$helpText* und beginnen Sie eine *here*-Zeichenfolge mit den Zeichen *@"*. Die *here*-Zeichenfolge wird mit den Zeichen *"@* beendet. Der Vorteil der *here*-Zeichenfolge ist, dass Sie die Regeln für Anführungszeichen ignorieren und den Text so eingeben können, wie dieser angezeigt werden soll. Definieren Sie in der *here*-Zeichenfolge jeweils einen Abschnitt für die Beschreibung, Parameter und Syntax. Die Funktion *funhelp* wird beendet, indem der Inhalt der Variablen *$helpText* ausgegeben und die *exit*-Anweisung aufgerufen wird. Die Funktion *funhelp* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: EnableDisableOffLineFiles.ps1
Aktiviert oder deaktiviert Offline-Dateien auf der lokalen Arbeitsstation oder auf einem Remotecomputer.
Ein Neustart des Computers KÖNNTE erforderlich sein. Die entsprechenden Informationen werden in der
```

```
Statusbenachrichtigung nach Abschluss des Skripts angezeigt.

PARAMETERS:
-computer Gibt den Namen des Computers an, der mit diesem Skript überprüft werden soll.
-a(ktion) < e (aktivieren), d (deaktivieren) >
-help     Zeigt das Hilfethema für dieses Skript an.

SYNTAX:
EnableDisableOffLineFiles.ps1 -computer MunichServer -a e

Aktiviert die Offline-Dateien auf einem Computer namens MunichServer.

EnableDisableOffLineFiles.ps1 -a d

Deaktiviert die Offline-Dateien auf einem Computer namens MunichServer.

EnableDisableOffLineFiles.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Deklarieren Sie nach Abschluss der Funktion *funhelp* die Funktion *funtranslatemethod*. Diese Funktion wandelt den für den Parameter **-a** angegebenen Eingabeparameter um. Der Wert in der Variablen *$a* wird beim Ausführen an das Skript übergeben. Das Skript generiert einen Fehler, wenn der Parameter **-a** nicht vorhanden ist, da Sie Offline-Dateien für die Skriptausführung aktivieren oder deaktivieren müssen.

Variablenbereiche

Ein Vorteil von globalen Variablen in einem Skript ist, dass Sie diese innerhalb und außerhalb einer Funktion verwenden können. Variablen, die in einer Funktion erstellt werden, sind ausschließlich in dieser Funktion verfügbar. Möglicherweise werden zwei Variablen mit dem Namen *$a* und unterschiedlichen Werten erstellt, abhängig davon, welcher Skriptabschnitt ausgeführt wird. Dies ist im Skript *CreateVariableInFunctionAndOutsideFunction.ps1* veranschaulicht.

Deklarieren Sie in diesem Skript *$a* als Variable außerhalb der Funktion und weisen Sie einen Zeichenfolgenwert zu. Rufen Sie anschließend eine Funktion auf. Die Funktion kann nun auf die Variable *$a* zugreifen und dieser einen anderen Zeichenfolgenwert zuweisen. Aufgrund der Bereichsdefinition werden jedoch zwei Variablen mit dem Namen *$a* erstellt. Jede Variable hat einen eindeutigen Wert. Wenn Sie die Funktion beenden, wurde die Variable *$a* außerhalb der Funktion nicht geändert. Sie können in der Funktion nicht auf den Wert von *$a* außerhalb der Funktion zugreifen, da die beiden Variablen in einem jeweils anderen Bereich definiert wurden. Das Skript *CreateVariableInFunctionAndOutsideFunction.ps1* gibt folgende Ausgabe zurück:

```
Innerhalb der Funktion MeinTest:
Dies ist eine Variable in der Funktion MeinTest.

Außerhalb der Funktion MeinTest:
Dies ist eine Variable, die außerhalb der Funktion erstellt wurde.
```

Das vollständige Skript *CreateVariableInFunctionAndOutsideFunction.ps1* verwendet folgende Anweisungen:

CreateVariableInFunctionAndOutsideFunction.ps1

```
function MeinTest
{
 $a = "Dies ist eine Variable in der Funktion MeinTest.`n"
 Write-Host "Innerhalb der Funktion MeinTest: `n$a"
}

$a = "Dies ist eine Variable, die außerhalb der Funktion erstellt wurde.`n"
MeinTest
Write-Host "Außerhalb der Funktion MeinTest: `n$a"
```

Wie wirkt sich die Bereichsdefinition jedoch auf die Arbeit mit Variablen und Funktionen aus, wenn zwei Variablen mit dem gleichen Namen in verschiedenen Bereichen definiert sind? Im Skript *CreateVariableInFunction.ps1* wird die umgekehrte Situation verdeutlicht. Erstellen Sie die Variable *$a* innerhalb der Funktion. Da Sie außerhalb der Funktion keine Variable namens *$a* erstellt haben, ist die Variable außerhalb der Funktion nicht verfügbar.

Das Skript *CreateVariableInFunction.ps1* führt die Funktion *MeinTest* aus und weist der Variablen *$a* einen Wert zu. Der Wert von *$a* wird ausgegeben. Anschließend wird die Funktion beendet und der Wert von *$a* wird nicht angezeigt, da die in der Funktion erstellte Variable nicht außerhalb der Funktion verfügbar ist.

Die Ausgabe des Skripts *CreateVariableInFunction.ps1*:

```
Innerhalb der Funktion MeinTest:
Dies ist eine Variable in der Funktion MeinTest.

Außerhalb der Funktion MeinTest:
```

Das vollständige Skript *CreateVariableInFunction.ps1*:

CreateVariableInFunction.ps1

```
function MeinTest
{
 $a = "Dies ist eine Variable in der Funktion MeinTest. `n"
 Write-Host "Innerhalb der Funktion MeinTest: `n$a"
}

MeinTest
 Write-Host "Außerhalb der Funktion MeinTest: `n$a"
```

Für den Zugriff auf eine Variable innerhalb und außerhalb einer Funktion müssen Sie eine globale Variable verwenden. Geben Sie zum Deklarieren einer globalen Variablen das Tag *$global* vor dem Variablennamen an. Syntax:

```
$global:myvariable = "Dies ist eine globale Zeichenfolge."
```

Erstellen Sie im Skript *CreateGlobalVariableInFunction.ps1* in der Funktion *MeinTest* eine globale Variable, um innerhalb und außerhalb der Funktion *MeinTest* auf den gleichen Wert der Variablen zugreifen zu können. Sie können die globale Variable auch außerhalb der Funktion erstellen und anschließend innerhalb der Funktion verwenden. Das Skript *CreateGlobalVariableInFunction.ps1* zeigt folgenden Text an:

```
Innerhalb der Funktion MeinTest:
Dies ist eine Variable in der Funktion MeinTest.

Außerhalb der Funktion MeinTest:
Dies ist eine Variable in der Funktion MeinTest.
```

Das vollständige Skript *CreateGlobalVariableInFunction.ps1* verwendet folgende Anweisungen:

CreateGlobalVariableInFunction.ps1

```
function MeinTest
{
 $global:a = "Dies ist eine Variable in der Funktion MeinTest.`n"
 Write-Host "Innerhalb der Funktion MeinTest: `n$a"
}

MeinTest
Write-Host "Außerhalb der Funktion MeinTest: `n$a"
```

Die Funktion *funtranslatemethod* im Skript *EnableDisableOfflineFiles.ps1* verwendet die *switch*-Anweisung, um den Wert des Parameters **-a** auszuwerten. Wenn der Wert in der Variablen *$a* der Buchstabe *e* ist, führen Sie im Codeblock zwei Verarbeitungsschritte aus: Weisen Sie die systeminterne Variable *$true* der globalen Variablen *$m* zu. Speichern Sie anschließend in der globalen Variablen *$msg* die Zeichenfolge, die auf dem Bildschirm angezeigt werden soll. Setzen Sie die Zeichenfolge auf "Offline-Dateien aktivieren", um die Aktion anzuzeigen, die ausgeführt wird.

Die andere in der Funktion *funtranslatemethod* definierte Aktion wird zum Deaktivieren verwendet. Wenn der Benutzer während der Skriptausführung den Buchstaben *d* eingibt, enthält die globale Variable *$m* die systeminterne Variable *$false*. Speichern Sie die Zeichenfolge "Offline-Dateien deaktivieren" entsprechend in der globalen Variablen *$msg*.

Die Standardaktion der *switch*-Anweisung weist der globalen Variablen *$msg* die Zeichenfolge "*... it keine zulässige Antwort.*" zu. Geben Sie den in der Variablen *$a* angegebenen Wert der Aktion aus und geben Sie das Sonderzeichen **`n** an, um die Zeichenfolge mit einem Zeilenwechsel am Ende auszugeben. Die Funktion *funtranslatemethod* ist wie folgt defniert:

```
function funtranslatemethod($a)
{
 switch($a)
  {
   "e" { $global:m = $true
         $global:msg = "Offline-Dateien aktivieren."
       }
   "d" {
        $global:m = $false
        $global:msg = "Offline-Dateien deaktivieren."
       }
  default{
          $global:msg = "$a ist keine zulässige Antwort.`n"
          }
  }
}
```

Überprüfen Sie, ob zwei Variablen vorhanden sind. Suchen Sie als Erstes nach der Variablen *$help*. Wenn die Variable vorhanden ist, wurde das Skript mit dem Parameter **-help** ausgeführt und Sie müssen den Hilfetext anzeigen. Verwenden Sie hierzu eine *if*-Anweisung und suchen Sie nach der Variablen. Wenn die Variable vorhanden ist, rufen Sie im Codeblock die Funktion *funline* auf, geben Sie eine Meldung aus und rufen Sie anschließend die Funktion *funhelp* auf. Codezeile:

```
if($help){ funline("Ausgeben der Hilfeinformationen ...") ; funhelp }
```

Stellen Sie sicher, dass die Variable *$a* vorhanden ist. Wenn die Variable nicht vorhanden ist, wurde das Skript ohne Parameter *$a* ausgeführt. Da dieser Parameter erforderlich sein soll, geben Sie eine Meldung mit der *throw*-Anweisung aus.

Hinweis Die *throw*-Anweisung unterbricht die Skriptausführung und gibt die Meldung in Rot aus. Diese Methode ist der *raise*-Methode des *error*-Objekts in anderen Programmiersprachen ähnlich.

Die ausgegebene Zeichenfolge gibt den aufgetretenen Fehler an (dem Parameter **-a** wurde kein Wert zugewiesen). Verweisen Sie den Benutzer anschließend auf die Hilfedatei. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
if(!$a)
   {
    $(throw "Sie müssen eine Aktion angeben. Für weitere Informationen geben Sie folgenden Befehl ein:
   EnableDIsableOfflineFiles.ps1 -help ?")
   }
```

Deklarieren Sie für den Fall, dass der Benutzer die Hilfe nicht anzeigen möchte und einen Wert für den Parameter **-a** angibt, mehrere globale Variablen, legen Sie diese auf Null fest und rufen Sie die Funktion *funtranslatemethod* auf, um festzustellen, welche Aktion Sie ausführen müssen. Die beiden Codezeilen lauten:

```
$global:msg = $global:m = $null
funtranslatemethod($a)
```

Sie müssen nun eine Verbindung zu WMI herstellen. Verwenden Sie hierzu den *[wmiclass]*-Accelerator, um auf die Microsoft .NET Framework-Klasse *System.Management.ManagementObject* zuzugreifen. Diese .NET Framework-Klasse ermöglicht den Zugriff auf die WMI-Methoden, die mit dem Cmdlet **Get-WmiObject** möglicherweise nicht verfügbar sind. Sie können das Cmdlet **Get-Member** und das Windows Software Development Kit (SDK) verwenden, um weitere Informationen zum Thema Methodenaufrufe zu erhalten. Die Syntax zum Herstellen einer Verbindung mit einem Remotecomputer unter Verwendung dieser Klasse ist etwas komplizierter. Sie müssen die Variable *$computer* einbeziehen, die über die Befehlszeile in der Verbindungszeichenfolge angegeben wird, um den Zugriff auf andere Computer zu vereinfachen. Weisen Sie das erstellte *System.Management.ManagementObject*-Objekt der Variablen *$objWMI* zu. Codezeile:

```
$objWMI = [wmiclass]"\\$computer\root\cimv2:win32_offlinefilescache"
```

Rufen Sie die *Enable*-Methode auf und aktivieren oder deaktivieren Sie die Offline-Dateien in Windows Vista oder Windows Server 2008. Geben Sie mit der Funktion *funline* die Statusmeldung aus und rufen Sie die *Enable()*-Methode auf. Code:

```
funline("Konfigurieren der Offline-Dateien auf $computer ...")
$rtn = $objwmi.enable($m)
```

Überprüfen Sie nun den von der *Enable*-Methode zurückgegebenen Wert. Wenn die Eigenschaft *ReturnValue* des Rückgabecodes *0* ist, war der Aufruf erfolgreich. Geben Sie ansonsten den Rückgabecode aus und geben Sie an, dass der Aufruf fehlgeschlagen ist. Ein Problem mit der WMI-Klasse ist,

dass diese nicht immer einen Wert ungleich Null zurückgibt. Die Klasse gibt jedoch immer *0* zurück, wenn der Aufruf erfolgreich ist. Dieser Code ist im Folgenden dargestellt:

```
if($rtn.returnvalue -eq 0)
 {
  Write-Host -ForegroundColor green "$msg Erfolgreich abgeschlossen."
 }
ELSE
 {
  Write-Host -ForegroundColor red "$msg Fehlgeschlagen mit $($rtn.returnvalue)."
 }
```

Überprüfen Sie die Eigenschaft *RebootRequired*. Wenn die *Enable*-Methode erfolgreich abgeschlossen wurde, aber festgestellt wird, dass ein Neustart erforderlich ist, legt diese Methode die Eigenschaft *RebootRequired* auf *True* fest. Geben Sie gegebenenfalls an, dass ein Neustart erforderlich ist. Code:

```
if($rtn.rebootrequired)
  { Write-Host -ForegroundColor cyan "Ein Neustart ist erforderlich." }
```

Das vollständige Skript *EnableDisableOfflineFiles.ps1*:

EnableDisableOfflineFiles.ps1

```
param($computer="localhost", $a, $help)

function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline += "=" }
    Write-Host -ForegroundColor yellow $strIN
    Write-Host -ForegroundColor darkYellow $funline
}

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: EnableDisableOffLineFiles.ps1
Aktiviert oder deaktiviert Offline-Dateien auf der lokalen Arbeitsstation oder auf einem Remotecomputer.
Ein Neustart des Computers KÖNNTE erforderlich sein. Die entsprechenden Informationen werden in der
Statusbenachrichtigung nach Abschluss des Skripts angezeigt.

PARAMETER:
-computer Gibt den Namen des Computers an, der mit diesem Skript überprüft werden soll.
-a(ktion) < e (aktivieren), d (deaktivieren) >
-help     Zeigt das Hilfethema für dieses Skript an.

SYNTAX:
EnableDisableOffLineFiles.ps1 -computer MunichServer -a e

Aktiviert die Offline-Dateien auf einem Computer namens MunichServer.

EnableDisableOffLineFiles.ps1 -a d

Deaktiviert die Offline-Dateien auf dem lokalen Computer.

EnableDisableOffLineFiles.ps1 -help ?
```

```
Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

function funtranslatemethod($a)
{
 switch($a)
  {
   "e" { $glogal:m = $true
         $global:msg = "Offline-Dateien aktivieren."
       }
   "d" {
        $global:m = $false
        $global:msg = "Offline-Dateien deaktivieren."
       }
  default{
          $global:msg = "$a a ist keine zulässige Antwort.`n"
          }
  }
}

if($help){ funline("Ausgeben der Hilfeinformationen ...") ; funhelp }
if(!$a)
   {
    $(throw "Sie müssen eine Aktion angeben. Für weitere Informationen geben Sie folgenden Befehl ein:
   EnableDIsableOfflineFiles.ps1 -help ?")
   }
$global:msg =$global:m = $null
funtranslatemethod($a)

$objWMI = [wmiclass]"\\$computer\root\cimv2:win32_offlinefilescache"
funline("Konfigurieren der Offline-Dateien auf $computer ...")
$rtn = $objwmi.enable($m)
if($rtn.returnvalue -eq 0)

 {
  Write-Host -ForegroundColor green "$msg Erfolgreich abgeschlossen."
 }
ELSE
 {
  Write-Host -ForegroundColor red "$msg Fehlgeschlagen mit $($rtn.returnvalue)."
 }
if($rtn.rebootrequired)
  { Write-Host -ForegroundColor cyan "Neustart erforderlich." }
```

Arbeiten mit der Systemwiederherstellung

Die Systemwiederherstellung kann mit zwei WMI-Klassen verwaltet werden. Diese Klassen sind *SystemRestore* und *SystemRestoreConfig*. In diesem Abschnitt wird die Verwendung beider Klassen auf lokalen Computern und Remotecomputern erklärt.

Abrufen der Einstellungen der Systemwiederherstellung

Das Skript *GetSystemRestoreSettings.ps1* demonstriert das Abrufen von Systemwiederherstellungseinstellungen. Geben Sie im Skript eine *param*-Anweisung an, um die Verwendung von Befehlszeilenargumenten zuzulassen. Definieren Sie die beiden Parameter **-computer** und **-help**, um einen Remotecomputer als Ziel angeben zu können und gegebenenfalls die Hilfe abzurufen. Der Parameter **-computer** ist auf den Standardwert *localhost* festgelegt. Codezeile:

```
Param($computer = "localhost", $help)
```

Implementieren Sie die Funktion *funhelp*, um eine Hilfemeldung anzuzeigen, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Erstellen Sie die Hilfeinformationen mit der Variablen *$helpText* und einer *here*-Zeichenfolge. Die *here*-Zeichenfolge gibt Ihnen die Möglichkeit, die üblichen Regeln für Anführungszeichen außer Kraft zu setzen. Definieren Sie in der *here*-Zeichenfolge jeweils einen Abschnitt für die Beschreibung, Parameter und Syntax. Nachdem Sie die *here*-Zeichenfolge erstellt haben, geben Sie den Wert der Variablen *$helpText* aus und beenden Sie das Skript. Beispiel:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GetOffLineFiles.ps1
Ermittelt die Systemwiederherstellungskonfiguration auf der lokalen Arbeitsstation oder auf einem Remotecomputer.

PARAMETER:
-computer Gibt den Namen des Computers an, der mit diesem Skript überprüft werden soll.
-help     Zeigt das Hilfethema für dieses Skript an.

SYNTAX:
GetSystemRestoreSettings.ps1 -computer MunichServer

Ermittelt die Systemwiederherstellungskonfiguration auf einem Computer namens MunichServer.

GetSystemRestoreSettings.ps1

Ermittelt die Systemwiederherstellungskonfiguration auf dem lokalen Computer.

ReportTerminalServiceSetting.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Um zu bestimmen, ob die Hilfe angezeigt werden muss, suchen Sie nach der Variablen *$help*. Wenn die Variable vorhanden ist, rufen Sie die Funktion *funline* auf und geben eine unterstrichene Statusmeldung aus. Rufen Sie anschließend die Funktion *funhelp* auf. Dieser Code ist im Folgenden dargestellt:

```
if($help){ funline("Ausgeben der Hilfeinformationen ...") ; funhelp }
```

Erstellen Sie die Konstante*SecInDay* mit dem Wert 86400. Verwenden Sie hierzu das Cmdlet **New-Variable** und geben Sie den Parameter **-option** mit dem Argument **constant** an. Beispiel:

```
New-Variable -Name SecInDay -option constant -value 86400
```

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Sie müssen nun eine Verbindung mit WMI herstellen. Stellen Sie mit dem Cmdlet **Get-WmiObject** die Verbindung mit dem WMI-Namespace *root\default* her. Geben Sie mit dem Parameter **-class** die WMI-Klasse *SystemRestoreConfig* und mit dem Parameter **-computername** den Computer an. Weisen Sie das *Management*-Objekt der Variablen *$objWMI* zu. Code:

```
$objWMI = Get-WmiObject -Namespace root\default `
         -Class SystemRestoreConfig -computername $computer
```

Geben Sie in der *for*-Anweisung die Werte 0 bis 15 an und imkrementieren Sie die Variable *$i*. Verwenden Sie im Codeblock der *for*-Anweisung das Cmdlet **Write-Host** und übergeben Sie die Variable *$i* an den Parameter **-foregroundcolor**. Geben Sie eine Statusmeldung aus, warten Sie 60 Millisekunden, löschen Sie den Bildschirm und wiederholen Sie den Vorgang. Das Ergebnis ist eine mehrfarbige Statusanzeige, die die Aufmerksamkeit des Benutzers erregt. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
for($i=0; $i -le 15; $i++)
{
 Write-Host -ForegroundColor $i "Ermitteln der Systemwiederherstellungskonfiguration"
 Start-Sleep -Milliseconds 60
 cls
}
```

Sie müssen entweder die Umgebungsvariable *%COMPUTERNAME%* aus Windows PowerShell *PSDrive* oder den Wert im Parameter **-computer** verwenden. Windows PowerShell *PSDrive* ist nur auf dem lokalen Computer verfügbar (*localhost*).

```
if($computer -eq "localhost")
 {
  Write-Host "Systemwiederherstellungseinstellungen auf $env:computername"
 }
```

Wenn der Computername einen anderen Wert hat, verwenden Sie diesen Wert. Beispiel:

```
ELSE
 {
  Write-Host "Systemwiederherstellungseinstellungen auf $computer"
 }
```

Sie müssen die Ausgabe mit dem Cmdlet **Format-Table** formatieren. Verwenden Sie den Parameter **-inputobject** und geben Sie das *Management*-Objekt in der Variablen *$objWMI* für das Cmdlet an. Geben Sie mit dem Parameter **-property** die Eigenschaften an, die in der Tabelle aufgelistet werden sollen. Das Skript ist insofern ungewöhnlich, dass die Formatierung der Ausgabe der Eigenschaftenwerte mit einer Hashtabelle geändert wird. Die Hashtabelle beginnt mit dem Symbol *at* (@) und einem Codeblock. Geben Sie die beiden Bezeichnungen und den Ausdruck zum Berechnen des Eigenschaftswerts an. Die Ausgabe zeigt die Sicherungszeit in Tagen anstatt in Sekunden und die Datenträgerbelegung in Prozent an. Dieser Codeabschnitt ist wie folgt implementiert:

```
format-table -InputObject $objWMI -property `
  @{
    Label="Maximale Datenträgerauslastung" ;
   expression={  "{0:n0}"-f ($_.DiskPercent ) + " %"}
   },
  @{
    Label="Geplante Datensicherung" ;
   expression={  "{0:n2}"-f ($_.RPGlobalInterval / $SecInDay) + " Tage"}
   },
```

```
  @{
    Label="Maximale Aufbewahrungszeit für Datensicherungen" ;
   expression={ "{0:n2}"-f ($_.RPLifeInterval / $SecInDay) + " Tage" }
   }
```

Das vollständige Skript *GetSystemRestoreSettings.ps1* hat folgenden Aufbau:

GetSystemRestoreSettings.ps1

```
Param($computer = "localhost", $help)

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GetOffLineFiles.ps1
Ermittelt die Systemwiederherstellungskonfiguration auf der lokalen Arbeitsstation oder auf einem Remotecomputer.

PARAMETER:
-computer Gibt den Namen des Computers an, der mit diesem Skript überprüft werden soll.
-help     Zeigt das Hilfethema für dieses Skript an.

SYNTAX:
GetSystemRestoreSettings.ps1 -computer MunichServer

Ermittelt die Systemwiederherstellungskonfiguration auf einem Computer namens MunichServer.

GetSystemRestoreSettings.ps1

Ermittelt die Systemwiederherstellungskonfiguration auf dem lokalen Computer.

GetSystemRestoreSettings.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help){ funline("Ausgeben der Hilfeinformationen ...") ; funhelp }

New-Variable -Name SecInDay -option constant -value 86400
$objWMI = Get-WmiObject -Namespace root\default `
         -Class SystemRestoreConfig -computername $computer
for($i=0; $i -le 15; $i++)
{
 Write-Host -ForegroundColor $i "Ermitteln der Systemwiederherstellungskonfiguration"
 Start-Sleep -Milliseconds 60
 cls
}

if($computer -eq "localhost")
 {
  Write-Host "Systemwiederherstellungseinstellungen auf $env:computername"
 }
 ELSE
```

```
 {
  Write-Host "Systemwiederherstellungseinstellungen auf $computer"
 }

format-table -InputObject $objWMI -property `
  @{
    Label="Maximale Datenträgerauslastung" ;
   expression={  "{0:n0}"-f ($_.DiskPercent ) + " %"}
   },
  @{
    Label="Geplante Datensicherung" ;
   expression={  "{0:n2}"-f ($_.RPGlobalInterval / $SecInDay) + " Tage"}
   },
  @{
    Label="Maximale Aufbewahrungszeit für Datensicherungen" ;
   expression={ "{0:n2}"-f ($_.RPLifeInterval / $SecInDay) + " Tage" }
   }
```

Auflisten der verfügbaren Systemwiederherstellungspunkte

Sie sollten mit den aktuellen Einstellungen der Systemwiederherstellung vertraut sein, da diese nützlich sein können. Sie müssen jedoch mindestens wissen, welche Wiederherstellungspunkte verfügbar sind. Um die Wiedherstellungspunkte zu ermitteln, können Sie beispielsweise das Skript *ListSystem-RestorePoints.ps1* verwenden.

Definieren Sie mit der *param*-Anweisung zwei Befehlszeilenargumente. Hierbei handelt es sich um die gleichen Parameter, die im letzten Skript verwendet wurden: **-computer** und **-help**. Codezeile:

```
param($computer="localhost", $help)
```

Die nächste Funktion ist *funlookup*. Diese Funktion wandelt den codierten Wert um, der von der WMI-Klasse *SystemRestore* zurückgegeben wird, um den Typ des Wiederherstellungspunkts anzuzeigen. Übergeben Sie hierzu den in der Variablen *$strIN* gespeicherten Wert. Sie können den Wert der Variablen *$strIN* in der Funktion ändern und die Variable anschließend außerhalb der Funktion verwenden. Die *switch*-Anweisung stimmt den ursprünglichen Wert mit der Funktion *funlookup* ab. Dieser Code ist im Folgenden dargestellt:

```
function funLookup([ref]$strIN)
{
 switch($strIN.value)
  {
   0   { $strIN.value = "ANWENDUNGSINSTALLATION" }
   1   { $strIN.value = "ANWENDUNGSDEINSTALLATION" }
   7   { $strIN.value = "GEPLANTER WIEDERHERSTELLUNGSPUNKT" }
   13  { $strIN.value = "ABGEBROCHENER VORGANG" }
   10  { $strIN.value = "GERäTETREIBERINSTALLATION " }
   12  { $strIN.value = "KONFIGURATIONSÄNDERUNG" }
  }
}
```

Verwenden Sie als Nächstes die Funktion *funhelp*, die der Funktion im letzten Skript ähnlich ist. Die Funktion zeigt die Hilfe an, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Der Code umfasst eine *here*-Zeichenfolge zum Erstellen des Hilfetextes und weist den Text der Variablen *$helpText* zu. Anschließend wird der Inhalt der Variablen ausgegeben und das Skript beendet. Code:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ListSystemRestorePoints.ps1
Ermittelt die Systemwiederherstellungspunkte auf der lokalen Arbeitsstation oder auf einem Remotecomputer.

PARAMETER:
-computer Gibt den Namen des Computers an, der mit diesem Skript überprüft werden soll.
-help     Zeigt das Hilfethema für dieses Skript an.

SYNTAX:
ListSystemRestorePoints.ps1-computer MunichServer

Ermittelt die Systemwiederherstellungspunkte auf einem Computer namens MunichServer.

ListSystemRestorePoints.ps1

Ermittelt die Systemwiederherstellungspunkte auf dem lokalen Computer.

ListSystemRestorePoints.ps1-help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Überprüfen Sie, ob die Variable *$help* vorhanden ist. Gibt es die Variable, rufen Sie zuerst die Funktion *funline* auf, um die Statusmeldung zu unterstreichen, und anschließend die Funktion *funhelp*. Code:

```
if($help){ funline("Ausgeben der Hilfeinformationen  ...") ; funhelp }
```

Stellen Sie anschließend eine Verbindung zum WMI-Dienst her und rufen Sie die Liste der Wiederherstellungspunkte ab. Verwenden Sie hierzu das Cmdlet **Get-WmiObject** und geben Sie den Parameter **-class** an, um die WMI-Klasse *SystemRestore* abzurufen. Da sich diese Klasse im WMI-Namespace *root\default* befindet, müssen Sie das Verzeichnis mit dem Parameter **-namespace** angeben. Stellen Sie die Verbindung im Skript mit dem im Parameter **-computername** angegebenen Computer her und übergeben Sie das resultierende Objekt an das nächste Cmdlet. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
Get-WmiObject -Class systemrestore -namespace root\default  `
              -computername $computer |
```

Verwenden Sie als Nächstes das Cmdlet **Format-Table**. Erstellen Sie mit dem aus dem Cmdlet **Get-WmiObject** zurückgegebenen Objekt eine Ausgabetabelle. Geben Sie mit einer Hashtabelle eine Tabelle mit berechneten Werten und geänderten Spaltenheadern aus. Um die von WMI zurückgegebene Datum/Zeit-Zeichenfolge in einen normalen Datum/Zeit-Wert zu konvertieren, verwenden Sie die .NET Framework-Klasse *Management.ManagementDatetimeCoverter*. Diese .NET Framework-Klasse verwendet die *toDateTime()*-Methode, um das WMI-Zeitformat zu konvertieren. Wandeln Sie mit der Funktion *funlookup* den Wert des Wiedherstellungspunkts in einen übersichtlicheren Zeichenfolgenwert um. Geben Sie die Sequenznummer unverändert aus und geben Sie den Parameter **-autosize** an. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
format-Table -property `
   @{
      Label = "Erstellungszeitpunkt" ;
      expression = { $([Management.ManagementDatetimeConverter]::`
     toDateTime($_.creationTime)) }
   },
   "description",
   @{
      Label = "Typ des Wiederherstellungspunktes" ;
      expression = { $strIN = $_.restorepointtype ;
     funlookup([ref]$strIN) ; $strIN }
    },
   "SequenceNumber" -autosize
```

Das vollständige Skript *ListSystemRestorePoints.ps1* hat folgenden Aufbau:

ListSystemRestorePoints.ps1

```
param($computer="localhost", $help)

function funLookup([ref]$StrIN)
{
 switch($strIN.value)
  {
   0   { $strIN.value = "ANWENDUNGSINSTALLATION" }
   1   { $strIN.value = "ANWENDUNGSDEINSTALLATION" }
   7   { $strIN.value = "GEPLANTER WIEDERHERSTELLUNGSPUNKT" }
   13  { $strIN.value = "ABGEBROCHENER VORGANG" }
   10  { $strIN.value = "GERäTETREIBERINSTALLATION " }
   12  { $strIN.value = "KONFIGURATIONSÄNDERUNG" }
  }
}

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ListSystemRestorePoints.ps1
Ermittelt die Systemwiederherstellungspunkte auf der lokalen Arbeitsstation oder auf einem Remotecomputer.

PARAMETER:
-computer Gibt den Namen des Computers an, der mit diesem Skript überprüft werden soll.
-help     Zeigt das Hilfethema für dieses Skript an.

SYNTAX:
ListSystemRestorePoints.ps1-computer MunichServer

Ermittelt die Systemwiederherstellungspunkte auf einem Computer namens MunichServer.

ListSystemRestorePoints.ps1

Ermittelt die Systemwiederherstellungspunkte auf dem lokalen Computer.

ListSystemRestorePoints.ps1-help ?

Zeigt das Hilfethema für dieses Skript an.
```

```
"@
$helpText
exit
}

if($help){ funline("Ausgeben der Hilfeinformationen ...") ; funhelp }
Get-WmiObject -Class systemrestore -namespace root\default  `
              -computername $computer |
format-Table -property `
   @{
      Label = "Erstellungszeitpunkt" ;
      expression = { $([Management.ManagementDatetimeConverter]::`
     toDateTime($_.creationTime)) }
   },
   "description",
   @{
      Label = "Typ des Wiederherstellungspunktes" ;
      expression = { $strIN = $_.restorepointtype ;
     funlookup([ref]$strIN) ; $strIN }
    },
   "SequenceNumber" -autosize
```

Zusammenfassung

In diesem Kapitel wurden die verschiedenen Methoden zum Verwalten von Benutzerdaten auf einem Windows Vista- oder Windows Server 2008-Computer beschrieben. Als Erstes wurde das Sichern von Daten mit einem Skript erklärt, um den Inhalt eines Ordners in einer Dateifreigabe im Netzwerk zu sichern. Ein Netzwerkadministrator kann anschließend die Dateien mit einem Sicherungsprogramm auf Band oder in einem SAN archivieren.

Als Nächstes wurde die Konfiguration von Offline-Dateien in Windows Server 2008 und Windows Vista erklärt, einschließlich der entsprechenden Einstellungen sowie die Erstellung eines Skripts, das Offline-Dateien aktiviert oder deaktiviert. Außerdem wurden die Bereichsdefinitionen für Variablen und deren Verwendung in Funktionen beschrieben.

Das Kapitel wurde mit einer Beschreibung der Systemwiederherstellung abgeschlossen, einschließlich eines Skripts, das die aktuellen Einstellungen zurückgibt, und eines weiteren Skripts, um alle Wiederherstellungspunkte auf einem Computer aufzulisten.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

KAPITEL 12

Behandeln von Windows-Problemen

Nach Abschluss dieses Kapitels können Sie:

- Startprobleme beheben
- Mit Dienstabhängigkeiten arbeiten
- Hardwareprobleme beheben
- Netzwerkprobleme beheben

Auf der Begleit-CD Alle Skripts in diesem Kapitel befinden sich auf der Begleit-CD im Ordner *\Scripts\Chapter12*.

Beheben von Startproblemen

Wenn Windows nicht ordnungsgemäß gestartet wird, sollten Sie einige Komponenten überprüfen, um festzustellen, was mit dem Computer passiert ist. In Windows Vista und Windows Server 2008 können Sie die Startkonfiguration auf Konfigurationsprobleme hin überprüfen. Außerdem können Sie die Startdienste und Abhängigkeiten überprüfen. Die Dienstabhängigkeiten können wertvolle Hinweise auf ein Problem bieten. Wenn Dienst A von Dienst B abhängt und Dienst A nicht ausgeführt wird, sollten Sie Dienst B überprüfen. Normalerweise ist die Problembehandlung jedoch komplizierter und Sie benötigen Skripts, um die Problemursachen zu erkennen. Diese Vorgehensweise wird in dieser Lektion beschrieben.

Überprüfen der Startkonfiguration

In der Startkonfiguration eines Windows Vista- oder Windows Server 2008-Computers finden Sie möglicherweise wichtige Informationen zum Beheben von Startproblemen. Beispielsweise sind die Informationen zur Startpartition, zum Startverzeichnis und zum Arbeitsverzeichnis hilfreich. Beispielsweise können Sie das Skript *DisplayBootConfig.ps1* verwenden, um diese Informationen zu analysieren. Das Abrufen dieser Informationen kann jedoch längere Zeit in Anspruch nehmen und sich negativ auf die Betriebszeit auswirken.

Geben Sie im Skript *DisplayBootConfig.ps1* als Erstes eine *param*-Anweisung an, die Ihnen das Ändern des Zielcomputers beim Ausführen des Skripts ermöglicht. Sie können außerdem die Hilfeinforma-tionen abrufen. Der Parameter **-help** ist ein feststehender Parameter. Das heißt, dass für diesen Para-meter normalerweise kein Wert angegeben wird. Die *param*-Anweisung ist in folgender Codezeile dargestellt:

```
param($computer="localhost", [switch]$help)
```

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Rufen Sie die Funktion *funhelp* auf, wenn der Parameter **-help** beim Ausführen des Skripts angegeben wird. Erstellen Sie in der Funktion die Variable *$helpText* mit einer *here*-Zeichenfolge, um den Hilfetext zu definieren. Beziehen Sie die Beschreibung, Parameter und Syntax des Skripts in den Hilfetext ein. Geben Sie den Text in der Variablen *$helptext* aus und beenden Sie danach die Ausführung des Skripts. Die Funktion *funhelp* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: DisplayBootConfig.ps1
Zeigt die Startkonfiguration eines Windows-Systems an.

PARAMETER:
-computer    Der Name des Computers.
-help        Zeigt dieses Hilfethema an.

SYNTAX:
DisplayBootConfig.ps1 -computer munich

Zeigt die Startkonfiguration eines Computers namens munich an.

DisplayBootConfig.ps1

Zeigt die Startkonfiguration des lokalen Computers an.

DisplayBootConfig.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Wenn die Variable *$help* vorhanden ist, wurde das Skript mit dem Parameter **-help** ausgeführt. Geben Sie eine Meldung aus, dass die Hilfe abgerufen wird und rufen Sie die Funktion *funhelp* auf. Um zwei Befehle in der gleichen Zeile einzugeben, trennen Sie diese durch ein Semikolon:

```
if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
```

Rufen Sie mit dem Cmdlet **Get-WmiObject** die Informationen aus der WMI-Klasse *Win32_BootConfiguration* ab und weisen Sie die Zeichenfolge der Variablen *$computer* dem Parameter **-computername** des Cmdlets **Get-WmiObject** zu. Auf diese Weise können Sie gegebenenfalls Remoteverbindungen unterstützen. Die Codezeile ist im Folgenden dargestellt. Beachten Sie, dass der Code mit dem Graviszeichen auf der nächsten Zeile fortgesetzt wird. Der Zeilenumbruch verbessert lediglich die Lesbarkeit und hat keinen Einfluss auf die Codeausführung.

```
$wmi = Get-WmiObject -Class win32_BootConfiguration `
     -computername $computer
```

Übergeben Sie das resultierende *Management*-Objekt an das Cmdlet **Format-List**. Verwenden Sie den Bereichsoperator *[a-z]**, um ausschließlich die Eigenschaften auszuwählen, die mit einem Buchstaben beginnen. Der Bereichsoperator schließt alle Systemeigenschaften aus dem Bericht aus. Codezeile:

```
format-list -InputObject $wmi [a-z]*
```

Das vollständige Skript *DisplayBootConfig.ps1* lautet:

DisplayBootConfig.ps1

```
param($computer="localhost", [switch]$help)

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: DisplayBootConfig.ps1
Zeigt die Startkonfiguration eines Windows-Systems an.

PARAMETER:
-computer    Der Name des Computers.
-help        Zeigt dieses Hilfethema an.

SYNTAX:
DisplayBootConfig.ps1 -computer munich

Zeigt die Startkonfiguration eines Computers namens munich an.

DisplayBootConfig.ps1

Zeigt die Startkonfiguration des lokalen Computers an.

DisplayBootConfig.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }

$wmi = Get-WmiObject -Class win32_BootConfiguration `
     -computername $computer
format-list -InputObject $wmi [a-z]*
```

Überprüfen der Startdienste

Viele Dienste werden automatisch gestartet. Wenn einer dieser Dienste nicht gestartet wird, kann dies zur Instabilität des Systems oder unvorhergesehenen Ergebnissen führen. Überprüfen Sie bei Problemen als Erstes die Konsole **Dienste**, sortieren Sie nach dem Starttyp **Automatisch** und suchen Sie nach beendeten Diensten (siehe Abbildung 12.1).

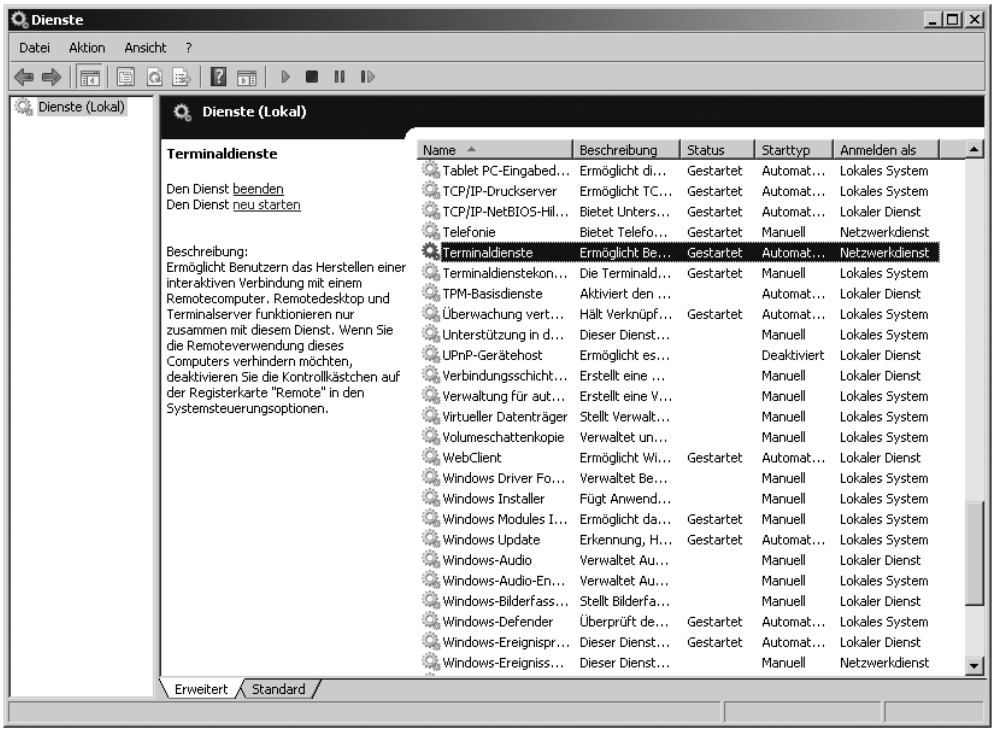


Abbildung 12.1 Das Suchen nach beendeten Diensten ist der erste Schritt bei der Problembehandlung

Der erste Schritt beim Ausführen des Skripts *AutoServicesNotRunning.ps1* ist das Abrufen der WMI-Klasse *Win32_Service* mit dem Cmdlet **Get-WmiObject**. Passen Sie die Abfrage an, um ausschließlich die Dienste zurückzugeben, die automatisch gestartet, aber nicht ausgeführt werden. Wenn keiner der automatisch gestarteten Dienste beendet ist, geben Sie eine entsprechende Meldung aus.

Beginnen Sie das Skript *AutoServicesNotRunning.ps1* mit der *param*-Anweisung.

```
param($computer="localhost", [switch]$help)
```

Weisen Sie unter Verwendung der Funktion *funhelp* der Variablen *$helpText* eine *here*-Zeichenfolge zu. Die *here*-Zeichenfolge umfasst jeweils einen Abschnitt für die Beschreibung, Parameter und Syntax. Nachdem Sie die *here*-Zeichenfolge erstellt haben, zeigen Sie den Inhalt der Variablen *$helptext* an und beenden Sie die Ausführung des Skripts. Die Funktion *funhelp* ist wie folgt definiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: AutoServicesNotRunning.ps1
Zeigt eine Liste der Dienste an, die automatisch gestartet, aber nicht ausgeführt werden.

PARAMETER:
-computer   Der Name des Computers.
-help       Zeigt dieses Hilfethema an.

SYNTAX:
AutoServicesNotRunning.ps1 -computer munich
```

```
Zeigt eine Liste der Dienste auf einem Computer namens munich an,
die automatisch gestartet, aber nicht ausgeführt werden.

AutoServicesNotRunning.ps1

Zeigt eine Liste der Dienste auf dem lokalen Computer an,
die automatisch gestartet, aber nicht ausgeführt werden.

AutoServicesNotRunning.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Sie müssen feststellen, ob die Hilfe angezeigt werden soll. Die Hilfe wird entsprechend der Variablen *$help* angezeigt. Die Variable *$help* existiert jedoch nur, wenn der Parameter **-help** beim Aufrufen des Skripts angegeben wurde. Codezeile:

```
if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
```

Der nächste Schritt basiert auf einer WMI-Abfrage. Rufen Sie mit dem Cmdlet **Get-WmiObject** die WMI-Klasse *Win32_Service* ab und geben Sie den Parameter **-computername** an, um das lokale System oder ein Remotesystem abzufragen. Geben Sie den Parameter **-filter** an, um die Anzahl der zurückgegebenen Instanzen der Klasse *Win32_Service* zu reduzieren. Beachten Sie, dass Sie nur Dienste mit dem Starttyp **Automatisch** abrufen möchten, die nicht ausgeführt werden. Dieser Skriptabschnitt ist im Folgenden dargestellt:

```
$wmi = Get-WmiObject -Class win32_service -computername $computer `
   -filter "state <> 'running' and startmode = 'auto'"
```

Im Anschluss an die WMI-Abfrage müssen Sie die Ergebnisse auswerten. Wenn die Variable *$wmi* den Wert 0 hat, sind keine automatisch gestarteten, aber beendete Dienste vorhanden. Codebeispiel:

```
if($wmi -eq $null)
  { "Es gibt keine automatisch gestartete, aber beendete Dienste." }
```

Wenn Dienste beendet sind, geben Sie die Dienstnamen aus. Listen Sie die beendeten Dienste auf und geben Sie mit einer *foreach*-Anweisung die Namen dieser Dienste aus. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
Else
  {
   "Es gibt $($wmi.count) automatisch gestartete, aber beendete Dienste ... "
   foreach($service in $wmi) { $service.name }
   }
```

Das vollständige Skript *AutoServicesNotRunning.ps1* hat folgenden Inhalt:

AutoServicesNotRunning.ps1

```
param($computer="localhost", [switch]$help)

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
```

```
NAME: AutoServicesNotRunning.ps1
Zeigt eine Liste der Dienste an, die automatisch gestartet, aber nicht ausgeführt werden.

PARAMETER:
-computer    Der Name des Computers.
-help        Zeigt dieses Hilfethema an.

SYNTAX:
AutoServicesNotRunning.ps1 -computer munich

Zeigt eine Liste der Dienste auf einem Computer namens munich an,
die automatisch gestartet, aber nicht ausgeführt werden.

AutoServicesNotRunning.ps1

Zeigt eine Liste der Dienste auf dem lokalen Computer an,
die automatisch gestartet, aber nicht ausgeführt werden.

AutoServicesNotRunning.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }

$wmi = Get-WmiObject -Class win32_service -computername $computer `
   -filter "state <> 'running' and startmode = 'auto'"
if($wmi -eq $null)
  { "Es gibt keine automatisch gestarteten, aber beendete Dienste." }
Else
  {
   "Es gibt $($wmi.count) automatisch gestartete, aber beendete Dienste ... "
   foreach($service in $wmi) { $service.name }
   }
```

Anzeigen der Dienstabhängigkeiten

Wenn ein Dienst nicht gestartet werden kann, wirkt sich dies möglicherweise nicht nur auf die von diesem Dienst bereitgestellte Funktionalität aus. Diese Tatsache wird als Dienstabhängigkeit bezeichnet, die auf zahlreiche Anwendungen zutrifft. Als ich beispielsweise die Software installierte, die ich mit meinem Zune erhalten habe, wurde der Netzwerkfreigabedienst (Network Sharing Service) auf meinem Computer installiert. Dieser Dienst verwendet den Dienst UPnP-Gerätehost, um andere Zune-Geräte im Netzwerk zu suchen. Wenn der Dienst UPnP-Gerätehost nicht ausgeführt wird, kann der Netzwerkfreigabedienst nicht gestartet werden. Der Netzwerkfreigabedienst kann jedoch auch für die Kommunikation mit anderen Geräten über das Internet verwendet werden. Hierzu hängt der Dienst vom HTTP-Dienst ab.

Der Vorteil von Dienstabhängigkeiten ist, dass ein Entwickler die Funktionen der bereits auf dem Computer vorhandenen Dienste wiederverwenden kann. Der Nachteil der Dienstabhängigkeiten ist, dass das Nachverfolgen der Beziehungen zwischen scheinbar nicht verwandten Diensten schwierig ist.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Sie können die Abhängigkeitsinformationen in der Konsole **Dienste** zusammenstellen. Doppelklicken Sie in der Dienstliste auf den gewünschten Dienst und wählen Sie die Registerkarte **Abhängigkeiten** aus (siehe Abbildung 12.2).

Mit dem Skript *ServiceDependencies.ps1* können Sie alle Dienste und ihre jeweiligen Abhängigkeiten anzeigen.

Beginnen Sie das Skript mit der Anweisung *$erroractionpreference = "SilentlyContinue"*, da möglicherweise Dienste vorhanden sind, auf die Sie auch als Mitglied der lokalen Administratorgruppe nicht zugreifen können. Damit Sie die Sicherheitseinstellungen der Dienste nicht zu ändern brauchen, sollten Sie zum Behandeln von Fehlern die automatische Variable *$erroractionpreference* verwenden und dieser die Zeichenfolge *SilentlyContinue* zuweisen. Dieser Vorgang wird mit folgender Codezeile ausgeführt:

```
$erroractionpreference = "SilentlyContinue"
```

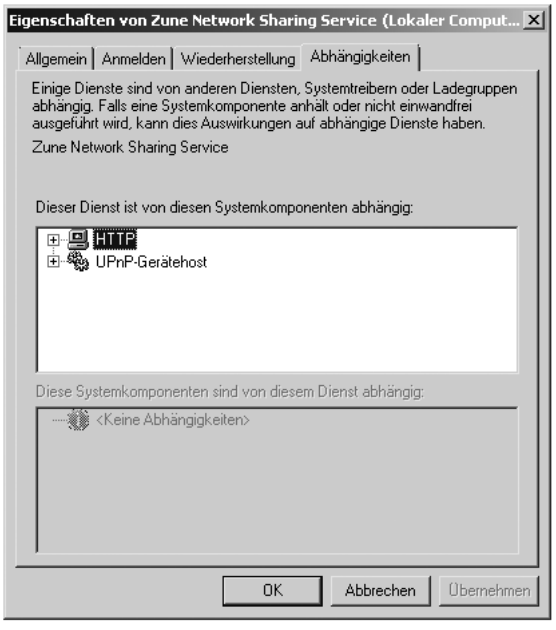


Abbildung 12.2 Die in der Konsole *Dienste* angezeigten Abhängigkeiten eines Dienstes

Definieren Sie mit der *param*-Anweisung mehrere Befehlszeilenparameter für das Skript. Der erste Parameter ist **-computer**, um den Computer angeben zu können, für den das Skript ausgeführt werden soll. Der zweite Parameter ist ein feststehender Parameter, um auf Anforderung die Hilfe anzuzeigen. Codezeile:

```
Param($computer = "localhost", [switch]$help)
```

Definieren Sie als Nächstes die Funktion *funline*, um Abschnitte der Ausgabe zu unterstreichen. Übergeben Sie hierzu einen Zeichenfolgenwert an die Funktion, mit dem die Länge der Eingabezeichenfolge bestimmt wird und die Zeichen anschließend mit einer *for*-Anweisung zusammengefügt werden. Die Zeichenanzahl wird verwendet, um die entsprechende Anzahl an Gleichheitszeichen zu verknüpfen. Die Gleichheitszeichen werden in der Variablen *$funline* gespeichert, die mit dem Cmdlet **Write-Host** unter der Textzeile ausgegeben wird. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline = $funline + "=" }
    Write-Host -ForegroundColor yellow $strIN
    Write-Host -ForegroundColor darkYellow $funline
}
```

Erstellen Sie die Funktion *funhelp*, um die Hilfe anzuzeigen. Diese Funktion besteht hauptsächlich aus einer *here*-Zeichenfolge, die der Variablen *$helpText* zugewiesen ist. Nachdem die *here*-Zeichenfolge erstellt und der Variablen *$helpText* zugewiesen wurde, zeigen Sie den Inhalt der Variablen an und beenden die Ausführung des Skripts:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ServiceDependencies.ps1
Zeigt eine Liste der Dienste und deren Abhängigkeiten an.

PARAMETER:
-computer    Der Name des Computers.
-help        Zeigt dieses Hilfethema an.

SYNTAX:
ServiceDependencies.ps1 -computer munich

Zeigt eine Liste der Dienste und deren Abhängigkeiten auf einem Computer namens munich an.

ServiceDependencies.ps1

Zeigt eine Liste der Dienste und deren Abhängigkeiten auf dem lokalen Computer an.

ServiceDependencies.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Sie müssen bestimmen, ob die Hilfe angezeigt werden soll. Zeigen Sie die Hilfe nur dann an, wenn die Variable *$help* vorhanden ist. Überprüfen Sie mit *if*, ob dies der Fall ist und geben Sie eine Zeichenfolge mit dem Hilfetext aus. Rufen Sie anschließend die Funktion *funhelp* mit folgender Codezeile auf:

```
if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
```

Wenn die Hilfe nicht anzeigt werden muss, erstellen Sie zwei Variablen für die Listen der abzufragenden Eigenschaften. Diese Variablen sind $*dependentProperty* und *$antecedentProperty*. Die Eigenschaften entsprechen den Eigenschaften, die Sie mit der WMI-Abfrage aus den WMI-Klassen abrufen möchten. Da Sie eine Assoziationsklasse verwenden, gibt diese die gewünschten Daten nicht zurück. Stattdessen werden die von der Abfrage zurückgegebenen Zeiger aufgeführt. Verwenden Sie diese Zeiger, um die Dienstinformationen abzurufen. Der folgende Code demonstriert diese Vorgehensweise:

```
$dependentProperty =  "name", "displayname", "pathname",
                      "state", "startmode", "processID"
$antecedentProperty = "name", "displayname",
                      "state", "processID"
```

Überprüfen Sie den in der Variablen *$computer* gespeicherten Wert. Die Variable *$computer*, die in der *param*-Anweisung auf *localhost* festgelegt ist, verweist standardmäßig auf den lokalen Computer. Wenn das Skript mit dem Parameter **-computer** ausgeführt wird, hat die Variable *$computer* einen anderen Wert. Verwenden Sie den tatsächlichen Computernamen, wenn der Benutzer den Wert von *$computer* nicht geändert hat. Rufen Sie den Wert mit einer Abfrage der *PSDrive*-Umgebungsvariablen ab. Codezeile:

```
if($computer = "localhost") { $computer = $env:computername }
```

Nachdem Sie die Variable *$computer* überprüft haben, geben Sie mit der Funktion *funline* eine Kopfzeile für den Bericht mit den Dienstabhängigkeiten aus. Codezeile für die Überschriftausgabe:

```
funline("Dienstabhängigkeiten auf $($computer)")
```

Erstellen Sie mit dem Cmdlet **New-Variable** und dem Parameter **-option** die Konstante *c_padline*. Der Parameter **-name** des Cmdlets **New-Variable** erfordert nicht, dass dem Variablennamen ein Dollarzeichen vorangestellt wird.

Hinweis Eine im Methodenaufruf hart codierte Zahl wird auch als Magic Number bezeichnet . Das Vermeiden von Magic Numbers ist eine sinnvolle Programmierrichtlinie.

Der Grund für das Erstellen einer Konstanten ist das Auffüllen der 14 Zeichen in der Zeile, um den Code einfacher lesen und verwalten zu können. Codezeile:

```
New-Variable -Name c_padline -value 14 -option constant
```

Fragen Sie mit dem Cmdlet **Get-WmiObject** die WMI-Klasse *Win32_DependentService* ab. Diese WMI-Klasse ist eine Assoziationsklasse, die die beiden WMI-Klassen miteinander verknüpft. Die beiden Klassen sind *Win32_BaseService* und *Win32_BaseService*. Nein, dies ist kein Versehen. Die Klasse wird mit sich selbst verknüpft. Auf diese Art können Sie feststellen, welche Dienste von anderen Diensten abhängig sind. Verwenden Sie den Parameter **-computername** des Cmdlets **Get-WmiObject**, um die Abfrage des lokalen Computers oder von Remotecomputern mit dem Skript zu ermöglichen. Schließen Sie den Befehl mit einer Pipeline ab. Codezeile:

```
Get-WmiObject -Class Win32_DependentService -computername $computer |
```

Fügen Sie das resultierende Objekt in das Cmdlet **ForEach-Object** ein. Da die Ausgabe möglicherweise sehr lang und verwirrend ist, sollten Sie zueinander gehörige Dienste markieren. Erstellen Sie hierzu eine Trennzeile, die so lang wie die längste angezeigte Eigenschaft ist und geben Sie die Kopfzeile für die Ausgabe aus:

```
Foreach-object `
 {
  "=" * ((([wmi]$_.dependent).pathname).length + $c_padline)
  Write-Host -ForegroundColor blue "Der Dienst:"
```

Rufen Sie mit dem *[WMI] Management*-Objekt die Informationen zum abhängigen Dienst ab und fügen Sie das resultierende *Management*-Objekt ein:

```
[wmi]$_.Dependent |
```

Geben Sie mit dem Cmdlet **Format-List** alle in der Variablen *$dependentproperty* gespeicherten Eigenschaften aus. Codezeile:

```
format-list -Property $dependentProperty
```

Ändern Sie die Farben und geben Sie eine weitere Kopfzeile für die Dienste aus, von denen der Dienst abhängt. Diese Dienste finden Sie in der Eigenschaft *Antecedent*. Fügen Sie diese Informationen wie folgt ein:

```
Write-Host -ForegroundColor cyan "Von diesem Dienst abhängig:"
    [wmi]$_.Antecedent |
```

Geben Sie die Informationen zu diesem Dienst mit dem Cmdlet **Format-List** aus. Die entsprechenden Eigenschaften sind in der Variablen *$antecedentproperty* gespeichert. Fügen Sie, wie in folgendem Codeabschnitt dargestellt, eine Trennzeile am Ende der Ausgabe ein:

```
      format-list -Property $antecedentProperty
        "=" * ((([wmi]$_.dependent).pathname).length + $c_padline) + "`n"
 }
```

Das vollständige Skript *ServiceDependencies.ps1* hat folgenden Aufbau:

ServiceDependencies.ps1

```
$erroractionpreference = "SilentlyContinue"
Param($computer = "localhost", [switch]$help)

function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline = $funline + "=" }
    Write-Host -ForegroundColor yellow $strIN
    Write-Host -ForegroundColor darkYellow $funline
}

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ServiceDependencies.ps1
Zeigt eine Liste der Dienste und deren Abhängigkeiten an.

PARAMETER:
-computer    Der Name des Computers.
-help        Zeigt dieses Hilfethema an.

SYNTAX:
ServiceDependencies.ps1 -computer munich

Zeigt eine Liste der Dienste und deren Abhängigkeiten auf einem Computer namens munich an.

ServiceDependencies.ps1

Zeigt eine Liste der Dienste und deren Abhängigkeiten auf dem lokalen Computer an.

ServiceDependencies.ps1 -help

Zeigt das Hilfethema für dieses Skript an.
```

```
"@
$helpText
exit
}

if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
$dependentProperty =  "name", "displayname", "pathname",
                      "state", "startmode", "processID"
$antecedentProperty = "name", "displayname",
                      "state", "processID"

if($computer = "localhost") { $computer = $env:computername }
funline("Dienstabhängigkeiten auf $($computer)")

New-Variable -Name c_padline -value 14 -option constant
Get-WmiObject -Class Win32_DependentService -computername $computer |
Foreach-object `
 {
  "=" * ((([wmi]$_.dependent).pathname).length + $c_padline)
  Write-Host -ForegroundColor blue "Dieser Dienst:"
    [wmi]$_.Dependent |
      format-list -Property $dependentProperty
  Write-Host -ForegroundColor cyan "Von diesem Dienst abhängig:"
    [wmi]$_.Antecedent |
      format-list -Property $antecedentProperty
        "=" * ((([wmi]$_.dependent).pathname).length + $c_padline) + "`n"
 }
```

Überprüfen der Gerätetreiber

Gerätetreiber sind Diensten insofern ähnlich, dass sie automatisch gestartet werden und Funktionen für den Computer bereitstellen. Gerätetreiber sind jedoch nicht so einfach wie Dienste zu finden. Bei bestimmten Gerätetreibern ist es oft schwierig, zu verstehen, welche Vorgänge diese tatsächlich ausführen.

Beginnen Sie das Skript *CheckDeviceDrivers.ps1* mit einer *param*-Anweisung und definieren Sie drei Parameter. Der erste Parameter ist -**computer** und standardmäßig auf *localhost* festgelegt. Der zweite Parameter lautet **-a**, um die auszuführende Aktion anzugeben. Dieser Parameter ist standardmäßig auf *h* festgelegt, was bewirkt, dass das Skript eine kurze Hilfemeldung anzeigt. Der dritte Parameter ist **-help**, um die Hilfe anzuzeigen. Codezeile:

```
param($computer="localhost", $a="h", [switch]$help)
```

Definieren Sie die Funktion *funhelp*, um die Hilfemeldung auszugeben, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Der Hilfetext besteht aus einer *here*-Zeichenfolge mit jeweils einem Abschnitt für die Beschreibung, Parameter und Syntax. Der Inhalt der *here*-Zeichenfolge wird der Variablen *$helpText* zugewiesen und vor Abschluss der Funktion *funhelp* angezeigt. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: CheckDeviceDrivers.ps1
Zeigt eine Liste der Gerätetreiber an, die entweder alle oder nur bestimmte Treiber umfasst
```

```
(Automatisch, Manuell, Boot, System).

PARAMETER:
-computer    Der Name des Computers.
-a(ktion)    < a (alle), r (ausgeführte), s (beendete), b (boot),
               m (manuelle), au (automatische),  sy (system), h(elp) >
-help        Zeigt dieses Hilfethema an.

SYNTAX:
CheckDeviceDrivers.ps1 -computer munich -a b

Zeigt eine Liste der Gerätetreiber an, die
auf einem Computer namens munich für den
Start während des Bootprozesses konfiguriert wurden.

CheckDeviceDrivers.ps1 -a auto

Zeigt eine Liste der Gerätetreiber an, die auf dem lokalen Computer
für den automatischen Start konfiguriert wurden.

CheckDeviceDrivers.ps1 -computer munich -a m

Zeigt eine Liste aller Gerätetreiber an,  die
auf einem Computer namens munich für den
manuellen Start konfiguriert wurden.

CheckDeviceDrivers.ps1 -help

 Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Um festzustellen, ob Sie eine Hilfezeichenfolge anzeigen müssen, überprüfen Sie mit einer *if*-Anweisung, ob die Variable *$help* vorhanden ist. Existiert die Variable *$help*, zeigen Sie die Hilfemeldung an, indem Sie die Funktion *funhelp* aufrufen. Codezeile:

```
if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
```

Die *switch*-Anweisung im Skript *CheckDeviceDrivers.ps1* ist ziemlich komplex und ermöglicht dem Benutzer das Ausführen mehrerer Abfragen bezüglich der Gerätetreiber. Die *switch*-Anweisung gibt eine Statusmeldung aus und weist der Variablen *$filter* einen Wert zu. Die Variable *$filter* wird später an den Parameter **-filter** des Cmdlets **Get-WmiObject** übergeben.

Der Befehl *$MyInvocation.MyCommand.Definition* bietet eine Möglichkeit, den Namen des ausgeführten Skripts zu ermitteln. Sie können den Befehl in mehreren *switch*-Anweisungen verwenden, um in der Meldung auf das ausgeführte Skript hinzuweisen.

Tipp Um eine Funktion in einem anderen Skript erneut zu verwenden und das ausgeführte Skript anzuzeigen, führen Sie die Anweisung *$MyInvocation.MyCommand.Definition* aus. Das Skript *CheckDeviceDrivers.ps1* verwendet diesen Befehl, um eine Hilfemeldung für den Benutzer auszugeben. Dieser Befehl ist in Anhang D „Skriptrichtlinien“ beschrieben.

Die vollständige *switch*-Anweisung:

```
switch($a)
{
 "a" {
      "Ermitteln aller Gerätetreiber."
     $filter = "started = 'true' or started = 'false'"
    }
 "r" {
      "Ermitteln aller ausgeführten Gerätetreiber."
     $filter = "started = 'true'"
    }
 "s" {
      "Ermitteln aller beendeten Gerätetreiber."
     $filter = "started = 'false'"
    }
 "b" {
      "Ermitteln aller Bootgerätetreiber."
     $filter = "startmode = 'boot'"
    }
 "m" {
      "Ermitteln aller für den manuellen Start konfigurierten Gerätetreiber."
     $filter = "startmode = 'manual'"
    }
 "au" {
      "Ermitteln aller für den automatischen Start konfigurierten Gerätetreiber."
     $filter = "startmode = 'auto'"
    }
 "sy" {
      "Ermitteln aller Systemgerätetreiber."
     $filter = "startmode = 'system'"
    }
 "h" {
      "Sie müssen eine Aktion angeben. Der Parameter -a ist erforderlich."
     "Für weitere Informationen führen Sie folgenden Befehl aus: " +  $MyInvocation.MyCommand.Definition + " -h"
     exit
    }
 DEFAULT
      {
      "Sie müssen eine Aktion angeben. Der Parameter -a ist erforderlich."
     "Für weitere Informationen führen Sie folgenden Befehl aus:  " +  $MyInvocation.MyCommand.Definition + " -h"
     exit
    }
}
```

Nachdem Sie den Wert der Variablen *$a* überprüft haben, fahren Sie mit dem Cmdlet **Get-WmiObject** fort, das die WMI-Klasse *Win32_SystemDriver* abfragt. Führen Sie die Abfrage für den in der Variablen *$computer* angegebenen Computer aus und geben Sie über die *switch*-Anweisung einen Filter an. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
$wmi = Get-WmiObject -Class win32_systemdriver `
     -computername $computer -filter $filter
```

Formatieren Sie die in der Variablen *$wmi* gespeicherte Ausgabe. Verwenden Sie hierzu das Cmdlet **Format-Table** mit dem Parameter **-inputobject** und geben Sie das in der Variablen *$wmi* gespeicherte *Management*-Objekt an. Wählen Sie drei Eigenschaften aus und geben Sie den Parameter **-autosize** an. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
format-table -InputObject $wmi -property `
      displayname, pathname, name  -autosize
```

Das vollständige Skript *CheckDeviceDrivers.ps1* umfasst folgende Anweisungen:

CheckDeviceDrivers.ps1

```
param($computer="localhost", $a="h", [switch]$help)

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: CheckDeviceDrivers.ps1
Zeigt eine Liste der Gerätetreiber an, die entweder alle oder nur bestimmte Treiber umfasst
(Automatisch, Manuell, Boot, System).

PARAMETER:
-computer    Der Name des Computers.
-a(ktion)    < a (alle), r (ausgeführte), s (beendete), b (boot),
               m (manuelle), au (automatische),  sy (system), h(elp) > >
-help        Zeigt dieses Hilfethema an.

SYNTAX:
CheckDeviceDrivers.ps1 -computer munich -a b

Zeigt eine Liste der Gerätetreiber an, die
auf einem Computer namens munich für den
Start während des Bootprozesses konfiguriert wurden.

CheckDeviceDrivers.ps1 -a auto

Zeigt eine Liste der Gerätetreiber an, die auf dem lokalen Computer
für den automatischen Start konfiguriert wurden.

CheckDeviceDrivers.ps1 -computer munich -a m

Zeigt eine Liste aller Gerätetreiber an,  die
auf einem Computer namens munich für den
manuellen Start konfiguriert wurden.

CheckDeviceDrivers.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
```

```
switch($a)
{
 "a" {
      "Ermitteln aller Gerätetreiber."
     $filter = "started = 'true' or started = 'false'"
    }
 "r" {
      "Ermitteln aller ausgeführten Gerätetreiber."
     $filter = "started = 'true'"
    }
 "s" {
      "Ermitteln aller beendeten Gerätetreiber."
     $filter = "started = 'false'"
    }
 "b" {
      "Ermitteln aller Bootgerätetreiber."
     $filter = "startmode = 'boot'"
    }
 "m" {
      "Ermitteln aller für den manuellen Start konfigurierten Gerätetreiber."
     $filter = "startmode = 'manual'"
    }
 "au" {
      "Ermitteln aller für den automatischen Start konfigurierten Gerätetreiber."
      $filter = "startmode = 'auto'"
    }
 "sy" {
      "Ermitteln aller Systemgerätetreiber."
     $filter = "startmode = 'system'"
    }


 "h" {
      "Sie müssen eine Aktion angeben. Der Parameter -a ist erforderlich."
     "Für weitere Informationen führen Sie folgenden Befehl aus: " +  $MyInvocation.MyCommand.Definition + " -h"
     exit
    }
 DEFAULT
      {
      "Sie müssen eine Aktion angeben. Der Parameter -a ist erforderlich."
     "Für weitere Informationen führen Sie folgenden Befehl aus: " +  $MyInvocation.MyCommand.Definition + " -h"
     exit
    }
}

$wmi = Get-WmiObject -Class win32_systemdriver `
     -computername $computer -filter $filter
format-table -InputObject $wmi -property `
      displayname, pathname, name  -autosize
```

Überprüfen der Startprozesse

Einige Prozesse werden automatisch gestartet. Beispielsweise können Sie Programme zur Startgruppe auf einem Windows Vista- oder Windows Server 2008-Computer hinzufügen, um die Prozesse automatisch zu starten. Windows Defender zeigt automatisch ausgeführte Programme an und ermöglicht das Ändern des Startverhaltens (siehe Abbildung 12.3).

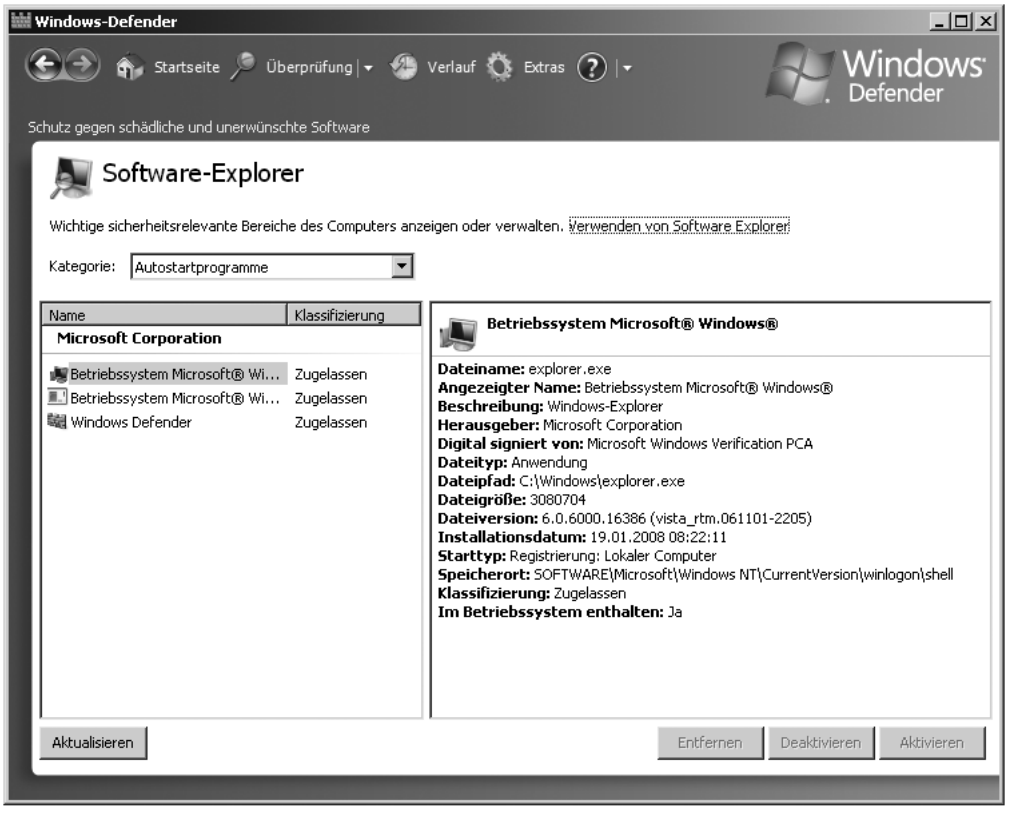


Abbildung 12.3 Windows Defender kann den Start von Prozessen kontrollieren

Um diese Prozesse mit einem Skript zu identifizieren, verwenden Sie die WMI-Klasse *Win32_StartUpCommand*. Mit dem Skript *DetectStartupPrograms.ps1* können Sie beispielsweise die Startprogramme auf einem lokalen Computer oder einem Remotecomputer anzeigen. Außerdem können Sie eine Basisansicht oder eine vollständige Ansicht der Programminformationen zusammenstellen.

Definieren Sie mit der *param*-Anweisung den Parameter **-computer**, um eine Verbindung mit dem lokalen Computer oder einem Remotecomputer herzustellen. Definieren Sie außerdem die beiden Parameter **-full** und **-help**, um umfassende Prozessinformationen bzw. Hilfeinformationen auszugeben. Codezeile:

```
param($computer="localhost", [switch]$full, [switch]$help)
```

Definieren Sie als Nächstes die Funktion *funhelp*. Die Funktion weist der Variablen *$helpText* eine *here*-Zeichenfolge zu. Wenn der Parameter **-help** zur Laufzeit angegeben wird, wird die Funktion *funhelp* ausgeführt. Zeigen Sie den Inhalt der Variablen *$helptext* an und beenden Sie danach die Skriptausführung. Diese Funktion ist im Folgenden dargestellt:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: DetectStartUpPrograms.ps1
Zeigt eine Liste der Programme an, die für den automatischen Start konfiguriert wurden.
```

```
PARAMETER:
-computer    Der Name des Computers.
-full        Zeigt detaillierte Informationen an.
-help        Zeigt dieses Hilfethema an.

SYNTAX:
DetectStartUpPrograms.ps1 -computer munich -full

Zeigt den Namen, die Befehlszeile, das Verzeichnis und Benutzerinformationen
über Programme an, die für den automatischen Start
auf einem Computer namens munich konfiguriert wurden.

DetectStartUpPrograms.ps1 -full

Zeigt den Namen, die Befehlszeile, das Verzeichnis und Benutzerinformationen
über Programme an, die für den automatischen Start
auf dem lokalen Computer konfiguriert wurden.

DetectStartUpPrograms.ps1 -computer munich

Zeigt eine Liste der Programme an, die
auf einem Computer namens munich automatisch gestartet werden.

DetectStartUpPrograms.ps1 -help ?

 Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Sie müssen ermitteln, ob die Funktion *funhelp* aufgerufen werden soll. Die Funktion wird nur aufgerufen, wenn die Variable *$help* vorhanden ist. Die Variable ist nur verfügbar, wenn das Skript mit dem Parameter **-help** ausgeführt wird:

```
if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
```

Bestimmen Sie anschließend, ob umfassende Prozessinformationen angezeigt werden müssen. Wurde der Parameter **-full** beim Ausführen des Skripts angegeben, muss der Name, die Befehlszeile, das Verzeichnis und der Benutzername ausgegeben werden. Zeigen Sie ansonsten nur den Namen des Programms an. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
if($full)
 { $property = "name", "command", "location", "user" }
else
 { $property = "name" }
```

Führen Sie das Cmdlet **Get-WmiObject** aus, um alle Startbefehle abzurufen. Fügen Sie das resultierende Objekt in das Cmdlet **Sort-Object** ein, um die Eigenschaftennamen zu sortieren. Wählen Sie mit dem Cmdlet **Format-List** nur die in der Variablen *$property* angegebenen Eigenschaften aus. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
Get-WmiObject -Class win32_startupcommand -computername $computer |
Sort-Object -property name  |
format-list -property $property
```

Das vollständige Skript *DetectStartupPrograms.ps1* hat folgenden Inhalt:

DetectStartupPrograms.ps1

```
param($computer="localhost", [switch]$full, [switch]$help)

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: DetectStartUpPrograms.ps1
Zeigt eine Liste der Programme an, die für den automatischen Start konfiguriert wurden.

PARAMETER:
-computer    Der Name des Computers.
-full        Zeigt detaillierte Informationen an.
-help        Zeigt dieses Hilfethema an.

SYNTAX:
DetectStartUpPrograms.ps1 -computer munich -full

Zeigt den Namen, die Befehlszeile, das Verzeichnis und Benutzerinformationen
über Programme an, die für den automatischen Start
auf einem Computer namens munich konfiguriert wurden.

DetectStartUpPrograms.ps1 -full

Zeigt den Namen, die Befehlszeile, das Verzeichnis und Benutzerinformationen
über Programme an, die für den automatischen Start
auf dem lokalen Computer konfiguriert wurden.

DetectStartUpPrograms.ps1 -computer munich

Zeigt eine Liste der Programme an, die
auf einem Computer namens munich automatisch gestartet werden.

DetectStartUpPrograms.ps1 -help ?

 Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }

if($full)
 { $property = "name", "command", "location", "user" }
else
 { $property = "name" }

Get-WmiObject -Class win32_startupcommand -computername $computer |
Sort-Object -property name  |
format-list -property $property
```

Überprüfen von Hardwareproblemen

Hardwareprobleme stehen nicht immer mit Hardware in Beziehung. Die meisten elektronischen Geräte können für lange Zeit verwendet werden, wenn Sie innerhalb der entsprechenden Leistungsmatrix betrieben werden. Ein Gerät fällt normalerweise während der ersten Wochen aus, die es in Betrieb ist. Dieser Zeitraum wird als Warmlaufphase bezeichnet. Anschließend sollte das Gerät ordnungsgemäß funktionieren. Das sagt jedoch nichts über die Software aus, die für den Betrieb des Geräts erforderlich ist. Beinahe alle seriösen Hersteller signieren ihre Gerätetreiber digital. Die digitale Signatur stellt sicher, dass der Treiber authentisch ist. Die Signatur ist ausgesprochen wichtig, da die meisten Gerätetreiber mit erhöhten Rechten ausgeführt werden.

Vorsicht Aufgrund des möglichen Mißbrauchs wurde die Richtlinie für Treibersignaturen in Windows Vista und Windows Server 2008 so geändert, dass bei der Installation eines nicht signierten Treibers eine Meldung angezeigt wird. Diese Richtlinie kann nicht umgangen werden. Sie sollten unabhängig vom Hardwarehersteller immer auf signierten Treibern bestehen.

Nicht signierte Gerätetreiber können die Instabilität von Windows Vista und Windows Server 2008 zur Folge haben. Mit dem Skript *CheckSignedDeviceDrivers.ps1* können Sie Ihren Computer dahingehend überprüfen.

Beginnen Sie das Skript *CheckSignedDeviceDrivers.ps1* mit einer *param*-Anweisung. Diese *param*-Anweisung ist anders als die anderen *param*-Anweisungen ausgelegt, die bisher erklärt wurden. Die Anweisung führt jedoch die gleiche Aufgabe aus und ermöglicht dem Benutzer, das Verhalten des Skripts zur Laufzeit zu ändern. Die *param*-Anweisung definiert vier Parameter. Der Parameter **-computer** gibt den Zielcomputer des Vorgangs an. Der Parameter **-unsigned** ist als feststehender Parameter definiert. Der Parameter **-full** erstellt eine detaillierte Liste der Informationen. Der Parameter **-help** zeigt die Hilfe an. Die *param*-Anweisung lautet dementsprechend:

```
param(
      $computer="localhost",
     [switch]$unsigned,
     [switch]$full,
     [switch]$help
    )
```

Anschließend wird die Funktion *funline* erstellt. Sie haben diese Funktion bereits in diesem Kapitel verwendet. In diesem Beispiel wird diese Funktion verwendet, um bestimmte Textabschnitte für die visuelle und räumliche Trennung bei der Ausgabe zu unterstreichen:

```
function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline += "=" }
    Write-Host -ForegroundColor green $strIN
    Write-Host -ForegroundColor darkgreen $funline
}
```

Die nächste Funktion ist *funhelp*, um die Hilfeinformationen anzuzeigen. Diese Funktion wird aufgerufen, wenn der Parameter **-help** angegeben wurde. Der Hilfetext, der der Variablen *$helpText* zugewiesen wird, wird mit einer *here*-Zeichenfolge erstellt. Geben Sie den Inhalt der Variablen *$helptext* aus und beenden Sie danach die Skriptausführung. Die Funktion ist im Folgenden dargestellt:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: CheckSignedDeviceDrivers.ps1
Zeigt eine Liste der Programme an,
die entweder signiert oder nicht signiert wurden.

PARAMETER:
-computer    Der Name des Computers.
-unsigned    Listet die nicht signierten Treiber auf.
-full        Listet die Eigenschaften Description, driverProviderName,
             Driverversion,DriverDate und infName auf.
-help        Zeigt dieses Hilfethema an.

SYNTAX:
CheckSignedDeviceDrivers.ps1 -computer munich -unsigned

Zeigt eine Liste aller nicht signierten Treiber
auf einem Computer namens munich an.

CheckSignedDeviceDrivers.ps1 -unsigned -full

Zeigt eine Liste aller nicht signierten Treiber auf dem lokalen
Computer an und führt in dieser Liste die Treibereigenschaften
Description, driverProviderName, Driverversion,DriverDate und infName auf.

CheckSignedDeviceDrivers.ps1 -computer munich -full

Zeigt eine Liste aller signierten Treiber auf einem Computer namens munich an
und führt in dieser Liste die Treibereigenschaften
Description, driverProviderName, Driverversion,DriverDate und infName auf.

CheckSignedDeviceDrivers.ps1 -help ?

 Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Sie müssen festlegen, ob die Funktion *funhelp* zum Anzeigen der Hilfe aufgerufen werden soll. Stellen Sie als Erstes sicher, dass die Variable *$help* vorhanden ist. Rufen Sie anschließend die Funktion *funhelp* auf. Beispiel:

```
if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
```

Wenn die Variable *$unsigned* vorhanden ist, weisen Sie der Variablen *$filter* die Zeichenfolge *"isSigned = 'false'"* zu. Diese Variable gibt den Parameter **-filter** des Cmdlets **Get-WmiObject** an. Speichern Sie in der Variablen *$mode* eine Statusmeldung, um anzugeben, welche WMI-Abfrage verwendet wird. Wenn die Variable *$unsigned* nicht vorhanden ist, suchen Sie anstatt von nicht signierten nach signierten Treibern. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
if($unsigned)
  { $filter = "isSigned = 'false'" ; $mode = "unsigned" }
ELSE
  { $filter = "isSigned = 'true'" ; $mode = "signed" }
```

Wählen Sie die gewünschten Eigenschaften aus, indem Sie die Namen der Eigenschaften einem Array zuweisen. Dieser Code ist im Folgenden dargestellt:

```
$property = "Description", "driverProviderName", `
            "Driverversion","DriverDate","infName"
```

Der nächste Schritt besteht aus der Abfrage der WMI-Klasse *Win32_PnPSignedDriver* mit dem Cmdlet **Get-WmiObject**. Führen Sie die Abfrage für den im Parameter **-computer** angegebenen Computer aus. Wählen Sie die aufgeführten Eigenschaften mit einem Filter aus. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
$wmi = Get-WmiObject -Class Win32_PnPSignedDriver `
     -computername $computer -property $property -filter $filter
```

Bestimmen Sie mit der Eigenschaft *Count* wie viele Treiber die Kriterien erfüllen und verwenden Sie die Informationen in der Skriptausgabe. Wenn auf dem Computer keine signierten Treiber vorhanden sind, ist der Wert leer anstatt 0. Dieser Skriptabschnitt ist im Folgenden dargestellt:

```
funline("Es gibt $($wmi.count) $mode Treiber, siehe folgende Liste:")
```

Sie müssen den Informationsumfang der generierten Ausgabe festlegen. Wenn der Parameter **-full** beim Ausführen des Skripts angegeben ist, geben Sie alle Eigenschaften in der Variablen *$property* aus. Wenn Sie keine vollständige Ausgabe benötigen, geben Sie nur die Eigenschaft *Description* des Gerätetreibers aus. Hierzu ist folgender Code erforderlich:

```
if($full)
 {
  format-list -InputObject $wmi -property `
      $property
 }
ELSE
 {
  format-table -inputobject $wmi -Property description
 }
```

Das vollständige Skript *CheckSignedDeviceDrivers.ps1* hat folgenden Aufbau:

CheckSignedDeviceDrivers.ps1

```
param(
      $computer="localhost",
     [switch]$unsigned,
     [switch]$full,
     [switch]$help
    )

function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline += "=" }
    Write-Host -ForegroundColor green $strIN
    Write-Host -ForegroundColor darkgreen $funline
}
```

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: CheckSignedDeviceDrivers.ps1
Zeigt eine Liste der Programme an,
die entweder signiert oder nicht signiert wurden.

PARAMETERS:
-computer    Der Name des Computers.
-unsigned    Listet die nicht signierten Treiber auf.
-full        Listet die Eigenschaften Description, driverProviderName,
             Driverversion,DriverDate und infName auf.
-help        Zeigt dieses Hilfethema an.

SYNTAX:
CheckSignedDeviceDrivers.ps1 -computer munich -unsigned

Zeigt eine Liste aller nicht signierten Treiber
auf einem Computer namens munich an.

CheckSignedDeviceDrivers.ps1 -unsigned -full

Zeigt eine Liste aller nicht signierten Treiber auf dem lokalen
Computer an und führt in dieser Liste die Treibereigenschaften
Description, driverProviderName, Driverversion,DriverDate und infName auf.

CheckSignedDeviceDrivers.ps1 -computer munich -full

Zeigt eine Liste aller signierten Treiber auf einem Computer namens munich an
und führt in dieser Liste die Treibereigenschaften
Description, driverProviderName, Driverversion,DriverDate und infName auf.

CheckSignedDeviceDrivers.ps1 -help ?

 Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }

if($unsigned)
  { $filter = "isSigned = 'false'" ; $mode = "unsigned" }
ELSE
  { $filter = "isSigned = 'true'" ; $mode = "signed" }

$property = "Description", "driverProviderName", `
            "Driverversion","DriverDate","infName"

$wmi = Get-WmiObject -Class Win32_PnPSignedDriver `
     -computername $computer -property $property -filter $filter

funline("Es gibt $($wmi.count) $mode Treiber, siehe folgende Liste:")
```

```
if($full)

 {
  format-list -InputObject $wmi -property `
      $property
 }
ELSE
 {
  format-table -inputobject $wmi -Property description
 }
```

Beheben von Netzwerkproblemen

Das Beheben von Netzwerkproblemen in Windows Server 2008 und Windows Vista ist aufgrund der zahlreichen Elemente, die das Betriebssystem als Netzwerkadapter behandelt, schwierig. Um die Problembehandlung zu erleichtern, können Sie jedoch das Skript *GetActiveNicAndConfig.ps1* verwenden.

Beginnen Sie das Skript mit einer *param*-Anweisung, legen Sie aber nicht wie in den vorherigen Skripts den Standardwert auf *localhost* fest, sondern überprüfen Sie den Wert der Umgebungsvariablen %COMPUTERNAME% und verwenden Sie diesen Namen zum Festlegen des Standardwerts der Variablen *$computer*. Der Rest der *param*-Anweisung ist den anderen Skripts ähnlich. Definieren Sie die Parameter **-help** und **-full** wie gehabt. Codezeile:

```
param($computer = $env:computername, [switch]$full, [switch]$help)
```

Definieren Sie die Funktion *funline* wie im Skript *FunLine3.ps1* im Ordner *Extras* auf der Begleit-CD demonstriert. Der Unterschied zwischen dieser *funline*-Funktion und den vorherigen Funktionen besteht darin, dass Sie die Länge der Eingabezeichenfolge zum Multiplizieren des Werts für die Zeilentrennung verwenden. Speichern Sie die Ergebnisse in der Variablen *$strLine* und geben Sie die Zeichenfolge sowie den Unterstrichwert aus. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
function funline ($strIN)
{
 $strLine= "=" * $strIn.length
 Write-Host -ForegroundColor yellow $strIN
 Write-Host -ForegroundColor darkYellow $strLine
}
```

Erstellen Sie als Nächstes die Funktion *funhelp*. Die Funktion *funhelp* bietet keine Überraschungen: Erstellen Sie eine *here*-Zeichenfolge für den Hilfetext und weisen Sie diese der Variablen *$helptext* zu. Geben Sie den Wert aus und beenden Sie das Skript. Codebeispiel:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GetActiveNicAndConfig.ps1
Zeigt die Konfiguration der aktiven Netzwerkkarten an.

PARAMETER:
-computer    Der Name des Computers.
-full        Zeigt alle Informationen an.
-help        Zeigt dieses Hilfethema an.

SYNTAX:
```

```
GetActiveNicAndConfig.ps1 -computer munich

Zeigt die Konfiguration der aktiven Netzwerkkarten
auf einem Computer  namens munich an.

GetActiveNicAndConfig.ps1

Zeigt die Konfiguration der aktiven Netzwerkkarten
auf dem lokalen Computer an.

GetActiveNicAndConfig.ps1 -computer munich -full

Zeigt sämtliche Konfigurationsinformationen für die aktiven Netzwerkkarten
auf einem Computer  namens munich an.

GetActiveNicAndConfig.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Stellen Sie sicher, dass die Variable *$help* vorhanden ist. Wenn die Variable existiert, rufen Sie die Funktion *funhelp* mit folgender Codezeile auf:

```
if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
```

Erstellen Sie eine Konstante mit dem Codewert für Netzwerkadapter, die mit dem Netzwerk verbunden sind. Dieser Wert ist im Windows Software Development Kit (SDK) dokumentiert. Codezeile:

```
New-Variable -Name c_netConnected -value 2 -option constant
```

Sie müssen nun die Verbindung mit WMI herstellen. Verwenden Sie hierzu das Cmdlet **Get-WmiObject** und wählen Sie die WMI-Klasse *Win32_NetworkAdapter* aus. Stellen Sie eine Verbindung mit dem in der Variablen *$computer* angegebenen Computer her und suchen Sie nur nach den derzeit verbundenen Netzwerkkarten. Nachdem Sie die Netzwerkkarten ermittelt haben, weisen Sie das resultierende *Management*-Objekt der Variablen *$nic* zu. Codezeile:

```
$nic = Get-WmiObject -Class win32_networkadapter -computername $computer `
          -filter "NetConnectionStatus  = $c_netConnected"
```

Verwenden Sie das *NetworkAdapter*-Objekt in der Variablen *$nic*, um nach einem zugehörigen *NetworkAdapterConfiguration*-Objekt zu suchen. Verwenden Sie die Eigenschaft *$nic.InterfaceIndex*, da diese auch in der WMI-Klasse *Win32_NetworkAdapterConfiguration* verfügbar ist. Speichern Sie das resultierende *Management*-Objekt in der Variablen *$nicConfig*. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
$nicConfig = Get-WmiObject -Class win32_networkadapterconfiguration `
                -filter "interfaceindex = $($nic.interfaceindex)"
```

Sie müssen nun bestimmen, welche Informationen zurückgegeben werden sollen. Wenn Sie den Parameter **-full** beim Aufrufen des Skripts angeben, sollen alle Informationen sowohl zur Netzwerkkarte als auch zur Netzwerkkartenkonfiguration ausgegeben werden. Mit der Funktion *funline* können Sie einen Header zwischen den jeweiligen Ausgabeabschnitten einfügen. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
if($full)
  {
   funline("Vollständige Informationen für die Netzwerkkarten auf $($computer)")
   format-list -InputObject $nic -property [a-z]*
   funline("Vollständige Informationen für die Konfiguration der Netzwerkkarten auf $($computer)")
   format-list -InputObject $nicConfig -property [a-z]*
  }
```

Wenn der Parameter **-full** nicht existiert, werden nur die Standardwerte der WMI-Klassen angezeigt. Geben Sie die Informationen mit dem Cmdlet **Format-List** aus. Verwenden Sie hierzu den Parameter **-inputobject**, da die Objekte, die die WMI-Informationen bereitstellen, bereits vom Skript abgerufen wurden. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
ELSE
{
   funline("Standardinformationen für die Netzwerkkarten auf $($computer)")
   format-list -InputObject $nic
   funline("Standardinformationen für die Konfiguration der Netzwerkkarten auf $($computer)")
   format-list -InputObject $nicConfig
  }
```

Das vollständige Skript *GetActiveNicAndConfig.ps1* hat folgenden Inhalt:

GetActiveNicAndConfig.ps1

```
param($computer = $env:computername, [switch]$full, [switch]$help)

function funline ($strIN)
{
 $strLine= "=" * $strIn.length
 Write-Host -ForegroundColor yellow $strIN
 Write-Host -ForegroundColor darkYellow $strLine
}

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GetActiveNicAndConfig.ps1
Zeigt die Konfiguration der aktiven Netzwerkkarten an.

PARAMETER:
-computer    Der Name des Computers.
-full        Zeigt alle Informationen an.
-help        Zeigt dieses Hilfethema an.

SYNTAX:
GetActiveNicAndConfig.ps1 -computer munich

Zeigt die Konfiguration der aktiven Netzwerkkarten
auf einem Computer  namens munich an.

GetActiveNicAndConfig.ps1

Zeigt die Konfiguration der aktiven Netzwerkkarten
auf dem lokalen Computer an.

GetActiveNicAndConfig.ps1 -computer munich -full
```

```
Zeigt sämtliche Konfigurationsinformationen für die aktiven Netzwerkkarten
auf einem Computer  namens munich an.

GetActiveNicAndConfig.ps1 -help ?

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }

New-Variable -Name c_netConnected -value 2 -option constant
$nic = Get-WmiObject -Class win32_networkadapter -computername $computer `
          -filter "NetConnectionStatus  = $c_netConnected"
$nicConfig = Get-WmiObject -Class win32_networkadapterconfiguration `
                -filter "interfaceindex = $($nic.interfaceindex)"

if($full)
  {
   funline("Vollständige Informationen für die Netzwerkkarten auf $($computer)")
   format-list -InputObject $nic -property [a-z]*
   funline("Vollständige Informationen für die Konfiguration der Netzwerkkarten auf $($computer)")
   format-list -InputObject $nicConfig -property [a-z]*
  }
ELSE
{
   funline("Vollständige Informationen für die Netzwerkkarten auf $($computer)")
   format-list -InputObject $nic
   funline("Vollständige Informationen für die Konfiguration der Netzwerkkarten auf $($computer)")
   format-list -InputObject $nicConfig
  }
```

Zusammenfassung

In diesem Kapitel wurden mehrere Bereiche für die Problembehandlung in Windows Vista und Windows Server 2008 erklärt. Das erste Thema bezog sich auf die Startkonfigurationseinstellungen. Sie haben mit der WMI die Start- und Arbeitsverzeichnisse der aktuellen Windows-Installation ermittelt. Außerdem haben Sie die Startdienste überprüft, insbesondere die für den automatischen Start konfigurierten Dienste, die nicht ausgeführt werden.

Darüber hinaus wurden die Dienstabhängigkeiten erklärt. Das Verstehen der Abhängigkeiten ist wichtig, da ein abhängiger Dienst möglicherweise nicht mehr ausgeführt werden kann, wenn sein übergeordneter Dienst ausfällt.

Als Nächstes haben Sie nach nicht signierten Gerätetreibern gesucht. Das Kapitel wurde mit einer Erklärung der Konfigurationsinformationen der aktiven Netzwerkkarten (Network Interface Card, NIC) abgeschlossen. Außerdem wurde das Multiplizieren von Zeichenfolgen und der direkte Zugriff auf die *PSDrive*-Umgebung beschrieben. Sie haben sich mit einer Methode zum Erstellen kurzer und vollständiger Listen mit Verwaltungsinformationen vertraut gemacht. Die Ausgabe wurde mit einem festen Parameter über die Befehlszeile gesteuert.

KAPITEL 13

Verwalten von Domänenbenutzern

Nach Abschluss dieses Kapitels können Sie:

- Organisationseinheiten erstellen
- Domänenbenutzer und Gruppen erstellen
- Domänenbenutzer und Gruppen ändern
- Mehrere Benutzer mit mehreren Attributen hinzufügen

Auf der Begleit-CD Alle Skripts in diesem Kapitel befinden sich auf der Begleit-CD im Ordner *\Scripts\Chapter13*.

Erstellen von Organisationseinheiten

Ein Hauptelement von Active Directory ist die Organisationseinheit (Organizational Unit, OU). Benutzerkonten müssen sich in einer Organisationseinheit befinden, bevor eine Gruppenrichtlinie angewendet werden kann. Gruppenrichtlinien erleichtern die zahlreichen Aufgaben eines Netzwerkadministrators. Sie können auch Druckerobjekte, Dateifreigaben, Computerkonten, Gruppen und viele andere Objekte in Organisationseinheiten gruppieren. Eine Organisationseinheit kann außerdem untergeordnete Organisationseinheiten umfassen.

Natürlich können Sie Organisationseinheiten auch mit einem Skript erstellen. Das Verfahren zum Erstellen einer Organisationseinheit in Active Directory entspricht im Wesentlichen dem Verfahren zum Erstellen von Benutzern, Gruppen und anderen Objekten. Sie stellen eine Verbindung mit Active Directory her, wählen den zu erstellenden Objekttyp aus und geben den Objektnamen an. Der Unterschied zwischen dem Erstellen einer Organisationseinheit und anderen Erstellungsverfahren besteht lediglich im erstellten Objektyp und den Namen der konfigurierten Attribute.

Beginnen Sie das Skript *CreateOU.ps1* mit einer *param*-Anweisung, um den Argumenten Werte zuzuweisen, die zum Erstellen der Organisationseinheit dienen. Sie benötigen den Namen, den Pfad und die Domäne. Der Parameter **-ou** ist optional, wenn Sie eine Organisationseinheit auf höchster Ebene erstellen. Berücksichtigen Sie außerdem den festen Parameter **-help**, um, falls erforderlich, Hilfeinformationen anzuzeigen. Codezeile:

```
param($name,$ou,$dc,[switch]$help)
```

Erstellen Sie als Nächstes die Funktion *funhelp()*, um die Hilfemeldung mit Informationen über das Skript, die Parameter und Syntax zu implementieren. Definieren Sie die Hilfeinformationen in einer *here*-Zeichenfolge. Der größte Vorteil von *here*-Zeichenfolgen ist, dass Sie die Regeln für Anführungszeichen ignorieren können. Nachdem Sie die *here*-Zeichenfolge erstellt und der Variablen *$helptext* zugewiesen haben, geben Sie den Inhalt der Variablen aus und beenden die Skriptausführung. Die Funktion *funhelp()* ist wie folgt implementiert:

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: CreateOU.ps1
Erstellt eine Organisationseinheit.

PARAMETER:
-name       Der Name der zu erstellenden Organisationseinheit.
-ou         Die übergeordnete OU  der zu erstellenden Organisationseinheit.
-dc         Die Domäne der zu erstellenden Organisationseinheit.
-help       Zeigt dieses Hilfethema an.

SYNTAX:
CreateOU.ps1 -name "OU=MyNewOU" -ou "myOU" `
               -dc "dc=nwtraders,dc=com"

Erstellt eine Organisationseinheit namens MyNewOU in der
Organisationseinheit myOU in der Domäne nwtraders.com.

CreateOU.ps1 -name "ou=mynewou" -dc "dc=nwtraders,dc=com"

Erstellt eine Organisationseinheit namens MyNewOU auf
oberster Ebene in der Domäne nwtraders.com.

CreateOU.ps1 -help

 Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Sie müssen bestimmen, ob der Hilfetext beim Ausführen des Skripts angezeigt werden soll. Wenn die Variable *$help* vorhanden ist, geben Sie eine entsprechende Meldung aus und rufen die Funktion *funhelp()* auf. Codezeile:

```
if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
```

Stellen Sie sicher, dass mehrere erforderliche Parameter vorhanden sind, um Fehler während der Skriptausführung zu vermeiden. Da Objekte ohne Namen in Active Directory nicht erstellt werden können, muss die Variable *$name* vorhanden sein. Das neue Active Directory-Objekt muss auch irgendwo angelegt werden. Die Variable *$dc* verweist auf den vollständigen Pfad zur Domäne, in der die Organisationseinheit erstellt wird. Der entsprechende Parameter ist deshalb ebenfalls erforderlich. Wenn die Variable *$name* oder *$dc* nicht existiert, geben Sie eine Meldung aus, dass ein erforderlicher Parameter nicht vorhanden ist, und rufen Sie die Funktion *funhelp()* auf. Codezeile:

```
if(!$name -or !$dc) { "Erforderlicher Parameter fehlt ..." ; funhelp }
```

Wenn die Variable *$ou* vorhanden ist, können Sie diese für den Active Directory-Verbindungsaufbau verwenden. Sollte die Variable *$ou* nicht vorhanden sein, beziehen Sie diese nicht in *adsPath* ein. Geben Sie auf jeden Fall den Namen und den Pfad zur neuen Organisationseinheit an, die Sie erstellen möchten. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
if($ou)
 {  "Erstellen der OU $name im Pfad LDAP://$ou,$dc"
    $ADSI = [ADSI]"LDAP://$ou,$dc"
 }
ELSE
 { "Erstellen der OU $name im Pfad LDAP://$dc"
    $ADSI = [ADSI]"LDAP://$dc"
 }
```

Sie müssen den Typ des zu erstellenden Objekts festlegen. Da Sie eine Organisationseinheit erstellen möchten, verwenden Sie für die Active Directory-Dienstschnittstellen (Active Directory Services Interfaces, ADSI) die Klasse *OrganizationalUnit*. Speichern Sie diesen Wert in der Variablen *$class*, rufen Sie die Methode *Create()* auf und übergeben Sie die Variablen *$class* und *$name* an diese Methode. Das von der Methode zurückgegebene Objekt wird der Variablen *$OrganizationalUnit* zugewiesen. Verwenden Sie die Methode *Setinfo()* dieses Objekts, um die Änderungen in Active Directory zu speichern. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
$CLass = "OrganizationalUnit"
$OrganizationalUnit = $ADSI.create($Class, $Name)
$OrganizationalUnit.setInfo()
```

Das vollständige Skript *CreateOU.ps1* hat folgenden Aufbau:

CreateOU.ps1

```
param($name,$ou,$dc,[switch]$help)
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: CreateOU.ps1
Erstellt eine Organisationseinheit.

PARAMETER:
-name        Der Name der zu erstellenden Organisationseinheit.
-ou          Die übergeordnete OU  der zu erstellenden Organisationseinheit.
-dc          Die Domäne der zu erstellenden Organisationseinheit.
-help        Zeigt dieses Hilfethema an.

SYNTAX:
CreateOU.ps1 -name "OU=MyNewOU" -ou "myOU" `
             -dc "dc=nwtraders,dc=com"

Erstellt eine Organisationseinheit namens MyNewOU in der
Organisationseinheit myOU in der Domäne nwtraders.com.

CreateOU.ps1 -name "ou=mynewou" -dc "dc=nwtraders,dc=com"

Erstellt eine Organisationseinheit namens MyNewOU auf
oberster Ebene in der Domäne nwtraders.com.

CreateOU.ps1 -help

 Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
```

```
exit
}

if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
if(!$name -or !$dc) { "Erforderlicher Parameter fehlt ..." ; funhelp }
if($ou)
 {  "Erstellen der OU $name im Pfad LDAP://$ou,$dc"
    $ADSI = [ADSI]"LDAP://$ou,$dc"
 }
ELSE
 { "Erstellen der OU $name im Pfad LDAP://$dc"
    $ADSI = [ADSI]"LDAP://$dc"
 }

$Class = "OrganizationalUnit"
$OrganizationalUnit = $ADSI.create($CLass, $Name)
$OrganizationalUnit.setInfo()
```

Erstellen von Domänenbenutzern

Das Erstellen von Benutzern ist eine grundlegende Aufgabe der Netzwerkverwaltung. In Windows Server 2008 können Sie ein Benutzerobjekt in Active Directory erstellen, indem Sie die Methode *Create()* verwenden und einen Namen angeben, ohne Werte für die Attribute festzulegen. Wenn Sie ein Benutzerobjekt mit dieser Methode erstellen, wird das Benutzerkonto deaktiviert und viele Attribute erhalten zufällige Werte. Diese Methode eignet sich zum Erstellen zahlreicher Benutzer, beispielsweise für Testzwecke in einer Laborumgebung. Sie können die Benutzer in nur einem Schritt erstellen und die Informationen zu einem späteren Zeitpunkt eingeben.

Geben Sie im Skript *CreateUser.ps1* die *param*-Anweisung an und deklarieren Sie vier Parameter. Der Parameter **-help** hat einen fest Wert und muss beim Ausführen des Skripts nicht angegeben werden. Dieser Parameter wird angegeben, um die Hilfe anzuzeigen. Codezeile:

```
param($name,$ou,$dc,[switch]$help)
```

Definieren Sie die Funktion *funhelp()*, um eine Hilfemeldung anzuzeigen, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Die Funktion *funhelp( )* verwendet eine *here*-Zeichenfolge zur Definition des Hilfetextes, die der Variablen *$helpText* zugewiesen ist. Sie können den Hilfetext ohne Anführungszeichen und das **`t** für Tabzeichen in die *here*-Zeichenfolge eingeben. Diese Methode vereinfacht die Texteingabe. Die Funktion *funhelp()* zeigt eine Beschreibung, die Parameter und die Syntax des Skripts an. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: CreateUser.Ps1
Erstellt ein Benutzerkonto.

PARAMETER:
-name        Der Name des zu erstellenden Benutzerkontos.
-ou          Die übergeordnete OU des zu erstellenden Benutzerkontos.
-dc          Die Domäne des zu erstellenden Benutzerkontos.
-help        Zeigt dieses Hilfethema an.

SYNTAX:
```

```
CreateUser.Ps1 -name "CN=MyNewUser" -ou "ou=myOU" `
               -dc "dc=nwtraders,dc=com"

Erstellt ein Benutzerkonto namens MyNewUser in der
Organisationseinheit myOU in der Domäne nwtraders.com.

CreateUser.ps1 -name "cn=myuser" -ou "ou=ou2,ou=mytestou" `
               -dc "dc=nwtraders,dc=com"

Erstellt ein Benutzerkonto namens MyNewUser in der Organisationseinheit ou2.
Diese Organisationseinheit ist der Organisationseinheit mytestou
in der Domäne nwtraders.com untergeordnet.

CreateUser.Ps1 -name "CN=MyNewUser" `
               -dc "dc=nwtraders,dc=com"

Erstellt ein Benutzerkonto namens MyNewUser
im Container Users in der Domäne nwtraders.com.

CreateUser.Ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Überprüfen Sie, ob die Variable *$help* vorhanden ist. Wird die Variable gefunden, geben Sie eine Meldung aus und rufen Sie die Funktion *funhelp()* auf. Codezeile:

```
if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
```

Suchen Sie anschließend nach den Variablen *$name* und *$dc*. Wenn eine dieser Variablen nicht vorhanden ist, hat der Benutzer beim Ausführen des Skripts den entsprechenden Parameter nicht angegeben. Sie können jedoch einen Benutzer nur dann erstellen, wenn dieser einen Namen hat und Sie auf Active Directory zugreifen können. Rufen Sie die Funktion *funhelp()* auf, wenn die erforderlichen Parameter fehlen. Codezeile:

```
if(!$name -or !$dc) { "Erforderlicher Parameter fehlt ..." ; funhelp }
```

Suchen Sie als Nächstes nach der Variablen *$ou*. Wenn die Variable existiert, geben Sie die in der Variablen *$ou* angegebene Organisationseinheit im *adsPath* mit an, um die Verbindung mit Active Directory herzustellen. Sollte die Variable nicht vorhanden sein, stellen Sie die Verbindung mit der Stammdomäne her. Wenn die Organisationseinheit beim Ausführen des Skripts nicht angegeben wird, stellen Sie die Verbindung mit Active Directory her, ohne die Organisationseinheit in *adsPath* anzugeben. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
if($ou)
 {  "Erstellen des Benutzerkontos $name im Pfad LDAP://$ou,$dc"
    $ADSI = [ADSI]"LDAP://$ou,$dc"
 }
ELSE
 { "Erstellen des Benutzerkontos $name im Pfad LDAP://cn=users,$dc"
   $ADSI = [ADSI]"LDAP://cn=users,$dc"
 }
```

Sie müssen den Typ des zu erstellenden Objekts festlegen. Geben Sie im Skript *CreateUser.ps1* als Objekttyp *user* an. Verwenden Sie die Methode *Create()*, um ein Benutzerobjekt mit dem in der Variablen *$name* angegebenen Namen zu erstellen. Rufen Sie anschließend die Methode *SetInfo()* auf. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
$CLass = "User"
$User = $ADSI.create($CLass, $Name)
$User.setInfo()
```

Das vollständige Skript *CreateUser.ps1* hat folgenden Inhalt:

CreateUser.ps1

```
param($name,$ou,$dc,[switch]$help)
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: CreateUser.Ps1
Erstellt ein Benutzerkonto.

PARAMETER:
-name        Der Name des zu erstellenden Benutzerkontos.
-ou          Die übergeordnete OU des zu erstellenden Benutzerkontos.
-dc          Die Domäne des zu erstellenden Benutzerkontos.
-help        Zeigt dieses Hilfethema an.

SYNTAX:
CreateUser.Ps1 -name "CN=MyNewUser" -ou "ou=myOU" `
               -dc "dc=nwtraders,dc=com"

Erstellt ein Benutzerkonto namens MyNewUser in der
Organisationseinheit myOU in der Domäne nwtraders.com.

CreateUser.ps1 -name "cn=myuser" -ou "ou=ou2,ou=mytestou" `
               -dc "dc=nwtraders,dc=com"

Erstellt ein Benutzerkonto namens MyNewUser in der Organisationseinheit ou2.
Diese Organisationseinheit ist der Organisationseinheit mytestou
in der Domäne nwtraders.com untergeordnet.

CreateUser.Ps1 -name "CN=MyNewUser" `
               -dc "dc=nwtraders,dc=com"

Erstellt ein Benutzerkonto namens MyNewUser
im Container Users in der Domäne nwtraders.com.

CreateUser.Ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
```

```
if(!$name -or !$dc) { "Erforderlicher Parameter fehlt ..." ; funhelp }
if($ou)
 {  "Erstellen des Benutzerkontos $name im Pfad LDAP://$ou,$dc"
    $ADSI = [ADSI]"LDAP://$ou,$dc"
 }
ELSE
 { "Erstellen des Benutzerkontos $name im Pfad LDAP://cn=users,$dc"
   $ADSI = [ADSI]"LDAP://cn=users,$dc"
 }

$CLass = "User"
$User = $ADSI.create($CLass, $Name)
$User.setInfo()
```

Ändern der Benutzerattribute

Für Benutzerkonten sind zahlreiche Attribute verfügbar. Sie müssen das entsprechende Attribut identifizieren und den Wert mit der Methode *Put()* aktualisieren. Im Folgenden werden Skripts beschrieben, mit denen Sie die Attribute für Benutzerobjekte wie in Active Directory-Benutzer und -Computer festlegen können.

Bewährte Vorgehensweise Die fünf in diesem Abschnitt erklärten Skripts enthalten hart codierte Werte, mit denen die Attribute in Active Directory festgelegt werden. Tatsächlich werden Sie Skripts dieser Art jedoch nicht verwenden. Diese fünf Skripts veranschaulichen das Zuweisen von Werten zu Attributen. Die schwierigste Aufgabe beim Erstellen von Skripts für ADSI ist das Ermitteln der Attributsnamen. Die beste Methode zum Festlegen der Attribute hängt davon ab, wo sich die Informationen befinden. Die für das Skript erforderlichen Informationen können in einer Textdatei, einer .csv-Datei, einem Microsoft Excel-Arbeitsblatt, einer Microsoft Access-Datenbank, einer Microsoft SQL Server-Datenbank oder in einer anderen Datenquelle gespeichert sein.

Ändern der allgemeinen Benutzerinformationen

Die Registerkarte **Allgemein** in Active Directory-Benutzer und -Computer zeigt die allgemeinen Benutzerinformationen an. Auf dieser Registerkarte sind neun verschiedene Attribute aufgeführt, beispielsweise der Vorname und der Nachname des Benutzers. Das Problem ist, dass die in Active Directory gespeicherten Attribute nicht mit den auf der Registerkarte **Allgemein** in Active Directory-Benutzer und -Computer angezeigten Attributsnamen identisch sind.

In Tabelle 13.1 sind die Attributsnamen auf der Registerkarte und die entsprechenden Namen in Active Directory-Benutzer und -Computer aufgeführt. In Skripts müssen Sie die in der Spalte ADSI aufgeführten Namen verwenden.

Tabelle 13.1 Zuordnung der Attributsnamen auf der Registerkarte Allgemein

Namen auf der Registerkarte Allgemein	ADSI
Vorname	givenName
Initialen	Initials
Nachname	Sn
Anzeigename	DisplayName
Beschreibung	Description

Tabelle 13.1 Zuordnung der Attributsnamen auf der Registerkarte Allgemein *(Fortsetzung)*

Namen auf der Registerkarte Allgemein	ADSI
Büro	physicalDeliveryOfficeName
Telefonnummer	telephoneNumber
E-Mail	Mail
Webseite	wwwHomePage

Im Skript *ModifyGeneralProperties.ps1* weisen Sie allen Attributen auf der Registerkarte **Allgemein** in Active Directory-Benutzer und -Computer Werte zu. Verwenden Sie den *[ADSI]*-Accelerator und geben Sie den *adsPath* zum jeweiligen Benutzerobjekt an, das Sie ändern möchten. *adsPath* besteht aus dem definierten Namensattribut des Benutzerobjekts, dem der Moniker *LDAP://* vorangeht. Der Moniker *LDAP://* benachrichtigt ADSI, dass Sie mit dem LDAP-Anbieter (Lightweight Directory Access Protocol) auf das Verzeichnis zugreifen möchten. Ein Beispiel für die Bindungszeichenfolge ist im Folgenden dargestellt:

```
$objUser = [ADSI]"LDAP://cn=MyNewUser,ou=myTestOU,dc=nwtraders,dc=msft"
```

Weisen Sie den Attributen mit der Methode *Put()* Werte zu. Dabei müssen Sie das gewünschte Attribut mit dem Active Directory-Attributsnamen in runden Klammern angeben. Geben Sie den durch ein Komma getrennten Wert ein, den Sie in Active Directory einfügen möchten. Das Skript enthält für jedes Attribut eine Zeile. Dieser Skriptabschnitt ist im Folgenden dargestellt:

```
$objUser.put("SamaccountName", "myNewUser")
$objUser.put("givenName", "Mein")
$objUser.Put("initials", "N.")
$objUser.Put("sn", "Benutzer")
$objUser.Put("DisplayName", "Mein neuer Benutzer")
$objUser.Put("description" , "Ein einfaches Benutzerkonto.")
$objUser.Put("physicalDeliveryOfficeName", "RQ2")
$objUser.Put("telephoneNumber", "999-222-1111")
$objUser.Put("mail", "mnu@hotmail.com")
$objUser.Put("wwwHomePage", "http://www.mnu.msn.com")
```

Um die Änderungen in Active Directory zu übernehmen, verwenden Sie die Methode *SetInfo()*. Codezeile:

```
$objUser.setInfo()
```

Das vollständige Skript *ModifyGeneralProperties.ps1* lautet:

ModifyGeneralProperties.ps1

```
$objUser = [ADSI]"LDAP://cn=MyNewUser,ou=myTestOU,dc=nwtraders,dc=msft"
$objUser.put("SamaccountName", "myNewUser")
$objUser.put("givenName", "Mein")
$objUser.Put("initials", "N.")
$objUser.Put("sn", "Benutzer")
$objUser.Put("DisplayName", "Mein neuer Benutzer")
$objUser.Put("description" , "Ein einfaches Benutzerkonto.")
$objUser.Put("physicalDeliveryOfficeName", "RQ2")
$objUser.Put("telephoneNumber", "999-222-1111")
$objUser.Put("mail", "mnu@hotmail.com")
$objUser.Put("wwwHomePage", "http://www.mnu.msn.com")
$objUser.setInfo()
```

Bearbeiten der Registerkarte Adresse

Auf der Registerkarte **Adresse** in Active Directory-Benutzer und -Computer werden sechs Attribute für das Benutzerobjekt angezeigt. Die auf dieser Registerkarte angezeigten Namen stimmen ebenfalls nicht mit den in Active Directory gespeicherten Attributsnamen überein. In Tabelle 13.2 sind die Attributsnamen und die entsprechenden Namen in Active Directory-Benutzer und -Computer aufgeführt. In Skripts müssen Sie die in der rechten Spalte aufgeführten Namen verwenden.

Tabelle 13.2 Zuordnung der Attributsnamen auf der Registerkarte Adresse

Namen auf der Registerkarte Adresse	ADSI
Adresse	streetAddress
Postfach	postOfficeBox
Ort	L
Bundesland/Kanton	St
Postleitzahl	postalCode
Land/Region	C,co,countryCode

Das Skript *ModifyAddressProperties.ps1* veranschaulicht, wie Sie die Attribute auf der Registerkarte **Adresse** eines Benutzerobjekts ändern können. Die Skripts *ModifyAddressProperties.ps1* und *ModifyGeneralProperties.ps1* sind identisch. Erstellen Sie zuerst die Objektbindung in Active Directory. Rufen Sie dann die Methode *Put()* auf, geben Sie den Attributsnamen und den Attributswert an und übernehmen Sie mit *SetInfo()* die Änderungen in Active Directory.

Das Skript *ModifyAddressProperties.ps1* lautet:

ModifyAddressProperties.ps1

```
$objUser = [ADSI]"LDAP://cn=MyNewUser,ou=myTestOU,dc=nwtraders,dc=msft"
$objUser.put("streetAddress", "123 main st")
$objUser.put("postOfficeBox", "po box 12")
$objUser.put("l", "Bedrock")
$objUser.put("st", "Arkansas")
$objUser.put("postalCode" , "12345")
$objUser.put("c", "US")
$objUser.put("co", "United States")
$objUser.put("countryCode", "840")
$objUser.setInfo()
```

Bearbeiten der Registerkarte Profil

Die Registerkarte **Profil** zeigt die Informationen über ein Benutzerprofil an. Das Benutzerprofil besteht aus dem Speicherpfad, dem Anmeldeskript, dem Basislaufwerk und dem Basisverzeichnis. In Tabelle 13.3 sind die Eigenschaften auf der Registerkarte **Profil** und die entsprechenden in Active Directory gespeicherten Attribute aufgeführt, die im Skript *ModifyProfileProperties.ps1* verwendet werden. Die Eigenschaftsnamen auf der Registerkarte sind den Attributsnamen in Active Directory zugeordnet.

Tabelle 13.3 Zuordnung der Attributsnamen auf der Registerkarte Profil

Namen auf der Registerkarte Profil	ADSI
Profilpfad	profilePath
Anmeldeskript	scriptPath
Lokaler Pfad	homeDrive
Verbinden mit	homeDirectory

Das Skript *ModifyProfileProperties.ps1* entspricht im Wesentlichem dem Skript *ModifyGeneralProperties.ps1* und ändert die Werte für den Benutzer in Active Directory. Erstellen Sie zuerst die Objektbindung in Active Directory. Rufen Sie dann die Methode Put() auf, geben Sie den Attributsnamen und den Attributswert an und übernehmen Sie mit *SetInfo*() die Änderungen in Active Directory.

Das vollständige Skript *ModifyProfileProperties.ps1* lautet:

ModifyProfileProperties.ps1

```
$objUser = [ADSI]"LDAP://cn=MyNewUser,ou=myTestOU,dc=nwtraders,dc=msft"
$objUser.put("profilePath", "\\London\profiles\myNewUser")
$objUser.put("scriptPath", "logon.vbs")
$objUser.put("homeDirectory", "\\london\users\myNewUser")
$objUser.put("homeDrive", "H:")
$objUser.setInfo()
```

Bearbeiten der Registerkarte Telefon

Auf der Registerkarte **Telefon** können Sie mit Hilfe von Windows PowerShell sechs Optionen festlegen. In Tabelle 13.4 sind die Anzeigenamen in Active Directory-Benutzer und -Computer und die entsprechenden in Active Directory gespeicherten Attributsnamen aufgeführt, die im Skript *ModifyTelephoneProperties.ps1* verwendet werden. Zwei dieser Attribute, *pager* und *mobile*, stimmen genau überein. Andere Attribute, beispielsweise *Fax* und *Hinweise*, haben keine Ähnlichkeit mit den in Active Directory gespeicherten Attributen.

Tabelle 13.4 Zuordnung der Attributsnamen auf der Registerkarte Telefon

Namen auf der Registerkarte Telefon	ADSI
Privat	homePhone
Pager	Pager
Mobil	Mobile
Fax	facsimileTelephoneNumber
IP-Telefon	ipPhone
Hinweise	Info

Das Skript *ModifyTelephoneProperties.ps1* legt die Optionen auf der Registerkarte fest und entspricht im Wesentlichen dem Skript *ModifyGeneralProperties.ps1*. Erstellen Sie zuerst die Objektbindung in Active Directory. Rufen Sie die Methode *put()* auf, geben Sie den Attributsnamen und den Attributswert an und übernehmen Sie mit *SetInfo()* die Änderungen in Active Directory.

Das vollständige Skript *ModifyTelephoneProperties.ps1* lautet:

ModifyTelephoneProperties.ps1

```
$objUser = [ADSI]"LDAP://cn=MyNewUser,ou=myTestOU,dc=nwtraders,dc=msft"
$objUser.Put("homePhone", "(215)788-4312")
$objUser.Put("pager", "(215)788-0112")
$objUser.Put("mobile", "(715)654-2341")
$objUser.Put("facsimileTelephoneNumber", "(215)788-3456")
$objUser.Put("ipPhone", "192.168.6.112")
$objUser.Put("info", "Alle Kontaktinformationen sind vertraulich " `
    + "und nur für Geschäftszwecke vorgesehen.")
$objUser.setInfo()
```

Bearbeiten der Registerkarte Organisation

Auf der Registerkarte **Organisation** in Active Directory-Benutzer und -Computer werden fünf Optionen für das Benutzerobjekt angezeigt. Da die Benutzerobjekte verknüpft sind, können die Werte nicht einfach in die Felder eingegeben werden. In Tabelle 13.5 sind die auf der Registerkarte angezeigten Namen und die entsprechenden in Active Directory gespeicherten Attributsnamen aufgeführt.

Tabelle 13.5 Zuordnung der Attributsnamen auf der Registerkarte Organisation

Namen auf der Registerkarte Organisation	ADSI
Titel	Title
Abteilung	Department
Firma	Company
Manager	Manager
Mitarbeiter	DirectReports

Das Skript *ModifyOrganizationProperties.ps1* legt die Optionen auf der Registerkarte fest und entspricht im Wesentlichen dem Skript *ModifyGeneralProperties.ps1*. Erstellen Sie zuerst die Objektbindung in Active Directory. Rufen Sie danach die Methode *Put()* auf, geben Sie den Attributsnamen und den Attributswert an und übernehmen Sie mit *SetInfo*() die Änderungen in Active Directory.

Das vollständige Skript *ModifyOrganizationProperties.ps1* lautet:

ModifyOrganizationProperties.ps1

```
$strDomain = "dc=nwtraders,dc=msft"
$strOU = "ou=myTestOU"
$strUser = "cn=MyNewUser"
$strManager = "cn=myBoss"

$objUser = [ADSI]"LDAP://$strUser,$strOU,$strDomain"
$objUser.put("title", "Abteilungsleiter")
$objUser.put("department", "Vertrieb")
$objUser.put("company", "North Wind Traders")
$objUser.put("manager", "$strManager,$strou,$strDomain")

$objUser.setInfo()
```

Ändern eines bestimmten Benutzerattributs

Es ist nicht sinnvoll ein Skript für die Profilseite und ein weiteres Skript für die Telefonseite in Active Directory-Benutzer und -Computer zu verwenden. Das Skript *ModifyUser.ps1* greift auf Skripts zu, die Befehlszeilenargumente akzeptieren. Der Netzwerkadministrator kann mit diesem Skript alle Attribute für ein Benutzerkonto in jeder Organisationseinheit in einer beliebigen Domäne ändern.

Beginnen Sie das Skript *ModifyUser.ps1* mit einer *param*-Anweisung, um die für das Skript erforderlichen Informationen zusammenzustellen. Sie benötigen den Namen des Objekts, die zu modifizierende Eigenschaft, den Wert für die Eigenschaft und den Pfad zum Benutzerobjekt. Mit dem Parameter **-name** können Sie den zu ändernden Benutzer ermitteln. Die Parameter **-property** und **-value** geben die zu ändernde Eigenschaft an und die Parameter **-ou** und **-dc** ermitteln den Pfad zum Benutzerobjekt. Außerdem wurde der feste Parameter **-help** berücksichtigt. Codezeile:

```
param($name,$property,$value,$ou,$dc,[switch]$help)
```

Definieren Sie im Anschluss an die *param*-Anweisung die Funktion *funhelp()*, um eine Hilfemeldung anzuzeigen, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Diese Funktion erstellt eine lange *here*-Zeichenfolge, speichert diese in der Variablen *$helpText*, gibt den Inhalt der Variablen aus und beendet die Skriptausführung. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ModifyUser.ps1
Modifiziert ein Benutzerkonto.

PARAMETER:
-name        Der Name des zu modifizierenden Benutzerkontos.
-ou          Die übergeordnete OU des zu modifizierenden Benutzerkontos.
-dc          Die Domäne des zu modifizierenden Benutzerkontos.
-property    Der Name des zu modifizierenden Attributs.
-value       Der neue Wert des zu modifizierenden Attributs.
-help        Zeigt dieses Hilfethema an.

SYNTAX:
ModifyUser.ps1 -name "CN=MyNewUser" -ou "ou=myOU" `
               -dc "dc=nwtraders,dc=com" `
               -property "SamaccountName" `
               -value "MyNewUser"

Modifiziert ein Benutzerkonto namens MyNewUser in der
Organisationseinheit myOU in der Domäne nwtraders.com
und weist dem Attribut SamaccountName den Wert
MyNewUser zu.

ModifyUser.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Sie müssen ermitteln, ob das Skript mit dem Parameter **-help** ausgeführt wird. Suchen Sie nach der Variablen *$help*, die nur vorhanden ist, wenn das Skript mit dem Parameter **-help** ausgeführt wurde. Ist die Variable *$help* vorhanden, geben Sie eine Statusmeldung aus und rufen die Funktion *funhelp()* auf:

```
if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
```

Überprüfen Sie anschließend, ob die erforderlichen Parameter vorhanden sind. Verwenden Sie hierzu eine *if*-Anweisung. Wenn die erforderlichen Variablen nicht vorhanden sind, geben Sie die Meldung *"Erforderlicher Parameter fehlt ..."* aus und rufen die Funktion *funhelp()* auf. Die beiden Codezeilen lauten:

```
if(!$name -or !$dc -or !$property -or !$value)
  { "Erforderlicher Parameter fehlt ..." ; funhelp }
```

Geben Sie in der Variablen *$class* an, dass ein Benutzerobjekt erstellt wird, und geben Sie eine entsprechende Meldung aus. Verwenden Sie die Variablen *$name, $ou* und *$dc*, um anzuzeigen, welcher Benutzer geändert werden soll.

Tipp Sie können das Skript *ModifyUser.ps1* erweitern, indem Sie eine Zeile hinzufügen, die den Inhalt der Variablen *$property* und *$value* ausgibt. Diese Meldung teilt dem Benutzer mit, welche Eigenschaft geändert wird. Sie können auch eine Eingabeaufforderung hinzufügen, um die Änderung vom Benutzer bestätigen zu lassen. Verwenden Sie hierzu das Cmdlet **Read-Host** mit dem Parameter **-prompt**.

Die beiden Codezeilen lauten:

```
$Class = "User"
"Modifizieren von $name,$ou,$dc"
```

Verwenden Sie den *[ADSI]*-Accelerator und geben Sie den *adsPath* zu dem Benutzerkonto an, das Sie ändern möchten. Speichern Sie das zurückgegebene Objekt in der Variablen *$ADSI*. Codezeile:

```
$ADSI = [ADSI]"LDAP://$name,$ou,$dc"
```

Verwenden Sie das in der Variablen *$ADSI* gespeicherte Objekt und rufen Sie die Methode *Put()* auf, um den Wert in der Variablen *$value* in die Eigenschaft *$property* einzufügen. Verwenden Sie anschließend die Methode *SetInfo()* des in der Variablen *$ADSI* gespeicherten Objekts. Die beiden Codezeilen lauten:

```
$ADSI.put($property, $value)
$ADSI.setInfo()
```

Das vollständige Skript *ModifyUser.ps1* hat folgenden Inhalt:

ModifyUser.ps1

```
param($name,$property,$value,$ou,$dc,[switch]$help)
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ModifyUser.ps1
Modifiziert ein Benutzerkonto.

PARAMETER:
-name        Der Name des zu modifizierenden Benutzerkontos.
-ou          Die übergeordnete OU des zu modifizierenden Benutzerkontos.
-dc          Die Domäne des zu modifizierenden Benutzerkontos.
-property    Der Name des zu modifizierenden Attributs.
-value       Der neue Wert des zu modifizierenden Attributs.
```

```
-help        Zeigt dieses Hilfethema an.

SYNTAX:
ModifyUser.ps1 -name "CN=MyNewUser" -ou "ou=myOU" `
               -dc "dc=nwtraders,dc=com" `
               -property "SamaccountName" `
               -value "MyNewUser"

Modifiziert ein Benutzerkonto namens MyNewUser in der
Organisationseinheit myOU in der Domäne nwtraders.com
und weist dem Attribut SamaccountName den Wert
MyNewUser zu.

ModifyUser.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
if(!$name -or !$dc -or !$property -or !$value)
  { "Erforderlicher Parameter fehlt ..." ; funhelp }

$CLass = "User"
"Modifying $name,$ou,$dc"
$ADSI = [ADSI]"LDAP://$name,$ou,$dc"
$ADSI.put($property, $value)
$ADSI.setInfo()
```

Erstellen von Benutzern mit einer .csv-Datei

Sie können Benutzer auch unter Verwendung einer .csv-Datei als Quelldatei erstellen. Diese Methode hat mehrere Vorteile. Eine .csv-Datei kann in Windows PowerShell einfach erstellt und geändert werden. In einer Textdatei müssen Sie die entsprechende Zeile finden und möglicherweise in ein Array umwandeln, wenn die Zeile mehrere Elemente enthält. Ein zusätzlicher Vorteil von .csv-Dateien sind die Spaltenheader, mit denen das Skript übersichtlicher gestaltet werden kann. Außerdem muss für eine .csv-Datei keine zusätzliche Software auf dem Computer installiert werden. Sie können .csv-Dateien mit Notepad erstellen und bearbeiten.

Weisen Sie im Skript *CreateAndEnableUser.ps1* Werte für Kontonamen und Kennworte der Sicherheitskontenverwaltung (Security Account Manager, SAM) zu und aktivieren Sie die Benutzerkonten. Die Standard-Domänensicherheitsrichtlinie in Windows Server 2008 untersagt das Erstellen von aktivierten Benutzerkonten ohne Kennwort. Da das Skript im Wesentlichen dem Skript *CreateUser.ps1* entspricht, werden im Folgenden nur die Unterschiede erklärt.

Hinweis Das SAM-Namensattribut *SamAccountMame* ist für die Abwärtskompatibilität vorgesehen. Dies ist etwas irreführend, da dieses Attribut noch von einigen Anwendungen verwendet wird. Microsoft Exchange Server 2007 kann dieses Attribut beispielsweise zum Abrufen von E-Mail-Nachrichten verwenden. Die Benutzer können sich mit diesem Wert an der Domäne anmelden. Das Attribut ist immer vorhanden. Wenn Sie für das Attribut *SamAccountMame* keinen Wert angeben, genieriert Windows Server 2008 automatisch einen Wert, der maximal 15 Zeichen lang sein kann.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Festlegen des Kennworts

Ein in Active Directory erstelltes Benutzerkonto ist standardmäßig deaktiviert. Um das Benutzerkonto zu aktivieren, müssen Sie ein Kennwort festlegen.

Bewährte Vorgehensweise Kennwörter ermöglichen den Zugriff auf Netzwerkressourcen, die der Netzwerkadministrator schützen möchte. Im Skript *CreateAndEnableUser.ps1* ist das Kennwort in der Textdatei hart codiert, was in manchen Situationen bedenklich ist. Sie können das Skript mit dem verschlüsselnden Dateisystem (Encrypting File System, EFS) in Windows Vista und Windows Server 2008 verschlüsseln, das Kennwort in einer separaten mit EFS verschlüsselten Datei speichern oder das Cmdlet **Get-Credential** verwenden.

Weisen Sie im Skript *CreateAndEnableUser.ps1* das Kennwort mit der Methode *Put()* dem Attribut *userPassword* in Active Directory zu. Ermitteln Sie das Kennwort aus der Spalte *Password* in der Datei *EnabledUsers.csv*. Wenn Sie die Spalte aus der .csv-Datei verwenden, müssen Sie verhindern, dass diese in ein Objekt umgewandelt wird. Um dieses Verhalten zu verhindern, verwenden Sie in der *Put*-Anweisung einen Unterausdruck, indem Sie dem *$strUser.Password*-Abschnitt ein Dollarzeichen voranstellen. Schließen Sie in diesem Fall den auszuführenden Code in Klammern ein. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
User.put("userPassword", $($strUser.Password))
```

Aktivieren des Benutzerkontos

Da ein neu erstelltes Benutzerkonto standardmäßig deaktiviert ist, müssen Sie das Konto explizit aktivieren, um es verwenden zu können. In Kapitel 10 haben Sie das Skript *EnableDisableUser.ps1* erstellt und dem Attribut *userflags* den Wert 512 zugewiesen, um das Benutzerkonto zu aktivieren. Dieses Attribut ist in Active Directory nicht vorhanden und nur über den *WinNT*-Anbieter verfügbar. Sie können das Benutzerkonto also nicht mit dem *[ADSI]*-Accelerator aktivieren. Um das ADSI-Attribut *AccountDisabled* zu ändern, müssen Sie das unbearbeitete Objekt direkt bearbeiten. Verwenden Sie hierzu das Windows PowerShell-Objekt *PSBase*. Das Objekt *PSBase* umfasst die Methode *InvokeSet()*. Mit dieser Methode können Sie einen Wert für das Attribut *AccountDisabled* festlegen. Codezeile:

```
$user.psbase.invokeset("AccountDisabled", "False")
```

Das vollständige Skript *CreateAndEnableUser.ps1* hat folgenden Aufbau:

CreateAndEnableUser.ps1

```
param([switch]$help)
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: CreateAndEnableUser.Ps1
Erstellt aktivierte Benutzerkonten anhand einer .csv-Datei.

PARAMETER:

-help        Zeigt dieses Hilfethema an.

SYNTAX:
CreateAndEnableUser.Ps1

Erstellt aktivierte Benutzerkonten anhand einer .csv-Datei.
```

```
CreateAndEnableUser.Ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }

$aryUser= import-csv -Path C:\PSBOOK\Enabledusers.csv
$Class = "User"
$dc = "dc=nwtraders,dc=com"

foreach($strUser in $aryUser)
{
 $ou = "ou="+$strUser.OU
 $ADSI = [ADSI]"LDAP://$ou,$dc"
 $cnuser="cn="+$($strUser.userName)
 $User = $ADSI.create($CLass,$cnuser)
 $User.put("SamaccountName", $($strUser.username))
 $User.setInfo()
 $User.put("userPassword", $($strUser.Password))
 $user.psbase.invokeset("AccountDisabled", "False")
 $User.setInfo()
}
```

Erstellen von Domänengruppen

Nachdem Sie Benutzer erstellt haben, können Sie mit dem Erstellen von Gruppen für diese Benutzer fortfahren. Das Skript *CreateGroup.ps1* ist dem Skript *CreateUser.ps1* zum Erstellen von Domänenbenutzern ähnlich.

Beginnen Sie das Skript *CreateGroup.ps1* mit einer *param()*-Anweisung. Erstellen Sie den Parameter **-help** sowie drei weitere Parameter, um Gruppen zu erstellen. Codezeile:

```
param($name,$ou,$dc,[switch]$help)
```

Erstellen Sie die Funktion *funhelp()*, um die Hilfe anzuzeigen. Die *here*-Zeichenfolge enthält die Beschreibung, Parameter und Syntax, die angezeigt werden, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: CreateGroup.ps1
Erstellt eine Gruppe.

PARAMETER:
-name        Der Name der zu erstellenden Gruppe.
-ou          Die übergeordnete OU der zu erstellenden Gruppe.
-dc          Die Domäne der zu erstellenden Gruppe.
-help        Zeigt dieses Hilfethema an.
```

```
SYNTAX:
CreateGroup.ps1 -name "CN=MyNewGroup" -ou "myOU" `
               -dc "dc=nwtraders,dc=com"

Erstellt eine Gruppe namens MyNewGroup in der
Organisationseinheit myOU in der Domäne nwtraders.com.

CreateGroup.ps1 -name "CN=MyNewGroup" `
               -dc "dc=nwtraders,dc=com"

Erstellt eine Gruppe namens MyNewGroup
im Container Users in der Domäne nwtraders.com.

CreateGroup.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Wenn die Variable *$help* vorhanden ist, wurde das Skript mit dem Parameter **-help** ausgeführt. Sind die Variablen *$name* und *$dc* nicht vorhanden, sollten Sie ebenfalls eine Meldung ausgeben und die Funktion *funhelp* aufrufen. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
if(!$name -or !$dc) { "Erforderlicher Parameter fehlt ..." ; funhelp }
```

Sie können eine Gruppe auch in einer Organisationseinheit oder im Stammverzeichnis der Domäne erstellen. Legen Sie in diesem Fall fest, dass der Parameter **-ou** optional ist. Diese Flexibilität erhöht jedoch die Komplexität des Skripts, da der Parameter *adsPath* den Wert *Null* oder einen leeren Parameter nicht verarbeiten kann und deshalb zwei separate Verbindungszeichenfolgen erfordert. Wenn die Variable *$ou* vorhanden ist, geben Sie eine Statusmeldung aus und greifen Sie mit dem für die Organisationseinheit angegebenen Wert auf Active Directory zu. Wenn die Variable nicht vorhanden ist, stellen Sie eine andere Verbindung her. Dieser Skriptabschnitt ist im Folgenden dargestellt:

```
if($ou)
 {  "Erstellen der Gruppe $name im Pfad LDAP://$ou,$dc"
  $ADSI = [ADSI]"LDAP://$ou,$dc"
 }
ELSE
 { "Erstellen der Gruppe $name im Pfad LDAP://cn=users,$dc"
  $ADSI = [ADSI]"LDAP://cn=users,$dc"
 }
```

Das restliche Skript ist unkompliziert. Geben Sie die zu erstellende Objektklasse an, rufen Sie die Methode *Create()* auf und übernehmen Sie die Änderungen mit der Methode *SetInfo()* in Active Directory. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
$Class = "Group"
$Group = $ADSI.create($Class, $Name)
$Group.setInfo()
```

Das vollständige Skript *CreateGroup.ps1* lautet:

CreateGroup.ps1

```
param($name,$ou,$dc,[switch]$help)
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: CreateGroup.ps1
Erstellt eine Gruppe.

PARAMETER:
-name        Der Name der zu erstellenden Gruppe
-ou          Die übergeordnete OU der zu erstellenden Gruppe.
-dc          Die Domäne der zu erstellenden Gruppe.
-help        Zeigt dieses Hilfethema an.

SYNTAX:
CreateGroup.ps1 -name "CN=MyNewGroup" -ou "myOU" `
                -dc "dc=nwtraders,dc=com"

Erstellt eine Gruppe namens MyNewGroup in der
Organisationseinheit myOU in der Domäne nwtraders.com.

CreateGroup.ps1 -name "CN=MyNewGroup" `
                -dc "dc=nwtraders,dc=com"

Erstellt eine Gruppe namens MyNewGroup
im Container Users in der Domäne nwtraders.com.

CreateGroup.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
if(!$name -or !$dc) { "Erforderlicher Parameter fehlt ..." ; funhelp }
if($ou)
 {  "Erstellen der Gruppe $name im Pfad LDAP://$ou,$dc"
  $ADSI = [ADSI]"LDAP://$ou,$dc"
 }
ELSE
 { "Erstellen der Gruppe $name im Pfad LDAP://cn=users,$dc"
  $ADSI = [ADSI]"LDAP://cn=users,$dc"
 }

$Class = "Group"
$Group = $ADSI.create($Class, $Name)
$Group.setInfo()
```

Hinzufügen eines Benutzers zu einer Domänengruppe

Gruppen sind nicht sehr interessant, da Sie nur die Mitglieder ändern können. In diesem Abschnitt sind die Schritte zum Hinzufügen von Domänenbenutzern zu Domänengruppen erklärt.

Das Verfahren zum Hinzufügen eines Benutzers ist etwas kompliziert. Obwohl Gruppen ein *member*-Attribut haben, ist es nicht einfach, auf die Gruppe zuzugreifen, das DN-Namensattribut (Distinguished Name) des Benutzers zum Attribut *member* hinzufügen, die Methode *SetInfo()* aufzurufen und den Vorgang abzuschließen. Dieser Prozess erfordert, dass Sie als Erstes das Gruppenobjekt ermitteln. Fügen Sie anschließend mit der Methode *Add()* den *adsPath* des Benutzers zum Attribut *member* hinzu. Rufen Sie *SetInfo()* jedoch nicht auf.

Beginnen Sie das Skript *AddUserToGroup.ps1* mit einer *param()*-Anweisung, um den Namen des Benutzers und der Gruppe sowie die Domäne anzugeben, in der diese Objekte gespeichert sind. Der Parameter **-ou** ist für Active Directory optional, aber für das Skript erforderlich. Der Parameter verhindert einen ADSI-Fehler, der durch einen fehlenden Parameter verursacht wird. Die *param*-Anweisung lautet:

```
param($name,$group,$ou,$dc,[switch]$help)
```

Die nächste Funktion ist *funhelp()*. Der folgende Code erstellt eine lange *here*-Zeichenfolge, die der Variablen *$helpText* zugewiesen wird:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: AddUserToGroup.ps1
Fügt ein Benutzerkonto zu einer Gruppe hinzu.

PARAMETER:
-name        Der Name des Benutzerkontos.
-ou          Die Organisationseinheit der Gruppe.
-dc          Die Domäne des Benutzers.
-group       Die zu modifizierende Gruppe.
-help        Zeigt dieses Hilfethema an.

SYNTAX:
AddUserToGroup.ps1 -name "cn=MyNewUser" -ou "ou=myOU" `
               -dc "dc=nwtraders,dc=com" `
               -group "cn=MyGroup"

Fügt das Benutzerkonto namens MyNewUser aus der Organisationseinheit myOU
in der Domäne nwtraders.com zur Gruppe MyGroup group in der gleichen OU hinzu.

AddUserToGroup.ps1 -help

 Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Verwenden Sie nach der Funktion *funhelp()* eine *if*-Anweisung, um zu überprüfen, ob das Skript mit dem Parameter **-help** ausgeführt wird. Überprüfen Sie außerdem, ob alle erforderlichen Parameter angegeben sind. Die Codezeilen lauten:

```
if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
if(!$name -or !$dc -or !$group -or !$ou)
  { "Erforderlicher Parameter fehlt ..." ; funhelp }
```

Geben Sie als Nächstes eine Statusmeldung aus. Greifen Sie auf die Gruppe zu und verwenden Sie die Methode *Add()*, um den Benutzer zur Gruppe hinzuzufügen. Sie müssen hierzu den *adsPath* des Benutzerkontos verwenden, den Sie aus dem Attribut *distinguishedName* und dem Moniker LDAP:// generieren können. Der folgende Codeabschnitt veranschaulicht diese Vorgehensweise:

```
"Modifying $name,$ou,$dc"
$ADSI = [ADSI]"LDAP://$group,$ou,$dc"
$ADSI.add("LDAP://$name,$ou,$dc")
```

Das vollständige Skript *AddUserToGroup.ps1* hat folgenden Inhalt:

AddUserToGroup.ps1

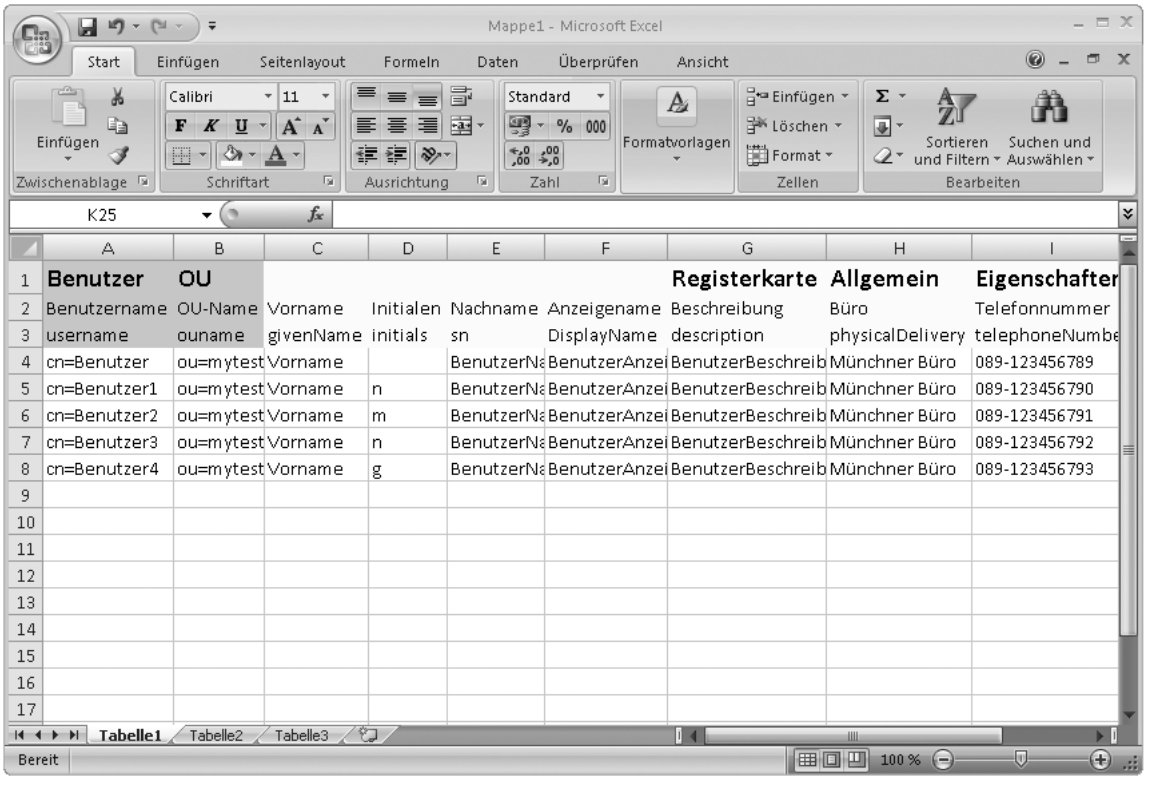
```
param($name,$group,$ou,$dc,[switch]$help)
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: AddUserToGroup.ps1
Fügt ein Benutzerkonto zu einer Gruppe hinzu.

PARAMETER:
-name        Der Name des Benutzerkontos.
-ou          Die Organisationseinheit der Gruppe.
-dc          Die Domäne des Benutzers.
-group       Die zu modifizierende Gruppe.
-help        Zeigt dieses Hilfethema an.

SYNTAX:
AddUserToGroup.ps1 -name "cn=MyNewUser" -ou "ou=myOU" `
               -dc "dc=nwtraders,dc=com" `
               -group "cn=MyGroup"

Fügt das Benutzerkonto namens MyNewUser aus der Organisationseinheit myOU
in der Domäne nwtraders.com zur Gruppe MyGroup group in der gleichen OU hinzu.

AddUserToGroup.ps1 -help

 Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help){ "Ausgeben der Hilfeinformationen ..." ; funhelp }
if(!$name -or !$dc -or !$group -or !$ou)
  { "Erforderlicher Parameter fehlt ..." ; funhelp }
```

```
$CLass = "User"
"Modifying $name,$ou,$dc"
$ADSI = [ADSI]"LDAP://$group,$ou,$dc"
$ADSI.add("LDAP://$name,$ou,$dc")
```

Hinzufügen mehrerer Benutzer mit mehreren Attributen

Das Erstellen nur eines Benutzers ist nicht schwierig. Sie können diesen Vorgang mit dem Assistenten in Active Directory-Benutzer und -Computer in weniger als einer Minute ausführen. Wenn Sie jedoch zusätzliche Attribute festlegen möchten, verlängert sich auch die zum Erstellen erforderliche Zeitdauer. Das Erstellen mehrerer Benutzer mit mehreren Attributen kann Tage oder sogar Wochen in Anspruch nehmen. Die Verwendung eines Excel-Arbeitsblatts bietet sich hier als Möglichkeit an, um das Erstellen mehrerer Benutzer und Attribute zu vereinfachen.

Um ein Excel-Arbeitsblatt zu öffnen, müssen Sie den Pfad zum Arbeitsblatt angeben. Erstellen Sie anschließend eine Instanz des COM-Objekts *Excel.Application*, um mit dem Excel-Objektmodell zu arbeiten, was manchmal einfacher ist als die Verwendung von Active X-Datenobjekten (ADO). Nachdem Sie ein *Excel.Application*-Objekt erstellt haben, können Sie das Arbeitsblatt öffnen und die in den Zellen gespeicherten Werte abrufen. Die Zeilen und Spalten sind mit Zahlen referenziert (beispielsweise 1,1 für die oberste linke Zelle). Diese Nummerierung unterscheidet sich von der in Abbildung 13.1 dargestellten Buchstaben/Zahlen-Kombination.



	A	B	C	D	E	F	G	H	I
1	Benutzer	OU					Registerkarte	Allgemein	Eigenschafter
2	Benutzername	OU-Name	Vorname	Initialen	Nachname	Anzeigename	Beschreibung	Büro	Telefonnummer
3	username	ouname	givenName	initials	sn	DisplayName	description	physicalDelivery	telephoneNumbe
4	cn=Benutzer	ou=mytest	Vorname		BenutzerNa	BenutzerAnzei	BenutzerBeschreib	Münchner Büro	089-123456789
5	cn=Benutzer1	ou=mytest	Vorname	n	BenutzerNa	BenutzerAnzei	BenutzerBeschreib	Münchner Büro	089-123456790
6	cn=Benutzer2	ou=mytest	Vorname	m	BenutzerNa	BenutzerAnzei	BenutzerBeschreib	Münchner Büro	089-123456791
7	cn=Benutzer3	ou=mytest	Vorname	n	BenutzerNa	BenutzerAnzei	BenutzerBeschreib	Münchner Büro	089-123456792
8	cn=Benutzer4	ou=mytest	Vorname	g	BenutzerNa	BenutzerAnzei	BenutzerBeschreib	Münchner Büro	089-123456793
9									
10									
11									
12									
13									
14									
15									
16									
17									

Abbildung 13.1 Ein Excel-Arbeitsblatt zum Verwalten mehrerer Benutzer

Geben Sie im Skript *ReadExcelModifyUsers.ps1* als Erstes den Pfad zum Excel-Arbeitsblatt an:

```
$strPath="C:\Chapter13\NewUser.xls"
```

Erstellen Sie anschließend mit dem Cmdlet **New-Object** und dem Parameter **-comobject** eine Instanz des COM-Objekts *Excel.Application*. Weisen Sie das erstellte Objekt der Variablen *$objExcel* zu:

```
$objExcel=New-Object -ComObject Excel.Application
```

Blenden Sie das Arbeitsblatt mit folgender Codezeile aus, um die Skriptausführung zu beschleunigen und weniger Speicher zu belegen.

```
$objExcel.Visible=$false
```

Problembehandlung Wenn Sie das Excel-Objektmodell verwenden und die Sichtbarkeit auf *$false* festgelegt ist, kann sich das Erkennen von Fehlern schwierig gestalten. Legen Sie die Sichtbarkeit auf *$true* fest, um die Problembehandlung zu vereinfachen. Außerdem sollten Sie im Windows Task-Manager überprüfen, ob mehrere Instanzen von Excel ausgeführt werden.

Öffnen Sie mit der Methode *Open* das in der Variablen *$strPath* angegebene Excel-Arbeitsblatt und speichern Sie das resultierende Arbeitsmappenobjekt in der Variablen *$workbook*:

```
$WorkBook=$objExcel.Workbooks.Open($strPath)
```

Sie können nun auf das Arbeitsblatt in der Arbeitsmappe zugreifen (in diesem Beispiel auf das Arbeitsblatt *NewUser*):

```
$worksheet = $workbook.sheets.item("NewUser")
```

Weisen Sie den drei Variablen, die die Zellen im Skript referenzieren, Werte zu. Die erste Variable ist *$intRow*. Dies ist die erste Zeile im Arbeitsblatt, die Benutzerdaten enthält. Die ersten drei Zeilen sind die Kopfzeilen der Spalten. Um herauszufinden, wie viele Benutzer geändert werden müssen, verwenden Sie die Eigenschaft *Rows* der Eigenschaft *UsedRange* des Arbeitsblatts. *UsedRange* zeigt an, wie viele Zeilen des Arbeitsblatts Einträge enthalten. Fragen Sie die Anzahl ab und speichern Sie diese in der Variablen *$intRowMax*. In der Variablen *$intHdrRow* ist die Anzahl der Kopfzeilen im Excel-Arbeitsblatt gespeichert. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
$intRow = 4
$intRowMax =  ($worksheet.UsedRange.Rows).count
$intHdrRow = 3
```

Die verbleibenden Variablen werden mit Standardwerten initialisiert. Der Vorname der Benutzer ist in der ersten Spalte eingetragen und die Organisationseinheit, in der sich der jeweilige Benutzer befindet, in der zweiten Spalte. Die Variable *$lname* enthält den Nachnamen des Benutzers, der in der dritten Spalte gespeichert ist. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
$intcolumn = $null
$lname = 3
$intName = 1
$intOU = 2
$Class = "User"
$dc = "dc=nwtraders,dc=com"
```

Im nächsten Codeabschnitt werden die Zeilen des Arbeitsblatts durchlaufen. Rufen Sie mit der Methode *item()* die in der Eigenschaft *Value2* gespeicherten Daten ab. Speichern Sie den Benutzernamen und den Namen der Organisationseinheit in den Variablen *$name* und *$ou*. Geben Sie eine Statusmeldung aus und stellen Sie die Verbindung mit Active Directory her:

```
for($introw = 4 ; $intRow -le $intRowMax ; $intRow++)
{
 $name = $worksheet.cells.item($intRow,$intName).value2
 $ou =  $worksheet.cells.item($intRow,$intOU).value2
 "Modifizieren von $name,$ou,$dc"
 $ADSI = [ADSI]"LDAP://$name,$ou,$dc"
```

Überprüfen Sie den aus dem Excel-Arbeitsblatt abgerufenen Wert. Wenn der Wert Null ist, geben Sie eine entsprechende Meldung mit dem fehlenden Benutzernamen aus. Wenn das Benutzerobjekt gefunden wird, aktualisieren Sie die Werte in Active Directory und beenden das Skript. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
for($intcolumn = 1 ; $intcolumn -le 30 ; $intcolumn++)
  {
   if ($worksheet.cells.item($intRow,$intcolumn).value2 -eq $null)
   {
    "Der Wert $($worksheet.cells.item($intHdrRow,$intcolumn).value2)" +
   "wurde für das Benutzerkonto $($worksheet.cells.item($intRow,$lname).value2) nicht angegeben."
   }
   ELSE {
 Write-host -ForegroundColor green
worksheet.cells.item($intHdrRow,$intcolumn).value2
$worksheet.cells.item($intRow,$intcolumn).value2
$ADSI.put($property, $value)
       }
   }
$ADSI.setInfo()
}
$objexcel.quit()
```

Das vollständige Skript *ReadExcelModifyUsers.ps1* hat folgenden Aufbau:

ReadExcelModifyUsers.ps1

```
$strPath="C:\Chapter13\NewUser.xls"
$objExcel=New-Object -ComObject Excel.Application
$objExcel.Visible=$false
$WorkBook=$objExcel.Workbooks.Open($strPath)
$worksheet = $workbook.sheets.item("NewUser")
$intRow = 4
$intRowMax =  ($worksheet.UsedRange.Rows).count
$intHdrRow = 3
$intcolumn = $null
$lname = 3
$intName = 1
$intOU = 2
$Class = "User"
$dc = "dc=nwtraders,dc=com"

for($introw = 4 ; $intRow -le $intRowMax ; $intRow++)
{
 $name = $worksheet.cells.item($intRow,$intName).value2
 $ou =  $worksheet.cells.item($intRow,$intOU).value2
 "Modifizieren von $name,$ou,$dc"
 $ADSI = [ADSI]"LDAP://$name,$ou,$dc"

  for($intcolumn = 1 ; $intcolumn -le 30 ; $intcolumn++)
```

```
  {
   if ($worksheet.cells.item($intRow,$intcolumn).value2 -eq $null)
   {
    "Der Wert  $($worksheet.cells.item($intHdrRow,$intcolumn).value2)" +
   "wurde für das Benutzerkonto $($worksheet.cells.item($intRow,$lname).value2) nicht angegeben."
   }
   ELSE {
         $value = $worksheet.cells.item($intHdrRow,$intcolumn).value2
         $property= $worksheet.cells.item($intRow,$intcolumn).value2
         $ADSI.put($property, $value)
       }
   }
$ADSI.setInfo()
}
$objexcel.quit()
```

Zusammenfassung

In diesem Kapitel wurde die Arbeit mit Benutzerkonten erklärt. Sie haben Benutzer und Gruppen in Active Directory erstellt. Anschließend haben Sie sowohl Benutzerkonten als auch Domänengruppen geändert. Das Kapitel wurde mit dem Erstellen mehrerer Benutzer mit mehreren Attributen unter Verwendung eines Excel-Arbeitsblatts abgeschlossen.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

KAPITEL 14

Konfigurieren des Clusterdienstes

Nach Abschluss dieses Kapitels können Sie:

- Die Netzwerkanforderungen konfigurieren
- Die Datenträgerressourcen verwalten
- Die Clusterressourcen verwalten
- Clusterprobleme beheben

Auf der Begleit-CD Alle Skripts in diesem Kapitel befinden sich auf der Begleit-CD im Ordner *\Scripts\Chapter14*.

Überprüfen von Clusterservern

Mit den WMI-Klassen im WMI-Namespace *root\MSCluster* können Sie zahlreiche Aufgaben ausführen. Der Vorteil von WMI ist, dass diese Technologie lokal und remote verwendet werden kann. Die WMI-Klasse *MSCluster_Cluster* ist die wichtigste Klasse zum Abrufen von Informationen über Clusterserver.

Hinweis Alle WMI-Klassen zum Verwalten von Clusterservern befinden sich im WMI-Namespace *root\MSCluster*. Diese Klassen beginnen mit *MSCluster*, was die Arbeit mit diesen WMI-Klassen vereinfacht.

Um eine Liste der WMI-Klassen zum Verwalten des Clusterfailovers in Windows Server 2008 Enterprise oder Data Center abzurufen, führen Sie folgenden Windows PowerShell-Befehl aus:

```
Get-WmiObject -Namespace root\mscluster -list
```

Dieser Befehl gibt eine umfangreiche Liste der WMI-Klassen zurück, die der folgenden gekürzten Liste entspricht:

```
__IndicationRelated
__FilterToConsumerBinding
__EventConsumer
__AggregateEvent
__SystemEvent
__EventDroppedEvent
__EventQueueOverflowEvent
__QOSFailureEvent
__ConsumerFailureEvent
MSCluster_Event
MSCluster_EventObjectRemove
MSCluster_EventObjectAdd
MSCluster_EventPropertyChange
MSCluster_EventRegistryChange
MSCluster_EventClusterCallback
MSCluster_EventStateChange
```

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

```
MSCluster_EventResourceStateChange
MSCluster_EventGroupStateChange
__EventGenerator
__SecurityDescriptor
__PARAMETERS
CIM_ManagedSystemElement
CIM_LogicalElement
CIM_System
CIM_ComputerSystem
CIM_Cluster
MSCluster_Cluster
CIM_UnitaryComputerSystem
MSCluster_Node
CIM_LogicalDevice
MSCluster_NetworkInterface
```

Bei der Auflistung der WMI-Klassennamen können mehrere Probleme auftreten. Beispielsweise sind die meisten angezeigten Elemente für einen Netzwerkadministrator nicht von Interesse. Ein weiteres Problem ist, dass die Liste keine erkennbare Reihefolge aufweist. Sie können dieses Problem beheben, indem Sie die Ergebnisse filtern, um die Ausgabe zu sortieren. Speichern Sie den Befehl beispielsweise in einem Skript namens *ListClusterWMIClasses.ps1* ab.

Tipp Das Skript *ListClusterWMIClasses.ps1* wurde zum Anzeigen der WMI-Clusterklassen erstellt. Da jedoch der Parameter ***-namespace*** angegeben ist, können Sie mit dem Skript eine gefilterte Liste der WMI-Klassen aus den WMI-Namespaces anzeigen.

Das Skript *ListClusterWMIClasses.ps1* beginnt mit einer *param*-Anweisung. Sie können sowohl die Skriptausführung steuern als auch den Namespace ändern. Die drei erforderlichen Parameter sind **-computer**, **-namespace**, und **-help**. Die *param*-Anweisung lautet:

```
param(
      $computer = "localhost",
      $namespace = "root\mscluster",
      [switch]$help
     )
```

Erstellen Sie als Nächstes die Funktion *funhelp()*, um eine Hilfemeldung anzuzeigen, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Die Funktion beginnt mit der *function*-Anweisung und weist den Inhalt einer *here*-Zeichenfolge der Variablen *$helptext* zu. Die *here*-Zeichenfolge beginnt mit *@"* und endet mit *"@*. Sie können die Regeln für Anführungszeichen ignorieren, da die gesamte Eingabe als Zeichenfolge behandelt wird. Sie können also den Text beliebig bearbeiten, ohne die Syntax berücksichtigen zu müssen. Teilen Sie den Hilfetext in drei Bereiche ein: Beschreibung, Parameter und Syntax. Nachdem Sie die *here*-Zeichenfolge erstellt haben, weisen Sie diese der Variablen *$helptext* zu. Anschließend wird die Zeichenfolge ausgegeben und das Skript mit der *exit*-Anweisung beendet. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ListClusterWMIClasses.ps1
Listet die WMI-Klassen aus dem WMI-Namespace auf.

PARAMETER:
```

```
-computer   Der Name des Computers.
-namespace  Der Name des WMI-Namespaces.
-help       Zeigt dieses Hilfethema an.

SYNTAX:
ListClusterWMIClasses.ps1
Zeigt eine Liste aller Cluster-WMI-Klassen aus dem
WMI-Namespace root\mscluster auf dem lokalen Computer an,
ohne CIM- und Systemklassen zu berücksichtigen.

ListClusterWMIClasses.ps1 -computer cluster1
Zeigt eine Liste aller Cluster-WMI-Klassen aus dem
WMI-Namespace root\mscluster auf einem Remotecomputer namens cluster1 an,
ohne CIM- und Systemklassen zu berücksichtigen.

ListClusterWMIClasses.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Überprüfen Sie den WMI-Namespace, bevor Sie die WMI-Befehle ausführen. Erstellen Sie die Funktion *funtestns()*, um den zum Testen des Namespaces erforderlichen Code zu implementieren. Legen Sie als Erstes die automatische Variable *$erroractionpreference* auf den Wert *SilentlyContinue* fest. Dieser Wert verhindert das Anzeigen von Fehlermeldungen und setzt die Skriptausführung auch dann fort, wenn ein Problem auftritt.

Hinweis Die automatische Variable *$erroractionpreference="SilentlyContinue"* entspricht der *On Error Resume Next*-Anweisung in VBScript. Legen Sie diese Variable nur fest, um möglicherweise auftretende Fehler zu überspringen. Sie können die Variable verwenden, wenn Sie Informationen abrufen. Wenn Sie jedoch Änderungen an Dateien oder den Informationen in Active Directory vornehmen, kann ein nicht behandelter Fehler zu einem schwerwiegenden Problem werden.

Um zu überprüfen, ob der Namespace vorhanden ist, erstellen Sie ein COM-Objekt. Da das Cmdlet **Test-Path** diesen Vorgang nicht ausführen kann, müssen Sie wie folgt vorgehen: Erstellen Sie eine neue Instanz des *SWbemLocator*-Objekts. Das *SWbemLocator*-Objekt umfasst die Methode *ConnectServer.* Diese Methode greift auf den angegebenen Namespace zu und überprüft, ob der Befehl erfolgreich ausgeführt werden kann. Verwenden Sie eine *[void]*-Einschränkung, um Meldungen zu unterdrücken. Überprüfen Sie mit der automatischen Variablen *$?*, ob der Befehl ausgeführt wurde. Die Variable gibt *True* oder *False* zurück, abhängig davon, ob der Befehl ausgeführt wurde oder fehlgeschlagen ist.

Wenn der Befehl fehlschlägt, verwenden Sie in der Funktion *funtestns* das Cmdlet **Write-Host**, um eine Meldung anzuzeigen, dass der Namespace ungültig ist, und das Skript zu beenden. Legen Sie die automatische Variable *$erroractionpreference* wieder auf den Standardwert *Continue* fest. Die Funktion *funtestns()* ist wie folgt implementiert:

```
Function funTestNS()
{
 $erroractionpreference="silentlycontinue"
 $objWMI = New-Object -ComObject wbemscripting.swbemlocator
 [void]$objWMI.ConnectServer($computer,$namespace)
 if(!$?)
   {
    Write-host -foregroundcolor red "$namespace ist kein" `
    "gültiger WMI-Namespace auf $computer"
    exit
   }
 $erroractionpreference="continue"
}
```

Die nächste zu implementierende Funktion ist *funwmiclass()*. Stellen Sie mit dem Cmdlet **Get-WmiObject** die Verbindung mit dem in der Variablen *$computer* angegebenen Computer und mit dem in der Variablen *$namespace* angegebenen Namespace her. Legen Sie den Parameter **-list** fest. Das Skript wird standardmäßig für den lokalen Computer und den Clusternamespace ausgeführt. Sie können jedoch in der Befehlszeile andere Werte für die Parameter angeben. Nachdem die WMI-Klassen aus dem angegebenen Namespace abgerufen wurden, speichern Sie das Ergebnis in der Variablen *$wmiclasses* und geben Sie die Informationen aus, einschließlich der Anzahl der ermittelten Klassen.

Filtern Sie alle Systemklassen heraus (die Klassen, die mit einem doppelten Unterstrich (__) beginnen). Geben Sie hierzu den Filter *[a-z]** an. Das heißt, der Name beginnt mit einem der Buchstaben zwischen *a* bis *z* gefolgt von einem beliebigen Zeichen. Der zweite Teil der Abfrage entfernt die abstrakten WMI-Klassen. Diese Klassen beginnen mit CIM (Common Information Model). Verwenden Sie hierzu den *-notlike*-Operator, um anzugeben, dass der Name der WMI-Klasse nicht mit den Buchstaben *cim* beginnen darf. Wählen Sie mit dem Cmdlet **Select-Object** die Eigenschaft *Name* aus und sortieren Sie die Liste mit dem Cmdlet **Sort-Object**. Die Funktion *funwmiclass()* ist wie folgt implementiert:

```
Function funWMIClass()
{
 $wmiClasses = Get-wmiobject -computername $computer `
             -namespace $namespace -list
 "Es gibt $($wmiclasses.count) Klassen im Namespace $namespace" `
  + " auf $computer `nDie WMI-Klassen lauten: "

 Get-WmiObject -computername $computer -Namespace $namespace -list |
 Where-Object { $_.name -like '[a-z]*' -and $_.name -notlike 'cim*' } |
 select-object -property name |
 sort-object -property name
}
```

Überprüfen Sie mittels der Variablen *$help*, ob das Skript mit dem Parameter **-help** ausgeführt wurde. Wenn die Variable vorhanden ist, rufen Sie die Funktion *funhelp()* auf. Ist die Variable *$help* hingegen nicht vorhanden, überprüfen Sie den WMI-Namespace, indem Sie die Funktion *funtestns()* aufrufen. Wird die Funktion erfolgreich ausgeführt, rufen Sie die Funktion *funwmiclass()* auf. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
if($help) { "Ausgeben der Hilfeinformationen ..." ; funhelp }
funTestNS
funWMIClass
```

Das vollständige Skript *ListClusterWMIClasses.ps1* hat folgenden Aufbau:

ListClusterWMIClasses.ps1

```
param(
      $computer = "localhost",
      $namespace = "root\mscluster",
      [switch]$help
     )

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ListClusterWMIClasses.ps1
Listet die WMI-Klassen aus dem WMI-Namespace auf.

PARAMETER:
-computer   Der Name des Computers.
-namespace  Der Name des WMI-Namespaces.
-help       Zeigt dieses Hilfethema an.

SYNTAX:
ListClusterWMIClasses.ps1
Zeigt eine Liste aller Cluster-WMI-Klassen aus dem
WMI-Namespace root\mscluster auf dem lokalen Computer an,
ohne CIM- und Systemklassen zu berücksichtigen.

ListClusterWMIClasses.ps1 -computer cluster1
Zeigt eine Liste aller Cluster-WMI-Klassen aus dem
WMI-Namespace root\mscluster auf einem Remotecomputer namens cluster1 an,
ohne CIM- und Systemklassen zu berücksichtigen.

ListClusterWMIClasses.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}

Function funTestNS()
{
 $erroractionpreference="silentlycontinue"
 $objWMI = New-Object -ComObject wbemscripting.swbemlocator
 [void]$objWMI.ConnectServer($computer,$namespace)
 if(!$?)
   {
    Write-host -foregroundcolor red "$namespace ist kein" `
    "gültiger WMI-Namespace auf $computer."
    exit
   }
 $erroractionpreference="continue"
}

Function funWMIClass()
{
```

```
 $wmiClasses = Get-wmiobject -computername $computer `
             -namespace $namespace -list
 "Es gibt $($wmiclasses.count) Klassen im Namespace $namespace" `
  + " auf $computer `nDie WMI-Klassen lauten: "

 Get-WmiObject -computername $computer -Namespace $namespace -list |
 Where-Object { $_.name -like '[a-z]*' -and $_.name -notlike 'cim*' } |
 select-object -property name |
 sort-object -property name
}

if($help) { "Ausgeben der Hilfeinformationen ..." ; funhelp }
funTestNS
funWMIClass
```

Die Ausgabe des Skripts *ListClusterWMIClasses.ps1* ist wesentlich hilfreicher als die ungefilterte Ausgabe des Cmdlets **Get-WmiObject**. Die Ausgabe des Skripts *ListClusterWMIClasses.ps1* ist in folgendem Listing aufgeführt. Im Gegensatz zu den 130 Klassen in der ungefilterten Liste werden 40 WMI-Klassen zurückgegeben. Die WMI-Klassen beziehen Sie auf Cluster, Knoten, Dienste, Netzwerkschnittstellen, Datenträger und andere Komponenten des Clusterservers.

```
MSCluster_AvailableDisk
MSCluster_Cluster
MSCluster_ClusterToAvailableDisk
MSCluster_ClusterToNetwork
MSCluster_ClusterToNetworkInterface
MSCluster_ClusterToNode
MSCluster_ClusterToQuorumResource
MSCluster_ClusterToResource
MSCluster_ClusterToResourceGroup
MSCluster_ClusterToResourceType
MSCluster_Disk
MSCluster_DiskPartition
MSCluster_DiskToDiskPartition
MSCluster_Event
MSCluster_EventClusterCallback
MSCluster_EventGroupStateChange
MSCluster_EventObjectAdd
MSCluster_EventObjectRemove
MSCluster_EventPropertyChange
MSCluster_EventRegistryChange
MSCluster_EventResourceStateChange
MSCluster_EventStateChange
MSCluster_LogicalElement
MSCluster_Network
MSCluster_NetworkInterface
MSCluster_NetworkToNetworkInterface
MSCluster_Node
MSCluster_NodeToActiveGroup
MSCluster_NodeToActiveResource
MSCluster_NodeToHostedService
MSCluster_NodeToNetworkInterface
MSCluster_Property
MSCluster_Property_Cluster_PrivateProperties
MSCluster_Property_Group_PrivateProperties
MSCluster_Property_NetInterface_PrivateProperties
```

```
MSCluster_Property_Node_PrivateProperties
MSCluster_Resource
MSCluster_ResourceGroup
MSCluster_ResourceGroupToPreferredNode
MSCluster_ResourceGroupToResource
MSCluster_ResourceToDependentResource
MSCluster_ResourceToDisk
MSCluster_ResourceToPossibleOwner
MSCluster_ResourceType
MSCluster_ResourceTypeToResource
MSCluster_Service
```

Überprüfen der Clusterkonfiguration

Sie können die aktuelle Konfiguration eines Windows Server 2008-Failoverclusters beispielsweise für Verwaltungszwecke überprüfen oder mit einer Basislinienkonfiguration vergleichen. Mit dem Skript *ReportClusterConfig.ps1* können Sie über die WMI-Klasse *MSCluster_Cluster* detaillierte Clusterinformationen abrufen.

Beginnen Sie das Skript *ReportClusterConfig.ps1* mit einer *param*-Anweisung und geben Sie die folgenden vier Parameter an: -**computer**, **-namespace**, **-class** und **-help**. Weisen Sie den ersten drei Parametern Standardwerte zu. Der feste Parameter **-help** erfordert keinen Standardwert, da dieser direkt in die Befehlszeile eingegeben werden muss, um die Hilfeinformationen anzuzeigen. Wenn das Skript ohne diese Parameter ausgeführt wird, wird die Konfiguration des lokalen Clusterservers abgerufen. Die *param*-Anweisung lautet:

```
param(
      $computer="localhost",
      $namespace="root\mscluster",
      $class = "mscluster_cluster",
      [switch]$help
     )
```

Erstellen Sie als Nächstes die Funktion *funhelp()*, indem Sie die Variable *$helptext* deklarieren. Weisen Sie den Inhalt der *here*-Zeichenfolge dieser Variablen zu. Die *here*-Zeichenfolge besteht aus drei Abschnitten: Der Beschreibung des Skripts, den vom Skript akzeptierten Parametern und Syntaxbeispielen. Nachdem Sie die *here*-Zeichenfolge der Variablen *$helptext* zugewiesen haben, können Sie den Inhalt der Variablen anzeigen und das Skript beenden. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ReportClusterConfig.ps1
Listet die aktuelle Clusterkonfiguration auf.

PARAMETER:
-computer  Der Name des Computers.
-namespace Der Name des WMI-Namespaces.
-class     Der Name der abgefragten WMI-Klasse.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
ReportClusterConfig.ps1
```

```
Zeigt eine Liste der aktuellen Clusterkonfiguration
des lokalen Computers an.

ReportClusterConfig.ps1 -computer cluster1
Zeigt eine Liste der aktuellen Clusterkonfiguration
eines Remotecomputers namens cluster1 an.

ReportClusterConfig.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Erstellen Sie nun die Funktion *funline()*, die eine Zeichenfolge akzeptiert. Die Funktion speichert die Länge der Zeichenfolge in der Variablen *$num* und verwendet eine *for*-Anweisung, um eine Zeichenfolge gleicher Länge mit Gleichheitszeichen zu generieren. Verwenden Sie die Variable *$i* als Enumerator, um den Status nachzuverfolgen. Führen Sie diesen Vorgang aus, bis der Wert von *$i* kleiner oder gleich dem Wert der Variablen *$num* ist. Geben Sie *$i++* an, um die Variable *$i* jeweils um 1 zu erhöhen. Speichern Sie die Zeichenfolge mit den Gleichheitszeichen (=) in der Variablen *$funline*. Diese Zeichenfolge wird bei der Ausgabe des Textes als Unterstrich verwendet. Verwenden Sie hierzu zwei **Write-Host**-Anweisungen. Die erste **Write-Host**-Anweisung gibt die Zeichenfolge aus und die zweite **Write-Host**-Anweisung gibt den Inhalt der Variablen *$funline* aus. Die Funktion *funline()* ist wie folgt implementiert:

```
function funline($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  {
   $funline = $funline + "="
  }
   Write-Host -ForegroundColor yellow `n$strIN
   Write-Host -ForegroundColor darkYellow $funline
}
```

Erstellen Sie nun die Funktion *funwmi()*, indem Sie als Erstes eine Verbindung mit dem Namespace im Parameter **-namespace**, dem Computer im Parameter **-computer** und der gewünschten Klasse im Parameter **-class** herstellen. Das Cmdlet **Get-WmiObject** verwendet diese Parameter, um auf WMI zuzugreifen und die Informationen aus der WMI-Klasse abzurufen. Fügen Sie das resultierende WMI-Verwaltungsobjekt in das Cmdlet **ForEach-Object** ein. Dieser Abschitt der Funktion *funwmi()* ist im Folgenden dargestellt:

```
Get-WmiObject -class $class -computername $computer `
                -namespace $namespace |
   foreach-object `
```

Rufen Sie die Funktion *funline()* auf und geben Sie eine Meldung aus, dass die in der Variablen *$class* angegebene Klasse auf dem Computer in der Variablen *$computer* abgerufen wird. Die Funktion *funline()* berechnet die Länge des Zeichenfolgenausdrucks. Die Zeichenfolge wird ausgegeben und unterstrichen. Beispiel:

```
funLine("Abfragen von: $class auf $computer")
```

Verweisen Sie in der automatischen Variablen *$_* auf das aktuelle Objekt in der Pipeline und rufen Sie mit dem *.PSObject*-Objekt alle Eigenschaften der WMI-Klasse ab. Übergeben Sie diese Eigenschaften ebenfalls an die Pipeline. Verwenden Sie das Cmdlet **ForEach-Object** und überprüfen Sie die Werte der Eigenschaften für alle Instanzen der Klasse *MSCluster_Cluster*. Dieser Abschnitt der Funktion *funwmi()* ist im Folgenden dargestellt:

```
funLine("Abfragen von: $class on $computer")
      $_.psobject.properties |
      foreach-object `
       {
        If($_.value)
```

Wenn der Wert der Eigenschaft mit einem doppelten Unterstrich (__) übereinstimmt, führen Sie keinen weiteren Vorgang aus. Sollte der Wert der Eigenschaft jedoch nicht dem doppelten Unterstrich entsprechen, speichern Sie den Namen und den Wert der Eigenschaft in einer Hashtabelle. Dies ermöglicht Ihnen, den Namen und den entsprechenden Wert in der Variablen *$aryprop* zu speichern, um das Array der Eigenschaftenwerte zu verwenden. Nachdem Sie die Informationen einer Variablen zugewiesen haben, können Sie den Wert ausgeben und die Funktion beenden.

Tipp Da die Funktion *funwmi()* geschachtelt ist und zahlreiche geschweifte Klammern enthält, sind die unterschiedlichen Ebenen mit Hinweisen versehen, um die Problembehandlung und das Vornehmen von Änderungen zu vereinfachen.

Dieser Funktionsabschnitt ist im Folgenden dargestellt:

```
if ($_.name -match "__"){}
          ELSE
      {
            $aryProp +=@{ $($_.name)=$($_.value) }
           } #else
         } #if($_.value)
       } #foreach-object $_.psobject.properties
      $aryProp
    } #foreach-object mscluster_cluster
  } #funwmi
```

Die vollständige Funktion *funwmi()* ist wie folgt implementiert:

```
function funwmi($class)
 {
  Get-WmiObject -class $class -computername $computer `
                -namespace $namespace |
   foreach-object `
    {
     funLine("Abfragen von: $class auf $computer")
      $_.psobject.properties |
      foreach-object `
       {
        If($_.value)
         {
     if ($_.name -match "__"){}
          ELSE
      {
            $aryProp +=@{ $($_.name)=$($_.value) }
```

```
          } #else
        } #if($_.value)
      } #foreach-object $_.psobject.properties
     $aryProp
   } #foreach-object mscluster_cluster
 } #funwmi
```

Nachdem Sie die Funktion *funwmi()* erstellt haben, überprüfen Sie, ob die Variable *$help* vorhanden ist. Ist dies der Fall, rufen Sie die Funktion *funhelp()* auf. Sollte die Variable *$help* jedoch nicht vorhanden sein, rufen Sie die Funktion *funwmi()* auf. Codeabschnitt:

```
if($help) { "Ausgeben der Hilfeinformationen" ; funhelp }
funwmi($class)
```

Das vollständige Skript *ReportClusterConfig.ps1* hat folgenden Inhalt:

ReportClusterConfig.ps1

```
param(
      $computer="localhost",
      $namespace="root\mscluster",
      $class = "mscluster_cluster",
      [switch]$help
     )

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ReportClusterConfig.ps1
Listet die WMI-Klassen aus dem WMI-Namespace auf.

PARAMETER:
-computer  Der Name des Computers.
-namespace Der Name des WMI-Namespaces.
-class     Der Name der abgefragten WMI-Klasse.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
ReportClusterConfig.ps1
Zeigt eine Liste der aktuellen Clusterkonfiguration
des lokalen Computers an.

ReportClusterConfig.ps1 -computer cluster1
Zeigt eine Liste der aktuellen Clusterkonfiguration
eines Remotecomputers namens cluster1 an.

ReportClusterConfig.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}

function funline($strIN)
{
```

```
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  {
   $funline = $funline + "="
  }
   Write-Host -ForegroundColor yellow `n$strIN
   Write-Host -ForegroundColor darkYellow $funline
}

function funwmi($class)
 {
  Get-WmiObject -class $class -computername $computer `
                -namespace $namespace |
   foreach-object `
    {
     funLine("Abfragen von: $class auf $computer")
      $_.psobject.properties |
      foreach-object `
       {
        If($_.value)
         {
     if ($_.name -match "__"){}
          ELSE
      {
            $aryProp +=@{ $($_.name)=$($_.value) }
          } #else
        } #if($_.value)
       } #foreach-object $_.psobject.properties
      $aryProp
    } #foreach-object mscluster_cluster
  } #funwmi

if($help) { "Ausgeben der Hilfeinformationen ..." ; funhelp }
funwmi($class)
```

Überprüfen der Knotenkonfiguration

Für die Knoten eines Windows Server 2008-Failoverclusters sind zahlreiche spezielle Konfigurationseinstellungen verfügbar. Ein detaillierter Konfigurationsbericht kann Ihnen beim Überprüfen dieser Einstellungen helfen. Erstellen Sie den Bericht mit dem Skript *ReportNodeConfig.ps1* unter Verwendung der WMI-Klasse *MSCluster_Node* im WMI-Namespace *root\MSCluster*.

Das Skript beginnt mit einer *param*-Anweisung, die vier Parameter definiert. Die ersten drei Parameter, **-computer**, **-namespace**, und **-class**, sind auf Standardwerte festgelegt. Die Standardwerte vereinfachen das Skript und stellen eine gewisse Flexibilität sicher. Beispielsweise kann das Skript auf einem beliebigen Computer eine WMI-Klasse aus einem WMI-Namespace abfragen. Für den Parameter *-help* kann kein Wert angegeben werden. Die *param*-Anweisung:

```
param(
      $computer="localhost",
      $namespace="root\mscluster",
      $class = "mscluster_node",
      [switch]$help
     )
```

Erstellen Sie als Nächstes die Funktion *funhelp()*, die eine Hilfemeldung anzeigt, wenn das Skript mit dem Parameter *-help* ausgeführt wird. Die Funktion *funhelp()* erstellt die Variable *$helptext*, in der die *here*-Zeichenfolge gespeichert wird. Die *here*-Zeichenfolge ist in drei Abschnitte unterteilt: Beschreibung, Parameter und Syntax. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ReportNodeConfig.ps1
Listet die aktuelle Clusterkonfiguration auf.

PARAMETER:
-computer  Der Name des Computers.
-namespace Der Name des WMI-Namespaces.
-class     Der Name der abgefragten WMI-Klasse.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
ReportNodeConfig.ps1
Listet die aktuelle Knotenkonfiguration für einen Failover-Cluster auf.

ReportNodeConfig.ps1
Listet die aktuelle Knotenkonfiguration für einen Failover-Cluster
auf dem lokalen Computer auf.

ReportNodeConfig.ps1 -computer cluster1
Listet die aktuelle Knotenkonfiguration für einen Failover-Cluster
auf einem Remotecomputer namens cluster1 auf.

ReportNodeConfig.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Erstellen Sie als Nächstes die Funktion *funline()*. Die Funktion *funline()* ist mit der Funktion im Skript *ReportClusterConfig.ps1* identisch. Die Funktion *funline()* ist wie folgt implementiert:

```
function funline($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  {
   $funline = $funline + "="
  }
   Write-Host -ForegroundColor yellow `n$strIN
   Write-Host -ForegroundColor darkYellow $funline
}
```

Der nächste Schritt umfasst die Funktion *funwmi()*. Die Funktion *funwmi()* unterscheidet sich etwas von der Funktion im Skript *ReportClusterConfig.ps1*. Das Skript *ReportClusterConfig.ps1* verarbeitet die Ergebnisse einer einzigen Instanz der WMI-Klasse. Das ist ausreichend, da ein Windows Server 2008-Failoverserver nur Mitglied in einem Cluster sein kann. Da ein Cluster jedoch normalerweise mehr als

einen Knoten umfasst, muss das Skript mehrere Instanzen verarbeiten können. Stellen Sie mit dem Cmdlet **Get-WmiObject** die Verbindung mit der Klasse, dem Computer und dem Namespace her, die in den Befehlszeilenparametern des Skripts angegeben sind. Wenn das Skript mit den Standardwerten ausgeführt wird, wird der lokale Host und der WMI-Namespace *root\MSCluster* abgefragt, um die Instanzen der WMI-Klasse *MSCluster_Node* abzurufen.

Hinweis Wie für das Skript *ReportClusterConfig.ps1* können alle Parameter des Skripts *ReportNodeConfig.ps1* als Befehlszeilenparameter angegeben werden. Sie können mit dem Skript auf einem beliebigen Computer alle WMI-Klassen aus einem WMI-Namespace abfragen. Da das Skript leere Eigenschaftswerte herausfiltert, ist die Ausgabe übersichtlich.

Nachdem das Cmdlet **Get-WmiObject** ein Verwaltungsobjekt zurückgegeben hat, durchlaufen Sie die Verwaltungsobjekte mit dem Cmdlet **ForEach-Object**. Das Cmdlet **ForEach-Object** durchläuft alle Objekte und geniert keinen Fehler, wenn nur ein Objekt vorhanden ist. Verwenden Sie die Funktion *funline()*, um die abgefragte Klasse und den Namen des Computers hervorzuheben, auf den die Klasse verweist. Sie müssen die Eigenschaften des Objekts mit der Eigenschaft *Properties* abrufen, um auf die Methoden und Eigenschaften des WMI-Objekts zuzugreifen, die nicht vom Cmdlet **Get-WmiObject** angezeigt werden. Die Eigenschaft *Properties* aus *PSObject* gibt ein Objekt zurück, das alle Eigenschaften der WMI-Klasse repäsentiert. Nachdem Sie die Eigenschaften abgerufen haben, überprüfen Sie die Werte mit dem Cmdlet **ForEach-Object**. Stellen Sie unter Verwendung einer *if*-Anweisung sicher, dass ein Wert vorhanden ist. Wenn kein Wert vorhanden ist, ist die Eigenschaft leer. Filtern Sie außerdem die Eigenschaftsnamen, die mit einem doppelten Unterstrich (__) beginnen, heraus. Wenn der Name und der Wert der Eigenschaft angezeigt werden, speichern Sie diese in einer Hashtabelle. Geben Sie die Hashtabelle aus und fahren Sie mit dem nächsten Verwaltungsobjekt fort, bis Sie alle vom Cmdlet **Get-WmiObject** zurückgegebenen Elemente verarbeitet haben. Die vollständige Funktion *funwmi()* ist wie folgt implementiert:

```
function funwmi($class)
 {
  Get-WmiObject -class $class -computername $computer `
                -namespace $namespace |
   foreach-object `
    {
     funLine("Abfragen von: $class auf $computer")
      $_.psobject.properties |
      foreach-object `
       {
        If($_.value)
         {
     if ($_.name -match "__"){}
          ELSE
      {
            $aryProp +=@{ $($_.name)=$($_.value) }
           } #else
         } #if($_.value)
       } #foreach-object $_.psobject.properties
      $aryProp
      $aryProp = $null
    } #foreach-object mscluster_node
  } #funwmi
```

Stellen Sie sicher, dass die Variable *$help* vorhanden ist. Ist dies der Fall, rufen Sie die Funktion *funhelp()* auf. Sollte die Variable *$help* nicht vorhanden sein, führen Sie die Funktion *funwmi()* aus. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
if($help) { "Ausgeben der Hilfeinformationen ..." ; funhelp }
funwmi($class)
```

Das vollständige Skript *ReportNodeConfig.ps1* hat folgenden Aufbau:

ReportNodeConfig.ps1

```
param(
      $computer="localhost",
      $namespace="root\mscluster",
      $class = "mscluster_node",
      [switch]$help
     )

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ReportNodeConfig.ps1
Listet die aktuelle Clusterkonfiguration auf.

PARAMETER:
-computer  Der Name des Computers.
-namespace Der Name des WMI-Namespaces.
-class     Der Name der abgefragten WMI-Klasse.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
ReportNodeConfig.ps1
Listet die aktuelle Knotenkonfiguration für einen Failover-Cluster auf.

ReportNodeConfig.ps1
Listet die aktuelle Knotenkonfiguration für einen Failover-Cluster
auf dem lokalen Computer auf.

ReportNodeConfig.ps1 -computer cluster1
Listet die aktuelle Knotenkonfiguration für einen Failover-Cluster
auf einem Remotecomputer namens cluster1 auf.

ReportNodeConfig.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}

function funline($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  {
```

```
   $funline = $funline + "="
  }
   Write-Host -ForegroundColor yellow `n$strIN
   Write-Host -ForegroundColor darkYellow $funline
}

function funwmi($class)
 {
  Get-WmiObject -class $class -computername $computer `
                -namespace $namespace |
   foreach-object `
    {
     funLine("Abfragen von: $class auf $computer")
      $_.psobject.properties |
      foreach-object `
       {
        If($_.value)
         {
     if ($_.name -match "__"){}
          ELSE
      {
            $aryProp +=@{ $($_.name)=$($_.value) }
          } #else
        } #if($_.value)
       } #foreach-object $_.psobject.properties
      $aryProp
      $aryProp = $null
    } #foreach-object mscluster_node
  } #funwmi

if($help) { "Ausgeben der Hilfeinformationen ..." ; funhelp }
funwmi($class)
```

Abfragen mehrerer Clusterklassen

Eine der interessanteren Aufgaben, die Sie mit Windows PowerShell ausführen können, ist das gleichzeitige Abfragen mehrerer WMI-Klassen. Dieser Prozess ist relativ einfach, da Windows PowerShell Arrays automatisch verarbeiten kann und die Eigenschaften der WMI-Klassen automatisch auflistet. Wenn Sie diese beiden Features in einem Skript miteinander kombinieren, z.B. *ReportMultipleClasses.ps1*, können Sie ein interessantes Tool erstellen. Die Datei *Cluster.txt* enthält ein Beispiel der Ausgabe, die vom Skript *ReportMultipleClasses.ps1* mit dem Parameter **-all** erstellt wird. Dieser Parameter listet alle WMI-Klassem im Namespace auf, um alle Klassen abzufragen, und speichert das Resultat in einer temporären Textdatei.

Um die WMI-Klassen im Namespace nur aufzulisten, führen Sie das Skript mit dem Parameter **-list** aus. Wenn Sie die Parameter **-file** und **-list** zusammen angeben, werden die Ergebnisse in eine temporäre Textdatei geschrieben. Die Datei *ClusterClasses.txt* enthält eine Beispielausgabe.

Das Skript *ReportMultipleClasses.ps1* beginnt mit einer *param*-Anweisung. Dieses Skript umfasst die normalen Parameter **-computer**, **-namespace**, und **-class** sowie die folgenden Parameter mit festen Werten: **-file** (schreibt die Informationen in eine Datei), **-list** (erstellt eine Liste der WMI-Klassen im Namespace) und **-all** (zeigt alle WMI-Klassen im Namespace an). Jeder dieser Parameter fragt die Klasse bzw. die Klassen ab und speichert die Ergebnisse in einer Datei. Die *param*-Anweisung lautet:

```
param(
      $computer="localhost",
      $namespace="root\mscluster",
      $class,
      [switch]$file,
      [switch]$list,
      [switch]$all,
      [switch]$help
     )
```

Erstellen Sie als Nächstes die Funktion *funhelp()*. Diese Funktion listet alle Parameter des Skripts, einige Syntaxbeispiele und eine Beschreibung auf. Die Funktion speichert die Informationen in der Variablen *$helptext* und zeigt diese an, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Die Funktion ist im Folgenden dargestellt:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ReportMultipleClasses.ps1
Fragt eine oder mehrere WMI-Klassen eines geclusterten Server ab.
Zeigt die Informationen entweder auf dem Bildschirm an
oder speichert diese in einer temporären Textdatei ab.

PARAMETER:
-computer  Der Name des Computers.
-namespace Der Name des WMI-Namespaces.
-class     Die Namen der abgefragten WMI-Klasse(n).
-file      Speichert die Ergebnisse in einer temporären Textdatei ab
           und zeigt diese in Notepad an.
-list      Listet die WMI-Klassen aus dem Namespace auf.
-all       Fragt alle WMI-Klassen ab und speichert die Ergebnisse in einer temporären Textdatei ab.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
ReportMultipleClasses.ps1
Listet die WMI-Clusterklassen des lokalen Computers auf.

ReportMultipleClasses.ps1 -class MSCluster_Network
Gibt detaillierte Informationen über die Konfiguration der
Netzwerkschnittstellen auf dem aktuellen Cluster aus.

ReportMultipleClasses.ps1 -class mscluster_service, mscluster_cluster
Ermittelt Informationen zum Clusterdienst und zum Failover-Cluster
über eine Abfrage der beiden WMI-Klassen: mscluster_service und mscluster_cluster.
Beachten Sie, dass Anführungszeichen nicht erforderlich sind, die
Klassen jedoch durch Kommas voneinander getrennt werden müssen.

ReportMultipleClasses.ps1 -all
Fragt alle WMI-Klassen aus dem Namespace ab und speichert
die Ergebnisse in einer temporären Textdatei.

ReportMultipleClasses.ps1 -list
Erzeugt ein Listing aller WMI-Klassen aus dem Namespace.

ReportMultipleClasses.ps1 -list -file
Erzeugt ein Listing aller WMI-Klassen aus dem Namespace
```

```
und speichert die Ergebnisse in einer temporären Textdatei.

ReportMultipleClasses.ps1 -class mscluster_service -file
Fragt die WMI-Klasse mscluster_service auf dem lokalen Computer ab
und speichert die Ergebnisse in einer temporären Textdatei.


ReportMultipleClasses.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
} #end function
```

Die nächste Funktion ist *funline()*, die Sie bereits in anderen Skripts in diesem Kapitel verwendet haben. Die Funktion *funline()* ist im Abschnitt „Überprüfen der Clusterkonfiguration“ ausführlich beschrieben. Die Funktion *funline()* umfasst folgende Anweisungen:

```
function funline($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  {
   $funline = $funline + "="
  }
   Write-Host -ForegroundColor yellow `n$strIN
   Write-Host -ForegroundColor darkYellow $funline
} #end function
```

Die nächste Funktion ist *funtestns()*, mit der sichergestellt wird, dass das Skript einen WMI-Namespace abfragt, der tatsächlich vorhanden ist. Erstellen Sie eine Instanz des *SWbemLocator*-Objekts und verwenden Sie die Methode *ConnectServer()*. Sie haben diese Funktion bereits im Skript *ListClusterWMIClasses.ps1* verwendet, das im Abschnitt „Überprüfen des Clusterservers“ ausführlich beschrieben ist. Die Funktion *funtestns()* ist wie folgt implementiert:

```
Function funTestNS()
{
 $erroractionpreference="silentlycontinue"
 $objWMI = New-Object -ComObject wbemscripting.swbemlocator
 [void]$objWMI.ConnectServer($computer,$namespace)
 if(!$?)
   {
    Write-host -foregroundcolor red "$namespace ist kein" `
    " gültiger WMI-Namespace auf $computer."
    exit
   }
 $erroractionpreference="continue"
} #end function
```

Die Funktion *funlist()* ist der Funktion gleichen Namens im Skript *ListClusterWMIClasses.ps1* ähnlich. Es bestehen jedoch einige wichtige Unterschiede. Die Funktion *funlist()* ruft zuerst die Funktion *funtestns()* auf, um sicherzustellen, dass der WMI-Namespace gültig ist. Wenn der Namespace gültig ist, stellen Sie mit dem Cmdlet **Get-WmiObject** eine Verbindung mit dem im Parameter **-computer** angegebenen Computer her, beziehen den Namespace ein, der im Parameter **-namespace** angegeben ist, und listen mit dem Parameter **-list** des Cmdlets **Get-WmiObject** alle WMI-Klassen im Namespace auf.

Der WMI-Standardnamespace ist *root\MSCluster*. Speichern Sie die aufgelisteten WMI-Klassen in der Variablen *$wmiclasses*. Erstellen Sie unter Verwendung der Eigenschaft *Count* des Objekts, auf das die Variable *$wmiclasses* verweist, eine Kopfzeile für die Liste. Die Kopfzeile besteht aus einer Zeichenfolge mit der Anzahl der WMI-Klassen im Namespace, dem Namespace und dem Computernamen.

Hinweis Damit die Kopfzeile gelesen werden kann, müssen Sie diese in der Funktion *funlist()* umbrechen. Verwenden Sie hierzu das Graviszeichen (`) und das Pluszeichen (+), um die Zeile fortzusetzen und zu verknüpfen. Wenn Sie die Informationen mit dem Cmdlet **Write-Host** ausgeben, müssen Sie nur das Graviszeichen eingeben.

Die Kopfzeile wird in der Variablen *$header* gespeichert. Dieser Abschitt der Funktion *funlist()* ist im Folgenden dargestellt:

```
Function funList()
{
 funtestNS
 $wmiClasses = Get-wmiobject -computername $computer `
             -namespace $namespace -list
 $header = "Es gibt $($wmiclasses.count) Klassen" `
         + " im Namespace $namespace auf $computer.
          Die WMI-Klassen lauten:
           "
```

Überprüfen Sie, ob das Skript mit dem Parameter **-file** ausgeführt wurde und leiten Sie in diesem Fall die Ausgabe in eine Datei um, indem Sie den Wert in der Variablen *$header* in das Cmdlet **Out-File** einfügen. Geben Sie den in der Variablen *$tmpfile* gespeicherten Dateinamen und Pfad im Parameter **-filepath** an und legen Sie den Parameter **-append** fest, damit der Dateiinhalt nicht überschrieben wird. Im vorliegenden Fall ist der Parameter **-append** jedoch nicht unbedingt erforderlich. Codeabschnitt:

```
 if($file)
  {
   $header |
   out-file -filepath $tmpfile -append
  }
```

Wenn Sie den Parameter **-file** nicht in der Befehlszeile eingeben, wird keine Datei erstellt und die Ausgabe wird nur auf dem Bildschirm angezeigt. Geben Sie einfach die zuvor erstellte Kopfzeile aus. Verwenden Sie das Cmdlet **Get-WmiObject** und geben Sie die Parameter **-computername**, **-namespace** und **-list** an. Fügen Sie das Ergebnis in das Cmdlet **Where-Object** ein und filtern Sie alle Systemklassen sowie die Klassen, die mit *cim* beginnen, heraus. Wählen Sie anschließend die Eigenschaft *Name* aus und sortieren Sie die Liste nach dem Namen. Speichern Sie die Ausgabe des Befehls in der Variablen *$classes*. Dieser Abschitt der Funktion *funlist()* ist im Folgenden dargestellt:

```
 ELSE
  {
   $header
  }

  $classes = Get-WmiObject -computername $computer -Namespace `
            $namespace -list |
 Where-Object { $_.name -like '[a-z]*' -and $_.name -notlike 'cim*' } |
 select-object -property name |
 sort-object -property name
```

Sie müssen nun noch einmal unterscheiden, ob das Skript mit oder ohne Parameter **-file** ausgeführt wird. Wenn das Skript mit dem Parameter **-file** ausgeführt wird, fügen Sie die WMI-Klassennamen, die in der Variablen *$classes* gespeichert sind, in das Cmdlet **Out-File** ein und zeigen Sie den Inhalt der Datei in Notepad an. Der Dateiname und der Pfad im Cmdlet **Out-File** verweisen auf eine temporäre Datei im Temp-Verzeichnis. Wenn das Skript ohne Parameter **-file** ausgeführt wird, geben Sie die Klassenliste einfach auf dem Bildschirm aus. Dieser Abschitt der Funktion *funlist()* ist im Folgenden dargestellt:

```
 if($file)
  {
   $classes |
   out-file -filepath $tmpfile -append
   notepad $tmpfile
  }
 ELSE
  {
   $classes
  }
exit
} #end function funlist
```

Die nächste Funktion ist *funall()*. Diese Funktion ruft zuerst die Funktion *funtestns* auf, um sicherzustellen, dass der WMI-Namespace gültig ist. Anschließend stellt die Funktion mit dem Cmdlet **Get-WmiObject** eine Verbindung mit dem in der Variablen *$namespace* angegebenen Namespace her und erstellt mit dem Parameter **-list** eine Liste aller WMI-Klassen im Namespace. Fügen Sie die Liste der WMI-Klassen in das Cmdlet **Where-Object** ein und filtern Sie nach Namen, die mit einem Buchstaben (*a* bis *z)* beginnen. Suchen Sie jedoch nicht nach Namen, die mit den Buchstaben *cim* beginnen. Fügen Sie das gefilterte Objekt in das Cmdlet **ForEach-Object** ein und geben Sie nur den Namen des aktuellen Pipelinezeichens aus. Dieser Abschnitt der Funktion *funall()* ist im Folgenden dargestellt:

```
function funall()
 {
  funtestNS
  Get-WmiObject -Namespace $namespace -list |
  Where-Object { $_.name -like '[a-z]*' -and `
  $_.name -notlike 'cim*' } |
  foreach-object `
   {
    $_.name ;
```

Nachdem Sie den Namen ausgegeben haben, fragen Sie mit dem Cmdlet **Get-WmiObject** die Klasse in *$_.name* ab. Verwenden Sie weiterhin den in der Variablen *$namespace* angegebenen Namespace, fügen Sie die Ergebnisse in das Cmdlet **Out-File** ein und verwenden Sie den Dateipfad, der in der Variablen *$tmpfile* gespeichert ist. Stellen Sie mit dem Parameter **-append** sicher, dass die bereits gespeicherten Ergebnisse nicht überschrieben werden. Nachdem Sie die WMI-Klasse abgefragt und die Ergebnisse in der temporären Datei gespeichert haben, öffnen Sie die Datei in Notepad. Schließen Sie die Funktion *funall()* mit einer *exit*-Anweisung ab, um das Skript zu beenden. Dieser Abschnitt der Funktion *funall()* ist im Folgenden dargestellt:

```
    Get-WmiObject -class $_.name -namespace $namespace |
    out-file -filepath $tmpfile -append
   }
  notepad $tmpfile
  exit
 } #end function funall
```

Erstellen Sie als Nächstes die Funktion *funwmi()*. Überprüfen Sie den WMI-Namespace mit der Funktion *funtestns* und verwenden Sie eine *foreach*-Anweisung, um die WMI-Klassen zu durchlaufen, die in der Variablen *$class* angegeben sind. Fragen Sie mit dem Cmdlet **Get-WmiObject** die WMI-Klasse ab, die in der Variablen *$objclass* gespeichert ist. Stellen Sie eine Verbindung mit dem in der Variablen *$computer* angegebenen Computer her und verwenden Sie den Namespace aus der Variablen *$namespace*. Fügen Sie das resultierende Objekt in das Cmdlet **ForEach-Object** ein und unterstreichen Sie den abgefragten WMI-Klassennamen mit der Funktion *funline()*. Dieser Abschnitt der Funktion *funwmi()* ist im Folgenden dargestellt:

```
function funwmi($class)
 {
  funtestNS
  Foreach($objClass in $class)
   {
    Get-WmiObject -class $objclass -computername $computer `
                  -namespace $namespace |
    foreach-object `
     {
      funLine("Abfragen von: $objclass auf $computer")
```

Nachdem der WMI-Klassenname ausgegeben wurde, fragen Sie die .NET Framework-Klasse *System.Management.Automation.PSObject* ab, um die Eigenschaften des Basisobjekts abzurufen. Da die .NET Framework-Klasse *PSObject* die WMI-Klasse einkapselt, um eine konsistente Schnittstelle zur WMI-Klasse bereitzustellen, können Sie die Klasse *PSObject* abfragen und die Eigenschaften der WMI-Klasse ermitteln. Übergeben Sie die Eigenschaften an das Cmdlet **ForEach-Object**. Wenn das Objekt über eine *Value*-Eigenschaft verfügt, überprüfen Sie, ob eine Übereinstimmung für __ vorhanden ist, um die Systemeigenschaften herauszufiltern. Sollte es sich nicht um eine Systemeigenschaft handeln, erstellen Sie für den Namen und den Wert die Hashtabelle *$aryprop*. Dieser Funktionsabschnitt ist im Folgenden dargestellt:

```
      $_.psobject.properties |
       foreach-object `
        {
         If($_.value)
          {
      if ($_.name -match "__"){}
           ELSE
       {
             $aryProp +=@{ $($_.name)=$($_.value) }
            } #else
         } #if($_.value)
        } #foreach-object $_.psobject.properties
```

Wenn das Skript mit dem Parameter **-file** ausgeführt wird, muss die Ausgabe in eine Textdatei umgeleitet werden. Speichern Sie in diesem Fall den Namen der WMI-Klasse mit dem Cmdlet **Out-File** in der temporären Datei ab. Speichern Sie außerdem die Hashtabelle, die die Namen der Eigenschaften und die Werte enthält, die nicht Null sind, ebenfalls in der Datei. Wenn der Parameter **-file** nicht angegeben wurde, geben Sie den Inhalt der Variablen *$aryprop* auf dem Bildschirm aus und legen danach den Wert von *$aryprop* auf *$null* fest. Dieser Vorgang bereinigt die Variable, die Sie anschließend wiederverwenden können. Schließen Sie die Schleifen, zeigen Sie den Inhalt der temporären Datei in Notepad an und beenden Sie die Funktion *funwmi()*. Codeabschnitt:

```
      If($file)
       {
        $($objClass) | out-file -filepath $tmpfile -append
        $aryProp |
        out-file -filepath $tmpfile -append
       }
      ELSE
       {
        $aryProp
       }
      $aryProp = $null
    } #foreach-object mscluster_node
  } #foreach $objClass
  if($file) { notepad $tmpfile }
 } #Ende der Funktion funwmi
```

Das Skript *ReportMultipleClasses.ps1* ist nun beinahe fertig, aber Sie müssen noch die Befehlszeile überprüfen. Überprüfen Sie, ob die Variable *$help* vorhanden ist. Wenn die Variable vorhanden ist, rufen Sie die Funktion *funhelp()* auf. Wenn die Variable *$file* vorhanden ist, rufen Sie die statische Methode *GetTempFileName()* aus der .NET Framework -Klasse *IO.Path* auf, und speichern Sie den temporären Dateinamen und Pfad in der Variablen *$tmpfile*. Ist die Variable *$list* vorhanden, rufen Sie die Funktion *funtestns()* auf, um den WMI-Namespace zu testen, und rufen Sie anschließend die Funktion *funlist()* auf. Wenn der Parameter **-all** angegeben wurde, erstellen Sie den temporären Dateinamen, testen Sie den WMI-Namespace und rufen die Funktion *funall()* auf. Wurde keine Klasse angegeben, rufen Sie die Funktion *funhelp()* auf. Führen Sie ansonsten die Funktion *funwmi()* aus und übergeben Sie den Klassennamen, der in der Variablen *$class* gespeichert ist, an diese Funktion. Dieser Abschnitt des Skripts *ReportMultipleClasses.ps1* ist im Folgenden dargestellt:

```
if($help) { "Ausgeben der Hilfeinformationen ..." ; funhelp }
if($file) { $tmpfile = [io.path]::getTempfilename() }
if($list) { "Auflisten der Klassen ..." ; funTestNS ; funList }
if($all)  {
            $tmpfile = [io.path]::getTempfilename()
            "Abfragen aller WMI-Klassen im Namespace $namespace ..." ;
            funTestNS ; funAll
          }
if(!$Class) { "Es ist eine Klasse erforderlich ..." ; funhelp }
funwmi($class)
```

Das vollständige Skript *ReportMultipleClasses.ps1* hat folgenden Aufbau:

ReportMultipleClasses.ps1

```
param(
      $computer="localhost",
      $namespace="root\mscluster",
      $class,
      [switch]$file,
      [switch]$list,
      [switch]$all,
      [switch]$help
     )

function funHelp()
{
$helpText=@"
```

```
BESCHREIBUNG:
NAME: ReportMultipleClasses.ps1
Fragt eine oder mehrere WMI-Klassen eines geclusterten Server ab.
Zeigt die Informationen entweder auf dem Bildschirm an
oder speichert diese in einer temporären Textdatei ab.

PARAMETER:
-computer  Der Name des Computers.
-namespace Der Name des WMI-Namespaces.
-class     Die Namen der abgefragten WMI-Klasse(n).
-file      Speichert die Ergebnisse in einer temporären Textdatei ab
           und zeigt diese in Notepad an.
-list      Listet die WMI-Klassen aus dem Namespace auf.
-all       Fragt alle WMI-Klassen ab und speichert die Ergebnisse in einer temporären Textdatei ab.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
ReportMultipleClasses.ps1
Listet die WMI-Clusterklassen des lokalen Computers auf.

ReportMultipleClasses.ps1 -class MSCluster_Network
Gibt detaillierte Informationen über die Konfiguration der
Netzwerkschnittstellen auf dem aktuellen Cluster aus.

ReportMultipleClasses.ps1 -class mscluster_service, mscluster_cluster
Ermittelt Informationen zum Clusterdienst und zum Failover-Cluster
über eine Abfrage der beiden WMI-Klassen: mscluster_service und mscluster_cluster.
Beachten Sie, dass Anführungszeichen nicht erforderlich sind, die
Klassen jedoch durch Kommas voneinander getrennt werden müssen.

ReportMultipleClasses.ps1 -all
Fragt alle WMI-Klassen aus dem Namespace ab und speichert
die Ergebnisse in einer temporären Textdatei.

ReportMultipleClasses.ps1 -list
Erzeugt ein Listing aller WMI-Klassen aus dem Namespace.

ReportMultipleClasses.ps1 -list -file
Erzeugt ein Listing aller WMI-Klassen aus dem Namespace
und speichert die Ergebnisse in einer temporären Textdatei.

ReportMultipleClasses.ps1 -class mscluster_service -file
Fragt die WMI-Klasse mscluster_service auf dem lokalen Computer ab
und speichert die Ergebnisse in einer temporären Textdatei.


ReportMultipleClasses.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
} #end function

function funline($strIN)
```

```
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  {
   $funline = $funline + "="
  }
   Write-Host -ForegroundColor yellow `n$strIN
   Write-Host -ForegroundColor darkYellow $funline
} #Ende der Funktion funhelp

Function funTestNS()
{
 $erroractionpreference="silentlycontinue"
 $objWMI = New-Object -ComObject wbemscripting.swbemlocator
 [void]$objWMI.ConnectServer($computer,$namespace)
 if(!$?)
   {
    Write-host -foregroundcolor red "$namespace ist kein" `
    "gültiger WMI-Namespace auf $computer."

    exit
   }
 $erroractionpreference="continue"
} #Ende der Funktion funtestns

Function funList()
{
 funtestNS
 $wmiClasses = Get-wmiobject -computername $computer `
             -namespace $namespace -list
 $header = "Es gibt $($wmiclasses.count) Klassen" `
         + " im Namespace $namespace auf $computer
          Die WMI-Klassen lauten:
           "
 if($file)
  {
   $header |
   out-file -filepath $tmpfile -append
  }
 ELSE
  {
   $header
  }

  $classes = Get-WmiObject -computername $computer -Namespace `
            $namespace -list |
 Where-Object { $_.name -like '[a-z]*' -and $_.name -notlike 'cim*' } |
 select-object -property name |
 sort-object -property name
 if($file)
  {
   $classes |
   out-file -filepath $tmpfile -append
   notepad $tmpfile
  }
 ELSE
```

```
  {
   $classes
  }
exit
} #Ende der Funktion funlist

function funall()
 {
  funtestNS
  Get-WmiObject -Namespace $namespace -list |
  Where-Object { $_.name -like '[a-z]*' -and `
  $_.name -notlike 'cim*' } |
  foreach-object `
   {
    $_.name ;
    Get-WmiObject -class $_.name -namespace $namespace |
    out-file -filepath $tmpfile -append
   }
  notepad $tmpfile

  exit
 } #Ende der Funktion funall

function funwmi($class)
 {
  funtestNS
  Foreach($objClass in $class)
   {
    Get-WmiObject -class $objclass -computername $computer `
                  -namespace $namespace |
    foreach-object `
     {
      funLine("Querying: $objclass on $computer")
      $_.psobject.properties |
       foreach-object `
        {
         If($_.value)
          {
      if ($_.name -match "__"){}
           ELSE
       {
             $aryProp +=@{ $($_.name)=$($_.value) }
            } #else
          } #if($_.value)
       } #foreach-object $_.psobject.properties
       If($file)
        {
         $($objClass) | out-file -filepath $tmpfile -append
         $aryProp |
         out-file -filepath $tmpfile -append
        }
       ELSE
        {
         $aryProp
        }
       $aryProp = $null
```

```
    } #foreach-object mscluster_node
   } #foreach $objClass
   if($file) { notepad $tmpfile }
  } #Ende der Funktion funwmi

if($help) { "Ausgeben der Hilfeinformationen..." ; funhelp }
if($file) { $tmpfile = [io.path]::getTempfilename() }
if($list) { "Auflisten der Klassen ..." ; funTestNS ; funList }
if($all)  {
            $tmpfile = [io.path]::getTempfilename()
            "Abfragen aller WMI-Klassen im Namespace $namespace ..." ;
            funTestNS ; funAll
          }
if(!$Class) { "Es ist eine Klasse erforderlich ..." ; funhelp }
funwmi($class)
```

Verwalten von Knoten

Nachdem ein Cluster erstellt wurde, müssen Sie möglicherweise Knoten zum Cluster hinzufügen oder aus diesem entfernen. Verwenden Sie hierzu die WMI-Klasse *MSCluster_Cluster* und die Methode *Add()* bzw. die Methode *Evict()*. Diese Vorgänge sind im Skript *AddNodeEvictNode.ps1* veranschaulicht.

Hinzufügen und Entfernen von Knoten

Beginnen Sie das Skript *AddNodeEvictNode.ps1* mit einer *param*-Anweisung und definieren Sie die Befehlszeilenparameter **-computer**, **-namespace** und **-help,** die bereits in anderen Skripts in diesem Kapitel verwendet wurden. Legen Sie außerdem den Parameter **-node** fest, dem kein Standardwert zugewiesen wird. Erstellen Sie darüber hinaus die folgenden festen Parameter: **-add**, **-evict**, **-list**, **-whatif** und **-help**. Diese Parameter vereinfachen die Verwendung des Skripts. Die *param*-Anweisung lautet:

```
param(
      $computer="localhost",
      $namespace="root\mscluster",
      $node,
      [switch]$add,
      [switch]$evict,
      [switch]$list,
      [switch]$whatif,
      [switch]$help
     )
```

Erstellen Sie als Nächstes die Funktion *funhelp()*, um eine Hilfemeldung anzuzeigen, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Die Variable *$helptext* enthält eine *here*-Zeichenfolge, die Informationen zum Skript auflistet, einschließlich einer Beschreibung der Parameter und der Syntax. Erstellen Sie die *here*-Zeichenfolge, weisen Sie diese der Variablen *$helptext* zu und zeigen Sie deren Inhalt an, wenn der Parameter **-help** bei der Skriptausführung angegeben wird. Beenden Sie danach die Ausführung des Skripts. Beispiel:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: AddNodeEvictNode.ps1
```

```
Ermittelt die Knoten eines Clusters, fügt Knoten hinzu oder entfernt Knoten.

PARAMETER:
-computer  Der Name des Computers.
-namespace Der Name des WMI-Namespaces.
-node      Der Name des Clusterknotens.
-add       Gibt an, dass der Knoten zum Cluster hinzugefügt werden soll.
-evict     Gibt an, dass der Knoten aus dem Cluster entfernt werden soll.
-list      Listet die aktuelle Clusterkonfiguration auf.
-whatif    Testet den Befehl ohne Änderungen vorzunehmen.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
AddNodeEvictNode.ps1
Zeigt die fehlenden Parameter an und ruft die Hilfe auf.

AddNodeEvictNode.ps1 -list
Listet die Knotenkonfiguration des Clusters auf.

AddNodeEvictNode.ps1 -node node2 -evict
Entfernt node2 aus dem Cluster.

AddNodeEvictNode.ps1 -node node2 -evict -whatif
Zeigt folgende Informationen an: whatif: Perform
operation evict node node2

AddNodeEvictNode.ps1 -node node2 -add
Fügt node2 zum Cluster hinzu.

AddNodeEvictNode.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
} #Ende der Funktion funhelp
```

Die nächste Funktion ist *funwmi()*. Erstellen Sie die Variable *$class*, in der der WMI-Klassenname *MSCluster_Node* gespeichert wird. Stellen Sie mit dem Cmdlet **Get-WmiObject** eine Verbindung zu WMI her und fügen Sie das Ergebnis der WMI-Abfrage in das Cmdlet **ForEach-Object** ein. Rufen Sie die Eigenschaften des Basisobjekts ab. Wenn die Eigenschaft einen Wert hat und keine Systemeigenschaft ist, erstellen Sie eine Hashtabelle mit den Namen und Werten, und zeigen Sie die Tabelle an. Setzen Sie die Hashtabelle danach zurück, indem Sie dieser *$null* zuweisen und wechseln Sie zum nächsten Element in der Auflistung. Nachdem Sie alle Elemente überprüft haben, beenden Sie das Skript. Die vollständige Funktion *funwmi()* ist wie folgt implementiert:

```
function funwmi()
 {
  $class = "mscluster_node"
  Get-WmiObject -class $class -computername $computer `
                -namespace $namespace |
   foreach-object `
    {
     "Abfragen von: $class auf $computer"
      $_.psobject.properties |
```

```
     foreach-object `
      {
       If($_.value)
        {
    if ($_.name -match "__"){}
         ELSE
     {
          $aryProp +=@{ $($_.name)=$($_.value) }
         } #else
        } #if($_.value)
      } #foreach-object $_.psobject.properties
     $aryProp
     $aryProp = $null
   } #foreach-object mscluster_node
  exit
  } #Ende der Funktion funwmi
```

Nachdem Sie die Funktion *funwmi()* erstellt haben, erstellen Sie die Funktion *funadd()*, die einen Knoten zum Cluster hinzufügt. Weisen Sie als Erstes der Variablen *$class* die Zeichenfolge *"mscluster_cluster"* zu. Stellen Sie eine Verbindung zu WMI über die WMI-Klasse *MSCluster_Cluster* im WMI-Namespace *root\MSCluster* her. Wenn in der Variablen *$objwmi* eine Instanz des Verwaltungsobjekts gespeichert ist, fügen Sie mit der Methode *addnode()* einen Knoten hinzu, der den in der Variablen *$node* angegebenen Namen hat, und beenden Sie das Skript. Die Funktion *funadd()* lautet:

```
function funadd()
{
 $class = "mscluster_cluster"
 $objWMI = Get-wmiobject -namespace $namespace -class $class `
           -computername $computer
 $objwmi.addnode($node)
 exit
} #Ende der Funktion funadd
```

Die nächste Funktion ist *funevict()*, die einen Knoten aus dem Cluster entfernt. Weisen Sie der Variablen *$class* die Zeichenfolge *"mscluster_cluster"* zu und stellen Sie anschließend mit dem Cmdlet **Get-WmiObject** die Verbindung mit WMI her. Greifen Sie unter Verwendung des zurückgegebenen Verwaltungsobjekts auf die Methode *evictnode()* der WMI-Klasse *MSCluster_Cluster* zu. Rufen Sie anschließend die *exit*-Anweisung auf, um das Skript zu beenden. Die Funktion *funevict()* lautet:

```
function funevict()
{
 $class = "mscluster_cluster"
 $objWMI = Get-wmiobject -namespace $namespace -class $class `
           -computername $computer
 $objwmi.evictnode($node)
 exit
} #Ende der Funktion funevict
```

Die Funktion *funwhatif()* modifiziert die auszuführenden Befehle, wenn das Skript mit dem Parameter **-whatif** ausgeführt wird.

Wichtig Wenn Skripts, die mehrere Parameter umfassen, kritische Vorgänge ausführen, beispielsweise einen Knoten aus einem Produktionscluster entfernen, sollten Sie die Funktion *whatif* im Skript unterstützen, um den Befehl anzuzeigen, bevor dieser ausgeführt wird. Diese einfache Methode kann Ihnen in der Zukunft viel Zeit sparen.

Beginnen Sie die Funktion *funwhatif()*, indem Sie den Parameter **-evict** überprüfen. Wenn der Parameter den Wert *True* hat, geben Sie eine Meldung aus, dass der Knoten entfernt wird, und geben Sie den Wert *$node* an. Wenn der Parameter **-add** angegeben wurde, geben Sie eine Meldung aus, dass der in der Variablen *$node* angegebene Knoten hinzugefügt wird. Beenden Sie danach das Skript. Die Funktion *funwhatif()* hat folgenden Aufbau:

```
function funwhatif()
{
 if($evict)
 {
  "whatif: Entfernen des Knotens $node aus dem Cluster."
 }
 if($add)
 {
  "whatif: Hinzufügen des Knotens $node zum Cluster."
 }
 exit
} #Ende der Funktion funwhatif
```

Überprüfen Sie nun die Befehlszeileneingabe.

Wichtig Die Reihenfolge der Überprüfung ist für die Funktionalität des Skripts ausgesprochen wichtig.

Überprüfen Sie zuerst, ob der Parameter **-help** angegeben wurde. Existiert die Variable *$help*, rufen Sie die Funktion *funhelp()* auf. Ist der Parameter **-list** vorhanden, rufen Sie die Funktion *funwmi()* auf. Wenn **-whatif** vorhanden ist, rufen Sie die Funktion *funwhatif()* auf. Ist jedoch die Variable *$node* nicht vorhanden, erstellen Sie eine Fehlermeldung und rufen Sie die Funktion *funhelp()* auf, um die Hilfeinformationen anzuzeigen. Ist der Parameter *$add* vorhanden, rufen Sie die Funktion *funadd()* auf, um den Knoten hinzuzufügen. Suchen Sie nach *$evict* und rufen Sie *funevict()* auf, wenn dieser Schalter angegeben wurde. Suchen Sie anschließend nach einer Kombination der fehlenden Parameter und rufen Sie die Funktion *funhelp()* erneut auf, falls ungültige Kombinationen festgestellt wurden. Dieser Skriptabschnitt ist im Folgenden dargestellt:

```
if($help)   {   "Ausgeben der Hilfeinformationen"   ;   funhelp   }
if($list)   { "Auflisten der Knotenkonfiguration." ; funwmi }
if($whatif) { funwhatif }
if(!$node)  { "Ein Knoten muss angegeben werden." ; funhelp }
if($add)    { "Hinzufügen des Knotens $node" ; funadd }
if($evict)  { "Entfernen des Knotens $node" ; funevict }
if(!$add -or !$evict) { "Erforderlicher Parameter fehlt" ; funhelp }
```

Das vollständige Skript *AddNodeEvictNode.ps1* hat folgenden Inhalt:

AddNodeEvictNode.ps1

```
param(
      $computer="localhost",
      $namespace="root\mscluster",
      $node,
      [switch]$add,
      [switch]$evict,
      [switch]$list,
      [switch]$whatif,
      [switch]$help
     )
```

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: AddNodeEvictNode.ps1
Ermittelt die Knoten eines Clusters, fügt Knoten hinzu oder entfernt Knoten.

PARAMETER:
-computer  Der Name des Computers.
-namespace Der Name des WMI-Namespaces.
-node      Der Name des Clusterknotens.
-add       Gibt an, dass der Knoten zum Cluster hinzugefügt werden soll.
-evict     Gibt an, dass der Knoten aus dem Cluster entfernt werden soll.
-list      Listet die aktuelle Clusterkonfiguration auf.
-whatif    Testet den Befehl ohne Änderungen vorzunehmen.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
AddNodeEvictNode.ps1
Zeigt die fehlenden Parameter an und ruft die Hilfe auf.

AddNodeEvictNode.ps1 -list
Listet die Knotenkonfiguration des Clusters auf.

AddNodeEvictNode.ps1 -node node2 -evict
Entfernt node2 aus dem Cluster.

AddNodeEvictNode.ps1 -node node2 -evict -whatif
Zeigt folgende Informationen an: whatif: Entfernen des Knotens node2 aus dem Cluster.

AddNodeEvictNode.ps1 -node node2 -add
Fügt node2 zum Cluster hinzu.

AddNodeEvictNode.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
} #Ende der Funktion funhelp

function funwmi()
 {
  $class = "mscluster_node"
  Get-WmiObject -class $class -computername $computer `
                -namespace $namespace |
   foreach-object `
    {
     "Abfragen von: $class auf $computer"
      $_.psobject.properties |
      foreach-object `
       {
        If($_.value)
         {
     if ($_.name -match "__"){}
```

```
          ELSE
      {
            $aryProp +=@{ $($_.name)=$($_.value) }
           } #else
         } #if($_.value)
       } #foreach-object $_.psobject.properties
      $aryProp
      $aryProp = $null
    } #foreach-object mscluster_node
  exit
  } #Ende der Funktion funwmi

function funadd()
{
 $class = "mscluster_cluster"
 $objWMI = Get-wmiobject -namespace $namespace -class $class `
           -computername $computer
 $objwmi.addnode($node)
 exit
} #Ende der Funktion funadd

function funevict()
{
 $class = "mscluster_cluster"
 $objWMI = Get-wmiobject -namespace $namespace -class $class `
           -computername $computer
 $objwmi.evictnode($node)
 exit
} #Ende der Funktion funevict

function funwhatif()
{
 if($evict)
 {
  "whatif: Entfernen des Knotens $node aus dem Cluster."
 }
 if($add)
 {
  "whatif: Hinzufügen des Knotens $node zum Cluster."
 }
 exit
} #Ende der Funktion funwhatif

if($help)   { "Ausgeben der Hilfeinformationen"   ;   funhelp   }
if($list)   { "Auflisten der Knotenkonfiguration" ; funwmi }
if($whatif) { funwhatif }
if(!$node)  { "Ein Knoten muss angegeben werden" ; funhelp }
if($add)    { "Hinzufügen des Knotens $node" ; funadd }
if($evict)  { "Entfernen des Knotens $node" ; funevict }
if(!$add -or !$evict) { "Erforderlicher Parameter fehlt" ; funhelp }
```

Entfernen eines Clusterservers

Mit der WMI-Klasse *MSCluster_Cluster* können Sie einen Clusterserver entfernen. Das Skript *RemoveCluster.ps1* enthält ein Beispiel der WMI-Klasse *MSCluster_Cluster.*

Beginnen Sie das Skript *RemoveCluster.ps1* mit einer *param*-Anweisung, die die Parameter **-computer** und **-namespace** mit Standardwerten verwendet. Zeigen Sie mit dem Parameter **-help** die Hilfe an. Die folgenden Parameter haben feste Werte: **-remove** entfernt den Clusterserver, **-list** listet die aktuelle Clusterkonfiguration auf, **-force** überpringt bestimmte Parametertests und **-whatif** testet den Befehl. Die *param*-Anweisung lautet:

```
param(
      $computer="localhost",
      $namespace="root\mscluster",
      [switch]$remove,
      [switch]$list,
      [switch]$force,
      [switch]$whatif,
      [switch]$help
     )
```

Der nächste Schritt umfasst das Erstellen der Funktion *funhelp().* Erstellen Sie als Erstes die Variable *$helptext* und weisen Sie dieser eine *here*-Zeichenfolge mit den Hilfeinformationen zu. Die *here*-Zeichenfolge enthält die Beschreibung, Parameter und Syntax. Nachdem die Variable *$helptext* erstellt wurde, zeigen Sie den Inhalt der Variablen an und beenden Sie das Skript. Die Funktion *funhelp()* hat folgendes Aussehen:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: RemoveCluster.ps1
Entfernt einen Failover-Cluster.

PARAMETER:
-computer  Der Name des Computers.
-namespace Der Name des WMI-Namespaces.
-remove    Entfernt den Cluster.
-list      Zeigt die Clusterinformationen an.
-whatif    Testet den Befehl ohne Änderungen vorzunehmen.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
RemoveCluster.ps1
Weist darauf hin, dass ein Parameter erforderlich ist, und zeigt die Hilfeinformationen an.

RemoveCluster.ps1 -list
Zeigt die Clusterinformationen an.

RemoveCluster.ps1 -remove

Entfernt den Cluster.

RemoveCluster.ps1 -remove -whatif
```

```
Zeigt folgende Meldung an: whatif: Entfernt den Cluster.

RemoveCluster.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
} #Ende der Funktion funhelp
```

Erstellen Sie die Funktion *funlist()*, die eine Verbindung zur WMI-Klasse *MSCluster_Cluster* unter Verwendung des Cmdlets **Get-WmiObject** herstellt. Die Funktion gibt alle Verwaltungsobjekte zurück und zeigt die aktuelle Konfiguration an. Die Funktion *funlist()* beendet danach die Skriptausführung. Die Funktion *funlist()* ist wie folgt implementiert:

```
function funList()
 {
  $class = "mscluster_cluster"
  $objWMI = Get-WmiObject -class $class `
            -computername $computer `
            -namespace $namespace
   $objWMI
  exit
  } #Ende der Funktion funList
```

Die nächste Funktion ist *funcountresource()*, die die Anzahl der Clusterressourcen ermittelt, die derzeit auf dem Server konfiguriert sind. Wenn Clusterressourcen auf dem Server vorhanden sind, geben Sie eine Fehlermeldung aus und beenden Sie das Skript. Um den Clusterserver zu entfernen, verwenden Sie den Parameter **-force**. Die Funktion *funcountresource()* schließt das Cmdlet **Get-WmiObject** und die Parameter in Klammern ein, bevor die Eigenschaft *Count* aufgerufen wird. Die Funktion *funcount-resource()* ist wie folgt implementiert:

```
function funCountResource()
{
 $count = (Get-WmiObject -computername $computer -Namespace `
           $namespace -Class mscluster_resource).count
 if($count -gt 0)
   {
    "Es gibt noch $($count) Ressourcen auf $computer."
    "Sie sollten den Cluster nicht entfernen, solange es noch"
    "bereitgestellte Ressourcen gibt. Wenn Sie sicher sind, dass Sie den Cluster"
    "entfernen möchten, können Sie den Parameter -force verwenden, um diese Überprüfung auszulassen."
    }
    exit
}
```

Die nächste Funktion ist *funremovecluster()*, mit der überprüft wird, ob der Parameter **-force** verwendet wird. Wenn der Parameter vorhanden ist, rufen Sie die Funktion *funcountresource()* auf. Stellen Sie mit dem Cmdlet **Get-WmiObject** eine Verbindung zur WMI-Klasse *MSCluster_Cluster* im WMI-Namespace *root\MSCluster* her. Fügen Sie anschließend über die Eigenschaft *PSBase.Scope.Options.Enable-Privileges* besondere Berechtigungen hinzu und legen Sie die Eigenschaft auf *True* fest. Rufen Sie die Methode *DestroyCluster()* auf und übergeben Sie den booleschen Wert *$true*. Beenden Sie anschließend das Skript. Die Funktion *funremovecluster()* ist wie folgt implementiert:

```
function funRemoveCluster()
 {
  if(!$force) { funCountResource }
  $class = "mscluster_cluster"
  $objWMI = Get-WmiObject -class $class `
            -computername $computer `
            -namespace $namespace
  $objWMI.psbase.Scope.Options.EnablePrivileges = $true
  $objWMI.DestroyCluster($true)
  exit
  } #Ende der Funktion funRemoveCluster
```

Die nächste Funktion ist *funwhatif()*, die den Befehl testet, bevor dieser ausgeführt wird. Verwenden Sie hierzu die Parameter **-computer** und **-namespace** aus der *param*-Anweisung und übergeben Sie die Werte an das Cmdlet **Get-WmiObject**. Fragen Sie die WMI-Klasse *MSCluster_Cluster* ab und geben Sie den Namen des Clusters aus. Der Clusterserver wird mit der Methode *DestroyCluster($true)* entfernt. Die Funktion *funwhatif()* lautet:

```
function funwhatif()
{
 $class = "mscluster_cluster"
 $objWMI = Get-WmiObject -class $class `
              -computername $computer `
              -namespace $namespace
 "whatif: Entfernt den Cluster $($objwmi.name)"
  exit
}
```

Überprüfen Sie nun die Befehlszeilenargumente. Wenn der Parameter **-help** vorhanden ist, rufen Sie die Funktion *funhelp()* auf. Wenn der Parameter **-list** vorhanden ist, rufen Sie die Funktion *funlist()* auf. Wurde der Parameter **-whatif** angegeben, rufen Sie die Funktion *whatif()* auf. Ist der Parameter **-remove** vorhanden, führen Sie die Funktion *funremovecluster()* aus. Wenn die Variablen *$help, $list* und *$remove* nicht vorhanden sind, rufen Sie die Funktion *funhelp()* auf. Dieser Skriptabschnitt ist im Folgenden dargestellt:

```
if($help)   { "Ausgeben der Hilfeinformationen"   ;   funhelp   }
if($list)   { "Aktuelle Konfiguration" ; funlist }
if($whatif) { funwhatif }
if($remove) { funRemoveCluster }
if(!$help -or !$list -or !$remove) { funhelp }
```

Das vollständige Skript *RemoveCluster.ps1* hat folgenden Inhalt:

RemoveCluster.ps1

```
param(
      $computer="localhost",
      $namespace="root\mscluster",
      [switch]$remove,
      [switch]$list,
      [switch]$force,
      [switch]$whatif,
      [switch]$help
     )

function funHelp()
{
$helpText=@"
```

```
BESCHREIBUNG:
NAME: RemoveCluster.ps1
Entfernt einen Failover-Cluster.

PARAMETER:
-computer  Der Name des Computers.
-namespace Der Name des WMI-Namespaces.
-remove    Entfernt den Cluster.
-list      Zeigt die Clusterinformationen an.
-whatif    Testet den Befehl ohne Änderungen vorzunehmen.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
RemoveCluster.ps1
Weist darauf hin, dass ein Parameter erforderlich ist, und zeigt die Hilfeinformationen an.

RemoveCluster.ps1 -list
Zeigt die Clusterinformationen an.

RemoveCluster.ps1 -remove
Entfernt den Cluster.

RemoveCluster.ps1 -remove -whatif
Zeigt folgende Meldung an: whatif: Entfernt den Cluster.

RemoveCluster.ps1 -help
Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
} #Ende der Funktion funhelp

function funList()
 {
  $class = "mscluster_cluster"
  $objWMI = Get-WmiObject -class $class `
            -computername $computer `
            -namespace $namespace
   $objWMI
  exit
  } #Ende der Funktion funList

function funCountResource()
{
 $count = (Get-WmiObject -computername $computer -Namespace `
          $namespace -Class mscluster_resource).count
 if($count -gt 0)
   {
    "Es gibt noch $($count) Ressourcen auf $computer."
    "Sie sollten den Cluster nicht entfernen, solange es noch"
    "bereitgestellte Ressourcen gibt. Wenn Sie sicher sind, dass Sie den Cluster"
    "entfernen möchten, können Sie den Parameter -force verwenden, um diese Überprüfung auszulassen."
    }
    exit
}
```

```
function funRemoveCluster()
 {
  if(!$force) { funCountResource }
  $class = "mscluster_cluster"
  $objWMI = Get-WmiObject -class $class `
            -computername $computer `
            -namespace $namespace
  $objWMI.psbase.Scope.Options.EnablePrivileges = $true
  $objWMI.DestroyCluster($true)
  exit
  } #Ende der Funktion funRemoveCluster

function funwhatif()
{
 $class = "mscluster_cluster"
 $objWMI = Get-WmiObject -class $class `
              -computername $computer `
              -namespace $namespace
 "whatif: Entfernt den Cluster $($objwmi.name)."
  exit
}

if($help)   { "Ausgeben der Hilfeinformationen"   ;   funhelp   }
if($list)   { "Aktuelle Konfiguration" ; funlist }
if($whatif) { funwhatif }
if($remove) { funRemoveCluster }
if(!$help -or !$list -or !$remove) { funhelp }
```

Zusammenfassung

In diesem Kapitel wurden die Arbeit mit einem Windows Server 2008-Failovercluster beschrieben. Sie haben als Erstes die WMI-Klassen identifiziert, die zum Verwalten von Windows Server 2008-Failoverclustern verwendet werden. Anschließend haben Sie die aktuelle Clusterkonfiguration überprüft und die Knotenkonfiguration abgerufen. Außerdem wurde ein leistungsfähiges Skript erklärt, das mehrere Klassen gleichzeitig abfragt und die Ergebnisse in einer Textdatei speichert. Das Kapitel wurde mit dem Hinzufügen und Entfernen von Clusterknoten sowie dem Entfernen eines Clusterservers abgeschlossen.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

KAPITEL 15

Verwalten der Internetinformationsdienste

Nach Abschluss dieses Kapitels können Sie:

- Die IIS-Konfigurationsinformationen überprüfen
- Eine neue Website erstellen
- Eine vorhandene Website ändern
- Eine Website sichern
- Die IIS-Optionen ändern

Auf der Begleit-CD Alle Skripts in diesem Kapitel befinden sich auf der Begleit-CD im Ordner *\Scripts\Chapter15*.

Aktivieren der IIS-Verwaltung

Die meisten Optionen in den Microsoft Internetinformationsdiensten sind optional. Sie können IIS ohne die Verwaltungsfunktionen installieren. Dies ist für einen eigenständigen Webserver hilfreich, der keine umfassende Verwaltung erfordert. Wie in Abbildung 15.1 dargestellt, wird IIS auf Windows Server 2008 als Serverrolle installiert.

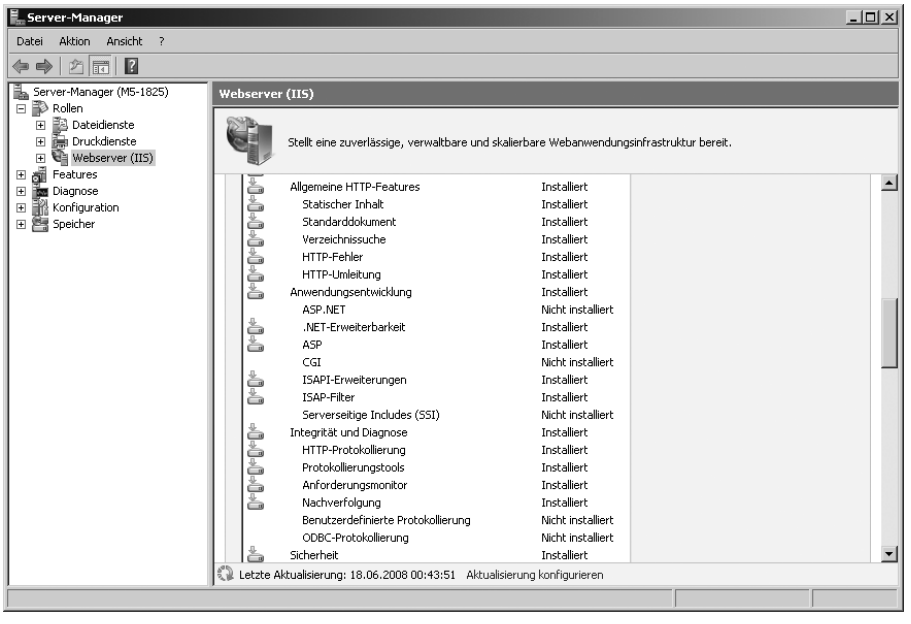


Abbildung 15.1 Zum Installieren von IIS als Serverrolle auf Windows Server 2008 stehen Ihnen zahlreiche Optionen zur Verfügung

Sie haben zwei Möglichkeiten die Remoteverwaltung in Windows PowerShell zu aktivieren. Sie können die Kompatibilität mit WMI für IIS 6 oder die IIS-Verwaltungsskripts und -tools hinzufügen. Wenn Sie die IIS-Verwaltungsskripts und -tools installieren, können Sie auf einige neue WMI-Klassen (Windows Management Instrumentation) im WMI-Namespace *root\WebAdministration* zugreifen. Wenn Sie die Kompatibilität mit WMI für IIS 6 installieren, können Sie die gleichen WMI-Klassen für die Verwaltung von IIS 6 im WMI-Namespace *root\MicrosoftIISv2* verwenden. Diese Klassen können sowohl für IIS 7 als auch für IIS 6 eingesetzt werden.

Auswählen der geeigneten IIS 7 WMI-Klassen

Die Auswahl einer geeigneten WMI-Klasse ist schwierig, da der WMI-Namespace *root\WebAdministration* zahlreiche Klassen umfasst. Ein Windows PowerShell-Skript kann diese Aufgabe vereinfachen. Das Skript *FindIISClasses.ps1* sucht im Namespace *IIS 7* nach WMI-Klassen, die mit den angegebenen Suchkriterien übereinstimmen. Die Funktion überprüft die Namen aller Klassen und gibt Übereinstimmungen in einer Liste zurück. Wenn Sie häufig mit IIS 7 WMI arbeiten, können Sie diese Funktion in Ihr Profil einfügen. Das Skript *FindIISClasses.ps1* ist wie folgt aufgebaut:

FindIISClasses.ps1

```
function funIIS($strIN)
 {
  Get-WmiObject -Namespace root\webadministration -list |
  where-object { $_.name -match $strIN }
 }

funIIS("site")
```

Weiterführende Informationen Weitere Informationen zu den Windows PowerShell-Profilen finden Sie in *Microsoft Windows PowerShell Step by Step* (Microsoft Press, 2007).

Überprüfen der IIS-Konfiguration

Nachdem Sie die IIS-Verwaltungstools installiert haben, sollten Sie die Konfiguration des Servers überprüfen. Überprüfen Sie unter anderem die auf dem IIS-Server konfigurierten Websites und gegebenenfalls die Anwendungspools.

Überprüfen der Websiteinformationen

Als Erstes müssen Sie unter Verwendung der IIS-Verwaltungskonsole herausfinden, welche Websites auf dem Server gehostet werden. Die neue und verbesserte IIS-Verwaltungskonsole umfasst viele nützliche Features, einschließlich eine neue Benutzeroberfläche (siehe Abbildung 15.2).

Darüber hinaus können Sie aber auch mit einem Skript, z.B. *GetSites.ps1*, das die WMI-Klasse *Site* im WMI-Namespace *root\WebAdministration* verwendet, die auf einem Server gehosteten Websites anzeigen.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

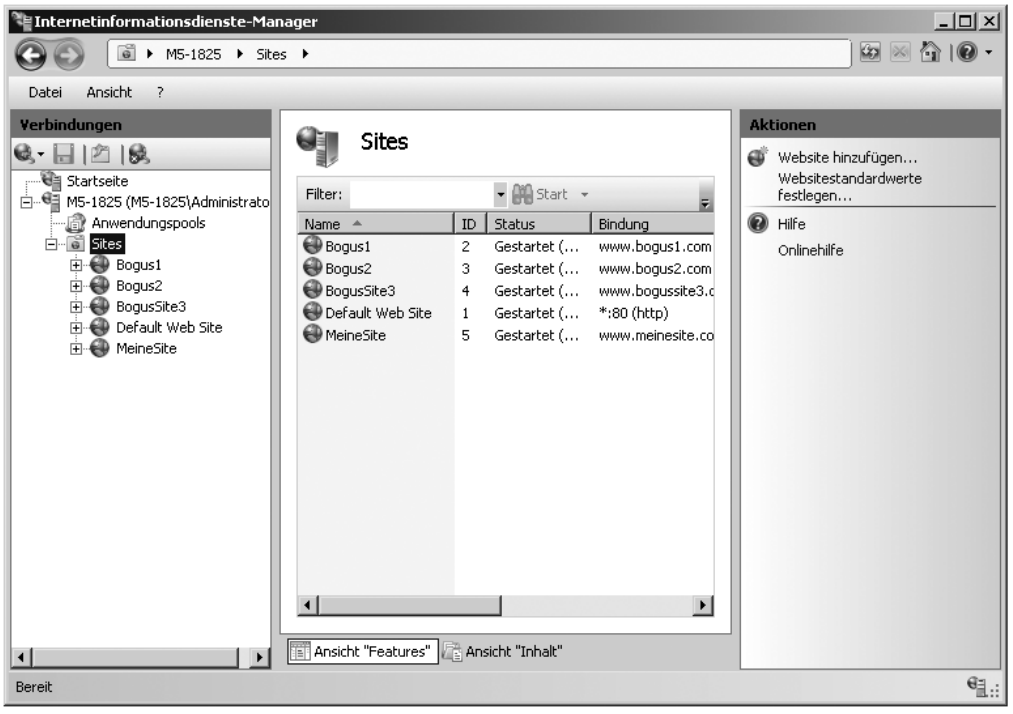


Abbildung 15.2 In der IIS-Verwaltungskonsole angezeigte Websites

Das Skript *GetSites.ps1* beginnt mit einer *param*-Anweisung. Diese Anweisung ist relativ einfach, da nur zwei Parameter festgelegt werden: **-computer** für den Zielcomputer und **-help** zum Anzeigen der Skriptsyntax. Codezeile:

```
param($computer="localhost", [switch]$help)
```

Die nächste Funktion ist *funhelp()*, die eine *here*-Zeichenfolge verwendet. Da das Skript nur zwei Parameter unterstützt, **-help** und **-computer**, ist keine umfangreiche Hilfe erforderlich. Nachdem Sie die *here*-Zeichenfolge definiert und der Variablen *$helptext* zugewiesen haben, geben Sie den Inhalt der Variablen aus und beenden die Skriptausführung. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GetSites.ps1
Ermittelt die Liste der Websites auf dem lokalen Computer oder einem Remotecomputer.

PARAMETER:
-computer  Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
GetSites.ps1

Ermittelt die Liste der Websites auf dem lokalen Computer.
```

```
GetSites.ps1 -computer "webserverII"

Ermittelt die Liste der Websites auf einem Webserver namens webserverII.

GetSites.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Wenn das Skript mit dem Parameter **-help** gestartet wird, ist die Variable *$help* im Stack vorhanden. Geben Sie in diesem Fall die Hilfeinformationen aus. Codezeile:

```
if($help)      {      "Ausgeben der Hilfeinformationen..."      ;      funHelp      }
```

Rufen Sie anschließend die Websiteinformationen aus WMI ab. Greifen Sie auf den WMI-Namespace *root\WebAdministration* zu und rufen Sie alle Objekte ab, die sich auf das Websiteobjekt beziehen. Fügen Sie das resultierende Objekt in das Cmdlet **Format-Table** ein, um die Ausgabe übersichtlicher zu formatieren. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
Get-WmiObject -Namespace root\webadministration `
              -computername $computer -class site |
format-table -property name
```

Das vollständige Skript *GetSites.ps1* hat folgenden Inhalt:

GetSites.ps1

```
param($computer="localhost", [switch]$help)

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GetSites.ps1
Ermittelt die Liste der Websites auf dem lokalen Computer oder einem Remotecomputer.

PARAMETER:
-computer  Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
GetSites.ps1

Ermittelt die Liste der Websites auf dem lokalen Computer.

GetSites.ps1 -computer "webserverII"

Ermittelt die Liste der Websites auf einem Webserver namens webserverII.

GetSites.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
```

```
 exit
}

if($help)      { "Ausgeben der Hilfeinformationen..." ; funHelp }

Get-WmiObject -Namespace root\webadministration `
              -computername $computer -class site |
format-table -property name
```

Überprüfen der Anwendungspools

Anwendungspools wurden in IIS Version 6.0 eingeführt. Sie können über die IIS-Verwaltungskonsole auf die Anwendungspools zugreifen (siehe Abbildung 15.3).

Sie können auch die neue WMI-Klasse *ApplicationPool* im WMI-Namespace *root\WebAdministration* verwenden, um mit Anwendungspools zu arbeiten. Das Skript *GetAppPool.ps1* verwendet die WMI-Klasse *ApplicationPool*, um die Informationen zu den Anwendungspools auf dem Server zusammenzustellen.

Das Skript GetAppPool.ps1 beginnt mit einer *param*-Anweisung. Die *param*-Anweisung definiert die beiden Parameter: **-computer** mit dem Standardwert *localhost* und **-help**, für den kein Wert angegeben werden kann. Die *param*-Anweisung lautet:

```
param($computer="localhost", [switch]$help)
```

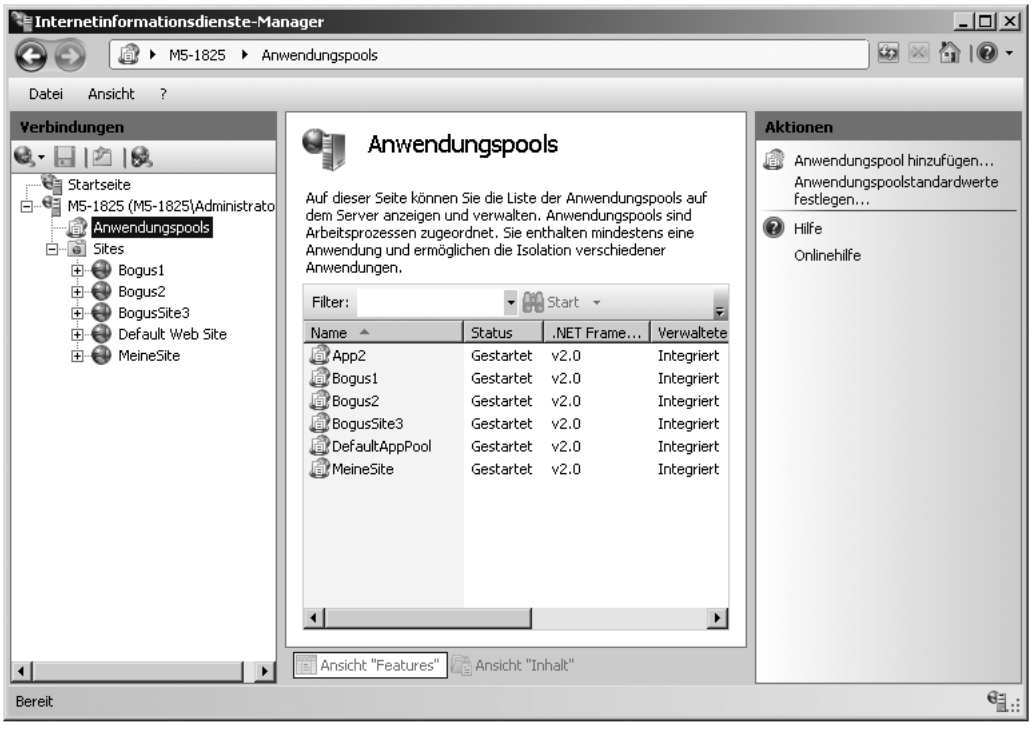


Abbildung 15.3 Anwendungspools in der IIS-Verwaltungskonsole

Erstellen Sie die Funktion *funhelp()*, um eine Hilfemeldung anzuzeigen, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Die Funktion *funhelp()* beginnt mit dem Schlüsselwort *function* gefolgt vom Funktionsnamen. Da die Funktion keine Parameter umfasst, sind die Klammern leer. Deklarieren Sie im Codeblock die Variable *$helptext,* und erstellen Sie eine *here*-Zeichenfolge. Die *here*-Zeichenfolge beginnt mit @" und endet mit "@. Sie müssen in der *here*-Zeichenfolge keine Anführungszeichen eingeben. Der Vorteil einer *here*-Zeichenfolge ist, dass die Ausgabe des Skripts auf dem Bildschirm angezeigt wird. Sie können leere Zeilen überspringen oder einbeziehen, aber der gesamte eingegebene Text wird als eine Zeichenfolge behandelt. Definieren Sie drei Abschnitte: Beschreibung, Parameter und Syntax. Nachdem Sie die *here*-Zeichenfolge erstellt haben, zeigen Sie den Inhalt der Variablen *$helptext* an und beenden Sie das Skript. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GetAppPool.ps1
Ermittelt die Anwendungspools auf einem lokalen Computer oder einem Remotecomputer.

PARAMETER:
-computer  Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
GetAppPool.ps1

Ermittelt die Anwendungspools auf dem lokalen Computer.

GetAppPool.ps1 -computer "webserverII"

Ermittelt die Anwendungspools auf einem Webserver namens webserverII.

GetAppPool.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Sie müssen das Skript ausführen oder die Hilfe anzeigen. Stellen Sie sicher, dass die Variable *$help* vorhanden ist, um diese Entscheidung zu treffen. Existiert die Variable *$help*, wurde das Skript mit dem Parameter **-help** ausgeführt. Geben Sie in diesem Fall eine Statusmeldung aus und rufen Sie die Funktion *funhelp()* auf. Codezeile:

```
if($help)      { "Ausgeben der Hilfeinformationen..." ; funHelp }
```

Der nächste Skriptabschnitt greift auf WMI zu, um eine Instanz der WMI-Klasse *ApplicationPool* abzurufen. Verwenden Sie das Cmdlet **Get-Object** und geben Sie den WMI-Namespace *root\WebAdministration* an. In diesem Verzeichnis sind die neuen WMI-Klassen für IIS 7.0 gespeichert. Geben Sie dem Benutzer die Möglichkeit, den Parameter **-computername** anzugeben, um einen neuen Computernamen im Skript verwenden zu können. Fügen Sie das resultierende Verwaltungsobjekt in das nächste Cmdlet ein und geben Sie das Graviszeichen an, um die Zeile in zwei Zeilen aufzuteilen. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
Get-WmiObject -Namespace root\webadministration `
              -computername $computer -Class applicationpool |
```

Der nächste Abschnitt des Skripts erstellt die Ausgabe. Verwenden Sie das eingefügte Objekt vom Cmdlet **Get-WmiObject** und übergeben Sie das Objekt an das Cmdlet **Format-Table**. Wählen Sie die Eigenschaften *Name* und *AutoStart* aus und erstellen Sie unter Verwendung des Graviszeichens eine Hashtabelle.

Tipp Eine Frage, die mir häufig gestellt wird, ist: „Wie kann ich mit dem Cmdlet **Format-Table** einen anderen Tabellenheader erstellen?“ Die Antwort ist, eine Hashtabelle zu verwenden und für die Bezeichnung und den Ausdruck andere Werte anzugeben. Die Antworten auf andere häufig gestellte Fragen finden Sie in Abhang C „Häufig gestellte Fragen“.

Die Hashtabelle erlaubt Ihnen das Ändern der Tabellenbezeichnung *ManagedRuntimeVersion*. Beispielsweise können Sie die dritte Spalte in *.Net Version* umbenennen. Geben Sie hierzu als Erstes einen Wert für die Bezeichnung ein (in diesem Fall *.Net Version)*. Geben Sie anschließend einen Wert für den Ausdruck an. Legen Sie den Wert auf den aktuellen Wert der Eigenschaft *ManagedRuntimeVersion* fest. Vervollständigen Sie den Befehl mit der Eigenschaft *QueueLength* und dem Parameter **-autosize**. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
format-table -property name, autostart, `
       @{
         Label = ".Net Version" ;
         Expression = { $_.ManagedRuntimeVersion }
        }, `
          QueueLength -autosize
```

Das vollständige Skript *GetAppPool.ps1* hat folgenden Inhalt:

GetAppPool.ps1

```
param($computer="localhost", [switch]$help)

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GetAppPool.ps1
Ermittelt die Anwendungspools auf einem lokalen Computer oder einem Remotecomputer.

PARAMETER:
-computer  Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
GetAppPool.ps1

Ermittelt die Anwendungspools auf dem lokalen Computer.

GetAppPool.ps1 -computer "webserverII"

Ermittelt die Anwendungspools auf einem Webserver namens webserverII.

GetAppPool.ps1 -help
```

```
Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}

if($help)      { "Ausgeben der Hilfeinformationen..." ; funHelp }

Get-WmiObject -Namespace root\webadministration `
              -computername $computer -Class applicationpool |
format-table -property name, autostart, `
       @{
         Label = ".Net Version" ;
         Expression = { $_.ManagedRuntimeVersion }
        }, `
          QueueLength -autosize
```

Überprüfen der Standardwerte eines Anwendungspools

Die Funktionalität der Anwendungspools wird auf Webserverebene von mehreren Standardwerten gesteuert. Diese Werte legen das Verhalten der Anwendungspools unter anderem für den automatischen Start und das Ausführen von 32-Bit-Anwendungen auf 64-Bit-Hardware fest. Die Standardwerte sind normalerweise für kleine Anwendungen ausreichend. Achten Sie für spezialisiertere Anwendungen jedoch insbesondere auf die Einstellungen, die alle Anwendungspools beeinflussen. Die Standardwerte wirken sich auch auf die CPU-Auslastung aus.

Sie können beispielsweise mit dem Skript *GetApplicationPoolDefaults.ps1* die WMI-Klasse *Server* im WMI-Namespace *root\WebAdministration* verwenden, um die Standardwerte für alle Anwendungspools auf einem gegebenen Server abzufragen.

Das Skript *GetApplicationPoolDefaults.ps1* beginnt mit einer *param*-Anweisung und zwei Parametern, **-computer** und **-help**. Der Parameter **-computer** hat den Standardwert *localhost* und für den Parameter **-help** kann kein Wert angegeben werden. Codezeile:

```
param($computer="localhost", [switch]$help)
```

Implementieren Sie nun die Funktion *funhelp()*, um die Hilfe für das Skript anzuzeigen. Die Funktion zeigt die Verwendungsinformationen für das Skript und Syntaxbeispiele an. Anschließend ruft das Skript die *exit*-Anweisung auf, um die Skriptausführung zu beenden. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GetApplicationPoolDefaults.ps1
Erstellt eine Liste mit den Standardeinstellungen für die Anwendungspools
auf dem lokalen Computer oder einem Remotecomputer.

PARAMETERS:
-computer  Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-help      Zeigt dieses Hilfethema an.
```

```
SYNTAX:
GetApplicationPoolDefaults.ps1

Erstellt eine Liste mit den Standardeinstellungen für die Anwendungspools auf dem lokalen Computer.

GetApplicationPoolDefaults.ps1 -computer "webserverII"

Erstellt eine Liste mit den Standardeinstellungen für die Anwendungspools
auf einem Webserver namens webserverII.

GetApplicationPoolDefaults.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Implementieren Sie nun die Skriptanweisung, die unmittelbar nach der *param*-Codezeile ausgeführt wird und falls erforderlich den Hilfetext anzeigt. Überprüfen Sie, ob die Variable *$help* vorhanden ist. Rufen Sie anschließend die Funktion *funhelp()* auf. Wenn die Variable *$help* nicht existiert, hat die Codezeile keine Auswirkungen auf das Skriptverhalten. Codezeile:

```
if($help) { "Ausgeben der Hilfeinformationen..." ; funHelp }
```

Greifen Sie auf WMI zu, indem Sie das Cmdlet **Get-WmiObject** verwenden. Da die WMI-Klassen in *IIS 7.0* in einem nicht dem Standard entsprechenden WMI-Namespace gespeichert sind, müssen Sie den Parameter **-namespace** angeben und das Skript für die Verwendung des WMI-Namespaces *root\WebAdministration* konfigurieren. Geben Sie den Parameter **-computername** an, um einen anderen Computer festzulegen, und wählen Sie die WMI-Klasse *Server* aus. Dieser Skriptabschnitt ist im Folgenden dargestellt:

```
$server = Get-WmiObject -Namespace root\webadministration `
        -class server -computername $computer
```

Der nächste Abschnitt des Skripts generiert die Ausgabe. Wenn Sie die WMI-Klasse *Server* abfragen, werden die Standardwerte der Anwendungspools als eine Instanz der WMI-Klasse *ApplicationPoolDefaults* zurückgegeben. Diese Instanz wird in das zurückgegebene Objekt eingefügt, das die WMI-Klasse *Server* enthält. Die angezeigten Eigenschaften (beispielsweise *AutoStart und Enable32BitAppOnWin64*) sind Eigenschaften der WMI-Klasse *ApplicationPoolDefaults* , nicht der Klasse *Server.* Codeabschnitt:

```
$server.ApplicationPoolDefaults.autostart
$server.ApplicationPoolDefaults.Enable32BitAppOnWin64
$server.ApplicationPoolDefaults.ManagedPipelineMode
$server.ApplicationPoolDefaults.ManagedRuntimeVersion
$server.ApplicationPoolDefaults.Name
$server.ApplicationPoolDefaults.PassAnonymousToken
$server.ApplicationPoolDefaults.QueueLength
```

Nachdem Sie die Informationen aus der WMI-Klasse *ApplicationPoolDefaults* abgerufen haben, verwenden Sie die Klasse *CPU*, um weitere Informationen zu ermitteln. Da die CPU-Informationen für Anwendungspools als eine Instanz der WMI-Klasse *CPU* zurückgegeben werden, müssen Sie einen weiteren WMI-Klassennamen angeben. Die Variable *$server* enthält nur eine Instanz der WMI-Klasse

Server. Die nächste WMI-Klasse ist *ApplicationPoolDefaults* und über diese können Sie auf die *CPU*-Klasse zugreifen. Das klingt komplizierter, als es wirklich ist. Der Codeabschnitt ist im Folgenden dargestellt:

```
$server.ApplicationPoolDefaults.cpu.Action
$server.ApplicationPoolDefaults.cpu.limit
$server.ApplicationPoolDefaults.cpu.resetinterval
$server.ApplicationPoolDefaults.cpu.SmpAffinitized
$server.ApplicationPoolDefaults.cpu.SmpAffinityMask
```

Das vollständige Skript *GetApplicationPoolDefaults.ps1* hat folgenden Aufbau:

GetApplicationPoolDefaults.ps1

```
param($computer="localhost", [switch]$help)

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GetApplicationPoolDefaults.ps1
Erstellt eine Liste mit den Standardeinstellungen für die Anwendungspools
auf dem lokalen Computer oder einem Remotecomputer.

PARAMETER:
-computer  Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
GetApplicationPoolDefaults.ps1

Erstellt eine Liste mit den Standardeinstellungen für die Anwendungspools auf dem lokalen Computer.

GetApplicationPoolDefaults.ps1 -computer "webserverII"

Erstellt eine Liste mit den Standardeinstellungen für die Anwendungspools
auf einem Webserver namens webserverII.

GetApplicationPoolDefaults.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}

if($help) { "Ausgeben der Hilfeinformationen ..." ; funHelp }

$server = Get-WmiObject -Namespace root\webadministration `
        -class server -computername $computer
$server.ApplicationPoolDefaults.autostart
$server.ApplicationPoolDefaults.Enable32BitAppOnWin64
$server.ApplicationPoolDefaults.ManagedPipelineMode
$server.ApplicationPoolDefaults.ManagedRuntimeVersion
$server.ApplicationPoolDefaults.Name
$server.ApplicationPoolDefaults.PassAnonymousToken
```

```
$server.ApplicationPoolDefaults.QueueLength
$server.ApplicationPoolDefaults.cpu.Action
$server.ApplicationPoolDefaults.cpu.limit
$server.ApplicationPoolDefaults.cpu.resetinterval
$server.ApplicationPoolDefaults.cpu.SmpAffinitized
$server.ApplicationPoolDefaults.cpu.SmpAffinityMask
```

Überprüfen der Websiteeinschränkungen

Sie können für Websites mehrere Einschränkungen festlegen, um sicherzustellen, dass die Websites nicht alle Ressourcen auf dem Server belegen. Überprüfen Sie die maximale Anzahl der Verbindungen, die für eine Website festgelegt sind. Eine hohe Verbindungsanzahl kann einen Server mit nicht ausreichender Kapazität in die Knie zwingen. Sie sollten auch den Zeitüberschreitungswert für die Verbindungen überprüfen. Dieser Wert ist ein zweischneidiges Schwert: Bei einem zu niedrigen Wert werden die Verbindungen eventuell zu früh getrennt und die Clients müssen neue Verbindungen herstellen. Dieser Vorgang erzeugt Netzwerkverkehr und belegt Prozessorzeit. Wenn der Wert zu hoch ist, belegen die Verbindungen zu viel Arbeitsspeicher. Die Lösung besteht im Testen der Anwendungen.

Eine weitere Einschränkung, die Sie überprüfen sollten, betrifft die von der Website belegte Bandbreite. Wenn beispielsweise fünf Websites auf einem Server gehostet werden und Sie die Bandbreite gleichmäßig auf diese Websites verteilen möchten, müssen Sie jeder Website 20 Prozent der verfügbaren Bandbreite zuweisen. Sie können die Zeit- und Bandbreiteneinschränkungen in der IIS-Verwaltungskonsole anzeigen (siehe Abbildung 15.4).

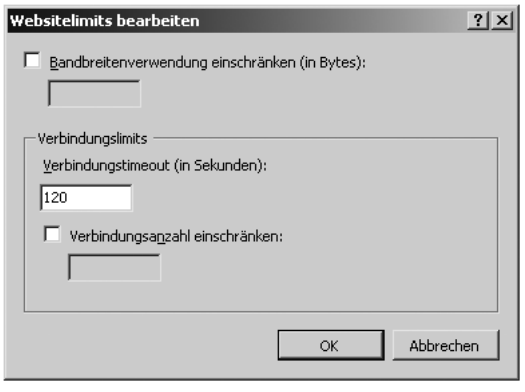


Abbildung 15.4 Der Zeitüberschreitungswert für eine Website in der IIS-Verwaltungskonsole

Wenn Sie über mehrere Webserver verfügen, können Sie die Websiteeinschränkungen mit einem Skript ermitteln. Skripts vereinfachen das Zusammenstellen von Informationen, sind schneller und ermöglichen das Speichern der Ergebnisse in einer Textdatei oder Datenbank.

Das Skript *GetSiteLimits.ps1* beginnt mit einer *param*-Anweisung. Die *param*-Anweisung definiert die beiden Parameter: **-computer** für den Zielcomputer und **-help** zum Anzeigen der Hilfe. Für den Parameter **-help** kann kein Wert angegeben werden. Der Parameter **-computer** ist standardmäßig auf den lokalen Computer festgelegt. Codezeile:

```
param($computer="localhost", [switch]$help)
```

Erstellen Sie die Funktion *funhelp()*, um eine Hilfemeldung anzuzeigen, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Definieren Sie den Hilfetext mit einer *here*-Zeichenfolge und weisen

Sie diese der Variablen *$helptext* zu. Nachdem Sie die *here*-Zeichenfolge erstellt haben, zeigen Sie den Inhalt der Variablen an und beenden Sie das Skript. Der Hilfetext besteht aus drei Abschnitten: Beschreibung, Parameter und Syntax. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GetSiteLimits.ps1
Erstellt eine Liste mit den Siteeinschränkungen auf dem lokalen Computer oder einem Remotecomputer.

PARAMETERS:
-computer  Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
GetSiteLimits.ps1

Erstellt eine Liste mit den Siteeinschränkungen auf dem lokalen Computer.

GetSiteLimits.ps1 -computer "webserverII"

Erstellt eine Liste mit den Siteeinschränkungen auf einem Webserver namens webserverII.

GetSiteLimits.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
```

Bestimmen Sie anschließend, ob Sie die Hilfe anzeigen müssen. Überprüfen Sie hierzu, ob die Variable *$help* existiert. Die Variable *$help* ist nur vorhanden, wenn das Skript mit dem Parameter **-help** ausgeführt wurde. Wird die Variable gefunden, geben Sie eine Statusmeldung aus und rufen Sie die Funktion *funhelp()* auf. Beachten Sie, dass das Semikolon zwischen der Meldung und dem Funktionsaufruf zwei separate Befehle in der gleichen Zeile trennt. Codezeile:

```
if($help) { "Ausgeben der Hilfeinformationen..." ; funHelp }
```

Greifen Sie mit dem Cmdlet **Get-WmiObject** auf den WMI-Namespace *root\WebAdministration* zu. Verwenden Sie hierzu den Parameter **-namespace**. Geben Sie den Parameter **-computername** an, um eine Verbindung mit einem Remotecomputer zu ermöglichen. Geben Sie außerdem im Parameter **-class** die WMI-Klasse *Server* an. Das resultierende Verwaltungsobjekt wird in der Variablen *$server* gespeichert. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
$server = Get-WmiObject -Namespace root\webadministration `
            -computername $computer -class server
```

Die Ausgabe des Skripts ist ungewöhnlich, aber veranschaulicht ein Feature der neuen mit IIS 7.0 bereitgestellten WMI-Klassen. Um den Wert der Eigenschaft *MaxConnections* anzuzeigen, müssen Sie eine WMI-Zwischenklasse verwenden.

Wichtig Die neuen WMI-Klassen von IIS 7 unterstützen eine höhere Vererbungsebene. Beim Abfragen einer Kernklasse werden häufig eingebettete Objekte zurückgegeben. Das Cmdlet **Get-Member** kann diese Objekte nicht verarbeiten. Weitere Informationen finden Sie in der Windows SDK-Dokumentation.

Sie müssen mehrere WMI-Zwischenklassen verwenden. Um die Websitestandardwerte zu ermitteln, verwenden Sie die Klasse *SiteDefaults*. Mit der Klasse *SiteDefaults* können Sie die Website-einschränkungen anzeigen. Die Klasse *Limits* umfasst die Eigenschaften *MaxConnections, ConnectionTimeout* und *MaxBandwidth*. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
$server.SiteDefaults.limits.maxconnections
$server.SiteDefaults.limits.ConnectionTimeout
$server.SiteDefaults.limits.MaxBandwidth
```

Das vollständige Skript GetSiteLimits.ps1 hat folgenden Inhalt:

GetSiteLimits.ps1

```
param($computer="localhost", [switch]$help)

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GetSiteLimits.ps1
Erstellt eine Liste mit den Siteeinschränkungen auf dem lokalen Computer oder einem Remotecomputer.

PARAMETER:
-computer  Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
GetSiteLimits.ps1

Erstellt eine Liste mit den Siteeinschränkungen auf dem lokalen Computer.

GetSiteLimits.ps1 -computer "webserverII"

Erstellt eine Liste mit den Siteeinschränkungen auf einem Webserver namens webserverII.

GetSiteLimits.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help)      { "Ausgeben der Hilfeinformationen ..." ; funHelp }


$server = Get-WmiObject -Namespace root\webadministration `
            -computername $computer -class server
$server.SiteDefaults.limits.maxconnections
$server.SiteDefaults.limits.ConnectionTimeout
$server.SiteDefaults.limits.MaxBandwidth
```

Auflisten der virtuellen Verzeichnisse

Virtuelle Verzeichnisse werden von IIS zum Zuordnen physischer Verzeichnisse verwendet. Jede Webanwendung in IIS 7 hat ein virtuelles Stammverzeichnis, das die Webanwendung dem physischen Verzeichnis zuordnet. Eine Webanwendung kann mehr als ein virtuelles Verzeichnis umfassen. Sie können die Informationen zu den virtuellen Verzeichnissen über die IIS-Verwaltungskonsole oder die WMI-Klasse *VirtualDirectory* im WMI-Namespace *root\WebAdministration* abrufen.

Beginnen Sie das Skript *ListVirtualDirectory.ps1* mit einer *param*-Anweisung. Die *param*-Anweisung legt das Skriptverhalten zur Laufzeit fest.

Tipp Jedem Parameter muss ein Wert zugewiesen sein, da das Skript ansonsten einen Fehler generiert. Sie können dieses Verhalten auf zwei Arten steuern. Weisen Sie einen Standardwert zu oder definieren Sie mit der *[switch]*-Einschränkung einen festen Parameter.

Die *param*-Anweisung akzeptiert zwei Argumente. Der Parameter **-computer** ist mit dem Standardwert *localhost* konfiguriert, der auf den lokalen Computer verweist. Der zweite Parameter ist **-help**, um die Funktion *funhelp()* aufzurufen. Für den Parameter **-help** kann kein Wert angegeben werden, da dieser ein boolescher Wert mit fester Zuordnung ist, beispielsweise *True/False* oder *0/-1*. Die *param*-Anweisung lautet:

```
param($computer="localhost", [switch]$help)
```

Die Funktion *funhelp()* zeigt die Hilfe an, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Deklarieren Sie die Variable *$helptext* und erstellen Sie eine *here*-Zeichenfolge. Die *here*-Zeichenfolge besteht aus drei Abschnitten: Beschreibung, Parameter und Syntax. Nachdem die Variable *$helptext* definiert wurde, geben Sie den Wert der Variablen *$helptext* aus und beenden Sie das Skript. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ListVirtualDirectory.ps1
Erstellt eine Liste mit den virtuellen Verzeichnissen auf dem lokalen Computer oder einem Remotecomputer.

PARAMETER:
-computer  Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
ListVirtualDirectory.ps1

Erstellt eine Liste mit den virtuellen Verzeichnissen auf dem lokalen Computer.

ListVirtualDirectory.ps1 -computer "webserverII"

Erstellt eine Liste mit den virtuellen Verzeichnissen auf einem Webserver namens webserverII.

ListVirtualDirectory.ps1 -help

Zeigt das Hilfethema für dieses Skript an.
```

```
"@
$helpText
exit
}
```

Nach der Funktion *funhelp()* folgt der Skriptabschnitt, der unmittelbar nach der *param*-Anweisung ausgeführt wird. Die Funktionsdefinition wird übersprungen, bis sie aufgerufen wird. Verwenden Sie eine *if*-Anweisung, um die Variable *$help* zu überprüfen. Wenn die Variable existiert, geben Sie eine Statusmeldung aus und rufen Sie die Funktion *funhelp()* auf. Die Codezeile zum Überprüfen der Variablen *$help* und Aufrufen der Funktion *funhelp()* lautet:

```
if($help) { "Ausgeben der Hilfeinformationen ..." ; funHelp }
```

Greifen Sie nun mit dem Cmdlet **Get-WmiObject** auf den WMI-Namespace *root\WebAdministration* zu. Verwenden Sie hierzu den Parameter **-namespace**. Wählen Sie mit dem Parameter **-class** die gewünschte WMI-Klasse aus. Dieses Skript verwendet die WMI-Klasse *VirtualDirectory*. Geben Sie die Variable *$computer* im Parameter **-computername** an und geben Sie das Ergebnis aus. Dieser Skriptabschnitt ist im Folgenden dargestellt:

```
Get-WmiObject -Namespace root\webadministration `
              -class virtualdirectory -computername $computer
```

Das vollständige Skript *ListVirtualDirectory.ps1*hat folgenden Inhalt:

ListVirtualDirectory.ps1

```
param($computer="localhost", [switch]$help)

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ListVirtualDirectory.ps1
Erstellt eine Liste mit den virtuellen Verzeichnissen auf dem lokalen Computer oder einem Remotecomputer.

PARAMETER:
-computer  Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
ListVirtualDirectory.ps1

Erstellt eine Liste mit den virtuellen Verzeichnissen auf dem lokalen Computer.

ListVirtualDirectory.ps1 -computer "webserverII"

Erstellt eine Liste mit den virtuellen Verzeichnissen auf einem Webserver namens webserverII.

ListVirtualDirectory.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

```
if($help) { "Ausgeben der Hilfeinformationen ..." ; funHelp }

Get-WmiObject -Namespace root\webadministration `
              -class virtualdirectory -computername $computer
```

Erstellen einer neuen Website

Für viele Unternehmen ist das Erstellen neuer Websites mühsam und schwierig. Sie können eine Website zwar unter Verwendung des Dialogfelds **Website hinzufügen** erstellen, aber zum Erstellen mehrerer Websites sollten Sie das Skript *CreateSite.ps1* verwenden.

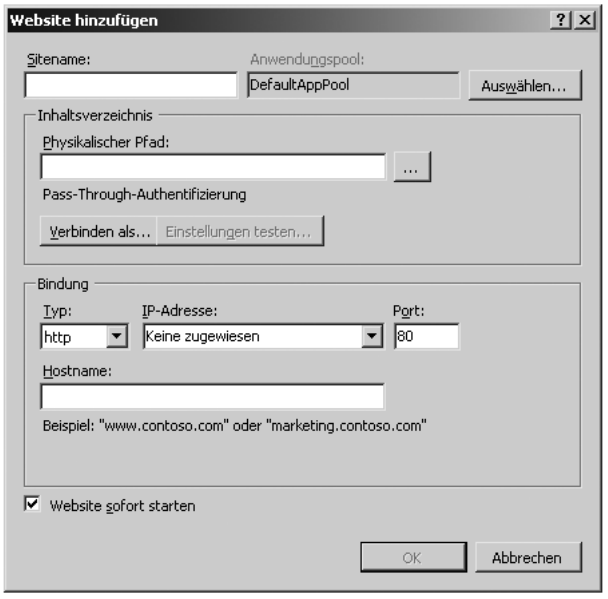


Abbildung 15.5 Das Dialogfeld *Website hinzufügen* ermöglicht das Konfigurieren einer neuen Website

Das Skript *CreateSite.ps1* verwendet zwei neue WMI-Klassen aus dem WMI-Namespace *root\WebAdministration*. Mit der Klasse *Site* wird die Website erstellt. Diese Klasse erfordert jedoch die WMI-Klasse *BindingElement*, um die Bindungsinformationen der Website festzulegen. Die *Create*-Methode aus der WMI-Klasse *Site* muss als Array angegeben werden. Verwenden Sie hierzu den *[array]*-Accelerator, um die Typenkonvertierung auszuführen.

Beginnen Sie das Skript *CreateSite.ps1* mit der Definition der Parameter. Um das Skript zu vereinfachen, legen Sie mehrere Standardwerte für die Parameter fest, auch wenn Sie nur den Namen der Website angeben müssen. Legen Sie für den Parameter **-computer** den Standardwert *localhost* fest. Das bedeutet, dass eine neue Website standardmäßig auf dem lokalen Computer erstellt wird. Der Standardpfad zeigt auf das Verzeichnis *drive*\inetpub\wwwroot. Dieses Vezeichnis eignet sich aufgrund der Standardsicherheitseinstellungen zum Erstellen von Websites. Der Standardwert für den Parameter **-port** ist 80 (der Standardwebport). Wählen Sie nur in besonderen Situationen einen anderen TCP-Port aus. Der Parameter **-tld** ist auf *com* festgelegt. Der Parameter **-protocol** hat den Wert *http* und der letzte Parameter ist **-help**, um die Hilfe für das Skript anzuzeigen. Die *param*-Anweisung lautet:

```
param(
      $sitename,
      $computer="localhost",
      $path="C:\Inetpub\WWWRoot",
      $port=80,
      $tld="com",
      $protocol="http",
      [switch]$help
     )
```

Die nächste Funktion ist *funhelp()*, um eine Hilfemeldung anzuzeigen, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Die Funktion *funhelp()* verwendet eine *here*-Zeichenfolge mit dem angezeigten Text. Der *here*-Zeichenfolge ist die Variable *$helptext* zugewiesen. Wenn das Skript mit dem Parameter **-help** ausgeführt wird, wird der Inhalt der Variablen *$helptext* angezeigt und das Skript anschließend beendet. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: CreateSite.ps1
Erstellt eine Website auf dem lokalen Computer oder einem Remotecomputer.

PARAMETER:
-computer  Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-sitename  Gibt den Namen der neuen Website an.
-path      Gibt den physischen Pfad zum Webverzeichnis an.
-port      Legt den eingehenden Port für die Website fest.
-tld       Gibt die Domäne oberster Ebene an: com, net, org ...
-protocol  Legt das zu verwendende Protokoll fest: http, https ...
-help      Zeigt dieses Hilfethema an.

SYNTAX:
CreateSite.ps1 -sitename "nwtraders"

Erstellt eine Website namens nwtrades auf dem lokalen Computer. Als Pfad
zu den Dateien der Website wird C:\Inetpub\wwwroot verwendet. Die Verbindungen
zur Site werden über Port 80 und www.nwtraders.com aufgebaut.
Die neue Website verwendet das HTTP-Protokoll.

CreateSite.ps1 -sitename "nwtraders" -computer "webserverII"

Erstellt eine Website namens nwtrades auf dem  Webserver namens webserverII. Der Name
der neuen Website lautet nwtraders. Als Pfad zu den Dateien der Website
wird C:\Inetpub\wwwroot verwendet. Die Verbindungen zur Site werden
über Port 80 und www.nwtraders.com aufgebaut. Die neue Website verwendet das HTTP-Protokoll.

CreateSite.ps1 -sitename "nwtraders" -computer "webserverII" -port 8080

Erstellt eine Website namens nwtrades auf dem  Webserver namens webserverII. Der Name
der neuen Website lautet nwtraders. Als Pfad zu den Dateien der Website
wird C:\Inetpub\wwwroot verwendet. Die Verbindungen zur Site werden
über Port 8080 und www.nwtraders.com aufgebaut. Die neue Website verwendet das HTTP-Protokoll.

CreateSite.ps1 -sitename "nwtraders" -path "D:\MeinWebVerzeichnis"
```

```
Erstellt eine Website namens nwtrades auf dem lokalen Computer. Als Pfad
zu den Dateien der Website wird D:\MeinWebVerzeichnis verwendet. Die Verbindungen
zur Site werden über Port 80 und www.nwtraders.com aufgebaut.
Die neue Website verwendet das HTTP-Protokoll.

CreateSite.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Überprüfen Sie, ob bestimmte Parameter vorhanden sind. Suchen Sie als Erstes nach dem Parameter **-help**. Wenn der Parameter gefunden wird, geben Sie eine Statusmeldung aus und rufen Sie die Funktion *funhelp()* auf. Überprüfen Sie als Nächstes, ob der erforderliche Parameter **-sitename** angegeben wurde. Dieser Parameter ist erforderlich, da Sie eine Website nicht ohne Namen erstellen können. Geben Sie den *not*-Operator (!) vor der Variablen an, die dem Parameter **-sitename** zugeordnet ist. Wenn die Variable *$sitename* nicht existiert, geben Sie eine Statusmeldung aus und rufen die Funktion *funhelp()* auf. Die beiden Codezeilen lauten:

```
if($help)      {      "Ausgeben der Hilfeinformationen ..."      ;      funHelp      }
if(!$sitename) { "Der Sitename wurde nicht angegeben ..." ; funHelp}
```

Der nächste Vorgang betrifft die Websitebindung. Das Format für die Bindungszeichenfolge ist im Windows SDK beschrieben. Dieses Format umfasst einen Platzhalter, eine Portnummer, einen Hostnamen (z.B. www), den Websitenamen und einen Domänennamen. Das Erstellen der Bindungszeichenfolge wird durch eine Variable vereinfacht. Die Bindungszeichenfolge und die Variablenzuweisung lauten:

```
$siteBinding = "*:$($port):www.$($sitename).$($tld)"
```

Erstellen Sie im nächsten Abschnitt des Skripts eine Instanz der WMI-Klasse *Site* unter Verwendung der .NET Framework-Klasse *System.Management.ManagementObject ManagementClass*. Der Accellerator für diese .NET Framework-Klasse ist *[wmiclass]*. Diese Klasse ermöglicht das Instanziieren von WMI-Klassen. Rufen Sie anschließend die Methode *Create()* auf.

Tipp Wenn Sie das Cmdlet **Get-WmiObject** verwenden, müssen Sie nicht auf die Methode *Create()* zugreifen. Die Verwendung von *[wmiclass]* entspricht im Wesentlichen der *Get()*-Methode aus dem COM-Objekt *sWbemServices*. Beachten Sie dies beim Umwandeln von alten VBScript-Skripts.

Möglicherweise müssen Sie beim Erstellen des Pfads zum WMI-Namespace und der Klasse *Site* eine Verbindung zu einem anderen Computer herstellen. Geben Sie hierzu die Variable *$computer* in der ersten Position des Pfads an. Der Pfad besteht aus dem Computernamen, dem Namespace und der Klasse. Da Sie die Verbindung mit einem Remotecomputer herstellen, müssen Sie in der ersten Position einen Wert angeben. Da Sie mit einem WMI-Namespace arbeiten, der nicht dem Standard entspricht, müssen Sie die Namespace-Informationen an der zweiten Position eingeben. Wenn Sie mit dem lokalen Computer und dem WMI-Standardnamespace *root\cimv2* arbeiten würden, lautete die Verbindungszeichenfolge *[wmiclass]"win32_service"*.

Die Verbindungszeichenfolge für die WMI-Klasse *Site* lautet:

```
$site = [wmiclass]\\$computer\root\WebAdministration:site
```

Sie müssen außerdem eine neue Instanz der WMI-Klasse *BindingElement* erstellen. Diese WMI-Klasse gibt die Parameter für die Methode *Create()* in der Klasse *Site* an. Verwenden Sie die Klasse *[wmiclass] Management*, um das Erstellen einer neuen Instanz der WMI-Klasse *BindingElement* zu ermöglichen. Rufen Sie anschließend die Methode *CreateInstance()* auf, um eine neue Instanz der Klasse *BindingElement* anzulegen. Codezeile:

Warnung In folgender Codezeile zum Erstellen einer neuen Instanz der WMI-Klasse *BindingElement* ist das Graviszeichen angegeben. Die Codezeile wurde aus Gründen der Lesbarkeit in zwei Zeilen angegeben. Beachten Sie jedoch, dass Sie die Codezeile an dieser Stelle allerdings nicht mit einem Graviszeichen unterbrechen können. Der Code könnte nicht verarbeitet werden. Das Skript funktioniert, wenn der Code in einer Zeile angegeben wird.

```
$binding = ([wmiclass]\\$computer\root\WebAdministration: `
bindingElement).createinstance()
```

Geben Sie nun die Parameter für die Bindungsinformationen an. Das erste Element ist die Bindungszeichenfolge, die Sie erstellt und der Variablen *$sitebinding* zugewiesen haben. Geben Sie als Nächstes das zu verwendende Protokoll an. Wandeln Sie die Elemente des *$binding*-Objekts mit der *[array]*-Typeneinschränkung in ein Array um. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
$binding.bindinginformation = $siteBinding
$binding.protocol = $protocol
$bindingArray = [array]$binding
```

Rufen Sie nun die Methode *Create()* auf. Die Methode *Create()* akzeptiert drei Parameter: Den Namen der Website, die in einer neuen Instanz der WMI-Klasse gespeicherten *BindingElement*-Bindungsinformationen und den Pfad zum Websiteverzeichnis. Codezeile:

```
$site.create($sitename, $bindingArray, $path)
```

Das vollständige Skript *CreateSite.ps1* hat folgenden Aufbau:

CreateSite.ps1

```
param(
      $sitename,
      $computer="localhost",
      $path="C:\Inetpub\WWWRoot",
      $port=80,
      $tld="com",
      $protocol="http",
      [switch]$help
     )

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: CreateSite.ps1
Erstellt eine Website auf dem lokalen Computer oder einem Remotecomputer.

PARAMETER:
-computer  Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-sitename  Gibt den Namen der neuen Website an.
-path      Gibt den physischen Pfad zum Webverzeichnis an.
-port      Legt den eingehenden Port für die Website fest.
-tld       Gibt die Domäne oberster Ebene an: com, net, org ...
```

```
-protocol  Legt das zu verwendende Protokoll fest: http, https ...
-help      Zeigt dieses Hilfethema an.

SYNTAX:
CreateSite.ps1 -sitename "nwtraders"

Erstellt eine Website namens nwtrades auf dem lokalen Computer. Als Pfad
zu den Dateien der Website wird C:\Inetpub\wwwroot verwendet. Die Verbindungen
zur Site werden über Port 80 und www.nwtraders.com aufgebaut.
Die neue Website verwendet das HTTP-Protokoll.

CreateSite.ps1 -sitename "nwtraders" -computer "webserverII"

Erstellt eine Website namens nwtrades auf dem  Webserver namens webserverII. Der Name
der neuen Website lautet nwtraders. Als Pfad zu den Dateien der Website
wird C:\Inetpub\wwwroot verwendet. Die Verbindungen zur Site werden
über Port 80 und www.nwtraders.com aufgebaut. Die neue Website verwendet das HTTP-Protokoll.

CreateSite.ps1 -sitename "nwtraders" -computer "webserverII" -port 8080

Erstellt eine Website namens nwtrades auf dem  Webserver namens webserverII. Der Name
der neuen Website lautet nwtraders. Als Pfad zu den Dateien der Website
wird C:\Inetpub\wwwroot verwendet. Die Verbindungen zur Site werden
über Port 8080 und www.nwtraders.com aufgebaut. Die neue Website verwendet das HTTP-Protokoll.

CreateSite.ps1 -sitename "nwtraders" -path "D:\MeinWebVerzeichnis"

Erstellt eine Website namens nwtrades auf dem lokalen Computer. Als Pfad
zu den Dateien der Website wird D:\MeinWebVerzeichnis verwendet. Die Verbindungen
zur Site werden über Port 80 und www.nwtraders.com aufgebaut.
Die neue Website verwendet das HTTP-Protokoll.

CreateSite.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help)      { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if(!$sitename) { "Der Sitename wurde nicht angegeben ..." ; funHelp}

$siteBinding = "*:$($port):www.$($sitename).$($tld)"

$site = [wmiclass]"\\$computer\root\WebAdministration:site"
$binding = ([wmiclass]"\\$computer\root\WebAdministration: `
bindingElement").createinstance()
$binding.bindinginformation = $siteBinding
$binding.protocol = $protocol
$bindingArray = [array]$binding
$site.create($sitename, $bindingArray, $path)
```

Erstellen eines neuen Anwendungspools

Zum Erstellen mehrerer Anwendungspools auf einem Webserver können Sie auch ein Skript verwenden. Das Erstellen der Pools ist mit einem Skript einfacher als mit dem Dialogfeld **Anwendungspool hinzufügen** (siehe Abbildung 15.6). Folgen Sie den Anweisungen, um das Skript *CreateApplicationPool.ps1* zu erstellen, und Sie können Anwendungspools mit Skripts anlegen.

Abbildung 15.6 Das Erstellen neuer Anwendungspools ist einfach

Beginnen Sie das Skript *CreateApplicationPool.ps1* mit einer *param*-Anweisung. Für dieses Skript definieren Sie vier Parameter. Der erste Parameter lautet **-appname**. Dieser Parameter ist erforderlich, da Sie einen Anwendungspool nicht ohne Namen erstellen können. Die nächsten beiden Parameter, **-autostart** und **-computer**, haben Standardwerte und können ausgelassen werden, wenn die Standardwerte zutreffend sind. Der letzte Parameter ist **-help**, für den kein Wert angegeben wird. Die *param*-Anweisung lautet:

```
param(
      $appName,
      $autoStart = $true,
      $computer="localhost",
      [switch]$help
     )
```

Die nächste Funktion, *funhelp()*, zeigt die Hilfeinformationen an, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Erstellen Sie eine *here*-Zeichenfolge und weisen Sie diese der Variablen *$helptext* zu. Zeigen Sie den Inhalt der Variablen an und beenden Sie die Ausführung des Skripts mit einer *exit*-Anweisung, wenn der Parameter **-help** angegeben wurde. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: CreateApplicationPool.ps1
Erstellt einen Anwendungspool auf dem lokalen Computer oder einem Remotecomputer.

PARAMETER:
-appname   Der Name des Anwendungspools.
-autostart Gibt an, ob der Anwendungspool automatisch gestartet werden soll.
-computer  Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-help      Zeigt dieses Hilfethema an.
```

```
SYNTAX:
CreateApplicationPool.ps1 -appname MyNewAppPool

Erstellt einen neuen Anwendungspool namens MyNewAppPool auf dem lokalen Computer.
Der Anwendungspool wird automatisch gestartet.

CreateApplicationPool.ps1 -computer "webserverII" -appname MyApp `
                          -autostart 0

Erstellt einen neuen Anwendungspool namens MyApp auf einem Webserver namens
webserverII. Der Anwendungspool wird  nicht automatisch gestartet.

CreateApplicationPool.ps1 -help

Zeigt dieses Hilfethema an.

"@
$helpText
exit
}
```

Sie müssen mittels der Parameter überprüfen, ob die Variable *$help* angegeben wurde, da Sie in diesem Fall das Skript beenden müssen, nachdem die Hilfeinformationen ausgegeben wurden. Es ist effizienter, das Skript vorzeitig zu beenden. Überprüfen Sie nach der Variablen *$help*, ob der erforderliche Parameter **-appname** angegeben wurde, da das Skript ohne die Variable *$appname* nicht fortgesetzt werden kann. Rufen Sie die Funktion *funhelp()* auf, wenn der Parameter nicht existiert. Die beiden Codezeilen lauten:

```
if($help) { "Ausgeben der Hilfeinformationen..." ; funHelp }
if(!$appname) { "Für -appname wurde kein Wert angegeben." ; funHelp }
```

Nachdem Sie sichergestellt haben, dass die Parameter vorhanden sind, greifen Sie auf den WMI-Dienst zu. Die Methode *Create()* ist nur verfügbar, wenn Sie die Verbindung mit dem *[wmiclass]*-Accelerator herstellen. Sie können mit dem Cmdlet **Get-WmiObject** nicht auf die Methode *Create()* zugreifen. Der *[wmiclass]*-Accelerator akzeptiert einen WMI-Pfad als Argument. Der WMI-Pfad umfasst den Namen des Computers, den Namespace und die gewünschte WMI-Klasse. Das zurückgegebene Objekt ist eine Instanz der WMI-Klasse *ApplicationPool*. Verwenden Sie die Methode *Create()* und geben Sie den Namen des Anwendungspools in der Variablen *$appname* sowie den Wert für die Eigenschaft *AutoStart* an. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
$AppPool = [wmiclass]"\\$computer\root\WebAdministration:applicationpool"
$appPool.Create($appName,$autostart)
```

Das vollständige Skript *CreateApplicationPool.ps1* hat folgenden Inhalt:

CreateApplicationPool.ps1

```
param(
      $appName,
      $autoStart = $true,
      $computer="localhost",
      [switch]$help
     )

function funHelp()
{
$helpText=@"
```

```
BESCHREIBUNG:
NAME: CreateApplicationPool.ps1
Erstellt einen Anwendungspool auf dem lokalen Computer oder einem Remotecomputer.

PARAMETER:
-appname   Der Name des Anwendungspools.
-autostart Gibt an, ob der Anwendungspool automatisch gestartet werden soll.
-computer  Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
CreateApplicationPool.ps1 -appname MyNewAppPool

Erstellt einen neuen Anwendungspool namens MyNewAppPool auf dem lokalen Computer.
Der Anwendungspool wird automatisch gestartet.

CreateApplicationPool.ps1 -computer "webserverII" -appname MyApp `
                          -autostart 0

Erstellt einen neuen Anwendungspool namens MyApp auf einem Webserver namens
webserverII. Der Anwendungspool wird  nicht automatisch gestartet.

CreateApplicationPool.ps1 -help

Zeigt dieses Hilfethema an.

"@
$helpText
exit
}

if($help) { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if(!$appname) { "Für -appname wurde kein Wert angegeben." ; funHelp }

$AppPool = [wmiclass]"\\$computer\root\WebAdministration:applicationpool"
$appPool.Create($appName,$autostart)
```

Starten und Beenden von Websites

In bestimmten Situationen müssen Sie eine Website entweder starten oder beenden. Möglicherweise müssen Sie eine Website zu Wartungszwecken oder aus Sicherheitsgründen beenden. Die Gründe zum Starten einer Website sind offensichtlich. Um eine Website zu starten oder zu beenden, verwenden Sie die WMI-Klassen des IIS 7 WMI-Anbieters.

Das Skript *StartStopSite.ps1* startet und beendet Websites. Das Skript beginnt mit einer *param*-Anweisung und definiert mehrere Parameter. Der Parameter **-site** gibt die Website an, die gestartet oder beendet werden soll. Die anderen Parameter sind optional. Der feste Parameter **-start** zeigt an, dass die Website gestartet werden soll. Mit dem Parameter **-stop** wird die Website beendet. Diese Parameter schließen sich gegenseitig aus und können nicht in der gleichen Befehlszeile angegeben werden. Sie haben die anderen Parameter bereits zuvor verwendet. Die *param*-Anweisung:

```
param(
      $site,
      $computer="localhost",
      [switch]$start,
      [switch]$stop,
      [switch]$help
     )
```

Die Funktion *funhelp()* zeigt die Hilfe auf Anforderung des Benutzers an. Der Hilfetext besteht aus einer *here*-Zeichenfolge, die in der Variablen *$helptext* gespeichert ist. Nachdem die *here*-Zeichenfolge definiert wurde, wird der Inhalt der Variablen ausgegeben und das Skript beendet. Die Funktion *funhelp()* ist wie folgt definiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: StartStopSite.ps1
Startet oder beendet eine Website auf dem lokalen Computer oder einem Remotecomputer.

PARAMETER:
-site      Der Name der Site, die gestartet oder beendet werden soll.
-computer  Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-start     Startet die Website.
-stop      Beendet die Website
-help      Zeigt dieses Hilfethema an.

SYNTAX:
StartStopSite.ps1

Ermittelt eine Liste der Websites auf dem lokalen Computer.

StartStopSite.ps1 -computer "webserverII"

Ermittelt eine Liste der Websites auf einem Webserver namens webserverII.

StartStopSite.ps1 -site mysite -stop

Beendet eine Website namens mysite auf dem lokalen Computer.

StartStopSite.ps1 -site mysite -start -computer "webserverII"

Startet eine Website namens mysite auf einem Webserver namens webserverII.

StartStopSite.ps1 -help

Zeigt dieses Hilfethema an.

"@
$helpText
exit
}
```

Überprüfen Sie die beim Skriptaufruf angegebenen Parameter. Stellen Sie fest, ob Sie die Hilfe anzeigen müssen. Sollte die Variable *$help* nicht vorhanden sein, fahren Sie mit der nächsten Zeile fort. Wenn die Variablen *$start* und *$stop* existieren, generieren Sie eine Fehlermeldung, um den Benutzer

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

zu informieren, dass eine Website nicht gleichzeitig gestartet und beendet werden kann. Rufen Sie die Funktion *funhelp()* auf. Die Funktion *funhelp()* wird nur bei diesen beiden Bedingungen ausgeführt. Wenn weder die Variable *$start* noch die Variable *$stop* vorhanden ist, informieren Sie den Benutzer, dass die Standardaktion ausgeführt wird, und geben Sie eine Liste der Websites auf dem Server aus. Weisen Sie den Benutzer an, die Hilfe für weitere Optionen anzuzeigen. Codeabschnitt:

```
if($help) { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if($start -and $stop) {
                       "Sie können die Site $site nicht gleichzeitig starten und beenden."
                       "Für weitere Informationen lesen Sie die Onlinehilfe." ;
                       funHelp
                      }
if(!$start -or !$stop)
 {
  "Es wurde keine Aktion angegeben. Die WMI-Sites werden abgefragt."
  "Für weitere Informationen lesen Sie die Onlinehilfe."
   Get-WmiObject -Namespace root\webadministration `
                 -computername $computer -class site |
   format-table -property name
   exit
 }
```

Der nächste Skriptabschnitt führt die Methodenaufrufe aus. Wenn die Variable *$start* vorhanden ist, greifen Sie mit dem Cmdlet **Get-WmiObject** auf WMI zu. Stellen Sie eine Verbindung zum WMI-Namespace *root\webadministration* her und fragen Sie die WMI-Klasse *Site* ab. Übergeben Sie das resultierende Objekt an das Cmdlet **Where-Object** und suchen Sie nach dem Namen, der mit dem Namen im Parameter **-site** übereinstimmt. Rufen Sie anschließend die Methode *Start()* auf. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
if($start)
 {
  $objSite = Get-WmiObject -Namespace root\webadministration -class site `
                           -computername $computer|
             Where-object { $_.name -eq $site }
  $objSite.Start()
  exit
 }
```

Für den Fall, dass der Benutzer die Website beenden möchte, greifen Sie wieder mit dem Cmdlet **Get-WmiObject** auf den WMI-Namespace *root\WebAdministration* zu und fragen Sie die WMI-Klasse *Site* ab. Fügen Sie das resultierende Objekt in das Cmdlet **Where-Object** ein und filtern Sie nach dem Namen der Website. Speichern Sie das Ergebnis in der Variablen *$objsite*, rufen Sie die Methode *Stop()* auf und beenden Sie das Skript. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
if($stop)
 {
  $objSite = Get-WmiObject -Namespace root\webadministration -class site `
                           -computername $computer|
             Where-object { $_.name -eq $site }
  $objSite.Stop()
  exit
 }
```

Das vollständige Skript *StartStopSite.ps1* hat folgenden Inhalt:

StartStopSite.ps1

```
param(
      $site,
      $computer="localhost",
      [switch]$start,
      [switch]$stop,
      [switch]$help
     )

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: StartStopSite.ps1
Startet oder beendet eine Website auf dem lokalen Computer oder einem Remotecomputer.

PARAMETER:
-site      Der Name der Site, die gestartet oder beendet werden soll.
-computer  Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-start     Startet die Website.
-stop      Beendet die Website
-help      Zeigt dieses Hilfethema an.

SYNTAX:
StartStopSite.ps1

Ermittelt eine Liste der Websites auf dem lokalen Computer.

StartStopSite.ps1 -computer "webserverII"

Ermittelt eine Liste der Websites auf einem Webserver namens webserverII.

StartStopSite.ps1 -site mysite -stop

Beendet eine Website namens mysite auf dem lokalen Computer.

StartStopSite.ps1 -site mysite -start -computer "webserverII"

Startet eine Website namens mysite auf einem Webserver namens webserverII.

StartStopSite.ps1 -help

Zeigt dieses Hilfethema an.

"@
$helpText
exit
}

if($help) { "Ausgeben der Hilfeinformationen ..." ; funHelp }
```

```
if($start -and $stop) {
                      "Sie können die Site $site nicht gleichzeitig starten und beenden."
                      "Für weitere Informationen lesen Sie die Onlinehilfe." ;
                      funHelp
                     }

if($start)
 {
  $objSite = Get-WmiObject -Namespace root\webadministration -class site `
                           -computername $computer |
             Where-object { $_.name -eq $site }
  $objSite.Start()
  exit
 }
if($stop)
 {
  $objSite = Get-WmiObject -Namespace root\webadministration -class site `
                           -computername $computer |
             Where-object { $_.name -eq $site }
  $objSite.Stop()
  exit
 }
if(!$start -or !$stop)
 {
  "Es wurde keine Aktion angegeben. Die WMI-Sites werden abgefragt."
  "Für weitere Informationen lesen Sie die Onlinehilfe."
   Get-WmiObject -Namespace root\webadministration `
                 -computername $computer -class site |
   format-table -property name
   exit
 }
```

Zusammenfassung

In diesem Kapitel wurden Skripts für die Arbeit mit einem IIS-Server beschrieben. Diese Aktivitäten umfassen das Dokumentieren der Serverkonfiguration, das Abrufen der Anwendungspooleinstellungen und das Überprüfen der Standardwerte für die Anwendungspools sowie der Websiteeinschränkungen. Es wurden darüber hinaus die Methoden zum Überprüfen der virtuellen Verzeichnisse erklärt. Außerdem wurde das Verwalten eines Webservers, das Erstellen von Websites und das Erstellen von Anwendungspools beschrieben. Das Kapitel wurde mit dem Starten und Beenden von Websites abgeschlossen.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

KAPITEL 16

Arbeiten mit dem Zertifikatspeicher

Nach Abschluss dieses Kapitels können Sie:

- Bestimmte Zertifikate im Zertifikatspeicher auffinden
- Zertifikatspeicher auflisten
- Zertfikate auflisten
- Abgelaufene Zertifikate auffinden
- Zertfikate importieren
- Zertfikate löschen

Auf der Begleit-CD Alle Skripts in diesem Kapitel befinden sich auf der Begleit-CD im Ordner *\Scripts\Chapter16*.

Auffinden bestimmter Zertifikate im Zertifikatspeicher

Auf jedem Windows Vista- oder Windows Server 2008-Computer sind mehrere Zertifikate gespeichert. Da Zertifikate immer wichtiger werden, nimmt auch die Verwaltung dieser Zertifikate an Wichtigkeit zu. Eine Herausforderung im Zusammenhang mit Zertifikaten ist allerdings, dass Zertifikate nicht intuitiv angezeigt werden. Beispielsweise zeigt das Zertifikate **Snap-In** zahlreiche Ordner mit bedeutungslosen Namen und ohne ausreichende Erklärung an (siehe Abbildung 16.1).

Wenn Sie jedoch den Zertifikatanbieter in Windows PowerShell verwenden, können Sie einen einfachen Befehl ausführen, ohne das Dialogfeld **Benutzerkontensteuerung** bestätigen zu müssen. Mit dem Cmdlet **Get-ChildItem** können Sie Informationen über die Zertifikatspeicherpfade abrufen:

```
Get-ChildItem cert:\
```

Der Befehl gibt folgende Informationen zu den Pfaden der Zertifikatspeicher *CurrentUser* und *LocalMachine* zurück:

```
Location   : CurrentUser
StoreNames : {UserDS, AuthRoot, CA, Trust...}

Location   : LocalMachine
StoreNames : {AuthRoot, CA, Trust, Disallowed...}
```

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

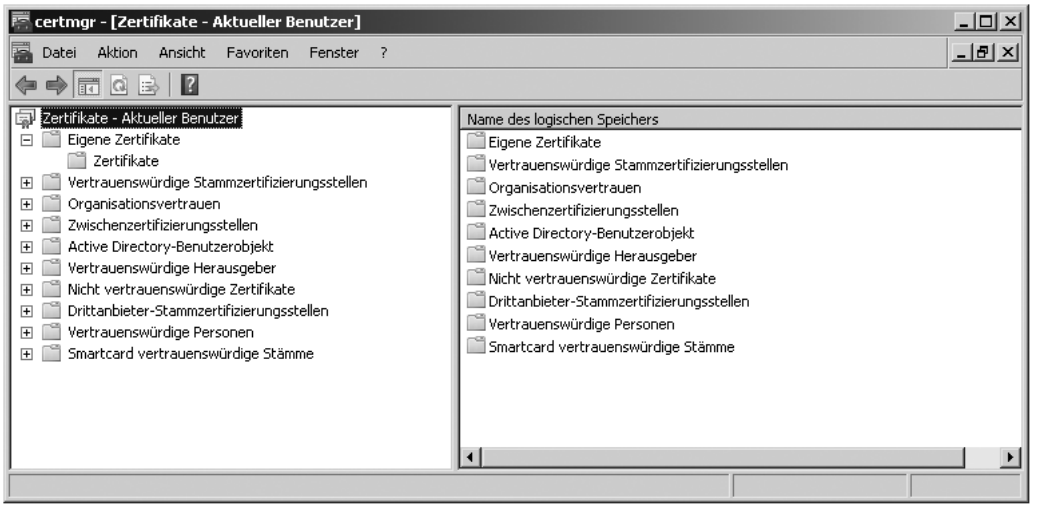


Abbildung 16.1 Das *Zertifikate* Snap-In ist aufgrund der vielen angezeigten Ordner verwirrend

Es ist zwar interessant, den Zertifikatspeicher-Namespaces *CurrentUser* ermitteln zu können, es ist aber viel wichtiger, auf die einzelnen Zertifikatspeicher unter *CurrentUser* oder *LocalMachine* zuzugreifen. Um beispielsweise die Zertifikatspeicher des aktuellen Benutzerkontos unter *CurrentUser* zu ermitteln, führen Sie folgenden Befehl aus:

```
Get-ChildItem cert:\CurrentUser
```

Die Zertifikatspeicher, die dem Namespace *CurrentUser* untergeordnet sind, werden mit diesem Befehl aufgelistet. Die angezeigten Zertifikatspeicher hängen jedoch von den installierten Anwendungen und den konfigurierten Zertifikatspeichern ab:

```
Name : UserDS
Name : AuthRoot
Name : CA
Name : Trust
Name : Disallowed
Name : My
Name : Root
Name : TrustedPeople
Name : ACRS
Name : TrustedPublisher
Name : REQUEST
```

Um die für einen Benutzer ausgestellten Zertifikate zu überprüfen, verwenden Sie den Zertifikatspeicher *My*. Dieser Zertifikatspeicher entspricht dem im **Zertifikate** Snap-In angezeigten Zertifikatspeicher **Eigene Zertifikate**. Der Zertifikatspeicher **Eigene Zertifikate** ist in Abbildung 16.2 dargestellt.

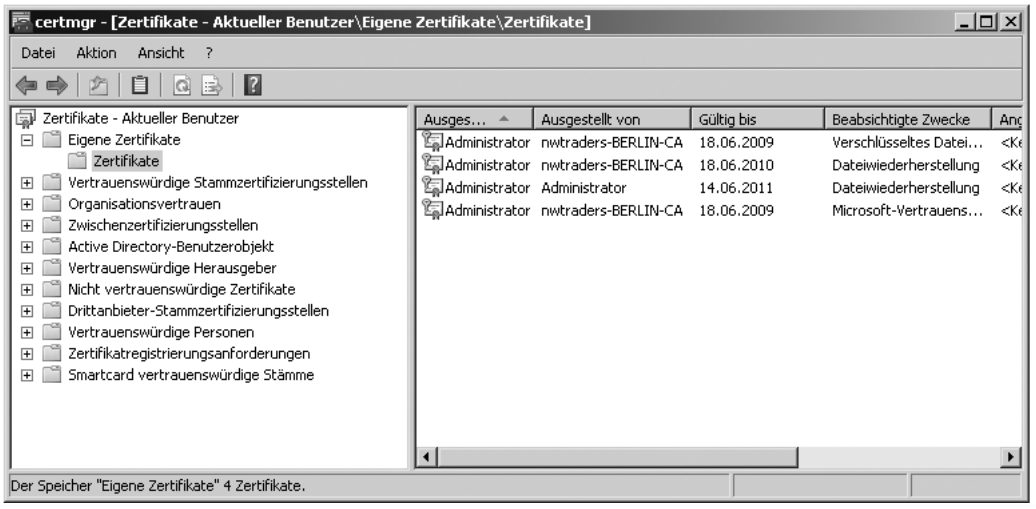


Abbildung 16.2 Persönliche Zertifikate werden im Zertifikatspeicher *Eigene Zertifikate* gespeichert

Um die für den aktuellen Benutzer ausgestellten persönlichen Zertifikate aufzulisten, führen Sie folgenden Befehl aus:

```
Get-ChildItem cert:\CurrentUser\My
```

Dieser Befehl gibt folgende Ausgabe zurück:

```
    Directory: Microsoft.PowerShell.Security\Certificate::CurrentUser\My


Thumbprint                                Subject
----------                                -------
7D7F4414CCEF168ADF6BF40753B5BECD78375931  OU=Microsoft Corporation, CN=M..
77085D8E6E5645C42DDC31771F1090D54C92FF96  CN=Administrator
```

Das Auffinden eines für einen bestimmten Verwendungszweck ausgestellten Zertifikats ist schwieriger. Ein für einen Benutzer ausgestelltes Zertifikat wird oft im Zertfikatspeicher *CurrentUser\My* gespeichert. Wenn Sie jedoch nach Zertifikaten in Windows PowerShell suchen, werden nur die Felder *Thumbprint* und *Subject* angezeigt. Die Ansicht des Zertifikatspeichers **Eigene Zertifikate**, der mit *CurrentUser\My* identisch ist, unterscheidet sich jedoch wesentlich von der Befehlsausgabe. Beispielsweise können Sie sofort feststellen, welches Zertifikat für die Codesignierung ausgestellt wurde. Um diesen Mangel zu beheben, können Sie das Skript *FindCertificates.ps1* verwenden. Das Skript *FindCertificates.ps1* verwendet die Eigenschaft *FriendlyName* aus *EnhancedKeyUses*. Diese Eigenschaften befinden sich in der Microsoft .NET Framework-Klasse *System.Security.Cryptography.X509Certificates.X509ExtensionCollection*.

Beginnen Sie das Skript *FindCertificates.ps1* mit einer *param*-Anweisung, die die Befehlszeilenargumente definiert, mit denen die Skriptausführung gesteuert werden kann. Es werden zwei Parameter unterstützt. Der erste Parameter lautet **-use**, um das Zertifikat anhand des Verwendungszwecks zu ermitteln. Sie können einen beliebigen Wert angeben, der sich auf das Zertifikat bezieht, beispielsweise *Codesignatur*, *Smardcard-Anmeldung* oder *Dokumentsignatur*. Da Sie im Code einen regulären Ausdruck verwenden, müssen Sie den Namen nicht vollständig eingeben.

Der Parameter **-help** hat einen festen Wert und muss nicht angegeben werden. Wenn dieser Parameter beim Ausführen des Skripts angegeben wird, wird die Hilfe angezeigt und das Skript beendet. Die *param*-Anweisung lautet:

```
param($use, [switch]$help)
```

Erstellen Sie als Nächstes die Funktion *funhelp()*, um eine Hilfemeldung anzuzeigen, wenn das Skript mit dem Parameter *-help* ausgeführt wird. Erstellen Sie im Codeblock der Funktion die Variable *$helptext* und weisen Sie dieser Variablen eine *here*-Zeichenfolge zu. Die *here*-Zeichenfolge beginnt mit @" und endet mit "@. Für die Eingabe zwischen diesen beiden Tags können Sie die Windows PowerShell-Regeln für Anführungszeichen ignorieren. Dies vereinfacht die Eingabe von langen Texten. Der Hilfetext ist in drei Abschnitte aufgeteilt: Beschreibung, Parameter und Syntax. Nachdem Sie die *here*-Zeichenfolge für die Variable *$helptext* definiert haben, zeigen Sie den Inhalt der Variablen an und beenden Sie das Skript. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: FindCertificates.ps1
Ermittelt die Zertifikate anhand des angegebenen Verwendungszwecks auf dem lokalen Computer.

PARAMETER:
-use       Der Verwendungszweck des Zertifikats, z.B.: Codesignierung.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
FindCertificates.ps1
Zeigt eine Liste aller Zertifikate aus dem Zertifikatspeicher My an.

FindCertificates.ps1 -use "Dokumentsignatur"

Zeigt eine Liste der Zertifikate aus dem Zertifikatspeicher My an,
die für digitale Unterschriften auf dem lokalen Computer zur Verfügung stehen.

FindCertificates.ps1 -use "Codesignatur"

Zeigt eine Liste der Zertifikate aus dem Zertifikatspeicher My an,
die für die Codesignierung auf dem lokalen Computer zur Verfügung stehen.

FindCertificates.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Überprüfen Sie, ob die Variable $help vorhanden ist. Ist dies der Fall, wurde das Skript mit dem Parameter **-help** ausgeführt. Geben Sie entsprechend eine Statusmeldung aus und rufen Sie die Funktion *funhelp()* auf. Codezeile:

```
if($help) { "Ausgeben der Hilfeinformationen ..." ; funHelp }
```

Überprüfen Sie nun, ob die Variable *$use* vorhanden ist. Wenn die Variable nicht vorhanden ist, wurde diese nicht in der Befehlszeile angegeben. Da Sie keinen Standardwert für *$use* angegeben haben, ist keine Standardaktion für das Skript verfügbar. Entsprechend müssen Sie deshalb eine Statusmeldung ausgeben und die Funktion *funhelp()* aufrufen. Codezeile:

```
if(!$use) { "Der Parameter -use ist erforderlich..." ; funHelp }
```

Ermitteln Sie danach sämtliche Zertifikatobjekte aus dem Zertifikatspeicher *My* und speichern Sie diese in der Variablen *$mycert*. Verwenden Sie hierzu das Cmdlet **Get-ChildItem**, verweisen Sie auf das *PSDrive*-Objekt *cert:*\ und durchsuchen Sie den Zertifikatspeicher *CurrentUser\My*. Codezeile:

```
$myCert = Get-ChildItem cert:\CurrentUser\My
```

Nachdem Sie die Zertifikatobjekte in der Variablen *$mycert* gespeichert haben, müssen die Objekte durchlaufen. Verwenden Sie hierzu eine *foreach*-Anweisung mit der Variablen *$cert* als Enumerator. Rufen Sie für jedes Zertifikatobjekt die Methode *get_extensions()* auf. Die von der Methode zurückgegebenen Objekte werden der Variablen *$certext* zugewiesen. Durchlaufen Sie die Erweiterungsobjekte mit der Variablen *$ext* als Enumerator. Jedes Erweiterungsobjekt besteht aus zwei Eigenschaften. Sie sind jedoch nur an der Eigenschaft *FriendlyName* interessiert. Durchlaufen Sie die Objekte erneut mit der Variablen *$name*. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
ForEach( $cert in $myCert)
 {
   $certExt = $cert.get_extensions()
    Foreach( $ext in $certExt )
      {
        foreach( $name in $ext.enhancedKeyUsages )
```

Verwenden Sie in der letzten *foreach*-Schleife eine *if*-Anweisung, um die Eigenschaft *FriendlyName* auszuwerten. Wenn für den regulären Ausdruck in der Variablen *$use* eine Übereinstimmung gefunden wird, geben Sie eine entsprechende Meldung mit einer Kopfzeile aus. Verwenden Sie einen Unterausdruck, um den Wert *FriendlyName* aus *$name.friendlyname* zu erweitern. Der Unterausdruck beginnt mit einem $, schließt $name.friendlyname ein und endet mit einer runden Klammer. Geben Sie ein Graviszeichen ein, um die Lesbarkeit zu verbessern, und den Befehl in der nächsten Zeile fortzusetzen. Geben Sie das Sonderzeichen **`n** ein, um eine neue Zeile anzuzeigen. Verwenden Sie einen weiteren Unterausdruck und geben Sie die Werte für *Thumbprint* und *Subject* des in der Variablen *$cert* gespeicherten Zertifikatsobjekts aus. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
        {
         if($name.friendlyname -match $use)
            {
             "Die folgenden Zertifkate entsprechen: $use."
             "Zertifkat $($name.friendlyname): `
             `n$($cert.thumbprint) `n$($cert.subject)`n"
            }
        }
    }
 }
```

Das vollständige Skript *FindCertificates.ps1* hat folgenden Inhalt:

FindCertificates.ps1

```
param($use, [switch]$help)

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: FindCertificates.ps1
Ermittelt die Zertifikate anhand des angegebenen Verwendungszwecks auf dem lokalen Computer.

PARAMETER:
-use       Der Verwendungszweck des Zertifikats, z.B.: Codesignierung.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
FindCertificates.ps1
Zeigt eine Liste aller Zertifikate aus dem Zertifikatspeicher My an.

FindCertificates.ps1 -use "Dokumentsignatur"

Zeigt eine Liste der Zertifikate aus dem Zertifikatspeicher My an,
die für digitale Unterschriften auf dem lokalen Computer zur Verfügung stehen.

FindCertificates.ps1 -use "Codesignatur"

Zeigt eine Liste der Zertifikate aus dem Zertifikatspeicher My an,
die für die Codesignierung auf dem lokalen Computer zur Verfügung stehen.

FindCertificates.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}

if($help) { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if(!$use) { "Der Parameter -use ist erforderlich ..." ; funHelp }

$myCert = Get-ChildItem cert:\CurrentUser\My
ForEach( $cert in $myCert)
 {
   $certExt = $cert.get_extensions()
    Foreach( $ext in $certExt )
      {
        foreach( $name in $ext.enhancedKeyUsages )
          {
           if($name.friendlyname -match $use)
              {
               "Die folgenden Zertifkate entsprechen: $use"
               "Zertifkat $($name.friendlyname): `
               `n$($cert.thumbprint) `n$($cert.subject)`n"
              }
          }
      }
  }
```

Auflisten von Zertfikaten

Sie können alle Zertifikate auflisten, die in einem bestimmten Zertifikatspeicher gespeichert sind. Sie können zu diesem Zweck das *PSDrive*-Objekt *cert:* verwenden, das jedoch nicht viele Optionen zum Steuern der zurückgegebenen Informationen bietet. Deshalb basiert das Skript *ListCertificates.ps1* auf der .NET Framework-Klasse *X509Store* aus dem Namespace *System.Security.Cryptography.X509Certificates*. Erstellen Sie mit dem Cmdlet **New-Object** eine Instanz dieser Klasse. Das Skript *ListCertificates.ps1* veranschaulicht diesen Prozess.

Das Skript *ListCertificates.ps1* beginnt mit einer *param*-Anweisung. Legen Sie drei Parameter in der Anweisung fest. Der erste Parameter, **-store**, gibt den Zertifikatspeicher an, der abgefragt werden soll. Dieser Parameter ist standardmäßig auf den Zertifikatspeicher *My* festgelegt. Der zweite Parameter ist **-liststores**, um eine vollständige Liste der auf dem lokalen Computer gespeicherten Zertifikate auszugeben. Der dritte Parameter lautet **-help**, um die Hilfe anzuzeigen. Die Anweisung ist im Folgenden dargestellt:

```
param($store="my", [switch]$listStores, [switch]$help)
```

Die Funktion *funhelp()* zeigt die Hilfeinformationen an. Nachdem Sie die Funktion deklariert haben, definieren Sie die Variable *$helptext* und erstellen Sie eine *here*-Zeichenfolge, in der Sie die Beschreibung, Parameter und Syntax des Skripts beschreiben. Der Vorteil einer *here*-Zeichenfolge ist, dass Sie den Text ohne besondere Regeln für Anführungszeichen angeben können. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ListCertificates.ps1
Zeigt alle Zertifikate auf dem lokalen Computer an.

PARAMETER:
-store    Der zu durchsuchende Zertifikatspeicher.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
ListCertificates.ps1
Zeigt eine Liste aller Zertifikate auf dem lokalen Computer an.

ListCertificates.ps1 -store "authroot"

Zeigt eine Liste der Zertifikate aus dem Zertifikatspeicher authroot
auf dem lokalen Computer an.

ListCertificates.ps1 -store "my"

Zeigt eine Liste der Zertifikate aus dem Zertifikatspeicher My
auf dem lokalen Computer an.

ListCertificates.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Die nächste Funktion ist *funstore()*, um eine Liste aller auf dem lokalen Computer gespeicherten Zertifikate anzuzeigen. Geben Sie als Erstes mit dem Cmdlet **Write-Host** eine Kopfzeile in Grün für den Zertifikatspeicher *CurrentUser* aus. Verwenden Sie das Cmdlet **Get-ChildItem**, verweisen Sie auf den Speicher *CurrentUser* auf dem *PSDrive*-Laufwerk *cert:* und führen Sie den gleichen Vorgang für den Zertifikatspeicher *LocalMachine* aus. Die Funktion *funstore()* ist wie folgt implementiert:

```
Function funstore()
{
 write-host -foregroundcolor green "Auflisten der Zertifikatspeicher unter CurrentUser:"
  Get-ChildItem cert:\CurrentUser
  write-host -foregroundcolor green "Auflisten der Zertifikatspeicher unter LocalMachine:`n"
  Get-ChildItem cert:\LocalMachine
  exit
}
```

Überprüfen Sie anschließend die Parameter. Um zu bestimmen, ob die Hilfe angezeigt werden muss, suchen Sie nach der Variablen *$help*. Wenn die Variable vorhanden ist, rufen Sie die Funktion *funhelp()* auf. Überprüfen Sie danach, ob die Variable *$liststore* vorhanden ist. Ist die Variable vorhanden, rufen Sie die Funktion *funstore()* auf, um die auf dem Computer verfügbaren Zertifikatspeicher anzuzeigen. Die beiden Codezeilen lauten:

```
if($help) { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if($liststore) { funstore }
```

Sie müssen mit dem Cmdlet **New-Variable** die schreibgeschützte Variable *userstore* erstellen. Legen Sie den Wert der Variablen auf *CurrentUser* fest und geben Sie mit dem Parameter **-option** an, dass die Variable schreibgeschützt ist. Erstellen Sie die Variable *$crypto* und legen Sie den Wert auf den Pfad zur .NET Framework-Klasse *x509Store* fest, um den Code übersichtlicher zu gestalten, da die Angabe des Klassennamens und Namespaces ziemlich lang ist. Die beiden Codezeilen lauten:

```
new-variable -name userStore -value "currentUser" -option readonly
$crypto = "System.Security.Cryptography.X509Certificates.X509Store"
```

Erstellen Sie eine neue Instanz der .NET Framework-Klasse *.X509Store*. Übergeben Sie dabei den Pfad zur .NET Framework-Klasse *x509Store* und zum Speicher, der in der Variablen *$store* gespeichert ist. Speichern Sie das resultierende Objekt in der Variablen *$objstore*, und öffnen Sie den Zertifikatspeicher im Schreibschutzmodus, indem Sie das Schlüsselwort *readonly* in der Methode *Open()* angeben. Um alle Zertifikate im Zertifikatspeicher aufzulisten, fragen Sie die Eigenschaft *Certificates* ab und weisen Sie die zurückgegebenen Zertifikate der Variablen *$colcerts* zu. Die drei erforderlichen Codezeilen lauten:

```
$objStore = new-object $crypto $store
$objstore.Open("Readonly")
$colcerts = $objstore.Certificates
```

Erstellen Sie mit dem Cmdlet **Write-Host** eine Kopfzeile für die Zertifikatsliste. Geben Sie die Kopfzeile mit dem Parameter *-foreground* in Blau aus. Geben Sie eine Meldung aus und ermitteln Sie mit einem Unterausdruck die Anzahl der Zertifikate im Zertifikatspeicher. Stellen Sie hierzu der Eigenschaft *$ColCerts.Count* ein weiteres Dollarzeichen voran und schließen Sie die Eigenschaft bis auf das Dollarzeichen in Klammern ein: *$($ColCerts.Count).* Diese Codezeile ruft die tatsächliche Anzahl der Zertifikate ab, anstatt den Objektnamen zu erweitern. Der Code zur Erstellung der Kopfzeile für die Ausgabe lautet:

```
Write-Host -ForegroundColor blue
 "
  Es gibt $($colcerts.count) Zertifikate im Zertifikatspeicher $store.
  Die folgenden Zertifikate wurden ermittelt:
 "
```

Da Sie eine Liste der Zertifikate abgerufen haben, müssen Sie eine *foreach*-Anweisung mit der Variablen *$cert* als Enumerator verwenden, um die Zertifikate zu durchlaufen. Verwenden Sie für alle Eigenschaften, die Sie abfragen möchten, einen Unterausdruck für die gefundenen Zertifikate. Nachdem die Eigenschaften ausgegeben wurden, schließen Sie den Zertifikatspeicher. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
foreach($cert in $colCerts)
 {
  "Anzeigename:  $($cert.FriendlyName)"
  "Seriennummer: $($cert.SerialNumber)"
  "Fingerabdruck: $($cert.thumbprint)"
  "Betreff: $($cert.subject)`n"
 }
$objstore.Close()
```

Das vollständige Skript *ListCertificates.ps1* hat folgenden Aufbau:

ListCertificates.ps1

```
param($store="my", [switch]$listStores, [switch]$help)

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ListCertificates.ps1
Zeigt alle Zertifikate auf dem lokalen Computer an.

PARAMETER:
-store    Der zu durchsuchende Zertifikatspeicher.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
ListCertificates.ps1
Zeigt eine Liste aller Zertifikate auf dem lokalen Computer an.

ListCertificates.ps1 -store "authroot"

Zeigt eine Liste der Zertifikate aus dem Zertifikatspeicher authroot
auf dem lokalen Computer an.

ListCertificates.ps1 -store "my"

Zeigt eine Liste der Zertifikate aus dem Zertifikatspeicher My
auf dem lokalen Computer an.

ListCertificates.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
```

```
 exit
}

Function funstore()
{
  write-host -foregroundcolor green "Auflisten der Zertifikatspeicher unter CurrentUser:"
  Get-ChildItem cert:\CurrentUser
  write-host -foregroundcolor green "Auflisten der Zertifikatspeicher unter LocalMachine:`n"
  Get-ChildItem cert:\LocalMachine
  exit
}

if($help) { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if($liststore) { funstore }

new-variable -name userStore -value "currentUser" -option readonly
$crypto = "System.Security.Cryptography.X509Certificates.X509Store"

$objStore = new-object $crypto $store
$objstore.Open("Readonly")
$colcerts = $objstore.Certificates
Write-Host -ForegroundColor blue
 "
  Es gibt $($colcerts.count) Zertifikate im Zertifikatspeicher $store.
  Die folgenden Zertifikate wurden ermittelt:
 "
foreach($cert in $colCerts)
 {
  "Anzeigename:  $($cert.FriendlyName)"
  "Seriennummer: $($cert.SerialNumber)"
  "Fingerabdruck: $($cert.thumbprint)"
  "Betreff: $($cert.subject)`n"
 }
$objstore.Close()
```

Ermitteln abgelaufener Zertifikate

Mit der Verbreitung von Zertifikaten nimmt auch die Anzahl der abgelaufenen Zertifikate zu. Beim Zugriff auf eine Website, beispielsweise einer Bankwebseite oder einer kommerziellen Website, kann es vorkommen, dass eine Warnung bezüglich eines abgelaufenen Zertifikats angezeigt wird. Beim Auftreten solcher Probleme müssen Sie in der Lage sein, abgelaufene Zertifikate schnell und effizient ermitteln zu können. Verwenden Sie hierzu den Zertifikatanbieter für Windows PowerShell. Ermitteln Sie im Skript *FindExpiredCertificates.ps1* das aktuelle Datum und durchsuchen Sie den Zertifikatspeicher, der vom Benutzer in der Befehlszeile angegeben wurde.

Das Skript *FindExpiredCertificates.ps1* beginnt mit einer *param*-Anweisung und umfasst vier Befehlszeilenparameter. Der Parameter **-store** gibt an, auf welchen Zertifikatspeicher das Skript zugreift. Dieser Parameter ist erforderlich, da Sie keinen Standardwert festgelegt haben. Wenn der Benutzer beim Ausführen des Skripts keinen Wert eingibt, können Sie auch den Speicher *My* als Standardwert festlegen. Geben Sie in der *param*-Anweisung keinen Wert an, da Sie den Benutzer darüber informieren möchten, dass weitere Optionen verfügbar sind, indem Sie auf den Parameter **-help** verweisen. Teilen Sie dem Benutzer außerdem mit, dass Sie im Weiteren den Standardwert *My* als Parameter verwenden. Die anderen *switch*-Anweisungen sind **-listcu**, um die Zertifikatspeicher unter

CurrentUser aufzulisten, **-listlm**, um die Zertifikatspeicher unter *LocalMachine* aufzulisten, und **-help**, um die Hilfe anzuzeigen. Die *param*-Anweisung lautet:

```
param(
      $store,
      [switch]$listcu,
      [switch]$listlm,
      [switch]$help
     )
```

Die Funktion *funhelp()* zeigt die Hilfeinformationen an. Erstellen Sie die Variable *$helptext* und weisen dieser eine *here*-Zeichenfolge mit dem Hilfetext zu. Geben Sie den Inhalt der Variablen aus und beenden Sie das Skript. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: FindExpiredCertificates.ps1
Ermittelt die abgelaufenen Zertifikate auf dem lokalen Computer.

PARAMETER:
-store     Der Zertifikatspeicher auf dem lokalen Computer.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
FindExpiredCertificates.ps1
Ermittelt eine Liste der  abgelaufenen Zertifikate
aus dem Zertifikatspeicher CurrentUser\My.

FindExpiredCertificates.ps1 -store "currentuser\my"

Ermittelt eine Liste der  abgelaufenen Zertifikate
aus dem Zertifikatspeicher CurrentUser\My.

FindExpiredCertificates.ps1 -store "currentuser\smartcardroot"

Ermittelt eine Liste der  abgelaufenen Zertifikate
aus dem Zertifikatspeicher CurrentUser\SmartcardRoot.

FindExpiredCertificates.ps1 -listcu

Ermittelt eine Liste aller Zertifikatspeicher
unter CurrentUser.

FindExpiredCertificates.ps1 -listlm

Ermittelt eine Liste aller Zertifikatspeicher
unter LocalMachine.

FindExpiredCertificates.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Überprüfen Sie in der Befehlszeile, welche Parameter angegeben wurden. Überprüfen Sie zuerst den Parameter **-help**. Ist die Variable *$help* vorhanden, wurde das Skript mit dem Parameter **-help** ausgeführt. Geben Sie in diesem Fall eine Meldung aus und rufen Sie die Funktion *funhelp()* auf. Codezeile:

```
if($help)   {   "Ausgeben der Hilfeinformationen ..."   ;   funHelp   }
```

Untersuchen Sie danach den Parameter **-listcu**. Wenn die Variable *$listcu* vorhanden ist, wurde das Skript mit dem Parameter **-listcu** ausgeführt. Geben Sie eine Statusmeldung aus und erstellen Sie mit dem Cmdlet **Get-ChildItem** eine Liste der Zertifikatspeicher unter *CurrentUser*. Beenden Sie anschließend das Skript. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
if($listcu) {
             "Zertifikatspeicher unter CurrentUser:"
             get-childitem cert:\currentuser ; exit
            }
```

Der nächste Parameter ist **-listlm**. Wenn die Variable *$listlm* vorhanden ist, wurde das Skript mit dem Parameter **-listlm** ausgeführt. Geben Sie in diesem Fall eine Statusmeldung aus und erstellen Sie mit dem Cmdlet **Get-ChildItem** eine Liste der Zertifikatspeicher unter *LocalMachine*. Beenden Sie anschließend das Skript. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
if($listlm) {
             "Zertifikatspeicher unter LocalMachine:"
             get-childitem cert:\localmachine ; exit
            }
```

Der Parameter **-store** bestimmt, welcher Zertifikatspeicher nach abgelaufenen Zertifikaten durchsucht werden soll. Wenn der Parameter **-store** nicht angegeben wird, wird standardmäßig der Speicher *CurrentUser\My* durchsucht. Geben Sie eine Meldung aus, dass die Standardwerte verwendet werden. Verwenden Sie dabei die Anweisung *$myinvocation.mycommand*, um den Namen des ausgeführten Skripts anzugeben, und weisen Sie auf den Parameter **-help** hin, mit dem weitere Beispiele angezeigt werden können. Codezeile:

```
if(!$store) {
             $store = "currentuser\my"
             "Verwenden des Standardzertifikatspeichers: $store"
             "Siehe $($myinvocation.mycommand) -help" `
             + " für weitere Beispiele."
            }
```

Das Skript *FindExpiredCertificates.ps1* gibt eine Meldung aus, dass der Standardzertifikatspeicher verwendet wird. Da die Ausgabe der Hilfeinformationen nur eine Zeile belegen soll, schließen Sie die Anführungszeichen und geben Sie in der ersten Zeile das Graviszeichen ein. Verknüpfen Sie die zweite Hilfetextzeile, indem Sie für die restliche Zeichenfolge ein Pluszeichen eingeben. Wenn Sie die Zeichenfolge ohne schließendes Anführungszeichen in der nächsten Zeile fortsetzen, würden zwei Zeilen angezeigt.

Der nächste Abschnitt des Skripts ist der Referenzabschnitt.

Weiterführende Informationen Die vier Abschnitte eines Skripts sind im Buch *Microsoft VBScript Step by Step* (Microsoft Press, 2006) ausführlich beschrieben. Obwohl dieses Buch VBScript behandelt, ist es eine hervorragende Grundlage für das Scripting mit Windows PowerShell, da die meisten Richtlinien auch auf das Erstellen von PowerShell-Skripts zutreffen.

Weisen Sie mit dem Cmdlet **Get-Date** der Variablen *$currentdate* ein *DateTime*-Objekt zu und rufen Sie anschließend mit dem Cmdlet **Get-ChildItem** die Zertifikarte aus dem Speicher ab, der in der Variablen *$store* angegeben ist. Die Variable *$colcert* ermöglicht dann den Zugriff auf die Liste der Zertifikate. Die beiden Codezeilen lauten:

```
$currentDate = Get-Date
$colcert = Get-ChildItem cert:\$store
```

Geben Sie mit dem Cmdlet **Write-Host** und dem Parameter **-foregroundcolor** eine Meldung in Blaugrün aus. Verwenden Sie eine *foreach*-Anweisung, um die Liste der Zertifikate mit der Variablen *$cert* als Enumerator zu durchlaufen. Nachdem Sie der Variablen *$cert* ein Zertifikat zugewiesen haben, überprüfen Sie, ob der Wert der Eigenschaft *NotAfter* kleiner als der in der Variablen *$currentdate* gespeicherte Wert ist. Wenn der Wert kleiner ist, geben Sie den Fingerabdruck und das Ablaufdatum des Zertifikats aus. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
Write-host -foregroundcolor cyan "Abgelaufene Zertifikate im Speicher $store:"
foreach($cert in $colcert)
{
 if($cert.notafter -lt $currentDate)
   {
    Write-host `
      "
       $($cert.thumbprint) `t $($cert.Notafter)
      "
   }
}
```

Das vollständige Skript *FindExpiredCertificates.ps1* hat folgenden Aufbau:

FindExpiredCertificates.ps1

```
param(
      $store,
      [switch]$listcu,
      [switch]$listlm,
      [switch]$help
     )
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: FindExpiredCertificates.ps1
Ermittelt die abgelaufenen Zertifikate auf dem lokalen Computer.

PARAMETER:
-store     Der Zertifikatspeicher auf dem lokalen Computer.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
FindExpiredCertificates.ps1
Ermittelt eine Liste der  abgelaufenen Zertifikate
aus dem Zertifikatspeicher CurrentUser\My.

FindExpiredCertificates.ps1 -store "currentuser\my"

Ermittelt eine Liste der  abgelaufenen Zertifikate
aus dem Zertifikatspeicher CurrentUser\My.
```

```
FindExpiredCertificates.ps1 -store "currentuser\smartcardroot"

Ermittelt eine Liste der  abgelaufenen Zertifikate
aus dem Zertifikatspeicher CurrentUser\SmartcardRoot.

FindExpiredCertificates.ps1 -listcu

Ermittelt eine Liste aller Zertifikatspeicher
unter CurrentUser.

FindExpiredCertificates.ps1 -listlm

Ermittelt eine Liste aller Zertifikatspeicher
unter LocalMachine.

FindExpiredCertificates.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}

if($help)   { "Ausgeben der Hilfeinformationen..." ; funHelp }
if($listcu) {
             "Zertifikatspeicher unter CurrentUser:"
             get-childitem cert:\currentuser ; exit
            }
if($listlm) {
             "Zertifikatspeicher unter LocalMachine:"
             get-childitem cert:\localmachine ; exit
            }
if(!$store) {
             $store = "currentuser\my"
             "Verwenden des Standardzertifikatspeichers: $store"
             "Siehe $($myinvocation.mycommand) -help" `
             + " für weitere Beispiele."
            }

$currentDate = Get-Date
$colcert = Get-ChildItem cert:\$store
Write-host -foregroundcolor cyan "Abgelaufene Zertifikate im Speicher $store:"
foreach($cert in $colcert)
{
 if($cert.notafter -lt $currentDate)
   {
    Write-host `
      "
       $($cert.thumbprint) `t $($cert.Notafter)
      "
   }
}
```

Identifizieren von Zertifikaten, die in Kürze ablaufen

Für Benutzer ausgestellte Zertifikate sind normalerweise nur für ein oder zwei Jahre gültig. Es wird also immer wieder Benutzer geben, die beispielsweise E-Mail nicht mehr signieren, eine Remoteverbindung nicht mehr herstellen oder eine Datei nicht mehr verschlüsseln können, da das jeweilige Zertifikat abgelaufen ist. Deshalb ist proaktives Scripting so wichtig. Verwenden Sie das Skript *FindCertificatesAboutToExpire.ps1*, um das Ablaufdatum von Zertifikaten zu überprüfen.

Das Skript *ListCertificates.ps1* beginnt mit einer *param*-Anweisung und umfasst fünf Parameter. Ein Parameter ist erforderlich, ein Parameter hat einen Standardwert und die anderen drei Parameter haben feste Werte. Der Parameter **-store** ist erforderlich. Wie im Skript *FindExpiredCertificates.ps1* müssen Sie einen Wert angeben, wenn die Variable *$store* nicht vorhanden ist. Der Parameter **-days** ist standardmäßig auf 30 Tage festgelegt. Der Parameter **-listcu** listet die Zertifikatspeicher unter *CurrentUser* auf. Der Parameter **-listlm** listet die Zertifikatspeicher unter *LocalMachine* auf. Der Parameter **-help** gibt die Hilfe aus. Die *param*-Anweisung lautet:

```
param(
      $store,
      $days=30,
      [switch]$listcu,
      [switch]$listlm,
      [switch]$help
     )
```

Die nächste Funktion ist *funhelp()*, um die Hilfe für das Skript anzuzeigen, einschließlich einiger Syntaxbeispiele. In der Funktion *funhelp()* wird die Variable *$helptext* erstellt, der der Hilfetext zugewiesen wird. Erstellen Sie den Hilfetext in einer *here*-Zeichenfolge, in der Sie keine Anführungszeichen angeben müssen. Erstellen Sie die Abschnitte für die Beschreibung, Parameter und Syntax, geben Sie den Inhalt der Variablen *$helptext* aus und beenden Sie das Skript. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: FindCertificatesAboutToExpire.ps1
Finds certificates about to expire with in a certain
number of days on the local machine

PARAMETER:
-store     Der Zertifikatspeicher auf dem lokalen Computer.
-days      Die Anzahl der Tage, für die zukünftig ablaufende Zertifikate ermittelt werden sollen.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
FindCertificatesAboutToExpire.ps1

Gibt eine Liste der Zertifikate aus dem Zertifikatspeicher CurrentUser\My aus,
die innerhalb der nächsten 30 Tage ablaufen.

FindCertificatesAboutToExpire.ps1 -days 45

Gibt eine Liste der Zertifikate aus dem Zertifikatspeicher CurrentUser\My aus,
die innerhalb der nächsten 45 Tage ablaufen.
```

```
FindCertificatesAboutToExpire.ps1 -store "currentuser\my" -days 60

Gibt eine Liste der Zertifikate aus dem Zertifikatspeicher CurrentUser\My aus,
die innerhalb der nächsten 60 Tage ablaufen.

FindCertificatesAboutToExpire.ps1 -store "currentuser\smartcardroot"

Gibt eine Liste der Zertifikate aus dem Zertifikatspeicher CurrentUser\SmartcardRoot aus,
die innerhalb der nächsten 30 Tage ablaufen.

FindCertificatesAboutToExpire.ps1 -listcu

Ermittelt eine Liste aller Zertifikatspeicher
unter CurrentUser.

FindCertificatesAboutToExpire.ps1 -listlm

Ermittelt eine Liste aller Zertifikatspeicher
unter LocalMachine.

FindCertificatesAboutToExpire.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Nachdem die Funktion *funhelp()* abgeschlossen ist, überprüfen Sie die Parameter. Wenn der Parameter **-help** angegeben wurde, können Sie die Funktion *funhelp()* aufrufen und das Skript ausführen. Codezeile:

```
if($help)   {   "Ausgeben der Hilfeinformationen ..."   ;   funHelp   }
```

Wenn die Variable *$listcu* vorhanden ist, geben Sie eine Statusmeldung aus, erstellen Sie mit dem Cmdlet **Get-ChildItem** eine Liste der Zertifikatspeicher unter *CurrentUser* und beenden Sie das Skript. Führen Sie die gleichen Vorgänge für den Parameter **-listlm** aus. Wenn die Variable *$listlm* existiert, geben Sie eine Statusmeldung aus, erstellen Sie mit dem Cmdlet **Get-ChildItem** eine Liste der Zertifikatspeicher unter *LocalMachine* und beenden Sie das Skript. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
if($listcu) {
             "Zertifikatspeicher unter CurrentUser:"
             get-childitem cert:\currentuser ; exit
            }
if($listlm) {
             "Zertifikatspeicher unter LocalMachine:"
             get-childitem cert:\localmachine ; exit
            }
```

Wenn der erforderliche Parameter **-store** nicht angegeben wurde, ist die Variable *$store* nicht vorhanden. Fragen Sie in diesem Fall den Zertifikatspeicher *My* ab. Informieren Sie den Benutzer, dass weitere Optionen verfügbar sind. Zeigen Sie eine Meldung an, dass ein Standardwert verwendet wird, und geben Sie mit der Anweisung *$myinvocation.mycommand* den Skriptnamen aus. Zum Abrufen des Skriptnamens müssen Sie einen Unterausdruck verwenden. Vorschlag: Verwenden Sie die Hilfe, um Beispiele anzuzeigen. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
if(!$store) {
             $store = "currentuser\my"
             "Verwenden des Standardzertifikatspeichers: $store"
             "Verwenden Sie $($myinvocation.mycommand) -help" `
             + ", um Beispiele anzuzeigen."
            }
```

Nachdem Sie die Parameter überprüft haben, erstellen Sie eine Instanz des .NET Framework-Objekts *System.DateTime* und legen Sie mit der Methode *AddDays()* ein späteres Datum fest. Speichern Sie das Datum in der Variablen *$currentdate*. Rufen Sie dann mit dem Cmdlet **Get-ChildItem** eine Liste der Zertifikate ab. Die beiden Codezeilen lauten:

```
$currentDate = (Get-Date).adddays($days)
$colcert = Get-ChildItem cert:\$store
```

Geben Sie mit dem Cmdlet **Write-Host** eine Kopfzeile aus:

```
Write-host -foregroundcolor cyan "Zertifikate im Zertifikatspeicher $store," `
                                 " die innerhalb der nächsten $days Tage ablaufen."
```

Verwenden Sie eine *foreach*-Anweisung, um die Liste der Zertifikate mit der Variablen *$cert* als Enumerator zu durchlaufen. Überprüfen Sie die Eigenschaft *NotAfter* aller Zertifikate, die das Ablaufdatum festlegt. Wenn das Ablaufdatum vor dem in der Variablen *$currentdate* gespeicherten Datum liegt, geben Sie den Fingerabdruck und das Ablaufdatum aus. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
foreach($cert in $colcert)
{
 if($cert.notafter -lt $currentDate)
   {
    Write-host `
      "
       $($cert.thumbprint) `t $($cert.Notafter)
      "
   }
}
```

Das vollständige Skript *FindCertificatesAboutToExpire.ps1* hat folgenden Inhalt:

FindCertificatesAboutToExpire.ps1

```
param(
      $store,
      $days=30,
      [switch]$listcu,
      [switch]$listlm,
      [switch]$help
     )
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: FindCertificatesAboutToExpire.ps1
Finds certificates about to expire with in a certain
number of days on the local machine

PARAMETER:
-store     Der Zertifikatspeicher auf dem lokalen Computer.
-days      Die Anzahl der Tage, für die zukünftig ablaufende Zertifikate ermittelt werden sollen.
-help      Zeigt dieses Hilfethema an.
```

```
SYNTAX:
FindCertificatesAboutToExpire.ps1

Gibt eine Liste der Zertifikate aus dem Zertifikatspeicher CurrentUser\My aus,
die innerhalb der nächsten 30 Tage ablaufen.

FindCertificatesAboutToExpire.ps1 -days 45

Gibt eine Liste der Zertifikate aus dem Zertifikatspeicher CurrentUser\My aus,
die innerhalb der nächsten 45 Tage ablaufen.

FindCertificatesAboutToExpire.ps1 -store "currentuser\my" -days 60

Gibt eine Liste der Zertifikate aus dem Zertifikatspeicher CurrentUser\My aus,
die innerhalb der nächsten 60 Tage ablaufen.

FindCertificatesAboutToExpire.ps1 -store "currentuser\smartcardroot"

Gibt eine Liste der Zertifikate aus dem Zertifikatspeicher CurrentUser\SmartcardRoot aus,
die innerhalb der nächsten 30 Tage ablaufen.

FindCertificatesAboutToExpire.ps1 -listcu

Ermittelt eine Liste aller Zertifikatspeicher
unter CurrentUser.

FindCertificatesAboutToExpire.ps1 -listlm

Ermittelt eine Liste aller Zertifikatspeicher
unter LocalMachine.

FindCertificatesAboutToExpire.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}

if($help)   { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if($listcu) {
             "Zertifikatspeicher unter CurrentUser:"
             get-childitem cert:\currentuser ; exit
            }
if($listlm) {
             "Zertifikatspeicher unter LocalMachine:"
             get-childitem cert:\localmachine ; exit
            }
if(!$store) {
             $store = "currentuser\my"
             "Verwenden des Standardzertifikatspeichers: $store"
             "Verwenden Sie $($myinvocation.mycommand) -help" `
             + ", um Beispiele anzuzeigen."
            }
```

```
$currentDate = (Get-Date).adddays($days)
$colcert = Get-ChildItem cert:\$store
Write-host -foregroundcolor cyan "Zertifikate im Zertifikatspeicher $store," `
                                 " die innerhalb der nächsten $days Tage ablaufen."
foreach($cert in $colcert)
{
 if($cert.notafter -lt $currentDate)
   {
    Write-host `
      "
       $($cert.thumbprint) `t $($cert.Notafter)
      "
   }
}
```

Verwalten von Zertifikaten

Die Verwaltung von Zertifikaten umfasst mehrere Aufgaben, einschließlich das Importieren, Überprüfen und Löschen von Zertifikaten. Diese Aufgaben werden im folgenden Abschnitt beschrieben.

Überprüfen eines Zertifikats

Bevor Sie ein Zertifikat importieren, müssen Sie dieses überprüfen, um sicherzustellen, dass es sich um das korrekte Zertifikat handelt. Diese Aufgabe kann nicht unmittelbar mit dem Snap-In **Zertifikate** ausgeführt werden. Um ein Zertifikat zu überprüfen, verwenden Sie die .NET Framework-Klasse *X509Certificate*. Die Klasse *X509Certificate* befindet sich im .NET Framework-Namespace *Security.Cryptography.X509Certificates*. Die zu überprüfenden Eigenschaften sind mit den im Snap-In **Zertifikate** angezeigten Eigenschaften identisch (siehe Abbildung 16.3).

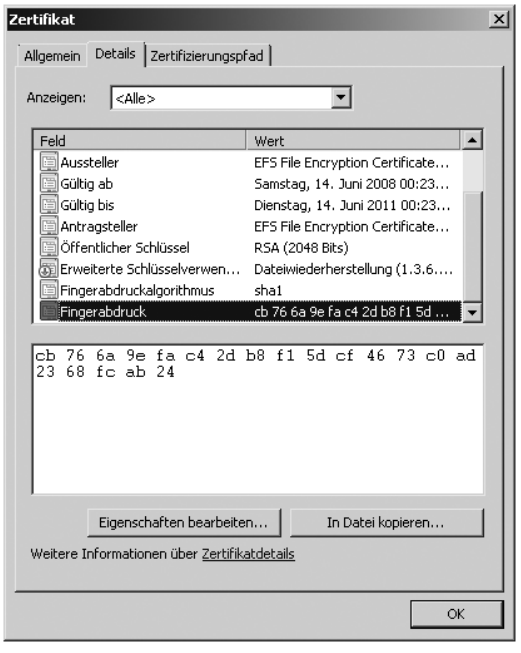


Abbildung 16.3 Zertifikateigenschaften im *Snap-In Zertifikate*

Beginnen Sie das Skript *InspectCertificate.ps1* mit einer *param*-Anweisung. Für das Skript sind zwei Parameter erforderlich. Der erste Parameter ist **-cert**, um den vollständigen Pfad und den Namen des Zertifikats anzugeben, das überprüft werden soll. Der zweite Parameter ist **-help**, um die Hilfe anzuzeigen. Die *param*-Anweisung lautet:

```
param($cert, [switch]$help)
```

Die nächste Funktion ist *funhelp()*, um die eine Hilfemeldung anzuzeigen, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Erstellen Sie mit der *function*-Anweisung die Funktion *funhelp()*. Beginnen Sie den Code für die Funktion mit geschweiften Klammern ({ }). Deklarieren Sie im Codeblock die Variable *$helptext*, und erstellen Sie eine *here*-Zeichenfolge. Die *here*-Zeichenfolge beginnt mit @" und endet mit "@. Sie müssen in der *here*-Zeichenfolge keine Anführungszeichen eingeben. Nachdem Sie die *here*-Zeichenfolge erstellt haben, zeigen Sie den Inhalt der Variablen *$helptext* an und beenden Sie das Skript mit der *exit*-Anweisung. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: InspectCertificate.ps1
Ermittelt die Zertifikate für einen bestimmten Verwendungszweck auf dem lokalen Computer.

PARAMETER:
-cert      Der vollständige Pfad zu dem Zertifikat, das untersucht werden soll.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
InspectCertificate.ps1
Generiert eine Fehlermeldung, da ein Zertifikat angegeben werden muss.

InspectCertificate.ps1 -cert "C:\FSO\Filerecovery.cer"

Inspiziert ein Zertifikat namens Filerecovery, das sich im
Verzeichnis C:\FSO befindet. Dieses Zertifikat kann in einer DER-kodierten
oder Base64-kodierten .cer-Datei gespeichert sein.

InspectCertificate.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Überprüfen Sie nun die Befehlszeilenparameter. Die erste Auswertung unter Verwendung einer *if*-Anweisung betrifft die Variable *$help*. Geben Sie im Code für die *if*-Anweisung eine Zeichenfolge aus und rufen Sie die Funktion *funhelp()* auf. Wenn Sie zwei Befehle in der gleichen Zeile eingeben, trennen Sie diese durch ein Semikolon. Überprüfen Sie mit dem *not*-Operator (!), ob die Variable *$cert* vorhanden ist. Wird die Variable *$cert* nicht gefunden, rufen Sie die Funktion *funhelp()* auf. Die beiden Codezeilen lauten:

```
if($help) { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if(!$cert) { "Es ist ein Zertifikat erforderlich ..." ; funHelp }
```

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Greifen Sie unter Verwendung der .NET Framework-Klasse *X509Certificate* aus dem .NET Framework-Namespace *Security.Cryptography.X509Certificates* auf das Zertifikat zu.

Arbeiten mit .NET Framework-Klassen

Im Skript *InspectCertificate.ps1* verwenden Sie eine Verknüpfungsmethode zum Erstellen einer Instanz der Klasse *X509Certificate*. Um beispielsweise eine Zeichenfolge zu erstellen, können Sie diese mit folgender Syntax dem Typ *System.String* zuweisen:

```
[string]"Dies ist eine Zeichenfolge."
```

Um eine Instanz des *X509Certifikate*-Objekts zu erstellen, geben Sie also Folgendes ein:

```
$objCert=[security.cryptography.x509certificates.x509certificate]"$cert"
```

Dieser Prozess ist im Skript *ThreeStrings.ps1* veranschaulicht. Speichern Sie eine Zeichenfolge in der Variablen *$a*. Stellen Sie mit der Methode *GetType()* sicher, dass es sich um ein Objekt der .NET Framework-Klasse *System.String* handelt. Verwenden Sie einen Accelerator, *[string]*, um erneut eine Zeichenfolge zu erstellen. Weisen Sie eine Zeichenfolge, die anzeigt, dass der Objekttyp *System.String* ist, der Variablen *$b* zu. Geben Sie den vollen Namen *[system.string]* in eckigen Klammern ([ ]) an und weisen Sie das Ergebnis der Variablen *$c* zu, die *System.String* als Objekttyp anzeigt. Das Skript *ThreeStrings.ps1* ist wie folgt aufgebaut:

ThreeStrings.ps1

```
$a = "`$a ist eine Zeichenfolge."
$a
"$a : Ist eine $($a.gettype())`n"

$b = [string]"`$b ist eine Zeichenfolge."
$b
"$b : Ist eine $($b.gettype())`n"

$c = [system.string]"`$c ist eine Zeichenfolge."
$c
"$c : Ist eine $($c.gettype())`n"

"Die $($c.gettype()) .NET Framework-Klasse hat die folgenden " `
 + "Methoden und Eigenschaften."
$a | get-member
```

Um auf das Zertifikat zuzugreifen, verwenden Sie die Variable *$cert*, die auf das Zertifikatobjekt verweist, und referenzieren Sie die .NET Framework-Klasse *X509Certificate*. Weisen Sie das Objekt der Variablen *$objcert* zu. Codezeile:

```
$objCert=[security.cryptography.x509certificates.x509certificate]"$cert"
```

Das restliche Skript verwendet Unterausdrücke, um die Ergebnisse mehrerer Methodenaufrufe auszugeben. Diese Anweisungen demonstrieren erstmalig die Verwendung eines Unterausdrucks, um das Ergebnis einer Methode in einer Textzeichenfolge zurückzugeben. Die letzten beiden Elemente im Ausgabeabschnitt des Skripts betreffen die Eigenschaften: *Aussteller* und *Betreff*. Der vollständige Ausgabeabschnitt des Skripts lautet:

```
"Fingerabdruck: $($objCert.GetCertHashString())"
"Gültig ab: $($objCert.GetEffectiveDateString())"
"Gültig bis: $($objCert.GetExpirationDateString())"
"Hashcode: $($objCert.GetHashCode())"
```

```
"Schlüsselalgorithmus: $($objCert.GetKeyAlgorithm())"
"Schlüsselalgorithmusparameter: $($objCert.GetKeyAlgorithmParametersString())"
"Name: $($objCert.GetName())`n"
"Öffentlicher Schlüssel: $($objCert.GetPublicKeyString())`n"
"Rohdaten: $($objCert.GetRawCertDataString())`n"
"Seriennummer: $($objCert.GetSerialNumberString())"
"Zertifikat: $($objCert.ToString())"
"Aussteller: $($objCert.Issuer)"
"Betreff: $($objCert.Subject)"
```

Das vollständige Skript *InspectCertificate.ps1* hat folgenden Inhalt:

InspectCertificate.ps1

```
param($cert, [switch]$help)

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: InspectCertificate.ps1
Ermittelt die Zertifikate für einen bestimmten Verwendungszweck auf dem lokalen Computer.

PARAMETER:
-cert      Der vollständige Pfad zu dem Zertifikat, das untersucht werden soll.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
InspectCertificate.ps1
Generiert eine Fehlermeldung, da ein Zertifikat angegeben werden muss.

InspectCertificate.ps1 -cert "C:\FSO\Filerecovery.cer"

Inspiziert ein Zertifikat namens Filerecovery, das sich im
Verzeichnis C:\FSO befindet. Dieses Zertifikat kann in einer DER-kodierten
oder Base64-kodierten .cer-Datei gespeichert sein.

InspectCertificate.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}

if($help) { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if(!$cert) { "Es ist ein Zertifikat erforderlich ..." ; funHelp }

$objCert=[security.cryptography.x509certificates.x509certificate]"$cert"

"Fingerabdruck: $($objCert.GetCertHashString())"
"Gültig ab: $($objCert.GetEffectiveDateString())"
"Gültig bis: $($objCert.GetExpirationDateString())"
"Hashcode: $($objCert.GetHashCode())"
"Schlüsselalgorithmus: $($objCert.GetKeyAlgorithm())"
"Schlüsselalgorithmusparameter: $($objCert.GetKeyAlgorithmParametersString())"
```

```
"Name: $($objCert.GetName())`n"
"Öffentlicher Schlüssel: $($objCert.GetPublicKeyString())`n"
"Rohdaten: $($objCert.GetRawCertDataString())`n"
"Seriennummer: $($objCert.GetSerialNumberString())"
"Zertifikat: $($objCert.ToString())"
"Aussteller: $($objCert.Issuer)"
"Betreff: $($objCert.Subject)"
```

Importieren eines Zertifikats

Nachdem Sie ein neues Zertifikat erhalten haben, müssen Sie dieses in den Zertifikatspeicher importieren. In Abbildung 16.4 ist der **Zertifikatimport-Assistent** dargestellt. Sie können ein Zertifikat auch mit einem Windows PowerShell-Skript importieren. Dieser Vorgang ist im Skript *ImportCertificate.ps1* veranschaulicht.

Abbildung 16.4 Der *Zertifikatimport-Assistent* importiert Zertifikate standardmäßig in den Zertifikatspeicher *Eigene Zertifikate*

Das Skript *ImportCertificate.ps1* beginnt mit einer *param*-Anweisung, die vier Parameter definiert. Der erste Parameter, **-cert**, gibt den Pfad zu dem zu importierenden Zertifikat an. Der Parameter **-store** verweist standardmäßig auf den Zertifikatspeicher *My*, der im Snap-in **Zertifikate** als **Eigene Zertifikate** bezeichnet ist. Die anderen beiden Parameter haben fest Werte. Der Parameter **-liststores** listet die im Namespace *CurrentUser* verfügbaren Zertifikatspeicher auf. Der Parameter **-help** zeigt die Hilfe an. Die *param*-Anweisung lautet:

```
param(
      $cert,
      $store = "my",
      [switch]$liststores,
      [switch]$help
     )
```

Die nächste Funktion ist *funhelp()*, um eine Hilfemeldung anzuzeigen, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Die Funktion *funhelp()* besteht aus drei Abschnitten in einer *here*-Zeichenfolge, die jeweils die Beschreibung, die Parameter und die Syntax verdeutlichen. Der Inhalt der *here*-Zeichenfolge wird der Variablen *$helptext* zugewiesen und am Ende der Funktion angezeigt. Die Funktion *funhelp()* beendet außerdem die Skriptausführung. Die Funktion ist im Folgenden dargestellt:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ImportCertificate.ps1
Importiert ein Zertifikat in einen Zertifikatspeicher.

PARAMETER:
-cert       Der Pfad zum Zertifikat, das importiert werden soll.
-store      Der Zertifikatspeicher auf dem Computer.
-liststores Listet die Zertifikatspeicher auf dem lokalen Computer auf.
-help       Zeigt dieses Hilfethema an.

SYNTAX:
ImportCertificate.ps1

Generiert eine Fehlermeldung, da ein Zertifikat angegeben werden muss,
und gibt die Hilfeinformationen aus.

ImportCertificate.ps1 -cert "c:\fso\mycert.pfx"

Importiert ein Zertifikat in den Zertifikatspeicher CurrentUser\My,
das im Verzeichnis C:\FSO mit dem Namen Mycert.pfx gespeichert ist.

ImportCertificate.ps1 -store "my" -cert
"c:\fso\mycert.pfx"

Importiert ein Zertifikat in den Zertifikatspeicher CurrentUser\My,
das im Verzeichnis C:\FSO mit dem Namen Mycert.pfx gespeichert ist.

ImportCertificate.ps1 -store "smartcardroot"
-cert "c:\fso\mycert.pfx"

Importiert ein Zertifikat in den Zertifikatspeicher CurrentUser\SmartcardRoot,
das im Verzeichnis C:\FSO mit dem Namen Mycert.pfx gespeichert ist.

ImportCertificate.ps1 -liststores

Zeigt eine Liste aller Zertifikatspeicher an.

ImportCertificate.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Die Funktion *funstore()* zeigt alle Zertifikatspeicher auf dem aktuellen Computer an. Die Funktion beginnt mit dem Zertifikatspeicher *CurrentUser* und endet mit *LocalMachine*. Verwenden Sie das Cmdlet **Get-ChildItem**, um die Liste zu erstellen, und geben Sie mit dem Cmdlet **Write-Host** die Listenüberschriften aus. Die Funktion *funstore()* ist wie folgt implementiert:

```
Function funstore()
{
 write-host -foregroundcolor green "Zertifikatspeicher unter CurrentUser:"
  Get-ChildItem cert:\CurrentUser
  write-host -foregroundcolor green "Zertifikatspeicher unter LocalMachine:`n"
  Get-ChildItem cert:\LocalMachine
  exit
}
```

Überprüfen Sie danach die Befehlszeilenparameter. Suchen Sie als Erstes nach dem Parameter **-help**. Überprüfen Sie, ob die Variable *$help* vorhanden ist. Existiert die Variable *$help*, geben Sie eine Meldung aus und rufen Sie die Funktion *funhelp()* auf. Suchen Sie nach dem Parameter **-liststores**. Wenn die Variable *$liststores* vorhanden ist, rufen Sie die Funktion *funstore()* auf. Überprüfen Sie danach, ob die Variable *$cert* vorhanden ist. Wenn weder der Parameter **-cert** noch die Parameter **-help** oder **-liststores** angegeben sind, geben Sie eine Fehlermeldung aus, dass ein Zertifikat erforderlich ist, rufen Sie die Funktion *funhelp()* auf und beenden Sie das Skript. Die erforderlichen Codezeilen lauten:

```
if($help)  {  "Ausgeben der Hilfeinformationen ..."  ;  funHelp  }
if($liststores) { funStore }
if(!$cert) {
             "Es muss ein Zertifikatspfad angegeben werden ..." ;
             funhelp
           }
```

Sie müssen zwei Variablen deklarieren. Die erste Variable ist *$userstore*. Geben Sie den Wert *CurrentUser* an und legen Sie die Variable als schreibgeschützt fest. Die zweite Variable ist *$crypto*. Weisen Sie der Variablen den Wert einer Zeichenfolge für die gewünschte .NET Framework-Klasse zu. Die beiden Variablenzuweisungen sind im Folgenden dargestellt:

```
new-variable -name userStore -value "currentUser" -option readonly
$crypto = "System.Security.Cryptography.X509Certificates.X509Store"
```

Sie müssen eine Instanz der Klasse *X509Store* erstellen. Verwenden Sie hierzu das Cmdlet **New-Object** und übergeben Sie die in der Variablen *$crypto* gespeicherte Zeichenfolge sowie den Pfad aus der Variablen *$store*. Der letzte Parameter für die Klasse *X509Store* gibt den Zertifikatspeicher in der Variablen *$userstore* an. Weisen Sie das zurückgegebene *X509Store*-Objekt der Variablen *$objstore* zu. Codezeile:

```
$objStore = new-object $crypto $store, $userStore
```

Nachdem Sie das *X509Store*-Objekt angelegt haben, können Sie mit der Methode *Open()* den *readwrite*-Modus auswählen. Rufen Sie danach die Methode *Add()* auf und übergeben Sie an diese Methode die Variable *$cert*, die auf die Instanz des Zertifikatobjekts verweist. Rufen Sie anschließend die Methode *Close()* auf. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
$objstore.Open("ReadWrite")
$objstore.Add($cert)
$objstore.Close()
```

Das vollständige Skript *ImportCertificate.ps1* hat folgenden Inhalt:

ImportCertificate.ps1

```
param(
      $cert,
      $store = "my",
      [switch]$liststores,
      [switch]$help
     )

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ImportCertificate.ps1
Importiert ein Zertifikat in einen Zertifikatspeicher.

PARAMETER:
-cert       Der Pfad zum Zertifikat, das importiert werden soll.
-store      Der Zertifikatspeicher auf dem Computer.
-liststores Listet die Zertifikatspeicher auf dem lokalen Computer auf.
-help       Zeigt dieses Hilfethema an.

SYNTAX:
ImportCertificate.ps1

Generiert eine Fehlermeldung, da ein Zertifikat angegeben werden muss,
und gibt die Hilfeinformationen aus.

ImportCertificate.ps1 -cert "c:\fso\mycert.pfx"

Importiert ein Zertifikat in den Zertifikatspeicher CurrentUser\My,
das im Verzeichnis C:\FSO mit dem Namen Mycert.pfx gespeichert ist.

ImportCertificate.ps1 -store "my" -cert
"c:\fso\mycert.pfx"

Importiert ein Zertifikat in den Zertifikatspeicher CurrentUser\My,
das im Verzeichnis C:\FSO mit dem Namen Mycert.pfx gespeichert ist.

ImportCertificate.ps1 -store "smartcardroot"
-cert "c:\fso\mycert.pfx"

Importiert ein Zertifikat in den Zertifikatspeicher CurrentUser\SmartcardRoot,
das im Verzeichnis C:\FSO mit dem Namen Mycert.pfx gespeichert ist.

ImportCertificate.ps1 -liststores

Zeigt eine Liste aller Zertifikatspeicher an.

ImportCertificate.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
```

```
 exit
}

Function funstore()
{
  write-host -foregroundcolor green "Zertifikatspeicher unter CurrentUser:"
  Get-ChildItem cert:\CurrentUser
  write-host -foregroundcolor green "Zertifikatspeicher unter LocalMachine:`n"
  Get-ChildItem cert:\LocalMachine
  exit
}

if($help)  { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if($liststores) { funStore }
if(!$cert) {
             "Es muss ein Zertifikatspfad angegeben werden ..." ;
             funhelp
           }
new-variable -name userStore -value "currentUser" -option readonly
$crypto = "System.Security.Cryptography.X509Certificates.X509Store"
$objStore = new-object $crypto $store, $userStore
$objstore.Open("ReadWrite")
$objstore.Add($cert)
$objstore.Close()
```

Löschen eines Zertifikats

Wenn ein Zertifikat abgelaufen, die Zertifizierungsstelle nicht mehr vertrauenswürdig oder die Zertifikatkette unterbrochen ist, müssen Sie das entsprechende Zertifikat aus dem Zertifikatspeicher entfernen. Um nur einige Zertifikate zu löschen, können Sie das Snap-in **Zertifikate** verwenden. Zum Entfernen zahlreicher Zertifikate sollten Sie jedoch das Skript *DeleteCertificates.ps1* einsetzen.

Das Skript *DeleteCertificate.ps1* beginnt mit einer *param*-Anweisung für vier Parameter. Der Parameter **-cert** ist erforderlich. Der Parameter **-store** ist standardmäßig auf den Zertifikatspeicher *My* festgelegt. Die nächsten beiden Parameter haben feste Werte. Der Parameter **-listcerts** listet alle Zertifikate im angegebenen Zertifikatspeicher auf. Der Parameter **-help** zeigt die Hilfe an. Die *param*-Anweisung lautet:

```
param(
      $cert,
      $store = "my",
      [switch]$listcerts,
      [switch]$help
     )
```

Die Funktion *funhelp()* gibt eine Hilfemeldung aus, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Erstellen Sie die Variable *$helptext* und weisen Sie dieser eine *here*-Zeichenfolge zu. Definieren Sie in der *here*-Zeichenfolge drei Abschnitte für die Beschreibung, die Parameter und die Syntax. Nachdem Sie die *here*-Zeichenfolge der Variablen *$helptext* zugewiesen haben, zeigen Sie den Inhalt der Variablen an und beenden das Skript. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: DeleteCertificate.ps1
```

```
Entfernt ein Zertifikat aus einem Zertifikatspeicher.

PARAMETER:
-store      Der Zertifikatspeicher auf dem Computer.
-cert       Das zu entfernende Zertifikat.
-listcerts  Listet die Zertifikate im angegebenen Zertifikatspeicher auf.
-help       Zeigt dieses Hilfethema an.

SYNTAX:
DeleteCertificate.ps1

Generiert eine Fehlermeldung, da ein Zertifikat angegeben werden muss,
und gibt die Hilfeinformationen aus.

DeleteCertificate.ps1 -cert "B67BAFECA1E77B8F3AEAB8EB9054D5D31C3C0A03"

Entfernt das Zertifikat mit dem Fingerabdruck
B67BAFECA1E77B8F3AEAB8EB9054D5D31C3C0A03
aus dem Zertifikatspeicher CurrentUser\My.

DeleteCertificate.ps1 -store "my" -cert "OU=EFS File Encryption
Certificate"

Entfernt das Zertifikat mit dem Betreff
OU=EFS File Encryption Certificate
aus dem Zertifikatspeicher CurrentUser\My.

DeleteCertificate.ps1 -store "smartcardroot"
-cert "E47F375796238DB54CB70DA7A5E88F79"

Entfernt das Zertifikat mit der Seriennummer
E47F375796238DB54CB70DA7A5E88F79
aus dem Zertifikatspeicher CurrentUser\SmartcardRoot.

DeleteCertificate.ps1 -listcerts

Erstellt eine Liste der Zertifikate im Zertifikatspeicher
CurrentUser\My.

DeleteCertificate.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Erstellen Sie nun die Funktion *funcert()*, indem Sie zuerst die Variable *$crypto* anlegen, um eine Zeichenfolge mit dem Namespace und dem Namen der .NET Framework-Klasse *X509Store* anzugeben. Erstellen Sie anschließend mit dem Cmdlet **New-Object** eine neue Instanz der Klasse *X509Store*. Der Konstruktor für diese Klasse erfordert eine Pfadangabe, die auf den Namen eines Zertifikatspeichers verweist. Verwenden Sie die Variablen *$store* und *$userstore* und speichern Sie das zurückgegebene *X509Store*-Objekt in der Variablen *$objstore*.

Öffnen Sie den Zertifikatspeicher mit der Methode *Open()* im *readwrite*-Modus, um den Zugriff auf die Zertifikate im Speicher zu ermöglichen. Erstellen Sie eine Liste der Zertifikate, indem Sie die Eigenschaft *Certificates* abfragen, und speichern Sie die Liste in der Variablen *$colcerts*. Geben Sie mit dem Cmdlet **Write-Host** eine Listenüberschrift aus, bevor Sie die Informationen ausgeben. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
Function funcert()
{
 $crypto = "System.Security.Cryptography.X509Certificates.X509Store"
 $objStore = new-object $crypto $store, $userStore
 $objstore.Open("ReadWrite")
 $colcerts = $objstore.Certificates
 Write-Host -ForegroundColor blue
  "
   Es gibt $($colcerts.count) Zertifikate im Zertifikatspeicher $store.
   Die folgenden Zertifikate wurden ermittelt:
  "
```

Nachdem Sie die Listenüberschrift ausgegeben haben, durchlaufen Sie die Liste aus der Variablen *$colcerts*. Verwenden Sie die Variable *$cert* als Enumerator. Speichern Sie jeweils ein Zertifikat in der Variablen *$cert* und geben Sie den Namen, die Seriennummer, den Fingerabdruck und den Betreff aus. Schließen Sie danach den Zertifikatspeicher und beenden Sie das Skript. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
 foreach($cert in $colCerts)
  {
   "Name:  $($cert.FriendlyName)"
   "Seriennummer: $($cert.SerialNumber)"
   "Fingerabdruck: $($cert.thumbprint)"
   "Betreff: $($cert.subject)`n"
  }
 $objstore.Close()
 exit
}
```

Die nächste Funktion ist *findcert()*, um auf ein bestimmtes Zertifikat zuzugreifen. Wenn Sie das Zertifikat finden, speichern Sie das zurückgegebene Zertifikatobjekt in der globalen Variablen *$mycert*. Beginnen Sie die Funktion *findcert()*, indem Sie die Variable *$crypto* erstellen, um die Zeichenfolge für die .NET Framework-Klasse *X509Store* und den Namespace *System.Security.Cryptography.X509Certificates* anzugeben. Erstellen Sie mit dem Cmdlet **New-Object** eine Instanz des *X509Store*-Objekts. Geben Sie die Zeichenfolge mit dem Klassenpfad, die Variable mit dem Pfad zum Zertifikatspeicher und die Variable mit dem Namen des Zertifikatspeichers an. Speichern Sie das zurückgegebene *X509Store*-Objekt in der Variablen *$objstore*. Nachdem Sie das *X509Store*-Objekt erstellt haben, öffnen Sie den Zertifikatspeicher mit der Methode *Open()*. Geben Sie das Schlüsselwort *readwrite* an, um das Ändern des Zertifikatspeichers zuzulassen. Fragen Sie danach die Eigenschaft *Certificates* ab, um die Zertifikate im Speicher zu ermitteln. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
Function findcert($key)
{
 $crypto = "System.Security.Cryptography.X509Certificates.X509Store"
 $objStore = new-object $crypto $store, $userStore
 $objstore.Open("ReadWrite")
 $colcerts = $objstore.Certificates
```

Nachdem Sie die Liste der Zertifikate ermittelt haben, durchlaufen Sie die Liste in der Variablen *$colcerts* mit einer *foreach*-Anweisung. Verwenden Sie die Variable *$cert* als Enumerator. Wenn in der Variablen *$cert* eine Variable gespeichert ist, fragen Sie die Eigenschaften *Thumbprint, SerialNumber, FriendlyName* und *Subject* des Zertifikatobjekts ab und suchen Sie nach einer Übereinstimmung mit dem Wert in der Variablen *$key*. Gibt es eine Übereinstimmung, speichern Sie das Zertifikatobjekt in der globalen Variablen *$mycert*. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
foreach($cert in $colCerts)
 {
  if($cert.thumbprint -match $key)   { $global:mycert = $cert }
  if($cert.serialnumber -match $key) { $global:mycert = $cert }
  if($cert.friendlyname -match $key) { $global:mycert = $cert }
  if($cert.subject -match $key)      { $global:mycert = $cert }
 }
}
```

Im Anschluss an diese Funktionen erstellen Sie die Codezeilen, die unmittelbar im Skript ohne Funktionsaufruf ausgeführt werden. Erstellen Sie als Erstes die Variable *$userstore* und weisen Sie dieser den Wert *CurrentUser* zu. Initialisieren Sie anschließend die Variable *$mycert* als eine globale Variable und weisen Sie dieser den Wert *Null* zu. Die beiden Codezeilen lauten:

```
new-variable -name userStore -value "currentUser" -option readonly
$global:mycert = $null
```

Sie müssen nun die Befehlszeilenparameter überprüfen. Suchen Sie als Erstes nach dem Parameter **-help**. Wenn Sie die Variable *$help* finden, geben Sie eine Meldung aus und rufen Sie die Funktion *funhelp()* auf. Wenn die Variable *$listcerts* vorhanden ist, rufen Sie die Funktion *funcert()* auf. Überprüfen Sie danach, ob die Variable *$cert* existiert. Wenn die Variable nicht gefunden wird, geben Sie eine Fehlermeldung aus und rufen Sie die Funktion *funhelp()* auf. Die erforderlichen Codezeilen lauten:

```
if($help)  {  "Ausgeben der Hilfeinformationen ..."  ;  funHelp  }
if($listcerts) { "Zertifikate im Speicher $store:" ; funcert }
if(!$cert) {
             "Es muss ein Zertifikat angegeben werden ..." ;
             funhelp
           }
```

Wurden die Befehlszeilenparameter korrekt angegeben, rufen Sie die Funktion *findcert()* auf und übergeben Sie den Zertifikatnamen in der Variablen *$cert*. Nachdem Sie das Zertifikatobjekt abgerufen und in der Variablen *$mycert* gespeichert haben, erstellen Sie eine Instanz der .NET Framework-Klasse *X509Store*, öffnen Sie den Zertifikatspeicher, rufen Sie die Methode *Remove()* auf und übergeben Sie das in der Variablen *$mycert* gespeicherte Zertifikatobjekt. Vergessen Sie nicht, nach diesen Schritten den Zertifikatspeicher zu schließen. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
Findcert($cert)

$crypto = "System.Security.Cryptography.X509Certificates.X509Store"
$objStore = new-object $crypto $store, $userStore
$objstore.Open("ReadWrite")
$objstore.remove($mycert)
$objstore.Close()
```

Das vollständige Skript *DeleteCertificate.ps1* hat folgenden Inhalt:

DeleteCertificate.ps1

```
param(
      $cert,
      $store = "my",
      [switch]$listcerts,
      [switch]$help
     )

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: DeleteCertificate.ps1
Entfernt ein Zertifikat aus einem Zertifikatspeicher.

PARAMETER:
-store      Der Zertifikatspeicher auf dem Computer.
-cert       Das zu entfernende Zertifikat.
-listcerts  Listet die Zertifikate im angegebenen Zertifikatspeicher auf.
-help       Zeigt dieses Hilfethema an.

SYNTAX:
DeleteCertificate.ps1

Generiert eine Fehlermeldung, da ein Zertifikat angegeben werden muss,
und gibt die Hilfeinformationen aus.

DeleteCertificate.ps1 -cert "B67BAFECA1E77B8F3AEAB8EB9054D5D31C3C0A03"

Entfernt das Zertifikat mit dem Fingerabdruck
B67BAFECA1E77B8F3AEAB8EB9054D5D31C3C0A03
aus dem Zertifikatspeicher CurrentUser\My.

DeleteCertificate.ps1 -store "my" -cert "OU=EFS File Encryption
Certificate"

Entfernt das Zertifikat mit dem Betreff
OU=EFS File Encryption Certificate
aus dem Zertifikatspeicher CurrentUser\My.

DeleteCertificate.ps1 -store "smartcardroot"
-cert "E47F375796238DB54CB70DA7A5E88F79"

Entfernt das Zertifikat mit der Seriennummer
E47F375796238DB54CB70DA7A5E88F79
aus dem Zertifikatspeicher CurrentUser\SmartcardRoot.

DeleteCertificate.ps1 -listcerts

Erstellt eine Liste der Zertifikate im Zertifikatspeicher
CurrentUser\My.

DeleteCertificate.ps1 -help
```

```
 Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}

Function funcert()
{
 $crypto = "System.Security.Cryptography.X509Certificates.X509Store"
 $objStore = new-object $crypto $store, $userStore
 $objstore.Open("ReadWrite")
 $colcerts = $objstore.Certificates
 Write-Host -ForegroundColor blue
  "
   Es gibt $($colcerts.count) Zertifikate im Zertifikatspeicher $store.
   Die folgenden Zertifikate wurden ermittelt:
  "
 foreach($cert in $colCerts)
  {
   "Name:  $($cert.FriendlyName)"
   "Seriennummer: $($cert.SerialNumber)"
   "Fingerabdruck: $($cert.thumbprint)"
   "Betreff: $($cert.subject)`n"
  }
 $objstore.Close()
 exit
}

Function findcert($key)
{
 $crypto = "System.Security.Cryptography.X509Certificates.X509Store"
 $objStore = new-object $crypto $store, $userStore
 $objstore.Open("ReadWrite")
 $colcerts = $objstore.Certificates

 foreach($cert in $colCerts)
  {
   if($cert.thumbprint -match $key)   { $global:mycert = $cert }
   if($cert.serialnumber -match $key) { $global:mycert = $cert }
   if($cert.friendlyname -match $key) { $global:mycert = $cert }
   if($cert.subject -match $key)      { $global:mycert = $cert }
  }
 }

new-variable -name userStore -value "currentUser" -option readonly
$global:mycert = $null

if($help)  { Ausgeben der Hilfeinformationen ..." ; funHelp }
if($listcerts) { "Zertifikate im Speicher $store:" ; funcert }
if(!$cert) {
             "Es muss ein Zertifikat angegeben werden ..." ;
             funhelp
           }

Findcert($cert)
```

```
$crypto = "System.Security.Cryptography.X509Certificates.X509Store"
$objStore = new-object $crypto $store, $userStore
$objstore.Open("ReadWrite")
$objstore.remove($mycert)
$objstore.Close()
```

Zusammenfassung

In diesem Kapitel wurden die verschiedenen Methoden für die Arbeit mit Zertifikatdiensten beschrieben. Sie haben im Zertifikatspeicher nach einem bestimmten Zertifikat gesucht, unter Verwendung der .NET Framework-Klassen die Zertifikate aus den Zertifikatspeichern aufgelistet sowie nach abgelaufenen und in Kürze ablaufenden Zertifikaten gesucht.

Anschließend wurden die Aufgaben für die Zertifikatverwaltung erklärt, einschließlich dem Überprüfen und Importieren von Zertifikaten. Das Kapitel wurde mit dem Löschen von Zertifikaten aus den Zertifikatspeichern abgeschlossen.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

KAPITEL 17

Verwalten der Terminaldienste

Nach Abschluss dieses Kapitels können Sie:

- Die Windows-Terminaldienste konfigurieren
- Die Netzwerkprotokolle der Terminaldienste überprüfen
- Die Benutzereinstellungen für Terminalserver konfigurieren

Auf der Begleit-CD Alle Skripts in diesem Kapitel befinden sich auf der Begleit-CD im Ordner *\Scripts\Chapter17*.

Konfigurieren der Terminaldienst-Installation

Wenn die Serverrolle **Terminaldienste** auf Windows Server 2008 installiert ist, können Benutzer auf die Desktopumgebung von Windows zugreifen. Zahlreiche konfigurierbare Einstellungen können dabei helfen, die Leistung, Skalierbarkeit und Benutzerfreundlichkeit zu verbessern. Abhängig von der Umgebung können sich diese Einstellungen jedoch gegenseitig ausschließen. Deshalb verbringen Netzwerkadministratoren oft viel Zeit damit, eine optimale Kombination der Einstellungen anhand der gegebenen Benutzeranforderungen zu ermitteln.

Dokumentieren der Terminaldienstekonfiguration

Da zahlreiche Einstellungen konfiguriert werden können, ist es wichtig, die Konfiguration der Windows Server 2008-Terminalserver zu überprüfen. Verwenden Sie hierzu das in Abbildung 17.1 dargestellte Dienstprogramm für die Terminaldienstekonfiguration.

Wenn mehrere Terminalserver vorhanden sind, können Sie ein Skript erstellen, um diesen Vorgang zu automatisieren. Verwenden Sie hierzu die WMI-Klassen im WMI-Namespace *root\cimv2\terminal-services*. Die wichtigste WMI-Klasse ist *Win32_TerminalServiceSetting*, da diese eine Übersicht der allgemeinen Konfigurationseinstellungen bietet. Das Beispielskript *ReportTerminalServiceSetting.ps1* wurde unter Verwendung der WMI-Klasse *Win32_TerminalServiceSetting* erstellt.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

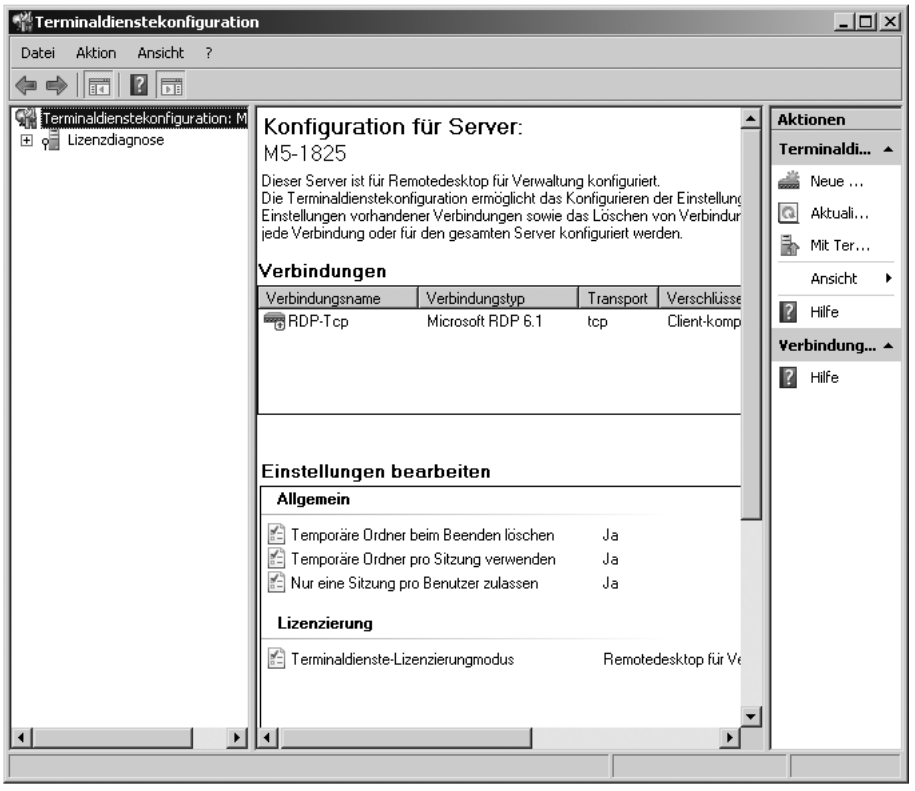


Abbildung 17.1 Mit dem Dienstprogramm *Terminaldienstekonfiguration* können Sie auf die Konfigurationsinformationen zugreifen

Beginnen Sie das Skript *ReportTerminalServiceSetting.ps1* mit einer *param*-Anweisung, um Parameter bei der Befehlseingabe zu unterstützen. Das Skript umfasst zwei Parameter, um den Computer anzugeben, für den das Skript ausgeführt werden soll, bzw. einen Hilfetext anzuzeigen. Der Parameter **-help** ist ein feststehender Parameter, für den kein Wert angegeben werden kann, da es sich um einen booleschen Datentyp handelt, der True/False, Ja/Nein oder 1/0 ist. Die *param*-Anweisung lautet:

```
param(
      $computer = "localhost",
      [switch]$help
     )
```

Erstellen Sie eine Funktion, um die Hilfeinformationen für den Benutzer anzuzeigen. Beginnen Sie mit der *function*-Anweisung, um die neue Funktion zu erstellen, und geben Sie dieser den Namen *funhelp()*. Erstellen Sie im Codeblock für die Funktion die Variable *$helptext* und weisen Sie dieser Variablen eine *here*-Zeichenfolge zu. Die *here*-Zeichenfolge unterliegt einer speziellen Konvention, die das Erstellen von Zeichenfolgen aus beliebigem Text ermöglicht. Die Zeichenfolge kann Sonderzeichen und Anführungszeichen umfassen. Sie müssen für die Sonderzeichen weder Escape-Zeichen noch umschließende Anführungszeichen eingeben. Außerdem können Sie eine Zeichenfolge mitten im Satz beenden. Der Inhalt der *here*-Zeichenfolge entspricht einem einzelnen Zeichenfolgenwert. Die *here*-Zeichenfolge beginnt mit @", endet mit "@ und enthält drei Textabschnitte für die Hilfemeldung: Beschreibung, Parameter und Syntax. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ReportTerminalServiceSetting.ps1
Zeigt die Terminalservereinstellungen auf dem lokalen Computer
oder einem Remote-Terminalserver an.

PARAMETER:
-computer Der Name des Computers, für den das Skript ausgeführt werden soll.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
ReportTerminalServiceSetting.ps1
Zeigt die Terminalservereinstellungen auf dem lokalen Computer an.

ReportTerminalServiceSetting.ps1 -computer ts1

Zeigt die Terminalservereinstellungen
auf einem Remote-Terminalserver namens ts1 an.

ReportTerminalServiceSetting.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Überprüfen Sie als Nächstes die Befehlszeilenparameter. Wurde der Parameter **-help** angegeben, existiert die Variable *$help*. Geben Sie in diesem Fall eine Statusmeldung aus und rufen Sie die Funktion *funhelp()* auf. Codezeile:

```
if($help)       { "Ausgeben der Hilfeinformationen ..." ; funHelp }
```

Erstellen Sie danach die beiden Variablen *$namespace* und *$class*. *$namespace* legt den Wert für den Parameter **-namespace** des Cmdlets **Get-WmiObject** fest. *$class* definiert den Wert des Parameters **-class** des Cmdlets **Get-WmiObject**. Die beiden Variablenzuweisungen sind im Folgenden dargestellt:

```
$namespace = "root\cimv2\TerminalServices"
$class = "win32_TerminalServiceSetting"
```

Greifen Sie mit dem Cmdlet **Get-WmiObject** auf den WMI-Namespace *root\cimv2\terminalservices* zu. Stellen Sie die Verbindung mit dem in der Variablen *$computer* angegebenen Computer her und geben Sie ein Objekt zurück, das die Instanzen der WMI-Klasse *Win32_TerminalServiceSetting* enthält. Übergeben Sie das zurückgegebene Objekt an das Cmdlet **Format-List** und geben Sie einen Filter an, der ausschließlich die Elemente zurückgibt, die mit den Buchstaben *a* bis *z* beginnen. Dieser Skriptabschnitt ist im Folgenden dargestellt:

```
get-wmiobject -namespace $namespace -computername $computer `
          -class $class  |
  format-list [a-z]*
```

Das vollständige Skript *ReportTerminalServiceSetting.ps1* hat folgenden Inhalt:

ReportTerminalServiceSetting.ps1

```
param(
      $computer = "localhost",
      [switch]$help
     )

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ReportTerminalServiceSetting.ps1
Zeigt die Terminalservereinstellungen auf dem lokalen Computer
oder einem Remote-Terminalserver an.

PARAMETER:
-computer Der Name des Computers, für den das Skript ausgeführt werden soll.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
ReportTerminalServiceSetting.ps1
Zeigt die Terminalservereinstellungen auf dem lokalen Computer an.

ReportTerminalServiceSetting.ps1 -computer ts1

Zeigt die Terminalservereinstellungen
auf einem Remote-Terminalserver namens ts1 an.

ReportTerminalServiceSetting.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}

if($help)       { "Ausgeben der Hilfeinformationen ..." ; funHelp }

$namespace = "root\cimv2\TerminalServices"
$class = "win32_TerminalServiceSetting"

get-wmiobject -namespace $namespace -computername $computer `
          -class $class  |
  format-list [a-z]*
```

Deaktivieren der Anmeldung

Sie müssen den Terminalserver oft so konfigurieren, dass keine neuen Benutzeranmeldungen mehr zugelassen werden. Dies ist beispielsweise vor einer geplanten Wartung erforderlich. Das Ziel ist, die Anzahl der mit dem Server verbundenen Benutzer zu reduzieren. Vorhandene Benutzer melden sich ab und neue Benutzer dürfen sich nicht mehr anmelden. Diese Einstellung kann mit dem Skript *DisableLogons.ps1* auf einem Terminalserver konfiguriert werden.

Beginnen Sie das Skript *DisableLogons.ps1* mit einer *param*-Anweisung. Vier der fünf Parameter sind feststehende Parameter. Der verbleibende Parameter wird für den Computernamen verwendet und ist auf den Standardwert *localhost* festgelegt. Wenn der Parameter **-allow** angegeben ist, werden Anmeldungen zugelassen. Der Parameter **-disallow** deaktiviert die Anmeldungen. Der Parameter **-list** erstellt eine Liste der aktuellen Terminaldienstekonfiguration. Der Parameter **-help** zeigt den Hilfetext an. Die *param*-Anweisung lautet:

```
param(
      $computer = "localhost",
      [switch]$allow,
      [switch]$disallow,
      [switch]$list,
      [switch]$help
     )
```

Erstellen Sie anschließend die Funktion *funhelp()*, die nur aufgerufen wird, wenn die Variable *$help* beim Parsen der Befehlszeile gefunden wird. Verwenden Sie in der Funktion *funhelp()* eine *here*-Zeichenfolge, um den Hilfetext zu definieren, und weisen Sie diese Zeichenfolge der Variablen *$helptext* zu. Die Funktion zeigt daraufhin den Inhalt der Variablen an und beendet die Skriptausführung. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: DisableLogons.ps1
Konfiguriert die Einstellungen für Clientsitzungen für die Computer,
die auf den lokalen Computer oder einen Remote-Terminalserver zugreifen.

PARAMETER:
-computer Der Computer, für den das Skript ausgeführt werden soll.
-disallow Deaktiviert neue Anmeldungen auf dem Terminalserver.
-allow    Aktiviert neue Anmeldungen auf dem Terminalserver.
-list     Zeigt die aktuelle Konfiguration an.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
DisableLogons.ps1
Zeigt eine Fehlermeldung an, dass eine Einstellung angegeben werden muss,
und gibt die Hilfeinformationen aus.

DisableLogons.ps1 -list

Listet die aktuellen Einstellungen für Clientsitzungen auf dem lokalen Terminalserver auf.

DisableLogons.ps1 -allow -computer TS2

Konfiguriert einen Remote-Terminalserver namens TS2,
um neue Anmeldungen zuzulassen.

DisableLogons.ps1 -disallow

Konfiguriert den lokalen Terminalserver,
```

```
um neue Anmeldungen zu unterbinden.

DisableLogons.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Erstellen Sie als Nächstes die Funktion *funlist()*. Diese Funktion listet die aktuelle Konfiguration auf, wenn das Skript mit dem Parameter **-list** ausgeführt wird. Verwenden Sie in der Funktion *funlist()* das Cmdlet **Get-WmiObject**, um auf den in der Variablen *$namespace* angegebenen Namespace zuzugreifen. Bestimmen Sie mittels des Werts in der Variablen *$computer*, zu welchem Computer das WMI-Skript eine Verbindung herstellen muss. Legen Sie anschließend mit dem Wert in der Variablen *$class* fest, welche WMI-Klasse abgefragt wird. Die Werte für diese Variablen werden später im Skript, außerhalb der Funktion definiert. Übergeben Sie das resultierende WMI-Verwaltungsobjekt an das Cmdlet **Format-List**. Ignorieren Sie die Systemeigenschaften, denen ein Unterstrich (_) vorangestellt ist. Beenden Sie anschließend das Skript. Die Funktion *funlist()* ist wie folgt implementiert:

```
Function funlist()
{
 get-wmiobject -namespace $namespace -computername $computer `
    -class $class  |
 format-list [a-z]*
 exit
}
```

Nachdem Sie die Funktion *funlist()* erstellt haben, implementieren Sie die Funktion *funchange()*, um Konfigurationsänderungen am Terminalserver vorzunehmen. Die Funktion verwendet das Cmdlet **Get-WmiObject,** um auf die WMI-Klasse *Win32_TerminalServiceSetting* zuzugreifen, wie in der Variablen *$class* angegeben. Die WMI-Klasse *Win32_TerminalServiceSetting* befindet sich im WMI-Namespace *root\cimv2\terminalservices*, der von der Variablen *$namespace* angegeben wird. Stellen Sie eine WMI-Verbindung mit dem in der Variablen *$computer* angegebenen Computer her (standardmäßig *localhost*) und weisen Sie das resultierende Verwaltungsobjekt der Variablen *$objTS* zu. Fragen Sie die Eigenschaft *Logons* ab und weisen Sie die in der Variablen *$action* angegebene Aktion zu. Übernehmen Sie die Änderungen mit der *Put()*-Methode in die WMI-Datenbank und beenden Sie das Skript. Die Funktion *funchange()* ist wie folgt implementiert:

```
Function Funchange()
{
 $objTS = get-wmiobject -class $class -namespace $namespace `
            -computername $computer
 $objTS.logons = $action
 $objTS.put()
 exit
}
```

Sie müssen noch die beiden Variablen *$namespace* und *$class* erstellen. *$namespace* teilt dem Cmdlet **Get-WmiObject** mit, wo sich die WMI-Klasse befindet. Die zweite Variable repräsentiert die WMI-Klasse, die abgefragt wird. Die beiden Codezeilen lauten:

```
$namespace = "root\cimv2\TerminalServices"
$class = "win32_TerminalServiceSetting"
```

Nachdem Sie die beiden Variablen erstellt haben, überprüfen Sie die Befehlsparameter. Wenn Sie die Variable *$help* finden, rufen Sie die Funktion *funhelp()* auf. Ist die Variable *$list* vorhanden, rufen Sie die Funktion *funlist()* auf und geben Sie die aktuelle Konfiguration aus. Existiert die Variable *$allow*, weisen Sie der Variablen *$action* den Wert *1* zu und rufen die Funktion *funchange()* auf. Ist die Variable *$disallow* vorhanden, weisen Sie der Variablen *$action* den Wert *0* zu und rufen Sie ebenfalls die Funktion *funchange()* auf. Sollten diese Variablen nicht gefunden werden, geben Sie eine Meldung aus, die den Benutzer anweist, die Hilfe zu überprüfen. Dieser Skriptabschnitt ist im Folgenden dargestellt:

```
if($help)     { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if($list)     {     funlist     }
if($allow)    { $action = 1 ; funchange }
if($disallow) { $action = 0 ; funchange }

"Es wurde keine Aktion angegeben. Für weitere Informationen führen Sie folgenden Befehl aus: DisableLogons.ps1 -help."
```

Das vollständige Skript *DisableLogons.ps1* hat folgenden Inhalt:

DisableLogons.ps1

```
param(
      $computer = "localhost",
      [switch]$allow,
      [switch]$disallow,
      [switch]$list,
      [switch]$help
     )

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: DisableLogons.ps1
Konfiguriert die Einstellungen für Clientsitzungen für die Computer,
die auf den lokalen Computer oder einen Remote-Terminalserver zugreifen.

PARAMETER:
-computer Der Computer, für den das Skript ausgeführt werden soll.
-disallow Deaktiviert neue Anmeldungen auf dem Terminalserver.
-allow    Aktiviert neue Anmeldungen auf dem Terminalserver.
-list     Zeigt die aktuelle Konfiguration an.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
DisableLogons.ps1
Zeigt eine Fehlermeldung an, dass eine Einstellung angegeben werden muss,
und gibt die Hilfeinformationen aus.

DisableLogons.ps1 -list

Listet die aktuellen Einstellungen für Clientsitzungen auf dem lokalen Terminalserver auf.

DisableLogons.ps1 -allow -computer TS2

Konfiguriert einen Remote-Terminalserver namens TS2,
um neue Anmeldungen zuzulassen.
```

```
DisableLogons.ps1 -disallow

Konfiguriert den lokalen Terminalserver,
um neue Anmeldungen zu unterbinden.

DisableLogons.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}

Function funlist()
{
 get-wmiobject -namespace $namespace -computername $computer `
    -class $class  |
 format-list [a-z]*
 exit
}

Function Funchange()
{
 $objTS = get-wmiobject -class $class -namespace $namespace `
           -computername $computer
 $objTS.logons = $action
 $objTS.put()
 exit
}

$namespace = "root\cimv2\TerminalServices"
$class = "win32_TerminalServiceSetting"

if($help)     { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if($list)     {     funlist     }
if($allow)    { $action = 1 ; funchange }
if($disallow) { $action = 0 ; funchange }

"Es wurde keine Aktion angegeben. Für weitere Informationen führen Sie folgenden Befehl aus: DisableLogons.ps1 -help."
```

Ändern der Clienteigenschaften

Sie können zahlreiche Clienteinstellungen ändern. Das Dienstprogramm **Terminaldienstekonfiguration** umfasst Registerkarten mit Einstellungen, die verschiedene Aspekte der Clientkonfiguration steuern. Einige dieser Einstellungen betreffen die Zuordnung von Peripheriegeräten (siehe Abbildung 17.2).

Sie können die in Abbildung 17.2 dargestellten Einstellungen mit dem Skript *ConfigureClientProperties.ps1* ändern. Das Skript *ConfigureClientProperties.ps1* verwendet die WMI-Klasse *Win32_TSClientSetting* aus dem WMI-Namespace *root\cimv2\terminalservices*. Da das Skript mehrere feststehende Parameter umfasst, ist die Ausführung über die Befehlszeile einfach.

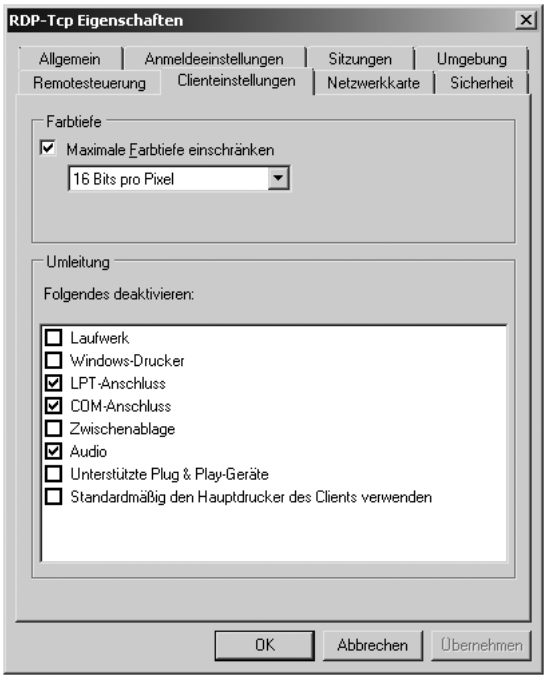


Abbildung 17.2 Die im Dienstprogramm *Terminaldienstekonfiguration* angezeigten Clienteinstellungen

Beginnen Sie das Skript *ConfigureClientProperties.ps1* mit einer *param*-Anweisung, um mehrere Parameter zu erstellen. Der erste Parameter ist **-computer**, der standardmäßig auf *localhost* festgelegt ist. Der nächste Parameter ist **-action**, der die auszuführende Aktion angibt. Die nächsten beiden Parameter, **-enable** und **-disable**, betreffen den Parameter **-action**. Der Parameter **-list** listet die aktuelle Konfiguration auf. Der Parameter **-help** zeigt den Hilfetext an. Die *param*-Anweisung lautet:

```
param(
      $computer = "localhost",
      $action,
      [switch]$enable,
      [switch]$disable,
      [switch]$list,
      [switch]$help
     )
```

Definieren Sie die Funktion *funhelp()*, um den Inhalt der Variablen *$helptext* anzuzeigen, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Die Variable *$helptext* enthält eine *here*-Zeichenfolge mit den Informationen zur Skriptausführung, einschließlich der Beschreibung, Parameter und Syntax. Nachdem die Variable *$helptext* angezeigt wurde, wird das Skript beendet. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ConfigureClientProperties.ps1
Konfiguriert die Clienteinstellungen für LPTPortMapping, COMPortMapping
AudioMapping, ClipboardMapping, DriveMapping und WindowsPrinterMapping
```

```
für die Computer, die auf den lokalen Computer
oder einen Remote-Terminalserver zugreifen.

PARAMETER:
-computer  Der Computer, für den das Skript ausgeführt werden soll.
-action    Die Art der Ressourcenzuordnung
           < LPT, COM, Audio, Zwischenablage, Laufwerk, Drucker >
-enable    Aktiviert die unter -action angegebene Ressourcenzuordnung.
-disable   Deaktiviert die unter -action angegebene Ressourcenzuordnung.
-list      Zeigt die aktuelle Konfiguration an.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
ConfigureClientProperties.ps1
Zeigt eine Fehlermeldung an, dass eine Einstellung angegeben werden muss,
und gibt die Hilfeinformationen aus.

ConfigureClientProperties.ps1 -list

Listet die aktuellen Einstellungen für Clientsitzungen auf dem lokalen Terminalserver auf.

ConfigureClientProperties.ps1 -action com -disable -computer TS2

Konfiguriert die Clienteinstellungen auf einem Remote-Terminalserver namens TS2,
um die COM-Portzuordnung für Clients zu deaktivieren.

ConfigureClientProperties.ps1 -action lpt -enable

Konfiguriert die Clienteinstellungen auf dem lokalen Terminalserver,
um die LPT-Portzuordnung für Clients zu aktivieren.

ConfigureClientProperties.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Die nächste Funktion ist *funlist()*, die das Cmdlet **Get-WmiObject** verwendet, um auf den in der Variablen *$namespace* angegebenen Namespace zuzugreifen. Die Funktion stellt eine Verbindung zu dem in der Variablen *$computer* angegebenen Computer her und fragt die in der Variablen *$class* angegebenen Klassen ab. Die Ergebnisse werden in das Cmdlet **Format-List** eingefügt und das Skript wird beendet. Code:

```
Function funlist()
{
 get-wmiobject -namespace $namespace -computername $computer `
    -class $class  |
 format-list [a-z]*
 exit
}
```

Sie müssen darüber hinaus den beiden Variablen die in den vorherigen Funktionen verwendeten Werte zuweisen. Der Wert der Variablen *$namespace* ist auf *root\cimv2\terminalservices* festgelegt. Dies ist der Namespace mit dem meisten WMI-Klassen für die Terminaldienste. Weisen Sie den Namen der WMI-Klasse *Win32_TSClientSetting* der Variablen *$class* zu. Code:

```
$namespace = "root\cimv2\TerminalServices"
$class = "win32_TSClientSetting"
```

Sie müssen nun wieder die Befehlszeile auswerten. Überprüfen Sie als Erstes, ob die Variable *$help* vorhanden ist. Rufen Sie in diesem Fall die Funktion *funhelp()* auf. Existiert die Variable *$list*, rufen Sie die Funktion *funlist()* auf. Sollte die Variable *$action* nicht vorhanden sein, rufen Sie die Funktion *funhelp()* auf. Ist die Variable *$disable* vorhanden, weisen Sie *$value* den Wert *0* zu. Weisen Sie stattdessen *$value* den Wert *1* zu, wenn Sie die Variable *$enable* finden. Dieser Skriptabschnitt ist im Folgenden dargestellt:

```
if($help) { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if($list) { funlist }
if(!$action) { "Sie müssen eine Aktion angeben." ; funhelp }
if($disable) { $value = 0 }
if($enable)  { $value = 1 }
```

Verwenden Sie eine *switch*-Anweisung, um die Auswertung der Befehlszeile übersichtlicher zu gestalten. Erstellen Sie Aliasnamen für die Eigenschaften, damit der Benutzer keine langen Namen eingeben muss. Diese Werte werden an den Parameter *$action* übergeben. Die *switch*-Anweisung lautet:

```
switch($action)
{
 "LPT"     { $action = "LPTPortMapping" }
 "COM"     { $action = "COMPortMapping" }
 "Audio"   { $action = "AudioMapping" }
 "Zwischenablage"    { $action = "ClipboardMapping" }
 "Laufwerk"   { $action = "DriveMapping" }
 "Drucker" { $action = "WindowsPrinterMapping  " }
}
```

Nachdem Sie der Variablen *$action* den entsprechenden Wert zugewiesen haben, müssen Sie den entsprechenden WMI-Befehl ausführen. Stellen Sie hierzu mit dem Cmdlet **Get-WmiObject** eine Verbindung zur WMI-Klasse *Win32_TSClientSetting* im Namespace *root\cimv2\terminalservices* auf dem in der Variablen *$computer* angegebenen Computer her. Nachdem Sie das resultierende Objekt in der Variablen *$objClient* gespeichert haben, rufen Sie die Methode *SetClientProperty()* auf und übergeben die in der Variablen *$action* angegebene Eigenschaft sowie den *Ja/Nein*-Wert der Variablen *$value*. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
$objClient=get-wmiobject -namespace $namespace -computername $computer `
          -class $class
$objClient.SetClientProperty($action, $value)
```

Das vollständige Skript *ConfigureClientProperties.ps1* hat folgenden Aufbau:

ConfigureClientProperties.ps1

```
param(
      $computer = "localhost",
      $action,
      [switch]$enable,
      [switch]$disable,
      [switch]$list,
```

```
      [switch]$help
     )

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ConfigureClientProperties.ps1
Konfiguriert die Clienteinstellungen für LPTPortMapping, COMPortMapping
AudioMapping, ClipboardMapping, DriveMapping und WindowsPrinterMapping
für die Computer, die auf den lokalen Computer
oder einen Remote-Terminalserver zugreifen.

PARAMETER:
-computer  Der Computer, für den das Skript ausgeführt werden soll.
-action    Die Art der Ressourcenzuordnung
           < LPT, COM, Audio, Zwischenablage, Laufwerk, Drucker >
-enable    Aktiviert die unter -action angegebene Ressourcenzuordnung.
-disable   Deaktiviert die unter -action angegebene Ressourcenzuordnung.
-list      Zeigt die aktuelle Konfiguration an.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
ConfigureClientProperties.ps1
Zeigt eine Fehlermeldung an, dass eine Einstellung angegeben werden muss,
und gibt die Hilfeinformationen aus.

ConfigureClientProperties.ps1 -list

Listet die aktuellen Einstellungen für Clientsitzungen auf dem lokalen Terminalserver auf.

ConfigureClientProperties.ps1 -action com -disable -computer TS2

Konfiguriert die Clienteinstellungen auf einem Remote-Terminalserver namens TS2,
um die COM-Portzuordnung für Clients zu deaktivieren.

ConfigureClientProperties.ps1 -action lpt -enable

Konfiguriert die Clienteinstellungen auf dem lokalen Terminalserver,
um die LPT-Portzuordnung für Clients zu aktivieren.

ConfigureClientProperties.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}

Function funlist()
{
 get-wmiobject -namespace $namespace -computername $computer `
    -class $class  |
 format-list [a-z]*
 exit
```

```
}

$namespace = "root\cimv2\TerminalServices"
$class = "win32_TSClientSetting"

if($help) { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if($list) { funlist }
if(!$action) { "Sie müssen eine Aktion angeben." ; funhelp }
if($disable) { $value = 0 }
if($enable)  { $value = 1 }

switch($action)
{
 "LPT"     { $action = "LPTPortMapping" }
 "COM"     { $action = "COMPortMapping" }
 "Audio"   { $action = "AudioMapping" }
 "Zwischenablage"    { $action = "ClipboardMapping" }
 "Laufwerk"   { $action = "DriveMapping" }
 "Drucker" { $action = "WindowsPrinterMapping  " }
}

$objClient=get-wmiobject -namespace $namespace -computername $computer `
          -class $class
$objClient.SetClientProperty($action, $value)
```

Verwalten von Benutzern

Für die Benutzerkonten, die auf Windows Server 2008-Terminalserver zugreifen, können zahlreiche Einstellungen direkt konfiguriert werden, einschließlich der Zugriff auf den Terminalserver. Andere Einstellungen wirken sich auf die Qualität der Benutzerumgebung und die Leistung des Terminalservers aus. Diese Einstellungen umfassen, aber sind nicht beschränkt auf, die Desktopfarbtiefe und die verfügbaren Desktopfeatures. Die Verwaltung dieser Einstellungen wird in diesem Abschnitt beschrieben.

Mit dem Skript *ReportClientSettings.ps1* können Sie die Konfigurationsinformationen für Clientobjekte anzeigen. Beginnen Sie mit einer *param*-Anweisung, die zwei Befehlszeilenparameter definiert. Der erste Parameter, **-computer**, legt fest, auf welchem Computer das Skript ausgeführt wird. Der andere Parameter ist **-help**, um die Hilfe anzuzeigen. Die beiden Befehlszeilenparameter sind wie folgt definiert:

```
param(
      $computer = "localhost",
      [switch]$help
     )
```

Die Funktion *funhelp()* zeigt die Hilfeinformationen an, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Weisen Sie den Inhalt der Variablen *$helptext* über eine *here*-Zeichenfolge der Variablen *$helptext* zu. Die Hilfeinformationen bestehen aus drei Abschnitten. Der erste Abschnitt ist die Beschreibung, der zweite Abschnitt umfasst die Parameter und der dritte Abschnitt enthält Syntaxbeispiele. Nachdem der Inhalt der Variablen *$helptext* angezeigt wurde, wird das Skript beendet. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ReportClientSettings.ps1
Zeigt die Clientkonfigurationseinstellungen
auf dem lokalen Computer oder einem Remote-Terminalserver an.

PARAMETER:
-computer Der Computer, für den das Skript ausgeführt werden soll.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
ReportClientSettings.ps1
Zeigt die Clientkonfigurationseinstellungen  auf dem lokalen Computer an.

ReportClientSettings.ps1 -computer ts1

Zeigt die Clientkonfigurationseinstellungen
auf einem Remote-Terminalserver namens TS1 an.

ReportClientSettings.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Wenn die Variable *$help* vorhanden ist, rufen Sie die Funktion *funhelp()* auf:

```
if($help){ "Ausgeben der Hilfeinformationen ..." ; funHelp }
```

Sie müssen außerdem zwei Variablen erstellen, um das Verhalten des Cmdlets **Get-WmiObject** zu steuern. Die erste Variable ist *$namespace*, die den Namespace festlegt, aus dem das Skript die Klasseninformationen bezieht. Die zweite Variable entspricht dem Namen der abgefragten WMI-Klasse. Die beiden Codezeilen lauten:

```
$namespace = "root\cimv2\TerminalServices"
$class = "win32_TSClientSetting"
```

Sie müssen nun eine Verbindung zu WMI herstellen. Verwenden Sie hierzu das Cmdlet **Get-WmiObject** mit dem Parameter **-namespace**, um auf den in der Variablen *$namespace* angegebenen Namespace zuzugreifen. Stellen Sie die Verbindung mit dem in der Variablen *$computer* angegebenen Computer her und fragen Sie die Klasse entsprechend der Variablen *$class* ab. Fügen Sie dann das Objekt zwecks Ausgabe in das Cmdlet **Format-List** ein. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
get-wmiobject -namespace $namespace -computername $computer `
          -class $class  |
  format-list [a-z]*
```

Das vollständige Skript *ReportClientSettings.ps1* hat folgenden Inhalt:

ReportClientSettings.ps1

```
param(
      $computer = "localhost",
      [switch]$help
     )
```

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ReportClientSettings.ps1
Zeigt die Clientkonfigurationseinstellungen
auf dem lokalen Computer oder einem Remote-Terminalserver an.

PARAMETER:
-computer Der Computer, für den das Skript ausgeführt werden soll.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
ReportClientSettings.ps1
Zeigt die Clientkonfigurationseinstellungen  auf dem lokalen Computer an.

ReportClientSettings.ps1 -computer ts1

Zeigt die Clientkonfigurationseinstellungen
auf einem Remote-Terminalserver namens TS1 an.

ReportClientSettings.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}

if($help){ "Ausgeben der Hilfeinformationen ..." ; funHelp }

$namespace = "root\cimv2\TerminalServices"
$class = "win32_TSClientSetting"

get-wmiobject -namespace $namespace -computername $computer `
          -class $class  |
  format-list [a-z]*
```

Benutzern den Zugriff auf den Server gewähren

Sie müssen den Benutzern explizit den Zugriff auf den Server ermöglichen, denn standardmäßig können die Benutzer nicht auf Terminalserver zugreifen. Um Benutzern die Berechtigungen für den Zugriff auf Terminalserver zu gewähren, können Sie das Skript *GrantUserTSPermission.ps1* verwenden.

Beginnen Sie das Skript *GrantUserTSPermission.ps1* mit einer *param*-Anweisung. Erstellen Sie als Erstes den Parameter **-computer**. Legen Sie einen Standardwert für den Parameter fest, indem Sie der Variablen *$computer* die Zeichenfolge *localhost* zuweisen. Erstellen Sie außerdem die Parameter **-user** und **-level**. Diese beiden Parameter steuern den Benutzerzugriff und welche Aktivitäten diese Benutzer ausführen können. Der letzte Parameter ist **-help**, um die Hilfe anzuzeigen. Die *param*-Anweisung lautet:

```
param(
      $computer = "localhost",
      $user,
      $level,
      [switch]$help
     )
```

Die Funktion *funhelp()* zeigt die Hilfeinformationen an, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Erstellen Sie als Erstes die Variable *$helptext* und weisen Sie dieser eine *here*-Zeichenfolge zu. Der Text der *here*-Zeichenfolge ist in drei Abschnitte aufgeteilt: Beschreibung, Parameter und Syntax. Diese Abschnitte bestehen nur aus Text. Nachdem Sie die *here*-Zeichenfolge definiert haben, zeigen Sie den Inhalt der Variablen *$helptext* an und beenden Sie das Skript. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GrantUserTSPermission.ps1
Gewährt einem Benutzerkonto Zugriffsberechtigungen
auf dem lokalen Terminalserver oder einem Remote-Terminalserver.

PARAMETER:
-computer Der Computer, für den das Skript ausgeführt werden soll.
-user     Das Benutzerkonto, dem die Berechtigungen gewährt werden sollen.
-level    Die zu gewährende Zugriffsebene: < Gast, Benutzer, Alle >.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
GrantUserTSPermission.ps1
Zeigt eine Fehlermeldung an, dass eine Einstellung angegeben werden muss,
und gibt die Hilfeinformationen aus.

GrantUserTSPermission.ps1 -user bob -level Gast

Gewährt dem Benutzer Bob Gastberechtigungen auf dem lokalen Terminalserver.

GrantUserTSPermission.ps1 -user sandra -level Benutzer -computer ts1

Gewährt dem Benutzer Sandra
Benutzerberechtigungen auf dem Remote-Terminalserver TS1.

GrantUserTSPermission.ps1 -user ed -level Alle

Gewährt dem Benutzer Ed alle Berechtigungen auf dem lokalen Terminalserver.

GrantUserTSPermission.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Überprüfen Sie nun die Befehlszeilenoptionen. Wenn die Variable *$help* vorhanden ist, geben Sie eine Statusmeldung aus und rufen Sie die Funktion *funhelp()* auf. Ist die Variable *$user* nicht vorhanden, geben Sie ebenfalls eine Meldung aus und rufen Sie die Funktion *funhelp()* auf. Führen Sie die gleiche Aktion aus, wenn die Variable *$level* nicht existiert. Die beiden folgenden Parameter sind erforderlich: *$user* und *$level*. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
if($help)       { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if(!$user)      { "Ein Benutzerkonto muss angegeben werden ..." ; funHelp }
if(!$level)     { "Die Zugriffsebene muss angegeben werden ..." ; funHelp }
```

Parsen Sie nun die Variable *$level*. Für die Variable *$level* sind drei Werte zulässig. Diese Werte entsprechen den drei Zugriffsebenen für Terminalserver. Die WMI-Klasse erwartet eine ganze Zahl als Wert. Um die Verwendung des Skripts zu vereinfachen, lassen Sie die Eingabe der Benutzernamen und Kennwörter über die Befehlszeile zu. Mit einer *switch*-Anweisung können Sie die Benutzereingabe in die für WMI erforderlichen Werte umwandeln. Die *switch*-Anweisung lautet:

```
switch($level)
{
  "Gast" { $level = 0 }
  "Benutzer"  { $level = 1 }
  "Alle"   { $level = 2 }
}
```

Erstellen Sie die beiden folgenden Variablen: *$namespace* und *$class*. *$namespace* verweist auf den WMI-Namespace von *root\cimv2\terminalservices*, in der sich die Terminalserverklassen befinden, und *$class* gibt die erforderliche WMI-Klasse an. Die beiden Variablenzuweisungen sind im Folgenden dargestellt:

```
$namespace = "root\cimv2\TerminalServices"
$class = "win32_TSPermissionsSetting"
```

Stellen Sie nun eine Verbindung zu WMI her. Verwenden Sie hierzu das Cmdlet **Get-WmiObject** und geben Sie die Werte für die Parameter **-namespace**, **-computername**, **-class** und **-filter** an. Der Parameter **-filter** gibt nur den Namen **rdp-tcp** zurück, obwohl zwei TerminalNamen vorhanden sind: Die Konsole und RDP-TCP. Für unsere Zwecke ist aber nur RDP-TCP von Interesse, da dieser Terminaltyp von den Benutzern verwendet wird. Geben Sie anschließend den Benutzernamen und die Zugriffsebene in der Methode *AddAccount()* an. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
$objClient=get-wmiobject -namespace $namespace -computername $computer `
          -class $class -filter "terminalName = 'rdp-tcp'"
$objClient.addAccount($user,$level)
```

Das vollständige Skript *GrantUserTSPermission.ps1* hat folgenden Inhalt:

GrantUserTSPermission.ps1

```
param(
      $computer = "localhost",
      $user,
      $level,
      [switch]$help
     )

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GrantUserTSPermission.ps1
```

```
Gewährt einem Benutzerkonto Zugriffsberechtigungen
auf dem lokalen Terminalserver oder einem Remote-Terminalserver.

PARAMETER:
-computer Der Computer, für den das Skript ausgeführt werden soll.
-user     Das Benutzerkonto, dem die Berechtigungen gewährt werden sollen.
-level    Die zu gewährende Zugriffsebene: < Gast, Benutzer, Alle >.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
GrantUserTSPermission.ps1
Zeigt eine Fehlermeldung an, dass eine Einstellung angegeben werden muss,
und gibt die Hilfeinformationen aus.

GrantUserTSPermission.ps1 -user bob -level Gast

Gewährt dem Benutzer Bob Gastberechtigungen auf dem lokalen Terminalserver.

GrantUserTSPermission.ps1 -user sandra -level Benutzer -computer ts1

Gewährt dem Benutzer Sandra
Benutzerberechtigungen auf dem Remote-Terminalserver TS1.

GrantUserTSPermission.ps1 -user ed -level Alle

Gewährt dem Benutzer Ed alle Berechtigungen auf dem lokalen Terminalserver.

GrantUserTSPermission.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}

if($help)       { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if(!$user)      { "Ein Benutzerkonto muss angegeben werden ..." ; funHelp }
if(!$level)     { "Die Zugriffsebene muss angegeben werden ..." ; funHelp }

switch($level)
{
  "Gast" { $level = 0 }
  "Benutzer"  { $level = 1 }
  "Alle"   { $level = 2 }
}

$namespace = "root\cimv2\TerminalServices"
$class = "win32_TSPermissionsSetting"
$objClient=get-wmiobject -namespace $namespace -computername $computer `
          -class $class  -filter  "terminalName  =  'rdp-tcp'"
$objClient.addAccount($user,$level)
```

Konfigurieren der Clienteinstellungen

Mit WMI können Sie darüber hinaus mehrere Clienteinstellungen konfigurieren. Diese Einstellungen umfassen die Farbtiefe, den aktiven Desktop und den Desktophintergrund. Das Konfigurieren dieser Einstellungen wird in diesem Abschnitt erklärt.

Mit dem Skript *ConfigureClientColor.ps1* können Sie die Einstellungen für die Farbtiefe konfigurieren. Beginnen Sie mit einer *param*-Anweisung, um mehrere Parameter zu definieren. Der erste Parameter ist **-depth**, der die Farbanzeige des Clients festlegt. Der zweite Parameter ist **-computer**, um angeben zu können, auf welchem Server das Skript ausgeführt werden soll. Der dritte Parameter hat den Namen **-list**, um die Konfigurationsinformationen abzurufen (anstatt diese zu ändern). Der vierte Parameter ist **-help**, um wieder die Hilfe anzuzeigen. Die entsprechende *param*-Anweisung lautet:

```
param(
      $depth,
      $computer = "localhost",
      [switch]$list,
      [switch]$help
     )
```

Die Funktion *funhelp()* zeigt die Hilfeinformationen an. Der Hilfetext ist in der Variablen *$helptext* definiert. Die Hilfe wird nur auf Anforderung des Benutzers oder aufgrund eines Eingabefehlers, beispielsweise aufgrund eines fehlenden Parameters, angezeigt. Die Variable *$helptext* ist der *here*-Zeichenfolge mit den Hilfeinformationen zugewiesen. Die Hilfeinformationen bestehen aus drei Abschnitten: Beschreibung, Parameter und Syntax. Nachdem Sie die Variable *$helptext* definiert haben, zeigen Sie den Inhalt der Variablen an und beenden Sie das Skript. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ConfigureClientColor.ps1
Konfiguriert die Einstellungen für die Farbtiefe auf den Clientcomputern,
die auf den lokalen oder einen Remote-Terminalserver zugreifen.

PARAMETER:
-computer  Der Computer, für den das Skript ausgeführt werden soll.
-depth     Die maximale Farbtiefe auf den Clientcomputern:
           < 8, 15, 16, 24 >
-list      Zeigt die aktuelle Konfiguration an.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
ConfigureClientColor.ps1
Zeigt eine Fehlermeldung an, dass eine Einstellung angegeben werden muss,
und gibt die Hilfeinformationen aus.

ConfigureClientColor.ps1 -depth 8

Konfiguriert die Clienteinstellungen auf dem lokalen Terminalserver,
um eine maximale Farbtiefe von 8 Bit zu unterstützen.

ConfigureClientColor.ps1 -depth 24 -computer TS2
```

```
Konfiguriert die Clienteinstellungen auf dem Remote-Terminalserver TS2,
um eine maximale Farbtiefe von 8 Bit zu unterstützen.

ConfigureClientColor.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Bearbeiten Sie nun die Funktion *funlist()*, die bei Angabe des Parameters **-list** aufgerufen wird. Verwenden Sie in der Funktion *funlist()* das Cmdlet **Get-WmiObject**, um auf den in der Variablen *$namespace* angegebenen Namespace zuzugreifen. Die Variable *$computer*, die ebenfalls in der *param*-Anweisung definiert wurde, wird an den Parameter **-computername** übergeben. Der Parameter **-class** wird über die Variable *$class* referenziert. Übergeben Sie das resultierende Objekt an das Cmdlet **Format-List** und beenden Sie das Skript. Die Funktion *funlist()* ist wie folgt implementiert:

```
Function funlist()
{
 get-wmiobject -namespace $namespace -computername $computer `
    -class $class  |
 format-list [a-z]*
 exit
}
```

Sie müssen nun einige weitere Variablen deklarieren. Erstellen Sie als Erstes die Variable *$namespace*. Weisen Sie der Variablen *$namespace* die Zeichenfolge *"root\cimv2\terminalservices"* zu. Die zweite Variable ist *$class*, die den Wert *Win32_TSClientSetting* erhalten muss. Diese beiden Variablen werden in den **Get-WmiObject**-Anweisungen verwendet. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
$namespace = "root\cimv2\TerminalServices"
$class = "win32_TSClientSetting"
```

Nachdem Sie die beiden Variablen deklariert und die Werte zugewiesen haben, müssen Sie die Befehlszeilenargumente überprüfen. Wenn Sie die Variable *$help* finden, rufen Sie die Funktion *funhelp()* auf. Ist die Variable *$list* vorhanden, rufen Sie die Funktion *funlist()* auf. Überprüfen Sie danach, ob die Variable *$depth* ausgelassen wurde. Sollten sowohl die vorherigen beiden Variablen als auch die Variable *$depth* nicht vorhanden sein, rufen Sie die Funktion *funhelp()* auf. Die drei erforderlichen Codezeilen lauten:

```
if($help) { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if($list) { funlist }
if(!$depth) { "Die Angabe der Farbtiefe ist erforderlich ..." ; funHelp }
```

Nachdem Sie die Parameter überprüft haben, können Sie mit einer *switch*-Anweisung den Wert überprüfen, der für den Parameter **-depth** angegeben wurde. Suchen Sie in der *switch*-Anweisung nach der Zahl *8*. Wurde diese Zahl angegeben, weisen Sie der Variablen *$depth* den Wert *1* zu. Möglicherweise möchte der Benutzer 8-Bit-Farben festlegen und merkt sich *8*, anstatt den codierten Wert *1*. Um die Benutzerfreundlichkeit zu verbessern, implementieren Sie den Parameter *switch*, wie in folgendem Code veranschaulicht:

```
 switch($depth)
 {
  8  { $depth = 1 }
  15 { $depth = 2 }
```

```
  16 { $depth = 3 }
  24 { $depth = 4 }
 }
```

Nachdem Sie die Variable *$depth* überprüft haben, sind Sie soweit, dass Sie die Einstellung für die Farbtiefe ändern können. Stellen Sie hierzu mit dem Cmdlet **Get-WmiObject** eine Verbindung mit dem Namespace *root\cimv2\terminalservices* auf dem lokalen Computer oder auf dem in der Variablen *$computer* angegebenen Computer her. Rufen Sie eine Instanz der Klasse *Win32_TSClientSetting* ab. Speichern Sie das resultierende Objekt in der Variablen *$objclient*, rufen Sie die Methode *SetColorDepth()* auf und legen Sie diese auf den Wert in der Variablen *$depth* fest. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
$objClient=get-wmiobject -namespace $namespace -computername $computer `
          -class $class
$objClient.SetColorDepth($depth)
```

Das vollständige Skript *ConfigureClientColor.ps1* hat folgenden Inhalt:

ConfigureClientColor.ps1

```
param(
      $depth,
      $computer = "localhost",
      [switch]$list,
      [switch]$help
     )

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ConfigureClientColor.ps1
Konfiguriert die Einstellungen für die Farbtiefe auf den Clientcomputern,
die auf den lokalen oder einen Remote-Terminalserver zugreifen.

PARAMETER:
-computer  Der Computer, für den das Skript ausgeführt werden soll.
-depth     Die maximale Farbtiefe auf den Clientcomputern:
           < 8, 15, 16, 24 >
-list      Zeigt die aktuelle Konfiguration an.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
ConfigureClientColor.ps1
Zeigt eine Fehlermeldung an, dass eine Einstellung angegeben werden muss,
und gibt die Hilfeinformationen aus.

ConfigureClientColor.ps1 -depth 8

Konfiguriert die Clienteinstellungen auf dem lokalen Terminalserver,
um eine maximale Farbtiefe von 8 Bit zu unterstützen.

ConfigureClientColor.ps1 -depth 24 -computer TS2

Konfiguriert die Clienteinstellungen auf dem Remote-Terminalserver TS2,
um eine maximale Farbtiefe von 8 Bit zu unterstützen.
```

```
ConfigureClientColor.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}

Function funlist()
{
 get-wmiobject -namespace $namespace -computername $computer `
    -class $class  |
 format-list [a-z]*
 exit
}

$namespace = "root\cimv2\TerminalServices"
$class = "win32_TSClientSetting"

if($help) { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if($list) { funlist }
if(!$depth) { "Die Angabe der Farbtiefe ist erforderlich ..." ; funHelp }
 switch($depth)
 {
  8  { $depth = 1 }
  15 { $depth = 2 }
  16 { $depth = 3 }
  24 { $depth = 4 }
 }


$objClient=get-wmiobject -namespace $namespace -computername $computer `
          -class $class
$objClient.SetColorDepth($depth)
```

Möglicherweise müssen Sie auf dem Terminaldiensteclient die Hintergrundeinstellungen konfigurieren. Verwenden Sie hierzu das Skript *ConfigureClientEnvironment.ps1*.

Das Skript *ConfigureClientEnvironment.ps1* beginnt mit einer *param*-Anweisung. Der erste Parameter, **-action**, gibt an, dass Sie den Hintergrund ändern möchten. Geben Sie den Wert für die Hintergrundeinstellung mit dem Parameter **-value** an. Mit dem Parameter **-computer** definieren Sie wiederum den Namen des Servers, auf dem Sie die Änderungen vornehmen möchten. Der erste feststehende Parameter ist **-list**. Dieser Parameter ruft die aktuellen Einstellungen ab. Der zweite feststehende Parameter ist **-help**, um die Hilfe anzuzeigen. Die *param*-Anweisung lautet:

```
param(
      $action,
      $value,
      $computer = "localhost",
      [switch]$list,
      [switch]$help
     )
```

Die Funktion *funhelp()* zeigt die Hilfeinformationen an. Weisen Sie die sich ergebende *here*-Zeichenfolge der Variablen *$helptext* zu, geben Sie den Inhalt der Variablen *$helptext* aus und beenden Sie das Skript. Diese Funktion ist im Folgenden dargestellt:

```
function funHelp()
{
$helpText=@"
DESCRIPTION:
NAME: ConfigureClientEnvironment.ps1
Konfiguriert die Terminalserver-Einstellungen für die Arbeitsumgebung auf den Clientcomputern,
die auf den lokalen oder einen Remote-Terminalserver zugreifen.

PARAMETER:
-action    Die auszuführende Aktion < wp (Hintergrund) >
-value     Der für die Aktion angegebene Wert.
-computer  Der Computer, für den das Skript ausgeführt werden soll.
-list      Listet die Terminalserver-Einstellungen für die Arbeitsumgebung auf.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
ConfigureClientEnvironment.ps1
Zeigt eine Fehlermeldung an, dass eine Einstellung angegeben werden muss,
und gibt die Hilfeinformationen aus.

ConfigureClientEnvironment.ps1 -list

Listet die Terminalserver-Einstellungen für die Arbeitsumgebung der Clients
auf dem lokalen  Terminalserver auf.

ConfigureClientEnvironment.ps1 -action wp -value 1

Konfiguriert die Terminalserver-Einstellungen für die Arbeitsumgebung der Clients
auf dem lokalen  Terminalserver, um den Bildschirmhintergrund zu unterdrücken.

ConfigureClientEnvironment.ps1 -action wp -value 0

Konfiguriert den lokalen  Terminalserver, um den Bildschirmhintergrund
auf den Terminaldiensteclients anzuzeigen.

ConfigureClientEnvironment.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Die Funktion *funlist()* zeigt die aktuelle Konfiguration an. Rufen Sie mit dem Cmdlet **Get-WmiObject** die WMI-Klasse *Win32_TSClientSetting* auf dem in der Variablen *$computer* angegebenen Computer ab. Formatieren Sie die Ausgabe als Liste und beenden Sie das Skript. Die Funktion *funlist()* ist wie folgt implementiert:

```
Function funlist()
{
 get-wmiobject -namespace $namespace -computername $computer `
    -class $class |
 format-list [a-z]*
 exit
}
```

Erstellen Sie die Funktion *funpaper()*, um die Hintergrundeinstellungen für die Clients festzulegen. Verwenden Sie hierzu die Methode *SetClientWallpaper()*. Greifen Sie mit dem Cmdlet **Get-WmiObject** auf die Methode zu. Der Unterschied zwischen der **Get-WmiObject**-Anweisung in diesem Beispiel und in der Funktion *funlist()* besteht darin, dass die Ergebnisse auf den Terminaltyp *RDP-TCP* beschränkt sind. Auf diese Art vermeiden Sie, dass Sie mit der Terminalserverkonsole arbeiten. Die Funktion *funpaper()* ist wie folgt implementiert:

```
Function funpaper($strin)
{
 $objClient=get-wmiobject -namespace $namespace -computername $computer `
          -class $class -filter "terminalname = 'rdp-tcp'"
 $objClient.SetClientWallPaper($strin)
 exit
}
```

Der nächste Schritt betrifft das Erstellen der beiden Variablen, die in den **Get-WmiObject**-Anweisungen verwendet werden. Die erste Variable gibt den WMI-Namespace an und die zweite Variable bestimmt die WMI-Klasse. Die beiden Codezeilen lauten:

```
$namespace = "root\cimv2\TerminalServices"
$class = "win32_TSEnvironmentSetting"
```

Überprüfen Sie nun die Befehlszeile. Wenn der Parameter **-help** vorhanden ist, rufen Sie die Funktion *funhelp()* auf. Existiert der Parameter **-list**, führen Sie wiederum die Funktion *funlist()* aus. Überprüfen Sie danach, ob die Variable *$action* vorhanden ist. Sollten Sie die Variable *$action* nicht finden, rufen Sie die Funktion *funhelp()* auf. Codebeispiel:

```
if($help) { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if($list) { funlist }
if(!$action -and !$list) { "Sie müssen eine Aktion angeben ..." ; funhelp }
```

Der letzte Abschnitt des Skripts umfasst eine *switch*-Anweisung, mit der Sie den Wert in der Variablen *$action* überprüfen. Wenn der Wert *"wp"* entspricht, rufen Sie die Funktion *funpaper()* auf. Codezeile:

```
switch($action)
{
  "wp" { funPaper($value) }
}
```

Das vollständige Skript *ConfigureClientEnvironment.ps1* hat folgenden Inhalt:

ConfigureClientEnvironment.ps1

```
param(
      $action,
      $value,
      $computer = "localhost",
      [switch]$list,
      [switch]$help
     )

function funHelp()
```

```
{
$helpText=@"
DESCRIPTION:
NAME: ConfigureClientEnvironment.ps1
Konfiguriert die Terminalserver-Einstellungen für die Arbeitsumgebung auf den Clientcomputern,
die auf den lokalen oder einen Remote-Terminalserver zugreifen.

PARAMETER:
-action    Die auszuführende Aktion < wp (Hintergrund) >
-value     Der für die Aktion angegebene Wert.
-computer  Der Computer, für den das Skript ausgeführt werden soll.
-list      Listet die Terminalserver-Einstellungen für die Arbeitsumgebung auf.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
ConfigureClientEnvironment.ps1
Zeigt eine Fehlermeldung an, dass eine Einstellung angegeben werden muss,
und gibt die Hilfeinformationen aus.

ConfigureClientEnvironment.ps1 -list

Listet die Terminalserver-Einstellungen für die Arbeitsumgebung der Clients
auf dem lokalen  Terminalserver auf.

ConfigureClientEnvironment.ps1 -action wp -value 1

Konfiguriert die Terminalserver-Einstellungen für die Arbeitsumgebung der Clients
auf dem lokalen  Terminalserver, um den Bildschirmhintergrund zu unterdrücken.

ConfigureClientEnvironment.ps1 -action wp -value 0

Konfiguriert den lokalen  Terminalserver, um den Bildschirmhintergrund
auf den Terminaldiensteclients anzuzeigen.

ConfigureClientEnvironment.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}

Function funlist()
{
 get-wmiobject -namespace $namespace -computername $computer `
    -class $class |
 format-list [a-z]*
 exit
}

Function funpaper($strin)
{
 $objClient=get-wmiobject -namespace $namespace -computername $computer `
          -class $class  -filter  "terminalname  =  'rdp-tcp'"
 $objClient.SetClientWallPaper($strin)
```

```
 exit
}

$namespace = "root\cimv2\TerminalServices"
$class = "win32_TSEnvironmentSetting"

if($help) { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if($list) { funlist }
if(!$action -and !$list) { "Sie müssen eine Aktion angeben ..." ; funhelp }


switch($action)
{
  "wp" { funPaper($value) }
}
```

Der letzte Vorgang, der in diesem Kapitel beschrieben werden soll, betrifft das Deaktivieren der aktiven Desktopfeatures für die Terminaldienstеclients. Sie führen diesen Vorgang mit dem Skript *DisableActiveDesktop.ps1* aus.

Beginnen Sie das Skript *DisableActiveDesktop.ps1* mit einer *param*-Anweisung, mit der Sie mehrere Parameter festlegen. Der erste Parameter lautet **-computer**, um den Computer anzugeben, auf dem der WMI-Befehl ausgeführt werden soll. Die übrigen Parameter sind feststehende Parameter. Die Parameter **-allow** und **-disallow** aktivieren bzw. deaktivieren die aktiven Desktopfeatures. Der Parameter **-list** zeigt die aktuelle Konfiguration an. Der letzte Parameter ist **-help**, um die Hilfeinformationen auszugeben. Die *param*-Anweisung lautet:

```
param(
      $computer = "localhost",
      [switch]$allow,
      [switch]$disallow,
      [switch]$list,
      [switch]$help
     )
```

Der nächste Schritt betrifft das Erstellen der Funktion *funhelp()*, um die Hilfeinformationen anzuzeigen. Verwenden Sie eine *here*-Zeichenfolge, um den Textwert für die Variable *$helptext* zu definieren. Die Hilfeinformationen umfassen eine Beschreibung, Parameter und Syntaxbeispiele zum Ausführen des Skripts. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: DisableActiveDesktop.ps1
Konfiguriert die Sitzungseinstellungen für die Clientcomputer,
die auf den lokalen oder einen Remote-Terminalserver zugreifen.

PARAMETER:
-computer Der Computer, für den das Skript ausgeführt werden soll.
-disallow Deaktiviert die aktiven Desktopfeatures in der aktuellen Sitzung.
-allow    Aktiviert die aktiven Desktopfeatures in der aktuellen Sitzung.
-list     Zeigt die aktuelle Konfiguration an.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
DisableActiveDesktop.ps1
```

```
Zeigt eine Fehlermeldung an, dass eine Einstellung angegeben werden muss,
und gibt die Hilfeinformationen aus.

DisableActiveDesktop.ps1 -list

Listet die Einstellungen für die aktiven Desktopfeatures der Clientsitzungen
auf dem lokalen  Terminalserver auf.

DisableActiveDesktop.ps1 -allow -computer TS2

Listet die Einstellungen für die aktiven Desktopfeatures der Clientsitzungen
auf dem Terminalserver TS2  auf.

DisableActiveDesktop.ps1 -disallow

Konfiguriert den Client, um die aktiven Desktopfeatures
auf dem lokalen Terminalserver zu deaktivieren.

DisableActiveDesktop.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Die Funktion *funlist()* ruft die aktuellen Konfigurationsinformationen ab und zeigt diese als formatierte Liste an. Verwenden Sie das Cmdlet **Get-WmiObject** und legen Sie für den Parameter **-namespace** den Wert der Variablen *$namespace* fest. Geben Sie den Computernamen in der Variablen *$computer* und die Klasse in der Variablen *$class* an. Formatieren Sie die Ausgabe als Liste und beenden Sie das Skript. Die Funktion *funlist()* ist wie folgt implementiert:

```
Function funlist()
{
 get-wmiobject -namespace $namespace -computername $computer `
    -class $class  |
 format-list [a-z]*
 exit
}
```

Sie müssen nun noch den Variablen *$namespace* und *$class* Werte zuweisen. Hierbei handelt es sich um einfache Zeichenfolgenzuordnungen:

```
$namespace = "root\cimv2\TerminalServices"
$class = "win32_TerminalServiceSetting"
```

Vergessen Sie nicht, die Befehlszeilenparameter auszuwerten. Da ein angegebener Befehlszeilenparameter die entsprechende Variable im Arbeitsspeicher erstellt, können Sie die jeweiligen Parameter untersuchen. Wenn beispielsweise die Variable *$help* gefunden wird, wurde das Skript mit dem Parameter **-help** ausgeführt. Rufen Sie dementsprechend die Funktion *funhelp()* auf. Wenn die Variable *$list* vorhanden ist, rufen Sie die Funktion *funlist()* auf. Existiert die Variable *$allow*, weisen Sie der Variablen *$action* den Wert *1* zu. Ist hingegen die Variable *$disallow* vorhanden, weisen Sie der Variablen *$action* den Wert *0* zu. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
if($help)     {     "Ausgeben der Hilfeinformationen ..."     ;     funHelp     }
if($list)     {     funlist     }
if($allow)    { $action = 1 }
if($disallow) { $action = 0 }
```

Sie müssen nun die angegebene Aktion für die *ActiveDesktop*-Eigenschaft ausführen. Verwenden Sie hierzu das Cmdlet **Get-WmiObject** und geben Sie die Werte *$class, $namespace,* und *$computer* für die entsprechenden Parameter an. Weisen Sie das resultierende WMI-Verwaltungsobjekt der Variablen *$objTS* zu. Setzen Sie danach die *ActiveDesktop*-Eigenschaft des Verwaltungsobjekts auf den Wert der Variablen *$action* und rufen Sie die *Put()*-Methode auf, um die Änderung zu übernehmen. Dieser Codeabschnitt ist im Folgenden dargestellt:

```
$objTS = get-wmiobject -class $class -namespace $namespace `
       -computername $computer
$objTS.ActiveDesktop = $action
$objTS.put()
```

Das vollständige Skript *DisableActiveDesktop.ps1* hat folgenden Inhalt:

DisableActiveDesktop.ps1

```
param(
      $computer = "localhost",
      [switch]$allow,
      [switch]$disallow,
      [switch]$list,
      [switch]$help
     )

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: DisableActiveDesktop.ps1
Konfiguriert die Sitzungseinstellungen für die Clientcomputer,
die auf den lokalen oder einen Remote-Terminalserver zugreifen.

PARAMETER:
-computer Der Computer, für den das Skript ausgeführt werden soll.
-disallow Deaktiviert die aktiven Desktopfeatures in der aktuellen Sitzung.
-allow    Aktiviert die aktiven Desktopfeatures in der aktuellen Sitzung.
-list     Zeigt die aktuelle Konfiguration an.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
DisableActiveDesktop.ps1
Zeigt eine Fehlermeldung an, dass eine Einstellung angegeben werden muss,
und gibt die Hilfeinformationen aus.

DisableActiveDesktop.ps1 -list

Listet die Einstellungen für die aktiven Desktopfeatures der Clientsitzungen
auf dem lokalen  Terminalserver auf.

DisableActiveDesktop.ps1 -allow -computer TS2

Listet die Einstellungen für die aktiven Desktopfeatures der Clientsitzungen
```

```
auf dem Terminalserver TS2  auf.

DisableActiveDesktop.ps1 -disallow

Konfiguriert den Client, um  die aktiven Desktopfeatures
auf dem lokalen Terminalserver zu deaktivieren.

DisableActiveDesktop.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}

Function funlist()
{
 get-wmiobject -namespace $namespace -computername $computer `
    -class $class  |
 format-list [a-z]*
 exit
}


$namespace = "root\cimv2\TerminalServices"
$class = "win32_TerminalServiceSetting"

if($help)     { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if($list)     {     funlist     }
if($allow)    { $action = 1 }
if($disallow) { $action = 0 }


$objTS = get-wmiobject -class $class -namespace $namespace `
       -computername $computer
$objTS.ActiveDesktop = $action
$objTS.put()
```

Zusammenfassung

In diesem Kapitel wurden die verschiedenen Einstellungen zum Konfigurieren von Windows Server 2008-Terminalservern beschrieben. Außerdem wurde das Konfigurieren der Sitzungseinstellungen zum Skalieren eines Windows Server 2008-Terminalservers für mehrere Benutzer sowie das Deaktivieren von Anmeldungen für Wartungszwecke erklärt.

Sie haben erfahren, wie Sie sowohl den Benutzerzugriff auf einen Terminalserver gewähren und die Remoteeinstellungen konfigurieren als auch die Farbtiefe und Hintergrundeinstellungen festlegen können. Außerdem wurde das Deaktivieren des aktiven Desktops zum Einsparen von Netzwerkbandbreite erklärt.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

KAPITEL 18

Konfigurieren der Netzwerkdienste

Nach Abschluss dieses Kapitels können Sie:

- Die DNS-Einstellungen überprüfen
- Die DNS-Protokolleinstellungen konfigurieren
- DNS-Stammhinweise überprüfen
- DNS-Zonen überprüfen
- DNS-Zonen erstellen
- WINS und DHCP verwalten

Auf der Begleit-CD Alle Skripts in diesem Kapitel befinden sich auf der Begleit-CD im Ordner *\Scripts\Chapter18*.

Überprüfen der DNS-Einstellungen

Die DNS-Konfiguration (Domain Name System) umfasst zahlreiche Einstellungen, die regelmäßig überprüft werden sollten, um sicherzustellen, dass die Einstellungen nicht geändert wurden oder aktualisiert werden müssen. Das Überprüfen der Einstellungen im DNS-Manager ist aufgrund der vielen Registerkarten zeitaufwändig. Außerdem kann eine selten verwendete Registerkarte schnell übersehen werden. In Abbildung 18.1 sind diese Registerkarten dargestellt.

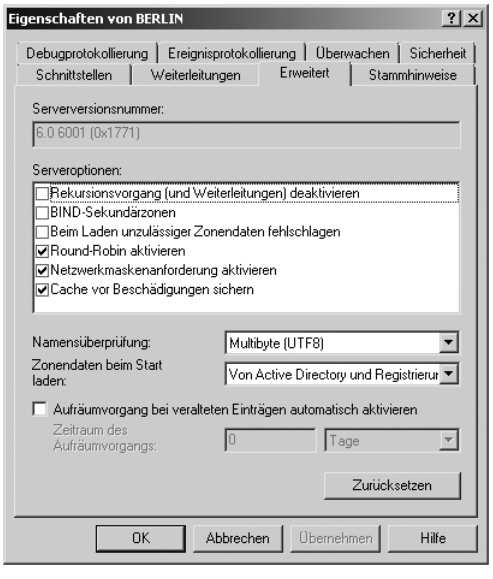


Abbildung 18.1 Der DNS-Manager ermöglicht den Zugriff auf zahlreiche Eigenschaften

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Für diese Aufgabe kommt Ihnen Windows PowerShell zur Hilfe. Beispielsweise können Sie mit dem Skript *GetDNSServerConfig.ps1* sicherstellen, dass alle für einen DNS-Server erforderlichen Konfigurationseinstellungen vorgenommen wurden.

Beginnen Sie das Skript *GetDNSServerConfig.ps1* mit einer *param()*-Anweisung zur Definition von drei Befehlszeilenparametern. Der erste Parameter lautet **-computer** und verwendet den Standardwert *localhost*, der auf den lokalen Computer verweist. Der zweite Parameter ist **-query**. Dieser Parameter ist erforderlich. Wenn der Variablen *$query* kein Wert zugewiesen ist, tritt ein Fehler auf. Der dritte Parameter hat einen festen Wert und führt keine Aktion aus, wenn dieser nicht angegeben wird. Dieser Parameter hat einen booleschen Wert und akzeptiert keine Argumente. Der in diesem Beispiel verwendete Parameter ist **-help**, um die Hilfe anzuzeigen. Codezeile:

```
param($computer="localhost",$query,[switch]$help)
```

Bewährte Vorgehensweise Wenn Sie ein Skript erstellen, das zahlreiche Parameter umfasst, sollten Sie eine Hilfefunktion mit Syntaxbeispielen erstellen.

Definieren Sie die Funktion *funhelp()*, um die Hilfe anzuzeigen. Die Definition einer Funktion besteht aus den drei Abschnitten, die in Tabelle 18.1 aufgeführt sind.

Tabelle 18.1 Die drei Abschnitte einer Funktionsdeklaration

Schlüsselwort	Eingabeparameter	Codeblock
Funktion	()	{ }

Nachdem Sie die Funktion definiert haben, definieren Sie für die Variable *$helptext* eine *here*-Zeichenfolge. Die *here*-Zeichenfolge beginnt mit @" und endet mit "@.

Tipp Der größte Vorteil von *here*-Zeichenfolgen ist, dass Sie die Regeln für Anführungszeichen ignorieren können. Wenn Sie beispielsweise einen Satz eingeben, der Anführungszeichen enthält, müssen Sie nach dem Anführungszeichen ein *Escape*-Zeichen eingeben, damit das Skript das Anführungszeichen nicht als Ende der Zeichenfolge erkennt. Die Eingabe doppelter oder dreifacher Anführungszeichen führt häufig zu Fehlern. Eine *here*-Zeichenfolge vermeidet diese durch Anführungszeichen verursachten Fehler.

Da die *here*-Zeichenfolge keinen ausführbaren Code darstellt, muss diese in einer Variablen gespeichert werden. Speichern Sie die *here*-Zeichenfolge in der Variablen *$helptext*. Nachdem Sie die *here*-Zeichenfolge in der Variablen *$helptext* erstellt haben, zeigen Sie den Inhalt der Variablen an und beenden Sie das Skript. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GetDNSServerConfig.ps1
Erstellt ein Listing der DNS Server-Konfiguration auf dem
lokalen Computer oder einem Remotecomputer.

PARAMETER:
-computer Gibt den Computer an, für den das Skript ausgeführt werden soll.
-query    Gibt die Art der Abfrage an  < all, advanced, cache, forward,
          interval, log, recurse >.
-help     Zeigt dieses Hilfethema an.
```

```
SYNTAX:
GetDNSServerConfig.ps1

Erstellt ein Listing der DNS Server-Standardkonfiguration auf dem lokalen Computer.

GetDNSServerConfig.ps1 -computer MunichServer -query advanced

Zeigt die Konfigurationsparameter RoundRobin, SecureResponses, EnableDnsSec und
BindSecondaries  des Computers MunichServer an.

GetDNSServerConfig.ps1 -computer MunichServer -query cache

Zeigt die Konfigurationsparameter AutoCacheUpdate, EDnsCacheTimeout, MaxCacheTTL und
MaxNegativeCacheTTL des Computers MunichServer an.

GetDNSServerConfig.ps1 -computer MunichServer -query forward

Zeigt die Konfigurationsparameter ForwardDelegations, Forwarders und
ForwardingTimeout des Computers MunichServer an.

GetDNSServerConfig.ps1 -computer MunichServer -query interval

Zeigt die Konfigurationsparameter DefaultNoRefreshInterval, DefaultRefreshInterval,
DisjointNets, DsPollingInterval, DsTombstoneInterval und
ScavengingInterval des Computers MunichServer an.

GetDNSServerConfig.ps1 -computer MunichServer -query log

Zeigt die Konfigurationsparameter EventLogLevel, LogFileMaxSize, LogFilePath,
LogIPFilterList und LogLevel des Computers MunichServer an.

GetDNSServerConfig.ps1 -computer MunichServer -query recurse

Zeigt die Konfigurationsparameter NoRecursion, RecursionRetry und
RecursionTimeout des Computers MunichServer an.

GetDNSServerConfig.ps1 -computer MunichServer -query ALL

Zeigt alle DNS Server-Konfigurationsinformationen
des Computers MunichServer an.

GetDNSServerConfig.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Überprüfen Sie mit einer *if*-Anweisung, ob die Variable *$help* vorhanden ist. Die Variable *$help* ist nur vorhanden, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Geben Sie in diesem Fall eine Meldung aus und rufen Sie die Funktion *funhelp()* auf. Da Sie beim Aufrufen dieser Funktion keinen Parameter übergeben müssen, werden die Klammern ausgelassen:

```
if($help) { "Ausgeben der Hilfeinformationen ..." ; funHelp }
```

Geben Sie nun die zu verwendende WMI-Klasse an. Um einen DNS-Server zu verwalten, verwenden Sie die WMI-Klasse *MicrosoftDNS_Server*. Weisen Sie den Namen der Klasse der Variablen *$class* zu:

```
$class="MicrosoftDNS_Server"
```

Definieren Sie anschließend mehrere Variablen für die Listen der anzuzeigenden Eigenschaften. Sie können beispielsweise Eigenschaften auswählen, die sich auf die Protokollierung, die Weiterleitung, die Rekursion, das Zwischenspeichern oder Aktualisierungsintervalle beziehen, um die WMI-Klasse *MicrosoftDNS_Server* koordiniert einzusetzen. Codebeispiel:

```
$logProperty = "EventLogLevel","LogFileMaxSize","LogFilePath", `
               "LogIPFilterList","LogLevel"
$forwardProperty = "ForwardDelegations", "Forwarders", `
                   "ForwardingTimeout"
$recurseProperty = "NoRecursion","RecursionRetry","RecursionTimeout"
$cacheProperty = "AutoCacheUpdate","EDnsCacheTimeout","MaxCacheTTL", `
                 "MaxNegativeCacheTTL"
$intervalProperty = "DefaultNoRefreshInterval", `
                    "DefaultRefreshInterval", "DisjointNets", `
                   "DsPollingInterval", "DsTombstoneInterval", `
                   "ScavengingInterval"
$advroperty = "roundrobin","SecureResponses","EnableDnsSec", `
              "BindSecondaries"
```

An dieser Stelle können Sie die Abfrage ausführen und auswerten. Stellen Sie zuerst sicher, dass der Parameter **-query** angegeben wurde. Ist dies der Fall, müssen Sie den Wert der Variablen *$query* überprüfen. Hat die Variable *$query* den Wert *log*, wählen Sie die in der Variablen *$logproperty* gespeicherte Eigenschaftsliste aus. Ist der Benutzer hingegen an Weiterleitungen interessiert, wählen Sie die Eigenschaftsliste aus, die in der Variablen *$forwardproperty* gespeichert ist. Werten Sie mit einer *switch*-Anweisung den Wert der Variablen *$query* aus und geben Sie die entsprechenden Eigenschaftsnamen an. Die *switch*-Anweisung umfasst einen Standardblock, der verwendet wird, wenn der Parameter **-query** einen unbekannten Wert bezeichnet. Fragen Sie in diesem Fall alle Elemente ab, um die Werte aller Eigenschaften der WMI-Klasse zurückzugeben. Die *switch*-Anweisung und der erforderliche Code sind wie folgt implementiert:

```
if($query)
{
 switch($query)
 {
  "log"       { $query=$logProperty }
  "forward"   { $query=$forwardProperty }
  "recurse"   { $query= $recurseProperty  }
  "cache"     { $query=$cacheProperty  }
  "interval"  { $query=$intervalProperty }
  "advanced"  { $query=$advroperty }
  "all"   {
            Get-WmiObject -class $class -computername $computer `
            -namespace root\microsoftDNS| format-list * ;
             exit
        }
  DEFAULT {  "
             Verwenden der Standardeinstellung: all. Für weitere Optionen führen Sie folgenden Befehl aus:
             GetDNSServerConfig.ps1 -help
             "
```

```
            Get-WmiObject -class $class -computername $computer `
            -namespace root\microsoftDNS| format-list * ;
            exit
          }
 }
}
```

Geben Sie als Nächstes eine *else*-Klausel an. Wenn das Skript ohne den Parameter **-query** ausgeführt wurde, müssen Sie alle Elemente abfragen. Codeabschnitt:

```
ELSE
 {
  "
  Verwenden der Standardeinstellung: all. Für weitere Optionen führen Sie folgenden Befehl aus:
  GetDNSServerConfig.ps1 -help
  "
  Get-WmiObject -class $class -computername $computer `
  -namespace root\microsoftDNS| format-list * ;
  exit
 }
```

Führen Sie nun die WMI-Abfrage basierend auf dem Wert aus, der zur Laufzeit mit dem Parameter **-query** angegeben wurde. Verwenden Sie hierzu das Cmdlet **Get-WmiObject** mit dem Parameter **-class**. Geben Sie im Parameter **-class** den Namen der WMI-Klasse an, der in der Variablen *$class* gespeichert ist. Stellen Sie eine Verbindung mit dem WMI-Dienst her, der auf dem im Parameter **-computer** angegebenen Computer ausgeführt wird, und ändern Sie den Namespace in *root\microsoftDNS*. Nachdem alle Instanzen der WMI-Klasse *MicrosoftDNS_Server* abgerufen wurden, übergeben Sie das zurückgegebene Objekt an das Cmdlet **Format-List**. Zeigen Sie ausschließlich die Werte der Eigenschaften an, die über den Parameter **-query** ausgewählt wurden. Die WMI-Abfrageanweisung lautet:

```
Get-WmiObject -class $class -computername $computer `
-namespace root\microsoftDNS |
format-list -property $query
```

Das vollständige Skript *GetDNSServerConfig.ps1* hat folgenden Inhalt:

GetDNSServerConfig.ps1

```
param($computer="localhost",$query,[switch]$help)
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: GetDNSServerConfig.ps1
Erstellt ein Listing der DNS Server-Konfiguration auf dem
lokalen Computer oder einem Remotecomputer.

PARAMETER:
-computer Gibt den Computer an, für den das Skript ausgeführt werden soll.
-query    Gibt die Art der Abfrage an  < all, advanced, cache, forward,
          interval, log, recurse >.
-help     Zeigt dieses Hilfethema an.
SYNTAX:
GetDNSServerConfig.ps1

Erstellt ein Listing der DNS Server-Standardkonfiguration auf dem lokalen Computer.

GetDNSServerConfig.ps1 -computer MunichServer -query advanced
```

```
Zeigt die Konfigurationsparameter RoundRobin, SecureResponses, EnableDnsSec und
BindSecondaries  des Computers MunichServer an.

GetDNSServerConfig.ps1 -computer MunichServer -query cache

Zeigt die Konfigurationsparameter AutoCacheUpdate, EDnsCacheTimeout, MaxCacheTTL und
MaxNegativeCacheTTL des Computers MunichServer an.

GetDNSServerConfig.ps1 -computer MunichServer -query forward

Zeigt die Konfigurationsparameter ForwardDelegations, Forwarders und
ForwardingTimeout des Computers MunichServer an.

GetDNSServerConfig.ps1 -computer MunichServer -query interval

Zeigt die Konfigurationsparameter DefaultNoRefreshInterval, DefaultRefreshInterval,
DisjointNets, DsPollingInterval, DsTombstoneInterval und
ScavengingInterval des Computers MunichServer an.

GetDNSServerConfig.ps1 -computer MunichServer -query log

Zeigt die Konfigurationsparameter EventLogLevel, LogFileMaxSize, LogFilePath,
LogIPFilterList und LogLevel des Computers MunichServer an.

GetDNSServerConfig.ps1 -computer MunichServer -query recurse

Zeigt die Konfigurationsparameter NoRecursion, RecursionRetry und
RecursionTimeout des Computers MunichServer an.

GetDNSServerConfig.ps1 -computer MunichServer -query ALL

Zeigt alle DNS Server-Konfigurationsinformationen
des Computers MunichServer an.

GetDNSServerConfig.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help) { "Ausgeben der Hilfeinformationen ..." ; funHelp }

$class="MicrosoftDNS_Server"
$logProperty = "EventLogLevel","LogFileMaxSize","LogFilePath", `
               "LogIPFilterList","LogLevel"
$forwardProperty = "ForwardDelegations", "Forwarders", `
                   "ForwardingTimeout"
$recurseProperty = "NoRecursion","RecursionRetry","RecursionTimeout"
$cacheProperty = "AutoCacheUpdate","EDnsCacheTimeout","MaxCacheTTL", `
                 "MaxNegativeCacheTTL"
$intervalProperty = "DefaultNoRefreshInterval", `
                    "DefaultRefreshInterval", "DisjointNets", `
                   "DsPollingInterval", "DsTombstoneInterval", `
```

```
                  "ScavengingInterval"
$advroperty = "roundrobin","SecureResponses","EnableDnsSec", `
              "BindSecondaries"
if($query)
{
 switch($query)
 {
  "log"       { $query=$logProperty }
  "forward"   { $query=$forwardProperty }
  "recurse"   { $query= $recurseProperty  }
  "cache"     { $query=$cacheProperty  }
  "interval"  { $query=$intervalProperty }
  "advanced"  { $query=$advroperty }
  "all"   {
            Get-WmiObject -class $class -computername $computer `
            -namespace root\microsoftDNS| format-list * ;
            exit
          }

  DEFAULT {  "
             Verwenden der Standardeinstellung: all. Für weitere Optionen führen Sie folgenden Befehl aus:
             GetDNSServerConfig.ps1 -help
                "
           Get-WmiObject -class $class -computername $computer `
           -namespace root\microsoftDNS| format-list * ;
           exit
         }
 }
}
ELSE
 {
  "
  Verwenden der Standardeinstellung: all. Für weitere Optionen führen Sie folgenden Befehl aus:
  GetDNSServerConfig.ps1 -help
  "
  Get-WmiObject -class $class -computername $computer `
  -namespace root\microsoftDNS| format-list * ;
  exit
 }

  Get-WmiObject -class $class -computername $computer `
  -namespace root\microsoftDNS |
  format-list -property $query
```

Konfigurieren der DNS-Protokolleinstellungen

Der DNS-Dienst verwendet das Systemereignisprotokoll und die Diagnoseprotokollierung, die standardmäßig Ereignisse in der Datei %Systemroot%\System32\dns\dns.log aufzeichnet. Sie können den Pfad ändern; allerdings nicht in einen UNC-Pfad. Um das DNS-Diagnoseprotokoll in einem UNC-Pfad zu speichern, verwenden Sie das Skript *ConfigureDNSLogging.ps1*. Dieses Skript beendet den DNS-Dienst, kopiert das Protokoll in die Netzwerkfreigabe und startet den Dienst neu. Die DNS-Protokolloptionen sind in Abbildung 8.2 dargestellt.

Beginnen Sie das Skript *ConfigureDNSLogging.ps1* mit einer *param*-Anweisung. Da für dieses Skript viele Parameter angegeben werden, geben Sie den Befehl in zwei Zeilen ein. Die *param*-Anweisung ermöglicht die Eingabe von Befehlszeilenparametern.

Tipp Wenn eine Anweisung als unvollständig erkannt wird, sucht Windows PowerShell in der nächsten Zeile nach den fehlenden Elementen der Anweisung. Nutzen Sie dieses Feature für die *param*-Anweisung, um den Code zu formatieren. Da die erste Zeile mit einem Komma endet, betrachtet die Windows PowerShell die Zeile als unvollständig und setzt die Anweisung in der nächsten Zeile fort. Der Zeilenumbruch erhöht die Lesbarkeit des Codes.

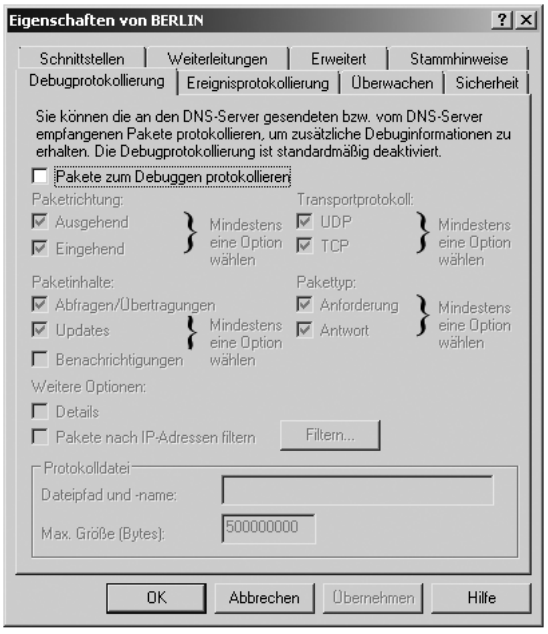


Abbildung 18.2 Die DNS-Protokollierung umfasst Ereignisprotokolle und Diagnoseprotokolle

Legen Sie den Parameter **-computer** auf den Standardwert *localhost* fest. Der nächste Parameter ist **-change**, um das Ändern eines Werts zu ermöglichen. Der dritte Parameter ist **-query**. Wenn die Variable *$query* vorhanden ist, führt das Skript eine Standardabfrage aus und gibt die Standardprotokolleinstellungen zurück. Der Parameter **-restart** bewirkt den Neustart des DNS-Dienstes. Sie können die Zeitdauer zwischen dem Beenden und Starten des Dienstes festlegen. Die übrigen Parameter **-stop**, **-start** und **-help** haben einen festen Wert. Codeabschnitt:

```
param(
      $computer="localhost", $change, [switch]$query, $restart,
      [switch]$stop, [switch]$start, [switch]$help
     )
```

Die nächste Funktion ist *funhelp()*, um die Hilfe anzuzeigen. Die Funktion *funhelp()* verwendet eine *here*-Zeichenfolge für den Hilfetext. Die *here*-Zeichenfolge umfasst jeweils einen Abschnitt für die Beschreibung, Parameter und Syntax. Diese Informationen werden als Text eingegeben und der Variablen *$helptext* zugewiesen. Nachdem Sie den Wert der Variablen *$helptext* festgelegt haben, zeigen Sie den Inhalt der Variablen an und beenden Sie das Skript. Codeabschnitt:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ConfigureDNSLogging.ps1
Konfiguriert die Diagnoseprotokollfunktionen eines DNS-Servers
auf dem lokalen Computer oder auf einem Remotecomputer.

PARAMETER:
-computer Gibt den Computer an, für den das Skript ausgeführt werden soll.
-change   Gibt die zu konfigurierenden Eigenschaften des DNS-Servers an: < LogLevel,
          LogPath, LogSize, LogIPFilter, EventLogLevel >.
-query    Zeigt die aktuelle Diagnoseprotokollkonfiguration an.
-stop     Beendet den DNS-Serverdienst.
-start    Startet den DNS-Serverdienst.
-restart  Beendet den DNS-Serverdienst und wartet eine bestimmte
          Zeitspanne bevor der DNS-Dienst wieder gestartet wird.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
ConfigureDNSLogging.ps1 -change loglevel,107009

Ändert die Diagnoseprotokollkonfiguration, um alle über TCP eingehenden
DNS-Abfragen und -Antworten auf dem lokalen Computer zu überwachen.

ConfigureDNSLogging.ps1 -computer MunichServer -change
logPath, "C:\fso"

Ändert das Standardverzeichnis für die DNS-Diagnoseprotokollierung
auf einem Remoteserver namens MunichServer in C:\FSO.

ConfigureDNSLogging.ps1 -computer MunichServer -query

Fragt einen Remoteserver namens MunichServer nach allen Diagnoseprotokolleinstellungen ab.

ConfigureDNSLogging.ps1 -computer MunichServer -change eventloglevel, 4

Konfiguriert einen Remoteserver namens MunichServer, um alle DNS-Ereignisse
im Systemereignisprotokoll zu erfassen.

ConfigureDNSLogging.ps1 -computer MunichServer -restart 5

Startet den DNS-Dienst auf einem Remoteserver namens MunichServer neu.
Wartet 5 Sekunden zwischen dem Beenden und Starten des DNS-Dienstes.

ConfigureDNSLogging.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Die nächste Funktion ist *funchange()*. Die Funktion *funchange()* akzeptiert den Wert des Parameters **-change**, wenn dieser bei der Skriptausführung angegeben wurde. Deklarieren Sie in der Funktion *funchange()* die Variable *$class* und setzen Sie diese auf den Namen der abgefragten WMI-Klasse: *MicrosoftDNS_Server*. Greifen Sie auf WMI mit dem Cmdlet **Get-WmiObject** zu. Dieses Cmdlet stellt die Verbindung mit der in der Variablen *$class* angegebenen WMI-Klasse auf dem Computer her, der vom Parameter **-computer** festgelegt wird. Die WMI-Klasse *MicrosoftDNS_Server* befindet sich im WMI-Namespace *root\microsoftDNS*. Dieser Wert ist in den Parametern des Cmdlets hart codiert. Speichern Sie das zurückgegebene Objekt in der Variablen *$dnsserver*. Codeabschnitt:

```
function funchange($change)
{
  $class="MicrosoftDNS_Server"
  $dnsServer=Get-WmiObject -class $class -computername $computer `
  -namespace root\microsoftDNS
```

Die folgende *switch*-Anweisung enthält die Logik für die Funktion *funchange()*. Diese *switch*-Anweisung wertet das erste Element des Arrays in der Variablen *$change* aus. Wenn für den Parameter eine Übereinstimmung gefunden wird, führt die *switch*-Anweisung den entsprechenden Code aus. In der *switch*-Anweisung wird der Wert des Elements *1* des Arrays *$change* in die entsprechende Eigenschaft des *$dnsserver*-Objekts eingefügt. Übermitteln Sie die Informationen unter Verwendung der Methode *Put()* zurück an die WMI-Datenbank. Codeabschnitt:

```
  switch($change[0])
   {
    "LogLevel" { $dnsServer.logLevel = $change[1] ; $dnsServer.put() }
    "LogPath" { $dnsServer.logFilePath = $change[1] ; $dnsServer.put() }
    "LogSize" { $dnsServer.LogFileMaxSize = $change[1] ; $dnsServer.put() }
    "LogIPFilter" { $dnsServer.LogIPFilterList = $change[1] ; $dnsServer.put() }
    "EventLogLevel" { $dnsServer.EventLogLevel = $change[1] ; $dnsServer.put() }
    DEFAULT { "Sie müssen eine Aktion angeben." ; funhelp }
   }
}
```

Der Funktion *funquery()* ähnelt der Funktion *funchange()*, führt allerdings eine Standardabfrage der WMI-Klasse *MicrosoftDNS_Server* aus. Verwenden Sie das Cmdlet **Get-WmiObject** und formatieren Sie die Informationen aus dem zurückgegebenen Objekt in einer Liste. Geben Sie das Platzhalterzeichen (*) ein, um mehrere Eigenschaften auszuwählen. Beenden Sie anschließend das Skript mit der *exit*-Anweisung. Die Funktion ist wie folgt implementiert:

```
function funQuery()
{
 $class="MicrosoftDNS_Server"
 Get-WmiObject -class $class -computername $computer `
 -namespace root\microsoftDNS |
 format-list -property Log*, *log*
 exit
}
```

Der Funktion *funquery()* folgt die Funktion *funstart()*, um den DNS-Dienst auf einem Remoteserver zu starten. Verwenden Sie den gleichen WMI-Befehl wie in der Funktion *funquery()*. Der einzige Unterschied ist, dass Sie die Ausgabe des Objekts anstatt mit dem Cmdlet **Format-List** direkt mit der Methode *StartService()* des Objekts generieren. Beenden Sie anschließend das Skript. Codeabschnitt:

```
function funStart()
{
 $class="MicrosoftDNS_Server"
```

```
 $dnsServer = Get-WmiObject -class $class -computername $computer `
 -namespace root\microsoftDNS
 $dnsServer.StartService()
 exit
}
```

Die nächste Funktion ist *funstop()*. Die Funktion *funstop()* ist bis auf eine Ausnahme mit der Funktion *funstart()* identisch: Sie verwenden die Methode *StopService()* anstatt der Methode *StartService()*. Codeabschnitt:

```
function funStop()
{
 $class="MicrosoftDNS_Server"
 $dnsServer = Get-WmiObject -class $class -computername $computer `
 -namespace root\microsoftDNS
 $dnsServer.StopService()
 exit
}
```

Sie benötigen einige weitere Funktionen. Die erste Funktion ist *funrestart()*. Für die WMI-Klasse *MicrosoftDNS_Server* ist zwar standardmäßig keine Neustartmethode verfügbar, die Funktionalität kann aber einfach implementiert werden. Ein Neustart besteht im Wesentlichen aus dem Beenden und Starten des Dienstes. Aufgrund der Vielfalt von DNS-Serverversionen, ist das Festlegen der Zeitdauer zwischen den entsprechenden Stop- und Startaufrufen schwierig. Um dieses Problem zu umgehen, können Sie die Zeitdauer auf beliebig viele Sekunden festlegen. Stellen Sie eine WMI-Verbindung her und rufen Sie ein Objekt ab, das den DNS-Server repräsentiert. Verwenden Sie die WMI-Klasse *MicrosoftDNS_Server* und greifen Sie auf den WMI-Namespace *root\microsoftDNS* zu. Speichern Sie das zurückgegebene Objekt in der Variablen *$dnsserver* und geben Sie eine Meldung aus, dass der DNS-Dienst beendet wird. Rufen Sie die Methode *StopService()* auf und warten Sie die im Parameter **-restart** angegebene Zeitdauer ab. Geben Sie für jede Sekunde einen Punkt auf dem Bildschirm aus. Um diese Punkte auszugeben, verwenden Sie eine *for*-Anweisung und verwenden Sie das Cmdlet **Write-Host** mit dem Parameter **-nonewline**. Rufen Sie anschließend die Methode *StartService()* aus der WMI-Klasse *MicrosoftDNS_Server* auf. Codeabschnitt:

```
function funRestart($restart)
{
 $class="MicrosoftDNS_Server"
 $dnsServer = Get-WmiObject -class $class -computername $computer `
 -namespace root\microsoftDNS
 "Beenden des Dienstes ..."
 $dnsServer.StopService()
 for($i = 0 ; $i -le $restart ; $i++)
 {
  Start-Sleep -Seconds 1
  Write-Host "." -NoNewline
 }
  "Starten des Dienstes ..."
  $dnsServer.StartService()
 exit
}
```

Im Anschluss an diese Funktionen erstellen Sie den Code, der unmittelbar mit dem Skript ausgeführt wird. Legen Sie unter Verwendung von *if*-Anweisungen fest, welche Funktion ausgeführt werden soll. Wenn Sie die Variable *$help* finden, rufen Sie die Funktion *funhelp()* auf. Sollte die Variable *$query* vorhanden sein, rufen Sie die Funktion *funquery()* auf. Wenn die Variable *$change* existiert, geben

Sie eine Statusmeldung aus, die besagt, dass die Eigenschaft im Element *0* des Arrays *$change* in den Wert im Element *1* des Arrays *$change* geändert wird. Rufen Sie die Funktion *funchange()* auf und übergeben Sie das gesamte *$change*-Array. Codeabschnitt:

```
if($help)   {   "Ausgeben der Hilfeinformationen ..."   ;   funHelp   }
if($query)  { "Ausgeben der aktuellen Protokolleinstellungen des DNS-Servers" ; funQuery }
if($change)
  {
    "Ändern von $($change[0]) in $($change[1]) ..." ;
    funChange($change)
  }
```

Wenn keine dieser Variablen vorhanden ist, suchen Sie nach der Variablen *$start*. Ist diese Variable vorhanden, rufen Sie die Funktion *funstart()* auf. Finden Sie die Variable *$stop*, führen Sie stattdessen die Funktion *funstop()* aus. Sollten Sie einen Neustart ausführen müssen, geben Sie eine Statusmeldung aus, dass der DNS-Dienst in einigen Sekunden entsprechend dem im Parameter **-restart** angegebenen Wert neu gestartet wird. Rufen Sie anschließend die Funktion *funrestart()* auf und übergeben Sie den Wert in der Variablen *$restart*. Codeabschnitt:

```
if($start)   { "Starten des Dienstes ..." ; funStart }
if($stop)    { "Beenden des Dienstes ..." ; funStop }
if($restart) { "Neustarten des Dienstes in $($restart) Sekunden ..."
;funRestart($restart) }
```

Wenn das Skript ohne Parameter ausgeführt wird, geben Sie eine Meldung aus, dass keine Aktion angegeben wurde und rufen Sie die Funktion *funhelp()* in der *else*-Anweisung auf:

```
ELSE
  { "Es wurde keine Aktion angegeben ..." ; funhelp }
```

Das vollständige Skript *ConfigureDNSLogging.ps1* hat folgenden Inhalt:

ConfigureDNSLogging.ps1

```
param(
      $computer="localhost", $change, [switch]$query, $restart,
      [switch]$stop, [switch]$start, [switch]$help
     )
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ConfigureDNSLogging.ps1
Konfiguriert die Diagnoseprotokollfunktionen eines DNS-Servers
auf dem lokalen Computer oder auf einem Remotecomputer.

PARAMETER:
-computer Gibt den Computer an, für den das Skript ausgeführt werden soll.
-change   Gibt die zu konfigurierenden Eigenschaften des DNS-Servers an: < LogLevel,
          LogPath, LogSize, LogIPFilter, EventLogLevel >.
-query    Zeigt die aktuelle Diagnoseprotokollkonfiguration an.
-stop     Beendet den DNS-Serverdienst.
-start    Startet den DNS-Serverdienst.
-restart  Beendet den DNS-Serverdienst und wartet eine bestimmte
          Zeitspanne bevor der DNS-Dienst wieder gestartet wird.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
```

```
ConfigureDNSLogging.ps1 -change loglevel,107009

Ändert die Diagnoseprotokollkonfiguration, um alle über TCP eingehenden
DNS-Abfragen und -Antworten auf dem lokalen Computer zu überwachen.

ConfigureDNSLogging.ps1 -computer MunichServer -change
logPath, "C:\fso"

Ändert das Standardverzeichnis für die DNS-Diagnoseprotokollierung
auf einem Remoteserver namens MunichServer in C:\FSO.

ConfigureDNSLogging.ps1 -computer MunichServer -query

Fragt einen Remoteserver namens MunichServer nach allen Diagnoseprotokolleinstellungen ab.

ConfigureDNSLogging.ps1 -computer MunichServer -change eventloglevel, 4

Konfiguriert einen Remoteserver namens MunichServer, um alle DNS-Ereignisse
im Systemereignisprotokoll zu erfassen.

ConfigureDNSLogging.ps1 -computer MunichServer -restart 5

Startet den DNS-Dienst auf einem Remoteserver namens MunichServer neu.
Wartet 5 Sekunden zwischen dem Beenden und Starten des DNS-Dienstes.

ConfigureDNSLogging.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}

function funchange($change)
{
  $class="MicrosoftDNS_Server"
  $dnsServer=Get-WmiObject -class $class -computername $computer `
  -namespace root\microsoftDNS
  switch($change[0])
   {
    "LogLevel" { $dnsServer.logLevel = $change[1] ; $dnsServer.put() }

    "LogPath" { $dnsServer.logFilePath = $change[1] ; $dnsServer.put() }
    "LogSize" { $dnsServer.LogFileMaxSize = $change[1] ; $dnsServer.put() }
    "LogIPFilter" { $dnsServer.LogIPFilterList = $change[1] ; $dnsServer.put() }
    "EventLogLevel" { $dnsServer.EventLogLevel = $change[1] ; $dnsServer.put() }
    DEFAULT { "Sie müssen eine Aktion angeben." ; funhelp }
   }
}

function funQuery()
{
 $class="MicrosoftDNS_Server"
 Get-WmiObject -class $class -computername $computer `
 -namespace root\microsoftDNS |
```

```
 format-list -property Log*, *log*
 exit
}

function funStart()
{
 $class="MicrosoftDNS_Server"
 $dnsServer = Get-WmiObject -class $class -computername $computer `
 -namespace root\microsoftDNS
 $dnsServer.StartService()
 exit
}

function funStop()
{
 $class="MicrosoftDNS_Server"
 $dnsServer = Get-WmiObject -class $class -computername $computer `
 -namespace root\microsoftDNS
 $dnsServer.StopService()
 exit
}

function funRestart($restart)
{
 $class="MicrosoftDNS_Server"
 $dnsServer = Get-WmiObject -class $class -computername $computer `
 -namespace root\microsoftDNS
 "Beenden des Dienstes ..."
 $dnsServer.StopService()
 for($i = 0 ; $i -le $restart ; $i++)
 {
  Start-Sleep -Seconds 1
  Write-Host "." -NoNewline
 }
  "Starten des Dienstes ..."
  $dnsServer.StartService()
 exit
}

if($help)   { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if($query)  { "Ausgeben der aktuellen Protokolleinstellungen des DNS-Servers ..." ; funQuery }
if($change)


  {
    "Change $($change[0]) to $($change[1]) now ..." ;
    funChange($change)
  }
if($start)   { "Starten des Dienstes ..." ; funStart }
if($stop)    { "Beenden des Dienstes ..." ; funStop }
if($restart) { "Neustarten des Dienstes in $($restart) Sekunden ..." ;
funRestart($restart) }
ELSE
  { "Es wurde keine Aktion angegeben ..." ; funhelp }
```

Überprüfen von Stammhinweisen

Das Konfigurieren von Stammhinweisen gehört normalerweise nicht zum Aufgabenbereich eines Netzwerkadministrators. Viele Netzwerkadministratoren haben sich tatsächlich noch nie mit Stammhinweisen befasst. Warum müssen Sie dann also die Stammhinweise überprüfen? Das Überprüfen der Stammhinweise ist erforderlich, da Probleme bei der DNS-Namensauflösung manchmal von veralteten Konfigurationseinstellungen für die Stammhinweise verursacht werden. Ein DNS-Server sucht die autorisierenden DNS-Stammserver über Stammhinweise. Diese Server lösen Namen für .com, .net und .org auf. Die Server müssen deshalb zuverlässig erreichbar sein und sollten möglichst nicht geändert werden. Stammhinweise werden in Windows normalerweise über Service Packs oder Hotfixes aktualisiert. Das Skript *DisplayRootHints.ps1* zeigt die Konfiguration der Stammhinweise eines Servers an. Die Stammhinweise sind in Abbildung 18.3 dargestellt.

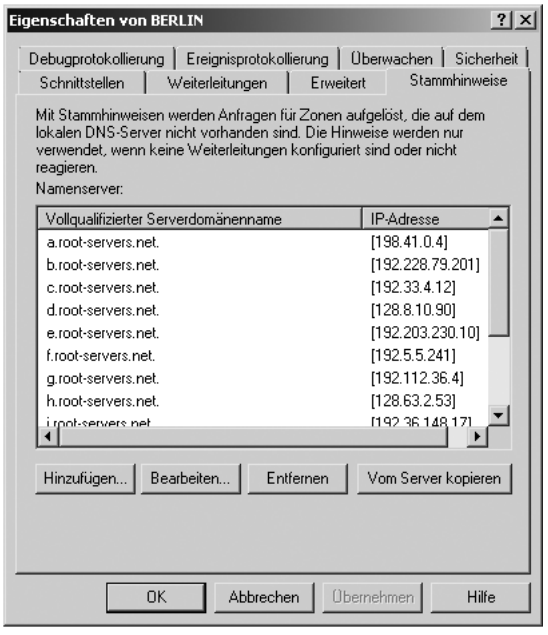


Abbildung 18.3 Stammhinweise im DNS-Manager

Das Skript *DisplayRootHints.ps1* stellt mit dem Cmdlet **Get-WmiObject** die Verbindung mit dem Namespace *root\microsoftDNS* her. Anschließend ruft das Cmdlet alle Instanzen der WMI-Klasse *MicrosoftDNS_AType* ab. Fügen Sie das resultierende Verwaltungsobjekt in das Cmdlet **Where-Object** ein. Verwenden Sie im Codeblock für das Cmdlet **Where-Object** die Variable *$_ automatic*, die das aktuelle Objekt in der Pipeline repräsentiert. Suchen Sie in der Eigenschaft *OwnerName* der WMI-Klasse *MicrosoftDNS_AType* nach der Zeichenfolge *root*. Wenn die Zeichenfolge *root* vorhanden ist, fügen Sie das gefilterte Objekt in das Cmdlet **Format-Table** ein und zeigen Sie nur die Eigenschaft *TextRepresentation* der WMI-Klasse *MicrosoftDNS_AType* an.

Das vollständige Skript *DisplayRootHints.ps1* hat folgenden Inhalt:

DisplayRootHints.ps1

```
Get-WmiObject -Namespace root\microsoftdns -Class MicrosoftDNS_AType |
Where-Object { $_.ownerName -match 'root' } |
format-table textRepresentation
```

Abrufen von A-Einträgen

Zusätzlich zu den Stammhinweisen können Sie auch alle A-Einträge für eine DNS-Domäne abfragen. (Ein A-Eintrag ordnet einen Hostnamen einer IP-Adresse zu). Verwenden Sie die gleiche DNS-Klasse wie zum Abrufen der Stammhinweise, aber filtern Sie die Ergebnisse, um diese auf eine bestimmte Domäne zu beschränken. Da auf einem DNS-Server mehrere DNS-Domänen vorhanden sein können, fügen Sie einen Befehlszeilenparameter zum Skript hinzu, um die Einträge für eine bestimmte DNS-Domäne abzurufen. Das fertige Skript hat den Namen *QueryDNSARecords.ps1*. Ein Beispiel für A-Einträge ist in Abbildung 8.4 dargestellt.

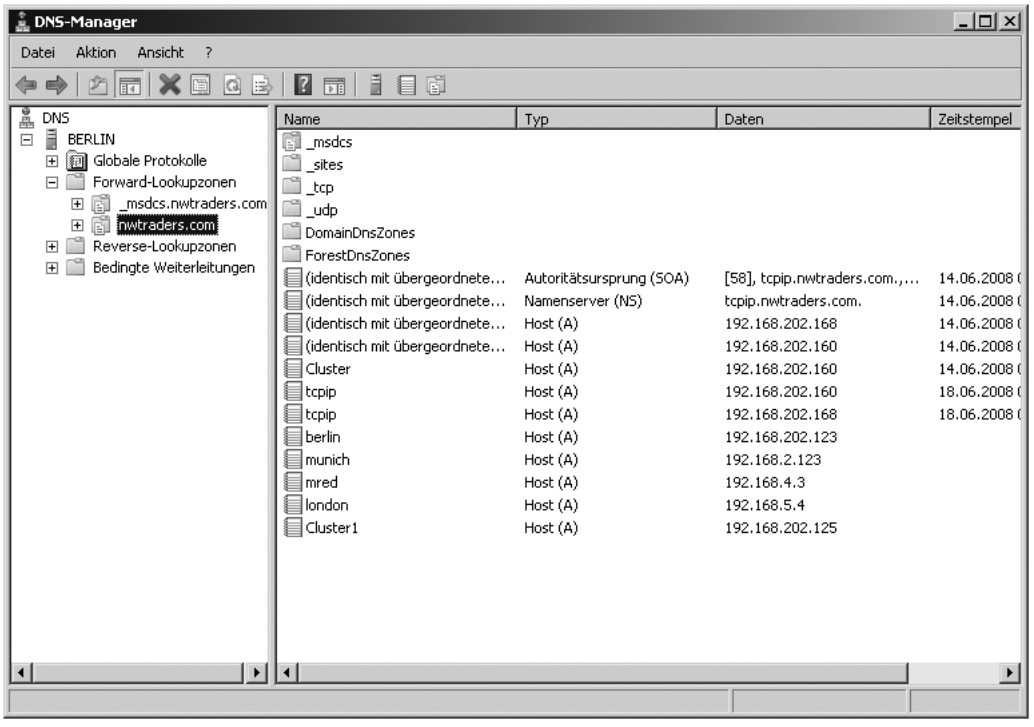


Abbildung 18.4 A-Einträge im DNS-Manager

Beginnen Sie das Skript *QueryDNSARecords.ps1* mit einer *param-Anweisung.* Da das Skript mehrere Befehlszeilenparameter umfasst, verwenden Sie diese Parameter zum Steuern der Skriptausführung anstatt das Skript direkt zu ändern. Die folgenden drei Parameter müssen definiert werden: Der Name des Computers (**-computer**), die Domäne (**-domain**) und die Anzeige der Hilfe (**-help**). Die *param*-Anweisung lautet:

```
param($computer="localhost",$domain,[switch]$help)
```

Die nächste Funktion ist *funhelp()*. Geben Sie die Hilfeinformationen in eine *here*-Zeichenfolge ein. Der Syntaxabschnitt ist der längste Abschnitt im Hilfetext, da an dieser Stelle Syntaxbeispiele für das Skript aufgeführt werden. Geben Sie in der *here*-Zeichenfolge eine kurze Beschreibung des Skripts ein. Fassen Sie im nächsten Abschnitt die Parameter mit einer kurzen Beschreibung zusammen. Der dritte Abschnitt enthält die bereits erwähnten Syntaxbeispiele für die Befehlszeilenparameter. Nachdem Sie die *here*-Zeichenfolge erstellt haben, weisen Sie diese der Variablen *$helptext* zu, geben Sie den Inhalt der Variablen aus und beenden Sie das Skript. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: QueryDNSArecords.ps1
Fragt die A-Einträge auf einem lokalen Computer oder einem Remotecomputer ab,
auf dem der Microsoft DNS-Dienst ausgeführt wird.

PARAMETER:
-computer  Gibt den Namen des Computer an, für den das Skript ausgeführt werden soll.
-domain    Gibt die Domäne an, für die die A-Einträge abgefragt werden.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
QueryDNSArecords.ps1 -domain contoso.com

Ermittelt die A-Einträge der Domäne contoso.com. Verwendet den lokalen Computer.

QueryDNSArecords.ps1 -domain nwtraders.com

Ermittelt die A-Einträge der Domäne nwtraders.com. Verwendet den lokalen Computer.

QueryDNSArecords.ps1 -computer MunichServer -domain nwtraders.com

Stellt eine Verbindung mit einem Computer namens MunichServer her, auf dem
der Microsoft DNS-Dienst ausgeführt wird. Ermittelt die A-Einträge der Domäne nwtraders.com.

QueryDNSArecords.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Sie müssen nun die Befehlszeilenparameter überprüfen. Ermitteln Sie mit der *if*-Anweisung, ob das Skript mit dem Parameter **-help** ausgeführt wurde, indem Sie überprüfen, ob die Variable *$help* vorhanden ist. Um die Verarbeitung nun etwas interessanter zu gestalten, verwenden Sie eine *for*-Schleife, die Sie acht Mal durchlaufen, wenn die Variable *$help* vorhanden ist. Geben Sie hierzu den Befehl *$i = 0 ; $i –le 15 ; $i+=2 ein.* Zählen Sie von 0 bis 15, indem Sie jede zweite Zahl auslassen. Der Zahlenbereich 0 bis 15 entspricht der Anzahl der Farben, die das Cmdlet **Write-Host** anzeigen kann. Wenn Sie in der *for*-Anweisung nur die Zahlen 0 bis 7 verwenden, werden nur die vier Standardfarben und einige abgeleitete Farben angezeigt. Indem Sie jede zweite Farbe überspringen, erhalten Sie die gesamte Farbpalette. Verwenden Sie anschließend die automatische Variable *$myinvocation*.

Was sind automatische Objekte?

Die automatische Variable *$myinvocation* wird in der Windows PowerShell-Dokumentation nicht beschrieben. Sie finden die Variable jedoch mit dem Cmdlet **Get-ChildItem** auf dem *PSDrive*-Laufwerk *variable:*\. Hierzu ist folgender Code erforderlich:

```
Get-ChildItem variable:\
```

Die Variable *$myinvocation* wird in eine Instanz des Microsoft .NET Framework-Objekts *System.Management.Automation.InvocationInfo* aufgelöst. Dies ist keine Überraschung, da mehrere automatische Variablen in Objekte aufgelöst werden. Zeigen Sie diese Informationen mit dem folgenden Befehl an:

```
get-childitem variable:\  |  where-object { $_.value -match 'system' }
```

Nachdem Sie die automatischen Variablen, die Objekte enthalten, identifiziert haben, überprüfen Sie das Objekt mit dem Cmdlet **Get-Member**.

```
$MyInvocation | Get-Member
```

Das in der Variablen *$myinvocation* gespeicherte Objekt stellt mehrere nützliche Eigenschaften bereit. Diese Eigenschaften sind verfügbar, da die Variable *$myinvocation* auf ein tatsächliches Objekt verweist.

Wenn Sie das *$MyInvocation.MyCommand*-Objekt überprüfen, werden Sie feststellen, dass dieses das Objekt *System.Management.Automation.ScriptInfo* zurückgibt. Zeigen Sie diese Informationen mit dem folgenden Befehl an:

```
$MyInvocation.MyCommand | Get-Member
```

Unter „Was sind automatische Objekte?“ wurde erklärt, wie Sie die für die automatische Variable *$myinvocation* konfigurierten Eigenschaften ermitteln können. Geben Sie den Namen des Skripts mit Hilfe der Eigenschaft *MyCommand* der automatischen Variablen *$myinvocation* aus. Dies ist eine gute Methode zum Überprüfen des Namens des ausgeführten Skripts. Halten Sie die Skriptausführung mit dem Cmdlet **Start-Sleep** an und löschen Sie den Bildschirm mit dem Cmdlet **Clear-Host**. Rufen Sie anschließend die Funktion *funhelp()* auf. Codeabschnitt:

```
if($help)   {
             for($i = 0 ; $i -le 15 ; $i+=2)
              {
               write-host -foregroundcolor $i `
               "Ausgeben der Hilfeinformationen für $($myinvocation.mycommand)"
               start-sleep -milliseconds 100
               clear-host
              }
             funHelp
            }
```

Sie müssen sicherstellen, dass das Skript mit dem Parameter **-domain** ausgeführt wurde, indem Sie überprüfen, ob die Variable *$domain* vorhanden ist. Wenn die Variable nicht vorhanden ist, geben Sie eine Meldung aus und rufen Sie die Funktion *funhelp()* auf.

```
if(!$domain)   { "Fehlender Parameter -domain ..." ; funHelp }
```

Das wichtigste Modul des Skripts ist das Cmdlet **Get-WmiObject**, das eine Verbindung mit dem WMI-Namespace *root\microsoftDNS* herstellt. Geben Sie mit dem Parameter **-class** die WMI-Klasse *MicrosoftDNS_AType* an und stellen Sie mit dem Parameter **-computername** eine Verbindung mit dem Zielcomputer her. Die Angabe des Filters ist etwas komplizierter.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Hinweis Aufgrund der WMI-Abhängigkeiten verwendet Windows PowerShell eine andere Syntax für WMI-Filter als für das Cmdlet **Where-Object**. Ein offensichtlicher Unterschied ist die Verwendung der Gleichheitszeichen (=) anstatt von **-eq**. Ein weiterer Unterschied ist, dass die Zeichenfolgenwerte in Anführungszeichen gesetzt werden müssen. Um Anführungszeichen an Variablen zuzuweisen, müssen Sie vor den Anführungszeichen ein Gravis-Zeichen (`) eingeben.

Damit der Benutzer im Parameter **-domain** den Domänennamen ohne Anführungszeichen eingeben kann, fügen Sie die für den Zeichenfolgenwert erforderlichen Anführungszeichen in die **Get-WmiObject**-Anweisung direkt ein. Sie müssen vor den Anführungszeichen ein Graviszeichen angeben, damit Windows PowerShell erkennt, dass die Zeichenfolge nicht mit dem ersten Anführungszeichen beendet ist, das der Variablen *$domain* vorausgeht. Weisen Sie das von der Anweisung und dem Filter zurückgegebene Verwaltungsobjekt der Variablen *$arydns* zu. Codeabschnitt:

```
$arydns = Get-WmiObject -Namespace root\microsoftdns -Class MicrosoftDNS_AType `
             -computername $computer -filter "domainName = `"$domain`" "
```

Geben Sie eine Kopfzeile für die Liste der DNS-Namen und Adressen aus. Sie können hierzu eine der *funline()*-Funktionen verwenden, die Sie in den Skripts auf der Begleit-CD finden können. Im Folgenden wird das Ausgeben einer Kopfzeile unter Verwendung von Inlinecode erklärt. Diese Methode unterliegt jedoch einigen Einschränkungen. Abhängig von der Länge des Namens des DNS-Servers, ist die Ausgabe möglicherweise nicht optimal ausgerichtet. Der folgende Codeabschnitt gibt eine Kopfzeile aus, gefolgt von einem Tabstoppzeichen und der Eigenschaft *DnsServerName* aus dem ersten von der Abfrage zurückgegebenen A-Eintrag. Der erste A-Eintrag ist das Element *0* im Array der DNS-Einträge, aus denen das Verwaltungsobjekt besteht, das von der WMI-Abfrage zurückgegeben wurde. Codeabschnitt:

```
"*** A-Einträge des DNS-Servers:
`t$($arydns[0].dnsServerName)
`t--------------------"
```

Sie müssen die Ausgabe mit der *foreach*-Anweisung formatieren und die Verwaltungsobjekte in der Variablen *$aryDNS* durchlaufen. Erstellen Sie im Codeblock eine Hashtabelle, die dem in VBScript und anderen Programmiersprachen verwendeten *Dictionary*-Objekt entspricht. Die Hashtabelle besteht aus einem Schlüssel/Wert-Paar. Fügen Sie in der Hashtabelle die Eigenschaft *OwnerName* zum Schlüssel und die Eigenschaft *RecordData* zum Wert hinzu. Erstellen Sie mit der Konstruktion += die Ausgabevariable *$hash*, die die vollständige Hashtabelle enthält. Geben Sie die Hashtabelle aus. Codebeispiel:

```
foreach($dns in $aryDNS)
{
  $hash += @{ $dns.ownername = $dns.recordData }
}
$hash
```

Das vollständige Skript *QueryDNSARecords.ps1* hat folgenden Inhalt:

QueryDNSARecords.ps1

```
param($computer="localhost",$domain,[switch]$help)

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: QueryDNSArecords.ps1
```

```
Fragt die A-Einträge auf einem lokalen Computer oder einem Remotecomputer ab,
auf dem der Microsoft DNS-Dienst ausgeführt wird.

PARAMETER:
-computer  Gibt den Namen des Computer an, für den das Skript ausgeführt werden soll.
-domain    Gibt die Domäne an, für die die A-Einträge abgefragt werden.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
QueryDNSArecords.ps1 -domain contoso.com

Ermittelt die A-Einträge der Domäne contoso.com. Verwendet den lokalen Computer.

QueryDNSArecords.ps1 -domain nwtraders.com

Ermittelt die A-Einträge der Domäne nwtraders.com. Verwendet den lokalen Computer.

QueryDNSArecords.ps1 -computer MunichServer -domain nwtraders.com

Stellt eine Verbindung mit einem Computer namens MunichServer her, auf dem
der Microsoft DNS-Dienst ausgeführt wird. Ermittelt die A-Einträge der Domäne nwtraders.com.

QueryDNSArecords.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}

if($help)   {
             for($i = 0 ; $i -le 15 ; $i+=2)
              {
               write-host -foregroundcolor $i `
               "Ausgeben der Hilfeinformationen für $($myinvocation.mycommand)"
               start-sleep -milliseconds 100
               clear-host
              }
             funHelp
            }
if(!$domain)   { "Fehlender Parameter -domain ..." ; funHelp }

$arydns = Get-WmiObject -Namespace root\microsoftdns -Class MicrosoftDNS_AType `
              -computername $computer -filter "domainName = `"$domain`" "

"*** A-Einträge des DNS-Servers:
`t$($arydns[0].dnsServerName)
`t--------------------"

foreach($dns in $aryDNS)
{
  $hash += @{ $dns.ownername = $dns.recordData }
}
$hash
```

Konfigurieren der DNS-Servereinstellungen

Anstatt die DNS-Servereinstellungen nacheinander zu bearbeiten, können Sie die Einstellungen gleichzeitig konfigurieren. Sie können das Skript *SetDNSServerConfig.ps1* beispielsweise so ändern, dass dieses alle Eigenschaften und Werte für die DNS-Serverkonfiguration akzeptiert. Dies ermöglicht das gleichzeitige Konfigurieren mehrerer Parameter.

Das Skript beginnt mit einer *param*-Anweisung, die die Eingabe von Werten zur Laufzeit ermöglicht. Der Parameter **-computer** gibt einen lokalen Server oder einen Remoteserver an. Wenn Sie das Skript auf dem lokalen Server ausführen, sind keine weiteren Konfigurationseinstellungen erforderlich, da ein Standardwert vorhanden ist. Um das Skript für einen Remoteserver auszuführen, müssen Sie den Namen des Remotecomputers im Parameter **-computer** angeben. Der Parameter **-change** ermöglicht die Eingabe eines Eigenschaftenarrays und der dazugehörigen Werte. Die übrigen Parameter haben feste Werte. Die *param*-Anweisung lautet:

```
param($computer="localhost", $change, [switch]$query,
      [switch]$list,[switch]$help)
```

Die nächste Funktion ist *funhelp()*. Verwenden Sie in dieser Funktion eine *here*-Zeichenfolge, um die Hilfeinformationen zu definieren. Speichern Sie die *here*-Zeichenfolge in der Variablen *$helptext*. Geben Sie anschließend den Inhalt der Variablen *$helptext* aus und beenden Sie das Skript. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: SetDNSServerConfig.ps1
Erstellt eine Liste mit den DNS Server-Konfigurationsinformationen des lokalen Computers
oder eines Remotecomputers und bietet die Möglichkeit, die Konfiguration zu ändern.

PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-list     Gibt die aktuellen Konfigurationsinformationen des DNS-Servers aus.
-change   Gibt die zu ändernden Eigenschaften und deren Werte an.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
SetDNSServerConfig.ps1 -list

Gibt die aktuelle DNS Server-Konfiguration des lokalen Computers aus.

SetDNSServerConfig.ps1 -computer MunichServer -list

Gibt die aktuelle DNS Server-Konfiguration
eines Remotecomputers namens MunichServer aus.

SetDNSServerConfig.ps1 -computer MunichServer -change "RoundRobin",0

Konfiguriert einen Remotecomputer namens MunichServer, um RoundRobin zu deaktivieren.

SetDNSServerConfig.ps1 -computer MunichServer -change "RoundRobin",-1,
"AllowUpdate",0,eventloglevel,1

Konfiguriert einen Remotecomputer namens MunichServer, um RoundRobin zu aktivieren,
```

```
uneingeschränkte Aktualisierungen zuzulassen und nur Fehler im Ereignisprotokoll zu erfassen.

SetDNSServerConfig.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Der Funktion *funhelp()* folgt die Funktion *funlist*(). Diese Funktion listet alle Eigenschaften auf, die für den DNS-Server festgelegt werden können. Da zahlreiche Eigenschaften aufgeführt werden, geben Sie nicht alle Informationen in der *here*-Zeichenfolge an, um das Skript übersichtlicher zu machen. Der Nachteil ist, dass die Textdatei im gleichen Pfad gespeichert werden muss, in dem das Skript ausgeführt wird, da das Skript ansonsten einen Fehler generiert. Suchen Sie mit dem Cmdlet **Test-Path** nach der Datei *SetDNSServerConfigOptions.txt*. Das Cmdlet **Test-Path** gibt *True* oder *False* zurück. Da ein boolescher Wert zurückgegeben wird, setzen Sie die gesamte *Test-Path*-Anweisung in runde Klammern und analysieren den Wert mit einer *if*-Anweisung. Existiert die Datei, zeigen Sie diese in Notepad an. Sollte die Datei nicht gefunden werden, geben Sie mit dem Cmdlet **Write-Host** eine entsprechende Meldung aus. Zeigen Sie die Meldung mit dem Parameter **-foregroundcolor** in Rot an. Codeabschnitt:

```
function funList()
{
 if(test-path .\SetDNSServerConfigOptions.txt)
  {
   .\SetDNSServerConfigOptions.txt
  }
 ELSE
  {
   Write-Host -foregroundcolor red `
   "SetDNSServerConfigOptions.txt konnte nicht gefunden werden."
  }

}
```

Die nächste Funktion ist *funquery()*. Diese Funktion fragt den DNS-Server ab und listet die aktuellen Einstellungen des Servers auf. Mit *Funquery()* können Sie die Konfiguration des DNS-Servers auch überprüfen, nachdem Sie Änderungen vorgenommen haben. Beginnen Sie die Funktion *funquery()*, indem Sie die WMI-Klasse *MicrosoftNDS_Server* angeben. Greifen Sie mit dem Cmdlet **Get-WmiObject** auf WMI zu. Geben Sie den Klassennamen in der Variablen *$class* und den Computernamen in der Variablen *$computer* an. Hart codieren Sie den Namespace *root\microsoftDNS*, da sich die Klasse nur in diesem Namespace befinden kann. Fügen Sie das zurückgegebene Objekt in das Cmdlet **Format-List** ein. Geben Sie nur die Elemente aus, die mit einem Buchstaben zwischen *a* und *z* beginnen, um Systemeigenschaften auszuschließen. Nachdem die Abfrage ausgeführt wurde, beenden Sie das Skript. Codeabschnitt:

```
function funQuery()
{
 $class="MicrosoftDNS_Server"
 Get-WmiObject -class $class -computername $computer `
 -namespace root\microsoftDNS |
 format-list [a-z]*
 exit
}
```

Der komplizierteste Abschnitt des Skripts ist die Funktion *funchange()*. Beginnen Sie die Funktion *funchange()*, indem Sie die WMI-Klasse *MicrosoftDNS_Server* der Variablen *$class* zuweisen und mit dem Cmdlet **Get-WmiObject** die Verbindung mit WMI herstellen. Speichern Sie das zurückgegebene Objekt in der Variablen *$dnsserver*. Dieser Abschnitt der Funktion *funchange()* ist anderen Funktionen ähnlich:

```
function funChange($change)
 {
  $class="MicrosoftDNS_Server"
  $dnsServer=Get-WmiObject -class $class -computername $computer `
  -namespace root\microsoftDNS
```

Nun folgt der schwierige Teil. Um das Ändern mehrerer Eigenschaften mit einem einzigen Parameter zu ermöglichen, müssen Sie die im Parameter **-change** angegebenen Informationen in die einzelnen Eigenschaft/Werte-Paare aufschlüsseln. Am besten konvertieren Sie das Array in eine Hashtabelle. Durchlaufen Sie eine *for*-Anweisung bis zur Anzahl der Elemente im *$change*-Array. Sie müssen dabei *1* von der Anzahl subtrahieren, da das Array auf Null basiert. Überspringen Sie jedes zweite Element in der Variablen *$element*, um nur jedes zweite Elemente aus dem Array abzurufen. Erstellen Sie die Hashtabelle, indem Sie die geraden Elementnummern in die Schlüsselposition in der Tabelle einfügen. Fügen Sie die ungeraden Elementnummern in die Wertposition in der Tabelle ein. Mit dem Operator += können Sie alle nachfolgenden Schlüssel/Wert-Paare zur Hashtabelle hinzufügen. Die Zeichenkombination @{} definiert eine Hashtabelle. Die Funktion *funchange()* ist dementsprechend wie folgt implementiert:

```
  for ($element=0 ; $element -le $change.length-1 ; $element+=2)
   {
    $hash += @{ $change[$element]=$change[$element+1] }
   }
```

Sie müssen nun die Hashtabelle durchlaufen. Generieren Sie hierzu mit der Eigenschaft *Keys* mehrere Hashtabellenschlüssel. Mit einer *foreach*-Anweisung können Sie einen dieser Schlüssel verwenden. Überwachen Sie mit der Variablen *$prop* die Position in der Schlüsselsammlung. Geben Sie im Codeblock für die *foreach*-Anweisung eine Statusmeldung aus. Erweitern Sie in den doppelten Anführungszeichen der Statusmeldung den Wert aus der Variablen *$prop*. Die Variable *$prop* wird als Enumerator für die Hashschlüssel verwendet. Geben Sie das **`t**-Zeichen ein, um zur nächsten Position zu wechseln. Für zwei Tabstopps in der Ausgabe geben Sie **`t`t** ein. Drei **`t`t`t** geben drei Tabstopps an. Rufen Sie mit der Variablen *$prop* den Wert aus der Hashtabelle ab. Um den Wert der Eigenschaft auszugeben, geben Sie folgenden Unterausdruck ein:

```
$($hash[$prop])
```

Der Unterausdruck stellt sicher, dass der Code in den Klammern zuerst ausgewertet wird. Geben Sie den Eigenschaftsnamen an, um die entsprechende Eigenschaft vom DNS-Server abzurufen und auf den Wert in der Hashtabelle zuzugreifen. Weisen Sie den Eigenschaftswert aus der Hashtabelle der entsprechenden Eigenschaft in WMI zu. Übermitteln Sie die Informationen unter Verwendung der Methode *Put()* an die WMI-Datenbank. Codeabschnitt:

```
  foreach($prop in $hash.keys)
 { "Vorbereiten der folgenden Konfigurationsänderung: "
  "$prop `t`t$($hash[$prop])"
  $dnsServer.$prop = $hash[$prop]
  $dnsServer.put()
 }
 }
```

Am Ende des Skript befinden sich vier Anweisungen, die die Flexibilität des Skripts sicherstellen. Die erste Anweisung sucht nach der Variablen *$help* und ruft die Funktion *funhelp()* auf, wenn die Variable existiert. Ist die Variable *$list* vorhanden, rufen Sie die Funktion *funlist()* auf und beenden Sie das Skript. Wenn die Variable *$query* vorhanden ist, rufen Sie die Funktion *funquery()* auf. Die nächste Variable ist *$change*. Wenn Sie diese Variable finden, rufen Sie die Funktion *funchange()* auf. Die vier Befehle lauten:

```
if($help)   { "Ausgeben der Hilfeinformationen ..."   ;   funHelp   }
if($list)   { "Ausgeben aller konfigurierbaren Eigenschaften ..." ; funList }
if($query)  { "Ausgeben der aktuellen DNS-Serverkonfiguration ..." ; funQuery }
if($change) { "Ändern von $change ..." ; funChange($change) }
```

Das vollständige Skript *SetDNSServerConfig.ps1* hat folgenden Inhalt:

SetDNSServerConfig.ps1

```
param($computer="localhost", $change, [switch]$query,
      [switch]$list,[switch]$help)
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: SetDNSServerConfig.ps1
Erstellt eine Liste mit den DNS Server-Konfigurationsinformationen des lokalen Computers
oder eines Remotecomputers und bietet die Möglichkeit, die Konfiguration zu ändern.

PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-list     Gibt die aktuellen Konfigurationsinformationen des DNS-Servers aus.
-change   Gibt die zu ändernden Eigenschaften und deren Werte an.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
SetDNSServerConfig.ps1 -list

Gibt die aktuelle DNS Server-Konfiguration des lokalen Computers aus.

SetDNSServerConfig.ps1 -computer MunichServer -list

Gibt die aktuelle DNS Server-Konfiguration
eines Remotecomputers namens MunichServer aus.

SetDNSServerConfig.ps1 -computer MunichServer -change "RoundRobin",0

Konfiguriert einen Remotecomputer namens MunichServer, um RoundRobin zu deaktivieren.

SetDNSServerConfig.ps1 -computer MunichServer -change "RoundRobin",-1,
"AllowUpdate",0,eventloglevel,1

Konfiguriert einen Remotecomputer namens MunichServer, um RoundRobin zu aktivieren,
uneingeschränkte Aktualisierungen zuzulassen und nur Fehler im Ereignisprotokoll zu erfassen.

SetDNSServerConfig.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
```

```
 $helpText
 exit
}

function funList()
{
 if(test-path .\SetDNSServerConfigOptions.txt)
  {
   .\SetDNSServerConfigOptions.txt
  }
 ELSE
  {
   Write-Host -foregroundcolor red `
   "SetDNSServerConfigOptions.txt konnte nicht gefunden werden."
  }

}

function funQuery()
{
 $class="MicrosoftDNS_Server"
 Get-WmiObject -class $class -computername $computer `
 -namespace root\microsoftDNS |
 format-list [a-z]*
 exit
}

function funChange($change)
 {
  $class="MicrosoftDNS_Server"
  $dnsServer=Get-WmiObject -class $class -computername $computer `
  -namespace root\microsoftDNS
  for ($element=0 ; $element -le $change.length-1 ; $element+=2)
   {

    $hash += @{ $change[$element]=$change[$element+1] }
   }
  foreach($prop in $hash.keys)
 { "Vorbereiten der folgenden Konfigurationsänderung: "
  "$prop `t`t$($hash[$prop])"
  $dnsServer.$prop = $hash[$prop]
  $dnsServer.put()
 }
 }

if($help)   { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if($list)   { "Ausgeben aller konfigurierbaren Eigenschaften ..." ; funList }
if($query)  { "Ausgeben der aktuellen DNS-Serverkonfiguration ..." ; funQuery }
if($change) { "Ändern von $change  ..." ; funChange($change) }
```

Überprüfen von DNS-Zonen

Die ordnungsgemäße Konfiguration von DNS ist ausschlaggebend für die Funktionalität von Active Directory und anderen Anwendungen, die Namensauflösungsdienste benötigen, um mit anderen Computern im Netzwerk zu kommunizieren. Wenn Sie Active Directory bereitgestellt haben, sind bereits einige Zonen auf Ihrem DNS-Server vorhanden. Mit dem Skript *ReportDNSZoneConfig.ps1* können Sie überprüfen, ob während der Installation die richtigen DNS-Zonen erstellt wurden. Sie können die DNS-Zonen auch im DNS-Manager überprüfen (siehe Abbildung 18.5).

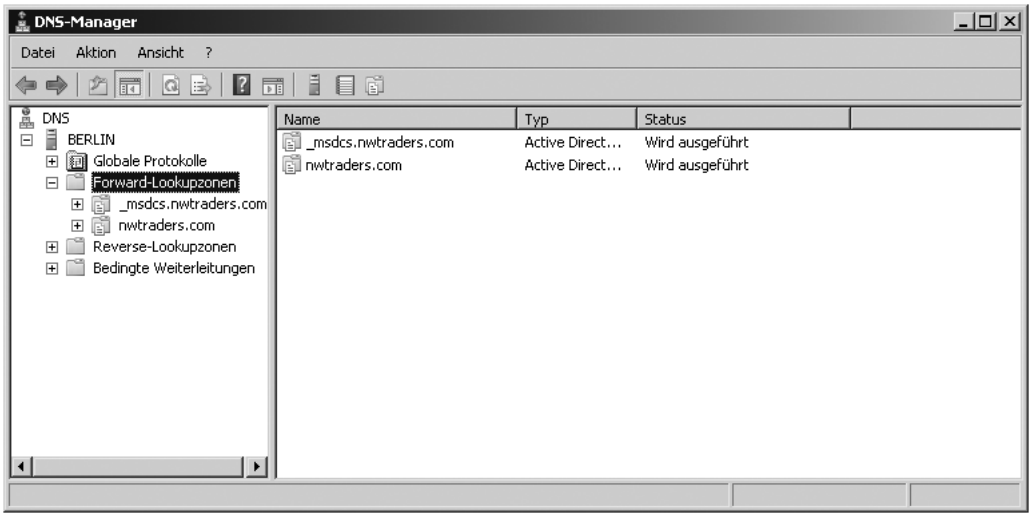


Abbildung 18.5 Die DNS-Zoneninformationen im DNS-Manager

Problembehandlung Die Installation von Active Directory kann aufgrund von DNS-Problemen fehlschlagen. DNS-Probleme treten gewöhnlich auf, wenn der einzige Domänencontroller auch als DNS-Server eingesetzt wird. Die DNS-Zonen werden unter Umständen nicht richtig erstellt, wenn der DNS-Dienst nicht schnell genug gestartet werden kann. Um dieses Problem zu vermeiden, beenden Sie den Serverdienst und starten Sie diesen Dienst nach einigen Sekunden neu. Der Neustart erzwingt das Erstellen der DNS-Zonen.

Das Skript *ReportDNSZoneConfig.ps1* beginnt mit einer einfachen *param*-Anweisung. Die *param*-Anweisung ermöglicht die Abfrage eines Remotecomputers und das Anzeigen von Hilfeinformationen. Die *param*-Anweisung lautet:

```
param($computer="localhost", [switch]$help)
```

Die nächste Funktion ist *funhelp()*. Deklarieren Sie als Erstes die Variable *$helptext* und weisen Sie dieser eine *here*-Zeichenfolge zu. Geben Sie in der *here*-Zeichenfolge eine Beschreibung des Skripts, die Parameter und einige Syntaxbeispiele an. Nachdem Sie die *here*-Zeichenfolge erstellt haben, geben Sie den Inhalt der Variablen *$helptext* aus und beenden Sie das Skript. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ReportDNSZoneConfig.ps1
```

```
Erstellt eine Liste der Konfigurationsinformationen für DNS-Zonen
auf dem lokalen Computer oder auf einem Remotecomputer.

PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
ReportDNSZoneConfig.ps1

Erstellt eine Liste der Konfigurationsinformationen für DNS-Zonen
auf dem lokalen Computer.

SetDNSServerConfig.ps1 -computer MunichServer

Erstellt eine Liste der Konfigurationsinformationen für DNS-Zonen
auf einem Remotecomputer namens MunichServer.

ReportDNSZoneConfig.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Überprüfen Sie anschließend, ob die Variable *$help* vorhanden ist. Wird die Variable *$help* gefunden, wurde das Skript mit dem Parameter **-help** ausgeführt und Sie müssen die Funktion *funhelp()* aufrufen. Codezeile:

```
if($help)   {   "Ausgeben der Hilfeinformationen ..."   ;   funHelp   }
```

Greifen Sie im nächsten Abschnitt des Skripts mit dem Cmdlet **Get-WmiObject** auf WMI zu. Verwenden Sie die WMI-Klasse *MicrosoftDNS_ZONE* und rufen Sie alle Eigenschaften ab, die mit den Buchstaben *a* bis *z* beginnen. Codeabschnitt:

```
Get-WmiObject -Class MicrosoftDNS_ZONE -computer $computer `
 -Namespace root\microsoftDNS |
 format-list [a-z]*
```

Das vollständige Skript *ReportDNSZoneConfig.ps1* hat folgenden Inhalt:

ReportDNSZoneConfig.ps1

```
param($computer="localhost", [switch]$help)
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ReportDNSZoneConfig.ps1
Erstellt eine Liste der Konfigurationsinformationen für DNS-Zonen
auf dem lokalen Computer oder auf einem Remotecomputer.

PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
```

```
ReportDNSZoneConfig.ps1

Erstellt eine Liste der Konfigurationsinformationen für DNS-Zonen
auf dem lokalen Computer.

SetDNSServerConfig.ps1 -computer MunichServer

Erstellt eine Liste der Konfigurationsinformationen für DNS-Zonen
auf einem Remotecomputer namens MunichServer.

ReportDNSZoneConfig.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}

if($help)   { "Ausgeben der Hilfeinformationen ..." ; funHelp }

Get-WmiObject -Class MicrosoftDNS_ZONE -computer $computer `
-Namespace root\microsoftDNS |
 format-list [a-z]*
```

Erstellen von DNS-Zonen

Nachdem Sie die vorhandenen DNS-Zonen überprüft haben, können Sie weitere DNS-Zonen erstellen. Sie können die Zonen mit dem DNS-Manager oder mit Windows PowerShell anlegen. Wenn Sie den DNS-Manager verwenden, wird der in Abbildung 18.6 dargestellte Assistent angezeigt.

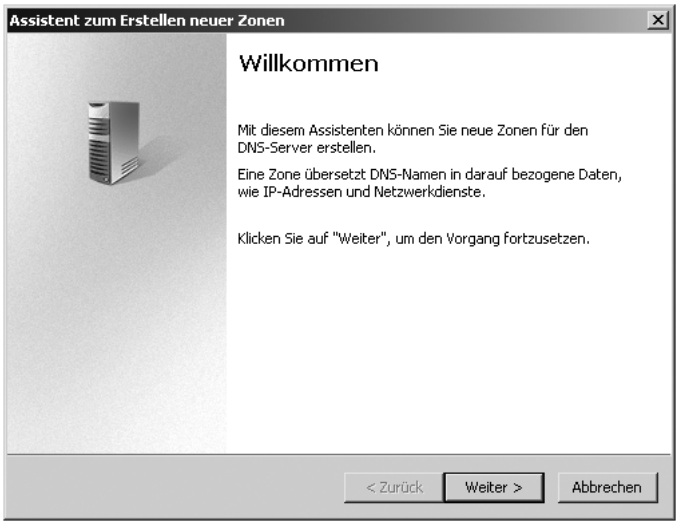


Abbildung 18.6 Neue DNS-Zonen können mit einem Assistenten schnell erstellt werden

Das Erstellen einer DNS-Zone mit Windows PowerShell ist im Skript *CreateDNSZone.ps1* veranschaulicht.

Beginnen Sie das Skript *CreateDNSZone.ps1* mit einer *param*-Anweisung. Diese *param*-Anweisung ist relativ kompliziert, da eine DNS-Zone mit mehreren Methoden in einer Windows-Umgebung angelegt werden kann. Definieren Sie nach der *param*-Anweisung den Parameter **-computer**. Geben Sie mit dem Parameter **-computer** den Namen des Zielcomputers an. Wenn die DNS-Zone eine integrierte Active Directory-Zone ist, kann es sich beim Zielserver um einen beliebigen DNS-Server handeln. Standardmäßig wird der lokale Computer verwendet. Der nächste Parameter ist **-zonename**. Dieser Parameter gibt den Namen der neuen DNS-Zone an. Der dritte Parameter ist **-action**, um einen Wert für die auszuführende Aktion angeben zu können. Die unterstützten Aktionen werden erläutert, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Die nächsten beiden Parameter, **-datafile** und **-ipadd**, sind auf *$null* festgelegt, um optionale Parameter zuzulassen. Der letzte Parameter ist **-help**, um die Hilfe anzuzeigen. Codeabschnitt:

```
Param(
      $computer="localhost",$ZoneName,
      $action, Datafile=$null,
      $IPaddr=$null,[switch]$help
      )
```

Die nächste Funktion ist *funhelp()*, um eine Hilfemeldung anzuzeigen, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Sie können den Hilfetext im Voraus erstellen und auf Anforderung anzeigen. Die Funktion *funhelp()* verwendet eine *here*-Zeichenfolge mit einer Beschreibung des Skripts, den Parametern und Syntaxbeispielen. Die *here*-Zeichenfolge ist der Variablen *$helptext* zugewiesen. Nachdem Sie die *here*-Zeichenfolge definiert haben, zeigen Sie den Inhalt der Variablen *$helptext* an und beenden das Skript. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: CreateDNSZone.ps1
Erstellt eine DNS-Zone auf dem lokalen Computer oder auf einem Remotecomputer.

PARAMETER:
-computer  Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-action    Gibt den Typ der zu konfigurierenden DNS-Zone an:
           adp Active Directory-integrierte primäre Zone: -zoneName
           ads Active Directory-integrierte sekundäre Zone: -zonename -ipaddr
           adst Active Directory-integrierte Stubb-Zone: -zonename
           nadp Primäre, nicht Active Directory-integrierte Zone: -zonename -datafile
           nads Sekundäre, nicht Active Directory-integrierte Zone: -zonename -datafile -ipaddr
           nadst Nicht Active Directory-integrierte Stubbzone: -zonename -datafile
-zoneName  Gibt den Namen der zu konfigurierenden DNS-Zone an.
-datafile  Muss beim Erstellen von  nicht  Active Directory-integrierten DNS-Zonen angegeben werden.
-IPaddr    Muss beim Erstellen von  sekundären DNS-Zonen angegeben werden.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
CreateDNSZone.ps1 -action adp -zonename vienna

Erstellt auf dem lokalen Computer
eine Active Directory-integrierte primäre Zone namens vienna.
```

```
CreateDNSZone.ps1 -action ads -zonename vienna -ipaddr
"192.168.3.100"

Erstellt auf dem lokalen Computer eine Active Directory-integrierte sekundäre Zone namens vienna
mit der IP-Adresse 192.168.3.100 für die Masterzone.

CreateDNSZone.ps1 -computer MunichServer -action nadp -zonename
Vienna -datafile c:\windows\system32\dns\vienna.dns

Erstellt auf einem Remotecomputer namens MunichServer eine primäre,
nicht Active Directory-integrierte DNS-Zone namens vienna,
die die DNS-Zonendatei C:\Windows\System32\DNS\vienna.dns verwendet.

CreateDNSZone.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Im Anschluss an die Funktion *funhelp()* folgt die Überprüfungsphase des Skripts. Die entsprechenden Codezeilen werden beim Starten des Skripts unmittelbar ausgeführt. Nach der *param*-Anweisung überspringt die Codeausführung die anderen Funktionen und fährt mit den folgenden beiden Zeilen fort. Überprüfen Sie zuerst, ob die Variable *$help* existiert. Wenn die Variable gefunden wird, wurde das Skript mit dem Parameter **-help** ausgeführt und Sie müssen die Funktion *funhelp()* aufrufen, um die Hilfe auszugeben. Sollte die Variable *$zonename* oder die Variable *$action* nicht vorhanden sein, geben Sie ebenfalls eine Statusmeldung aus und rufen die Funktion *funhelp()* auf. Die beiden Codezeilen lauten:

```
if($help)   {   "Ausgeben der Hilfeinformationen ..."   ;   funHelp   }
if(!$zonename -or !$action) { "Fehlende Parameter ..." ; funHelp}
```

Als Nächstes müssen Sie die Variablen initialisieren. Definieren Sie die Variable *$adintegrated* mit dem booleschen Wert *True*. Deklarieren Sie anschließend die boolesche Variable *$nonadintegrated*. Diese Variable ist auf *False* festgelegt. Definieren Sie die Variable *$primary* als ganze Zahl und legen Sie den Wert auf *0* fest. Setzen Sie die Variable *$secondary* auf *1*, die Variable *$stuby* auf *2* und die Variable *$forwarder* auf *3*. Die IP-Adresse wird als ein Array angegeben. Die Werte für diesen Skriptabschnitt sind im Windows Software Development Kit (SDK) dokumentiert. Codeabschnitt:

```
[bool]$adintegrated = -1
[bool]$nonadintegrated = 0
[int32]$Primary = 0
$secondary = 1
$stuby = 2
$forwarder = 3
[array]$aryIP = $IPaddr
```

Stellen Sie nun eine Verbindung mit WMI her. Diese Verbindung unterscheidet sich von der mit dem Cmdlet **Get-WmiObject** hergestellten Verbindung, da eine bestimmte Instanz der WMI-Klasse abgerufen wird. Verwenden Sie den *[wmiclass]*-Accelerator und greifen Sie direkt auf die WMI-Klasse *MicrosoftDNS_ZONE* zu. Speichern Sie das zurückgegebene Verwaltungsobjekt in der Variablen *$dnsserver*. Codeabschnitt:

```
$dnsServer = [wmiclass]"\\$computer\root\microsoftDNS:MicrosoftDNS_ZONE"
```

Nachdem Sie die Verbindung mit der WMI-Klasse *MicrosoftDNS_ZONE* hergestellt haben, müssen Sie die Aktion auswerten, die zur Laufzeit vom Parameter **-action** angegeben wurde. Jede Bedingung ruft die gleiche *CreateZone()*-Methode auf, jedoch mit unterschiedlichen Parametern. Nach dem Erstellen der Zone wird das Skript beendet. Die *switch*-Anweisung ruft standardmäßig die Funktion *funhelp()* auf und gibt eine Hilfemeldung aus. Codeabschnitt:

```
Switch($action)
 {
  "adp"    {
             $dnsServer.createZone($ZoneName, $primary, $adintegrated) ;
             exit
           }
  "ads"    {
             $dnsServer.createZone($ZoneName, $secondary, $adintegrated,
             $null, $aryIP) ; exit
           }
  "adst"   { $dnsServer.createZone($ZoneName, $stuby, $adintegrated) }
  "nadp"   {
             $dnsServer.createZone($ZoneName, $primary, $nonadintegrated,
             $Datafile) ; exit
           }
  "nads"   {
             $dnsServer.createZone($ZoneName, $secondary, $nonadintegrated,
 $Datafile, $aryIP) ; exit
           }
  "nadst"  {
              $dnsServer.createZone($ZoneName, $stuby, $nonadintegrated,
              $Datafile) ; exit
           }
  DEFAULT  {
            "Es wurde keine gültige Aktion angegeben. Ausgeben der Hilfeinformationen ..." ;
            $funHelp
           }
 }
```

Bei Angabe eines unbekannten Parameters wird der Standardblock der *switch*-Anweisung ausgeführt, oder die *switch*-Anweisung wird übersprungen. Rufen Sie deshalb nach der *switch*-Anweisung die Funktion *funhelp()* mit folgender Statusmeldung auf:

```
"Es wurde keine gültige Aktion angegeben. Ausgeben der Hilfeinformationen ... ; $funhelp "
```

Das vollständige Skript *CreateDNSZone.ps1* hat folgenden Inhalt:

CreateDNSZone.ps1

```
Param(
      $computer="localhost",$ZoneName,
      $action,$Datafile=$null,
      $IPAddr=$null,[switch]$help
      )

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: CreateDNSZone.ps1
Erstellt eine DNS-Zone auf dem lokalen Computer oder auf einem Remotecomputer.
```

```
PARAMETER:
-computer  Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-action    Gibt den Typ der zu konfigurierenden DNS-Zone an:
           adp Active Directory-integrierte primäre Zone: -zoneName
           ads Active Directory-integrierte sekundäre Zone: -zonename -ipaddr
           adst Active Directory-integrierte Stubb-Zone: -zonename
           nadp Primäre, nicht Active Directory-integrierte Zone: -zonename -datafile
           nads Sekundäre, nicht Active Directory-integrierte Zone: -zonename -datafile -ipaddr
           nadst Nicht Active Directory-integrierte Stubbzone: -zonename -datafile
-zoneName  Gibt den Namen der zu konfigurierenden DNS-Zone an.
-datafile  Muss beim Erstellen von  nicht  Active Directory-integrierten DNS-Zonen angegeben werden.
-IPaddr    Muss beim Erstellen von  sekundären DNS-Zonen angegeben werden.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
CreateDNSZone.ps1 -action adp -zonename vienna

Erstellt auf dem lokalen Computer
eine Active Directory-integrierte primäre Zone namens vienna.

CreateDNSZone.ps1 -action ads -zonename vienna -ipaddr
"192.168.3.100"

Erstellt auf dem lokalen Computer eine Active Directory-integrierte sekundäre Zone namens vienna
mit der IP-Adresse 192.168.3.100 für die Masterzone.

CreateDNSZone.ps1 -computer MunichServer -action nadp -zonename
Vienna -datafile c:\windows\system32\dns\vienna.dns

Erstellt auf einem Remotecomputer namens MunichServer eine primäre,
nicht Active Directory-integrierte DNS-Zone namens vienna,
die die DNS-Zonendatei C:\Windows\System32\DNS\vienna.dns verwendet.

CreateDNSZone.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}

if($help)   { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if(!$zonename -or !$action) { "Fehlende Parameter ..." ; funHelp}

[bool]$adintegrated = -1
$nonadintegrated = 0
[int32]$Primary = 0
$secondary = 1
$stuby = 2
$forwarder = 3
[array]$aryIP = $IPaddr
$dnsServer = [wmiclass]"\\$computer\root\microsoftDNS:MicrosoftDNS_ZONE"
Switch($action)
 {
  "adp"    {
```

```
            $dnsServer.createZone($ZoneName, $primary, $adintegrated) ;
            exit
          }
  "ads"   {
            $dnsServer.createZone($ZoneName, $secondary, $adintegrated,
            $null, $aryIP) ; exit
          }
  "adst"  { $dnsServer.createZone($ZoneName, $stuby, $adintegrated) }
  "nadp"  {
            $dnsServer.createZone($ZoneName, $primary, $nonadintegrated,
            $Datafile) ; exit
          }
  "nads"  {
            $dnsServer.createZone($ZoneName, $secondary, $nonadintegrated,
$Datafile, $aryIP) ; exit
          }
  "nadst" {
             $dnsServer.createZone($ZoneName, $stuby, $nonadintegrated,
             $Datafile) ; exit
          }
  DEFAULT {
           "Es wurde keine gültige Aktion angegeben. Ausgeben der Hilfeinformationen  ..." ;
           $funHelp
          }
}

"Es wurde keine gültige Aktion angegeben. Ausgeben der Hilfeinformationen ... ; $funhelp "
```

Verwalten von WINS und DHCP

Für viele Netzwerkadministratoren ist WINS (Windows Internet Naming Service) weiterhin wichtig. Viele Netzwerkadministratoren würden den Dienst gerne aus der Netzwerkinfrastruktur entfernen, fürchten aber unvorhersehbare Konsequenzen. Tatsächlich haben einige Netzwerkadministratoren festgestellt, dass sie ohne den WINS-Dienst zurechtkommen. Durch Entfernen des WINS-Dienstes können Sie den zum Sichern, Wiederherstellen und Warten des Dienstes erforderlichen Aufwand eliminieren. WINS war vor 25 Jahren bestens geeignet, als Netzwerke 20 bis 30 Computer umfassten, auf denen das NetBEUI-Protokoll ausgeführt wurde. Mit DNS, IPv4 und IPv6, die heute verfügbar sind, ist WINS für die meisten modernen Anwendungen nicht erforderlich. WINS wird nur benötigt, um veraltete Anwendungen zu unterstützen, die noch benötigt werden. Da WINS bald vollständig abgelöst sein wird, sind keine Updates für die Verwaltungs- und Automatisierungsfunktionen verfügbar. Die einzige Aktion, die Sie deshalb ausführen können, ist das Abrufen der Serverkonfigurationsinformationen.

Der DHCP-Dienst (Dynamic Host Configuration Protocol) ist relativ wartungsfrei. Wenn der Dienst erst einmal ordnungsgemäß konfiguriert ist, sind nur noch wenige Verwaltungsaufgaben auszuführen. Die an DHCP für Windows Server 2008 vorgenommenen Verbesserungen betreffen unter anderem ein Sorgenkind von Netzwerkadministratoren – das Verhindern von Adressbereichsüberlastungen mittels reduzierter Leasezeiten für Drahtloszugriffspunkte.

Mit dem Skript *ManageWinsDHCP.ps1* können Sie die Konfigurationsinformationen von WINS- und DHCP-Servern abrufen. Außerdem können Sie einen DHCP-Server in Active Directory aktivieren oder deaktivieren. Diese beiden Aufgaben fallen üblicherweise in den Zuständigkeitsbereich der Administratoren von Windows Server 2008-Netzwerken.

Das Skript *ManageWinsDHCP.ps1* beginnt mit einer *param*-Anweisung. Die Definition von Parametern ermöglicht das Steuern der Skriptausführung, ohne das Skript bei der Ausführung direkt bearbeiten zu müssen. Die folgenden vier Parameter werden definiert: **-computer**, **-ip**, **-action** und **-help**. Der Parameter **-computer** legt den Computer fest, für den das Skript ausgeführt wird. Der Parameter **-ip** wird nur verwendet, um einen DHCP-Server in Active Directory zu aktivieren oder zu deaktivieren. Der Parameter **-action** gibt die auszuführende Aktion an. Der Parameter **-help** hat einen festen Wert und bewirkt die Anzeige der Hilfeinformationen. Die *param*-Anweisung lautet:

```
param($computer, $ip, $action, [switch]$help)
```

Die nächste Funktion ist *funhelp()*, um eine Hilfemeldung anzuzeigen, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Die Funktion *funhelp()* besteht aus einer *here*-Zeichenfolge, die der Variablen *$helptext* zugewiesen ist. Nachdem Sie die *here*-Zeichenfolge der Variablen *$helptext* definiert haben, zeigen Sie den Inhalt der Variablen an und beenden Sie das Skript. Die *here*-Zeichenfolge besteht aus drei Abschnitten: Der erste Abschnitt enthält eine Beschreibung der Skriptfunktionalität. Im zweiten Abschnitt sind die Befehlszeilenparameter aufgelistet und der dritte Abschnitt enthält Syntaxbeispiele. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ManageWinsDHCP.ps1
Verwaltet DHCP- und WINS-Server auf dem lokalen Computer oder auf einem Remotecomputer.

PARAMETER:
-computer Der Computer, für den das Skript ausgeführt werden soll.
-ip       Die IP-Adresse des Servers, für den das Skript ausgeführt werden soll.
-action   Specifies action to perform < shoWins, shoDHCP,
          shoAllDHCP, addDHCP, deleteDHCP >
-help     Zeigt dieses Hilfethema an.

SYNTAX:
ManageWinsDHCP.ps1

Zeigt eine Meldung an,  die besagt, dass eine Aktion angegeben werden muss, und gibt die Hilfeinformationen aus.

ManageWinsDHCP.ps1 -computer MunichServer -action shoWins

Zeigt die Konfigurationsinformationen des WINS-Dienstes
auf einem Remoteserver namens MunichServer an.

ManageWinsDHCP.ps1 -computer MunichServer -action shoDHCP

Zeigt die Konfigurationsinformationen des DHCP-Dienstes
auf einem Remoteserver namens MunichServer an.

ManageWinsDHCP.ps1 -action shoAllDHCP

Listet alle in Active Directory autorisierten DHCP-Server auf.
```

```
ManageWinsDHCP.ps1 -action addDHCP -computer berlin -ip 192.168.1.1

Fügt einen DHCP-Server namens berlin mit der IP-Adresse 192.168.1.1
als autorisierten DHCP-Server zu Active Directory hinzu.

ManageWinsDHCP.ps1 -action deleteDHCP -computer berlin -ip 192.168.1.1

Entfernt einen zuvor autorisierten DHCP-Server namens berlin
mit der IP-Adresse 192.168.1.1 aus Active Directory.

ManageWinsDHCP.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

Die Befehlszeilenparameter werden mit drei Codezeilen überprüft. Wenn die Variable *$help* vorhanden ist, rufen Sie die Funktion *funhelp()* auf. Sollte die Variable *$action* nicht vorhanden sein, wurde beim Ausführen des Skripts keine Aktion angegeben. Da Sie keine Standardaktion definiert haben, geben Sie in diesem Fall mit dem Cmdlet **Write-Error** eine Fehlermeldung aus und rufen Sie die Funktion *funhelp()* auf. Wenn der Parameter **-computer** nicht angegeben wurde, ist die Variable *$computer* nicht vorhanden. Geben Sie in diesem Fall eine Meldung aus, dass das Skript lokal ausgeführt wird. Die drei erforderlichen Codezeilen lauten:

```
if($help)       {       "Ausgeben der Hilfeinformationen ..."       ;       funHelp       }
if(!$action)    {
                  Write-error "Es muss eine Aktion angegeben werden ..." ;
                  funHelp
                }
if(!$computer)  { Write-warning "Verwenden des Standardservers ..." }
```

Definieren Sie anschließend eine *switch*-Anweisung. Die *switch*-Anweisung legt das Skriptverhalten fest und analysiert die Befehlszeilenargumente. Das einzige Befehlszeilenargument, das in dieser *switch*-Anweisung ausgewertet wird, entspricht dem Parameter **-action**. Wird das Skript mit dem Parameter **-action shoWins** ausgeführt, geben Sie die WINS-Serverkonfiguration aus. Wurde das Skript hingegen mit dem Parameter **-action shoDHCP** gestartet, geben Sie die Informationen über den aktuellen DHCP-Server aus.

Sie können einen DHCP-Server in Active Directory aktivieren, indem Sie das Skript mit dem Parameter **-action addDHCP** ausführen. Dieser Befehl benötigt den DNS-Namen des Servers und die IP-Adresse. Um sicherzustellen, dass die erforderlichen Parameter angegeben sind, müssen Sie mit einer *if*-Anweisung die Variablen *$computer* und *$ip* analysieren. Sind die Variablen nicht vorhanden, geben Sie eine Meldung aus, dass beide Parameter erforderlich sind. Rufen Sie anschließend die Funktion *funhelp()* auf. Wurden die Parameter **-computer** und **-ip** hingegen angegeben, rufen Sie den Befehl **netsh** auf und aktivieren Sie den DHCP-Server. Codeabschnitt:

```
"addDHCP"    {
               if(!$computer -or !$ip)
                    { "Sowohl der Computername " +
                      "als auch die IP-Adresse müssen angegeben werden ..." ;
                      funHelp
                    }
               netsh dhcp add server $computer $ip
             }
```

Um einen DHCP-Server aus Active Directory zu entfernen, geben Sie **deleteDHCP** im Parameter **-action** an. Wie beim Hinzufügen eines DHCP-Servers sind zum Entfernen eines DHCP-Servers zwei Werte erforderlich. Sie müssen die IP-Adresse und den DNS-Hostnamen des DHCP-Servers angeben. Stellen Sie mit einer *if*-Anweisung sicher, dass die Variablen *$computer* und *$ip* vorhanden sind. Werden die Variablen nicht gefunden, geben Sie eine Meldung aus, dass der Computername und die IP-Adresse erforderlich sind. Zeigen Sie anschließend die Hilfe an. Sind die beiden erforderlichen Parameter hingegen vorhanden, rufen Sie den entsprechenden **netsh** Befehl auf. Codeabschnitt:

```
"deleteDHCP" {
               if(!$computer -or !$ip)
                    { "Sowohl der Computername " +
                      "als auch die IP-Adresse müssen angegeben werden ..." ;
                      funHelp
                    }
               netsh dhcp delete server $computer $ip
             }
```

Die vollständige *switch*-Anweisung lautet:

```
switch($action)
{
 "shoWins"    { netsh wins dump $computer }
 "shoDHCP"    { netsh dhcp show server $computer }
 "shoAllDHCP" { netsh dhcp show server }
 "addDHCP"    {
                if(!$computer -or !$ip)
                     { "Sowohl der Computername " +
                       "als auch die IP-Adresse müssen angegeben werden ..." ;
                       funHelp
                     }
                netsh dhcp add server $computer $ip
               }
 "deleteDHCP" {
                if(!$computer -or !$ip)
                     { "Sowohl der Computername " +
                       "als auch die IP-Adresse müssen angegeben werden .." ;
                       funHelp
                     }
                netsh dhcp delete server $computer $ip
              }
 }
```

Das vollständige Skript *ManageWinsDHCP.ps1* hat folgenden Inhalt:

ManageWinsDHCP.ps1

```
param($computer, $ip, $action, [switch]$help)

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: ManageWinsDHCP.ps1
Verwaltet DHCP- und WINS-Server auf dem lokalen Computer oder auf einem Remotecomputer.

PARAMETER:
-computer Der Computer, für den das Skript ausgeführt werden soll.
-ip       Die IP-Adresse des Servers, für den das Skript ausgeführt werden soll.
-action   Specifies action to perform < shoWins, shoDHCP,
          shoAllDHCP, addDHCP, deleteDHCP >
-help     Zeigt dieses Hilfethema an.

SYNTAX:
ManageWinsDHCP.ps1

Zeigt eine Meldung an,  die besagt, dass eine Aktion angegeben werden muss, und gibt die Hilfeinformationen aus.

ManageWinsDHCP.ps1 -computer MunichServer -action shoWins

Zeigt die Konfigurationsinformationen des WINS-Dienstes
auf einem Remoteserver namens MunichServer an.

ManageWinsDHCP.ps1 -computer MunichServer -action shoDHCP

Zeigt die Konfigurationsinformationen des DHCP-Dienstes
auf einem Remoteserver namens MunichServer an.

ManageWinsDHCP.ps1 -action shoAllDHCP

Listet alle in Active Directory autorisierten DHCP-Server auf.

ManageWinsDHCP.ps1 -action addDHCP -computer berlin -ip 192.168.1.1

Fügt einen DHCP-Server namens berlin mit der IP-Adresse 192.168.1.1
als autorisierten DHCP-Server zu Active Directory hinzu.

ManageWinsDHCP.ps1 -action deleteDHCP -computer berlin -ip 192.168.1.1

Entfernt einen zuvor autorisierten DHCP-Server namens berlin
mit der IP-Adresse 192.168.1.1 aus Active Directory.

ManageWinsDHCP.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
 $helpText
 exit
}
```

```
if($help)    { "Ausgeben der Hilfeinformationen ..." ; funHelp }
if(!$action) { "Es muss eine Aktion angegeben werden  ..." ; funHelp }
if(!$computer)  { "Verwenden des Standardservers ..." }


switch($action)
{
 "shoWins"    { netsh wins dump $computer }
 "shoDHCP"    { netsh dhcp show server $computer }
 "shoAllDHCP" { netsh dhcp show server }
 "addDHCP"    {
                if(!$computer -or !$ip)
                       { "Sowohl der Computername " +
                         "als auch die IP-Adresse müssen angegeben werden ..." ;
                         funHelp
                       }
                netsh dhcp add server $computer $ip
               }
 "deleteDHCP" {
                if(!$computer -or !$ip)
                       { "Sowohl der Computername " +
                         "als auch die IP-Adresse müssen angegeben werden ..." ;
                         funHelp
                       }
                netsh dhcp delete server $computer $ip
               }
 }
```

Zusammenfassung

In diesem Kapitel wurde die Konfiguration von Netzwerkdiensten, insbesondere DNS, DHCP und WINS, beschrieben. Zuerst wurde die Installation von DNS auf einem Windows Server 2008-Computer erklärt. Sie haben DNS-Zonen und Einträge erstellt sowie die aktuelle Konfiguration überprüft. Anschließend haben Sie die Konfiguration der WINS-Datenbank ausgegeben. Das Kapitel wurde mit der Verwaltung von DHCP abgeschlossen. Da WINS und DHCP häufig auf dem gleichen Server installiert sind und noch als gleichwertige Dienste betrachtet werden, haben Sie ein Skript erstellt, mit dem beide Dienste verwaltet werden können.

KAPITEL 19

Arbeiten mit der Windows Server 2008 Server Core

Nach Abschluss dieses Kapitels können Sie:

- Einer Domäne beitreten
- Eine statische IP-Adresse festlegen
- Die DNS-Einstellungen konfigurieren
- Den Server benennen und neu starten

Auf der Begleit-CD Alle Skripts in diesem Kapitel befinden sich auf der Begleit-CD im Ordner *\Scripts\Chapter19*.

Seit der ersten Version umfasst Windows eine grafische Benutzeroberfläche für den Zugriff auf Programme und Anwendungen. Nun ist eine Windows-Version ohne grafische Benutzeroberfläche verfügbar. Windows Server 2008 Server Core unterstützt zahlreiche Funktionen, die jeweils besondere Anforderungen an die Installation und Überwachung stellen. In diesem Kapitel sind die manchmal verwirrenden und zeitraubenden Aufgaben beim Konfigurieren von Windows Server 2008 Server Core beschrieben. Beinahe alle Funktionen von Windows Server 2008 Server Core erfordern die gleichen Konfigurationsschritte.

Ursprüngliche Konfiguration

Die folgenden beiden Konfigurationsschritte können für Windows Server 2008 Server Core nicht remote über Windows PowerShell ausgeführt werden: Das Aktivieren der Remoteverwaltung über die Windows Firewall und das Abrufen der IP-Adresse des Servers.

Um die Remoteverwaltung für Windows Server 2008 Server Core zu aktivieren, verwenden Sie das Dienstprogramm *Netsh.exe*. Obwohl Sie den Befehl **netsh** auch remote ausführen können, wird der Befehl durch die Firewall blockiert. Sie müssen den folgenden Befehl deshalb lokal ausführen:

```
netsh firewall set service remoteadmin enable
```

Anschließend können Sie Windows PowerShell zum Verwalten des Servers verwenden.

Hinweis Zum Verbinden und Verwalten von Remoteservern sind außer dem Aktivieren der Remoteverwaltung möglicherweise weitere Konfigurationseinstellungen erforderlich. Wenn der Remoteserver nicht in eine Domäne aufgenommen wurde, müssen Sie einige WMI-Aktionen explizit konfigurieren. Weitere Informationen finden Sie auf der MSDN-Website (Microsoft Developer Network).

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Um die Verbindung mit einem Server herzustellen, müssen Sie die IP-Adresse ermitteln, die dem Server während der Installation des Betriebssystems zugewiesen wurde. Führen Sie hierzu folgenden Befehl aus:

```
IPconfig / all
```

Nachdem Sie die IP-Adresse abgerufen und die Firewall konfiguriert haben, können Sie den Server mit Windows PowerShell verwalten. Um den Server zu konfigurieren, verwenden Sie die IP-Adresse, statt des Computernamens.

Aufnehmen eines Computers in eine Domäne

Eine der ersten Aufgaben, die Sie unter Umständen ausführen müssen, ist das Aufnehmen eines Servers in eine Domäne. Dies ist zur Gewährleistung eines einheitlichen Sicherheitskontextes erforderlich und vereinfacht das Abrufen von Informationen mit WMI sowie das Verwalten des Servers. Um einen Windows Server 2008 Server Core-Computer in eine Domäne aufzunehmen, können Sie das Skript *JoinDomain.ps1* verwenden.

Vorsicht Beachten Sie, dass die Firewallrichtlinien möglicherweise beim Aufnehmen eines Computers in eine Domäne geändert werden. Wenn die Standarddomänenrichtlinie die Firewallrichtlinien sperrt, kann die Remoteverbindung mit dem Server möglicherweise nicht mehr hergestellt werden.

Beginnen Sie das Skript *SetIP.ps1* mit einer *param*-Anweisung und legen Sie mehrere Befehlszeilenparameter fest. Der erste Parameter ist **-computer**, um den Zielcomputer anzugeben. Legen Sie den Parameter **-computer** auf den Standardwert *localhost* fest. Der nächste Parameter ist **-domainname**, um die Domäne anzugeben, in die der Computer aufgenommen wird. Geben Sie als Nächstes die Parameter **-username** und **-password** mit den Anmeldeinformationen für den Computer an.

Die nächsten drei Parameter haben feste Werte und müssen in der Befehlszeile angegeben werden. Der erste Parameter ist -**unjoin**, um den Computer aus einer Domäne zu entfernen. Der zweite Parameter ist **-reboot**, der zusammen mit den Parametern **-unjoin** und **-domainname** eingesetzt werden kann, um einen Remotecomputer neu zu starten. Der dritte Parameter ist **-help**, um die Hilfe anzuzeigen. Die *param*-Anweisung lautet:

```
param(
      $computer="localhost",
      $domainName,
      $username,
      $password,
      [switch]$unjoin,
      [switch]$reboot,
      [switch]$help
     )
```

Sie müssen die Funktion *funhelp()* erstellen. Diese Funktion ist aufgrund der zahlreichen vom Skript unterstützten Parameter wichtig, insbesondere da die Parameter in der richtigen Kombination angegeben werden müssen. Erstellen Sie als Erstes die Variable *$helptext*. Weisen Sie der Variablen *$helptext* eine *here*-Zeichenfolge zu, die den Hilfetext definiert. Der Hilfetext ist in die Abschnitte Beschreibung, Parameter und Syntax aufgeteilt. Nachdem Sie die *here*-Zeichenfolge definiert und der Variablen *$helptext* zugewiesen haben, zeigen Sie den Inhalt der Variablen an und beenden Sie das Skript. Die vollständige Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: JoinDomain.ps1
Fügt einen Computer zu einer Domäne hinzu.

PARAMETER:
-computer   Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-domainName Der Name der Domäne oder Arbeitsgruppe.
-username   Die Benutzerinformationen.
-password   Das Kennwort des Benutzers.
-unjoin     Entfernt den Computer aus der Domäne oder Arbeitsgruppe.
-reboot     Startet den Computer neu.
-help       Zeigt dieses Hilfethema an.

SYNTAX:
JoinDomain.ps1
Zeigt eine Meldung an, die Sie darauf hinweist, dass Sie eine Aktion angeben müssen,
und gibt die Hilfeinformationen aus.

JoinDomain.ps1 -reboot
Startet den lokalen Computer neu.

JoinDomain.ps1 -reboot -computer MunichServer
Startet einen Remotecomputer namens MunichServer neu. MunichServer muss Mitglied einer
Domäne sein, damit dieser Befehl ausgeführt werden kann.

JoinDomain.ps1 -computer MunichServer -domainName nwtraders.com `
-username nwtraders\administrator -password Password1

Fügt einen Remotecomputer namens MunichServer zur Domäne nwtraders.com unter
Verwendung des Benutzerkontos nwtraders\administrator mit dem Kennwort Password1 hinzu.
Sobald der Computer in die Domäne aufgenommen wurde, wird dieser neu gestartet. Das Computerkonto
wird im Standardcontainer angelegt. Standardmäßig ist dies der Container Computers.

JoinDomain.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Um das Skript zu vereinfachen, implementieren Sie die grundlegenden Verarbeitungsschritte als Funktionen. Die erste dieser Funktionen ist *funreboot()*, die mit der *function*-Anweisung und dem Namen der Funktion beginnt. Geben Sie im Codeblock eine *if*-Anweisung an, um zu überprüfen, ob mit der Funktion *funreboot()* der lokale Computer oder ein Remotecomputer neu gestartet werden soll.

Hinweis Sie müssen überprüfen, ob alternative Anmeldeinformationen verwendet werden müssen, da diese für lokale Verbindungen nicht zulässig sind. Auch wenn die Anmeldeinformationen mit denen des angemeldeten Benutzers identisch sind, wird beim Herstellen einer lokalen Verbindung ein Fehler verursacht.

Wenn der Name in der Variablen *$computer* nicht *localhost* ist, verwenden Sie den Benutzernamen aus der Variablen *$username* (falls diese vorhanden ist). Gibt es die Variable *$username*, geben Sie diese im Parameter **-credential** des Cmdlets **Get-WmiObject** an. Geben Sie für das Cmdlet **Get-WmiObject** den Wert in der Variablen *$computer* im Parameter **-computername** und den Wert in der Variablen *$username* im Parameter **-credential** an. Das Skript zeigt ein Dialogfeld an und fordert zur Eingabe des Kennworts an. Sie müssen die Berechtigung zum Herunterfahren aktivieren, um den Neustart ausführen zu können. Verwenden Sie hierzu die Eigenschaft *EnablePrivileges* der Bereichsoptionen, die vom Microsoft .NET Framework-Objekt bereitgestellt werden, auf das das WMI-Objekt verweist. Legen Sie die Option auf *True* fest. Rufen Sie anschließend die Methode *Reboot()* auf. Die Funktion *funreboot()* ist wie folgt implementiert:

```
Function funReboot()
{
 if($computer -ne "localhost")
 {
  if($username)
  {
   $objWMI = Get-WmiObject -Class Win32_operatingsystem `
                -computername $computer -credential $username
   $objWMI.psbase.Scope.Options.EnablePrivileges = $true
   $objWMI.reboot()
  }
```

Wenn Sie keinen Benutzernamen angeben, ist der Code etwas einfacher. Erstellen Sie mit dem Cmdlet **Get-WmiObject** eine Instanz der WMI-Klasse *Win32_OperatingSystem*. Geben Sie den Computernamen aus der Variablen *$computer* im Parameter **-computername** des Cmdlets **Get-WmiObject** an. Aktivieren Sie die erforderlichen Berechtigungen und rufen Sie die Methode *Reboot()* auf. Codeabschnitt:

```
  ELSE
   {
    $objWMI = Get-WmiObject -Class Win32_operatingsystem `
                -computername $computer
    $objWMI.psbase.Scope.Options.EnablePrivileges = $true
    $objWMI.reboot()
   }
```

Beenden Sie das Skript, unabhängig davon, ob die Variable *$username* vorhanden ist. Wenn der Name des Computers jedoch *localhost* ist, können Sie keine alternativen Anmeldeinformationen verwenden. Erstellen Sie in diesem Fall unter Verwendung des Cmdlets **Get-WmiObject** eine Instanz der WMI-Klasse *Win32_OperatingSystem*. Die Variable *$computer* enthält den Namen *localhost*. Aktivieren Sie die erforderlichen Berechtigungen, rufen Sie die Methode *Reboot()* auf und beenden Sie das Skript. Die Funktion *reboot()* ist wie folgt implementiert:

```
  exit
 }
 ELSE
 {
  $objWMI = Get-WmiObject -Class Win32_operatingsystem `
              -computername $computer
  $objWMI.psbase.Scope.Options.EnablePrivileges = $true
  $objWMI.reboot()
  exit
 }
}
```

Der Funktion *funreboot()* folgt die Funktion *funjoindomain()*, um den Computer in eine Domäne aufzunehmen. Bevor der Computer jedoch in die Domäne aufgenommen werden kann, müssen Sie einige spezielle WMI-Aspekte für die Remoteverwaltung innerhalb einer Arbeitsgruppe berücksichtigen. Um Probleme zu vermeiden, führen Sie den Befehl **netdom** aus, um den Computer in die Domäne aufzunehmen. Da die Syntax des Befehls kompliziert und die Online-Hilfe unübersichtlich ist, sollten Sie ein Skript erstellen.

Hinweis Der Befehl **netdom** ist im Windows Administration Tools Pack enthalten, das in einigen Versionen von Windows standardmäßig vorhanden ist, aber für einige Desktopversionen explizit installiert werden muss. Sie können das Windows Administration Tools Pack von *http://www.microsoft.com/downloads/* herunterladen.

Beginnen Sie die Funktion *funjoindomain()* mit dem vollständigen Befehl, rufen Sie den Befehl **netdom** auf und geben Sie den Parameter **join** an. Übergeben Sie den Computernamen in der Variablen *$computer* und den Domänennamen im Parameter *$domainname*. Dem Parameter **domain** muss ein Schrägstrich (/) oder ein Bindestrich (-) vorangestellt werden. Der Wert ist durch einen Doppelpunkt vom Domänennamen getrennt. Der Parameter für das Domänenbenutzerkonto lautet **/userD** und für das Kennwort **/passwordD**. Diesen beiden Parametern ist ebenfalls ein Schrägstrich (**/**) oder ein Bindestrich (**-**) vorangestellt.

Nachdem der Computer in die Domäne aufgenommen wurde, müssen Sie den Computer neu starten, um die Änderungen zu übernehmen. Um den Computer neu zu starten, führen Sie den Befehl **shutdown** mit dem Parameter **/m** aus und geben Sie den Computernamen aus der Variablen *$computer* an. Dem Computernamen müssen zwei Schrägstriche (\\) vorangestellt werden. Geben Sie im Befehl **shutdown** den Parameter **/r** an, um den Computer neu zu starten. Geben Sie den Parameter **/c** mit der Zeichenfolge *"Zur Domäne hinzugefügt."* an. Sie müssen zwischen **/c** und dem Kommentar kein Trennzeichen eingeben.

Führen Sie anschließend den Befehl **netdom** mit dem Cmdlet **Invoke-Expression** aus. Halten Sie mit dem Cmdlet **Start-Sleep** die Ausführung des Skripts zwei Sekunden lang an und führen Sie dann den Befehl **shutdown** aus, wie in der Variablen *$sdcommand* definiert, um den Computer herunterzufahren. Beenden Sie danach die Skriptausführung. Die Funktion *funjoindomain()* ist wie folgt implementiert:

```
Function FunJoinDomain()
{
 $command = "netdom join $computer /domain:$domainName " + `
            "/userD:$userName /passwordD:$password"
 $sdcommand = "shutdown /m \\$computer /r " + `
              "/c" + "Zur Domäne hinzugefügt."

 invoke-expression $command
 start-sleep -seconds 2
 "We will now reboot $computer"
 Invoke-expression $sdcommand
 exit
}
```

Erstellen Sie nun die Funktion *fununjoin()*, um einen Computer aus einer Domäne entfernen zu können. Da der jeweilige Computer Mitglied einer Domäne ist, führen Sie diesen Vorgang unter Verwendung von WMI aus. Beginnen Sie die Funktion *fununjoin()*, indem Sie die Variable *$option* erstellen und dieser den Wert *0* zuweisen. Der Wert *0* entfernt das Computerkonto, wenn der Computer aus der Domäne entfernt wird. Der Wert *2* deaktiviert das Konto, dieses wird jedoch nicht gelöscht.

Erstellen Sie mit dem Cmdlet **Get-WmiObject** eine Instanz der WMI-Klasse *Win32_ComputerSystem*. Geben Sie den Namen des Computers, der in der Variablen *$computer* gespeichert ist, im Parameter **-computername** an, und weisen Sie das zurückgegebene WMI-Objekt der Variablen *$objWMI* zu. Fordern Sie die erforderlichen Berechtigungen an und rufen Sie die Methode *UnjoinDomainOr-Workgroup()* auf. Diese Methode verwendet die drei Funktionsparameter *password*, *username* und *option*, denen Sie die entsprechenden Variablen zuweisen müssen. Beenden Sie anschließend das Skript. Die Funktion *fununjoin()* ist wie folgt implementiert:

```
Function FunUnjoin()
{
 $option = 0 # 2 = Deaktiviert das Computerkonto, entfernt dieses aber nicht aus Active Directory.
 $objWMI = Get-WmiObject -Class Win32_computersystem `
                -computername $computer
 $objWMI.psbase.Scope.Options.EnablePrivileges = $true
 $objWMI.UnjoinDomainOrWorkgroup($password,$userName,$option)
 exit
}
```

Überprüfen Sie, ob die Variable *$help* vorhanden ist. Gibt es die Variable, wurde das Skript mit dem Parameter **-help** ausgeführt. Rufen Sie entsprechend die Funktion *funhelp()* auf und beenden Sie das Skript. Wenn die Variable *$domainname* vorhanden ist, muss der Computer in die angegebene Domäne aufgenommen werden. Rufen Sie die Funktion *funjoindomain()* auf. Wenn der Parameter **-reboot** angegeben wurde, rufen Sie die Funktion *funreboot()* auf und beenden Sie das Skript. Sie müssen den Computer nicht neu starten, nachdem Sie diesen in die Domäne aufgenommen haben. In diesem Fall ruft das Skript lediglich die Funktion *funjoindomain()* auf und gibt die Ergebnisse des Methodenaufrufs zurück. Das Skript zeigt jedoch erneut die Hilfe an, da der Befehl als unvollständig betrachtet wird. Die Hilfe wird nicht angezeigt, wenn Sie das Skript nur verwenden, um den Computer mit dem Parameter **-reboot** neu zu starten. Sie sollten das Skript jedoch mit den Parametern **-domainname** und **-reboot** ausführen. Wenn Sie keinen Parameter angeben, ruft das Skript die Funktion *funhelp* auf. Codeabschnitt:

```
if($help)   { "Ausgeben der Hilfeinformationen ..." ; funhelp }
if($domainName)
            {
             "Hinzufügen des Computers $computer zur Domäne $domainName"
              FunJoinDomain
            }
if($reboot)
            {
             "Neustarten des Computers $computer ..."
             FunReboot
            }
if(!$help -or !$domainname -or !$reboot)
           {
            "Sie müssen eine Aktion angeben ..."
             funhelp
           }
```

Das vollständige Skript *JoinDomain.ps1* hat folgenden Inhalt:

JoinDomain.ps1

```
param(
      $computer="localhost",
      $domainName,
      $username,
      $password,
```

```
      [switch]$unjoin,
      [switch]$reboot,
      [switch]$help
     )

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: JoinDomain.ps1
Fügt einen Computer zu einer Domäne hinzu.

PARAMETER:
-computer   Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-domainName Der Name der Domäne oder Arbeitsgruppe.
-username   Die Benutzerinformationen.
-password   Das Kennwort des Benutzers.
-unjoin     Entfernt den Computer aus der Domäne oder Arbeitsgruppe.
-reboot     Startet den Computer neu.
-help       Zeigt dieses Hilfethema an.

SYNTAX:
JoinDomain.ps1
Zeigt eine Meldung an, die Sie darauf hinweist, dass Sie eine Aktion angeben müssen,
und gibt die Hilfeinformationen aus.

JoinDomain.ps1 -reboot
Startet den lokalen Computer neu.

JoinDomain.ps1 -reboot -computer MunichServer
Startet einen Remotecomputer namens MunichServer neu. MunichServer muss Mitglied einer
Domäne sein, damit dieser Befehl ausgeführt werden kann.

JoinDomain.ps1 -computer MunichServer -domainName nwtraders.com `
-username nwtraders\administrator -password Password1

Fügt einen Remotecomputer namens MunichServer zur Domäne nwtraders.com unter
Verwendung des Benutzerkontos nwtraders\administrator mit dem Kennwort Password1 hinzu.
Sobald der Computer in die Domäne aufgenommen wurde, wird dieser neu gestartet. Das Computerkonto
wird im Standardcontainer angelegt. Standardmäßig ist dies der Container Computers.

JoinDomain.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

Function funReboot()
{
 if($computer -ne "localhost")
 {
  if($username)
  {
```

```
   $objWMI = Get-WmiObject -Class Win32_operatingsystem `
                -computername $computer -credential $username
   $objWMI.psbase.Scope.Options.EnablePrivileges = $true
   $objWMI.reboot()
  }
  ELSE
   {
    $objWMI = Get-WmiObject -Class Win32_operatingsystem `
                -computername $computer
    $objWMI.psbase.Scope.Options.EnablePrivileges = $true
    $objWMI.reboot()
   }
  exit

 }
 ELSE
 {
  $objWMI = Get-WmiObject -Class Win32_operatingsystem `
              -computername $computer
  $objWMI.psbase.Scope.Options.EnablePrivileges = $true
  $objWMI.reboot()
  exit
 }
}

Function FunJoinDomain()
{
 $command = "netdom join $computer /domain:$domainName " + `
            "/userD:$userName /passwordD:$password"
 $sdcommand = "shutdown /m \\$computer /r " + `
              "/c" + "Zur Domäne hinzugefügt."

 invoke-expression $command
 start-sleep -seconds 2
 "We will now reboot $computer"
 Invoke-expression $sdcommand
 exit
}

Function FunUnjoin()
{
 $option = 0 # 2 = Deaktiviert das Computerkonto, entfernt dieses aber nicht aus Active Directory.
 $objWMI = Get-WmiObject -Class Win32_computersystem `
                -computername $computer
 $objWMI.psbase.Scope.Options.EnablePrivileges = $true
 $objWMI.UnjoinDomainOrWorkgroup($password,$userName,$option)
 exit
}


if($help)   { "Ausgeben der Hilfeinformationen ..." ; funhelp }
if($domainName)
            {
             "Hinzufügen des Computers $computer zur Domäne $domainName."
              FunJoinDomain
            }
```

```
if($reboot)
            {
             "Neustarten des Computers $computer ..."
             FunReboot
            }
if(!$help -or !$domainname -or !$reboot)
           {
            "Sie müssen eine Aktion angeben ..."
            funhelp
           }
```

Festlegen der IP-Adresse

Nachdem der Computer in die Domäne aufgenommen wurde, müssen Sie eventuell eine statische IP-Adresse festlegen. Rufen Sie hierzu mit dem Cmdlet **Get-WmiObject** eine WMI-Klasse ab. Beispielsweise können Sie mit dem Skript *SetIP.ps1* die IP-Adresse, die Subnetzmaske und das Standardgateway auf einem Computer festlegen. Sie können für Verwaltungs- und Konfigurationszwecke außerdem eine Liste der aktuellen Konfigurationseinstellungen der Netzwerkkarten abrufen.

Beginnen Sie das Skript *SetIP.ps1* mit den Befehlszeilenparametern, die Sie unter Verwendung der *param*-Anweisung definieren können. Geben Sie den Parameter **-computer** mit dem Standardwert *localhost* für den lokalen Computer an. Definieren Sie außerdem drei Parameter für die IP-Adresskonfiguration: **-ip** (für die IP-Adresse), **-sm** (für die Subnetzmaske) und **-dg** (für das Standardgateway). Fügen Sie den Parameter **-list** für weitere Funktionen hinzu. Wenn das Skript mit diesem Parameter ausgeführt wird, führen Sie zwei separate WMI-Abfragen aus, um Informationen zur Netzwerkkarte und zur Netzwerkkonfiguration aufzulisten. Fügen Sie den Parameter **-help** hinzu, um die Hilfe anzuzeigen. Die vollständige *param*-Anweisung lautet:

```
param(
      $computer="localhost",
      $ip,
      $sm,
      $dg,
      [switch]$list,
      [switch]$help
     )
```

Erstellen Sie als Nächstes den Hilfetext. Definieren Sie diesen mit einer *here*-Zeichenfolge in der Funktion *funhelp* und weisen Sie die *here*-Zeichenfolge der Variablen *$helptext* zu.

Hinweis Einige Skript-Editoren können *here*-Zeichenfolgen nicht richtig interpretieren. Das Problem ist offensichtlich, wenn Sie das Skript im Editor ausführen. Öffnen Sie das Skript in Notepad, entfernen Sie die Leerzeichen zwischen $helptext = @" BESCHREIBUNG: und geben Sie das Zeichen für den Zeilenumbruch ein. Nachdem Sie das Skript gespeichert haben, sollte dieses korrekt ausgeführt werden. Dieses Problem tritt mit mehreren Skript-Editoren auf.

Definieren Sie in der *here*-Zeichenfolge drei Abschnitte: Beschreibung, Parameter und Syntax. Schließen Sie die *here*-Zeichenfolge mit *"@* ab, zeigen Sie den Inhalt der Variablen *$funhelp* an und beenden Sie die Skriptausführung. Die vollständige Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
```

```
NAME: SetIP.ps1
Konfiguriert eine statische IP-Adresse auf dem lokalen Computer oder auf einem Remotecomputer.

PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-ip       Die zu konfigurierende IP-Adresse.
-sm       Die zu konfigurierende Subnetzmaske.
-dg       Das zu konfigurierende Standardgateway.
-list     Fragt alle Netzwerkkarten ab und gibt die Konfigurationsinformationen aus.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
SetIP.ps1

Zeigt eine Meldung an, die Sie darauf hinweist, dass Sie eine Aktion angeben müssen, und gibt die Hilfeinformationen
aus.

SetIP.ps1 -list -computer MunichServer

Listet alle Netzwerkkarten und deren Konfiguration
auf einem Computer namens MunichServer auf.

SetIP.ps1 -ip "10.0.0.1" -sm "255.0.0.0" -dg "10.0.0.5"

Setzt auf dem lokalen Computer die IP-Adresse auf 10.0.0.1,
die Subnetzmaske auf 255.0.0.0 und das Standardgateway auf 10.0.0.5.

SetIP.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Der Funktion *funhelp()* folgt die Funktion *funevalrtn()*. Diese Funktion besteht aus einer *switch*-Anweisung, die den von der aufgerufenen Methode zurückgegebenen ganzzahligen Wert in eine verständlichere Zeichenfolge umwandelt. Der zurückgegebene Wert wird über die Variable *$rtn* an die Funktion übergeben. Überprüfen Sie die Eigenschaft *ReturnValue* des *Error*-Objekts, das in der Variablen *$rtn* gespeichert ist. Der Wert *0* zeigt an, dass kein Fehler aufgetreten ist. Geben Sie mit dem Cmdlet **Write-Host** eine entsprechende Meldung in Grün aus. Zeigen Sie die Meldung für jeden anderen Wert in Rot an. Wenn der Wert nicht mit einem der Werte in der *switch*-Anweisung übereinstimmt, wird die Standardklausel verwendet. Geben Sie in diesem Fall die Fehlernummer in Rot aus. Initialisieren Sie anschließend alle Variablen mit dem Wert Null. Geben Sie zusätzlich zur Zeichenfolge für den Fehlercode den Wert in der Variablen *$strcall* aus. Diese Variable zeigt an, welche Methode aufgerufen wurde. Die vollständige Funktion *funevalrtn()* ist wie folgt implementiert:

```
function FunEvalRTN($rtn)
{
Switch ($rtn.returnvalue)
  {
   0 { Write-Host -foregroundcolor green "Beim Aufruf von $strCall sind keine Fehler aufgetreten." }
   66 { Write-Host -foregroundcolor red "$strCall berichtet:" `
       " Ungültige Subnetzmaske." }
```

```
   67 { Write-Host -foregroundcolor red "$strCall berichtet:" `
       " Bei der Verarbeitung ist ein Fehler aufgetreten." }
   70 { Write-Host -ForegroundColor red "$strCall berichtet:" `
       " Ungültige IP-Adresse." }
   71 { Write-Host -ForegroundColor red "$strCall berichtet:" `
         " Ungültiges Standardgateway." }
   91 { Write-Host -ForegroundColor red "$strCall berichtet:" `
         " Zugriff verweigert"}
   96 { Write-Host -ForegroundColor red "$strCall berichtet:" `
         " Der DNS-Server konnte nicht gefunden werden."}
   DEFAULT { Write-Host -ForegroundColor red "$strCall berichtet:" `
             " ERROR $($rtn.returnValue)" }
  }
  $rtn=$strCall=$null
}
```

Die nächste Funktion ist *funlist()*, die zwei separate WMI-Abfragen ausführt. Die erste WMI-Abfrage ruft die Informationen aus der Klasse *Win32_NetworkAdapter* ab. Diese Klasse repräsentiert die physischen Netzwerkkarten des Systems. Geben Sie mit dem Cmdlet **Write-Host** eine Kopfzeile aus und verwenden Sie das Sonderzeichen **`n** um einen Zeilenumbruch anzuschließen. Erstellen Sie mit dem Cmdlet **Get-WmiObject** eine Instanz der WMI-Klasse *Win32_NetworkAdapter* auf dem Computer, der in der Variablen *$computer* angegeben ist. Fügen Sie die Informationen in das Cmdlet **Format-List** ein und unterdrücken Sie alle Systemeigenschaften mit Hilfe der Einschränkung *[a-z]**. Diese Einschränkung stellt sicher, dass das Cmdlet **Format-List** ausschließlich die Eigenschaften anzeigt, die mit einem Buchstaben zwischen *a* und *z* beginnen. Erstellen Sie danach mit dem Cmdlet **Get-WmiObject** eine Instanz der WMI-Klasse *Win32_NetworkAdapterConfiguration* auf dem Computer, der in der Variablen *$computer* angegeben ist. Filtern Sie das Ergebnis nach den Buchstaben *a* bis *z*. Die vollständige Funktion *funlist()* ist wie folgt implementiert:

```
Function funlist()
{
 Write-host "Ermitteln der Netzwerkkarten auf dem Computer $($computer):`n"

 Get-WmiObject
 -Class
 win32_networkadapter
 `
 -computername $computer | format-list [a-z]*

 Write-host "Ermitteln der Netzwerkkonfiguration auf dem Computer " `
 "$($computer):`n"

 Get-WmiObject -Class win32_networkadapterconfiguration `
 -computername $computer | format-list [a-z]*
 exit
}
```

Sie müssen nun die Befehlszeilenparameter überpüfen. Suchen Sie nach dem Parameter **-help**. Wenn die Variable *$help* vorhanden ist, wurde der Parameter **-help** angegeben. Rufen Sie in diesem Fall die Funktion *funhelp()* auf. Wenn die Variable *$list* vorhanden ist, rufen Sie die Funktion *funlist()* auf und geben Sie die Netzwerkkonfiguration aus. Bei diesen beiden Aktionen wird das Skript mit der *exit*-Anweisung nach Abschluss der jeweiligen Funktion beendet. Wenn weder **-list** noch **-help** angegeben wurde, müssen Sie eine IP-Adresse konfigurieren. Sie benötigen hierzu eine IP-Adresse, eine Subnetz-

maske und das Standardgateway. Überprüfen Sie, ob die entsprechenden Parameter **-ip**, **-sm** und **-dg** angegeben wurden. Wenn einer dieser Parameter fehlt, geben Sie eine Meldung aus und rufen die Funktion *funhelp()* auf. Codeabschnitt:

```
if($help) { funhelp }
if($list) { funlist }
if(!$ip -or !$sm -or !$dg)
   { "Eine Aktion ist erforderlich ... " ; funhelp }
```

Nachdem Sie die Befehlszeilenparameter überprüft haben, müssen Sie die IP-Adresse konfigurieren. Erstellen Sie eine Instanz der WMI-Klasse *Win32_NetworkAdapterConfiguration*, um die Konfiguration der Netzwerkkarte zu bearbeiten. Verwenden Sie hierzu das Cmdlet **Get-WmiObject**, geben Sie die Klasse *Win32_NetworkAdapterConfiguration* sowie den Computernamen an und filtern Sie nach den für IP aktivierten Netzwerkkarten. Weisen Sie das zurückgegebene WMI-Verwaltungsobjekt der Variablen *$objWMI* zu. Codeabschnitt:

```
$global:RTN = $null
$metric = [int32[]]1
$objWMI = Get-WmiObject -Class win32_networkadapterconfiguration `
 -computer $computer -filter "ipenabled = 'true'"
```

Rufen Sie die Methode *EnableStatic()* auf, um eine statische IP-Adresse zu konfigurieren. Die Methode *EnableStatic()* erfordert jeweils eine Zeichenfolge für die IP-Adresse und die Subnetzmaske. Speichern Sie den zurückgegebenen Wert in der Variablen *$rtn*, rufen Sie die Funktion *funevalrtn()* auf und übergeben Sie die Variable *$rtn*. Erstellen Sie die Variable *$strcall*, um den umgewandelten Rückgabewert auszugeben. Codeabschnitt:

```
$RTN=$objwmi.EnableStatic($ip, $sm)
$strCall="Konfigurieren einer statischen IP-Adresse und Subnetzmaske."
FunEvalRTN($rtn)
```

Rufen Sie als Nächstes die Methode *SetGateways()* auf, um das Standardgateway zu konfigurieren. Die WMI-Methode *SetGateways()* erfordert ein Array. In Windows PowerShell müssen Sie die Einschränkung *[array]* jedoch nicht angeben. Windows PowerShell unterstützt die automatische Konvertierung. Zusätzlich zur IP-Adresse für das Standardgateway erfordert die Methode *SetGateways()* noch einen metrischen Wert. Im vorliegenden Skript ist der Wert der Variablen *$metric* hart codiert. Übergeben Sie den von der Methode zurückgegebenen Wert an die Funktion *funevalrtn()*. Codeabschnitt:

```
$RTN=$objwmi.SetGateways($dg, $metric)
$strCall="Konfigurieren von Standardgateway und Metrik."
FunEvalRTN($rtn)
```

Das vollständige Skript *SetIP.ps1* hat folgenden Inhalt:

SetIP.ps1

```
param(
      $computer="localhost",
      $ip,
      $sm,
      $dg,
      [switch]$list,
      [switch]$help
     )

function funHelp()
{
$helpText=@"
```

```
BESCHREIBUNG:
NAME: SetIP.ps1
Konfiguriert eine statische IP-Adresse auf dem lokalen Computer oder auf einem Remotecomputer.

PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-ip       Die zu konfigurierende IP-Adresse.
-sm       Die zu konfigurierende Subnetzmaske.
-dg       Das zu konfigurierende Standardgateway.
-list     Fragt alle Netzwerkkarten ab und gibt die Konfigurationsinformationen aus.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
SetIP.ps1

Zeigt eine Meldung an, die Sie darauf hinweist, dass Sie eine Aktion angeben müssen, und gibt die Hilfeinformationen
aus.

SetIP.ps1 -list -computer MunichServer

Listet alle Netzwerkkarten und deren Konfiguration
auf einem Computer namens MunichServer auf.

SetIP.ps1 -ip "10.0.0.1" -sm "255.0.0.0" -dg "10.0.0.5"

Setzt auf dem lokalen Computer die IP-Adresse auf 10.0.0.1,
die Subnetzmaske auf 255.0.0.0 und das Standardgateway auf 10.0.0.5.

SetIP.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

function FunEvalRTN($rtn)
{
Switch ($rtn.returnvalue)
  {
   0 { Write-Host -foregroundcolor green "Beim Aufruf von $strCall sind keine Fehler aufgetreten." }
   66 { Write-Host -foregroundcolor red "$strCall berichtet:" `
       " Ungültige Subnetzmaske." }
   67 { Write-Host -foregroundcolor red "$strCall berichtet:" `
       " Bei der Verarbeitung ist ein Fehler aufgetreten." }
   70 { Write-Host -ForegroundColor red "$strCall berichtet:" `
       " Ungültige IP-Adresse." }
   71 { Write-Host -ForegroundColor red "$strCall berichtet:" `
         " Ungültiges Standardgateway." }
   91 { Write-Host -ForegroundColor red "$strCall berichtet:" `
         " Zugriff verweigert."}
   96 { Write-Host -ForegroundColor red "$strCall berichtet:" `
         " Der DNS-Server konnte nicht gefunden werden."}
   DEFAULT { Write-Host -ForegroundColor red "$strCall berichtet:" `
             " ERROR $($rtn.returnValue)" }
```

```
  }
  $rtn=$strCall=$null
}

Function funlist()
{
 Write-host "Ermitteln der Netzwerkkarten auf dem Computer $($computer):`n"

 Get-WmiObject -Class win32
 _networkadapter
 -computername $computer | format-list [a-z]*

 Write-host "Ermitteln der Netzwerkkonfiguration auf dem Computer  " `
 "$($computer):`n"

 Get-WmiObject -Class win32_networkadapterconfiguration `
 -computername $computer | format-list [a-z]*
 exit
}

if($help) { funhelp }
if($list) { funlist }

if(!$ip -or !$sm -or !$dg) { "Eine Aktion ist erforderlich ... " ; funhelp }

$global:RTN = $null
$metric = [int32[]]1
$objWMI = Get-WmiObject -Class win32_networkadapterconfiguration `
 -computer $computer -filter "ipenabled = 'true'"

$RTN=$objwmi.EnableStatic($ip, $sm)
$strCall="Konfigurieren einer statischen IP-Adresse und Subnetzmaske."
FunEvalRTN($rtn)

$RTN=$objwmi.SetGateways($dg, $metric)
$strCall="Konfigurieren von Standardgateway und Metrik."
FunEvalRTN($rtn)
```

Konfigurieren der DNS-Einstellungen

Möglicherweise müssen Sie die DNS-Einstellungen (Domain Name System) auf einem Remotecomputer konfigurieren, beispielsweise die Suchreihenfolge, das Suffix und den DNS-Server. Sie können diese Aufgabe mit einem Skript auf der Grundlage von WMI ausführen.

Erstellen Sie im Skript *SetDNS.ps1* als Erstes die Befehlszeilenparameter. Der Parameter **-computer** gibt den Namen des Computer an, mit dem die Verbindung hergestellt wird. Der Parameter **-dnsdomain** enthält den Namen der DNS-Domäne, die auf dem Client konfiguriert wird. Der Parameter **-dnsserver** gibt die IP-Adresse des primären DNS-Servers an. Das DNS-Suffix wird mit dem Parameter **-dnssuffix** festgelegt. Die nächsten beiden Parameter haben feste Werte. Der erste Parameter mit einem festen Wert ist **-list**. Wenn dieser Parameter vorhanden ist, gibt das Skript Informationen zur Netzwerkkarte und zur IP-Konfiguration aus. Der zweite Parameter mit einem festen Wert ist -**help**. Bei Angabe dieses Parameters wird die Hilfe anzeigt. Die *param*-Anweisung lautet:

```
param(
      $computer="localhost",
      $dnsdomain,
      $dnsServer,
      $dnsSuffix,
      [switch]$list,
      [switch]$help
     )
```

Die Funktion *funhelp()* wird entsprechend dem Parameter **-help** aufgerufen. Erstellen Sie für die Funktion *funhelp()* die Variable *$helpText*, der eine *here*-Zeichenfolge zugewiesen wird. Die *here*-Zeichenfolge erlaubt Ihnen, bei der Eingabe der Hilfeinformationen die Regeln für Anführungszeichen zu ignorieren. Die *here*-Zeichenfolge umfasst jeweils einen Abschnitt für die Beschreibung, Parameter und Syntax. Zeigen Sie anschließend den Inhalt der Variablen *$helpText* an und beenden Sie das Skript. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: SetDNS.ps1
Konfiguriert die DNS-Einstellungen auf dem lokalen Computer oder einem Remotecomputer.

PARAMETER:
-computer  Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-list      Fragt alle für IP aktivierten Netzwerkkarten ab.
-dnsserver Gibt den DNS-Server an.
-dnsDomain Gibt den DNS-Domänennamen an.
-dnsSuffix Gibt das DNS-Suffix an.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
SetDNS.ps1

Zeigt eine Meldung an, die Sie darauf hinweist, dass Sie eine Aktion angeben müssen, und gibt die Hilfeinformationen
aus.

SetDNS.ps1 -list -computer MunichServer

Listet alle Netzwerkkarten und deren Konfigurationsinformationen auf einem
Computer namens MunichServer auf.

SetDNS.ps1 -dnsServer "10.0.0.2" -dnsDomain "nwtraders.com" `
           -dnsSuffix "nwtraders.com"

Legt auf dem lokalen Computer den DNS-Server auf 10.0.0.2, die DNS-Domäne auf nwtraders.com
und das DNS-Suffix auf nwtraders.com fest.

SetDNS.ps1 -dnsServer "10.0.0.2" -dnsDomain "nwtraders.com" `
           -dnsSuffix "nwtraders.com" -computer munichServer

Legt auf einem Remotecomputer namens munichServer
den DNS-Server auf 10.0.0.2, die DNS-Domäne auf nwtraders.com
und das DNS-Suffix auf nwtraders.com fest.

SetDNS.ps1 -help
```

```
Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Sie müssen eine Funktion erstellen, um den von den DNS-Methoden zurückgegebenen Code auszuwerten. Da dieser Code mit dem Code zum Festlegen von IP-Adressen identisch ist, können Sie die Funktion aus dem Skript *SetIP.ps1* übernehmen, ohne Änderungen vornehmen zu müssen. Unter „Einbeziehen von Funktionen aus anderen Skripts“ wird eine Methode zum Wiederverwenden von Funktionen beschrieben.

Einbeziehen von Funktionen aus anderen Skripts

Sie müssen die Funktion *funevalrtn()* nicht unbedingt in das Skript *SetDNS.ps1* kopieren und einfügen. Sie können stattdessen eine Methode verwenden, die als *Dot-Sourcing* bezeichnet wird. In anderen Programmiersprachen wird diese Methode mit dem Schlüsselwort *include* bezeichnet. Diese Methode hat den Vorteil, dass Sie eine Funktion nicht kopieren müssen, um diese in einem Skript zu verwenden.

Die *Dot-Sourcing* Methode ist jedoch nicht ganz unproblematisch. Beispielsweise muss das aufrufende Skript immer zusammen mit dem *Include*-Skript ausgeführt werden. Diese Abhängigkeit schränkt die Portabilität des Skripts ein, da dieses stets auf das *Include*-Skript zugreifen muss. Ein weiteres Problem mit dieser Methode ist, dass die Problembehandlung schwieriger ist, da zwei Skripts überprüft werden müssen. Außerdem ist das Skript schwieriger zu lesen, da Sie meist beide Skripts analysieren müssen, um die Problemursache herauszufinden.

Trotz dieser Nachteile ist die *Include*-Methode weit verbreitet. Viele Netzwerkadministratoren erstellen Skriptbibliotheken mit Funktionen, die die Funktionalität von Windows PowerShell erweitern.

Die Methode ist in den folgenden beiden Skripts veranschaulicht. Das Skript *CallFunctionLib.ps1* ruft das Skript *FunctionLib.ps1* mit folgender Codezeile auf:

```
. c:\fso\functionlib.ps1
```

Die Funktionen des Skripts *FunctionLib.ps1* sind nun im Skript *CallFunctionLib.ps1* verfügbar.

Im Skript *FunctionLib.ps1* wurden die beiden Funktionen *addOne()* und *addTwo()* erstellt. Nachdem die Funktionen in das aufrufende Skript einbezogen wurden, können diese genauso wie reguläre Funktionen aufgerufen werden:

```
addone(1)
addtwo(2)
```

Der Grund ist, dass die Funktionen zum *PSDrive*-Laufwerk *function:*\ hinzugefügt wurden, was mit der letzten Skriptzeile verdeutlicht wird:

```
get-childitem function:\
```

Das beiden Skripts *CallFunctionLib.ps1* und *FunctionLib.ps1* haben folgende Inhalte:

CallFunctionLib.ps1

```
. c:\fso\functionlib.ps1
addone(1)
addtwo(2)

get-childitem function:\
```

FunctionLib.ps1

```
function addOne($intIN)
 {
  $intIN ++
  $intIN
 }

function addTwo($intIN)
 {
  $intIn+=2
  $intIn
 }
```

Damit das Skript *SetDNS.ps1* unabhängig bleibt, geben Sie die Funktion *funevalrtn()* nicht in einer *Include*-Datei ein (siehe „Einbeziehen von Funktionen aus anderen Skripts“). Das Skript *WMIFunctions.ps1* auf der Begleit-CD enthält drei Funktionen, mit denen Sie Skripts anpassen können, wenn Sie eine Funktionsbibliothek verwenden möchten. Sie können auch die Funktion *funevalrtn()* kopieren und in das Skript *SetDNS.ps1* einfügen. Weitere Informationen zu dieser Funktion finden Sie im Abschnitt „Festlegen der IP-Adresse“.

```
function FunEvalRTN($rtn)
{
 Switch ($rtn.returnvalue)
  {
   0 { Write-Host -foregroundcolor green "Beim Aufruf von $strCall sind keine Fehler aufgetreten." }
   66 { Write-Host -foregroundcolor red "$strCall berichtet:" `
       " Ungültige Subnetzmaske." }
   70 { Write-Host -ForegroundColor red "$strCall berichtet:" `
       " Ungültige IP-Adresse." }
   71 { Write-Host -ForegroundColor red "$strCall berichtet:" `
         " Ungültiges Standardgateway." }
   91 { Write-Host -ForegroundColor red "$strCall berichtet:" `
         " Zugriff verweigert"}
   96 { Write-Host -ForegroundColor red "$strCall berichtet:" `
         " Der DNS-Server konnte nicht gefunden werden."}
   DEFAULT { Write-Host -ForegroundColor red "$strCall berichtet:" `
              " ERROR $($rtn.returnValue)" }
  }
   $rtn=$strCall=$null
}
```

Erstellen Sie die Funktion *funlist()*, um alle Netzwerkkarten und deren Konfigurationseinstellungen aufzulisten. Implementieren Sie diese Funktion, anstatt die im Abschnitt „Einbeziehen von Funktionen aus anderen Skripts“ beschriebene *Include*-Methode einzusetzen. Weitere Informationen zur Funktion *funlist()* finden Sie im Abschnitt „Festlegen der IP-Adresse“.

```
Function funlist()
{
 Write-host "Ermitteln der Netzwerkkarten auf dem Computer $($computer):`n"

 Get-WmiObject
 -Class win32_networkadapter `
 -computername $computer | format-list [a-z]*

 Write-host "Ermitteln der Netzwerkkonfiguration auf dem Computer " `
 "$($computer):`n"

 Get-WmiObject -Class win32_networkadapterconfiguration `
 -computername $computer | format-list [a-z]*
 exit
}
```

Der erste Befehlszeilenparameter, den Sie überprüfen müssen, ist **-help**. Wenn Sie die Variable *$help* finden, rufen Sie die Funktion *funhelp()* auf. Suchen Sie als Nächstes nach dem Parameter **-list**. Wenn der Parameter zur Laufzeit angegeben wurde, ist die Variable *$list* vorhanden und Sie müssen die Funktion *funlist()* aufrufen. Suchen Sie anschließend nach fehlenden Argumenten, da das Skript die drei DNS-Konfigurationsparameter erfordert. Überprüfen Sie, ob *$dnsdomain*, *$dnsserver,* und *$dnssuffix* vorhanden sind. Fehlt auch nur eine dieser Variablen, rufen Sie die Funktion *funhelp()* auf, um die Hilfe anzuzeigen und die Skriptausführung zu beenden. Codeabschnitt:

```
if($help) { funhelp }
if($list) { funlist }
if(!$dnsdomain -or !$dnsServer -or !$dnsSuffix)
  { "Eine Aktion ist erforderlich ... " ; funhelp }
```

Deklarieren Sie als Nächstes zwei Variablen. Die erste Variable ist *$rtn*, die als globale Variable initialisiert und auf *$null* festgelegt wird. Erstellen Sie außerdem die Variable *$namespace* und legen Sie diese auf den WMI-Namespace *root\cimv2* fest, in der die WMI-Klasse *Win32_NetworkAdapter-Configuration* zu finden ist.

Hinweis Da der WMI-Namespace *root\cimv2* der WMI-Standardnamespace von Windows Vista und Windows Server 2008 ist, ist dieser Parameter in vielen WMI-Skripts nicht erforderlich. Da der WMI-Standardnamespace jedoch auf einfache Weise geändert werden kann, geben einige Netzwerkadministratoren den Parameter routinemäßig in Skripts direkt an. Wenn ein Skript den WMI-Standardnamespace implizit verwendet und der Standardnamespace geändert wird, schlägt das Skript fehl. Dieses Problem ist schwierig zu beheben.

Sie müssen nun auf WMI zugreifen. Zuvor müssen Sie jedoch einige Variablen erstellen. Die erste Variable ist *$rtn*, die als globale Variable initialisiert und auf Null festgelegt wird. In der Variablen *$class* wird die abgefragte WMI-Klasse *Win32_NetworkAdapterConfiguration* gespeichert. In der Variablen *$namespace* wird der Name des WMI-Namespaces angegeben, der die WMI-Klasse *Win32_NetworkAdapterConfiguration* enthält. Legen Sie die Variable *$namespace* also auf die Zeichenfolge *"root\cimv2"* fest. Sie können nun die WMI-Klasse instantiieren. Geben Sie hierzu im

Cmdlet **Get-WmiObject** den Namen in der Variablen *$class*, den Namespace mit der Variablen *$namespace* und den Computernamen mit der Variablen *$computer* an. Verwenden Sie den Parameter **-filter**, um ausschließlich die Netzwerkkarten auszuwählen, die an das TCP/IP-Protokoll gebunden sind. Das resultierende WMI-Objekt wird in der Variablen *$objWMI* gespeichert. Codeabschnitt:

```
$global:RTN = $null
$class = "win32_networkadapterconfiguration"
$namespace = "root\cimv2"
$objWMI = Get-WmiObject -Class $class `
        -namespace $namespace -computername $computer
        -filter "ipenabled = 'true'"
```

Rufen Sie anschließend die Methode *SetDnsDomain()* auf und übergeben Sie den in der Variablen *$dnsdomain* gespeicherten Domänennamen. Die Methode *SetDnsDomain()* gibt ein Fehlerobjekt zurück. Übergeben Sie das Fehlerobjekt an die Funktion *funevalrtn()*, um zu überprüfen, ob die Methode erfolgreich aufgerufen wurde. Rufen Sie die Methode *SetDnsServerSearchOrder()* auf und geben Sie den in der Variablen *$dnsserver* gespeicherten Wert an. Rufen Sie die Funktion *funevalrtn()* auf, um den erfolgreichen Abschluss der Konfiguration für die Suchreihenfolge zu überprüfen. Codeabschnitt:

```
$RTN=$objwmi.SetDNSDomain($dnsdomain)
$strCall="Konfigurieren des DNS-Domänennamens."
FunEvalRTN($rtn)

$RTN=$objwmi.SetDNSServerSearchOrder($dnsServer)
$strCall="Konfigurieren der DNS-Server-Suchreihenfolge."
FunEvalRTN($rtn)
```

Der letzte Schritt betrifft das Festlegen der Suchreihenfolge für das DNS-Suffix. Rufen Sie hierzu die Methode *SetDnsSuffixSearchOrder()* der WMI-Klasse *Win32_NetworkAdapterConfiguration* auf. Sie verwenden an dieser Stelle eine andere WMI-Methode, da die Methode *SetDnsSuffixSearchOrder()* über das Cmdlet **Get-WmiObject** nicht verfügbar ist. Erstellen Sie stattdessen eine Instanz der .NET Framework-Klasse *System.Management.ManagementObject.ManagementClass* und greifen Sie über diese Klasse auf die Methode *SetDnsSuffixSearchOrder()* der Klasse *Win32_NetworkAdapterConfiguration* zu.

Da der Konstruktor für diese Klasse etwas verwirrend ist, sollten Sie Variablen verwenden. Variablen vereinfachen außerdem die Wiederverwendung dieses Codeabschnitts in anderen Skripts. Erstellen Sie die Variable *$wmiclass*, um den Pfad zur WMI-Klasse zu speichern, einschließlich des Computernamens, des Namespaces und des Klassennamens. Erstellen Sie eine Instanz der .NET Framework-Verwaltungsklasse und geben Sie den Wert in der Variablen *$wmiclass* als Konstruktor an. Nachdem das Verwaltungsobjekt abgerufen wurde, rufen Sie die Methode *SetDnsSuffixSearchOrder()* auf. Codeabschnitt:

```
$wmiclass = "\\$computer" + "\" + $namespace + ":" + $class
$wmi = [wmiclass]"$wmiclass"
$rtn = $wmi.SetDNSSuffixSearchOrder($dnsSuffix)
$strCall="Konfigurieren der DNS-Suffix-Suchreihenfolge."
FunEvalRTN($rtn)
```

Das vollständige Skript *SetDNS.ps1* hat folgenden Inhalt:

SetDNS.ps1

```
param(
      $computer="localhost",
      $dnsdomain,
      $dnsServer,
      $dnsSuffix,
      [switch]$list,
      [switch]$help
     )

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: SetDNS.ps1
Konfiguriert die DNS-Einstellungen auf dem lokalen Computer oder einem Remotecomputer.

PARAMETER:
-computer  Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-list      Fragt alle für IP aktivierten Netzwerkkarten ab.
-dnsserver Gibt den DNS-Server an.
-dnsDomain Gibt den DNS-Domänennamen an.
-dnsSuffix Gibt das DNS-Suffix an.
-help      Zeigt dieses Hilfethema an.

SYNTAX:
SetDNS.ps1

Zeigt eine Meldung an, die Sie darauf hinweist, dass Sie eine Aktion angeben müssen, und gibt die Hilfeinformationen
aus.

SetDNS.ps1 -list -computer MunichServer

Listet alle Netzwerkkarten und deren Konfigurationsinformationen auf einem
Computer namens MunichServer auf.

SetDNS.ps1 -dnsServer "10.0.0.2" -dnsDomain "nwtraders.com" `
           -dnsSuffix "nwtraders.com"

Legt auf dem lokalen Computer den DNS-Server auf 10.0.0.2, die DNS-Domäne auf nwtraders.com
und das DNS-Suffix auf nwtraders.com fest.

SetDNS.ps1 -dnsServer "10.0.0.2" -dnsDomain "nwtraders.com" `
           -dnsSuffix "nwtraders.com" -computer munichServer

Legt auf einem Remotecomputer namens munichServer
den DNS-Server auf 10.0.0.2, die DNS-Domäne auf nwtraders.com
und das DNS-Suffix auf nwtraders.com fest.

SetDNS.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
```

```
exit
}

function FunEvalRTN($rtn)
{
 Switch ($rtn.returnvalue)
  {
   0 { Write-Host -foregroundcolor green "Beim Aufruf von $strCall sind keine Fehler aufgetreten." }
   66 { Write-Host -foregroundcolor red "$strCall berichtet:" `
       " Ungültige Subnetzmaske." }
   70 { Write-Host -ForegroundColor red "$strCall berichtet:" `
       " Ungültige IP-Adresse." }
   71 { Write-Host -ForegroundColor red "$strCall berichtet:" `
         " Ungültiges Standardgateway." }
   91 { Write-Host -ForegroundColor red "$strCall berichtet:" `
         " Zugriff verweigert"}
   96 { Write-Host -ForegroundColor red "$strCall berichtet:" `
         " Der DNS-Server konnte nicht gefunden werden."}
   DEFAULT { Write-Host -ForegroundColor red "$strCall berichtet:" `
             " ERROR $($rtn.returnValue)" }
  }
   $rtn=$strCall=$null
}

Function funlist()
{
 Write-host "Ermitteln der Netzwerkkarten auf dem Computer $($computer):`n"

 Get-WmiObject -Class win32_networkadapter `
 -computername $computer | format-list [a-z]*

 Write-host "Ermitteln der Netzwerkkonfiguration auf dem Computer " `
 "$($computer):`n"

 Get-WmiObject -Class win32_networkadapterconfiguration `
 -computername $computer | format-list [a-z]*
 exit
}

if($help) { funhelp }
if($list) { funlist }
if(!$dnsdomain -or !$dnsServer -or !$dnsSuffix)
  { "Eine Aktion ist erforderlich ... " ; funhelp }

$global:RTN = $null
$class = "win32_networkadapterconfiguration"
$namespace = "root\cimv2"
$objWMI = Get-WmiObject -Class $class `
        -namespace $namespace -computername $computer
        -filter "ipenabled = 'true'"

$RTN=$objwmi.SetDNSDomain($dnsdomain)
$strCall="Konfigurieren des DNS-Domänennamens."
FunEvalRTN($rtn)

$RTN=$objwmi.SetDNSServerSearchOrder($dnsServer)
```

```
$strCall="Konfigurieren der DNS-Server-Suchreihenfolge."
FunEvalRTN($rtn)

$wmiclass = "\\$computer" + "\" + $namespace + ":" + $class
$wmi = [wmiclass]"$wmiclass"
$rtn = $wmi.SetDNSSuffixSearchOrder($dnsSuffix)
$strCall="Konfigurieren der DNS-Suffix-Suchreihenfolge."
FunEvalRTN($rtn)
```

Umbenennen des Servers

Das Umbenennen eines Servers unter Verwendung von WMI kann schwierig sein und sollte lokal und nicht mit einer WMI-Methode ausgeführt werden. Der Grund hierfür ist, dass die WMI-Methoden beschränkt sind. Sie können einen Server nicht umbenennen, nachdem dieser in eine Domäne aufgenommen wurde. Wenn Sie den Computer noch nicht in die Domäne aufgenommen und die Sicherheitseinstellungen nicht konfiguriert haben, um den Arbeitsgruppenzugriff zu gewähren, können Sie den Server ebenfalls nicht umbenennen. Sie können den Computer neu starten, wenn Sie über Zugriffsrechte für WMI verfügen.

Beginnen Sie das Skript *RenameReboot.ps1* mit einer *param*-Anweisung und definieren Sie die erforderlichen Befehlszeilenparameter, die Ihnen das Steuern der Skriptausführung zur Laufzeit ermöglichen. Der erste Parameter ist **-computer**, um den Zielcomputer anzugeben. Wenn der Windows Server 2008 Server Core einen zufällig generierten Computernamen verwendet, stellen Sie die Verbindung mit dem Server unter Verwendung der IP-Adresse her.

Vorsicht Wenn Sie im Skript *RenameReboot.ps1* für den Parameter *-computer* keinen Wert angeben, wird das Skript für den lokalen Computer ausgeführt. Wenn Sie den Parameter **-newname** angeben, wird der lokale Computer möglicherweise umbenannt. Sie müssen mindestens den lokalen Computer neu starten. Da das Skript Vorgänge automatisiert, werden keine Hinweise oder Warnungen angezeigt.

Dem Parameter **-computer** folgt der Parameter **-newname**, um den neuen Namen für den Server festzulegen. Die folgenden Parameter **-user** und **-password** dienen der Angabe von Anmeldeinformationen. Außerdem werden die Parameter **-reboot** und **-help** unterstützt. Der Parameter **-reboot** kann zusammen mit dem Parameter **-computer** verwendet werden, um einen Remoteserver neu zu starten. Sie müssen dem Server für einen Neustart keinen neuen Namen zuweisen. Codeabschnitt:

```
param(
      $computer="localhost",
      $newName,
      $user,
      $password,
      [switch]$reboot,
      [switch]$help
     )
```

Die nächste Funktion ist *funhelp()*, um eine Hilfemeldung anzuzeigen, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Beginnen Sie die Funktion *funhelp()* mit einer Definition der Variablen *$helptext*. Weisen Sie der Variablen *$helptext* eine *here*-Zeichenfolge zu. Die *here*-Zeichenfolge ist in drei Abschnitte unterteilt: Beschreibung, Parameter und Syntax. Die Funktion *funhelp* zeigt den Inhalt der Variablen *$helptext* an und beendet dann die Skriptausführung. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: RenameReboot.ps1
Benennt einen lokalen oder einen Remotecomputer um.

PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-newname  Der neue Name des Computers.
-user     Die Benutzerinformationen.
-password Das Kennwort des Benutzers.
-reboot   Startet den Computer neu.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
RenameReboot.ps1
Zeigt eine Meldung an, die Sie darauf hinweist, dass Sie eine Aktion angeben müssen, und gibt die Hilfeinformationen
aus.

RenameReboot.ps1 -reboot
Startet den lokalen Computer neu.

RenameReboot.ps1 -reboot -computer MunichServer
Startet einen Remotecomputer namens MunichServer neu.

RenameReboot.ps1 -computer MunichServer -newname BerlinServer

Benennt den lokalen Computer namens MunichServer in BerlinServer um.

RenameReboot.ps1 -computer MunichServer -newname BerlinServer
-user munich\admin -password Password1

Benennt einen Remotecomputer namens MunichServer in BerlinServer um und verwendet
das Benutzerkonto admin aus der Domäne munich mit dem Kennwort Password1.


RenameReboot.ps1 -computer MunichServer -newname BerlinServer
-user munich\admin -reboot

Benennt einen Remotecomputer namens MunichServer in BerlinServer um und verwendet
das Benutzerkonto admin aus der Domäne munich. Das Kennwort wird in einem separaten
Dialogfeld abgefragt. Der umbenannte Computer BerlinServer wird neu gestartet.

RenameReboot.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
```

Erstellen Sie nun die Funktion *funrename()*, um den angegebenen Computer mit dem Befehl **netdom** umzubenennen. Geben Sie in diesem **netdom**-Befehl den Parameter **renamecomputer** an. Diese Vorgehensweise hat den Vorteil, dass Sie den Befehl remote ausführen können. Geben Sie im Parameter **/newname** des **netdom**-Befehls den neuen Namen an, der in der Variablen *$newname* gespeichert ist.

Verwenden Sie die Anmeldeinformationen, die in den Befehlsparametern angegeben wurden. Der Befehl wird als Zeichenfolge erstellt und der Variablen *$command* zugewiesen. Führen Sie die Zeichenfolge mit dem Cmdlet **Invoke-Expression** aus und geben Sie die in der Variablen *$command* gespeicherte Zeichenfolge an. Die Funktion *funrename()* ist wie folgt implementiert:

```
Function funRename()
{
  $command = "Netdom renamecomputer $($computer) /newname:$newname" + `
  " /userD:$user /passwordD:$password"
  Invoke-expression $command
}
```

Die nächste Funktion ist *funreboot()*, um den Server unter Verwendung von WMI neu zu starten. Die Funktion *funreboot()* ist mit der Funktion *funreboot()* im Skript *JoinDomain.ps1* identisch, das bereits im Abschnitt „Aufnehmen eines Computers in eine Domäne“ beschrieben wurde. Die Funktion *funreboot()* ist wie folgt implementiert:

```
Function funReboot()
{
 if($computer -ne "localhost")
 {
  if($user)
  {
   $objWMI = Get-WmiObject -Class Win32_operatingsystem `
                -computername $computer -credential $user
   $objWMI.psbase.Scope.Options.EnablePrivileges = $true
   $objWMI.reboot()
  }
  ELSE
   {
    $objWMI = Get-WmiObject -Class Win32_operatingsystem `
                -computername $computer
    $objWMI.psbase.Scope.Options.EnablePrivileges = $true
    $objWMI.reboot()
   }
  exit
 }
 ELSE
 {
  $objWMI = Get-WmiObject -Class Win32_operatingsystem `
              -computername $computer
  $objWMI.psbase.Scope.Options.EnablePrivileges = $true
  $objWMI.reboot()
  exit
 }
}
```

Überprüfen Sie nun die Befehlszeilenparameter. Überprüfen Sie zuerst, ob die Variable *$help* angegeben wurde und rufen Sie in diesem Fall die Funktion *funhelp()* auf. Ist die Variable *$newname* vorhanden, rufen Sie die Funktion *funrename()* auf. Wenn Sie die Variable *$reboot* finden, wurde das Skript mit dem Parameter **-reboot** ausgeführt und Sie müssen die Funktion *reboot()* ausführen. Überprüfen Sie anschließend, ob die drei übrigen Parameter angegeben wurden. Wenn keiner dieser Parameter vorhanden ist, geben Sie eine Meldung aus und rufen Sie die Funktion *funhelp()* auf. Codeabschnitt:

```
if($help)   { "Ausgeben der Hilfeinformationen ..." ; funhelp }
if($newName)
            {
             "Umbenennen des Computers $computer in $newName."
              FunRename
            }
if($reboot)
            {
             "Neustarten des Computers $computer ..."
             FunReboot
            }
if(!$help -or !$newname -or !$reboot)
           {
            "Sie müssen eine Aktion angeben ..."
             funhelp
           }
```

Das vollständige Skript *RenameReboot.ps1* hat folgenden Inhalt:

RenameReboot.ps1

```
param(
      $computer="localhost",
      $newName,
      $user,
      $password,
      [switch]$reboot,
      [switch]$help
     )

function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: RenameReboot.ps1
Benennt einen lokalen oder einen Remotecomputer um.

PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-newname  Der neue Name des Computers.
-user     Die Benutzerinformationen.
-password Das Kennwort des Benutzers.
-reboot   Startet den Computer neu.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
RenameReboot.ps1
Zeigt eine Meldung an, die Sie darauf hinweist, dass Sie eine Aktion angeben müssen, und gibt die Hilfeinformationen
aus.

RenameReboot.ps1 -reboot
Startet den lokalen Computer neu.

RenameReboot.ps1 -reboot -computer MunichServer
Startet einen Remotecomputer namens MunichServer neu.

RenameReboot.ps1 -computer MunichServer -newname BerlinServer
```

```
Benennt den lokalen Computer namens MunichServer in BerlinServer um.

RenameReboot.ps1 -computer MunichServer -newname BerlinServer
-user munich\admin -password Password1

Benennt einen Remotecomputer namens MunichServer in BerlinServer um und verwendet
das Benutzerkonto admin aus der Domäne munich mit dem Kennwort Password1.


RenameReboot.ps1 -computer MunichServer -newname BerlinServer
-user munich\admin -reboot

Benennt einen Remotecomputer namens MunichServer in BerlinServer um und verwendet
das Benutzerkonto admin aus der Domäne munich. Das Kennwort wird in einem separaten
Dialogfeld abgefragt. Der umbenannte Computer BerlinServer wird neu gestartet.

RenameReboot.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

Function funRename()
{
  $command = "Netdom renamecomputer $($computer) /newname:$newname" + `
  " /userD:$user /passwordD:$password"
  Invoke-expression $command
}

Function funReboot()
{
 if($computer -ne "localhost")
 {
  if($user)
  {
   $objWMI = Get-WmiObject -Class Win32_operatingsystem `
               -computername $computer -credential $user
   $objWMI.psbase.Scope.Options.EnablePrivileges = $true
   $objWMI.reboot()
  }
  ELSE
   {
    $objWMI = Get-WmiObject -Class Win32_operatingsystem `
               -computername $computer
    $objWMI.psbase.Scope.Options.EnablePrivileges = $true
    $objWMI.reboot()
   }
  exit
 }
 ELSE
 {
  $objWMI = Get-WmiObject -Class Win32_operatingsystem `
```

```
              -computername $computer
  $objWMI.psbase.Scope.Options.EnablePrivileges = $true
  $objWMI.reboot()
  exit
 }
}

if($help)   { "Ausgeben der Hilfeinformationen ..." ; funhelp }
if($newName)
            {
             "Umbenennen des Computers $computer in $newName"
              FunRename


            }
if($reboot)
            {
             "Neustarten des Computers $computer ..."
             FunReboot
            }
if(!$help -or !$newname -or !$reboot)
           {
            "Sie müssen eine Aktion angeben ..."
             funhelp
           }
```

Verwalten von Windows Server 2008 Server Core

Mit WMI können Sie zahlreiche Verwaltungsaufgaben für einen Windows Server 2008 Server Core-Server ausführen. Beispielsweise können Sie mit dem Cmdlet **Get-WmiObject** die Datenträgerauslastung überprüfen. Der entsprechende Befehl hat folgende Syntax:

```
Get-wmiobject -class win32_volume -computername localhost
```

Führen Sie folgenden Befehl aus, um die CPU-Informationen des Servers anzuzeigen:

```
Get-wmiobject -class win32_processor -computername localhost
```

Der folgende Befehl zeigt an, welche Prozesse auf dem Server ausgeführt werden:

```
Get-wmiobject -class win32_process -computername localhost
```

Weiterführende Informationen Weitere Informationen zur Verwendung von Windows Management Instrumentation zum Verwalten von Windows-Servern finden Sie im Buch *Microsoft Windows Scripting with WMI: Self-Paced Learning Guide* (Microsoft Press, 2005). Das Buch wurde zwar speziell für Windows Server 2003 geschrieben, aber die behandelten Klassen sind auch in Windows Server 2008 vorhanden.

Das Ausführen der Befehle zum Überwachen von Servern lässt sich jedoch effizienter gestalten, wenn Sie hierzu ein Skript verwenden, beispielsweise das Skript *MonitorServer.ps1*.

Überwachen von Servern

Das Skript *MonitorServer.ps1* beginnt mit einer *param*-Anweisung zur Deklaration der beiden Parameter **-computer** und **-help**. Die *param*-Anweisung lautet:

```
Param($computer="localhost",[switch]$help)
```

Die nächste Funktion ist *funhelp()*, um wie üblich eine Hilfemeldung anzuzeigen, wenn das Skript mit dem Parameter **-help** ausgeführt wird. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funhelp()
{
 $helptext=@"
BESCHREIBUNG:
MonitorServer.ps1 führt einfache WMI-Abfragen aus.

PARAMETER:
-computer Gibt den Namen des Zielcomputers an.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
MonitorServer.ps1
Ermittelt Informationen über die Prozessoren, Prozesse,
Datenträger, Netzwerke und das BIOS auf dem lokalen Server.

MonitorServer.ps1 -computer Core
Ermittelt Informationen über die Prozessoren, Prozesse,
Datenträger, Netzwerke und das BIOS auf
einem Remotecomputer namens Core.

MonitorServer.ps1 -help
Zeigt das Hilfethema für dieses Skript an.

"@
$helptext
exit
}
```

Überprüfen Sie, ob die Variable *$help* vorhanden ist. Gibt es diese Variable, rufen Sie die Funktion *funhelp()* auf. Beispiel:

```
if($help)   { funhelp }
```

Das Skript umfasst eine neue Funktion. Um die Eingabe der WMI-Klassennamen zu vereinfachen, erstellen Sie eine Zeichenfolge, die alle WMI-Klassen enthält. Verwenden Sie Kommas, um die Namen der WMI-Klassen in der Zeichenfolge voneinander zu trennen. Codezeile:

```
$aryclass = "win32_processor,win32_process,win32_volume" + `
            ",win32_networkadapter,win32_bios"
```

Generieren Sie mit der statischen Methode *GetTempFileName()* der .NET Framework-Klasse *SystemIO.Path* einen temporären Dateinamen. Diese Methode erstellt einen Dateinamen, der auf das temporäre Verzeichnis verweist. Speichern Sie den von der Methode zurückgegebenen Dateinamen sowie den Pfad zum temporären Verzeichnis in der Variablen *$tempfilename*. Beispiel:

```
$tmpfilename = [io.path]::getTempFileName()
```

Sie müssen nun die Zeichenfolge mit den WMI-Klassennamen in ein Array umwandeln. Verwenden Sie hierzu die Methode *split()* der .NET Framework-Klasse *System.String*. Sie müssen keinen Wert angeben, da die Variable *$aryclass* bereits eine Zeichenfolge enthält und die Methode umgehend verfügbar ist. Anstatt das Array in einer separaten Variablen zu speichern, geben Sie die Methode *split()* direkt in der *foreach*-Anweisung an. Erstellen Sie die Variable *$class*, die als Enumerator verwendet wird. Codezeile:

```
foreach($class in $aryclass.split(","))
```

Verwenden Sie nun das Cmdlet **Get-WmiObject**, um unter Angabe der Variablen *$class* im Parameter **-class** und der Variablen *$computer* im Parameter **-computername** ein Verwaltungsobjekt zu erstellen. Fügen Sie das zurückgegebene Objekt in das Cmdlet **Format-List** ein und wählen Sie nur die Eigenschaften aus, die mit den Buchstaben *a* bis *z* beginnen. Übergeben Sie das Ergebnis an das Cmdlet **Out-File** und geben Sie den Dateipfad mit dem Parameter **-filepath** an. Geben Sie den Parameter **-append** an, um das Überschreiben der Ergebnisse zu verhindern. Zeigen Sie die Ergebnisse in Notepad an. Codeabschnitt:

```
 {
  Get-wmiobject -class $class -computername $computer |
  format-list [a-z]* |
  out-file -filepath $tmpfilename -append
 }

"Die Ergebnisse wurden in der Datei $tmpfilename gespeichert. Die Datei wird geöffnet ..."
Notepad $tmpfilename
```

Das vollständige Skript *MonitorServer.ps1* hat folgenden Inhalt:

MonitorServer.ps1

```
Param($computer="localhost",[switch]$help)

function funhelp()
{
 $helptext=@"
BESCHREIBUNG:
MonitorServer.ps1 führt einfache WMI-Abfragen aus.

PARAMETER:
-computer Gibt den Namen des Zielcomputers an.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
MonitorServer.ps1
Ermittelt Informationen über die Prozessoren, Prozesse,
Datenträger, Netzwerke und das BIOS auf dem lokalen Server.

MonitorServer.ps1 -computer Core
Ermittelt Informationen über die Prozessoren, Prozesse,
Datenträger, Netzwerke und das BIOS auf
einem Remotecomputer namens Core.

MonitorServer.ps1 -help
Zeigt das Hilfethema für dieses Skript an.

"@
$helptext
```

```
exit
}

if($help)   {   funhelp   }

$aryclass = "win32_processor,win32_process,win32_volume" + `
            ",win32_networkadapter,win32_bios"
$tmpfilename = [io.path]::getTempFileName()
foreach($class in $aryclass.split(","))
 {
  Get-wmiobject -class $class -computername $computer |
  format-list [a-z]* |
  out-file -filepath $tmpfilename -append
 }

"Die Ergebnisse wurden in der Datei $tmpfilename gespeichert. Die Datei wird geöffnet ..."
Notepad $tmpfilename
```

Abfragen von Ereignisprotokollen

Das Abfragen des lokalen Ereignisprotokolls ist einfach. Verwenden Sie hierzu das Cmdlet **Get-EventLog**:

```
Get-EventLog -LogName application |
Where-Object { $_.eventID -eq 25 }
```

Wenn Sie das Cmdlet **Get-EventLog** verwenden, müssen Sie lediglich den Protokollnamen angeben. Fügen Sie das zurückgegebene Objekt in das Cmdlet **Where-Object** ein, um beispielsweise nach der Ereignis-ID zu filtern. Sie können diese Methode jedoch nicht auf einem Remotecomputer ausführen. Um diese Hürde zu nehmen und den Zugriff auf einen Remotecomputer zu ermöglichen, müssen Sie die vom Cmdlet **Get-EventLog** verwendete .NET Framework-Klasse direkt einsetzen. Dies ist im Skript *QueryRemoteEventLog.ps1* veranschaulicht.

Das Skript *QueryRemoteEventLog.ps1* beginnt mit einer *param*-Anweisung, um die Parameter **-computer**, **-log**, **-id** und **-help** zu definieren. Der Parameter **-log** gibt den Namen des Ereignisprotokolls an und der Parameter **-id** enthält die zu ermittelnde Ereignis-ID. Die *param*-Anweisung lautet:

```
param(
      $computer=".",
      $log="system",
      $ID,
      [switch]$help
     )
```

Erstellen Sie nun wieder die Funktion *funhelp()*, um die Hilfe anzuzeigen. Diese Funktion erstellt die Variable *$helptext* und weist dieser eine *here*-Zeichenfolge zu. Die Funktion *funhelp()* zeigt dann den Inhalt der Variablen *$helptext* an und beendet die Skriptausführung. Die Funktion *funhelp()* ist wie folgt implementiert:

```
function funHelp()
{
$helpText=@"
BESCHREIBUNG:
NAME: QueryRemoteEventLog.ps1
Ruft das Ereignisprotokoll auf dem lokalen Computer oder auf einem Remotecomputer ab.

PARAMETER:
```

```
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-log      Das abzufragende Ereignisprotokoll: <application, system, security>.
-id       Die abzufragende Ereignis-ID.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
QueryRemoteEventLog.ps1

Zeigt eine Meldung an, die Sie darauf hinweist, dass Sie eine Aktion angeben müssen, und gibt die Hilfeinformationen
aus.

QueryRemoteEventLog.ps1 -computer MunichServer -log system -id 1002

Listet alle Ereignisse mit der ID 1002 (DHCP-Lease abgelaufen) aus dem
Systemereignisprotokoll auf dem Remoteserver MunichServer auf.

QueryRemoteEventLog.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}
```

Überprüfen Sie zuerst, dass die Variable *$help* vorhanden ist. Rufen Sie anschließend die Funktion *funhelp()* auf, wenn diese Variable existiert. Überprüfen Sie danach, ob die Variable *$id* vorhanden ist. Um im Ereignisprotokoll nach einem Ereignis zu suchen, benötigen Sie eine Ereignis-ID. Der Parameter *$id* ist deshalb erforderlich. Rufen Sie die Funktion *funhelp()* auf, wenn der Parameter nicht angegeben wurde. Codeabschnitt:

```
if($help) { funhelp }
if(!$id)   { "Der Parameter -ID wurde nicht angegeben." ; funhelp }
```

Erstellen Sie eine Instanz der .NET Framework-Klasse *System.Diagnostics.Eventlog* und geben Sie den Namen des Ereignisprotokolls sowie des Computers an, auf dem der Befehl ausgeführt werden soll. Diese beiden Werte bilden den Konstruktor zum Erstellen einer neuen Instanz der Klasse *EventLog*. Weisen Sie das neue *EventLog*-Objekt der Variablen *$objlog* zu. Rufen Sie anschließend die Methode *Get_Entries()* auf und fügen Sie das Ergebnis in das Cmdlet **Where-Object** ein. Suchen Sie im Codeblock des Cmdlets nach den Ereignis-IDs, die dem in der Befehlszeile für das Argument **-id** angegebenen Wert entsprechen. Codeabschnitt:

```
$objlog = New-Object system.diagnostics.eventLog($Log, $computer)
$objlog.get_entries() |
Where-object { $_.eventID -eq $id }
```

Das vollständige Skript *QueryRemoteEventLog.ps1* hat folgenden Inhalt:

QueryRemoteEventLog.ps1

```
param(
      $computer=".",
      $log="system",
      $ID,
      [switch]$help
     )

function funHelp()
```

```
{
$helpText=@"
BESCHREIBUNG:
NAME: QueryRemoteEventLog.ps1
Ruft das Ereignisprotokoll auf dem lokalen Computer oder auf einem Remotecomputer ab.

PARAMETER:
-computer Gibt den Namen des Computers an, für den das Skript ausgeführt werden soll.
-log      Das abzufragende Ereignisprotokoll: <application, system, security>.
-id       Die abzufragende Ereignis-ID.
-help     Zeigt dieses Hilfethema an.

SYNTAX:
QueryRemoteEventLog.ps1

Zeigt eine Meldung an, die Sie darauf hinweist, dass Sie eine Aktion angeben müssen,
und gibt die Hilfeinformationen aus.

QueryRemoteEventLog.ps1 -computer MunichServer -log system -id 1002

Listet alle Ereignisse mit der ID 1002 (DHCP-Lease abgelaufen) aus dem
Systemereignisprotokoll auf dem Remoteserver MunichServer auf.

QueryRemoteEventLog.ps1 -help

Zeigt das Hilfethema für dieses Skript an.

"@
$helpText
exit
}

if($help) { funhelp }
if(!$id)   { "Der Parameter -ID wurde nicht angegeben." ; funhelp }

$objlog = New-Object system.diagnostics.eventLog($Log, $computer)
$objlog.get_entries() |
Where-object { $_.eventID -eq $id }
```

Zusammenfassung

In diesem Kapitel wurden die Konfigurationsaufgaben für Windows Server 2008 Server Core beschrieben. Es wurden sowohl die Änderungen, die lokal an der Windows Firewall vorgenommen werden müssen, um eine Verbindung mit dem Server herstellen zu können, als auch die Schritte zum Hinzufügen des Servers zu einer Domäne erklärt. Außerdem wurden das Festlegen der IP-Adresse, der Subnetzmaske und des Standardgateways auf einem Server sowie das Konfigurieren der DNS-Serverinformationen, beispielsweise des Domänensuffixes und des primären DNS-Servers, sowie das Neustarten eines Remoteservers erklärt. Das Kapitel wurde mit einem Skript zum Verwalten von Windows Server 2008 Server Core abgeschlossen. Sie können mit Skripts WMI-Informationen abrufen und die Ereignisprotokolle durchsuchen.

ANHANG A

Cmdlet-Namenskonventionen

Die mit Windows PowerShell installierten Cmdlets entsprechen einer Standardnamenskonvention. Der Name setzt sich aus einem Verb/Substantiv-Paar zusammen. Beispielsweise beginnen vier Befehle mit dem englischen Verb *Add*: Add-Content, Add-History, Add-Member und Add-PSSnapin. Wenn Sie Cmdlets programmieren, sollten Sie ähnlichen Namenskonventionen folgen. Die Kenntnis der üblichen Namenskonventionen kann beim Erlernen von Windows PowerShell eine echte Hilfe sein.

In Tabelle A.1 sind die Anzahl, das Verb und jeweils einige Beispiele für Windows PowerShell-Cmdlets aufgeführt. Um eine vollständige Liste der Windows PowerShell-Cmdlets anzuzeigen, führen Sie den Befehl **get-command** aus.

Tabelle A.1 Cmdlet-Namenskonventionen

Anzahl	Verb	Cmdlets
4	Add	Add-Content, Add-History, Add-Member
4	Clear	Clear-Content, Clear-Item
1	Compare	Compare-Object
1	ConvertFrom	ConvertFrom-SecureString
1	Convert	Convert-Path
2	ConvertTo	ConvertTo-Html, ConvertTo-SecureString
2	Copy	Copy-Item, Copy-ItemProperty
4	Export	Export-Alias, Export-Clixml, Export-Console
1	ForEach	ForEach-Object
4	Format	Format-Custom, Format-List, Format-Table
29	Get	Get-Acl, Get-Alias
1	Group	Group-Object
3	Import	Import-Clixml, Import-Csv
3	Invoke	Invoke-Expression, Invoke-History
1	Join	Join-Path
2	Measure	Measure-Command, Measure-Object
2	Move	Move-Item, Move-ItemProperty
8	New	New-Alias, New-Item, New-ItemProperty
6	Out	Out-Default, Out-File, Out-Host,
1	Pop	Pop-Location
1	Push	Push-Location

Tabelle A.1 Cmdlet-Namenskonventionen *(Fortsetzung)*

Anzahl	Verb	Cmdlets
1	Read	Read-Host
5	Remove	Remove-Item, Remove-ItemProperty
2	Rename	Rename-Item, Rename-ItemProperty
1	Resolve	Resolve-Path
1	Restart	Restart-Service
1	Resume	Resume-Service
2	Select	Select-Object, Select-String
13	Set	Set-Acl, Set-Alias
1	Sort	Sort-Object
1	Split	Split-Path
3	Start	Start-Service, Start-Sleep, Start-Transcript
3	Stop	Stop-Process, Stop-Service, Stop-Transcript
1	Suspend	Suspend-Service
1	Tee	Tee-Object
1	Test	Test-Path
1	Trace	Trace-Command
2	Update	Update-FormatData, Update-TypeData
1	Where	Where-Object
7	Write	Write-Debug, Write-Error, Write-Host

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

A N H A N G B

Anbieternamen für ActiveX-Datenobjekte

Sie können mehrere Anbieter verwenden, um Datenbankverbindungen oder andere Datenquellen zu öffnen. Diese Anbieter sind in der Tabelle B.1 aufgeführt, die jedoch nicht vollständig ist, da viele Softwareunternehmen ihre eigenen Anbieter entwickeln.

Zum Öffnen einer Datenquelle können alle aufgeführten Anbieter verwendet werden. Geben Sie hierzu folgenden Code ein:

```
$strDB = "C:\FSO\ConfigurationMaintenance.mdb"
$objConnection = New-Object -ComObject ADODB.Connection
$objConnection.Open("Provider = Microsoft.Jet.OLEDB.4.0; `
  Data Source= $strDB")
```

In Kapitel 9 „Konfigurieren der Desktopeinstellungen“ finden Sie das Beispielskript *AuditScreenSaver-WriteToAccess.ps1*, das Microsoft ActiveX-Datenobjekte (ADO) für die Kommunikation mit einer Microsoft Access-Datenbank verwendet.

Die in Tabelle B.1 aufgeführten Anbieter stellen bei Verwendung von ADO eine Verbindung mit verschiedenen Datenquellen her.

Tabelle B.1 ADO-Anbieternamen

Anbietername	Anbieter
ADSDSOObject	Active Directory
Microsoft.Jet.OLEDB.4.0	Microsoft Jet-Datenbanken
MSDAIPP.DSO.1	Microsoft Internet Publishing
MSDAORA	Oracle-Datenbanken
MSDAOSP	Einfache Textdateien
MSDASQL	Microsoft OLE DB-Anbieter für ODBC
MSDataShape	Microsoft Data Shape
MSPersist	Lokal gespeicherte Dateien
SQLOLEDB	Microsoft SQL Server

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

ANHANG C

Häufig gestellte Fragen

Die in diesem Anhang beantworteten Fragen wurden mir während des letzten Jahres in Windows PowerShell-Kursen gestellt, die ich in zahlreichen Ländern abgehalten habe. Die Fragen und Antworten decken zahlreiche Themen ab und sparen Ihnen hoffentlich viel Zeit. Zu einigen Fragen gibt es mehrere Antworten, die alle richtig sind.

F. Wie viele Cmdlets sind in der Standardinstallation von Windows PowerShell verfügbar?

A. 129

F. Wie kann ich herausfinden, wie viele Cmdlets in der Standardinstallation von Windows PowerShell verfügbar sind?

A. Die folgenden drei Befehle geben das gleiche Ergebnis zurück:

```
(Get-Command).length
(Get-Command -CommandType cmdlet).count
Get-Command -CommandType cmdlet | foreach($_) { $i++ }
```

F. Was ist der Unterschied zwischen einer schreibgeschützten Variablen und einer Konstanten?

A. Der Inhalt einer schreibgeschützten Variablen kann nicht geändert werden. Sie können das Ändern des Inhalts jedoch mit dem Cmdlet **Set-Variable** und dem Parameter **-force** erzwingen. Mit dem Cmdlet **Remove-Variable** und dem Parameter **-force** können Sie die Variable löschen. Eine Konstante kann auch mit dem Parameter **-force** weder gelöscht noch geändert werden.

F. Was sind die drei wichtigsten Cmdlets?

A. Die drei wichtigsten Cmdlets sind **Get-Command**, **Get-Help** und **Get-Member**.

F. Welches Cmdlet kann ich für die Arbeit mit Ereignisprotokollen verwenden?

A. Das Cmdlet **Get-Eventlog**.

F. Wie finde ich dieses Cmdlet?

A. Führen Sie folgenden Befehl aus:

```
Get-Command -Noun eventlog.
```

F. Welche .NET Framework-Klasse wird vom Cmdlet Get-Eventlog verwendet?

A. Die Klasse *System.Diagnostics.EventLogEntry.*

F. Wie finde ich diese Informationen?

A. Führen Sie folgenden Befehl aus:

```
Get-Eventlog application | Get-Member
```

F. Was sind die leistungsfähigsten Anweisungen in Windows PowerShell?

A. Die *switch*-Anweisung ist die leistungsfähigste Anweisung, da sie mehrere Bedingungen überprüft und das Skript in die korrekte Richtung weist.

F. Für was wird das `*t* verwendet?

A. Dieses Zeichen repräsentiert Tabstoppzeichen.

F. Wie verwende ich das Zeichen `t in einem Skript, um einen Tabstopp zu erzeugen?

A. Geben Sie im Skript beispielsweise folgende Zeichen ein: "`tHallo".

F. Diese Syntax ist verwirrend. Was passiert, wenn ich zwischen dem t und dem H ein Leerzeichen einfüge?

A. Wenn Sie zwischen dem t und dem H ein Leerzeichen einfügen, enthält die Ausgabe nach dem Tabstopp ein zusätzliches Leerzeichen.

F. Muss ich für eine `*t*-Anweisung (beispielsweise "`tHallo:") die Groß-/Kleinschreibung beachten?

A. Ja. Diese Anweisung ist eine der wenigen Windows PowerShell-Anweisungen für die die Groß-/Kleinschreibung beachtet werden muss. Wenn Sie einen Großbuchstaben eingeben (THallo), werden die Buchstaben THallo in der Zeile angezeigt.

F. Wie führe ich ein Skript mit einem Leerzeichen im Pfad aus?

A. Verwenden Sie hierzu folgenden Code:

```
PS > c:\my`folder\myscript.ps1
PS> &("c:\my  folder\myscript.ps1")
```

F. Was ist die einfachste Methode zum Erstellen eines Arrays?

A. Die einfachste Methode zum Erstellen eines Arrays ist:

```
$array = "1","2","3","4"
```

F. Wie zeige ich einen berechneten Wert (MB anstatt Bytes) aus einer WMI-Abfrage an, wenn ich die Daten an das Cmdlet Format-Table übergebe?

A. Erstellen Sie eine Hashtabelle an der Position, an der Sie die Daten anzeigen möchten und führen Sie die Berechnung in geschweiften Klammern aus. Weisen Sie das Ergebnis dem Parameter **-expression** zu. Beispiel:

```
gwmi win32_logicaldisk -Filter "drivetype=3" | ft -Property name,
@{ Label="freespace"; expression={$_.freespace/1MB}}
```

F. Welcher Parameter des Cmdlets Get-WmiObject ersetzt eine WQL Where-Klausel?

A. Der Parameter **-filter**:

```
Get-wmiobject win32_logicaldisk -filter "drivetype  =  3"
```

F. Welcher Befehl am Anfang eines Skripts legt fest, dass Fehler ignoriert werden und die Skriptausführung fortgesetzt wird?

A. Der Befehl:

```
$erroractionpreference=SilentlyContinue
```

F. Wie kann ich nur das aktuelle Jahr anzeigen?

A. Um das aktuelle Jahr anzuzeigen, geben Sie folgende Codezeile ein: (Beachten Sie, dass yyyy durch die Jahreszahl ersetzt wird.)

```
Get-Date -format yyyy
```

F. Was ist die Windows PowerShell in 30 oder weniger Wörtern?

A. Windows PowerShell ist die nächste Generation der Befehlseingabe und Skriptsprache von Microsoft. Windows PowerShell kann VBScript und die Eingabeaufforderung in den meisten Fällen ersetzen.

F. Wie kann ich feststellen, dass diese Beschreibung aus 30 oder weniger Wörtern besteht?

A. Indem Sie folgenden Code ausführen:

```
$a = "Windows PowerShell ist die nächste Generation der Befehlseingabe und Skriptsprache von Microsoft. Windows
PowerShell kann VBScript und die Eingabeaufforderung in den meisten Fällen ersetzen."
Measure-Object -InputObject $a -Word
```

F. Was sind zwei Methoden für Active Directory-Abfragen in Windows PowerShell?

A. Sie können mit ADO eine LDAP-Abfrage oder eine SQL-Abfrage ausführen.

F. Wie kann ich den freien Speicherplatz auf einer Festplatte in MB mit zwei Dezimalstellen anzeigen?

A. Verwenden Sie folgenden Formatbezeichner:

```
"{0:n2}"-f ((gwmi win32_logicaldisk -Filter "drivetype='3'").freespace/1MB)
```

F. Wie kann ich die 2 in der Variablen $array = "1","2","3","4" durch 12 ersetzen?

A. Verwenden Sie hierzu folgenden Code:

```
$array=[regex]::replace($array,"2","12")
```

F. Ich habe folgende *switch*-Anweisung eingegeben. Wie kann ich verhindern, dass die letzte Zeile (Write-Host "switched") ausgeführt wird?

```
$a = 3
switch ($a) {
 1 { "Eins erkannt." }
 2 { "Zwei erkannt." }
}
Write-Host "switched"
```

A. Fügen Sie eine exit-Anweisung hinzu:

```
$a = 3
switch ($a) {
1 { "Eins erkannt." }
2 { "Zwei erkannt."}
DEFAULT { exit}
}
Write-Host "switched"
```

F. Wie gebe ich für einen WMI-Remoteaufruf andere Anmeldeinformationen ein, wenn ich das Cmdlet Get-WmiObject verwende?

A. Verwenden Sie hierzu den Parameter **-credential**:

```
Get-WmiObject Win32_BIOS -ComputerName Server01  -Credential (get-credential
Domain01@User01)
```

Oder verwenden Sie den Parameter **-credential** wie folgt:

```
$c = Get-Credential
Get-WmiObject Win32_DiskDrive -ComputerName Server01 -Credential $c
```

F. Wie kann ich eine Zufallszahl generieren?

A. Verwenden Sie die .NET Framework-Klasse *System.Random* und rufen Sie die Methode *Next()* auf:

```
([random]5).next()
```

F. Wie kann ich eine Zufallszahl zwischen 1 und 10 generieren?

A. Verwenden Sie die .NET Framework-Klasse *System.Random* und rufen Sie die Methode ***Next()*** auf:

```
([random]5).next("1","10")
```

F. Welche Windows PowerShell-Befehle unterstützen reguläre Ausdrücke?

A. Verwenden Sie das Cmdlet **Where-Object** mit dem Parameter **-match**:

```
get-process | where-object { $_.ProcessName -match "^p.*" }
```

Sie können auch eine *switch*-Anweisung mit dem Parameter **-regex** verwenden:

```
switch -regex ("Hi there") { "hi" { "found" } }
```

F. Wie kann ich für alle Befehle, die während einer Windows PowerShell-Sitzung eingegeben werden, eine Überwachungsdatei erstellen?

A. Verwenden Sie hierzu das Cmdlet **Start-Transcript**:

```
Start-transcript -Path c:\fso\mylog.txt -Force
```

F. Wie stelle ich fest, wie viele Sekunden das Abrufen von Objekten aus dem Anwendungsprotokoll dauert?

A. Geben Sie folgenden Befehl ein:

```
(Measure-Command { Get-EventLog application }).totalseconds
```

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

F. Wenn ich die Windows PowerShell-Konsole auf einem Exchange 2007-Server öffne, sind die Microsoft ExchangeCmdlets nicht verfügbar. Wodurch wird das Problem verursacht und wie kann ich dieses beheben?

A. Das Problem ist, dass die Exchange-Verwaltungsshell nicht geladen wurde. Geben Sie in der Windows PowerShell-Konsole folgenden Befehl ein:

```
add-psSnapin -Name microsoft.exchange.management.powershell.admin
```

F. Wie rufe ich eine Liste der Snap-Ins ab, die auf meinem Computer in Windows PowerShell registriert sind?

A. Geben Sie in der Windows PowerShell-Konsole folgenden Befehl ein:

```
Get-PSSnapin -Registered
```

F. Wie erstelle ich eine ASCII-Datei mit den Ergebnissen des Cmdlets Get-Process?

A. Fügen Sie die Ergebnisse in das Cmdlet **Out-File** ein und geben Sie *ASCII* mit dem Parameter **-encoding** an.

F. Ich habe gehört, dass das Cmdlet Write-Host die Ausgabe in verschiedenen Farben anzeigen kann. Welche Syntax muss ich hierzu verwenden?

A.Syntaxbeispiele:

```
write-host -ForegroundColor 12 "hi"
write-host -ForegroundColor 12 "hi" -BackgroundColor white
write-host -ForegroundColor blue -BackgroundColor white
write-host -ForegroundColor 2 hi
write-host -backgroundcolor 2 hi
write-host -backgroundcolor ("{0:X}" -f 2) hi
for($i=0 ; $i -le 15 ; $i++) { write-host -foregroundcolor $i "hi" }
```

F. Wie kann ich feststellen, ob ein Befehl erfolgreich ausgeführt wurde?

A. Fragen Sie die automatische Variable *$error* ab. Wenn *$error[0]* keine Informationen zurückgibt, sind keine Fehler aufgetreten. Sie können auch die automatische Variable *$?* abfragen. Wenn *$?* den Wert *True* hat, wurde der Befehl erfolgreich ausgeführt.

F. Wie trenne ich die folgende Zeichenfolge in der Variablen *$a* auf?

```
$a = "atl-ws-01,atl-ws-02,atl-ws-03,atl-ws-04"
```

A. Verwenden Sie die *split*-Methode:

```
$b = $a.split(",")
```

F. Wie verknüpfe ich das folgende Array in der Variablen *$a*?

```
$a = "h","e","l","l","o"
```

A. Verwenden Sie die statische Methode *join* aus der *String*-Klasse:

```
$b = [string]::join("", $a)
```

F. Wie erstelle ich einen Pfad zum Verzeichnis Windows\System32?

A. Verwenden Sie hierzu folgenden Code:

```
Join-Path -path (g-t-item env:\windir).value -ChildPa-h system32
```

F. Wie kann ich den Wert von %SYSTEMROOT% ausgeben?

A. Verwenden Sie hierzu folgenden Code:

```
(get-item Env:\systemroot).value
$env:systemroot
```

F. Ich muss die aktiven Prozesse anzeigen und die Ausgabe in einer Textdatei speichern. Welchen Code muss ich eingeben?

A. Verwenden Sie folgenden Code:

```
Get-process | Tee-Object -FilePath c:\fso\proc.txt
```

F. Wie zeige ich das ASCII-Zeichen für den ASCII-Wert 56 an?

A. Verwenden Sie folgenden Code:

```
[char]56
```

F. Ich möchte ein stark typisiertes Array vom Typ *System.Diagnostics.Processes* erstellen und der Variablen *$a* zuweisen. Welchen Code muss ich hierzu eingeben?

A. Verwenden Sie folgenden Code:

```
[diagnostics.process[]]$a=get-process
```

F. Wie kann ich die Zahl 1234 als Hexadezimalzahl anzeigen?

A. Verwenden Sie folgenden Code:

```
"{0:x}" -f 1234
```

F. Wie kann ich den Dezimalwert der Hexadezimalzahl 0x4d2 anzeigen?

A. Verwenden Sie folgenden Code:

```
0x4d2
```

F. Ich möchte herausfinden, ob eine Zeichenfolge den Buchstaben m enthält. Die Zeichenfolge ist der folgenden Variablen *$a* zugewiesen:

```
$a="northern hairy-nosed wombat"
```

A.Verwenden Sie folgenden Code:

```
[string]$a.contains("m")
$a.contains("m")
[regex]::match($a,"m")
([regex]::match($a,"m")).success
```

F. Wie kann ich einen Benutzer zur Eingabe auffordern?

A. Verwenden Sie das Cmdlet **Read-Host**:

```
$in = Read-host "Geben Sie die Daten ein."
```

F. Kann ich der Variablen *$input* den Inhalt des Cmdlets Read-Host zuweisen?

A. Die automatische Variable *$input* wird für Skriptblöcke in einer Pipeline verwendet und ist daher nicht geeignet. Nennen Sie die Variable *$userInput*, aber nicht *$input*.

F. Wie lege ich fest, dass das Skript einen Fehler generiert, wenn eine Variable nicht deklariert wurde?

A. Fügen Sie den Befehl **Set-PSDebug -strict** an beliebiger Stelle im Skript ein. Nicht deklarierte Variablen generieren dann einen Fehler, wenn auf diese zugegriffen wird.

F. Wie kann ich den vom Cmdlet Get-History belegten Puffer vergrößern?

A. Weisen Sie der automatischen Variablen *$MaximumHistoryCount* den gewünschten Wert zu:

```
$MaximumHistoryCount = 65
```

F. Wie geben ich die Zahl 1 als eine ganze Zahl eines 16-Bit-Arrays an?

A. Verwenden Sie folgenden Code:

```
$a=[int16[]][int16]1
```

F. Ich möchte das Anführungszeichen in der Zeichenfolge “Dies ''ist eine Zeichenfolge” angeben. Wie muss ich hierzu vorgehen?

A. Geben Sie vor dem Anführungszeichen ein Graviszeichen ein.

F. Wie entferne ich mit der Methode *replace* das Anführungszeichen, wenn die Zeichenfolge der Variablen *$arr* zugewiesen ist? Die Ausgabe soll wie folgt aussehen: “Dies ist eine Zeichenfolge.”

A. Verwenden Sie die Methode *replace* aus der .NET Framework-Klasse *System.String*:

```
$arr.Replace("`"","")
```

Sie können auch den ASCII-Wert des Anführungszeichens und die Methode *replace* aus der .NET Framework-Klasse *System.String* verwenden:

```
$arr.Replace([char]34,"")
```

F. Wie führe ich mit Invoke-Expression ein Skript in Windows PowerShell aus, wenn der Pfad Leerzeichen enthält?

A. Geben Sie vor den Leerzeichen ein Graviszeichen ein und setzen Sie den Pfad und Skriptnamen in einfache Anführungszeichen:

```
Invoke-Expression ('h:\LABS\extras\Run` With` Spaces.ps1')
```

F. Wie erstelle ich ein Array aus Bytewerten, das hexadezimale Werte enthält?

A. Verwenden Sie die *[byte]*-Einschränkung und beziehen Sie das Arrayzeichen (*[]*) ein: *[byte[]]*. Um eine Hexadezimalzahl anzugeben, verwenden Sie das 0x-Format:

```
[byte[]]$mac = 0x00,0x19,0xD2,0x72,0x0E,0x2A
```

F. Wie installiere ich die Microsoft Exchange-Verwaltungsshell auf meinem Windows Vista-Computer?

A. Sie können die Exchange-Verwaltungsshell nur auf Windows Server 2003 oder Windows Server 2008 installieren. Weitere Informationen finden Sie in der Microsoft-Hilfe und unter KB931903 auf der Supportwebsite.

ANHANG D

Skriptrichtlinien

In diesem Anhang werden die Skriptrichtlinien erklärt. Die Skriptrichtlinien wurden von mehreren Skriptentwicklern zusammengestellt. Die meisten Entwickler sind Mitarbeiter von Microsoft, die an der Entwicklung von Windows PowerShell beteiligt sind. Andere Entwickler sind Netzwerkadministratoren und Berater, die Windows PowerShell täglich einsetzen. Nicht jedes Skript hält alle Richtlinien ein. Wenn Sie sich jedoch an den Richtlinien orientieren, werden Ihre Skripts einfacher zu verstehen und zu verwalten sein. Außerdem können Sie die Gesamtkosten für die Skripterstellung reduzieren. Die drei wichtigsten Anforderungen an ein Skript sind, dass dieses einfach zu lesen, zu verstehen und zu verwalten ist.

Allgemeiner Skriptaufbau

In diesem Abschnitt wird der allgemeine Aufbau eines Skripts erklärt, einschließlich der Funktionen und anderer Aspekte.

Einbeziehen einer Funktion in das Skript, das die Funktion aufruft

Es ist zwar möglich, eine *Include*-Datei oder *dot source*-Funktion in Windows PowerShell zu verwenden, dies kann aber zu einem Support-Albtraum werden. Wenn Sie nicht wissen, in welchem Skript eine gewünschte Funktion erstellt wurde, müssen Sie unter Umständen nach der Funktion lange suchen. Das Skript, in dem die Funktion erstellt wurde, kann außerdem andere Elemente enthalten. Das Auffinden der Funktion in der Skriptdatei ist also schwierig. Außerdem müssen Sie die Namenskonventionen für Variablen beachten, da ansonsten möglicherweise Namenskonflikte auftreten. Wenn Sie eine *Include*-Datei verwenden, ist das Skript nicht mehr ohne die zusätzlichen Skriptdateien portabel. In diesem Fall sind die verwendeten Funktionsbibliotheken immer erforderlich.

Skripts, die alle Funktionen direkt beinhalten, sind einfacher zu lesen und zu verwalten. Wenn Sie die Funktionen in separaten Dateien speichern und *dot source* verwenden, werden zwei Anforderungen an die Skripterstellung nicht erfüllt.

Wenn ein Skript auf ein externes Skript mit verwendeten Funktionen verweist, können die entsprechend referenzierten Funktionen aus den externen Skripts nicht entfernt werden. Wenn Sie eine referenzierte Funktion ändern, wissen Sie möglicherweise nicht, wie viele Skripts diese Funktion aufrufen und wie sich die geänderte Funktion auf diese Skripts auswirkt. Sollte eine Funktion von nur einem Skript aufgerufen werden, kopieren Sie die Funktion einfach in die Skriptdatei.

Vollständige Cmdlet-Namen und Parameternamen

Das Ausschreiben von Cmdlet-Namen und die Vermeidung von Aliasen in Skripts hat mehrere Vorteile. Ein Vorteil ist, dass das Skript einfacher zu lesen ist. Außerdem ist das Skript stabiler, wenn der Benutzer Aliasänderungen vornimmt, und kompatibel mit künftigen Windows PowerShell-Versionen.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Verwenden von Aliasen

In Windows PowerShell sind drei Aliastypen verfügbar: Kompatibilitätsalias, kanonischer Alias und benutzerdefinierter Alias.

Sie können den Kompatibilitätsalias mit folgendem Befehl identifizieren:

```
Get-childitem alias: |
where-object {$_.options -notmatch "Readonly" }
```

Der Kompatibilitätsalias ermöglicht in Windows PowerShell den nahtlosen Übergang von älteren Befehlsshells. Sie können einen Kompatibilitätsalias mit folgendem Befehl entfernen:

```
Get-childitem alias: |
where-object {$_.options -notmatch "Readonly" } |
remove-item
```

Kanonische Aliase vereinfachen die Verwendung der Windows PowerShell-Cmdlets in der Windows PowerShell-Konsole. Kanonische Aliase sind kurz und können schnell eingegeben werden. Führen Sie folgenden Befehl aus, um kanonische Aliase zu identifizieren:

```
Get-childitem alias: |
where-object {$_.options -match "Readonly" }
```

Verwenden Sie in einem Skript ausschließlich kanonische Aliase

Die Verwendung kanonischer Aliase in einem Skript ist relativ ungefährlich. Das Skript ist jedoch schwieriger zu lesen und für das gleiche Cmdlet sind möglicherweise mehrere Aliase vorhanden, die von verschiedenen Benutzern definiert wurden. Obwohl kanonische Aliase schreibgeschützt sind, können diese entfernt oder geändert werden, wenn der Benutzer den jeweiligen Alias mit einer anderen Bedeutung neu definiert.

Geben Sie beim Erstellen eines Aliases stets die Eigenschaft *Description* an

Wenn Sie einen Alias zu Ihrem Profil hinzufügen, können Sie die Option *Readonly* aktivieren. Sie sollten außerdem die Eigenschaft *Description* für Ihre persönlichen Aliase stets mit einer konsistenten Beschreibung versehen. Hier ein Beispiel aus meinem persönlichen Windows PowerShell-Profil:

```
New-Alias -Name gh -Value Get-Help -Description "mred alias"
New-Alias -Name ga -Value get-alias -Description "mred alias"
```

Verwenden Sie Get-Item, um Pfadzeichenfolgen zu konvertieren

Wenn Sie mit Dateien arbeiten, können Sie mit dem Cmdlet **Get-Content** Dateiinhalte auslesen. Wenn Sie jedoch das Cmdlet **Get-Item** verwenden, erhalten Sie ein Objekt mit den Dateieigenschaften und -methoden. Dieses Feature ist im folgenden Beispiel veranschaulicht:

```
$files = Get-Content "filelist.txt" |
Get-Item $files |
Foreach-object { $_.Fullname }
```

Lesbarkeit von Skripts

Sie können auf mehrere Arten sicherstellen, dass Ihre Skripts lesbar sind. In diesem Abschnitt werden einige der wichtigeren Elemente erklärt.

- Wenn Sie einen Alias erstellen, geben Sie den Parameter **-description** an, mit dem Sie Ihre persönlichen Aliase ermitteln können. Beispiel:

```
Get-Alias |
where-object { $_.description -match 'mred' } |
Format-Table -Property "   ",name, definition -autosize `
             -hideTableHeaders
```

- Ihre Skripts sollten den Parameter **-help** unterstützen und die Hilfe ausgeben. Sie können **-help** als üblichen Parameter wie folgt implementieren:

```
Param($help)
```

- Der Parameter **-help** kann auch als fester Parameter implementiert werden:

```
Param([switch]$help)
```

- Alle Prozeduren sollten mit einer kurzen Beschreibung beginnen. Die Beschreibung sollte keine Details enthalten, da sich diese oft ändern und deshalb zu falschen Angaben führen können.
- Die an eine Funktion übergebenen Argumente sollten beschrieben werden, wenn deren Verwendungszweck nicht offensichtlich ist und die Funktion erwartet, dass die Argumente in einem bestimmten Bereich liegen.
- Die Rückgabewerte für Variablen, die von einer Funktion geändert werden, sollten ebenfalls am Anfang der Funktion beschrieben werden.
- Jede Variablendeklaration sollte einen Kommentar enthalten, der die Verwendung der Variablen beschreibt.
- Variablen und Funktionen sollte ein beschreibender Namen zugewiesen werden, um sicherzustellen, dass Kommentare ausschließlich für komplexe Funktionen erforderlich sind.
- Wenn Sie eine komplexe Funktion erstellen, die aus mehreren Codeblöcken besteht, fügen Sie nach jeder abschließenden geschweiften Klammer (}) einen Kommentar ein.
- Fügen Sie am Anfang des Skripts eine Beschreibung der wichtigen Objekte und Cmdlets sowie bestimmter Anforderungen für das Skript ein.
- Halten Sie sich beim Benennen von Funktionen an das Verb/Substantiv-Format der Cmdlets, aber geben Sie keinen Bindestrich ein. Auf diese Art können Sie Funktionen und Cmdlets einfach unterscheiden. Außerdem vermeidet dies Verwirrung, wenn beispielsweise Tabstopps für ein Cmdlet funktionieren und für ein anderes nicht.
- Ein Skript sollte benannte Parameter verwenden, wenn mehr als ein Argument akzeptiert wird. Wenn ein Skript nur ein Argument akzeptiert, können Sie ein unbenanntes Argument angeben.
- Gehen Sie immer davon aus, dass Benutzer Ihr Skript kopieren und an ihre Anforderungen anpassen möchten. Kommentieren Sie den Code, um dies zu unterstützen.
- Gehen Sie nie vom aktuellen Pfad aus. Geben Sie immer den vollständigen Pfad in einer Umgebungsvariablen oder einen expliziten Pfad an.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Formatieren des Codes

Die Ausgabe sollte nicht zu viel Platz auf dem Bildschirm einnehmen, aber die Codeformatierung muss logisch struktiert sein. Hier einige Vorschläge:

- Rücken Sie geschachtelte Standardblöcke zwei Leerzeichen ein.
- Blenden Sie die Kommentare für eine Funktion aus.
- Blenden Sie die Anweisungen der höchsten Ebene aus und rücken Sie geschachtelte Blöcke zwei Leerzeichen ein.
- Richten Sie die geschweiften Klammern aus, um den Code übersichtlicher zu gestalten.
- Vermeiden Sie einzeilige Anweisungen, damit fehlende geschweifte Klammern einfacher zu finden sind.
- Brechen Sie Pipeline-Objekte an der Pipe um. Lassen Sie alle Pipes an der rechte Seite.
- Vermeiden Sie die Zeilenfortsetzung (das Graviszeichen). Eine Ausnahme sind Zeilen mit mehr als 90 Zeichen, damit der Benutzer zum Lesen des Codes den Bildlauf nicht ausführen muss.
- Verwenden Sie für lange Variablennamen eine Schreibweise mit Großbuchstaben in der Mitte der Variablennamen, z.B. VariablenName.
- Verwenden Sie das Cmdlet **Write-Progress** für Skripts, deren Ausführung mehr als zwei Sekunden dauert.
- Sie sollten die Parameter **-whatif** und **-confirm** sowohl in Ihren Funktionen als auch in Ihren Skripts unterstützen, insbesondere wenn diese den Systemstatus ändern. Beispiel für den Parameter **-whatif**:

```
param(
      [switch]$whatif
     )

function funwhatif()
 {
  "whatif: Ausführen des Vorgangs xxxx"
 }

if($whatif)
  {
   funwhatif #Aufrufen der Funktion funwhatif().
  }
```

- Wenn Ihr Skript nur eine bestimmte Anzahl von Argumenten akzeptiert, können Sie den Wert von *$args.count* überprüfen und die Funktion *help* aufrufen, wenn eine falsche Anzahl an Argumenten angegeben wurde. Beispiel:

```
if($args.count -ge 0)
  {
   "Falsche Anzahl an Argumenten."
   Funhelp  #Aufrufen der Funktion funhelp().
  }
```

- Wenn Ihr Skript keine Argumente akzeptiert, verwenden Sie folgenden Code:

```
If($args -ge 0) { funhelp }
```

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Arbeiten mit Funktionen

Funktionen ermöglichen in Windows PowerShell das Einkapseln von Codesegmenten, beispielsweise um die Funktionen von Windows PowerShell zu erweitern oder das Skript übersichtlicher zu gestalten. Hier einige Richtlinien für die Arbeit mit Funktionen:

- Funktionen sollten erforderliche Parameter überprüfen. Beispiel:

```
Function GetProcess ($name = ($paramMissing=$true))
 {
  if($local:paramMissing)
    {
     throw "VERWENDUNG: GetProcess Name <name>"
    } #$local:paramMissing
  Get-Process -name $name
 } #Ende der Funktion GetProcess()
```

- Sie können Dienstprogramme oder Funktionen in gemeinsam verwendeten Funktionsbibliotheken zusammenfassen und anschließend in Skripts übernehmen. Der Dateiname sollte dem Format *Library-<Substantiv oder Featurename>.ps1* entsprechen. Beispiel:

```
. c:\lib\Library-WmiFunctions.ps1
```

- Wenn Sie ein Funktionsbibliothek-Skript schreiben, verwenden Sie eindeutige Feature- und Parametervariablennamen, um Konflikte mit anderen Variablen im Skript zu vermeiden.
- Sie sollten den Parameter **-erroraction** unterstützen. Dieser Parameter vereinfacht die Übergabe eines Parameters, wenn die Funktion aufgerufen wird. Beispiel:

```
function getProcess (
                     $name,
                     $ErrorAction=$ErrorActionPreference
                     )
{
 $private:ErrorActionPreference = $ErrorAction
 Get-Process -Name $name
 "Die verwendete Fehleraktion ist $ErrorActionPreference" #debug
} #Ende von getProcess()

getProcess -name notepad -ErrorAction "stop"
```

- Sie sollten den Parameter **-verbose** unterstützen. Dieser Parameter ermöglicht zwei Ebenen für die Ausgabe von Verarbeitungsinformationen. Beispiel:

```
Function GetProcess
                   (
                    $name,
                    [switch]$verbose
                    )
{
 If($verbose)
   {
     Get-Process -Name $name |
     Format-List *
   }
 ELSE
   {
```

```
    Get-Process -Name $name
   }
} #Ende von getprocess()

getprocess -name notepad -verbose
```

- Sie sollten den Parameter **-confirm** implementieren, wenn Sie den Systemstatus ändern. Beispiel:

```
Function StopProcess
                   (
                    $name,
                    [switch]$confirm
                    )
{
 If($confirm)
   {
    $response = Read-Host -Prompt `
   "Möchten Sie  $name wirklich beenden?
    < j(a)  n(ein) >
   "
   switch($response)
    {
     "j" {
          Stop-Process -Name $name
         }
      "n" {
          "$name wird nicht beendet."
         }
    }

   }
 ELSE
   {
    Stop-Process -Name $name
   }
} #Ende von getprocess()

stopprocess -name notepad -confirm
```

Erstellen von Vorlagendateien

Sie können Vorlagen für verschiedene Skripttypen erstellen, beispielsweise WMI-, ADSI- und ADO-Skripts. Einige Vorschläge zum Erstellen von Vorlagen:

- Fügen Sie allgemeine Funktionen hinzu, die Sie regelmäßig verwenden.
- Verbindungszeichenfolgen sollten nicht hart codiert werden, beispielweise Servernamen, Dateipfade, usw. Weisen Sie diese Werte stattdessen mit Hilfe von Variablen zu.
- Die Versionsinformationen sollten nicht hart codiert werden.
- Fügen Sie Kommentare ein, wenn die Vorlage geändert werden muss.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Erstellen von Funktionen

Einige Vorschläge zum Erstellen von Funktionen:

- Erstellen Sie spezialisierte Funktionen. Optimale Funktionen führen nur einen Vorgang aus.
- Stellen Sie sicher, dass die Funktionen in sich geschlossen sind. Optimale Funktionen sind portabel.
- Alphabetisieren Sie die Funktionen in Ihrem Skript, um die Lesbarkeit und Verwaltbarkeit zu verbessern.
- Geben Sie Ihren Funktionen beschreibende Namen, beispielsweise *funhelp, funline* oder *funcomputepercentage*. Ich beginne meine Funktionen mit *fun*, um die Verwendung eines Schlüsselworts zu vermeiden und die Namen hervorzuheben. Sie können das Wort *function* auch ausschreiben, aber dann müssen Sie mehr tippen.
- Jede Funktion sollte einen einzigen Einsprungpunkt haben.
- Jede Funktion sollte einen einzigen Ausgabepunkt haben.
- Verwenden Sie Parameter, um Probleme mit lokalen und globalen Variablenbereichen zu vermeiden.
- Implementieren Sie allgemeine Parameter: **-verbose**, **-debug**, **-whatif** und **-confirm**.

Erstellen und Benennen von Variablen und Konstanten

Einige Vorschläge zum Erstellen und Benennen von Variablen und Konstanten:

- Vermeiden Sie Zufallszahlen (Magic Number). Sie sollten beim Aufrufen von Methoden oder Funktionen keine hart codierten Ziffern eingeben. Erstellen Sie eine Konstante mit einem beschreibenden Namen, damit der Benutzer weiß, welche Aktion der Code ausführt. Im Skript *ServiceDependencies.ps1* wird beispielsweise die Ausgabe mit einer Zahl ausgeglichen. Die Zahl wird von der Position eines bestimmten Zeichens in der Ausgabe bestimmt. Anstatt +14 einzugeben, erstellen Sie eine Konstante mit einem beschreibenden Namen. Weitere Informationen zu diesem Skript finden Sie in Kapitel 12 „Behandeln von Windows-Problemen.“ Der betreffende Code lautet:

```
New-Variable -Name c_padline -value 14 -option constant
Get-WmiObject -Class Win32_DependentService -computername $computer |
Foreach-object `
 {
  "=" * ((([wmi]$_.dependent).pathname).length + $c_padline)
```

- Sie sollten Variablen nicht wiederverwenden. Variablen sollten nur einen einzigen Verwendungszweck haben.
- Geben Sie Variablen beschreibende Namen.
- Minimieren Sie den Variablenbereich. Wenn Sie eine Variable in einer Funktion verwenden, deklarieren Sie die Variable in dieser Funktion.
- Wenn Sie eine Konstante benötigen, verwenden Sie stattdessen eine schreibgeschützte Variable. Beachten Sie, dass Konstanten weder gelöscht noch deren Werte geändert werden können.
- Vermeiden Sie hart codierte Werte in Methodenaufrufen. Weisen Sie die Werte stattdessen Variablen zu.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

- Wenn möglich, gruppieren Sie Variablen in einem Skriptabschnitt.
- Vermeiden Sie ungarische Notationen. Beachten Sie, dass alles in Windows PowerShell ein Objekt ist. Deshalb ist der Variablenname *$objWMI* praktisch nicht vielsagend.
- Die Verwendung der Abkürzungen *bln, int, dbl, err, dte* und *str* ist manchmal sinnvoll, da die Windows PowerShell eine stark typisierte Sprache ist, auch wenn sie sich nicht so verhält.
- Skripts sollten den globalen Variablenbereich nicht belegen. Übergeben Sie die Werte stattdessen als Referenz *[ref]* an eine Funktion.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

ANHANG E

Allgemeine Tipps zur Problembehandlung

In diesem Anhang finden Sie Tipps zum Beheben allgemeiner Fehler.

Denken Sie daran, dass die Rechtschreibung wichtig ist.

Überprüfen Sie zuerst immer, ob beispielsweise die Namen von Cmdlets, Eigenschaften und Methoden richtig geschrieben wurden. Windows PowerShell generiert keinen Fehler, wenn ein Eigenschaftenname in einem Skript einen Tippfehler enthält. Folgender Code generiert keine Ausgabe, zeigt aber auch nicht an, dass Sie eine falsche Eigenschaft für die WMI-Klasse *Win32_Service* angegeben haben.

```
PS C:\> $wmi = Get-WmiObject -Class win32_service
PS C:\> $wmi.badproperty
PS C:\>
```

Unterbrechen Sie die Pipeline nicht.

Dieser Fehler tritt häufig auf. Um einen in der Windows PowerShell-Konsole eingegebenen Befehl zu erweitern, fügen Sie ein Pipeline-Zeichen ein. Wenn Sie den Befehl in ein Skript einbeziehen möchten, fügen Sie einen Spaltenheader für die Ausgabe hinzu. Dies ist im folgenden Code veranschaulicht. Die Zeile mit der **Get-WmiObject**-Anweisung wird mit einem Pipeline-Zeichen beendet und danach wird eine Funktion aufgerufen, um den Namen des Computers auszugeben. Das Problem ist, dass die Pipeline unterbrochen wurde. Das Skript endet mit der Zeile "Dienstabhängigkeiten auf dem Computer." Da Sie eine Funktion aufrufen, wird kein Fehler generiert.

```
Param($computer = "localhost")

function funline ($strIN)
{
 $num = $strIN.length
 for($i=1 ; $i -le $num ; $i++)
  { $funline = $funline + "=" }
    Write-Host -ForegroundColor yellow $strIN
    Write-Host -ForegroundColor darkYellow $funline
}


Get-WmiObject -Class Win32_DependentService -computername $computer |
funline("Dienstabhängigkeiten auf dem Computer $($computer)")
Foreach-object `
 {
  [wmi]$_.Antecedent
  [wmi]$_.Dependent
 }
```

Rufen Sie allerdings keine Funktion auf, wird ein Fehler generiert, wenn Sie die Zeile mit dem **Get-WmiObject**-Befehl wie im vorherigen Codebeispiel mit einem Pipeline-Zeichen. Unterbrecehen Sie beispielsweise die Pipeline, indem Sie die Zeichenfolge "Dienstabhängigkeiten auf dem lokalen Computer." ausgeben.

```
Get-WmiObject -Class Win32_DependentService |
 "Dienstabhängigkeiten auf dem lokalen Computer."
Foreach-object `
 {
  [wmi]$_.Antecedent
  [wmi]$_.Dependent
 }
```

Dieser Code generiert einen Fehler. Die Fehlermeldung gibt an, dass der Ausdruck in der Pipeline unzulässig ist.

```
Ausdrücke sind nur als Erstes Element einer Pipeline zulässig.
Bei C:\Users\EDWILS~1.NOR\AppData\Local\Temp\temp.ps1:4 Zeichen:44
+  "Dienstabhängigkeiten auf dem lokalen Computer."
```

Verwenden Sie debug, um das Skript zu überprüfen.

Wenn ein Skript unerwartete Ergebnisse erzeugt, geben Sie den Wert der Variablen aus. Sie können nach der Variablen einen *debug*-Kommentar einfügen, um die Debuganweisungen nach dem Testen des Skripts bequemer entfernen zu können. Fügen Sie im folgenden Beispielskript zwei Werte hinzu und stellen Sie sicher, dass die Ergebnisse korrekt ausgegeben werden. Verwenden Sie hierzu Debuganweisungen, um zu bestätigen, dass die Ausgabe richtig ist. Löschen Sie anschließend die Codezeilen, die die Debuganweisungen enthalten. Bereinigen Sie das Skript mit Suchen und Ersetzen in Notepad. Code:

```
$a = 5
$b = 4
'$a is ' + $a # debug
'$b is ' + $b # debug
$c = $a + $b
"Das Ergebnis von`$a + `$b ist $c"
```

Bestätigen Sie mit dem Cmdlet Test-Path, dass eine Datei oder ein anderes Objekt vorhanden ist, wenn Sie dieses verwenden möchten.

Fügen Sie Debuganweisungen nach der Anweisung ein, wenn diese kein wesentlicher Bestandteil des Skripts ist. Der folgende Code zeigt ein Beispiel der *Test-Path*-Methode:

```
$script = "c:\fso\mydebugscript.ps1"
Test-Path $script # debug
$debug = "# debug"

switch -regex -file $script
{
 "debug" { $switch.current }
}
```

Initialisieren Sie Variablen und legen Sie die Werte auf $null oder 0 fest.

Wenn Sie Elemente unter Verwendung von Variablen überprüfen, können die Werte der Variablen zu unerwarteten Ergebnissen führen, wenn die Variablen nicht richtig initialisiert wurden. Das folgende Skript *ParseAppLog.ps1* veranschaulicht diesen Sachverhalt an einem Beispiel. Das Skript *ParseAppLog.ps1* befindet sich auf der Begleit-CD im Ordner *\Scripts\Extras*.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

ParseAppLog.ps1

```
$tcp=$udp=$dns=$icmp=$PdnsServer=$SdnsServer=$web=$ssl=$null

$fwlog = get-content "C:\Windows\system32\LogFiles\Firewall\firewall.log"
switch -regex ($fwlog)
 {
  "65.53.192.15" { $PdnsServer+=1 }
  "65.53.192.14" { $SdnsServer+=1 }
  "tcp" { $tcp+=1 }
  "udp" { $udp+=1 }
  "icmp" { $icmp+=1 }
  "\s53"   { $dns+=1 }
  "\s80"   { $web+=1 }
  "\s443"   { $ssl+=1 ; $switch.current}
 }

"`$PdnsServer $Pdnsserver"
"`$SdnsServer $SdnsServer"
"`$tcp $tcp"
"`$udp $udp"
"`$icmp $icmp"
"`$dns $dns"
"`$web $web"
"`$ssl $ssl"
```

Geben Sie in $erroractionpreference die Aktion an, die ausgeführt werden soll, wenn Daten mit Write-Error in einem Skript ausgegeben werden.

Überprüfen Sie Ihre Skripts auf die Anweisung *$erroractionpreference = "SilentlyContinue"*. Windows PowerShell zeigt beim Auftreten eines Fehlers standardmäßig eine Fehlermeldung an. Um die Verarbeitung ohne Fehlermeldung fortzusetzen, legen Sie den Wert der automatischen Variablen *$erroractionpreference* auf *SilentlyContinue* fest.

Überprüfen Sie die Fehlerobjekte mit $error.

Das *$error*-Objekt enthält Einträge für alle Fehler, die während einer Windows PowerShell-Sitzung auftreten. Beispiel:

```
$erroractionpreference = "SilentlyContinue"
$a = New-Object foo #Generiert einen Fehler.
$b = New-Object bar #Generiert einen weiteren Fehler.
 if ($error.count -eq 1)
    {"Es wurde 1 Fehler festgestellt."}
    else
    {"Es wurden " + $error.count + " Fehler festgestellt"}

for ($i = 0 ; $error.count ; $i++)
   {$error[$i].CategoryInfo
   $error[$i].ErrorDetails
   $error[$i].Exception
   $error[$i].FullyQualifiedErrorId
   $error[$i].InvocationInfo
   $error[$i].TargetObject}
```

Mit Set-PSDebug können Sie das Skriptdebuggen aktivieren und deaktivieren sowie in den Strict-Modus wechseln.

Beispiel:

```
C:\PS>set-psdebug -step; foreach ($i in 1..3) {$i}
```

Dieser Befehl aktiviert das Stepping und zeigt die Zahlen *1, 2* und *3* an.

```
DEBUG:1+ Set-PsDebug -step; foreach ($i in 1..3) {$i}
Vorgang fortsetzen?
1+ Set-PsDebug -step; foreach ($i in 1..3) {$i}
[J] Ja  [A] Ja, alle  [N] Nein  [K] Nein, keine  [H] Anhalten  [?] Hilfe
 (Der Standardwert ist "J"):a
DEBUG:1+ Set-PsDebug -step; foreach ($i in 1..3) {$i}
1
2
3
```

Denken Sie daran, dass nicht alle Objekte gleich beschaffen sind.

Auch wenn ein altes COM-Objekt die Methode *Create()* unterstützt, heißt das nicht, dass diese Methode in Windows PowerShell verfügbar ist.

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Stichwortverzeichnis

.NET Framework
  System.Diagnostics.EventLog-Klasse von 66
  system.string aus 123
  Zertifikatspeicher und 455

A

abgekürzte Namen für Cmdlets
  erstellen 15
AcceptPause.ps1-Skript 87
AcceptStop-Eigenschaft 93
Access 2007
  beendete Dienste in 85
  Bericht für überwachte Freigaben 122
  Bildschirmschonerinformationen in 237
  Datei mit durch Kommas getrennten Werten (.csv) und 78
  Dienstdokumentation in 80, 84
  Dienstkonfigurationen in 86
  Domänenbenutzerattribute und 353
  Druckerinventur in 142
  Freigabedokumentation in 118
  Laufwerkinformationen 162, 165
  PhyDisk-Tabelle in 163
  Speicherplatzbelegung auf 178, 181
action-Parameter 477, 490, 532–533
Active Directory-Benutzer und -Computer (ADUC) 353–355, 357–358
Active Directory-Dienstschnittstelle (ADSI) 276, 278, 280, 283, 349, 356
ActiveX-Datenobjekte (ADO) 80, 82, 118, 143, 179, 240, 367
Adaptereinstellungen für Netzwerke 205, 218
  Dynamic Host Configuration Protocol (DHCP) in 215
  Excel-Arbeitsblatt für 206, 208
  mehrere ermitteln 205
  Problembehandlung 343, 345
  statische IP-Adresse 211, 214
  verbunden 209–210
Adaptereinstellungen für Netzwerkkonfigurationen 196, 199
AdapterType-Eigenschaft 205, 207
Add-Content-Cmdlet 140
addDHCP-Parameter 533
addnew-Methode 82, 164
AddNodeEvictNode.ps1-Skript 395, 398
add-Parameter 395, 398
AddUserToGroup.ps1-Skript 365
Administrative Freigaben 117–118
Adresse-Registerkarte, Domänenbenutzer 355
ADSI-Attribut AccountDisabled 361
A-Einträge 514
Aktives Energieschema 245
Aliase
  für Cmdlets 14
  für Datentypen 48
all-Argument 42
allgemeine Protokolle
  Einträge abrufen 60
  mehrere 59
allow-Parameter 473, 494
all-Parameter 385
AllSigned-Ausführungsrichtlinie 33
Alternative Anmeldeinformationen 539
Angriffsfläche verringern 104
Anmeldeinformationen 236
Anweisungen für die Entscheidungsfindung 44
Anwendungspools
  erstellen 427–428
  IIS-Konfiguration abrufen 411, 416
Anwendungsprotokollereignis 262–263
a-Parameter 266, 274, 276, 331
append-Parameter 388–389
appname-Parameter 427
ArgGetMultipleServices.ps1-Skript 91
ArgsShare.ps1-Skript 149
Argumente
  computer 174
  constant 313
  default 160–161
  example 13
  file 55
  filter 93
  foregroundcolor 59
  full 14
  groupby 137–138
  help 42, 67, 250
  id 567
  list 59
  logname 64
  match 64
  nonewline 180
  path 56
  query 250
  Skripts und 35
  wildcard 69
ausführbare Datei, aufrufen 3
Ausführungsrichtlinie für Skripts 33
Authentifizierung, digitale Signaturen für 339
Automatische Dienste 323
Automatische Objekte 516
Automatischer Startmodus 86
Automatisches Anzeigen von Variablen 202
Automatisierung 40
AutoServicesNotRunning.ps1-Skript 324

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

autosize-Parameter 19
Datenverwaltung 304
Freigabeverwaltung 108
IIS-Verwaltung 413
Problembehandlung 334
Systemwiederherstellung 317
AutoStart-Eigenschaft 413–415, 428

B

backgroundcolor-Parameter 144
BackupFolderToServer.ps1-Skript 299
Bandbreite 417, 419
Befehlsshell und Skriptsprache, PowerShell 3
Befehlszeilenargumente
auswerten 42
Dienstkonfiguration festlegen 91
Netzwerk 150
Netzwerkdienstkonfiguration 506
Rückgabecode umwandeln 126
SetEventLogRegention Policy.ps1-Skript 66
Skript akzeptiert 37
Systemwiederherstellung 313
Terminaldienstekonfiguration 488
verwenden 51
Befehlszeilenparameter
Clusterdienst 383, 398
Netzwerkverwaltung 211
Terminaldiensteverwaltung 481, 485
Windows Server 2008 Server Core 545, 547, 550
Befehlszeilenprogramm, Windows-Ereignis 69
begin-Parameter 37–38
Benutzerfreigaben 124
Bereich, Variable 309
Berichte
Access 2007 Designer für 142–143
IIS-Konfiguration
virtuelle Verzeichnisse 420–421
Speicherplatz 181–182
Bindung, Position 14
Bold-Eigenschaft 206
breite Formatierung 21, 23

C

CallFunctionLib.ps1-Skript 552–553
Capacity-Eigenschaft 177, 179
Caption-Eigenschaft 202
cert-Parameter 461
ChangeLogSettings-Funktion 67
ChangeModeThenStart.ps1-Skript 98–99
change-Parameter 506, 508, 519, 521
CheckServiceThenStart.ps1-Skript 96–97
CheckServiceThenStop.ps1-Skript 93
CheckSignedDeviceDrivers.ps1-Skript 339
CheckStoppedServices.ps1-Skript 102–103
CIM (Common Information Model) 374
class-Parameter
Clusterdienst 377–378, 381
Datenverwaltung 304
Desktopverwaltung 162, 170, 172
IIS-Verwaltung 421
Netzwerkdienstekonfiguration 502–503
Terminaldiensteverwaltung 485, 488
Windows Server 2008 Server Core 565
Clear-Host-Cmdlet 516
Clusterdienst 371, 405
Knotenkonfigurationen abrufen 381, 384
Knotenverwaltung 403
Konfigurationen abrufen 377, 380
ListClusterWMIClasses.ps1-Skript 371, 376
mehrere Klassen abfragen 391
CMD-Interpreter 15
Cmdlets
allgeimeine Parameter 10–11
Ausführung steuern 7, 9
Compare-Object 101
Copy-Item 301
Ereignisprotokolle 71
Export-Console 6
ForEach-Object
Clusterdienst 383, 389–390, 396
Desktopkonfiguration 234
Desktopverwaltung 183
Dienstverwaltung 81, 97, 102–103
Ereignisprotokollverwaltung 65
Netzwerkverwaltung 201, 210
Problembehandlung 329
Scripting 37–38
Formatierung 152
Ausgeben 141, 148
Dienste 88, 90
Freigaben 107, 116
IIS-Verwaltung 410, 413
Netzwerkadapter 199
Netzwerkdiensteverwaltung 503, 508, 513
Problembehandlung 330, 337, 345
Protokolle 61
Terminaldiensteverwaltung 474, 478, 482
Übersicht 16
Get-Alias 16
Get-ChildItem
Ausgabeverwaltung 152–153
Übersicht 16, 21
Zertifikatspeicher und 435, 439, 447, 450–451, 459
Get-Command 15, 23, 25
Get-Content 101, 140
Get-Credentials Windows PowerShell 361
Get-Date 38, 83, 258
Get-EventLog
Protokollverwaltung 60, 62–64
Windows Server 2008 Server Core 566
Get-Help 12, 14
Get-Member
Datenverwaltung 310
Desktopkonfiguration 229
Freigabeverwaltung 109
IIS-Verwaltung 418
Netzwerkdiensteverwaltung 516

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Cmdlets *(Fortsetzung)*
Protokollverwaltung 62
Scripting 44, 47
Übersicht 25, 28
Get-Serice 90
Get-Service 75
Get-WmiObject
angemeldete Benutzer 228
Ausgeben 137, 141, 150
Bildschirmschoner überwachen 229
Clusterdienst 374, 376, 382, 387–388, 390, 397, 402–403
credential-Parameter 292
Datenträgereigenschaften 176
Datenträgerleistung 187
Desktopeinstellungen 234, 240
Dienste 79, 93, 104
Freigaben 110, 117
IIS-Verwaltung 415, 431
Laufwerkeigenschaften 159, 163
Netzwerkadapter 198–199, 201, 206
Netzwerkdiensteverwaltung 508, 516, 520–521, 525, 528
Netzwerk-Problembehandlung 344
Problembehandlung 322, 325, 329, 332, 340–341
Seitenfehler 189
Systemspeicher 314, 317
Terminaldiensteverwaltung 471, 474, 479, 482, 489, 491–492, 496
Windows Server 2008 Server Core 540, 545, 547, 554, 563
Zeiteinstellungen 258
Invoke-Expression 541, 559
New-Item 128
New-Object
Aufgaben nach der Bereitstellung 263
Ausgabeverwaltung 142
Desktopkonfiguration 245, 250
Desktopverwaltung 162
Diensteverwaltung 82
Domänenbenutzerverwaltung 368
Freigabeverwaltung 114, 118
Protokollverwaltung 66
Zertifikatspeicher und 459, 462–463
New-Timespan 183
New-Variable 329
Optionen 12
Out-File
Ausgabeverwaltung 140–141
Clusterdienst 390
Diensteverwaltung 102
Out-String 71
Read-Host 359
Select-Object 229
Set-Alias 14
Set-Content 141
Set-Location 7
Sort-Object 20, 40
Start-Sleep 187
Stop-Process 10
Test-Path 373, 520
Textänderung 141
Where-Object
Ausgabeverwaltung 150
Diensteverwaltung 76
Netzwerkdiensteverwaltung 517
Protokollverwaltung 64
Windows Server 2008 Server Core 566
Write-Host 139
abgelaufene Dateien 184
Ausgabeverwaltung 141, 144–145
automatisches Anzeigen von Variablen 203
Clusterdienst 373, 378
Datenträgereigenschaften 175
Datenverwaltung 302, 306
Diensteverwaltung 81, 83–84, 92
Farbparameter 144
Firewallkonfiguration 221
Freigabeverwaltung 120, 123
Netzwerkdiensteverwaltung 515
Protokolle 59
Scripting 38, 42
Statusanzeige 179, 240
Statuszeile 240
Zeit festlegen 256
Zertifikatspeicher und 442, 447, 451
column-Parameter 22
commandType-Parameter 25
Community-Kennwörter 150
comobject-Parameter 82
CompareServicesTxt.ps1-Skript 102
CompareShares.ps1-Skript 122–123
Computer
remote 294, 296
umbenennen 291–292
ComputerName-Eigenschaft 81
computername-Parameter
Freigabeverwaltung 133
IIS-Verwaltung 418
computername-Umgebungsvariable 206
computername-Variable 304, 314
computer-Parameter
Aufgaben nach der Bereitstellung 268, 291
Clusterdienst 372, 401, 403
Datenverwaltung 306
Desktopkonfiguration 228, 232, 234
Desktopverwaltung 169
IIS-Verwaltung 409, 411, 414, 417
Netzwerkdienstekonfiguration 514, 527, 533
Netzwerkverwaltung 191
Problembehandlung 327, 329, 341
Terminaldiensteverwaltung 483, 487
Windows Server 2008 Server Core 558, 564, 566
Computerverwaltungskonsole 279, 281–282
ConfigureClientColor.ps1-Skript 489
ConfigureClientEnvironment.ps1-Skript 490, 492
ConfigureClientProperties.ps1-Skript 476
ConfigureDNSLogging.ps1-Skript 505

ConfigureScreenSaver.ps1-Skript 285
confirm-Parameter 8–9, 11
connection-Objekt
ADO 82
close-Methode 84, 165, 180
erstellen 118, 143, 163
open-Methode 119, 163–164, 179, 240
connection-Parameter 144
ConnectServer-Methode 387
Count-Eigenschaft 341, 402
CountRunningServices.ps1-Skript 76
CountServices.ps1-Skript 75–76
CreateAndEnableUser.ps1-Skript 360
CreateDNSZonesConfig.ps1-Skript 526
CreateEventSource-Methode 70, 72
CreateGlobalVariableInFunction.ps1-Skript 308
CreateGroup.ps1-Skript 362, 364
CreateLocalGroup.ps1-Skript 282
CreateLocalUser.ps1-Skript 279–280
CreateOU.ps1-Skript 347
CreateShare.ps1-Skript 127, 129
CreateVariableInFunction.ps1-Skript 308
CreateVariableInFunctionAndOutsideFunction.ps1-Skript 307–308
CreatMultipleShares.ps1-Skript 131
CreatShare.ps1-Skript 149
CreatUser.ps1-Skript 350
credential-Parameter 295
CurrentUser-Zertifikatspeicher 435, 437, 444, 457, 459, 464

D

Datei mit durch Kommas getrennten Werten (.csv)
Diensteverwaltung 79
von Domänenbenutzer erstellt 360–361
Datenbanken 276
Datenträgerverwaltungsprogramm 166
Datentypen, in Skripts 48
Datenverwaltung 299, 319
Offline-Dateien 311
aktivieren 305
konfigurieren 304
Systemwiederherstellung 312, 318
datetime-object` 183
days-Parameter 449
debug-Parameter 11
DeleteEventSource.ps1-Skript 73
DeleteFileAndFolder.txt-Befehl 5
DeleteUnauthorizedShares.ps1-Skript 135
DemoFormatWide.ps1-Skript 22
DemoWriteHostColors.ps1-Skript 144
depth-Parameter 488
Description-Eigenschaft
Freigabeverwaltung 107, 114, 124
Problembehandlung 341
Designansicht, Datenbanken 83, 142, 180
Desktopeinstellungen 225, 253
Bildschirmschoner 225, 241
Eigenschaften mit Werten 232, 235
sicher 236
DisableActiveDesktop.ps1-Skript 494, 496
Energie 242, 246
Energieschema ändern 247, 251
Konfigurationsprobleme 225
Desktopverwaltung 159, 189
Laufwerkinventur 159, 162
Leistungsüberwachung 185, 189
logische Datenträger 171, 174
Partitionen
Datenträger 168, 170
verwenden 166, 168
Speicherplatzbelegung 175, 185
Dateibeibehaltung 182
MonitorVolumeSpace.ps1-Skript 177
destination-Parameter 299–301
DestroyCluster-Methode 402
DetectStartupPrograms.ps1-Skript 336
Dialogfeld Erweiterte Freigabe 114
Dienstabhängigkeiten 338
Problembehandlung 330
Startprozesse 335
Dienste 75, 106
Dienste anhalten 87
Dienstprogrammskripts, Hilfefunktion 149
Dienststatus überprüfen 97
DisableLogons.ps1-Skript 472, 475
disk-Parameter 169–170
DisplayComputerRoles.ps1-Skript 41
DisplayLogSettings-Funktion 67
DisplayRootHints.ps1-Skript 513–514
distinguishedName-Attribut 366
DNS (Domain Name System)
Einstellungen 499, 517
GetDNSServerConfig.ps1-Skript 503
Protokollierung 503, 510
Servereinstellungen 519, 522
Windows Server 2008 Server Core-Einstellungen 550, 556
Zonen 524, 529
abrufen 525
DNSOwnerName-Eigenschaft 517
DnsServerName-Eigenschaft 517
Dokumentation
Dienste 86
abrufen 77
Textdatei 77, 79
Freigaben 107, 121
Benutzer 113, 116
ListShares.ps1-Skript 108
ListSharesDetailed.ps1-Skript 110, 112
ListSharesDetailedTranslateShareType.ps1-Skript 112–113
Textdatei 116–117
WMI-Klassen 110
Terminaldienste 469, 472
Domain-Eigenschaft 81
domainname-Parameter 538, 542
domain-Parameter 541
Domänen, beitreten 538, 542

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Domänenbenutzer 347, 370
Attribute 359
allgemeine Informationen 353–354
Registerkarte Organisation 357
Registerkarte Profile 355–356
Registerkarte Telefon 357
erstellen 350, 352
Gruppen
einen Benutzer hinzufügen 365–366
mehrere Benutzer hinzufügen 367, 369
Organisationseinheiten 347, 349
DoNotOverwrite-Beibehaltungsrichtlinie 65–66
Doppelte Anführungszeichen für Variablen 202–203
Dot-sourcing 552
Drahtlose Netzwerkadapter 205
Drucken 137, 158
Dienstabhängigkeiten 330
Druckerinventur 137, 146
ListPrinters.ps1-Skript 138
mehrere Computer abfragen 138, 140
Protokollierung in Dateien 140–141
Dynamic Host Configuration Protocol (DHCP) 531, 535

E

Einfache Anführungszeichen für Variablen 202
eingeschränkte Ausführungsrichtlinie 33
elevation-Zeichenfolge erforderlich 222
else-Anweisung 94, 503
Elseif, Else, If decision-Anweisung 40–41, 51
enableDHCP-Methode 217
EnableDisableOfflineFiles.ps1-Skript 307, 309
EnableDisableUser.ps1-Skript 274
EnabledUsers.csv-Datei 361
enable-Methode 310–311
EnablePrivileges-Eigenschaft 295
EnableRemoteAdmin.ps1-Skript 222
EnableSharedFolders 223
Ereignisprotokolle
erstellen 73
eventlog-Objekt 63
identifizieren 53
lesen 54, 58
schreiben 70, 72
suchen 64
verwalten 65, 67
Windows Management Instrumentation (WMI) 68, 70
Zeiteinstellung 261
erroraction-Parameter 11, 21
Error-Objekt 546
errorvariable-Parameter 11
Escape-Sequenzen 48–49
Ethernet-Adapter 207
EvaluateServicesAndCount.ps1-Skript 105
EventLogSpecificSource.ps1-Skript 262
Excel 2007 57
Datei mit durch Kommas getrennten Werten (.csv) und 79
mehrere Benutzer und Attribute 367
WriteUserSharesToExcel.ps1-Skript 114
ExcelApplicationCOM-Objekt 206
exclude-Parameter 152
Execquery-Methode 1
exit-Anweisung
Desktopverwaltung 173
IIS-Verwaltung 414
Netzwerkdienstekonfiguration 508
Netzwerkverwaltung 193
exit-Befehl 227
Exportieren
Ereignisprotokolle 54

F

fileAndPrint-Dienst 222
fileinfo-Objekt 183
file-Parameter 389–390
filepath-Parameter 78
Filter
Netzwerkeinstellungen 200
filter-Parameter
Freigabeverwaltung 124
Netzwerkverwaltung 210
FindCertificatesAboutToExpire.ps1-Skript 449
FindConfigurationOfConnected Adapters.ps1-Skript 209
FindIISClasses.ps1-Skript 408
FindPrinterDrivers.ps1-Skript 153
FindPrinterPorts.ps1-Skript 150
FindUSBEvents.ps1-Skript 63
Firewalls
Konfiguration 537–538
Netzwerk 191, 219, 223
Flow Stream-Anweisungen 37–38
For-Anweisung 39
foreach-Anweisung
Ausgabeverwaltung 145
Dienstverwaltung 104
Freigabeverwaltung 110, 123
Protokollverwaltung 59
foregroundcolor-Parameter 103
Formatierungs-Cmdlets
Protokolle 60
for-Schleife 201, 226
Freigaben 107, 136
erstellen 127, 131
löschen 134
Freigaben ändern 127
Freigaben löschen 133, 135
Freigabewerte umwandeln 111
FriendlyName-Eigenschaft, Zertifikatspeicher und 439
Fsutil.txt-Programm 4–5
full-Parameter 336, 344–345
funadd-Funktion 397–398
funall-Funktion 389
funcert-Funktion 464
funchange-Funktion
Netzwerkdienstekonfiguration 509, 521–522
Terminaldiensteverwaltung 474
funcountresource-Funktion 402
funeval-Funktion 287–288

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

FunEvalRTN-Funktion 98, 212–213, 216–217
funevict-Funktion 397
funhelp-Funktion
Aufgaben nach der Bereitstellung 256–257, 267, 270–271, 275, 279–280, 282, 286–287, 291–292, 294–295
Ausgabeverwaltung 148–149
Clusterdienst 372, 374, 377, 380, 382, 386, 391, 395, 401, 403
Datenverwaltung 299–300, 303, 306
Desktopkonfiguration 233, 239, 244–245, 248
Domänenbenutzerverwaltung 347–348, 350–351, 358–359, 362, 365–366
Freigabeverwaltung 128, 133
IIS-Verwaltung 409, 412, 415, 417, 420, 423–424, 427–428, 430–431
Netzwerkdienstekonfiguration 500–501, 506, 515, 519–520, 524–525, 527, 532, 553
Netzwerkproblembehandlung 343–344
Netzwerkverwaltung 193–194, 196, 213, 216
Problembehandlung 322, 324, 328, 331–332, 336, 339–340
Systemwiederherstellung 313, 316
Terminaldiensteverwaltung 470, 473, 475, 477, 479, 481, 484–485, 487, 491–492, 494
Windows Server 2008 Server Core 538, 542, 545, 548, 551, 558, 560, 564, 566–567
Zertifikatspeicher und 438–439, 441–442, 445, 449, 454, 458, 461
funjoindomain-Funktion 541–542
funline-Funktion
Aufgaben nach der Bereitstellung 258–259, 263, 269–270, 272, 285
Clusterdienst 378, 382, 387
Datenverwaltung 303, 310
Desktopkonfiguration 227, 230, 234, 238, 245, 248
Desktopverwaltung 183
Netzwerkproblembehandlung 343
Netzwerkverwaltung 194, 203
Problembehandlung 327, 329, 339
Systemwiederherstellung 313, 317
funlist-Funktion
Clusterdienst 388
Terminaldiensteverwaltung 479, 495
Windows Server 2008 Server Core 554
funlog-Funktion 261
funlookup-Funktion
Freigabeverwaltung 112, 126, 129
Systemwiederherstellung 316
funpaper-Funktion 492
funreboot-Funktion 539–540, 560
funrename-Funktion 560
funstart-Funktion 510
funstatus-Funktion 192
funstop-Funktion 509–510
funtestns-Funktion 373, 390
funtranslatemethod-Funktion 307, 309–310
funvalrtn-Funktion 546, 548, 553, 555
funwhatif-Funktion 397–398
funwmiclass-Funktion 374
funwmi-Funktion 380, 384, 397

G

Get32ndEventLogEntry.ps1-Skript 60
GetApplicationEventLogs.ps1-Skript 54
GetAppPool.ps1-Skript 413
GetAppPoolDefaults.ps1-Skript 414
GetDiskPerformance.ps1-Skript 186, 189
GetEventLogRetentionPolicy.ps1-Skript 65–66
GetEventLogs.ps1-Skript 54
GetFirstEntry.ps1-Skript 60–61
GetHalfDuplex.psi-Skript 64
GetHardDiskDetails.ps1-Skript 36
GetLastEvent.ps1-Skript 61
GetLogSources.ps1-Skript 65, 72
GetMultipleServices.ps1-Skript 91
GetNetAdapterConfig.ps1-Skript 196
GetNetAdapterStatus.ps1-Skript 191
GetNetID.ps1-Skript 205
GetNewestLogEntries.ps1-Skript 58–59
GetOfflineFiles.ps1-Skript 302
GetProcessByID.ps1-Skript 37
GetServiceStatus.ps1-Skript 41
GetSetTime.ps1-Skript 256, 259
GetSetTimeWriteToEventLog.ps1-Skript 261
GetSharesWithArgs.ps1-Skript 31
GetSingleEventEntry.ps1-Skript 60
GetSites.ps1-Skript 408
GetSpecificServices.ps1-Skript 89
GetStringValue 272
GetSystemLogErrors.ps1-Skript 64
GetSystemRestoreSettings.ps1-Skript 315
GetTimeSource.ps1-Skript 269
GetTopMemory.ps1-Skript 40
GetWmiAndQuery.ps1-Skript 37
GetWMILogLevel.ps1-Skript 69
Get-WmiObject-Cmdlet
Terminaldiensteverwaltung 488
Windows Server 2008 Server Core 540
GrantUserTSPermission.ps1-Skript 482, 485
Graviszeichen 38
Gruppenrichtlinie
PowerShell bereitstellen 3
sichere Bildschirmschoner 237
Windows-Firewall 219

H

help-Funktion
Domänenbenutzerverwaltung 351, 358–359, 362–363
help-Parameter
Aufgaben nach der Bereitstellung 256, 268
Ausgabeverwaltung 150
Clusterdienst 372, 382
Datenverwaltung 299, 303
Desktopverwaltung 169
IIS-Verwaltung 410, 412, 418, 420, 422, 424
Netzwerkdienstekonfiguration 500, 515, 528, 532
Problembehandlung 321, 325, 337

help-Parameter *(Fortsetzung)*
Systemwiederherstellung 313
Terminaldiensteverwaltung 470–471, 477, 484, 487
Windows Server 2008 Server Core 551, 554, 558
Zertifikatspeicher und 438, 441, 446, 450, 459, 461
Hilfe und Beenden 161
Hotfixes 2–3
HTTP-Dienst 326

I

if-Anweisung
Dienstverwaltung 86, 96
Freigabeverwaltung 135
Netzwerkverwaltung 197
inputobject-Parameter 314
InstallPrinterDriverFull.ps1-Skript 157
InstallPrinterDrivers.ps1-Skript 154–155
Internetinformationsdienste (IIS) 407, 433
aktivieren 407–408
Konfiguration überprüfen 408
Websiteeinschränkungen 417, 419
Websiteinformationen 408, 410
Website
erstellen 422, 425
Websites
starten und beenden 429, 432
IP-Adresse
Einstellung 548
Item-Methode 115, 207

J

Jet.OLEDB 4.0-Anbieter 119, 179

K

Kennwörter, Einstellungen 361
Keys-Eigenschaft 521
Konfiguration der Dienste
bestätigen 104
festlegen 100
beenden 91, 94
GetMultipleService.ps1-Skript 90
GetSpecificService.ps1-Skript 89
starten 95
verwalten 101, 103
Konfiguration der PowerShell 6–7
Konstanten
GB Windows PowerShell 177
in Skripts 36

L

Length-Eigenschaft
Freigabeverwaltung 123
Netzwerkverwaltung 194
Lightweight Directory Access Protocol (LDAP)-Dienstanbieter 366
listcu-Parameter 446
Listen, Format-List-Cmdlet
Übersicht 17–18
listlm-Parameter 446
ListNonAdminShares.ps1-Skript 114
list-Parameter
Clusterdienst 385
Terminaldiensteverwaltung 474
ListPerformanceCounterClasses.ps1-Skript 185–186
ListPrinterPorts.ps1-Skript 147, 149
ListPrintersFromMultipleComputers.ps1-Skript 139
ListProcessesSortResults.ps1-Skript 31
ListSharesDetailed.ps1-Skript 108
liststores-Parameter 459
ListVirtualDirectory.ps1-Skript 420
LocalDateTime-Eigenschaft 258
Locking, optimistic 143
Log-Eigenschaft 59
Lokale Benutzerkonten 284

M

Magic Number 329
ManagedRuntimeVersion-Eigenschaft 413
Management.ManagementDateTimeConverter.NET Framework-Klasse 217, 258
management-Objekt
Ausgabeverwaltung 155, 157
Desktopverwaltung 178, 187
Diensteverwaltung 99
Freigabeverwaltung 123
IIS-Verwaltung 412
Problembehandlung 322, 329
ManageWinsDHCP.ps1-Skript 532
Manueller Startmodus 86
match static-Methode 50
match-Anweisungen 204
matches-Methode 49
MaxmumAllowed-Eigenschaft 124
Media Access Control (MAC)-Adresse 205
member-Attribut 365
Microsoft Consulting Services 117
Microsoft Developer Network (MSDN) 28, 537
Microsoft Management Console (MMC) 185
MinimumRetentionDays-Richtlinie 65–66
ModifyAddressProperties.ps1-Skript 355
ModifyGeneralProperties.ps1-Skript 356
ModifyOrganizationProperties.ps1-Skript 357
ModifyProfileProperties.ps1-Skript 356
MonitorServer.ps1-Skript 565
moveTo-Methode 27

N

Name-Eigenschaft
Desktopkonfiguration 229
Übersicht 27
Namenskonventionen für Cmdlets 6
name-Parameter
Diensteverwaltung 91
namespace-Parameter
Clusterdienst 372
Net Time-Befehl 268
netdom-Befehl 541

netsh-Befehl
Netzwerkdienstekonfiguration 534
Netzwerkverwaltung 222
Network Adapter Configuration-Objekt 344
NetworkAdapterConfigFiltered.ps1-Skript 202–203
Networking 223
Netzwerk
Einstellungen 204
überprüfen 191, 195
Netzwerk- und Freigabecenter 192–193, 196
Netzwerkdienste 499, 536
newname-Parameter 558
nicht signierte Treiber 339, 341
NotAfter-Eigenschaft 451
notmatch-Operator 207
NtpServer-Registrierungswert 271

O

open-Methode 82
option-Parameter
Zertifikatspeicher und 442
Ordner, freigegeben 222
Organisationseinheiten 3, 363
outvariable-Parameter 12
OverwriteAsNeeded-Beibehaltungsrichtlinie 65
OverwriteOlder-Beibehaltungsrichtlinie 67

P

param-Anweisung
Ausgabeverwaltung 148
Datenverwaltung 302
Desktopkonfiguration 238, 243
Desktopverwaltung 169, 172
Domänenbenutzerverwaltung 365
Freigabelisten 112
Freigabeverwaltung 125, 127
IIS-Verwaltung 421, 429
Netzwerkdienstekonfiguration 506, 524
Netzwerkproblembehandlung 343
Netzwerkverwaltung 196, 211, 216
Zertifikatspeicher und 437, 454
ParseAppTextLog.ps1-Skript 55
ParseFWConfig.ps1-Skript 220–221
Partitionen
ausgeblendet 159
persönlicher Zertifikatspeicher 436–437
physischer Datenträger 165
PingsARange.ps1-Skript 39
Platzhalter
Cmdlets 13, 17–18
Eigenschaftsnamen vergleichen 204
Ports, Drucker 147, 151
PowerShell 1, 28
Sicherheitsprobleme 7, 10
PowerShell installieren 1, 6
bereitstellen 3
mit VBScript überprüfen 1–2
Problembehandlung 321, 346
Probleme nach der Bereitstellung 255, 297
Benutzerkonten
aktivieren 274, 277
Bildschirmschonerkonfiguration 289
Zeiteinstellung 255, 264
Zeitquelle 266, 272
Profil, PowerShell 6
Prompt-for-Information-Methode 161
Protokolle 53, 73
allgemein 58, 61
Psconsole-Datei 6
PsIsContainer-Eigenschaft 27
PSObject.NET Framework-Klasse
Clusterdienst 379, 390
Desktopkonfiguration 234
PSReference-Objekt 270

Q

Quellen, Ereignisprotokolle 71
QueryDNSARecords.ps1-Skript 514
QueryOldFile.ps1-Skript 183–184
query-Parameter
Diensteverwaltung 81
Freigabeverwaltung 118
Netzwerkverwaltung 198

R

ReadExcelModifyUsers.ps1-Skript 368
reboot-Funktion 540
reboot-Parameter 538
RebootRequired-Eigenschaft 311
recordset-Objekt
erstellen 143
open-Methode 82, 119, 179, 240
StoppedServices-Tabelle und 85
update-Methode 119, 164
regex-Parameter 43
RegExTab.ps1-Skript 49
reguläre Ausdrücke
Meldungen in Ereignisprotokollen analysieren 64
Netzwerkadressenübereinstimmung 150
Objekte vergleichen 220
Skripts 48, 51
switch-Anweisung 43
Zertifikatspeicher und 439
RegWhiteSpace.ps1-Skript 50
RemoteSigned-Ausführungsrichtlinie 33
RemoveCluster.ps1-Skript 401
RenameReboot.ps1-Skript 558, 561
ReportAvailableDrivers.ps1-Skript 153
ReportClientSetting.ps1-Skript 481–482
ReportDesktopSettings.ps1-Skript 232
ReportDiskDriveConfiguration.ps1-Skript 162
ReportDiskPartition.ps1-Skript 166
ReportLogicalDiskConfiguration.ps1-Skript 173
ReportPowerConfig.ps1-Skript 242
ReportSpecificLogicalDisk.ps1-Skript 174
restart-Parameter 510
ReturnValue-Eigenschaft 94, 99, 124

Rows-Eigenschaft 368
Rückgabecode, umwandeln 126

S

Schweregrad der Ereigniseinträge 63–64
Security Account Manager (SAM)-Namensattribut 360
Select case-Anweisung 41
Server system-Eigenschaft 177
ServerWMI-Klasse 416
Service.Name-Eigenschaft 81
SetEventLogRegention Policy.ps1-Skript 66–67
shareName-Parameter 133
shutdown-Befehl 541
Sicherheit
  administrative Freigaben 117
  Freigaben dokumentieren 107
Sichern 299, 301
SilentlyContinue-Variable
  Aufgaben nach der Bereitstellung 261
  Clusterdienst 373
  Problembehandlung 327
Skripts 29, 52
  AcceptPause.ps1 87
  ArgsShare.ps1 51
  ausführen 34–35
  Parameter 125
  Richtlinie 32, 34
  SearchByEventID.ps1 62
  SearchTypePerformanceCounterClasses.ps1 186
  ServiceDependencies.ps1 327
  SetDNS.ps1 552
  SetShareInfo.ps1 124–125
  SetShareInfoWithParameters.ps1 125
  SetStaticIP.psi 211
  SetWMILogLevel.ps1 69
  StartMultipleServices.ps1 95–96
  StartService.ps1 95
  StopMultipleServices.ps1 93
  StopServices.ps1 92
  StringMethods.ps1 228
  SwitchRegEx.ps1 43
  ThreeStrings.ps1 455
  Ursache 29, 31
  Variablen 35–36
  WorkWithDHCP.ps1 216
  WriteAppLogToText.ps1 54
  WriteAppLogToXML.ps1 57
  WritePrinterInfoToAccess.ps1 146
  WriteProcessesToAppLog.ps1 71
  WriteRunningServicesToAccess.ps1 84–85
  WriteRunningServicesToTxt.ps1 78
  WriteServiceConfigToAccess.ps1 87
  WriteServiceStatus.ps1 101
  WriteSharesToFile.ps1 116, 122
  WriteStoppedServices.ps1 102
  WriteToAppLogs.ps1 70–71
SNMP (Simple Network Management Protocol) 150
SourceExists-Methode 70
Sperrmethode, Datenbank 164
SQL Server 2007 78
Standardparameter 198
Startkonfiguration 321, 323
StartMode-Eigenschaft 86
Startmodus 86
Startmodus deaktivieren 86
Startprobleme
  Optionen konfigurieren 7
  Problembehandlung 321, 325
Statusanzeige 164
StatusCode-Eigenschaft 39
store-Parameter 446
suspend-Parameter 10
switch-Anweisung
  Bildschirmschonerkonfiguration 287
  ChangeLogSettings-Funktion 67
  CheckStatusWMILog.ps1 69
  Datenverwaltung 309
  EvaluateServicesAndCount.ps1-Skript 104
  Netzwerkdienste 508, 529, 534
  Problembehandlung 332
  Remotecomputer herunterfahren oder neu starten 295
  Terminaldiensteverwaltung 477, 479, 485
  whatif 11
SwitchIPConfig.ps1-Skript 43
System.Diagnostics.EventLog
  Ereignisprotokolle schreiben 70
System.Diagnostics.Eventlog.NET Framework-Klasse 263
System.Management.Automation.InvocationInfo Microsoft .NET Framework 516
System.Management.Automation.ScriptInfo Microsoft .NET Framework-Klasse 516
System.Management.ManagementObject Microsoft .NET Framework-Klasse 555
System.Security.Cryptography.X509Certificates-Namespace 455
System.String.NET Framework-Klasse 565
SystemIO.Path.NET Framework-Klasse 564
Systemstartmodus 86
Systemsteuerung 285, 302

T

Tabellen
  Format-Table-Cmdlet
    Übersicht 18
Terminaldienst 469, 497
  Benutzer verwalten
    Serverzugriff 483
  konfigurieren 469
    Clienteigenschaften 476, 479
Text
  aktuelle und erforderliche Freigaben 122
  Ereignisprotokolle exportieren 56
throw-Anweisung 276, 280, 283, 310
timespan-Objekt 183
Treiber
  drucken
    identifizieren 152–153
    installieren 154, 157
  Startprobleme 331, 334

U

Überwachung
- AuditScreenSaver.ps1-Skript 226, 228–229
- AuditUnauthorizedShares.ps1-Skript 123, 135
- Bildschirmschoner 225, 231
- Freigabe 121, 124

Umbrechen, Ereignisprotokolle 61
UsedRange-Eigenschaft 207
User Account Control (UAC) 4
- Ereignisprotokolle 57
- Scripting 33
- Zertifikate 435

UserDomain-Eigenschaft 142

V

Value-Eigenschaft 38, 204, 270
valuefunline-Funktion 230
Variablen
- Dienste als Datenbanken dokumentieren 82
- Funktionen 112
- SetShareInfo.ps1-Skript 125

VBScript 275, 424
verbose-Parameter 11
Verfügbare Systemwiederherstellungspunkte 316
Visible-Eigenschaft 115, 206

W

whatif-Parameter
- Clusterdienst 397

Win_32PrinterDriver-Eigenschaften 156
Windows Administration Tools Pack 541
Windows Explorer 159, 182
Windows Management Instrumentation (WMI)
- Abfragen 39
- Dienste 98–99, 103
- Druckertreiber installieren 155
- IIS 7-Klassen 408
- Testprogramm (wbemtest.exe) 109, 201

Windows Server 2008 Server Core 537, 568
- ursprüngliche Konfiguration 563
- verwalten 567

Windows Software Development Kit (SDK)
- Aufgaben nach der Bereitstellung 275
- Desktopkonfiguration 239
- Freigabeverwaltung 111
- Netzwerkdiensteverwaltung 528
- Problembehandlung 344

WMI Query Language (WQL) 81
WMI-Klasse BindingElement 422, 424–425
WriteServiceStatus.ps1-Skript 101
WriteStoppedServicesToAccess.ps1-Skript 83, 85
wscript.network Programm-ID 118
wshNetwork-Objekt 81

Z

Zeichenfolgenmethoden 228–229
Zeichenschemas 50
Zeitformat Distributed Management Task Force (DMTF) 258
Zeitstempel 119, 187
Zertifikatimport-Assistent 457
Zertifikatspeicher 435, 467
- abgelaufene Zertifikate 444, 447
- ablaufende Zertifikate 449, 451
- FindCertificates.ps1-Skript 437, 440
- Terminologie 435, 437
- Zertifikate auflisten 441, 443
- Zertifikate importieren 460
- Zertifikate löschen 461, 465
- Zertifikate überprüfen 453, 456

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Über den Autor

Ed Wilson ist ein Consultant bei Microsoft Corporation und ein bekannter Skriptexperte sowie ein Microsoft Certified Trainer. Er hält einen beliebten Windows PowerShell-Workshop, an dem Microsoft Premier-Kunden aus aller Welt teilnehmen. Ed Wilson hat mehrere Bücher über Windows-Scripting verfasst, einschließlich *Microsoft Windows PowerShell Step by Step* und *Microsoft VBScript Step by Step*. Ed Wilson besitzt mehr als 20 Branchenzertifizierungen, einschließlich Microsoft Certified Systems Engineer (MCSE) und Certified Information Systems Security Professional (CISSP).

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9

Microsoft Windows PowerShell-Scripting – Die technische Referenz, © 2008 Microsoft Press Deutschland, ISBN 978-3-86645-919-9