**Working with Databases**

There are a lot of ways to work with databases in PowerShell because, under the hood, it uses ADO.NET. ADO.NET provides a few different ways to work with databases. It's important for you to know that *no one* *way is better than any other*. In fact, the only reason different ways exist is that some of them are easier for certain tasks than others. For example, it's possible to create a "database" entirely in memory in PowerShell without an actual physical database existing underneath. The following example *is* *not* a script, it's just being run interactively at the command-line:

PS C:\> $table = new-object system.data.datatable "Machines"

PS C:\> $col1 = new-object system.data.datacolumn userid,([string])

PS C:\> $col2 = new-object system.data.datacolumn comp,([string])

PS C:\> $table.columns.add($col1)

PS C:\> $table.columns.add($col2)

PS C:\> $row1 = $table.newrow()

PS C:\> $row1.userid = "Don"

PS C:\> $row1.comp = "DON-PC"

PS C:\> $table.rows.add($row1)

PS C:\>

PS C:\> $table

userid comp

------ ----

Don DON-PC

PS C:\> ($table.rows)[0]

userid comp

------ ----

Don DON-PC

PS C:\>

Even though this isn't complicated, it *is* unlike anything else we've done in PowerShell. So, let's walk through it one line at a time. First, we'll use the **New-Object** cmdlet to create a new .NET object, the System.Data.Datatable object:

PS C:\> $table = new-object system.data.datatable "Machines"

Next, we'll create two new columns for the table, naming them UserID and Comp. We'll also make sure they're of the String type ([Appendix B](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/appendix-B-251.xhtml#ap02) lists more types you can use):

PS C:\> $col1 = new-object system.data.datacolumn userid,([string])

PS C:\> $col2 = new-object system.data.datacolumn comp,([string])

With the columns created, we need to attach them to the table itself:

PS C:\> $table.columns.add($col1)

PS C:\> $table.columns.add($col2)

Now we'll ask the table to create a new row. Note that this row exists entirely independently of the table at this point. Now we'll populate both columns with some data:

PS C:\> $row1 = $table.newrow()

PS C:\> $row1.userid = "Don"

PS C:\> $row1.comp = "DON-PC"

Next we append the row to the actual table:

PS C:\> $table.rows.add($row1)

PS C:\>

Viewing the table displays all the rows we've added:

PS C:\> $table

userid comp

------ ----

Don DON-PC

We can refer to a specific row like this:

PS C:\> ($table.rows)[0]

userid comp

------ ----

Don DON-PC

PS C:\>

So that's pretty straightforward. However, it doesn't show you how to make PowerShell utilize an existing database such as a SQL Server database or an Access database. However, it is useful to understand this technique of creating a DataTable in memory and using it without a real database underneath. For example, DataTables have methods that let you save and load data from XML files (the **ReadXml** and **WriteXml** methods). This gives you the ability to use portable data sources in your scripts. You can read more about the DataTable in Microsoft's .NET SDK documentation that is available online at [http://library.msdn.com](http://library.msdn.com/).

When you create a DataTable in this fashion, keep in mind it only lasts for the duration of the scope. So, the DataTable we created interactively in the shell in this example will exist until the shell is closed. If we put this database into a script, it will last until the script finishes running.

**Persisting DataTables**

Here's another example to start with:

PS C:\> $table = new-object system.data.datatable "Machines"

PS C:\> $comp = new-object system.data.datacolumn id,([string])

PS C:\> $ver = new-object system.data.datacolumn ver,([string])

PS C:\> $table.columns.add($comp)

PS C:\> $table.columns.add($ver)

PS C:\> $row = $table.newrow()

PS C:\> $row.id = "DON-PC"

PS C:\> $row.ver = "2"

PS C:\> $table.rows.add($row)

PS C:\> $row2 = $table.newrow()

PS C:\> $row2.id = "CHRIS-PC"

PS C:\> $row2.ver = "3"

PS C:\> $table.rows.add($row2)

PS C:\> $table

id ver

-- ---

DON-PC 2

CHRIS-PC 3

Okay, so we have a DataTable with two rows. Now we'll save it in a file to persist the data so it lasts beyond our current PowerShell session:

PS C:\> $table.writexml("c:\mydata.xml")

Easy! If you open the resulting file, you will see something like this:

<?xml version="1.0" standalone="yes"?>

<DocumentElement>

<Machines>

<id>DON-PC</id>

<ver>2</ver>

</Machines>

<Machines>

<id>CHRIS-PC</id>

<ver>3</ver>

</Machines>

</DocumentElement>

This is useful for a number of reasons. First, if you're familiar with XML Style Sheets (XSL/XSLT), you can write a corresponding style sheet and have a Web browser like Internet Explorer display your XML file as formatted data. This is a perfect way to create a nice report! However, as-is this file is missing important schema information about how the table is structured. You can also persist that:

PS C:\> $table.writexmlschema("c:\mydata.xsd")

This creates a file that describes how the table is structured like this:

<?xml version="1.0" standalone="yes"?>

<xs:schema id="NewDataSet" xmlns=""

xmlns:xs="<http://www.w3.org/2001/XMLSchema>"

xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">

<xs:element name="NewDataSet" msdata:IsDataSet="true"

msdata:MainDataTable="Machines" msdata:UseCurrentLocale="true">

<xs:complexType>

<xs:choice minOccurs="0" maxOccurs="unbounded">

<xs:element name="Machines">

<xs:complexType>

<xs:sequence>

<xs:element name="id" type="xs:string" minOccurs="0" />

<xs:element name="ver" type="xs:string" minOccurs="0" />

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:choice>

</xs:complexType>

</xs:element>

</xs:schema>

With the XML data and the XSD schema, you can re-create the DataTable in a future PowerShell session by loading the files in:

PS C:\> $table2.readxmlschema("c:\mydata.xsd")

PS C:\> $table2.readxml("c:\mydata.xml")

Here are the results:

PS C:\> $table2

id ver

-- ---

DON-PC 2

CHRIS-PC 3

Extending this idea, below is a more useful example. Datatable.ps1 reads a file named C:\Computers.txt. This file is expected to contain one username per line. For each computer, it attempts to retrieve the current service pack version, and the writes all of that information into an XML file. Note that there's no error handling since each computer is expected to be online and accessible. Keep in mind that you must have permission to remotely use WMI on each computer listed.

**Datatable.ps1**

#create table structure

$table = new-object system.data.datatable "SPInfo"

$col1 = new-object system.data.datacolumn machine,([string])

$col2 = new-object system.data.datacolumn ver,([string])

$table.columns.add($col1)

$table.columns.add($col2)

#get computer names from a file

$names = get-content "c:\computers.txt"

#go through names

foreach ($name in $names) {

$sp = get-wmiobject Win32\_OperatingSystem `

-property ServicePackMajorVersion `

-computername $name

$row = $table.newrow()

$row.machine = $name

$row.ver = $sp.servicepackmajorversion

$table.rows.add($row)

}

#save table

$table.writexml("c:\versions.xml")

$table.writexmlschema("c:\versions.xsd")

Our test output looks like this:

<?xml version="1.0" standalone="yes" ?>

<DocumentElement>

<SPInfo>

<machine>don-pc</machine>

<ver>2</ver>

</SPInfo>

<SPInfo>

<machine>testbed</machine>

<ver>1</ver>

</SPInfo>

</DocumentElement>

However, the *real* power in database scripting is being able to connect to a *real* database such as SQL Server or Access.

**Connecting to a Database**

If you want to work with an existing database, we prefer a different technique - using a DataReader. However, before you can do that, you need to tell PowerShell where the database resides. There are two ways to connect to a database:

*If you're using a SQL Server database, which has special support in .NET, you'll use a SqlConnection object.*

*For other databases, you'll use an OleDbConnection object.*

Both methods are used in conjunction with a *connection string,* which is essentially a set of information that tells .NET where the database lives and how to log in, if necessary. Here's an example of a SQL Server connection:

PS C:\> $conn = new-object system.data.sqlclient.sqlconnection

PS C:\> $conn.connectionstring = "server=don-pc;database=mydata;" `

>> + "trusted\_connection=true;"

>>

PS C:\> $conn.open()

If you're wondering where we got the connection string - well, we cheated. Point your Web browser to [www.connectionstrings.com](http://www.connectionstrings.com/), select the type of database you want, and you'll find sample connection strings. You'll typically want to look for the example named "SqlConnection" or "OleDbConnection" unless you specifically know otherwise. You *will* need to know a bit about your database in order to connect. For instance, in the above example, we knew that the SQL Server was using integrated authentication, which is why we added "trusted\_connection=true" to the connection string. For non-integrated authentication, you would add "user id=*username*;password=*pass*;" to the connection string, filling in the appropriate username and password.

Okay, now that you know how to establish a SQL Server connection, we'll show you how to connect to a database an administrator might use more often such as Microsoft Access. Here's an example of that you'll notice looks very similar, but uses the OleDbConnection object:

PS C:\> $conn = new-object system.data.oledb.oledbconnection

PS C:\> $conn.connectionstring = "provider=microsoft.jet.oledb.4.0;" `

>> + "data source=c:\mydata.mdb;"

>>

PS C:\> $conn.open()

PS C:\>

Again, we swiped the part of the connection string that changes from database to database from [www.connectionstrings.com](http://www.connectionstrings.com/).

**Defining a Query**

With an open connection to the database, you can issue queries. Some queries return data, while others modify existing data in some fashion. There are four basic types of queries that you will write in the industrystandard SQL (structured query language) syntax:

* SELECT
* INSERT
* UPDATE
* DELETE

This chapter is not intended to be an exhaustive reference to the SQL syntax. Instead, we'll give you a crash course and enough examples to let you be effective.

**Querying Existing Data**

To query data from the database, you'll need to know the table name, the column names you want, and any criteria you want to use to limit the data returned. For example, for a table named MyTable, I can query just columns Col1 and Col2 like this:

SELECT Col1,Col2 FROM MyTable

**Note**

This isn't case-sensitive, however in this book we will capitalize SQL keywords to make them stand out a bit more.

This query returns every row from the table. However, if you want the rows where the value in Col1 is "Don," you would do this instead:

SELECT Col1,Col2 FROM MyTable WHERE Col1= 'Don'

Notice that the string value is in *single quotation marks.* Numeric values don't go inside quotes or anything else. For *most* databases, dates also go into single quotes:

SELECT Col1,Col2 FROM MyTable WHERE Col1 = '1/1/2006'

However, for Access, hash marks are used around dates:

SELECT Col1,Col2 FROM MyTable WHERE Col1 = #1/1/2006#

The WHERE clause can include any normal operators such as:

* = equality
* > greater than
* < less than
* >= greater than or equal to
* <= less than or equal to

So, to query a range where Col1 contains values between 5 and 7 inclusive:

SELECT Col1,Col2 FROM MyTable WHERE Col1 >= 5 AND Col1 <= 7

Alternatively, you can use the special BETWEEN operator that works with most major database systems including SQL Server and Access:

SELECT Col1,Col2 FROM MyTable WHERE Col1 BETWEEN 5 AND 7

That's it - a crash course in querying data. Of course, in PowerShell you'll want to put your query into a variable to make it a bit easier to use:

PS C:\> $query = "select col1,col2 from mytable"

The only other tricky part is when you want to refer to a string value that contains the ' character. Since the SQL syntax uses the single quote as a string delimiter, having the same character appear inside a string confuses things. For example:

$q = "SELECT Col1 FROM MyTable WHERE Col2 = 'Don's'"

As you can imagine, this will cause problems for you. To include a single quote *within* a string, you need to double up on the single quotes:

$q = "SELECT Col1 FROM MyTable WHERE Col2 = 'Don''s'"

Doing so makes "Don's" a literal value that the database can handle. You can do this with a variable, too:

$query = [string]$var.replace("'","''")

This replaces all instances of a single quote with two single quotes. Even though this takes some practice, you will get used to it.

**Adding New Data**

An INSERT query is used to add new data to a table. You specify the list of columns that you're adding, and then a list of values, one value per column. In most cases, you need to specify *every* column in the table or the database may reject your change. Here's an example:

INSERT INTO MyTable (col1,col2) VALUES('Don','DON-PC')

Notice that the VALUES list goes in the same order as the column list you provided: "Don" goes into Col1 because they occupy the same position in their respective lists. Making this into a string variable makes it easier to use in PowerShell:

$q = "INSERT INTO MyTable (col1,col2) VALUES('Don','DON-PC')"

However, you're more likely to have the values inside variables. In this case these values are "Don" and "DON-PC". So, you can let PowerShell insert them for you:

$q = "INSERT INTO MyTable (col1,col2) VALUES('$val1','$val2')"

Because the string being placed into $q is delimited by double quotes, PowerShell evaluates any variables inside. This brief example proves that the *contents* of $val and $val1 are going into $q:

PS C:\> $val1="Don"

PS C:\> $val2="DON-PC"

PS C:\> $q = "INSERT INTO MyTable (col1,col2) VALUES('$val1','$val2')"

PS C:\> $q

INSERT INTO MyTable (col1,col2) VALUES('Don','DON-PC')

PS C:\>

**Updating Data**

The UPDATE query changes existing data. Typically, you'll include a WHERE clause to limit the effects to just a few rows. For example:

UPDATE MyTable SET Col1 = 'JOE' WHERE Col2 = 'JOES-PC'

Again, placing this into a string variable using the technique discussed in the [previous section](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/section-101.xhtml#ch14lev3sec2) makes it easier to use within PowerShell.

**Deleting Data**

Last but not least, the DELETE query will remove data. You'll nearly always use this with a WHERE clause, otherwise you'll delete *all* of the data in your table!

DELETE FROM MyTable WHERE Col1 = 'UNKNOWN'

Place the query into a variable to make it easier to use.

**Executing the Query**

Once you've defined a query, it can be executed. We'll assume you've put it into a variable named $query for the following examples. To execute this query you need a SqlCommand or OleDbCommand object. Be sure to pick the one that matches the type of connection you created.

In the case of INSERT, UPDATE, and DELETE queries, you won't get any data back, so you'll use the command object's **ExecuteNonQuery**method. However, first you need to connect the new command object to the open database connection. Here's a complete example:

PS C:\> $cmd = new-object system.data.oledb.oledbcommand

PS C:\> $cmd.connection = $conn

PS C:\> $cmd.commandtext = $query

PS C:\> $cmd.executenonquery()

This assumes that $conn is already an open OleDbConnection object (from our previous example), and that $query contains an INSERT, UPDATE, or DELETE query. If you were doing this with a SQL Server connection (that is, a SqlConnection object), substitute SqlCommand for OleDbCommand.

If you're trying to retrieve data you need to use the **ExecuteReader** method, which returns a SqlDataReader or OleDbDataReader object:

PS C:\> $dr = $cmd.executereader()

Once the query executes, $dr will be an OleDbDataReader that you can work with in a script or interactively. Note that the DataReader doesn't allow you to modify data; you can only *view* it. To modify the data, you need to issue a separate INSERT, UPDATE, or DELETE query as shown.

**Reading Data**

A DataReader is basically a big list of database rows, not unlike an Excel spreadsheet. A *pointer* keeps track of the current row. When you attempt to read data form a column, you'll be doing so with the current row in the DataReader. The DataReader starts with the pointer just before the first row. So you'll issue the **Read** method to move the pointer onto the first row. It will return True if there are still additional rows to be read:

PS C:\> $dr.read()

False

A few properties such as **HasRows** and **FieldCount** give you information about the DataReader:

PS C:\> $dr.fieldcount

2

PS C:\> $dr.hasrows

True

PS C:\>

Remember in the SELECT query how you specified the columns you wanted? That list is important because it determines how you'll access the data. The first column is number 0, the second is number 1, and so forth. To get the value from the first column:

$dr.getvalue(0)

You can also put all of this into a loop:

while ($dr.Read()) {

$dr.getvalue(0)

}

All of this is a bit different from working with the DataTable object we introduced earlier. However, it provides a more interactive, controllable experience and it's completely workable from the PowerShell prompt without even writing a script.

**Finishing Up**

When you're finished working with a DataReader or a connection, you should close them:

$dr.close()

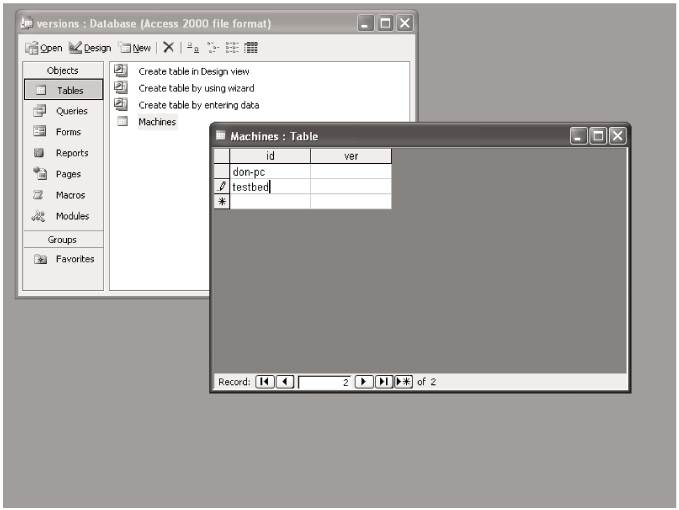
$conn.close()

If a connection is used to populate a DataReader, the connection cannot be used for anything else while the DataReader remains open. If you want to issue other queries at the same time, you need to create a second connection for that purpose.

**Database Example**

Let's take an earlier example and make it more fully databaseoriented. We'll create an Access database named Versions.mdb. In it, we'll create a table named Machines. Within the Machines table, we'll create two columns named ID and Ver. To keep things simple, each column will be of the Text type with all the default column settings. After creating the table, we'll open it and add several computer names in the ID column. All of the computers must exist, be turned on, be network-accessible, and you should be a local administrator on them. [Figure 14-1](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/section-102.xhtml#ch14fig01) shows the Access table with our initial data.

Figure 14-1: Initial Access database table



Next we'll run Database.ps1:

**Database.ps1**

#open database

$conn = new-object system.data.oledb.oledbconnection

$conn.connectionstring = "provider=microsoft.jet.oledb.4.0;" `

+ "data source=c:\test\versions.mdb"

$conn.open()

#query results

$query = "select id from machines"

$cmd = new-object system.data.oledb.oledbcommand

$cmd.commandtext = $query

$cmd.connection = $conn

$dr = $cmd.executereader()

while ($dr.read()) {

$name = $dr.getvalue(0)

$sp = get-wmiobject Win32\_OperatingSystem `

-property ServicePackMajorVersion `

-computername $name

$ver = $sp.servicepackmajorversion

#open second connection

$conn2 = new-object system.data.oledb.oledbconnection

$conn2.connectionstring = "provider=microsoft.jet.oledb.4.0;" `

+ "data source=c:\test\versions.mdb"

$conn2.open()

#update table

$query2 = "update machines set ver = '$ver' where id = '$name'"

$cmd2 = new-object system.data.oledb.oledbcommand

$cmd2.commandtext = $query2

$cmd2.connection = $conn2

$cmd2.executenonquery()

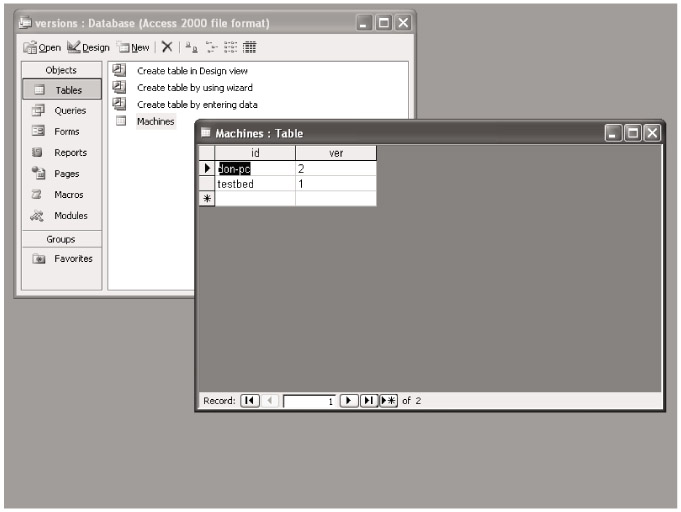
#close second connection

$conn2.close()

}

[Figure 14-2](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/section-102.xhtml#ch14fig02) shows the final populated table:

Figure 14-2: Service pack information added to table by script



## Chapter 15: Extending Types

### Overview

*PowerShell doesn't often force you to deal with specific data types. However, as you learned in*[*Chapter 5*](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/chapter-5-37.xhtml#ch05)*, PowerShell*does*support data types like strings and integers. You can extend those types to provide customized functionality.*

PowerShell's system of types is quite powerful and useful. For example, [Chapter 5](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/chapter-5-37.xhtml#ch05) introduced you to the concept of variables as objects, where a variable of the String type has methods like **Replace** and properties such as **Length.** The functionality built into these types directly from the .NET Framework that lies underneath PowerShell.

What makes PowerShell even more flexible is the fact that these types can be extended. This isn't inheritance in the true .NET Framework sense. However, it's close enough for Windows administrative purposes! For example, you can write script code that extends the String type so it has an IsEmail method. That method might use a regular expression to see if a string is formatted as an E-mail address, and return a True or False either way.

Type extensions are permanent, which means once you create them they're always available until you specifically remove them. Keep in mind that we're not making new types; we're simply adding functionality to types that already exist within PowerShell.

## The Types File

PowerShell's type extensions are defined in an XML file that is located in the PowerShell installation folder called Types.ps1xml. You can edit this file in any text editor such as Notepad. Alternatively, the file can be edited in an XML editor such as SAPIEN PrimalScript. Not every type supported by PowerShell is listed. Instead, only the types being extended need to be included in this file. Other types simply exist with their default capabilities.

You can also create your own type flies, giving them a .ps1xml filename extension. They'll use the same format as PowerShell's default file, and you load them into PowerShell using the Update-TypeData cmdlet. Using your own type files allows you to extend the type system without modifying the default file provided by Microsoft.

## Aliasing

One easy way to extend a type is to alias existing properties or methods. For example, the NET Framework's System.Array type has a property named Length that tells you how many elements an array contains. However, to make this more consistent with other PowerShell syntax, PowerShell's designers wanted to be able to refer to this property by the name Count. So, they simply aliased the property in a type extension:

<Type>

<Name>System.Array</Name>

<Members>

<AliasProperty>

<Name>Count</Name>

<ReferencedMemberName>Length</ReferencedMemberName>

</AliasProperty>

</Members>

</Type>

Here the original .NET type, System.Array, is called out. An AliasProperty section defines a new property, Count, which refers to the existing property, Length.

## Adding Features

Aliasing properties isn't really extending the functionality of a type; it just makes it more convenient or intuitive. However, you can extend a type's functionality. For example, we've added the following to our types.ps1xml file:

<Type>

<Name>System.String</Name>

<Members>

<ScriptProperty>

<Name>IsEmail</Name>

<GetScriptBlock>

$regex=[regex]"^([\w-]+)(\.[\w-]+)\*@([\w-]+\.)+[a-zA-Z]{2,7}$"

$var = $this

if ($regex.ismatch($var)) {

$true

} else {

$false

}

</GetScriptBlock>

</ScriptProperty>

</Members>

</Type>

This defines a brand-new property for the System.String type named IsEmail. In it we've performed a regular expression comparison that you may recognize from [Chapter 7](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/chapter-7-51.xhtml#ch07). The special $this variable contains whatever the instance of the type contains.

That's a bit complicated to read, so let's explain a bit. When you create a new variable and specify that it's a string, you're creating a new instance of the System.String type:

PS C:\> [string]$var = "<[don@sapien.com](mailto:don@sapien.com)>"

This new instance, named $var, contains the value "<[don@sapien.com](mailto:don@sapien.com)>", which is the value assigned to the variable. So, within the type extension, the special variable $this refers to "<[don@sapien.com](mailto:don@sapien.com)>" because that's the value inside this particular instance of the type.

The rest of our type extension code performs the regular expression match. If there's a match, the special variable $true is output for the IsEmail property; otherwise, $false is output. Here's how it works in the shell:

PS C:\> [string]$var = "<[don@sapien.com](mailto:don@sapien.com)>"

PS C:\> $var.isemail

True

PS C:\>

Pretty cool, right? We've provided another example to prove it works when the value isn't an E-mail address:

PS C:\> [string]$not = "hello!"

PS C:\> $not.isemail

False

PS C:\>

So we've added brand-new functionality to the System.String type within PowerShell just by adding a few lines of script to types.ps1xml. The downside to this approach is that your System.String type is only extended on your computer. This means if you write scripts that rely on this extension, the scripts will only run on your computer. To have this work elsewhere in your organization, you need to share your types.ps1xml file. Fortunately, that's easy to do since it's a normal text file. However, you may want to implement some procedures within your organization to control this file. For example, you might keep a centralized copy of it in a source control database such as Microsoft Visual SourceSafe so prior versions remain accessible. This will also ensure there's a single, central location from which all administrators can obtain the latest version.

## Why is This Useful?

So why should you care about extending the functionality in PowerShell's built-in types? The answer is modularization. For example, imagine you've written a function that tells if a string is a valid E-mail address, or a Universal Naming Convention (UNC) path, or something else. You could reuse that function by pasting it into all your scripts, but the code wouldn't be centralized. Any changes to your function would need to be manually replicated to every script containing the function. However, if you used that function to extend the String type to have an IsEmail method, IsUNCPath method, and so forth, then the functionality would be available to all of your scripts through a single block of code. True, you might need to manage multiple PS1XML files if the type extension was to be used on multiple computers. However, that's less of a burden than managing dozens of disparate scripts. In addition, if you're primarily writing scripts to run on your own computer, then you'd only have one file to manage.

## Chapter 16: Powershell Jump-Start for VBScripters

### Overview

*PowerShell's flexibility and high profile makes it attractive to many Windows administrators including those who've never worked with any kind of scripting before. But, for those who've worked with VBScript in the past, we'll give you a bit of a jump-start.*

Let's quickly clear up a potential point of confusion: There's no easy, set way to convert a VBScript to a PowerShell script. But why would you want to? If the VBScript works, keep it! After all, VBScript isn't going anywhere. However, in this chapter we will present a sort of "jump-start" guide to PowerShell using VBScript as a basis. That way, if you do know VBScript, you'll be able to start writing new scripts in PowerShell a bit more quickly. So, this chapter is about converting you to PowerShell, not your scripts.

Let's begin by acknowledging that PowerShell is very different from VBScript. You will need to learn new technologies and concepts to use PowerShell effectively. However, there are some similarities—especially in PowerShell's scripting language—that can be a bit easier to learn if you see them side-by-side with their VBScript counterparts. So in this chapter we'll cover the similarities between PowerShell and VBScript.

As we begin, keep in mind that PowerShell is a management shell. It isn't intended for logon scripts, so there are a lot of topics such as mapping drives and checking for group membership that we will not cover in this chapter. PowerShell's best use is not as a logon script processor. For the time being, stick with VBScript or KiXtart for those scripts.

## Variables

PowerShell variables do not need to be declared up front. That's true in VBScript, except VBScript does give you the option of doing so, while PowerShell does not. However, explicit variable declaration is always optional in PowerShell.

Variables in PowerShell, like those in VBScript, can contain any type of data. In VBScript, this is done by making all variables the Variant type. In PowerShell, variables are the more generic Object type. Unlike in VBScript, you can tell PowerShell to force a variable to be of a certain type:

[string]$var = "hello"

This creates a new variable, $var, and forces it to be a string. Notice that all variable names being with $. Apart from that, PowerShell variable naming rules are similar to the rules in VBScript.

Variable naming in VBScript typically uses Hungarian notation, where a three-letter prefix such as obj, str, or int is used to denote the type of data the variable is intended to hold. PowerShell does not require this. In fact, when working with PowerShell this isn't considered a best practice. However, you're welcome to name your variables in this fashion if you're accustomed to doing so.

## COM Objects

If you've used VBScript, KiXtart, or any similar scripting language for Windows administration, at some point you've almost certainly used a Component Object Model (COM) component. Windows is built on COM, and COM objects provide significant functionality for files, folders, WMI, and much more. Scripting without COM would be almost unthinkable.

However, PowerShell isn't built on COM; instead it's built on the .NET Framework. The Framework replaces much of the functionality you may have used COM for, but not all. As a result, there's often still a need to utilize an old COM component. Sometimes, that need might simply be that you know how to do something using a particular COM component and you don't have time to learn an alternative way in PowerShell.

Fortunately, PowerShell includes an adaptation layer that permits you to utilize COM components.

### Instantiating Objects

If you've used VBScript, you may be familiar with syntax like this:

Dim objFSO

Set objFSO = CreateObject("Scripting.FileSystemObject")

In VBScript this statement instantiates a COM component having the ProgID Scripting.FileSystemObject. When executed, VBScript asks Windows to instantiate the component. In turn, Windows looks up the ProgID in the registry to locate the actual DLL involved, loads the DLL into memory, and plugs it into the script. The variable objFSO represents the running DLL, providing an interface for working with it.

PowerShell can do nearly the same thing:

$fso = new-object -com Scripting.FileSystemObject

Using the same ProgID, PowerShell can instantiate the COM object and assign it to a variable so you can work with it. Notice the -com parameter, which is easy to forget. However, if you don't include it, PowerShell will not be able to "find" the COM object and instantiate it for you.

### Using Objects

Once instantiated, using a COM object's properties and methods is straightforward:

$file = $fso.OpenTextFile("C:\file.txt",8,True)

You can even pipe the COM object to the **Get-Member** cmdlet to see the available properties and methods of a COM object:

PS C:\> $fso | get-member

TypeName: System.\_\_ComObject#{2a0b9d10-4b87-11d3-a97a-00104b365c9

Name MemberType Definition

---- ---------- ----------

BuildPath Method string BuildPath (string, string)

CopyFile Method void CopyFile (string, string, bool)

CopyFolder Method void CopyFolder (string, string, bool

CreateFolder Method IFolder CreateFolder (string)

CreateTextFile Method ITextStream CreateTextFile (string, b

DeleteFile Method void DeleteFile (string, bool)

DeleteFolder Method void DeleteFolder (string, bool)

DriveExists Method bool DriveExists (string)

FileExists Method bool FileExists (string)

FolderExists Method bool FolderExists (string)

GetAbsolutePathName Method string GetAbsolutePathName (string)

GetBaseName Method string GetBaseName (string)

GetDrive Method IDrive GetDrive (string)

GetDriveName Method string GetDriveName (string)

GetExtensionName Method string GetExtensionName (string)

GetFile Method IFile GetFile (string)

GetFileName Method string GetFileName (string)

GetFileVersion Method string GetFileVersion (string)

GetFolder Method IFolder GetFolder (string)

GetParentFolderName Method string GetParentFolderName (string)

GetSpecialFolder Method IFolder GetSpecialFolder (SpecialFold

GetStandardStream Method ITextStream GetStandardStream (Standa

GetTempName Method string GetTempName ()

MoveFile Method void MoveFile (string, string)

MoveFolder Method void MoveFolder (string, string)

OpenTextFile Method ITextStream OpenTextFile (string, IOM

Drives Property IDriveCollection Drives () {get}

However, there's an important caveat here. PowerShell creates this list by looking at the COM object's type library that is either embedded in the DLL or included in a separate TLB file. If PowerShell can't find the type library, then it can't use the COM component. Most COM components come with type libraries, especially the COM components written by Microsoft. However, some COM components don't have a type library, or if they do, the type library isn't properly registered with Windows. In these cases, the COM component won't be usable within PowerShell.

In addition, if a type library is wrong, which happens occasionally, PowerShell may not be able to utilize the entire COM object. For example, the Microsoft-supplied type library for the WshController COM object provides an incorrect spelling for the Execute method. This makes the object difficult to use properly. However, in the case of this particular object, there's little reason to use it inside PowerShell.

### GetObject

Another way you may have used COM in VBScript was with GetObject(), which often connects to an existing object or service. In VBScript, you could do this:

Set objUser = GetObject("WinNT://don-pc/administrator,user")

This example uses the ADSI WinNT provider to retrieve the local administrator user.

GetObject() in PowerShell is a bit more difficult. Unfortunately, PowerShell doesn't have a cmdlet that does exactly this. In fact, PowerShell doesn't even provide a cmdlet for ADSI; instead, as outlined in [Chapter 12](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/chapter-12-82.xhtml#ch12), you use the [ADSI] type accelerator, which works similarly to GetObject() in VBScript. Had this GetObject() example been for WMI, we could use the **Get-Wmiobject** cmdlet instead. In many cases, WMI offers an alternative to what you were doing in ADSI, although certainly not always.

## Comments

VBScript uses a single quote (') to begin a comment, while PowerShell uses the hash (#) symbol.

**Loops and Constructs**

As illustrated in the following table, there's nearly a one-to-one correspondence between VBScript and PowerShell constructs:

| **In VBScript…** | **In PowerShell…** |
| --- | --- |
| Exit Do, Exit For | Break |
| For…Next | For |
| For Each…Next | Foreach |
| Function | Function |
| Sub | (no equivalent; use Function) |
| If…Then | If |
| If…ElseIf…Else | If, ElseIf, and Else |
| Select…Case | Switch |
| Do…Loop Until, Do Until…Loop | Do…until, Do until |
| Do…Loop While, Do While…Loop | Do…While, While |

Refer to [Chapter 8](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/chapter-8-56.xhtml#ch08) to review the discussion of these loops and constructs.

## Type Conversion

VBScript provides a number of specific functions to convert between data types including CStr(), CInt(), and CDate(). PowerShell uses a single operator, -as, to do the same thing:

$var = $var2 -as [string]

This example attempt to convert $var2 into a string and store the result in $var. Refer to [Chapter 5](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/chapter-5-37.xhtml#ch05) for more information on variables and types, and [Appendix B](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/appendix-B-251.xhtml#ap02) for a more detailed list of the major types used in PowerShell.

**Operators and Special Values**

in many cases PowerShell uses different operators than VBScript, and it uses some operators differently. The following table provides a summary of these operators:

| **In VBScript…** | **Purpose…** | **In PowerShell…** |
| --- | --- | --- |
| = | Assignment | = |
| = | Equality test | -eq |
| > | Greater than | -gt |
| < | Less than | -lt |
| >= | Greater than or equal to | -ge |
| <= | Less than or equal to | -le |
| True | Boolean True | $true |
| False | Boolean False | $false |
| AND | Boolean AND | -and |
| NOT | Boolean NOT | -not |
| OR | Boolean OR | -or |
| AND | Binary AND | -band |
| OR | Binary OR | -bor |
| NOT | Binary NOT | -bnot |
| &, + | String concatenation | + |

Refer to [Chapter 6](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/chapter-6-45.xhtml#ch06) to review the discussion of the additional operators offered in PowerShell.

## Functions and Subs

VBScript and PowerShell declare functions similarly. Here's a function in VBScript that returns True if the input parameter is more than 5, otherwise it returns False:

Function IsMoreThan5(intValue)

If intValue > 5 Then

IsMoreThan5 = True

Else

IsMoreThan5 = False

End if

End Function

Notice that VBScript returns a value by setting the function name equal to the return value. PowerShell works similarly:

Function IsMoreThan5($Value) {

If ($Value -gt 5) {

Return $true

} Else {

Return $false

}

}

Notice that the **Return** keyword is used to return the function's value. In fact, any output of the function will be appended to the return value. The following is functionally identical:

Function IsMoreThan5($Value) {

If ($Value -gt 5) {

$true

} Else {

$false

}

}

This example shows that outputting $true or $false into the pipeline makes those values the function's return value. Refer to [Chapter 10](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/chapter-10-72.xhtml#ch10) to review the discussion of how functions have significantly expanded capabilities in PowerShell.

PowerShell does not provide a separate Sub construct as VBScript does. However, a Function that returns no value is essentially the same as a Sub.

## Error Handling

VBScript's **On Error Resume Next** statement, and the corresponding **On Error Goto 0** statement, are used to implement error handling. Essentially, you execute **On Error Resume Next** before any operation that may result in an error, and then check the special **Err** object to see if an error did indeed occur. VBScript's error handling is actually quite primitive, while PowerShell's is much more advanced.

In brief, you declare a trap, which is what PowerShell executes when an error (or exception) occurs (or is thrown). You do whatever you need to do within the trap, and then tell PowerShell to either **continue**, which resumes execution on the line following whatever line caused the exception, or **break**, which halts execution. For example:

Trap {

# handle error here

Continue

}

Refer to [Chapter 11](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/chapter-11-78.xhtml#ch11) to review the discussion of how to define different trap blocks for different types of exceptions.

## Windows Management Instrumentation

This is an easy point of conversion. Anytime you used GetObject() or some other means of retrieving WMI information, just use the **Get-Wmiobject** cmdlet or its convenient alias, **Gwmi**, in PowerShell. However, notice that just running it with a class name won't return every property of the class by default:

PS C:\> gwmi win32\_operatingsystem

SystemDirectory : C:\WINDOWS\system32

Organization : SAPIEN Technologies, Inc.

BuildNumber : 2600

RegisteredUser : Don Jones

SerialNumber : 76487-338-1820253-22242

Version : 5.1.2600

Notice that this doesn't return every property for the class. Instead, the properties shown are defined by a special view within PowerShell. While you can update that view to list more properties, you can also use the -property parameter if there's a specific property you need:

PS C:\> gwmi win32\_operatingsystem -property buildnumber

BuildNumber : 2600

\_\_GENUS : 2

\_\_CLASS : Win32\_OperatingSystem

\_\_SUPERCLASS :

\_\_DYNASTY :

\_\_RELPATH :

\_\_PROPERTY\_COUNT : 1

\_\_DERIVATION : {}

\_\_SERVER :

\_\_NAMESPACE :

\_\_PATH :

If necessary, you can assign that to a variable. However, notice that you get back a collection of multiple instances that can be referred to individually by number as shown here:

PS C:\> $obj = gwmi win32\_logicaldisk

PS C:\> $obj[0]

DeviceID : A:

DriveType : 2

ProviderName :

FreeSpace :

Size :

VolumeName :

PS C:\> $obj[1]

DeviceID : C:

DriveType : 3

ProviderName :

FreeSpace : 91841773568

Size : 153006624768

VolumeName :

If you just need a specific property from an instance:

PS C:\> $obj[1].DriveType

3

Run **Help Gwmi** to get a comprehensive list of parameters, or refer to [Appendix A](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/appendix-A-121.xhtml#ap01) of this book.

## Active Directory Services Interface

It's unfortunate that PowerShell doesn't come with a built-in "Get-ADSIObject" cmdlet. Undoubtedly a similar cmdlet will be available in the future, but it is not yet included in PowerShell v1.0. You can use WMI to perform some ADSI queries. You can also use the underlying .NET Framework directory services classes to manipulate AD. However, using those classes moves beyond the realm of scripting and pretty firmly into the world of .NET Framework programming.

Review [Chapter 12](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/chapter-12-82.xhtml#ch12) for some examples of what you can do in PowerShell regarding ADSI. For now, continue using VBScript for more complex ADSI-related tasks.

**Common Tasks in VBScript**

You are probably familiar with the VBScript items listed below. Therefore, we'll quickly point out how to do nearly the same thing in PowerShell:

* **WScript.Echo.**

For producing output to the command-line, use the

**Write-Host** cmdlet.

* **MsgBox(), InputBox().**

No direct analog in PowerShell since it's intended to be entirely command-line. Use the

**Read-Host** cmdlet to accept input from the command-line.

* **WMI.**

Use

**Get-WMIObject**.

* **Working with text files.**

Refer to

[Chapter 12](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/chapter-12-82.xhtml#ch12). Several PowerShell cmdlets are available to manipulate text files. The basic technique is to read the entire file into an object and then enumerate through each line of the file as a child object.

* **ADSI.**

Tricky because there's not a direct equivalent in PowerShell v1. However, refer to the discussion earlier in this chapter for information on how to work with ADSI.

* **Working with the registry.**

Review

[Chapter 12](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/chapter-12-82.xhtml#ch12) for more information. PowerShell provides a registry "drive" and cmdlets for working with the registry.

**PowerShell Paradigm Change**

Probably the biggest mental leap you can make when moving from VBScript to PowerShell is that PowerShell is intended to deal with *objects* not text. That's all well and good in theory, but it can be a confusing concept to implement until you get used to it. For example, look at ServicePack.vbs, which is a VBScript that reads a list of names from C:\Computers.txt (one name per line) and uses WMI to display the service pack version for each.

**ServicePack.vbs**

Dim strFile

strFile = "C:\computers.txt"

Dim objFSO, objTS, strComputer

Set objFSO = CreateObject("Scripting.FileSystemObject")

If objFSO.FileExists(strFile) Then

Set objTS = objFSO.OpenTextFile(strFile)

Do Until objTS.AtEndOfStream

strComputer = objTS.ReadLine

Dim objWMI

Set objWMI = GetObject("winmgmts:\\" & strComputer & \_

"\root\cimv2")

Dim colResults, objResult, strWMIQuery

strWMIQuery = "SELECT \* FROM Win32\_OperatingSystem"

Set colResults = objWMI.ExecQuery(strWMIQuery)

For Each objResult In colResults

WScript.Echo strComputer & ":" & \_

objResult.ServicePackMajorVersion

Next

Loop

End If

objTS.Close

WScript.Echo "Complete"

As mentioned previously, you're probably familiar enough with VBScript to follow what this script is doing. However, take time to notice the methodology. Each time through the **Do** loop, a line is read from the text file, which is assumed to be a computer name. A WMI connection to that computer is created, and the Win32\_OperatingSystem class retrieved. For each instance of the class, a loop displays the ServicePackMajorVersion property.

Now look at ServicePack.ps1, which does the same thing.

**ServicePack.ps1**

$names = get-content "c:\computers.txt"

foreach ($name in $names) {

$wmi = get-wmiobject win32\_operatingsystem `

-property servicepackmajorversion `

-computer $name

$sp = $wmi.servicepackmajorversion

write-host "$name : $sp"

}

First of all, notice how much shorter this is! It actually could be shorter, but we wrote it more for clarity than brevity. Again, the *real* thing to notice is the methodology. The **Get-Content** cmdlet is used to retrieve the entire contents of the text file into $names, all in one step. A **foreach** loop goes through each child item (or line) of the text file, pulling each child item into $name. The **Get-Wmiobject** cmdlet retrieves just the desired property, placing the class instance into $wmi. The $sp variable is used to hold the actual property in which we're interested. The **Write-Host** is used to output the information to the screen.

Keep in mind that, where VBScript typically requires something to be done in a particular way, PowerShell is a lot more flexible. ServicePack2.ps1 demonstrates the same script, from a different approach.

**ServicePack2.ps1**

filter getversion {

$wmi = get-wmiobject win32\_operatingsystem `

-property servicepackmajorversion `

-computer $\_

$sp = $wmi.servicepackmajorversion

write-host "$\_ : $sp"

}

get-content "c:\computers.txt" | getversion

Here, the content of the text file is piped to "getversion," which is a custom filter. This filter is called once for each object or line of text in the text file. The special $\_ variable refers to the current object, which would be a single computer name. ServicePack3.ps1 simplifies this even further.

**ServicePack3.ps1**

filter getversion {

$wmi = get-wmiobject win32\_operatingsystem `

-property servicepackmajorversion `

-computer $\_

write-host "$\_ : " $wmi.servicepackmajorversion

}

get-content "c:\computers.txt" | getversion

All that's been done here is to remove the $sp variable. This was done because $wmi.servicepackmajorversion is now output directly, which removes a line of code. This could be simplified even further. However, the VBScript example we started with has been simplified about as much as possible.

Getting used to PowerShell's way of working with objects take some time. However, if you make the effort, these examples have demonstrated how much more quickly you can produce usable administrative scripts.

**Appendix A: Cmdlet Reference**

**Overview**

In [Appendix A](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/appendix-A-121.xhtml#ap01) we cover the cmdlets that are included with the base PowerShell shell. This doesn't include any cmdlets that are added by a custom shell such as the version of PowerShell included with Microsoft Exchange Server.

For each cmdlet we'll list the cmdlet name and all of its parameters. Certain parameters are referred to as *ubiquitous*, meaning they are shared by many cmdlets. We'll cover ubiquitous parameters first and you can refer back to them as necessary. In the syntax for each cmdlet, the ubiquitous parameters will be *italicized* so you can identify them more easily.

For each parameter, we'll give you the parameter name and its type:

[ -Name <String> ]

In this context, the outermost square brackets indicate that the parameter and its value are optional. Because the parameter and value appear in the brackets together, they must be used together. In other words, if you want to specify the value, then you must also specify the parameter name. We list the value type in angle brackets, while the primary possible value types are listed in [Table A-1](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/appendix-A-121.xhtml#ap01table01).

| **Table A-1: Parameter Value Types** | |
| --- | --- |
| **Value Type** | **Description** |
| <ActionPreference> |  |
| <Boolean> | $true or $false |
| <Int32> | Integer number. |
| <Object> | Any type is accepted. Usually a string or numeric value of some kind. |
| <PSCredential> | Security credential. May be created with the **Get-Credential**Cmdlet. |
| <String> | Series of characters enclosed in quotation marks. |
| <SwitchParameter> | Indicates that the parameter is just a switch that does not require a value. It's either present or not, to specify some option. |
| <enum> | Indicates that the parameter takes one of several predefined values. Generally, the cmdlet's help will specify the acceptable values. |

You may see some value types listed like this:

[ -Include <String[]> ]

The square brackets next to String indicate that the parameter can either accept a single value of the String type or an array of the String type.

You may also see parameter types listed as <System.String> instead of <String>. This indicates that PowerShell is listing the full .NET Framework type. Usually you'll just use the last portion of the type ("[String](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/section-256.xhtml#ap02lev1sec6)," in this case) to refer to the table above.

## Add-Content

Adds to the content(s) of the specified item(s).

Snap-In: Microsoft.PowerShell.Management

Syntax:

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>]

Description:

The Add-Content Cmdlet takes direct input or input from the pipeline and adds it to the specified item or file. The content is appended to the existing content of the file.

## Add-History

Adds entries to the session history.

Snap-In: Microsoft.PowerShell.Core

Syntax:

Add-History [

[-InputObject] <PSObject[]>]  
[-Passthru]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The add-history Cmdlet enables you to add entries to the session history list. Using the cmdlet to add a command to the list has the same effect as actually typing the command. The add-history Cmdlet makes it possible to reconstruct a history list saved during a previous Ps session. You can save the session history in XML or CSV format. By default, history files are saved in the home directory, but specifying a path allows you to save the file in any location.

## Add-Member

Adds a user-defined custom member to an object.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Add-Member

[-MemberType] <PSMemberTypes>  
[-Name] <String> [  
[-Value] <Object>] [  
[-SecondValue] <Object>] -InputObject <PSObject>  
[-Force]  
[-PassThru]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

Adds a user-defined custom member to an object. The Add-Member Cmdlet takes an Input object and adds to it one of the following user-defined member types: AliasProperty, CodeProperty, Property, NoteProperty, ScriptProperty, PropertySet, Method, CodeMethod, MemberSet, ScriptMethod Add-Member takes any arbitrary InputObject and outputs the noted object with the added member types. If the InputObjetc has N types, the output, OutputObject, will have N+1 types.

## Add-PSSnapin

Adds one or more PSSnapIn(s) to the current Ps console.

Snap-In: Microsoft.PowerShell.Core

Syntax:

Add-PSSnapin

[-Name] <String[]>  
[-PassThru]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The add-PSSnapIn Cmdlet adds one or more PSSnapIns to the current Ps console. After adding a PSSnapIn to the current console, the associated cmdlets can be run within the current console. Be aware that only the in-memory console information is updated. The corresponding console file is not updated. Use the export-console Cmdlet to reuse the console following the current session. The Add-PSSnapIn Cmdlet runs a version check test on the PSSnapIns before loading them in the current console. An error message is displayed if a PSSnapIn cannot be added. Loaded PSSnapIns can be saved in a console file by using the export-console Cmdlet. The current console file name is available in the read-only $console environment variable.

## Clear-Content

The clear-content Cmdlet removes the content from an item or file while leaving the file intact.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Clear-Content

[-Path] <String[]>  
[-Filter <String>]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Force]  
[-Credential <PSCredential>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

The Clear-Content Cmdlet clears the content from an item or a file. This is equivalent to setting a file size to zero. However, the file itself remains intact.

## Clear-Item

Sets the item(s) at the specified location to the "clear" value specified by the provider.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Clear-Item

[-Path] <String[]>  
[-Force]  
[-Filter <String>]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Credential <PSCredential>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

Clear is not the same as remove. In the case of clear, the object that is cleared is still in place. For example, if a variable is cleared then a null variable of that name still exists. Be aware that the value used to indicate a cleared item is provider-specific.

## Clear-ItemProperty

Removes the property value from a property.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Clear-ItemProperty

[-Path] <String[]>  
[-Name] <String>  
[-PassThru]  
[-Force]  
[-Filter <String>]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Credential <PSCredential>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

The Clear-ItemProperty Cmdlet removes the value from a property. However, unlike the Remove-Property Cmdlet, it leaves the property in place.

## Clear-Variable

Removes the value from a variable.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Clear-Variable

[-Name] <String[]>  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Force]  
[-PassThru]  
[-Scope <String>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

Clears the data stored in an existing variable. If the variable is typed, then clear-variable preserves the type of the object stored in the variable. Clear-variable sets the value to NULL, but leaves the variable intact.

## Compare-Object

Compares the properties of objects.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Compare-Object

[-ReferenceObject] <PSObject[]>  
[-DifferenceObject] <PSObject[]>  
[-SyncWindow <Int32>]  
[-Property <Object[]>]  
[-ExcludeDifferent]  
[-IncludeEqual]  
[-PassThru]  
[-Culture <String>]  
[-CaseSensitive]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The Compare-Object Cmdlet compares two object streams and returns the result of the comparison. In the return of the comparison, there are pointers to where e ach object or property came from. They are:

* "<=" means the data in this object is from the reference object
* "=>" means the data in this object is from the compared object
* "==" means the data came from both

## ConvertFrom-SecureString

Export a securestring to a safe, persistent format

Snap-In: Microsoft.PowerShell.Security

Syntax:

ConvertFrom-SecureString

[-SecureString] <SecureString> [  
[-SecureKey] <SecureString>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

ConvertFrom-SecureString

[-SecureString] <SecureString>  
[-Key <Byte[]>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

ConvertFrom-SecureString takes a secure string and converts it into a DPAPI-protected normal string. This string can be securely embedded in a script and reconstituted to a secure string with the ConvertTo-SecureString Cmdlet. If a key is provided, the encryption should not use the DPAPI encryption. Instead, another FIPS-140 compliant algorithm should be used. The encryption key should be 16, 24, or 32 bytes long. The Rijndael algorithm used supports keys with lengths of 128, 192, and 256 bits.

## Convert-Path

Converts the path of the item given from an Ps path to a provider path.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Convert-Path

[-Path] <String[]>  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

Not all providers may recognize standard pathname delimiters such as "\". With the Convert-Path Cmdlet a path is converted to a pathname object that is universally recognized by all providers.

## ConvertTo-Html

Converts the input to an HTML table.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

ConvertTo-Html [

[-Property] <Object[]>]  
[-InputObject <PSObject>]  
[-Body <String[]>]  
[-Head <String[]>]  
[-Title <String>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The convert-html Cmdlet creates an HTML page by taking the data from incoming objects and converting it to an HTML table. By default, all the properties of t he object are placed in an HTML table row. If the properties are not available an empty table cell is created.

## ConvertTo-SecureString

Creates a securestring from a normal string created by export-securestring

Snap-In: Microsoft.PowerShell.Security

Syntax:

ConvertTo-SecureString

[-String] <String> [  
[-SecureKey] <SecureString>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

ConvertTo-SecureString

[-String] <String>  
[-Key <Byte[]>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

ConvertTo-SecureString takes a normal string and converts it to a securestring using the DPAPIs. If a key is provided, the decryption will not use the DPAPI encryption, but another FIPS-140-compliant algorithm. The encryption key should be 16, 24, or 32 bytes long. The Rijndael algorithm used supports keys with lengths of 128, 192, and 256 bits.

## Copy-Item

Calls a provider to copy an item from one location to another within a namespace.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Copy-Item

[-Path] <String[]> [  
[-Destination] <String>]  
[-Container]  
[-Force]  
[-Filter <String>]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Recurse]  
[-PassThru]  
[-Credential <PSCredential>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

Copy-item Cmdlet copies an item from one location to another in a namespace. It is roughly equivalent to the 'cp' or 'copy' commands from other shells. Note that a directory specified as path does not automatically expand to the files in that directory. Use the syntax directory/\* to indicate all the files. If the expansion of the directory includes a subdirectory, and - Recurse or -Container is not t rue, then copy-item will copy the files over, but not the subdirectories. Next, produce the following warning:

*The source and destination paths are both containers, yet recurse was not specified. In order to copy a container into another container, the container parameter must be true.*

## Copy-ItemProperty

Copies a property between locations or namespaces.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Copy-ItemProperty

[-Path] <String[]>  
[-Destination] <String>  
[-Name] <String>  
[-PassThru]  
[-Force]  
[-Filter <String>]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Credential <PSCredential>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

The copy property Cmdlet takes a property and value from a specified location and copies it to another location.

## Export-Alias

Exports an alias list to a file.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Export-Alias

[-Path] <String> [  
[-Name] <String[]>]  
[-PassThru]  
[-As <ExportAliasFormat>]  
[-Append]  
[-Force]  
[-NoClobber]  
[-Description <String>]  
[-Scope <String>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

The export-alias Cmdlet writes the current alias list to a file. This file can be named in the command. If it does not exist, Ps creates it at the specified location. You can create the file in a convenient format such as text (.txt), and Microsoft Word (.doc).

## Export-Clixml

Produces a clixml representation of a Ps object or objects.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Export-Clixml

[-Path] <String>  
[-Depth <Int32>] -InputObject <PSObject>  
[-Force]  
[-NoClobber]  
[-Encoding <String>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

The export-clixml Cmdlet takes input from the pipeline or from a designated object and produces a Command Line Interface XML (clixml) file. That file can be sent down the pipeline or to a destination. The tagged XML file can then be used with any number of XML tools such as XPath. Objects are serialized, which means they are saved as they are with the tagging data intact. The objects are then free to be sent across networks and computers.

## Export-Console

Export the changes made to the current console. This action overwrites any existing console file.

Snap-In: Microsoft.PowerShell.Core

Syntax:

Export-Console [

[-Path] <String>]  
[-Force]  
[-NoClobber]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

The export-console Cmdlet is used to save a Ps console for later use. You can save the console to the current console file or a new console file. If you specify the -name parameter and a console file with that name already exists, you will be prompted before the file is overwritten. If you save the current console to a new console file, the $console environment variable is updated with the new file name. If you specify the -force parameter, you will receive no confirmation prompts even if an existing console file will be overwritten.

## Export-Csv

Forms CSV strings from input.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Export-Csv

[-Path] <String> -InputObject <PSObject>  
[-Force]  
[-NoClobber]  
[-Encoding <String>]  
[-NoTypeInformation]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

The export-csv Cmdlet takes various types of input and uses markers such as line breaks, commas, double quotes, and spaces to create a list of comma-separated values (CSV). Properties common to an object are grouped in the same CSV string. A missing property will show as a space between commas. Double quotes in a string are redoubled to show they are literal.

## ForEach-Object

Applies script blocks to each object in the pipeline.

Snap-In: Microsoft.PowerShell.Core

Syntax:

ForEach-Object

[-Process] <ScriptBlock[]>  
[-InputObject <PSObject>]  
[-Begin <ScriptBlock>]  
[-End <ScriptBlock>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

This ForEach-Object Cmdlet allows you to process the set of objects passing through the pipeline by applying a script block argument to each item. The result of the evaluation of the script block is passed down the pipe. The script block can contain any arbitrary Ps script elements. It is executed in the current context and the value of the current pipeline object is available in the $\_ variable. You can also specify -begin and -end actions. The -begin parameter specifies that the script block is to be executed before the first object is processed. The -end parameter specifies that the script block is to be executed after the last object has been processed. Any objects produced by either -begin or -end a re also added to the output.

## Format-Custom

Formats output display as defined in additions to the formatter file.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Format-Custom [

[-Property] <Object[]>]  
[-Depth <Int32>]  
[-GroupBy <Object>]  
[-View <String>]  
[-ShowError]  
[-DisplayError]  
[-Force]  
[-Expand <String>]  
[-InputObject <PSObject>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

There are several display formats defined in the Microsoft Shell's formatter. These are contained in Help.Format.Psxml, which is an XML file in the Shell's directory. If you define a custom view and add it to the XML file, you can use it with the format-custom Cmdlet.

## Format-List

Formats objects as a list with their properties displayed vertically.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Format-List [

[-Property] <Object[]>]  
[-GroupBy <Object>]  
[-View <String>]  
[-ShowError]  
[-DisplayError]  
[-Force]  
[-Expand <String>]  
[-InputObject <PSObject>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The format-list Cmdlet takes input from the pipeline and outputs a list of all specified properties of each piped object. You can specify which properties you wish displayed with the -Property parameter. When used with the out Cmdlets, format-list wraps lines instead of truncating them.

## Format-Table

Formats the output as a table.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Format-Table [

[-Property] <Object[]>]  
[-AutoSize]  
[-HideTableHeaders]  
[-Wrap]  
[-GroupBy <Object>]  
[-View <String>]  
[-ShowError]  
[-DisplayError]  
[-Force]  
[-Expand <String>]  
[-InputObject <PSObject>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

By default, input objects often format as a table. The format-table Cmdlet displays results as a table if not formatted that way.

## Format-Wide

Formats objects as a table of their properties.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Format-Wide [

[-Property] <Object>]  
[-AutoSize]  
[-Column <Int32>]  
[-GroupBy <Object>]  
[-View <String>]  
[-ShowError]  
[-DisplayError]  
[-Force]  
[-Expand <String>]  
[-InputObject <PSObject>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

Input objects often attempt to fit as many columns of values as possible on a line of output. The format-wide Cmdlet gives you an extra measure of control over the display. If the optional -Property parameter is given, it determines which property is displayed, and can also determine field format. If no -Property argument is supplied, the Default Parameter for the first object's type is used.

## Get-Acl

Gets the access control list (ACL) associated with a file or object.

Snap-In: Microsoft.PowerShell.Security

Syntax:

Get-Acl [

[-Path] <String[]>]  
[-Audit]  
[-Filter <String>]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The resources in the operating system, files, folders, shares, printers, and so on are guarded by security descriptors that are made up of Access Control Lists (ACL's) ACLs contain security information regarding access, permissions, etc. The get-acl Cmdlet retrieves the information contained in the ACL. It is displayed in list format by default.

## Get-Alias

Returns alias names for cmdlets.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Get-Alias [

[-Name] <String[]>]  
[-Exclude <String[]>]  
[-Scope <String>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The get-alias Cmdlet outputs the alias name and its corresponding cmdlet. If no alias name is entered, the command returns all the aliases from the alias variable table. The -Exclude parameter allows the user to exclude aliases from the returned list. Both -Name and -Exclude parameters take wild card inputs or a comma separated list.

## Get-AuthenticodeSignature

Gets the signature object associated with a file.

Snap-In: Microsoft.PowerShell.Security

Syntax:

Get-AuthenticodeSignature

[-FilePath] <String[]>  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The get-authenticodesignature Cmdlet retrieves the signature object that is associated with a file. If there is no signature, the path is returned and the other information is blank.

## Get-ChildItem

Retrieves the child items of the specified location(s) in a drive.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Get-ChildItem [  
[-Path] <String[]>] [  
[-Filter] <String>]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Recurse]  
[-Force]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Get-ChildItem [

[-Path] <String[]>] [  
[-Filter] <String>]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Recurse]  
[-Force]  
[-Name]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

Gets the child item(s) of the specified path using the appropriate cmdlet provider. Items are represented as a CLR type that defines the location at a particular path. If -Names is specified only the names of the item(s) are returned.

## Get-Command

Retrieves basic information about a command.

Snap-In: Microsoft.PowerShell.Core

Syntax:

Get-Command [

[-ArgumentList] <Object[]>]  
[-Verb <String[]>]  
[-Noun <String[]>]  
[-PSSnapin <String[]>]  
[-TotalCount <Int32>]  
[-Usage]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Get-Command [

[-Name] <String[]>] [  
[-ArgumentList] <Object[]>]  
[-CommandType <CommandTypes>]  
[-TotalCount <Int32>]  
[-Usage]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The get-command Cmdlet returns the command type, command name, and command syntax of a given cmdlet. Conversely, the get-help [Cmdlet noun or verb] will return cmdlet names when there is more than one instance of either a noun or verbt. If t here is only one cmdlet for a given noun or verb, then the help file is returned .

## Get-Content

The get-content command gets the content from the item at the specified location.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Get-Content

[-Path] <String[]>  
[-ReadCount <Int64>]  
[-TotalCount <Int64>]  
[-Filter <String>]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Force]  
[-Credential <PSCredential>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-Delimiter <String>]  
[-Wait]  
[-Encoding <FileSystemCmdletProviderEncoding>]

Description:

The get-content Cmdlet reads content from the item at the specified path and writes it to the pipeline. How that data is represented depends on the provider and on the -ReadCount parameter. A provider decides how content will be broken into elements and the -ReadCount parameter determines how many of those elements are written to the pipeline as an array.

## Get-Credential

Gets a credential object based on a password.

Snap-In: Microsoft.PowerShell.Security

Syntax:

Get-Credential

[-Credential] <PSCredential>  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The get-credential Cmdlet establishes a credential object called a Ps credential by pairing a given username with a prompted password. That credential object can then be used for other operations involving security.

## Get-Culture

Gets the culture information.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Get-Culture

[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The get-culture Cmdlet gets the culture information from the resource manager and creates an object. This object stores information such as the current language and keyboard layout, and makes that object available to the pipeline.

## Get-Date

Gets current date and time.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Get-Date [

[-Date] <DateTime>]  
[-Year <Int32>]  
[-Month <Int32>]  
[-Day <Int32>]  
[-Hour <Int32>]  
[-Minute <Int32>]  
[-Second <Int32>]  
[-DisplayHint <DisplayHintType>]  
[-Format <String>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Get-Date [

[-Date] <DateTime>]  
[-Year <Int32>]  
[-Month <Int32>]  
[-Day <Int32>]  
[-Hour <Int32>]  
[-Minute <Int32>]  
[-Second <Int32>]  
[-DisplayHint <DisplayHintType>]  
[-UFormat <String>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The get-date Cmdlet retrieves the date and time information from the system and returns it as a DateTime object.

## Get-PSSnapin

Lists registered PSSnapIns.

Snap-In: Microsoft.PowerShell.Core

Syntax:

Get-PSSnapin [

[-Name] <String[]>]  
[-Registered]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The get-PSSnapIn Cmdlet lists all the PSSnapIn available in the current console if no parameter is passed to it. If the -registered parameter is passed, it lists all the available PSSnapIn(s) in the registry. The order in which the PSSnapIn(s) are shown is same as the order they are listed in the console file. Users can use sort-object to sort the list -Name and -registered are not mutually exclusive. c:\PS c:\> Get-PSSnapIn Returns all PSSnapIns loaded and in the console file. c:\PS c:\> Get-PSSnapIn -Name Xyz Returns the Xyz PSSnapIn that is either loaded or in the current console. If the Xyz PSSnapIn is not loaded or in the current console, 'Object not found.' is returned. c:\PS c:\> Get-PSSnapIn -registered Returns all registered PSSnapIns. c:\PS c:\> Get-PSSnapIn -Name Xyz2 -registered Returns Xyz2 PSSnapIn info object and only looks in the registries.

## Get-TraceSource

Lists properties for given trace sources.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Get-TraceSource [

[-Name] <String[]>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The get-tracesource Cmdlet lists the trace sources and their properties visible at the time it is invoked. Trace sources are only visible for modules that are currently loaded. Trace sources are loaded on demand as the component is used .

## Get-UICulture

Gets the uiculture information.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Get-UICulture

[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The get-uiculture Cmdlet gets information about the current uiculture from the resource manager. This object stores information such as the current language and keyboard layout, and makes that object available to the pipeline. The cmdlet creates an object from the UI culture information.

## Get-Unique

Gets the unique items in a sorted list.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Get-Unique

[-InputObject <PSObject>]  
[-AsString]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Get-Unique

[-InputObject <PSObject>]  
[-OnType]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The get-unique Cmdlet examines the items in a collection and compares each item to the next item in the collection. It then separates out one unique copy of e ach sequential item and sends the list to the pipeline. To get a unique list of items, sort the objects before passing them to the get-unique Cmdlet.

## Get-Variable

Gets a Ps variable.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Get-Variable [

[-Name] <String[]>]  
[-ValueOnly]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Scope <String>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The get-variable Cmdlet gets a variable and its value, or its value only, and then allows it to be passed down the pipeline.

## Group-Object

Groups the objects that contain the same value for a common property.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Group-Object [

[-Property] <Object[]>]  
[-NoElement]  
[-InputObject <PSObject>]  
[-Culture <String>]  
[-CaseSensitive]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The group-object Cmdlet is used to group input objects based on the values of properties specified with the -Property parameter. The groups contain the count of objects in the group, as well as the members of that group.

## Import-Alias

Imports an alias list from a file.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Import-Alias

[-Path] <String>  
[-Scope <String>]  
[-PassThru]  
[-Force]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

The export-alias Cmdlet can send the command shell session's table of alias/Cmdlet pairings to a file where it is stored as a list of comma separated values (CSV). The import-alias Cmdlet brings that list back into the command shell as an object.

## Import-Clixml

Imports a clixml file and rebuilds the Ps object‥

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Import-Clixml

[-Path] <String[]>  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The export-clixml Cmdlet serializes or saves objects as XML files. The import -clixml Cmdlet takes serialized cliXML files and restructures them as PSObjects.

## Import-Csv

Takes values from a CSV list and sends objects down the pipeline.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Import-Csv

[-Path] <String[]>  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The import-CSV Cmdlet takes the CSV data listed in files created by the export-CSV Cmdlet, and sends them down the pipeline as objects.

## Invoke-Expression

Executes a string as an expression.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Invoke-Expression

[-Command] <String>  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The Invoke-Expression Cmdlet executes a string as an expression.

## Invoke-History

Invokes a previously executed command.

Snap-In: Microsoft.PowerShell.Core

Syntax:

Invoke-History [

[-Id] <String>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

Accepts a number value that indicates the identifier of a previously executed command. That command is then run again. This ID number can be found by running the get-history Cmdlet. The cmdlet history lasts only as long as the individual session.

## Invoke-Item

Invokes an executable or opens a file.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Invoke-Item

[-Path] <String[]>  
[-Filter <String>]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Credential <PSCredential>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

The invoke-item Cmdlet attempts to execute or open an item from a namespace. The behavior when an item is invoked depends on the provider. For filesystem items, this means calling ShellExecute with the path of the item. ShellExecute either runs the file or opens it with the application associated with that file type.

## Join-Path

Combines path elements into a single path.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Join-Path

[-Path] <String[]>  
[-ChildPath] <String>  
[-Resolve]  
[-Credential <PSCredential>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

Not all providers recognize standard pathname delimiters such as "\". With the combine-path Cmdlet, multiple paths can be converted to a pathname object that is universally recognized by all providers.

## Measure-Command

Tracks the running time for script blocks and cmdlets.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Measure-Command

[-Expression] <ScriptBlock>  
[-InputObject <PSObject>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The time-expression Cmdlet takes a script block or cmdlet and runs it internally. It times the execution of the operation and returns the results. It does not actually return anything from the script or cmdlet.

## Measure-Object

Measures various aspects of objects or their properties.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Measure-Object [

[-Property] <String[]>]  
[-InputObject <PSObject>]  
[-Sum]  
[-Average]  
[-Maximum]  
[-Minimum]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Measure-Object [

[-Property] <String[]>]  
[-InputObject <PSObject>]  
[-Line]  
[-Word]  
[-Character]  
[-IgnoreWhiteSpace]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

There are four basic modes that are used with the measure-object Cmdlet.

1. Default mode. No input parameters are specified. Performs a count of the number input objects. For example, get-childitem c:/smp | measure-object. This line returns the count of objects received from the get-childitem cmdlet. Be aware that there will be one object for each file and one object for each folder.
2. Specify the ?Property parameter with one or more of the specified sum or average parameters. In this case, the action determined by the sum and average parameters. For example, get-childitem c:/smp | measure-object ? Property Length - Average This sample line finds the average size of files in the folder C:/smp. The count contains the number of total number of objects.
3. Group by property(s) before performing the measure operations. For example, get-childitem c:/smp | measure-object ?Groupby Length. This sample command line counts the number of like sized groups. For example, all the files with equal lengths will be one group. In this example the count indicates how many groups there are.
4. Count text. For example, Get-content smp.txt | measure-object ?Words. This line counts the words in the file smp.txt using the default culture setting. Be aware that you can also count the text within properties. For example: Get-childitem | measure-object ?Words ` ?Property CreationTime.DayOfWeek.

## Move-ItemProperty

Moves a property from one location to another.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Move-ItemProperty

[-Path] <String[]>  
[-Destination] <String>  
[-Name] <String[]>  
[-PassThru]  
[-Force]  
[-Filter <String>]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Credential <PSCredential>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

As opposed to copy-property that makes a copy and places it into a different location, Move-ItemProperty Cmdlet removes the property from the old location and places it in the new one.

## New-Alias

Creates a new cmdlet-alias pairing.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

New-Alias

[-Name] <String>  
[-Value] <String>  
[-Description <String>]  
[-Option <ScopedItemOptions>]  
[-PassThru]  
[-Scope <String>]  
[-Force]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

The new-alias Cmdlet makes a new cmdlet-alias pairing. This alias name is added to the list for the session. If you exit the session or close the command shell, the alias name is deleted.

## New-ItemProperty

Sets a new property of an item at a location.

Snap-In: Microsoft.PowerShell.Management

Syntax:

New-ItemProperty

[-Path] <String[]>  
[-Name] <String>  
[-PropertyType <String>]  
[-Value <Object>]  
[-Force]  
[-Filter <String>]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Credential <PSCredential>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

The new-property Cmdlet

## New-Object

Creates a new .Net object.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

New-Object

[-TypeName] <String> [  
[-ArgumentList] <Object[]>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

New-Object

[-ComObject] <String>  
[-Strict]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

Parameters: -TypeName type-name [string] [mandatory] [positon 0] Full name of the type to create. -Arguments arguments [object []] [position 1] Arguments to the type's constructor. -AssemblyName assembly-name [string] -qualified name of an assembly. -FileName file-name [string] Path of an assembly file. ---

## New-PSDrive

Installs a new drive on the computer.

Snap-In: Microsoft.PowerShell.Management

Syntax:

New-PSDrive

[-Name] <String>  
[-PSProvider] <String>  
[-Root] <String>  
[-Description <String>]  
[-Scope <String>]  
[-Credential <PSCredential>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

The new-drive Cmdlet installs a new drive at the specified location and within the specified parameters.

## New-Service

Creates a new service.

Snap-In: Microsoft.PowerShell.Management

Syntax:

New-Service

[-Name] <String>  
[-BinaryPathName] <String>  
[-DisplayName <String>]  
[-Description <String>]  
[-StartupType <ServiceStartMode>]  
[-Credential <PSCredential>]  
[-DependsOn <String[]>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

Creates a new service.

## New-TimeSpan

Creates a timespan object.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

New-TimeSpan [

[-Start] <DateTime>] [  
[-End] <DateTime>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

New-TimeSpan

[-Days <Int32>]  
[-Hours <Int32>]  
[-Minutes <Int32>]  
[-Seconds <Int32>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The new-timespan Cmdlet creates a timespan object that acts like a defined block of time. You can then add or subtract the time from a DateTime object to create additional DateTime objects.

## New-Variable

Creates a new variable.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

New-Variable

[-Name] <String> [  
[-Value] <Object>]  
[-Description <String>]  
[-Option <ScopedItemOptions>]  
[-Force]  
[-PassThru]  
[-Scope <String>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

The new-variable Cmdlet creates a new variable within a namespace. It does not assign a value to the variable unless instructed to do so.

## Out-Default

The default controller of output.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Out-Default

[-InputObject <PSObject>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The standard treatment at the end of a pipeline is to send all objects to out-default. Out-default then sends them all to format-default. It takes the objects that return, and sends them to the default destination. For this reason, it is functionally equivalent to out-host, but is not called from the console.

## Out-Host

Sends the pipelined output to the host.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Out-Host

[-Paging]  
[-InputObject <PSObject>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The out-host Cmdlet does not function by itself, but is one of several output Cmdlets that take pipelined input. Typically, out-host does not need to be explicitly used since it is the default output. The typical behavior is to do some actions that produce objects, format those objects using a format-\* command, and then output those objects using out-\*.

## Out-Printer

Sends the output to a printer.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Out-Printer [

[-Name] <String>]  
[-InputObject <PSObject>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The out-printer Cmdlet does not function by itself, but is one of several output cmdlets that take pipelined input. When placed at the end of a pipeline, the cmdlet sends the result to the default printer unless a different one is specified.

## Out-String

Sends output to the pipeline as strings.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Out-String

[-Stream]  
[-Width <Int32>]  
[-InputObject <PSObject>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The out-string Cmdlet takes input from the pipeline and returns a list of strings. The length is determined as one string per line of input. This cmdlet does not work alone and must have pipeline input.

## Pop-Location

Changes the current working location to the location specified by the last entry pushed onto the stack.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Pop-Location

[-PassThru]  
[-StackName <String>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The place to which data is written and from which data is accessed is called the location. The current working location is the location that the shell uses by default unless you specify another one. If your working location is C:\Drawing s\CommercialBldgs, then data will be saved to and retrieved from there. Locations can be stored on a stack and pop will access the last location on top of the stack. Push will add a location to the top of the stack.

## Push-Location

Pushes a location to the stack.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Push-Location [

[-Path] <String>]  
[-PassThru]  
[-StackName <String>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

Pushes the current location onto the stack, and then changes the current working location.

## Read-Host

Reads a line of input from the host console.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Read-Host [

[-Prompt] <Object>]  
[-AsSecureString]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The read-host accepts or reads what is input into the host, usually from a keyboard. It allows you to compose a prompt and assign variables to what is input at the prompt. You can also assign a variable to the cmdlet itself.

## Remove-ItemProperty

Removes a property and its value from the location.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Remove-ItemProperty

[-Path] <String[]>  
[-Name] <String[]>  
[-Force]  
[-Filter <String>]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Credential <PSCredential>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

The remove-property Cmdlet removes a property from a location or namespace. The value of the property is also removed.

## Remove-PSDrive

Removes a drive from its location.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Remove-PSDrive

[-Name] <String[]>  
[-PSProvider <String[]>]  
[-Scope <String>]  
[-Force]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

Used to remove an existing drive. When a provider name and no drive name is specified, then all drives exposed by that provider are removed. An error will result if the current working location is set to a container on a drive that is being removed.

## Remove-PSSnapin

Remove PSSnapIn(s) from the current console process.

Snap-In: Microsoft.PowerShell.Core

Syntax:

Remove-PSSnapin

[-Name] <String[]>  
[-PassThru]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

The remove-PSSnapIn Cmdlet removes the given PSSnapIn(s) from the current console process. After removing the PSSnapIn(s) from the current console, the associated cmdlets or providers are not shown to the user in the current process. In other words, the console information in the memory is updated, but the console file is not updated. Remove-PSSnapIn does not check the PowerShell version before removing the PSSnapIn(s) from the current process. If PSSnapIn(s) cannot be removed, show a warning and fail the removal of specific PSSnapIn(s). Default PSSnapIn (s) cannot be removed. A user might be able to add a new version of same PSSnapIn(s) that was removed. Console file is updated only when the user calls the export-console Cmdlet. Current console file name is available in $console variable.

## Rename-Item

Changes the name of an existing item.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Rename-Item

[-Path] <String>  
[-NewName] <String>  
[-Force]  
[-PassThru]  
[-Credential <PSCredential>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

The rename-item Cmdlet changes an item's name within a namespace. It does not affect the stored data.

## Rename-ItemProperty

Renames a property at its location.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Rename-ItemProperty

[-Path] <String>  
[-Name] <String>  
[-NewName] <String>  
[-PassThru]  
[-Force]  
[-Filter <String>]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Credential <PSCredential>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

The rename-property Cmdlet renames a property at a given location. The value is unaltered.

## Resolve-Path

Resolves the wildcard characters in a path.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Resolve-Path

[-Path] <String[]>  
[-Credential <PSCredential>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

Given a Ps path, the resolve-path Cmdlet returns the files and containers (folders) at the specified location. The names appear just as they are, which means if they are capitalized in the file system, they will show as capitalized. File names will have an extension, while container names will not.

## Resume-Service

Resumes a suspended service.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Resume-Service

[-Name] <String[]>  
[-PassThru]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Resume-Service

[-PassThru] -DisplayName <String[]>  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Resume-Service

[-PassThru]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-InputObject <ServiceController[]>]  
[-Verbose]  
[-Debug]  
  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

The resume-service Cmdlet sends a resume message to the indicated services. If they have been suspended, they will resume service. If they are currently running, the message is ignored.

## Select-String

Lets you search through strings or files for patterns.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Select-String

[-Pattern] <String[]> -InputObject <PSObject>  
[-SimpleMatch]  
[-CaseSensitive]  
[-Quiet]  
[-List]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Select-String

[-Pattern] <String[]>  
[-Path] <String[]>  
[-SimpleMatch]  
[-CaseSensitive]  
[-Quiet]  
[-List]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

Objects provided by the pipeline or by the InputObject parameter, or a string presented by the -Text parameter are checked to see if they are a fileInfo object. If so, the file is read and the strings from it are checked. If the incoming objects are MatchInfo objects, the Line property is searched. If matched, the MatchInfo object is sent to the pipeline if appropriate. Otherwise, the object's ToString() method is called and the resulting string is checked. The matching is done in the current culture. If -SimpleMatch is false (the default case), a regular expression match is performed; otherwise, a simple string match is performed. If -CaseSensitive is false (the default case), the case of alphabetic character s is ignored in comparing them. If -List is false (the default condition), an output match object will result f or every match found in each file. If it is true, only one match object will result for each file evaluated, and the line numbers, string matched, and pattern will be set in these objects to the first found. Evaluation of each file stops at that point, for efficiency. If -Quiet is true, all the above mentions of output are overridden; only a Boolean is sent down the pipeline, true if the pattern was found and false otherwise. In this situation, the cmdlet stops processing at the first successful match and returns true. If both -List and -Quiet parameters are given, the match object for the first match is returned. The MatchInfoclass's ToString() method operates differently if the MatchInfo is operating on file contents. In this case, it displays "{0}:{1}:{2}", filename, lineNumber, line. However, when it is operating on objects or strings, it will simply give the line property of the MatchInfo object. The display mechanism displays MatchInfo objects in this format.

## Set-Acl

Set a resource's Access Control List Properties.

Snap-In: Microsoft.PowerShell.Security

Syntax:

Set-Acl

[-Path] <String[]>  
[-AclObject] <ObjectSecurity>  
[-Passthru]  
[-Filter <String>]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

The Set-Acl Cmdlet is a provider cmdlet that sets the security properties associated with the named resource as specified by the security Object. Since the security object includes the PowerShell path as a note from the original resource, the Path parameter is optional. If the Path parameter is null, then the path in the security object shall be used.

## Set-Alias

Maps an alias to a cmdlet.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Set-Alias

[-Name] <String>  
[-Value] <String>  
[-Description <String>]  
[-Option <ScopedItemOptions>]  
[-PassThru]  
[-Scope <String>]  
[-Force]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

You can give any cmdlet a nickname or alias. This cmdlet sets or changes that pairing. You can even give a cmdlet several alias names.

## Set-AuthenticodeSignature

Places an authenticode signature in an Ps script or other file.

Snap-In: Microsoft.PowerShell.Security

Syntax:

Set-AuthenticodeSignature

[-FilePath] <String[]>  
[-Certificate] <X509Certificate2>  
[-IncludeChain <String>]  
[-TimestampServer <String>]  
[-Force]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

The set-authenticodesignature Cmdlet places an authenticode signature in any file that has SIP support. In a Ps script file, this signature takes the form o f a block of text. This signature block indicates the end of the instructions that are executed in the script. No code shall be executed once the signature block is begun. If a signature exists in the file when this cmdlet is run, that signature is removed. The certificate is required, and if missing, the user shall be prompted to provide a certificate. However, in this case the user will be unable to enter a certificate, so the cmdlet will fail. If the certificate is invalid or not valid for signing, set-authenticodesignature shall fail. Before a signature is set in the file and after the certificate has been validated, the value of the Boolean shell variable "SigningApproved" must checked. If this variable is unset or set to something other than true, the ShouldProcess method shall prompt the user to confirm the signing of the script. If TimeStampServer is set, this indicates that the designated URL should be use d to acquire a TimeStamp for the signature. The "IncludeChain" parameter indicates how much of the certificate trust-chain to embed in the signature block. The three choices allowed for this parameter are: none Only the signer's certificate shall be encoded and embedded into the script file. notroot All certificates in the signer's trust-chain, except for the root certificate, shall be encoded and embedded into the script file. fullchain All certificates in the signer's trust-chain including the root certificate shall be encoded and embedded into the script file.

## Set-Content

The set-content command sets the content in the item at the specified location.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Set-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>]

Description:

The set-content Cmdlet sets the input content into the specified item or file. Whereas add-content appends to a file, set-content replaces whatever is in the file with the new content. The content can be input directly or pipelined into the cmdlet. The content may need to be providerspecific depending upon what the destination file is.

## Set-Date

Sets the system time on the host system.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Set-Date

[-Date] <DateTime>  
[-DisplayHint <DisplayHintType>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Set-Date

[-Adjust] <TimeSpan>  
[-DisplayHint <DisplayHintType>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

The set-date Cmdlet sets the system date and time and creates a DateTime object that can be passed down the pipeline.

## Set-Item

The set-item command sets the value of a pathname within a provider to the specified value.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Set-Item

[-Path] <String[]> [  
[-Value] <Object>]  
[-Force]  
[-PassThru]  
[-Filter <String>]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Credential <PSCredential>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

Sets data into the provider specific item. The cmdlet has many uses including set data in a file, assign a value in a provider, or create a registry key.

## Set-ItemProperty

Sets a property at the specified location to a specified value.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Set-ItemProperty

[-Path] <String[]> -InputObject <PSObject>  
[-PassThru]  
[-Force]  
[-Filter <String>]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Credential <PSCredential>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Set-ItemProperty

[-Path] <String[]>  
[-Name] <String>  
[-Value] <Object>  
[-PassThru]  
[-Force]  
[-Filter <String>]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Credential <PSCredential>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
  
[-WhatIf]  
[-Confirm]

Description:

There can be many different properties in many different locations. For example , there may be a specialized cursor program on the machine and it might have a list of properties available in the registry location HKLM:\Software\Vizsoft. You can access that list by setting a new location (set-location HKLM:\Software) and then calling the property list (get-property HKLM:\Software\Vizsoft). You can set some of those properties such as version number.

## Set-Location

Sets the current working location to a specified location.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Set-Location [

[-Path] <String>]  
[-PassThru]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Set-Location

[-PassThru]  
[-StackName <String>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The set-location Cmdlet sets the working location to a specified location. That location could be a directory, sub-directory, registry location, or another location stack.

## Set-PSDebug

Turn Ps script debugging features on and off, and set trace level.

Snap-In: Microsoft.PowerShell.Core

Syntax:

Set-PSDebug

[-Trace <Int32>]  
[-Step]  
[-Strict]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Set-PSDebug

[-Off]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

This command allows you to control the script debugging options in the Ps interpreter. When -trace is set to 1, only the line of script to be executed will be emitted. When -trace is set to 2, variable assignments, functions, and script calls will also be traced. If -step is set, the user is presented with the same choices as the -Confirm parameter before each line of the script is executed. The behavior of the system in response to the confirmation prompt in step mode is as follows:

* "Yes" executes the single line
* "Yes to All" exits step mode (the script continues to run with no confirmation prompts)"No" and "No To All" exit the script
* "Suspend" launches a nested shell and then returns to the confirmation prompt w hen the nested shell exits.

## Set-Service

Makes and sets changes to the properties of a service.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Set-Service

[-Name] <String>  
[-DisplayName <String>]  
[-Description <String>]  
[-StartupType <ServiceStartMode>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

The set-service Cmdlet can be used to make changes in the properties of an existing service.

## Set-TraceSource

Sets or removes the specified options and trace listeners from the specified trace source instance(s).

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Set-TraceSource

[-Name] <String[]> [  
[-Option] <PSTraceSourceOptions>]  
[-ListenerOption <TraceOptions>]  
[-FilePath <String>]  
[-Force]  
[-Debugger]  
[-PSHost]  
[-PassThru]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Set-TraceSource

[-Name] <String[]>  
[-RemoveListener <String[]>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Set-TraceSource

[-Name] <String[]>  
[-RemoveFileListener <String[]>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The Set-TraceSource Cmdlet has two general categories of usefulness.

1. Using Flags, you set what traceable entities the TraceSources selected are set to emit trace messages for. \ Examples are: constructor, error, method, property, event, executionFlow, data, errors, or all of these. The first five are specific areas; the next three are combinations of all the separate details specific to the general concept they name, and all sets all forms of tracing active.
2. Using the listener settings, the output of tracing can be directed to files, console, a host, or a custom listener (derived from System. Diagnostics.TraceListener) designed to receive and analyze the trace output. The categories available can be listed using get-tracesource. Different flags may be set for different categories in order to tune tracing to the need at hand, and different listeners may be used for different categories.

**NOTE:** File and custom listeners are easy to specify by name in order to remove them. Console, Host, and Debug listeners are created by default with the name Console, Host, and Debug, respectively, so they can be removed by name as well, using the removeListener option. It is possible to locate any of these listeners and alter their name property so that you can create more than one Console, Host, or Debug listener. If this is done, the -RemoveListener option will be able to remove those listeners by the name they have been given.

## Sort-Object

Sorts the input objects by property values.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Sort-Object [

[-Property] <Object[]>]  
[-Descending]  
[-Unique]  
[-InputObject <PSObject>]  
[-Culture <String>]  
[-CaseSensitive]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The Sort-Object Cmdlet is essentially a filter used to sort objects according to parameters, properties, or attributes, and then pass them down the pipeline. It is usually used in conjunction with cmdlets that collect data such as get-service or get-process. The cmdlet works with files, modules, registry keys, and other objects. Using sort-object without any parameters will work, but the cmdlet will try to sort with the properties in DefaultKeyPropertySet if it's defined. Otherwise it will sort by default methods that are dependent upon what is being sorted.

## Split-Path

Given an Ps path(s) it streams a string with the qualifier, parent path or leaf item.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Split-Path

[-Path] <String[]>  
[-Parent]  
[-Resolve]  
[-Credential <PSCredential>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Split-Path

[-Path] <String[]>  
[-Qualifier]  
[-Resolve]  
[-Credential <PSCredential>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Split-Path

[-Path] <String[]>  
[-NoQualifier]  
[-Resolve]  
[-Credential <PSCredential>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Split-Path

[-Path] <String[]>  
[-Leaf]  
[-Resolve]  
[-Credential <PSCredential>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Split-Path

[-Path] <String[]>  
[-Resolve]  
[-IsAbsolute]  
[-Credential <PSCredential>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The parse-path Cmdlet takes a Ps path and returns the result of the parse as an object. The object can be sent to the pipeline. Which node of the path is sent depends upon the parameters that are set in the cmdlet statement.

## Start-Sleep

Suspend shell, script, or runspace activity for the specified period of time.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Start-Sleep

[-Seconds] <Int32>  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Start-Sleep -Milliseconds <Int32>

[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The start-sleep Cmdlet suspends shell, script, or runspace activity for the specified period of time. It can be useful in a script to wait for an operation to complete, or in a loop to wait a specific time and do an operation repeatedly. The -seconds parameter takes an integer that represents the time, in seconds, to suspend execution. The -milliseconds parameter takes an integer that represents the time, in milliseconds, to suspend execution.

## Start-Transcript

Starts a transcript of a command shell session.

Snap-In: Microsoft.PowerShell.Host

Syntax:

Start-Transcript [

[-Path] <String>]  
[-Append]  
[-Force]  
[-NoClobber]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

The start-transcript Cmdlet makes a file in the My Documents folder of the home directory. In that file the entire session will be copied, line by line, until the transcript is stopped by ending the session or with the stoptranscript Cmdlet.

## Stop-Transcript

Stops the transcription process.

Snap-In: Microsoft.PowerShell.Host

Syntax:

Stop-Transcript

[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The stop-transcription Cmdlet simply stops a transcription that was previously started. The cmdlet displays the filename of the transcript file on the console and the transcript file shows the stop date and time. You can also stop a transcription by ending the session.

## Suspend-Service

Suspends a running service.

Snap-In: Microsoft.PowerShell.Management

Syntax:

Suspend-Service

[-Name] <String[]>  
[-PassThru]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Suspend-Service

[-PassThru] -DisplayName <String[]>  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Suspend-Service

[-PassThru]  
[-Include <String[]>]  
[-Exclude <String[]>]  
[-InputObject <ServiceController[]>]  
[-Verbose]  
[-Debug]  
  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]  
[-WhatIf]  
[-Confirm]

Description:

The suspend service Cmdlet suspends a running service. The service is still running, but its action is halted until resumed with resume-service.

## Tee-Object

Sends input objects to two places.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Tee-Object

[-FilePath] <String>  
[-InputObject <PSObject>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Tee-Object

[-InputObject <PSObject>] -Variable <String>  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

It is possible to take an input object, process it, and send it to more than one destination. However, it often requires more than one command. The Tee-Object Cmdlet allows you to send the result along two paths.

## Trace-Command

Enables tracing of the specified trace source instance(s) for the duration of the expression or command.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Trace-Command

[-Name] <String[]>  
[-Expression] <ScriptBlock> [  
[-Option] <PSTraceSourceOptions>]  
[-InputObject <PSObject>]  
[-ListenerOption <TraceOptions>]  
[-FilePath <String>]  
[-Force]  
[-Debugger]  
[-PSHost]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Trace-Command

[-Name] <String[]>  
[-Command] <String> [  
[-Option] <PSTraceSourceOptions>]  
[-InputObject <PSObject>]  
[-ArgumentList <Object[]>]  
[-ListenerOption <TraceOptions>]  
[-FilePath <String>]  
[-Force]  
[-Debugger]  
[-PSHost]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

This cmdlet turns on tracing for the trace source(s) for the given expression. The behavior is exactly same as in Set-TraceSource, except the setting is temporary and applied during the execution of the given expression. The second parameterSet using -Command behaves more like a command would in a pipeline. For example, command discovery is not repeated for each incoming object.

## Update-FormatData

Updates and appends format data files.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Update-FormatData [

[-AppendPath] <String[]>]  
[-PrependPath <String[]>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The Ps directory contains several format files. They have the suffix .ps1xml and are XML files. When you start a shell, Ps installs the format data from those files so the shell can run. You can build your own format.ps1xml files. When you do, the update-formatdata Cmdlet adds those files to the format data that it loads in the shell. You do not need to restart the shell to update the format data.

## Update-TypeData

Updates the types.ps1xml file in the Microsoft Shell.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Update-TypeData [

[-AppendPath] <String[]>]  
[-PrependPath <String[]>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The update-typedata is used with a minishell that can have its own, specific types and formats. That data is stored in a types.ps1xml file. For example, you could have an Admin.types.ps1xml that holds type data for a specific administrator's minishell. The update-typedata Cmdlet will update the system's extended type definitions from the one in the minishell.

## Write-Debug

Writes a debug message to the host display.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Write-Debug

[-Message] <String>  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

Used in scripts, the write-debug Cmdlet can be used to send specialized message s to the console from a script or command. It works from the WriteDebug() method.

## Write-Error

Writes an error object and sends it to the pipeline.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Write-Error

[-Message] <String>  
[-Category <ErrorCategory>]  
[-ErrorId <String>]  
[-TargetObject <Object>]  
[-RecommendedAction <String>]  
[-CategoryActivity <String>]  
[-CategoryReason <String>]  
[ -CategoryTargetName <String>]  
[-CategoryTargetType <String>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Write-Error -Exception <Exception>

[-Message <String>]  
[-Category <ErrorCategory>]  
[-ErrorId <String>]  
[-TargetObject <Object>]  
[-RecommendedAction <String>]  
[-CategoryActivity <String>]  
[-CategoryReason <String>]  
[-CategoryTargetName <String>]  
[-CategoryTargetType <String>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Write-Error -ErrorRecord <ErrorRecord>

[-RecommendedAction <String>]  
[-CategoryActivity <String>]  
[-CategoryReason <String>]  
[-CategoryTargetName <String>]  
[-CategoryTargetType <String>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The write-error Cmdlet is one that is used in scripts and routines. It allows you to write error messages and include other information such as Id, object data, and suggested actions.

## Write-Output

Writes an object to the pipeline.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Write-Output

[-InputObject] <PSObject[]>  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The write-verb Cmdlet is generally used in scripts and commands to create objects as input to the pipeline or the display. They vary mostly in what they send out - whether it's an error, message, object, and so on. The write-object Cmdlet specifically places an object or array of objects in the pipeline.

## Write-Progress

Sends a progress record to the host.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Write-Progress

[-Activity] <String>  
[-Status] <String> [  
[-Id] <Int32>]  
[-PercentComplete <Int32>]  
[-SecondsRemaining <Int32>]  
[-CurrentOperation <String>]  
[-ParentId <Int32>]  
[-Completed]  
[-SourceId <Int32>]  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

Some scripts or commands take some time to run. Familiar examples of a long running command would be Format or Defrag. The write-progress Cmdlet collects various parameter data, then describes and updates the state of the activity to t he console.

## Write-Warning

Write a warning message.

Snap-In: Microsoft.PowerShell.Utility

Syntax:

Write-Warning

[-Message] <String>  
[-Verbose]  
[-Debug]  
[-ErrorAction <ActionPreference>]  
[-ErrorVariable <String>]  
[-OutVariable <String>]  
[-OutBuffer <Int32>]

Description:

The Write-Warning Cmdlet allows the script author to create a warning message that may be displayed to the user. This message is sent directly to the host and allows for direct notification.

## Appendix B: Type Reference

### Overview

As discussed throughout this book, and in some detail in [Chapter 5](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/chapter-5-37.xhtml#ch05), PowerShell supports a number of data types for variables called types. [Appendix B](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/appendix-B-251.xhtml#ap02) includes all of the major types, along with their methods and properties. Note that the most generic-possible type, which is Object, is not included since you can't do much with it directly.

For each type, we'll also give you the PowerShell nickname, which is the type reference you can use to cast a variable into a specific type. For example:

$var = 5

Creates a variable $var with the contents 5, and allows $var to remain a generic Object. However, the following expression forces $var to be a System.Int32 or an integer:

[int]$var = 5

The [int] part is the "nickname" for the System.Int32 type.

Note that this isn't a comprehensive reference of every single type that PowerShell supports since that would be a huge list that would not be terribly useful as far as Windows administrative tasks go. With this in mind, we've limited this reference to those types you'll use often including:

* Boolean (System.Boolean)
* Datetime (System.DateTime)
* Double (System.Double)
* Hashtable (System.Collections.Hashtable. Covered in [Chapter 5](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/chapter-5-37.xhtml#ch05) instead of in this appendix.)
* Int (System.Int32)
* Regex (System.Text.RegularExpressions.Regex. Covered in [Chapter 7](https://cdn2.percipio.com/1652199683.50a22722a1db88d254475b9586ae1891e1952921/eod/books/18544/OEBPS/chapter-7-51.xhtml#ch07) instead of in this appendix.)
* String (System.String)

##### ***Official Names vs. PowerShell Names***

*In the bulleted list above, the name in parentheses is the official underlying .NET Framework class name for the type. The first name listed is the one you can place inside [square brackets] within PowerShell. We've included the .NET Framework name to aid you in learning more about the base type. Visit*[*http://msdn.microsoft.com/library*](http://msdn.microsoft.com/library)*and search for the .NET Framework name to find the full documentation on the type.*

We've also tried to filter the list of properties and methods to make it more useful and concise by focusing on the ones you'll use on a daily basis in administrative scripting‥ For a complete list you can refer to the Microsoft .NET SDK documentation for a given type.

Keep in mind that many objects have similar or identical properties and methods. We've tried to list all of them with each object for your convenience. However, our examples may be somewhat generic to help keep things clear. For example, most types have a **Clone** method, so we've used the same example for **Clone** throughout to help you spot the similarity and understand that it's used the same no matter what type you're working with.

## Boolean

A "true" or "false" value. This type doesn't include any significant methods or properties. Note that PowerShell includes built-in variables named $true and $false that represent the two possible Boolean values:

PS C:\> [boolean]$b = $true

PS C:\> $b

True

PS C:\>

## DateTime

A date, time, or date-and-time value. Note that all DateTime values include both a date and a time component. If you don't specify a date or a time, then that portion of the value is typically set to zero when the value is stored. Methods and properties include:

##### **AddDays**

##### **AddHours**

##### **AddMilliseconds**

##### **AddMinutes**

##### **AddMonths**

##### **AddSeconds**

##### **AddTicks**

##### **AddYears**

Adds the specified interval to the value. To subtract a value, specify a negative interval. Note that the type does include a **Subtract()** method; from within a PowerShell script one of these "Add" methods is generally easier to work with.

PS C:\> [datetime]$d = "1/1/2000 3:00 PM"

PS C:\> $d.adddays(1)

Sunday, January 02, 2000 3:00:00 PM

##### **GetDateTimeFormats**

Returns all possible string representations of the value.

PS C:\> [datetime]$d = "1/1/2000 3:00 PM"

PS C:\> $d.getdatetimeformats()

1/1/2000

1/1/00

01/01/00

01/01/2000

00/01/01

2000-01-01

01-Jan-00

Saturday, January 01, 2000

January 01, 2000

Saturday, 01 January, 2000

01 January, 2000

Saturday, January 01, 2000 3:00 PM

Saturday, January 01, 2000 03:00 PM

Saturday, January 01, 2000 15:00

Saturday, January 01, 2000 15:00

January 01, 2000 3:00 PM

January 01, 2000 03:00 PM

(output truncated in this example)

##### **IsDaylightSavingTime**

Indicates whether this instance of DateTime is within the Daylight Saving Time range for the current time zone.

PS C:\> [datetime]$d = "1/1/2000 3:00 PM"

PS C:\> $d.isdaylightsavingtime()

False

##### **ToFileTime ToFileTimeUTC**

Converts the value to a Windows file time. The "UTC" version provides a Universal Time Constant version of the value.

PS C:\> [datetime]$d = "1/1/2000 3:00 PM"

PS C:\> $d.tofiletimeutc()

125912124000000000

PS C:\> $d.tofiletime()

125912412000000000

##### **ToLocalTime**

Converts the value to a local time.

PS C:\> [datetime]$d = "1/1/2000 3:00 PM"

PS C:\> $d.tolocaltime()

Saturday, January 01, 2000 7:00:00 AM

##### **ToLongDateString**

##### **ToLongTimeString**

##### **ToShortDateString**

##### **ToShortTimeString**

Converts the value to the appropriate date or time format.

PS C:\> [datetime]$d = "1/1/2000 3:00 PM"

PS C:\> $d.tolongdatestring()

Saturday, January 01, 2000

PS C:\> $d.toshortdatestring()

1/1/2000

##### **ToUniversalTime**

Converts the value to a Universal Time Constant (UTC), taking time zone changes into account.

PS C:\> [datetime]$d = "1/1/2000 3:00 PM"

PS C:\> $d.touniversaltime()

Saturday, January 01, 2000 11:00:00 PM

##### **Day**

##### **DayOfWeek**

##### **DayOfYear**

##### **Hour**

##### **Millisecond**

##### **Minute**

##### **Month**

##### **Second**

##### **Ticks**

##### **TimeOfDay**

##### **Year**

Returns only the specified portion of the value.

PS C:\> [datetime]$d = "1/1/2000 3:00 PM"

PS C:\> $d.day

1

PS C:\> $d.month

1

PS C:\> $d.dayofweek

Saturday

PS C:\> $d.timeofday

Days : 0

Hours : 15

Minutes : 0

Seconds : 0

Milliseconds : 0

Ticks : 540000000000

TotalDays : 0.625

TotalHours : 15

TotalMinutes : 900

TotalSeconds : 54000

TotalMilliseconds : 54000000

## Double

A double-precision floating point number, capable of including fractions. Can store values ranging from negative 1.79769313486232e308 to positive 1.79769313486232e308. Methods and properties include:

##### **ToString**

Returns a string representation of the value.

PS C:\> [double]$d = 123.456789

PS C:\> $d.tostring()

123.456789

## Int

An integer (non-fractional) value. Can store values ranging from negative 2,147,483,648 through positive 2,147,483,647. Methods and properties include:

##### **ToString**

Returns a string representation of the value.

PS C:\> [double]$d = 123.456789

PS C:\> $d.tostring()

123.456789

## String

A string of text characters. Methods and properties include:

##### **Clone**

Returns a reference to this instance of String.

PS C:\> [string]$a = "hello"

PS C:\> $a

hello

PS C:\> $b = $a.clone()

PS C:\> $b

hello

PS C:\> $a

hello

PS C:\> $b = "world"

PS C:\> $b

world

PS C:\> $a

hello

PS C:\>

##### **Contains**

Returns a value indicating whether the specified String object occurs within this string.

PS> [string]$a = "hello"

PS> $a.contains("e")

True

##### **EndsWith**

Determines whether the end of an instance of String matches a specified string.

PS C:\> [string]$a = "hello"

PS C:\> $a.endswith("lo")

True

##### **Equals**

Determines whether two String objects have the same value.

PS C:\> [string]$a = "hello"

PS C:\> [string]$b = "world"

PS C:\> $a.equals($b)

False

##### **IndexOf**

Reports the index of the first occurrence of a String or one or more characters within this string.

PS C:\> [string]$a = "hello"

PS C:\> $a.indexof("l")

2

##### **Insert**

Inserts a specified instance of String at a specified index position in this instance.

PS C:\> [string]$a = "hello"

PS C:\> $b = $a.insert(2,"-st-")

PS C:\> $b

he-st-llo

##### **LastIndexOf**

Reports the index position of the last occurrence of a specified Unicode character or String within this instance.

PS C:\> [string]$a = "hello"

PS C:\> $a.lastindexof("l")

3

##### **Length**

Gets the number of characters in this instance.

PS C:\> [string]$a = "Hello"

PS C:\> $a.length

5

##### **PadRight**

Left-aligns the characters in this string, padding the right with spaces or a specified Unicode character for a specified total length.

PS C:\> [string]$a = "hello"

PS C:\> "-" + $a.padright(8) + "-"

-hello -

##### **Remove**

Deletes a specified number of characters from this instance.

PS C:\> [string]$a = "hello"

PS C:\> $a.remove(3,2)

hel

##### **Replace**

Replaces all occurrences of a specified Unicode character or String in this instance with another specified Unicode character or String.

PS C:\> [string]$a = "hello"

PS C:\> $a.replace("l","p")

heppo

##### **Split**

Returns a String array containing the substrings in this instance that are delimited by elements of a specified Char or String array.

PS C:\> [string]$a = "one,two,three"

PS C:\> $b = $a.split(",")

PS C:\> $b[0]

one

PS C:\> $b[1]

two

PS C:\> $b[2]

three

##### **StartsWith**

Determines whether the beginning of an instance of String matches a specified string.

PS C:\> [string]$a = "hello"

PS C:\> $a.startswith("f")

False

##### **Substring**

Retrieves a substring from this instance.

PS C:\> [string]$a = "hello"

PS C:\> $a.substring(2,3)

llo

##### **ToCharArray**

Copies the characters in this instance to a Unicode character array.

PS C:\> [string]$a = "hello"

PS C:\> $b = $a.tochararray()

PS C:\> $b[0]

h

PS C:\> $b[4]

o

##### **ToLower**

Returns a copy of this String converted to lowercase.

PS C:\> [string]$a = "Hello"

PS C:\> $a.tolower()

hello

##### **ToUpper**

Returns a copy of this String converted to uppercase.

PS C:\> [string]$a = "Hello"

PS C:\> $a.toupper()

HELLO

##### **Trim**

Removes all occurrences of a set of specified characters from the beginning and end of this instance.

PS C:\> [string]$a = " hello "

PS C:\> "-" + $a.trim() + "-"

-hello-

##### **TrimEnd**

Removes all occurrences of a set of characters specified in an array from the end of this instance.

PS C:\> [string]$a = " hello "

PS C:\> "-" + $a.trimend() + "-"

- hello-

##### **TrimStart**

Removes all occurrences of a set of characters specified in an array from the beginning of this instance.

PS C:\> [string]$a = " hello "

PS C:\> "-" + $a.trimstart() + "-"

-hello -

## Advanced Types

PowerShell also includes support for a number of advanced types that we don't cover. These include:

* XML (System.Xml.XmlDocument)
* Scriptblock (System.Management.Automation.Scriptblock)

These are very advanced classes, particularly in the case of the XML type, that require a good bit of instruction to utilize. Because manipulating these types within a PowerShell script is outside the scope of what most Windows administrators do, we've elected not to include these types. However, given the .NET Framework class names (shown in parentheses), you can easily research these at <http://msdn.microsoft.com/library> if you find you have need of them.

## All Types

In case you're interested in the complete list of available types, here it is:

* [int] - Integer
* [int[]] — Integer array
* [long] — Long integer
* [long[]] — Long integer array
* [string] — String of characters
* [string[]] — String array
* [char] — Single character
* [char[]] — Array of characters
* [bool] — True or False; [Boolean] is also valid
* [bool[]] — Array of Boolean values
* [byte] — Byte value
* [double] — Double-precision floating number
* [decimal] — Decimal number
* [float] — Floating number
* [single] — Single-precision floating number
* [regex] — Regular expression; the .NET Framework class is System.Text.RegularExpressions.Regex
* [array] — array
* [xml] — XML document; the .NET Framework class is System.Xml.XmlDocument
* [scriptblock] — a script block; the .NET Framework class is System.Management.Automation.ScriptBlock
* [hashtable] — a hashtable or associative array; the .NET Framework class is System.Collections.Hashtable
* [ref] — a PowerShell reference; the .NET Framework class is System.Management.Automation.PSReference
* [psobject] — a PowerShell object; the .NET Framework class is System.Management.Automation.PSObject
* [wmi] — a WMI object; the .NET Framework class is System.Management.ManagementObject
* [wmisearcher] — a WMI Searcher object; the .NET Framework class is System.Management.ManagementObjectSearcher
* [wmiclass] — a WMI class; the .NET Framework class is System.Management.ManagementClass