

Chapter 5

Managing Active Directory with PowerShell Fundamentals

As you know, PowerShell relies on the directory service .NET classes and is based on the .NET 2.0 Framework. The parent class is `System.DirectoryServices`. You'll most likely be utilizing the `System.DirectoryServices.DirectoryEntries` and `System.DirectoryServices.DirectoryEntry` classes. In this chapter, I'll demonstrate how to use these classes and a few others in PowerShell. Later in the book, I'll show more efficient and PowerShell-friendly ways to accomplish many of the same tasks covered below.

Using the .NET Framework DirectoryService classes

I'll start at the beginning. The main class you need to become familiar with is the `System.DirectoryServices.DirectoryEntry`. This class will return, as the name suggests, an entry or object from a directory service. In this case, the directory service will be Active Directory, although the .NET directory service classes are designed for any LDAP-compliant directory service. In PowerShell, you use the **New-Object** cmdlet to create a new object. If you are running PowerShell with appropriate credentials, creating such an object is as easy as:

```
PS C:\> New-Object DirectoryServices.DirectoryEntry "LDAP://mycompany.local"
distinguishedName
-----
{DC=mycompany,DC=local}
```

```
PS C:\>
```

All you need to do is specify the LDAP path of a specific Active Directory object. In the example above, I'm connecting to the root of the `mycompany.local` domain. You can also use the distinguished name

format, as shown below:

```
PS C:\> New-Object DirectoryServices.DirectoryEntry "LDAP://dc=mycompany,dc=local"
distinguishedName
-----
{DC=mycompany,DC=local}
```

```
PS C:\>
```

In fact, you need to use the distinguished name when you want to connect an organizational unit:

```
PS C:\> New-Object DirectoryServices.DirectoryEntry "LDAP://OU=Servers,DC=mycompany,dc=local"
distinguishedName
-----
{OU=Servers,DC=mycompany,DC=local}
```

If you need to authenticate with alternate credentials, you can add them as additional constructor parameters:

```
PS C:\> $admin="mycompany\da_jhicks"
PS C:\> $pwd=Read-Host "Enter the password for $admin"
Enter the password for mycompany\da_jhicks: P@ssw0rd
PS C:\> New-Object DirectoryServices.DirectoryEntry "LDAP://mycompany.local",$admin,$pwd
distinguishedName
-----
{DC=mycompany,DC=local}
```

```
PS C:\>
```

Don't use the `-AsSecureString` parameter with **Read-Host**. The password that you use in this example can't be a secure-string. But don't fear – the password is not passed as clear text over the network. If you prefer, you can use a `PSCredential`:

```
PS C:\> $cred=get-credential mycompany\da_jhicks
PS C:\> New-Object DirectoryServices.DirectoryEntry `
>> "LDAP://mycompany.local",$cred.Username,$cred.GetNetworkCredential().Password
>>
distinguishedName
-----
{DC=mycompany,DC=local}
```

```
PS C:\>
```

The `$cred` variable holds the `PSCredential` object. You can use its **Username** property to pass the string `mycompany\da_jhicks` to the **New-Object** cmdlet. To get the password as a string, call the **GetNetworkCredential()** method, which returns a `System.Net.NetworkCredential` object. The `Password` property of this object will be a string. This technique allows you to pass strings as necessary, but not directly expose them in the console.

Case Counts

As with the WinNT provider I discussed in the last chapter, the LDAP provider is also case-sensitive and must be uppercase.

Now that you know how to create a directory service entry object, you can save it to a variable so you can explore it further:

```
PS C:\> $dsroot=New-Object DirectoryServices.DirectoryEntry "LDAP://mycompany.local", `
>> $cred.Username,$cred.GetnetworkCredential().Password
>>
```

```
PS C:\> $dsroot
distinguishedName
-----
{DC=mycompany,DC=local}
```

```
PS C:\>
```

If you pipe \$dsroot to **Get-Member**, you should see these properties for the domain root:

Table 5-1 DirectoryEntry Properties

auditingPolicy	creationTime	dc
distinguishedName	dSASignature	forceLogoff
fSMORoleOwner	gPLink	instanceType
isCriticalSystemObject	lockoutDuration	lockOutObservationWindow
lockoutThreshold	masteredBy	maxPwdAge
minPwdAge	minPwdLength	modifiedCount
modifiedCountAtLastProm	ms-DS-MachineAccountQuota	msDS-AllUsersTrustQuota
msDS-Behavior-Version	msDs-masteredBy	msDS-PerUserTrustQuota
msDS-PerUserTrustTombstonesQuota	name	nextRid
nTMixedDomain	nTSecurityDescriptor	objectCategory
objectClass	objectGUID	objectSid
pwdHistoryLength	pwdProperties	replUpToDateVector
rIDManagerReference	serverState	subRefs
systemFlags	uASCompat	uSNChanged
uSNCreated	wellKnownObjects	whenChanged
whenCreated		

Your list of properties may vary, depending on your Windows version or network configuration.

Get Domain Information

Once you know what properties are available, you can wrap things up in a function. For example, if you want to display some information about your domain, use:

```
PS C:\> $DSRoot | Format-List @{label="DN";Expression={$_.DistinguishedName}},`
>> @{label="Name";Expression={$_.Name}},`
>> @{label="Created";Expression={$_.WhenCreated}},`
>> @{label="Last Modified";Expression={$_.WhenChanged}},`
>> @{label="FSMO";Expression={$_.FSMORoleOwner}}
>>
```

```
DN           : DC=mycompany,DC=local
Name          : mycompany
Created       : 4/14/2004 10:52:27 PM
Last Modified : 2/4/2008 5:02:49 AM
FSMO          : CN=NTDS Settings,CN=MYDC01,CN=Servers,CN=Default-First-Site-Name...
```

```
PS C:\>
```

In the example above, I created custom labels to make the output more user friendly.

Get Domain Account Policy Information

The domain's account policy information is stored in several properties such as **minPwdAge** and **LockoutDuration**. However, getting these values is not an easy task. You might think you can use a PowerShell expression like this:

```
PS C:\> $dsroot | Select *pwd*,lock*
```

But if you run this, your output will be:

```
maxPwdAge           : {System.__ComObject}
minPwdAge           : {System.__ComObject}
minPwdLength        : {7}
pwdProperties        : {1}
pwdHistoryLength    : {24}
lockoutDuration      : {System.__ComObject}
lockOutObservationWindow : {System.__ComObject}
lockoutThreshold    : {7}
```

The length and threshold properties are readable, but the rest are not. I'll discuss the **pwdProperties** value first.

This property is actually a bitmask flag representing different password properties. You can use constants like this:

```
New-Variable DOMAIN_PASSWORD_COMPLEX 1 -option constant
New-Variable DOMAIN_PASSWORD_NO_ANON_CHANGE 2 -option constant
New-Variable DOMAIN_PASSWORD_NO_CLEAR_CHANGE 4 -option constant
New-Variable DOMAIN_LOCKOUT_ADMINS 8 -option constant
New-Variable DOMAIN_PASSWORD_STORE_CLEARTEXT 16 -option constant
New-Variable DOMAIN_REFUSE_PASSWORD_CHANGE 32 -option constant
```

You can perform a binary AND operation using the **PWDProperties** value and each of these constants. I've created a function to simplify the process:

Get-PWDProperties.ps1

```

Function Get-PWDProperties {
    Param([int]$flag=$(Throw "You must specify a property flag value."))

    # constant values for pwdProperties bitmask flag
    New-Variable DOMAIN_PASSWORD_COMPLEX 1 -option constant
    New-Variable DOMAIN_PASSWORD_NO_ANON_CHANGE 2 -option constant
    New-Variable DOMAIN_PASSWORD_NO_CLEAR_CHANGE 4 -option constant
    New-Variable DOMAIN_LOCKOUT_ADMINS 8 -option constant
    New-Variable DOMAIN_PASSWORD_STORE_CLEARTEXT 16 -option constant
    New-Variable DOMAIN_REFUSE_PASSWORD_CHANGE 32 -option constant

    if ($flag -band $DOMAIN_PASSWORD_COMPLEX) {
        write "Complex passwords required"
    }

    if ($flag -band $DOMAIN_PASSWORD_NO_ANON_CHANGE) {
        write "Anonymous change not allowed"
    }

    if ($flag -band $DOMAIN_PASSWORD_NO_CLEAR_CHANGE) {
        write "No clear change allowed"
    }

    if ($flag -band $DOMAIN_LOCKOUT_ADMINS) {
        write "Admin lockout allowed"
    }

    if ($flag -band $DOMAIN_PASSWORD_STORE_CLEARTEXT) {
        write "Reversible encryption enabled"
    }

    if ($flag -band $DOMAIN_REFUSE_PASSWORD_CHANGE) {
        write "Refuse domain password change"
    }
}

```

The function merely does a binary AND operation, with each possible flag value and the password property value. If the result is 1, then the function writes a message to the screen. If the function is loaded in your profile and the **pwdProperties** value is 7, you will see a result like this:

```

PS C:\get-pwdproperties 7
Complex passwords required
Anonymous change not allowed
No clear change allowed

```

Now, what about the other properties with a value of System.__ComObject? Without confusing you with .NET programming details, these objects are not readily discoverable. However, these property values are stored as large integers of 100 nanosecond intervals. ADSI, including the .NET directory service classes, split this 64 bit number into high and low parts. The formula for returning a 32 bit value is:

$$\text{HighPart} * 2^{32} + \text{LowPart}$$

My fellow PowerShell MVP Brandon Shell created a PowerShell function to perform this calculation. In the example below, I've made some minor adjustments to Brandon's function:

Convert-ADSLargeInteger.ps1

```
Function Convert-ADSLargeInteger {
# Take a large value integer and return a 32 bit version
# Thanks to Brandon Shell (http://bsonposh.com/) for the function

    Param([object]$adsLargeInteger=$(Throw "You must specify an object."))

    $highPart = $adsLargeInteger.GetType().InvokeMember("HighPart", 'GetProperty', `
    $null, $adsLargeInteger, $null)
    $lowPart = $adsLargeInteger.GetType().InvokeMember("LowPart", 'GetProperty', `
    $null, $adsLargeInteger, $null)
    $bytes = [System.BitConverter]::GetBytes($highPart)
    $tmp = [System.Byte[]]@(0,0,0,0,0,0,0,0)
    [System.Array]::Copy($bytes, 0, $tmp, 4, 4)
    $highPart = [System.BitConverter]::ToInt64($tmp, 0)
    $bytes = [System.BitConverter]::GetBytes($lowPart)
    $lowPart = [System.BitConverter]::ToUInt32($bytes, 0)

    write ($lowPart + $highPart)
}

```

I'll forgo a detailed explanation since the function includes some .NET Framework programming beyond the scope of this book. Just know that it works.

```
PS C:\> convert-adslargeinteger $dsroot.maxpwdage.value
-518400000000000
PS C:\>
```

Wait a minute. You're probably thinking that this doesn't look like a possible value for a maximum password age property. This value is actually the number of 100 nanosecond intervals between the time a password is set and the time it expires. To get a value in days, you need to divide the value by the number of 100 nanosecond intervals in a day:

$$((60 \text{ sec} * 60 \text{ min} * 24 \text{ hours}) * (1000000000/100)) = 864000000000$$

Now, do this:

```
PS C:\> (convert-adslargeinteger $dsroot.maxpwdage.value)/-864000000000
60
PS C:\>
```

Now you have a more meaningful number. I put all of this together in a script that will return domain account policies:

Get-DomainAccountPolicy.ps1

```
Function Get-PWDProperties {
# returns a text list of password flag values

    Param([int]$flag=$(Throw "You must specify a property flag value."))

    # constant values for pwdProperties bitmask flag
    New-Variable DOMAIN_PASSWORD_COMPLEX 1 -option constant
    New-Variable DOMAIN_PASSWORD_NO_ANON_CHANGE 2 -option constant
    New-Variable DOMAIN_PASSWORD_NO_CLEAR_CHANGE 4 -option constant

```

```
New-Variable DOMAIN_LOCKOUT_ADMINS 8 -option constant
New-Variable DOMAIN_PASSWORD_STORE_CLEARTEXT 16 -option constant
New-Variable DOMAIN_REFUSE_PASSWORD_CHANGE 32 -option constant
```

```
New-Variable data
```

```
if ($flag -band $DOMAIN_PASSWORD_COMPLEX) {
    #write "Complex passwords required"
    $data=$data+"Complex passwords required"
}

if ($flag -band $DOMAIN_PASSWORD_NO_ANON_CHANGE) {
    #write "Anonymous change not allowed"
    $data=$data+",`nAnonymous change not allowed"
}

if ($flag -band $DOMAIN_PASSWORD_NO_CLEAR_CHANGE) {
    #write "No clear change allowed"
    $data=$data+",`nNo clear change allowed"
}

if ($flag -band $DOMAIN_LOCKOUT_ADMINS) {
    #write "Admin lockout allowed"
    $data=$data+",`nAdmin lockout allowed"
}

if ($flag -band $DOMAIN_PASSWORD_STORE_CLEARTEXT) {
    #write "Reversible encryption enabled"
    $data=$data+",`nReversible encryption enabled"
}

if ($flag -band $DOMAIN_REFUSE_PASSWORD_CHANGE) {
    #write "Refuse domain password change"
    $data=$data+",`nRefuse domain password change"
}

write $data
```

```
}
```

```
Function Convert-ADSLargeInteger {
# Take a large value integer and return a 32 bit version
# Thanks to Brandon Shell (http://bsonposh.com/) for the function
```

```
Param([object]$adsLargeInteger=$(Throw "You must specify an object."))
```

```
$highPart = $adsLargeInteger.GetType().InvokeMember("HighPart", 'GetProperty', `
    $null, $adsLargeInteger, $null)
$lowPart = $adsLargeInteger.GetType().InvokeMember("LowPart", 'GetProperty', `
    $null, $adsLargeInteger, $null)
$bytes = [System.BitConverter]::GetBytes($highPart)
$tmp = [System.Byte[]]@(0,0,0,0,0,0,0,0)
[System.Array]::Copy($bytes, 0, $tmp, 4, 4)
$highPart = [System.BitConverter]::ToInt64($tmp, 0)
$bytes = [System.BitConverter]::GetBytes($lowPart)
$lowPart = [System.BitConverter]::ToUInt32($bytes, 0)
```

```
write ($lowPart + $highPart)
```

```
}
```

```
#connection credentials are optional
$admin="mycompany\administrator"
$pwd="P@ssw0rd"
```

```
#the ADSI path to the domain root
$DN='LDAP://mycompany.local'

$DSRoot = New-Object DirectoryServices.DirectoryEntry $DN,$admin,$pwd

Write-Host "Account Policies for "$DSRoot.DC.Value.ToUpper()

$DSRoot | Format-List `
@{label="Minimum Password Length";Expression={$_.MinPwdLength}},`
@{label="Password History";Expression={$_.PwdHistoryLength}},`
@{label="Lockout Threshold";Expression={$_.LockoutThreshold}},`
@{label="Lockout Duration (min)";Expression={{(Convert-ADSLargeInteger `
    $_.lockoutduration.value)/-600000000 }},`
@{label="Lockout Window (min)";Expression={{(Convert-ADSLargeInteger `
    $_.lockoutobservationwindow.value)/-600000000 }},`
@{label="Password Properties";Expression={Get-PWDProperties $_.PwdProperties.value}},`
@{label="Max Password Age (days)";Expression={{(Convert-ADSLargeInteger `
    $_.maxpwdage.value) /-864000000000}},`
@{label="Min Password Age (days)";Expression={{(Convert-ADSLargeInteger `
    $_.minpwdage.value) /-864000000000}}
```

I've included my functions in the script so that everything is self-contained. I made a minor change to the original **Get-PWDProperties** function, so that a single value is returned, which is essentially a text list of password properties:

```
if ($flag -band $DOMAIN_REFUSE_PASSWORD_CHANGE) {
#   write "Refuse domain password change"
    $data=$data+`,`nRefuse domain password change"
}

write $data
```

The main part of the script is connecting to the domain. I'm using alternate credentials:

```
#connection credentials are optional
$admin="mycompany\administrator"
$pwd="P@ssw0rd"

#the ADSI path to the domain root
$DN='LDAP://mycompany.local'

$DSRoot = New-Object DirectoryServices.DirectoryEntry $DN,$admin,$pwd
```

The script writes an informational header with the domain name in uppercase:

```
Write-Host "Account Policies for "$DSRoot.DC.Value.ToUpper()
```

Finally, I pipe the **\$DSRoot** object to **Format-Table** and create custom labels:

```
$DSRoot | Format-List `
@{label="Minimum Password Length";Expression={$_.MinPwdLength}},`
@{label="Password History";Expression={$_.PwdHistoryLength}},`
@{label="Lockout Threshold";Expression={$_.LockoutThreshold}},`
@{label="Lockout Duration (min)";Expression={{(Convert-ADSLargeInteger `
    $_.lockoutduration.value)/-600000000 }},`
@{label="Lockout Window (min)";Expression={{(Convert-ADSLargeInteger `
    $_.lockoutobservationwindow.value)/-600000000 }},`
@{label="Password Properties";Expression={Get-PWDProperties $_.PwdProperties.value}},`
```

```
@{label="Max Password Age (days)";Expression={(Convert-ADSLargeInteger `
$_.maxpwdage.value) /-86400000000}},`
@{label="Min Password Age (days)";Expression={(Convert-ADSLargeInteger `
$_.minpwdage.value) /-86400000000}}
```

You'll notice that I pass the large integer properties to the **Convert-ADSLargeInteger** function and divide the result by a number to give an appropriate value in minutes or days:

```
@{label="Lockout Window (min)";Expression={(Convert-ADSLargeInteger `
$_.lockoutobservationwindow.value)/-600000000 }},`
@{label="Password Properties";Expression={Get-PWDProperties $_.PwdProperties.value}},`
@{label="Max Password Age (days)";Expression={(Convert-ADSLargeInteger `
$_.maxpwdage.value) /-86400000000}},`
```

Here's the result when I run the script:

```
PS C:\> C:\scripts\Get-DomainaccountPolicy.ps1
Account Policies for MYCOMPANY

Minimum Password Length : 7
Password History        : 24
Lockout Threshold       : 5
Lockout Duration (min)  : 30
Lockout Window (min)    : 30
Password Properties     : Complex passwords required,
                        Reversible encryption enabled
Max Password Age (days) : 60
Min Password Age (days) : 1
```

Get Child Items

It is very easy to view a list of child objects from a given directory services entry. Because of the way PowerShell adapts the .NET Framework classes, you'll need to access the underlying PSBase object:

```
PS C:\> $dsroot.psbases.children

distinguishedName
-----
{OU=Admini,DC=mycompany,DC=local}
{CN=Builtin,DC=mycompany,DC=local}
{CN=Computers,DC=mycompany,DC=local}
{OU=Domain Controllers,DC=mycompany,DC=local}
{CN=ForeignSecurityPrincipals,DC=mycompany,DC=local}
{CN=Infrastructure,DC=mycompany,DC=local}
{CN=LostAndFound,DC=mycompany,DC=local}
{CN=NTDS Quotas,DC=mycompany,DC=local}
{OU=omaha,DC=mycompany,DC=local}
{CN=Program Data,DC=mycompany,DC=local}
{OU=sapien,DC=mycompany,DC=local}
{OU=Servers,DC=mycompany,DC=local}
{OU=syracuse,DC=mycompany,DC=local}
{CN=System,DC=mycompany,DC=local}
{OU=Testing,DC=mycompany,DC=local}
{CN=Users,DC=mycompany,DC=local}
```

As you can see, the **Children** property returns a list of containers and organizational units from \$dsroot

which you created earlier in the chapter. Although this domain does not, it is possible to have user, computer, and group objects in the domain root. These objects would also be listed as child items. If you want to filter out these objects you can do this:

```
PS C:\> $dsroot.psbases.children | where {$_.objectcategory -notmatch "Person" `
-AND $_.objectcategory -notmatch "Computer" -AND $_.objectcategory -notmatch "Group"}
```

I'll put all of this together in a script:

Get-ChildEntries.ps1

```
$DN='LDAP://dc=mycompany,dc=local'
$dsroot = New-Object DirectoryServices.DirectoryEntry $DN

$children=$dsroot.psbases.children | where {$_.objectcategory -notmatch "Person" `
-AND $_.objectcategory -notmatch "Computer" `
-AND $_.objectcategory -notmatch "Group" `
-AND $_.objectcategory -notmatch "Contact"}

$children | sort Name | Format-List `
@{Label="DN";Expression={$_.DistinguishedName}},`
@{Label="Name";Expression={$_.Name}},`
@{Label="Description";Expression={$_.Description}},`
@{Label="Created";Expression={$_.WhenCreated}},`
ObjectClass,objectCategory

Write-Host "There are"($children | measure-object).count "child items" `
"under"$dsroot.distinguishedname.value
```

The script stores the child objects in `$children`, which are only containers or organizational units:

```
$children=$dsroot.psbases.children | where {$_.objectcategory -notmatch "Person" `
-AND $_.objectcategory -notmatch "Computer" `
-AND $_.objectcategory -notmatch "Group" `
-AND $_.objectcategory -notmatch "Contact"}
```

I pipe the `$children` object to the **Sort-Object** cmdlet to sort the containers and OUs by name and that output is piped to **Format-List**. As I did earlier, I use custom labels and expressions to return the property values.

When you run this script your output will look like this:

```
PS C:\> c:\scripts\get-childentries.ps1
```

```
DN           : CN=Builtin,DC=mycompany,DC=local
Name        : Builtin
Description  : {}
Created     : 2/24/2008 11:28:59 AM
ObjectClass : {top, builtinDomain}
objectcategory : {CN=Builtin-Domain,CN=Schema,CN=Configuration,DC=mycompany,DC=local}

DN           : OU=Company Desktops,DC=mycompany,DC=local
Name        : Company Desktops
Description  : {}
Created     : 3/4/2008 5:23:07 PM
ObjectClass : {top, organizationalUnit}
```

```
objectcategory : {CN=Organizational-Unit,CN=Schema,CN=Configuration,DC=bigcompany,DC=local}

DN           : OU=Company Shared Contacts,DC=mycompany,DC=local
Name         : Company Shared Contacts
Description  : company contact records
Created      : 3/26/2008 7:30:00 PM
ObjectClass  : {top, organizationalUnit}
objectcategory : {CN=Organizational-Unit,CN=Schema,CN=Configuration,DC=bigcompany,DC=local}

DN           : CN=Computers,DC=mycompany,DC=local
Name         : Computers
Description  : Default container for upgraded computer accounts
Created      : 2/24/2008 11:28:48 AM
ObjectClass  : {top, container}
objectcategory : {CN=Container,CN=Schema,CN=Configuration,DC=mycompany,DC=local}
...
```

The output above is truncated. Your results will vary depending on your domain.

What about a recursive search through the domain or any starting point for that matter? It's a little more complicated but achievable. Here's my solution:

Get-DSTree.ps1

```
Function Get-DSTree {
    Param([string]$container,[int]$i=0)

    [string]$rootDN="LDAP://" + $container
    [string]$leader=" "
    [int]$pad=$leader.length+$i

    Write-Host ($leader.Padleft($pad)+$container)

    $dse=New-Object DirectoryServices.DirectoryEntry $rootDN

    $dse.psbase.children | where {$_.objectcategory -notmatch "Person" `
    -AND $_.objectcategory -notmatch "Computer" `
    -AND $_.objectcategory -notmatch "Group" `
    -AND $_.objectcategory -notmatch "Contact"} |
    ForEach-Object {
        [string]$dn=$_.distinguishedName
        Get-DSTree $dn ($pad+1)
    }
}
```

The **Get-DSTree** function takes a directory service distinguished name as a parameter. Because I want the output to visually represent a hierarchy, the second parameter is an integer used in formatting the output.

I created a variable, `$leader`, that is a single space:

```
[string]$leader=" "
```

Then I define a variable that is the length of `$leader`, plus the value of whatever was passed as `$i`:

```
[int]$pad=$leader.length+$i
```

Finally, I write the container name that is preceded by `$leader` and is padded the number of spaces as

specified by \$pad:

```
Write-Host ($leader.Padleft($pad)+$container)
```

You'll see how this works at the end.

Now, I need to get the DirectoryServices.DirectoryEntry object:

```
$dse=New-Object DirectoryServices.DirectoryEntry $rootDN
```

I get the child objects and filter out everything except the containers and organizational units I want to display:

```
$dse.psbase.children | where {$_.objectcategory -notmatch "Person" `
-AND $_.objectcategory -notmatch "Computer" `
-AND $_.objectcategory -notmatch "Group" `
-AND $_.objectcategory -notmatch "Contact" }
```

That output is piped to **ForEach-Object**, which recursively calls the **Get-DSTree** function, incrementing the pad value by 1:

```
ForEach-Object {
    [string]$dn=$_.distinguishedName
    Get-DSTree $dn ($pad+1)
}
```

Here's an excerpt of the output:

```
PS C:\> Get-DSTree "dc=mycompany,dc=local"
DC=mycompany,DC=local
  CN=Builtin,DC=mycompany,DC=local
  OU=Company Desktops,DC=mycompany,DC=local
    OU=Customer Service,OU=Company Desktops,DC=mycompany,DC=local
    OU=Finance,OU=Company Desktops,DC=mycompany,DC=local
    OU=IT Desktops,OU=Company Desktops,DC=mycompany,DC=local
    OU=Sales,OU=Company Desktops,DC=mycompany,DC=local
    OU=Training,OU=Company Desktops,DC=mycompany,DC=local
  OU=Company Shared Contacts,DC=mycompany,DC=local
  CN=Computers,DC=mycompany,DC=local
  OU=Divisions,DC=mycompany,DC=local
    OU=Boston,OU=Divisions,DC=mycompany,DC=local
    OU=LA,OU=Divisions,DC=mycompany,DC=local
  OU=Domain Controllers,DC=mycompany,DC=local
  OU=Employees,DC=mycompany,DC=local
    OU=Consumer Affairs,OU=Employees,DC=mycompany,DC=local
...
```

Please be aware that in a large Active Directory environment this script may take a while to complete. The filtering is done post processing, after the retrieval of the \$dse object. There are better ways to search large volumes of data which I discuss below.

Using the DirectorySearcher

By now you may be thinking to yourself, “That’s a lot of work to find stuff in Active Directory.” Fortunately, the .NET Framework provides a better solution, the `DirectorySearcher` class:

```
PS C:\> $searcher = New-Object DirectoryServices.DirectorySearcher
PS C:\> $searcher | format-list *
```

```
CacheResults           : True
ClientTimeout          : -00:00:01
PropertyNamesOnly     : False
Filter                 : (objectClass=*)
PageSize               : 0
PropertiesToLoad       : {}
ReferralChasing        : External
SearchScope            : Subtree
ServerPageTimeLimit    : -00:00:01
ServerTimeLimit        : -00:00:01
SizeLimit              : 0
SearchRoot             : System.DirectoryServices.DirectoryEntry
Sort                   : System.DirectoryServices.SortOption
Asynchronous           : False
Tombstone              : False
AttributeScopeQuery    :
DerefAlias             : Never
SecurityMasks          : None
ExtendedDN             : None
DirectorySynchronization :
VirtualListView        :
Site                   :
Container              :
```

This object has the methods listed in Table 5-2, which you can discover for yourself:

```
$searcher = New-Object DirectoryServices.DirectorySearcher
$searcher | get-member -membertype method
```

Table 5-2 Directory Searcher Methods

<code>add_Disposed</code>	<code>CreateObjRef</code>	<code>Dispose</code>
<code>Equals</code>	<code>FindAll</code>	<code>FindOne</code>
<code>GetHashCode</code>	<code>GetLifetimeService</code>	<code>GetType</code>
<code>get_Asynchronous</code>	<code>get_AttributeScopeQuery</code>	<code>get_CacheResults</code>
<code>get_ClientTimeout</code>	<code>get_Container</code>	<code>get_DerefAlias</code>
<code>get_DirectorySynchronization</code>	<code>get_ExtendedDN</code>	<code>get_Filter</code>
<code>get_PageSize</code>	<code>get_PropertiesToLoad</code>	<code>get_PropertyNamesOnly</code>
<code>get_ReferralChasing</code>	<code>get_SearchRoot</code>	<code>get_SearchScope</code>
<code>get_SecurityMasks</code>	<code>get_ServerPageTimeLimit</code>	<code>get_ServerTimeLimit</code>
<code>get_Site</code>	<code>get_SizeLimit</code>	<code>get_Sort</code>
<code>get_Tombstone</code>	<code>get_VirtualListView</code>	<code>InitializeLifetimeService</code>
<code>remove_Disposed</code>	<code>set_Asynchronous</code>	<code>set_AttributeScopeQuery</code>
<code>set_CacheResults</code>	<code>set_ClientTimeout</code>	<code>set_DerefAlias</code>
<code>set_DirectorySynchronization</code>	<code>set_ExtendedDN</code>	<code>set_Filter</code>

set_PageSize	set_PropertyNamesOnly	set_ReferralChasing
set_SearchRoot	set_SearchScope	set_SecurityMasks
set_ServerPageTimeLimit	set_ServerTimeLimit	set_Site
set_SizeLimit	set_Sort	set_Tombstone
set_VirtualListView	To-String	

I know it looks like a daunting list, but you really only need a few of these methods for searching. By default, the directory searcher is configured to search for all objects:

```
Filter : (objectClass=*)
```

I'll show you how to change this a little later. The directory searcher begins its search from the SearchRoot:

```
SearchRoot : System.DirectoryServices.DirectoryEntry
```

You can see this property is a DirectoryEntry object like the one used earlier. In the current example, the SearchRoot is the domain root:

```
PS C:\> $searcher.searchroot
```

```
distinguishedName
-----
{DC=mycompany,DC=local}
```

The directory searcher has two primary methods: **FindOne()** and **FindAll()**. The first method will stop searching after the first object meeting the search criteria is found:

```
PS C:\> $searcher.findone() | format-list
```

```
Path : LDAP://DC=mycompany,DC=local
Properties : {fsmoroleowner, minpwdlength, adspath, msds-perusertrusttombstonesquota...}
```

The other method returns all objects in my Active Directory:

```
PS C:\> $searcher.findall() | select path
```

```
Path
----
LDAP://DC=mycompany,DC=local
LDAP://CN=Users,DC=mycompany,DC=local
LDAP://CN=Computers,DC=mycompany,DC=local
LDAP://OU=Domain Controllers,DC=mycompany,DC=local
LDAP://CN=System,DC=mycompany,DC=local
LDAP://CN=LostAndFound,DC=mycompany,DC=local
LDAP://CN=Infrastructure,DC=mycompany,DC=local
LDAP://CN=ForeignSecurityPrincipals,DC=mycompany,DC=local
LDAP://CN=Program Data,DC=mycompany,DC=local
LDAP://CN=Microsoft,CN=Program Data,DC=mycompany,DC=local
LDAP://CN=NTDS Quotas,DC=mycompany,DC=local
LDAP://CN=WinsockServices,CN=System,DC=mycompany,DC=local
```

```
LDAP://CN=RpcServices,CN=System,DC=mycompany,DC=local
LDAP://CN=FileLinks,CN=System,DC=mycompany,DC=local
LDAP://CN=VolumeTable,CN=FileLinks,CN=System,DC=mycompany,DC=local
LDAP://CN=ObjectMoveTable,CN=FileLinks,CN=System,DC=mycompany,DC=local
LDAP://CN=Default Domain Policy,CN=System,DC=mycompany,DC=local
LDAP://CN=AppCategories,CN=Default Domain Policy,CN=System,DC=mycompany,DC=local
...
```

Of course, you may want a more limited search query. The **Filter** property can be defined with any LDAP query:

```
PS C:\> $searcher.filter="(objectClass=organizationalunit)"
PS C:\> $searcher.findall() | Select Path
```

```
Path
----
LDAP://OU=Domain Controllers,DC=mycompany,DC=local
LDAP://OU=home,DC=mycompany,DC=local
LDAP://OU=Servers,DC=mycompany,DC=local
LDAP://OU=Testing,DC=mycompany,DC=local
LDAP://OU=omaha,DC=mycompany,DC=local
LDAP://OU=syracuse,DC=mycompany,DC=local
LDAP://OU=University,OU=syracuse,DC=mycompany,DC=local
LDAP://OU=Demo OU,OU=Testing,DC=mycompany,DC=local
LDAP://OU=sapien,DC=mycompany,DC=local
LDAP://OU=Demo,OU=Servers,DC=mycompany,DC=local
LDAP://OU=Admini,DC=mycompany,DC=local
```

The directory searcher is looking for all organizational unit objects and PowerShell is displaying just the **Path** property.

Below is a slightly more complicated example:

```
PS C:\> $searcher.Filter="(&(objectcategory=person)(objectclass=user))"
PS C:\> $searcher.findall() | Select Path
```

```
Path
----
LDAP://CN=Administrator,CN=Users,DC=mycompany,DC=local
LDAP://CN=Guest,CN=Users,DC=mycompany,DC=local
LDAP://CN=SUPPORT_388945a0,CN=Users,DC=mycompany,DC=local
LDAP://CN=krbtgt,CN=Users,DC=mycompany,DC=local
LDAP://CN=Jeffery Hicks,OU=home,DC=mycompany,DC=local
LDAP://CN=IUSR_MYDC01,CN=Users,DC=mycompany,DC=local
LDAP://CN=IWAM_MYDC01,CN=Users,DC=mycompany,DC=local
LDAP://CN=svcSharepoint,CN=Users,DC=mycompany,DC=local
LDAP://CN=Joe Hacker,OU=Testing,DC=mycompany,DC=local
LDAP://CN=Jane Cracker,OU=Testing,DC=mycompany,DC=local
LDAP://CN=ldog,OU=Testing,DC=mycompany,DC=local
LDAP://CN=test user1,OU=omaha,DC=mycompany,DC=local
LDAP://CN=Test User3,OU=Testing,DC=mycompany,DC=local
LDAP://CN=Test User4,OU=Testing,DC=mycompany,DC=local
LDAP://CN=Lucky Dog,OU=home,DC=mycompany,DC=local
LDAP://CN=da Hicks,CN=Users,DC=mycompany,DC=local
LDAP://CN=Steve McQueen,OU=Testing,DC=mycompany,DC=local
LDAP://CN=Bill Shakespeare,OU=Testing,DC=mycompany,DC=local
...
```

In this example, the search filter is a combination that should return all user objects in my domain. The output is truncated.

The search filters can get quite complicated. In fact, you can take any LDAP query that you create using the Saved Queries feature in Active Directory Users and Computers and use it in PowerShell:

```
PS C:\>$searcher.filter=`
>>"(&(objectCategory=person)(objectClass=user)(userAccountControl:1.2.840.113556.1.4.803:=2))"
>>
PS C:\> $searcher.findall() | Select Path

Path
----
LDAP://CN=Guest,CN=Users,DC=mycompany,DC=local
LDAP://CN=SUPPORT_388945a0,CN=Users,DC=mycompany,DC=local
LDAP://CN=krbtgt,CN=Users,DC=mycompany,DC=local
LDAP://CN=Joe Hacker,OU=Testing,DC=mycompany,DC=local
LDAP://CN=svcSonicAdmin,CN=Users,DC=mycompany,DC=local
LDAP://CN=ldog,OU=Testing,DC=mycompany,DC=local
LDAP://CN=test user1,OU=omaha,DC=mycompany,DC=local
LDAP://CN=Test User3,OU=Testing,DC=mycompany,DC=local
LDAP://CN=Test User4,OU=Testing,DC=mycompany,DC=local
LDAP://CN=Steve McQueen,OU=Testing,DC=mycompany,DC=local
LDAP://CN=azygot,OU=Testing,DC=mycompany,DC=local
LDAP://CN=Paul Jones,OU=Testing,DC=mycompany,DC=local
LDAP://CN=mapplebee,OU=Testing,DC=mycompany,DC=local
LDAP://CN=krogers,OU=Testing,DC=mycompany,DC=local
LDAP://CN=jjensen,OU=Testing,DC=mycompany,DC=local
LDAP://CN=stoobin,OU=Testing,DC=mycompany,DC=local
LDAP://CN=tsykes,OU=Testing,DC=mycompany,DC=local
LDAP://CN=amorales,OU=Testing,DC=mycompany,DC=local
```

I've taken the LDAP query to find all disabled users and used it as my search filter.

If you pipe the search results, you will see that you don't get the actual directory service entry, but rather a result object:

```
TypeName: System.DirectoryServices.SearchResult
```

Name	MemberType	Definition
Equals	Method	System.Boolean Equals(Object obj)
GetDirectoryEntry	Method	System.DirectoryServices.DirectoryEntry GetDirectoryEntry()
GetHashCode	Method	System.Int32 GetHashCode()
GetType	Method	System.Type GetType()
get_Path	Method	System.String get_Path()
get_Properties	Method	System.DirectoryServices.ResultPropertyCollection get_Properties()
ToString	Method	System.String ToString()
Path	Property	System.String Path {get;}
Properties	Property	System.DirectoryServices.ResultPropertyCollection Properties {get;}

This is important to know so you can understand how to get more information out of the search result. For example, the example below shows how to find all the disabled accounts, and display more information about each user:

Get-DisabledUserReport.ps1

```
Function Get-PwdLastSetDate {

    Param([int64]$LastSet=0)
    if ($LastSet -eq 0) {
        write "Never Set or Re-Set"
    } else {
        [datetime]$utc="1/1/1601"
        $i=$LastSet/864000000000
        write ($utc.AddDays($i))
    }
}

$searcher = New-Object DirectoryServices.DirectorySearcher

$searcher.filter="
(&(objectCategory=person)(objectClass=user)(userAccountControl:1.2.840.113556.1.4.803:=2))"

$searcher.findall() | Format-Table `
@{label="DN";expression={$_.properties.distinguishedname}},`
@{label="Last Modified";expression={$_.properties.whenchanged}},`
@{label="PwdLastSet";expression={`
Get-PwdLastSetDate $_.properties.item("pwdlastset")[0]}} `
-autosize
```

You've seen most of this script already. One thing that is new is the function I wrote to convert the value of **PwdLastSet** to a more meaningful value.

```
Function Get-PwdLastSetDate {

    Param([int64]$LastSet=0)
    if ($LastSet -eq 0) {
        write "Never Set or Re-Set"
    } else {
        [datetime]$utc="1/1/1601"
        $i=$LastSet/864000000000
        write ($utc.AddDays($i))
    }
}
```

The function takes the **PwdLastSet** value, which is a 64bit integer, as a parameter. This value is the number of 100 nanosecond intervals since January 1, 1601 when the user's password was last set. If the value is 0, then the password was never set or the account was flagged to force the user to change password at next logon. The function returns a message indicating that:

```
if ($LastSet -eq 0) {
    write "Never Set or Re-Set"
```

Otherwise, I divide the **PwdLastSet** value by 864000000000 to get a value in days:

```
$i=$LastSet/864000000000
```

To calculate what day that was, I use the value as a parameter for the **AddDays()** method from my [datetime] object:

```
write ($utc.AddDays($i))
```

Something else you might notice is the syntax I used to get the **PwdLastSet** value:

```
Get-PwdLastSetDate $_.properties.item("pwdlastset")[0]
```

PowerShell is usually pretty forgiving and flexible. It will do what it can to properly format data, which is why this works:

```
@{label="DN";expression={$_.properties.distinguishedname}},`
```

However, because I need to pass the value to the **Get-PwdLastSetDate** function, something like this fails:

```
Get-PwdLastSetDate $_.properties.pwdlastset
```

Do you recall what type of object the DirectorySearcher returned? It is a SearchResult object, not a user object, and the properties are stored as a collection. Therefore, I need more specific syntax to return a value that I can pass to the function. Even though there is only one value, the **PwdLastSet** property is a treated as a collection, so I specify that I want the first item in the collection, [0].

Here's the end result:

```
PS C:\> C:\scripts\get-disableduserreport.ps1
```

DN	Last Modified	PWDLastSet
--	-----	-----
CN=Guest,CN=Users,DC=mycompany,DC=local	4/14/2004 10:52:30 PM	Never Set or Re-Set
CN=SUPPORT_388945a0,CN=Users,DC=mycompany,DC=local	4/14/2004 10:52:30 PM	3/24/2004 10:41:04 P
CN=krbtgt,CN=Users,DC=mycompany,DC=local	4/14/2004 11:10:43 PM	4/14/2004 10:55:31 P
CN=Joe Hacker,OU=Testing,DC=mycompany,DC=local	6/18/2007 4:01:11 PM	11/12/2004 7:56:57 P
CN=svcSonicAdmin,CN=Users,DC=mycompany,DC=local	1/13/2007 1:35:56 PM	6/15/2004 6:09:31 PM
CN=ldog,OU=Testing,DC=mycompany,DC=local	6/18/2007 4:01:11 PM	6/15/2004 7:40:58 PM
CN=test user1,OU=omaha,DC=mycompany,DC=local	1/25/2007 9:44:52 PM	Never Set or Re-Set
CN=Test User3,OU=Testing,DC=mycompany,DC=local	6/18/2007 4:01:10 PM	Never Set or Re-Set
CN=Test User4,OU=Testing,DC=mycompany,DC=local	6/18/2007 4:01:10 PM	Never Set or Re-Set
CN=Steve McQueen,OU=Testing,DC=mycompany,DC=local	6/18/2007 4:01:10 PM	Never Set or Re-Set
CN=azygot,OU=Testing,DC=mycompany,DC=local	6/18/2007 4:01:10 PM	6/5/2005 11:44:27 PM
CN=Paul Jones,OU=Testing,DC=mycompany,DC=local	6/18/2007 4:01:10 PM	6/5/2005 11:45:46 PM
CN=mapplebee,OU=Testing,DC=mycompany,DC=local	6/18/2007 4:01:10 PM	6/5/2005 11:45:47 PM
CN=krogers,OU=Testing,DC=mycompany,DC=local	6/18/2007 4:01:10 PM	6/5/2005 11:45:47 PM
CN=jjensen,OU=Testing,DC=mycompany,DC=local	6/18/2007 4:01:10 PM	6/5/2005 11:45:47 PM
CN=stoobin,OU=Testing,DC=mycompany,DC=local	6/18/2007 4:01:11 PM	6/5/2005 11:45:47 PM
CN=tsykes,OU=Testing,DC=mycompany,DC=local	6/18/2007 4:01:11 PM	6/5/2005 11:45:47 PM
CN=amorales,OU=Testing,DC=mycompany,DC=local	6/18/2007 4:01:11 PM	6/5/2005 11:45:47 PM

I will show another search example that returns information about a user account, including when their password was last set and the password age in days.

Get-PasswordReport.ps1

```
#Get-PasswordReport.ps1
```

```
Function Get-PasswordReport {
# This function writes a custom object for each non-disabled
# user account in your Active Directory domain to the pipeline

    Function Get-PwdLastSetDate {

        Param([int64]$LastSet=0)
        if ($LastSet -eq 0) {
            write "Never Set or Re-Set"
        } else {
            [datetime]$utc="1/1/1601"
            $i=$LastSet/86400000000
            write ($utc.AddDays($i))
        }
    }

    Function Get-PwdAge {

        Param([int64]$LastSet=0)
        if ($LastSet -eq 0) {
            write "0"
        } else {
            [datetime]$ChangeDate=Get-PwdLastSetDate $LastSet
            [datetime]$RightNow=Get-Date

            write $RightNow.Subtract($ChangeDate).Days
        }
    }

    $Searcher = New-Object DirectoryServices.DirectorySearcher

    # find all non-disabled user objects
    $searcher.filter=`
    "(&(objectCategory=person)(objectClass=user)(!userAccountControl:1.2.840.113556.1.4.803:=2))"

    $searcher.findall() | ForEach-Object {

        $obj=New-Object PSObject

        $obj | Add-Member -MemberType NoteProperty -Name "Name" `
        -Value $_.properties.item("name")[0]
        $obj | Add-Member -MemberType NoteProperty -Name "DN" `
        -Value $_.properties.item("distinguishedname")[0]
        $obj | Add-Member -MemberType NoteProperty -Name "Description" `
        -Value $_.properties.item("description")[0]
        $obj | Add-Member -MemberType NoteProperty -Name "AccountCreated" `
        -Value $_.properties.item("whencreated")[0]
        $obj | Add-Member -MemberType NoteProperty -Name "AccountModified" `
        -Value $_.properties.item("whenchanged")[0]
        $obj | Add-Member -MemberType NoteProperty -Name "LastChanged" `
        -Value (Get-PwdLastSetDate $_.properties.item("pwdlastset")[0])
        $obj | Add-Member -MemberType NoteProperty -Name "PasswordAge" `
        -Value (Get-PwdAge $_.properties.item("pwdlastset")[0])

        write $obj
    }
}
}
```

This function uses much of the same code you've already seen. It uses a `DirectorySearcher` object that looks for all non-disabled user accounts:

```
$Searcher = New-Object DirectoryServices.DirectorySearcher

# find all non-disabled user objects
$searcher.filter=`
"(&(objectCategory=person)(objectClass=user)(!userAccountControl:1.2.840.113556.1.4.803:=2))"
```

The search results are piped to the **ForEach-Object** cmdlet. Instead of simply returning the search result values, I create a custom object:

```
$searcher.findall() | ForEach-Object {

    $obj=New-Object PSObject
```

The custom object has properties for the user's name, distinguishedname, description, when the account was created, when it was last changed, when the password was last set, and the password age in days.

Because the **Get-PasswordReport** function returns an object, you can use it in a PowerShell expression like this:

```
PS C:\> get-passwordreport | where {$_.Passwordage -gt 365} | sort PasswordAge -desc | `
>> format-table Name,LastChanged,PasswordAge -auto
>>
```

Name	LastChanged	PasswordAge
IUSR_MYDC01	4/20/2004 4:01:51 PM	1393
IWAM_MYDC01	4/20/2004 4:01:51 PM	1393
svcSharepoint	5/24/2004 5:36:37 PM	1359
Bill Shakespeare	10/27/2004 6:01:01 PM	1203
Jane Cracker	11/12/2004 7:57:13 PM	1187
da Hicks	5/17/2006 1:11:09 AM	637
Jack Frost	10/22/2006 3:36:49 AM	479
Administrator	1/3/2007 10:35:06 PM	405
Jeffery Hicks	1/18/2007 4:38:04 PM	390

Here's one more `DirectorySearcher` example that you might use on a regular basis.

Get-Computers.ps1

```
Function Get-Computers {
# This function writes a custom object for each computer
# object in your Active Directory domain to the pipeline

    Function Get-PwdLastSetDate {

        Param([int64]$LastSet=0)
        if ($LastSet -eq 0) {
            write "Never Set or Re-Set"
        } else {
            [datetime]$utc="1/1/1601"
            $i=$LastSet/864000000000
            write ($utc.AddDays($i))
        }
    }
}
```

```

Function Get-PwdAge {
    Param([int64]$LastSet=0)
    if ($LastSet -eq 0) {
        write "0"
    } else {
        [datetime]$ChangeDate=Get-PwdLastSetDate $LastSet
        [datetime]$RightNow=Get-Date

        write $RightNow.Subtract($ChangeDate).Days
    }
}

$searcher=New-Object DirectoryServices.DirectorySearcher
$searcher.Filter="(&(objectCategory=Computer)(objectClass=Computer))"

$searcher.findall() | ForEach-Object {

$objj=New-Object PSObject

$objj | Add-Member -MemberType NoteProperty -Name "Name" -Value $_.properties.item("name")[0]
$objj | Add-Member -MemberType NoteProperty -Name "DN" `
-Value $_.properties.item("distinguishedname")[0]
$objj | Add-Member -MemberType NoteProperty -Name "DNSName" `
-Value $_.properties.item("dnshostname")[0]
$objj | Add-Member -MemberType NoteProperty -Name "OS" `
-Value $_.properties.item("operatingsystem")[0]
$objj | Add-Member -MemberType NoteProperty -Name "ServicePack" `
-Value $_.properties.item("operatingsystemservicepack")[0]
$objj | Add-Member -MemberType NoteProperty -Name "OSVersion" `
-Value $_.properties.item("operatingsystemversion")[0]
$objj | Add-Member -MemberType NoteProperty -Name "AccountCreated" `
-Value $_.properties.item("whenevercreated")[0]
$objj | Add-Member -MemberType NoteProperty -Name "AccountModified" `
-Value $_.properties.item("whenchanged")[0]
$objj | Add-Member -MemberType NoteProperty -Name "PasswordLastChanged" `
-Value (Get-PwdLastSetDate $_.properties.item("pwdlastset")[0])
$objj | Add-Member -MemberType NoteProperty -Name "PasswordAge" `
-Value (Get-PwdAge $_.properties.item("pwdlastset")[0])

write $objj
}
}

```

The **Get-Computers** function is very similar to the previous example. It also creates a custom object, although as the name indicates, the searcher is looking for computer objects. The function returns a custom object with properties such as name, operating system, and service pack. Even though the function is returning all computer objects, you can use it to find a single computer object as well:

```
PS C:\> get-computers | Where {$_.name -match "dogtoy"}
```

```

Name           : DOGTOY
DN             : CN=DOGTOY,CN=Computers,DC=mycompany,DC=local
DNSName        : dogtoy.mycompany.local
OS             : Windows XP Professional
ServicePack    : Service Pack 2
OSVersion      : 5.1 (2600)
AccountCreated : 6/24/2004 12:10:11 PM
AccountModified : 2/13/2008 11:00:33 AM

```

```
PasswordLastChanged : 2/3/2008 3:15:21 AM
PasswordAge         : 10
```

Perhaps you'd like to find the servers in your Active Directory domain:

```
PS C:\> get-computers | Where {$_.OS -match "server"} | format-table Name,OS,ServicePack -auto
```

```
Name      OS              ServicePack
----      -
MYDC01    Windows Server 2003 Service Pack 2
SWITCH    Windows Server 2003 Service Pack 1
PORTAL    Windows Server 2003 Service Pack 1
```

This function can also help you identify obsolete computer accounts by checking the computer's password age:

```
PS C:\> get-computers | Where {$_.passwordage -gt 45} | sort PasswordAge -desc | `
>> Format-Table DN,PasswordLastChanged,PasswordAge -auto
```

```
>>
DN
--
CN=GODOT,CN=Computers,DC=mycompany,DC=local      10/24/2004 5:43:09 PM      1206
CN=SWITCH,CN=Computers,DC=mycompany,DC=local     9/14/2005 9:07:27 PM      881
CN=PORTAL,OU=Servers,DC=mycompany,DC=local       10/14/2005 11:41:32 PM    851
CN=DemoServer01,OU=Demo,OU=Servers,DC=mycompany,DC=local 3/13/2007 4:11:53 AM      337
CN=TestSrv02,OU=Servers,DC=mycompany,DC=local     3/13/2007 4:11:29 AM      337
CN=TestSrv01,OU=Servers,DC=mycompany,DC=local     3/13/2007 4:11:11 AM      337
CN=TestSrv207,OU=Demo,OU=Servers,DC=mycompany,DC=local 3/13/2007 11:14:09 PM    336
CN=TestSrv206,OU=Demo,OU=Servers,DC=mycompany,DC=local 3/13/2007 11:14:09 PM    336
CN=TestSrv205,OU=Demo,OU=Servers,DC=mycompany,DC=local 3/13/2007 11:14:09 PM    336
CN=TestSrv210,OU=Demo,OU=Servers,DC=mycompany,DC=local 3/13/2007 11:14:10 PM    336
CN=TestSrv209,OU=Demo,OU=Servers,DC=mycompany,DC=local 3/13/2007 11:14:10 PM    336
CN=TestSrv208,OU=Demo,OU=Servers,DC=mycompany,DC=local 3/13/2007 11:14:10 PM    336
CN=TestSrv201,OU=Demo,OU=Servers,DC=mycompany,DC=local 3/13/2007 11:14:09 PM    336
CN=TestSrv200,OU=Demo,OU=Servers,DC=mycompany,DC=local 3/13/2007 11:14:09 PM    336
CN=test123,OU=Demo,OU=Servers,DC=mycompany,DC=local 3/13/2007 11:13:21 PM    336
CN=TestSrv204,OU=Demo,OU=Servers,DC=mycompany,DC=local 3/13/2007 11:14:09 PM    336
CN=TestSrv203,OU=Demo,OU=Servers,DC=mycompany,DC=local 3/13/2007 11:14:09 PM    336
CN=TestSrv202,OU=Demo,OU=Servers,DC=mycompany,DC=local 3/13/2007 11:14:09 PM    336
```

In later chapters, I'll show how you can accomplish much of this with third-party cmdlets.

Fine Tuning the DirectorySearcher

My test domains are not very large, so I can usually get by with many of the DirectorySearcher default properties. However, if you have a larger domain and want to control the performance of your domain controller queries, then there are a few properties you should set. This is especially true if your query will return more than 1000 objects, otherwise you will only get the first 1000 objects.

The first property to tweak is **PageSize** which has a default of 0 and which uses a non-paged query. The domain controller returns the results in pages, which you can think of as a set. Non-paged queries are fine for small domains, but with large domains, performance will be better if you use a value such as:

```
$searcher.pagesize=100
```

Now the searcher will return 100 results per paged query.

The **SizeLimit** property determines how many total results the server returns. The default is 0. In a non-paged query, this will effectively set the maximum number of returned objects to the default server limit of 1000. Of course, you can set it to a smaller number:

```
$searcher.sizelimit=500
```

Now only 500 objects will be returned. However, if you want to ensure you retrieve all objects, set a value for **PageSize** greater than 0, such as 100. The DirectorySearcher will return all objects regardless of the **SizeLimit** value as long as the **PageSize** value is equal or greater than the **SizeLimit**.

Most of the examples in this section are searched from the domain root, which is the default. However, you may prefer to search from a particular organizational unit. In that case, you can specify a new search root like this:

```
$root=New-Object DirectoryServices.DirectoryEntry 'LDAP://OU=Employees,DC=MyCompany,DC=Local'
$searcher.searchroot=$root
```

In addition to modifying the search root, you may need to modify the search scope. The default search scope is Subtree, and the directory searcher will search the root and all child objects. Your other search scope options are Base and OneLevel. A search scope of OneLevel will return child objects within the current root, but it won't search any deeper. It's like SubTree, but without recursion.

```
$searcher.SearchScope="OneLevel"
```

Use the Base search scope when you are searching within a specific object like a group and you want to search based on a certain attribute. This is known as an Attribute Scope Query. Set your search root to a base object and set the search scope to Base.

```
PS C:\> $searcher.searchroot=[ADSI]"LDAP://CN=Sales Staff,OU=Groups,DC=mycompany,DC=Local"
PS C:\> $searcher.searchscope="Base"
```

Now you can specify the attribute you want to search on. In this example, I want to search based on the group's **Member** property.

```
PS C:\> $searcher.AttributeScopeQuery="member"
```

The searcher will search within the group object on the member property, and it should return user accounts. However, I'm going to suggest one more minor tweak. You may only want a subset of properties returned from the query. In that case, add those properties to **PropertiesToLoad**.

```
PS C:\> $searcher.PropertiesToLoad.Add("GivenName")
0
PS C:\> $searcher.PropertiesToLoad.Add("SN")
1
PS C:\> $searcher.PropertiesToLoad.Add("sAMAccountname")
2
```

All that's left is to set the filter and search.

```
PS C:\> $searcher.filter="(&(objectclass=user)(objectcategory=person))"
PS C:\> $searcher.findall() | foreach {
```

```
>> $_ | select @{Name="ADSPATH";Expression={$_.properties.adspath[0]}},`
>>         @{Name="SAM";Expression={$_.properties.samaccountname[0]}},`
>>         @{Name="FirstName";Expression={$_.properties.givenname[0]}},`
>>         @{Name="LastName";Expression={$_.properties.sn[0]}}
>> }
>>
```

ADSPATH	SAM	FirstName	LastName
-----	---	-----	-----
LDAP://CN=Tom Sawyer,OU=Sa...	tsawyer	Tom	Sawyer
LDAP://CN=Sales User,OU=Te...	salesuser	Sales	User
LDAP://CN=Cass Ino,OU=Sale...	cino	Cass	Ino

Finally, you may wish to sort results in a particular order. The **Sort** property is a System.DirectoryServices.SortOption object. With this object, you can specify the property name you wish to sort on and in what order. The default order is ascending.

```
$searcher.sort="Name"
```

Once you've fine tuned your searcher parameters, you can execute your searches using **FindOne()** or **FindAll()**.

Before we leave this topic, I want to return to my earlier directory tree example. Although I like the output, if my domain had more than 1000 container objects, it's not very efficient. Here's a much more efficient approach using the DirectorySearcher:

Get-DSTree2.ps1

```
Function Get-DSTree {
    Param([ADSI]$ADSPATH="LDAP://DC=mycompany,DC=local",[int]$i=0)

    [string]$leader=" "
    [int]$pad=$leader.length+$i

    $searcher=New-Object directoryservices.directorysearcher
    $searcher.pagesize=100
    $searcher.filter="(&(!objectcategory=person)(!objectcategory=computer)" `
    +"(!objectcategory=group)(!objectcategory=contact)(!objectcategory=domain))"
    $searcher.searchScope="OneLevel"
    $searcher.searchRoot=$ADSPATH
    $searcher.PropertiesToLoad.Add("DistinguishedName") | Out-Null

    $searcher.FindAll() | foreach {
        Write-Host ($leader.Padleft($pad)+$.properties.distinguishedname[0])
        Get-DSTree $_.path ($pad+1)
    }
}
```

At first glance, this function appears similar in structure to the first, and it is. The function takes a directory path and an integer to calculate the padded spaces. This gives the output its visual appeal. The function uses the DirectorySearcher and paging.

```
$searcher=New-Object directoryservices.directorysearcher
$searcher.pagesize=100
```

The filter is an LDAP version of my original filter to find container like objects.

```
$searcher.filter="(&(!objectcategory=person)(!objectcategory=computer)" `
+"(!objectcategory=group)(!objectcategory=contact)(!objectcategory=domain))"
```

Because I want to look at each “level,” I’m setting the search scope to OneLevel:

```
$searcher.searchScope="OneLevel"
```

The Directory Searcher will only find matches in the first level of the search root.

```
$searcher.searchRoot=$ADSPath
```

Since I’m only interested in the distinguished name, I’ll take advantage of the **PropertiesToLoad** property and limit my search.

```
$searcher.PropertiesToLoad.Add("DistinguishedName") | Out-Null
```

All that’s left is to start searching. Each result is piped to **ForEach**, which writes a padded space plus the distinguishedname of each object. The object’s path is then passed back to the function to search for any child objects.

```
$searcher.FindAll() | foreach {
    Write-Host ($leader.Padleft($pad)+$_.properties.distinguishedname[0])
    Get-DSTree $_.path ($pad+1)
}
```

This recursive search provides me the same output as I had before. But there is one major difference: my original version took over 26 seconds to complete. This version, using the DirectorySearcher is much more efficient. I got the same output in under five seconds!

The ADSI Type Adapter

When PowerShell was being developed, the product team realized ADSI support would be critical to PowerShell’s adoption and deployment. Unfortunately, as PowerShell architect Jeffrey Snover is fond of saying, “To ship is to choose.” The team couldn’t meet everyone’s expectations with v1.0, though they did add a new type adapter before the final release. This type adapter, [ADSI], gives PowerShell a hint that you will be working with directory service objects. Now, you no longer need to explicitly create System.DirectoryServices.DirectoryEntry objects. To create an object representing your root domain, all you need is this:

```
PS C:\> [ADSI]$dsroot="LDAP://dc=mycompany,dc=local"
PS C:\> $dsroot.whencreated
```

Wednesday, April 14, 2004 10:52:27 PM

Pipe \$dsroot to **Get-Member** and you’ll see that the underlying object isn’t anything other than what we’ve been working with this chapter.

```
PS C:\> $dsroot | get-member
```

```
TypeName: System.DirectoryServices.DirectoryEntry
```

Name	MemberType	Definition
auditingPolicy	Property	System.DirectoryServices.P
creationTime	Property	System.DirectoryServices.P
dc	Property	System.DirectoryServices.P
distinguishedName	Property	System.DirectoryServices.P
dSASignature	Property	System.DirectoryServices.P
...		

When using the [ADSI], you can't specify a connection credential as I did in the example for creating a new System.DirectoryServices.DirectoryEntry object.

Even with the type adapter, you still need the underlying psbase object to get things like child objects:

```
PS C:\> $dsroot.psbase.children
```

```
distinguishedName
-----
{OU=Admini,DC=mycompany,DC=local}
{CN=Builtin,DC=mycompany,DC=local}
{CN=Computers,DC=mycompany,DC=local}
{OU=Domain Controllers,DC=mycompany,DC=local}
{CN=ForeignSecurityPrincipals,DC=mycompany,DC=local}
{OU=home,DC=mycompany,DC=local}
{CN=Infrastructure,DC=mycompany,DC=local}
{CN=LostAndFound,DC=mycompany,DC=local}
{CN=NTDS Quotas,DC=mycompany,DC=local}
{OU=omaha,DC=mycompany,DC=local}
{CN=Program Data,DC=mycompany,DC=local}
{OU=sapien,DC=mycompany,DC=local}
{OU=Servers,DC=mycompany,DC=local}
{OU=syracuse,DC=mycompany,DC=local}
{CN=System,DC=mycompany,DC=local}
{OU=Testing,DC=mycompany,DC=local}
{CN=Users,DC=mycompany,DC=local}
```

The type adapter provides simplicity. You can create an ADSI object with a single line of code and access all of its properties:

```
PS C:\> [ADSI]$jeff="LDAP://CN=Jeffery Hicks,OU=home,DC=mycompany,DC=local"
PS C:\> $jeff.telephonenumber
555-1234
PS C:\> $jeff | Select Displayname,Department,Title,Company
```

displayName	department	title	company
{Jeffery Hicks}	{Projects,Training and Ser...	{SAPIEN Guru}	{SAPIEN}

The adaptation isn't perfect. For example, some properties like **PWDLastSet** are still stored as COM objects, so I need to use the functions I created earlier in the chapter to read the property:

```
PS C:\> get-pwdage (convert-adslargeinteger $jeff.pwdlastset.value)
391
PS C:\> get-pwdlastsetdate (convert-adslargeinteger $jeff.pwdlastset.value)
```

Thursday, January 18, 2007 4:38:04 PM

When you look at an object like `$jeff` piped to **Get-Member**, you'll see not every possible ADSI property and method is listed. However, if you know the property or method name, you can still call it. Here's an example:

```
PS C:\> [ADSI]$rgbiv="LDAP://CN=Roy G. Biv,OU=Employees,DC=SAPIEN,dc=local"
PS C:\> $rgbiv.SetPassword("N3wP@$")
PS C:\> $rgbiv.SetInfo()
```

You'll never see a **SetInfo()** method, yet that is the method to call when setting a user's password. There are some things you are expected to already know.

Once you have the knowledge, you can use a type adapter like this to create a user account in a given organizational unit:

```
PS C:\> [ADSI]$ou="LDAP://OU=Employees,DC=SAPIEN,dc=local"
PS C:\> $user=$ou.Create("user","CN=Jim Shortz")
PS C:\> $user.put("samaccountname","jshortz")
PS C:\> $user.setinfo()
PS C:\> $user.SetPassword("P@ssw0rd")
PS C:\> $user.put("title","Health Director")
PS C:\> $user.put("userprincipalname","jshortz@sapien.com")
PS C:\> $user.put("givenname","Jim")
PS C:\> $user.put("sn","Shortz")
PS C:\> #enable the user account
PS C:\> $user.put("Useraccountcontrol",544)
PS C:\> $user.put("DisplayName","Jim Shortz")
PS C:\> $user.SetInfo()
```

The `$ou` variable will be a `DirectoryServices.DirectoryEntry` type which I already know has a **Create()** method for creating child objects:

```
PS C:\> $user=$ou.Create("user","CN=Jim Shortz")
```

The `$user` object will automatically be defined also as a `DirectoryServices.DirectoryEntry` object. I know that the **samAccountName** property must be defined and the object saved before I can make any other changes to it:

```
PS C:\> $user.put("samaccountname","jshortz")
PS C:\> $user.setinfo()
```

Once the object is committed to Active Directory I can go ahead and set the rest of the user properties. If I wasn't sure about property name, I could pipe `$user` to **Get-Member** to discover them.

If this chapter seems like a lot of work, well...it is. The .NET Framework classes PowerShell requires abstraction, ostensibly to make life easier for you, isn't as complete as either you or the PowerShell team for the user, service management aren't the easiest to work with for non-developers and PowerShell team

would like. In later chapters, I'll cover some third-party cmdlets which will make this easier, although even those cmdlets rely on the underlying .NET Framework classes so understanding them now will definitely help you out later.