Bonus Chapter

Handling Exceptions

In This Chapter

- Addressing errors the old way
- Getting to know exceptions
- Doing something with an exception
- Generating your own exceptions

It'd really be nice if all the scripts you write work flawlessly each and every time, and you don't have to worry about running into any errors. When you write a script to automate some task that you're just going to run once, you usually write it to do only what you need it to do — nothing more. For scripts that you'll reuse — or, more important, that others will use — it's always a good idea to build in lots of error handling so that your script doesn't do anything you don't expect it to.

In this chapter, you practice techniques for dealing with runtime errors in your scripts through exception handling. This is really the recommended method for writing error handling code in Windows PowerShell because it applies the same concepts used by the different programming languages used in the .NET framework. You get to see for yourself how much more efficient it is to use it over traditional error handling techniques once you understand the basic concepts for how exception handling works.

Handling Errors the Old-Fashioned Way

The traditional way of handling errors, and the method used most in other Windows scripting languages, involves checking for an error condition and then acting on it if an error is detected. If the script is supposed to read in a file, for example, you'll usually see some code to check whether the file exists before attempting to open it. This system works just fine in most cases, but it can get ugly. If you really want to do your due diligence and check for errors after each piece of code that might generate an error, the error-handling code becomes heavily intertwined with the code that does the actual work, making it difficult to follow what the script is doing in the first place.

To solve this problem, object-oriented programming languages, such as those on which the .NET Framework is built, can use a concept called *exception handling* to defer error handing to other sections of code as needed.

Understanding Exceptions

An *exception* is simply a condition that a program, Cmdlet, function, or object doesn't know how to deal with. Trying to read a file that doesn't exist results in an exception, for example, and trying to divide a number by zero is also an exception. Exceptions aren't just error codes that get thrown about; instead, special `Exception` objects get created, and these objects contain various properties that you can use to get more information about the error. Table BC-1 shows some of the properties that often appear in `Exception` objects. (Not all properties are required to have values.)

Table BC-1 Exception Properties

Property Name	*Definition*
`Data`	Contains any data the exception may want to define in key/value pairs.
`HelpLink`	Defines the URL for the help file containing information about the exception.
`InnerException`	Contains another exception object if the current exception is encapsulating another exception.
`Message`	Contains details about an exception. This property is what you're most interested in.
`StackTrace`	Contains a stack trace so you can try to determine where the error came from. In the case of a script, the property may also show the filename and line within the script that caused the exception.

When an exception occurs, it's thrown up the stack until some piece of code handles it — a situation otherwise known as *trapping an exception.* In practice, if an exception occurs in one of your functions, the function gets the first opportunity to handle the exception. If nothing is defined in the function for that exception, the exception goes up to the code that called the function (such as your main script) and looks for an exception handler there. If you don't write any code whatsoever to handle the exception, eventually, the exception is simply trapped by Windows PowerShell, where the execution of the code usually stops, and the exception is displayed onscreen.

You can see this process in action yourself by running `Get-ChildItem` on a nonexistent folder, as in this example:

```
Get-ChildItem C:\MyMissingFolder
```

If that folder doesn't exist, Windows PowerShell spits out an error message, saying that it can't find the path `C:\MyMissingFolder` because the path doesn't exist. That message is from the `Message` property of the exception, which in this case is `ItemNotFoundException`.

Trapping Exceptions

In most cases, exceptions are created and thrown automatically by the different .NET classes and Cmdlets you run in Windows PowerShell, if PowerShell runs into a condition that triggers exceptions. In those cases, you can do either of two things: ignore the exceptions and allow your script simply to die with an error message, or try to detect the exceptions and take some appropriate action.

Here's a script that attempts to ping a given hostname by using a `Pingit` function, which in turn uses an instance of `System.Net.NetworkInformation.Ping`:

```
Function Pingit([string]$hostname) {
   Trap [Exception] {
      Write-Host "ERROR: Unable to ping $hostname"
      continue
   }

   $netping = New-Object System.Net.NetworkInformation.Ping
   $result = $netping.send($hostname)
   if ($result.status -eq "Success")
   {
      Write-Host "$hostname is online!"
   }
   else
   {
      Write-Host "$hostname is offline!"
   }
}

$targethost = "computer1"
Write-Host "Pinging $targethost"
Pingit $targethost
Write-Host "Script Complete!"
```

A few things can happen when you ping a hostname. If your computer can resolve the hostname to an IP address, it tries to ping it, and you either get a response or don't. If your computer can't figure out the IP address for the hostname, it can't do anything, and in the case of the `System.Net.NetworkInformation.Ping` class, it throws an exception. To handle the potential for getting an exception, you define a trap to handle the exception just like this:

```
Trap [Exception] {
      Write-Host "ERROR: Unable to ping $hostname - " + $_.Exception.Message
      continue
}
```

You define a trap by using the `Trap` keyword followed by the kind of exception you want to trap. Then, in the script block following the `Trap` keyword, you put whatever code you want to run to handle the exception. If you don't specify an exception and just use the `Trap` keyword by itself with nothing following it, you're telling PSH that you want this `Trap` to handle any kind of exception that may be generated, basically turning it into catch-all exception handler. You can also do what I did and just use `[Exception]`, which is functionally the same thing as not providing an exception name, because every `Exception` in .NET is derived from a parent `Exception` class.

You can use `$_.Exception` within a `Trap` code block to refer to the `Exception` object that triggered the trap to be executed.

One more thing to mention is the last part of the `Trap` statement, which defines what to do when the `Trap` code block completes. You can choose among three keywords:

- `Break`: Execution of the current scope (such as the function, if the trap is defined within a function) stops immediately, and the exception is thrown again to allow the higher-level scope (such as your script) to trap it again.
- `Continue`: The script continues from the next line after where the exception was thrown.
- `Return [object]`: Execution of the current scope stops immediately, but unlike `Break`, which throws the exception again, `Return [object]` simply returns the object you supply to it. If no object is specified, `Return [object]` contains the `ErrorRecord` object that was trapped by the `Exception`.

So why would you ever bother putting in a specific exception class for the trap? You'll want to do that if you want to do something different for different kinds of exceptions. If you use ADO.NET, for example, you can get

different exceptions depending on the provider you're using; you may get a `System.Data.Odbc.OdbcException` or a `System.Data.SqlClient.SqlException`. In general, if you know the exact `Exception` class you expect to see, use that in your trap statement and then create a generic trap statement for anything else that might come up.

Here's a bit of code that demonstrates this behavior. This mini script connects to a database called `testdb` on the Microsoft SQL server called `sqlsrv1` by using integrated security, performs a simple query, and then closes the connection:

```
Trap [System.Data.SqlClient.SqlException] {
   Write-Host "A SQL Exception occurred:`n" + $_.Exception.Message
   break
}

Trap {
   Write-Host "A general exception occurred:`n" + $_.Exception.Message
   break
}

$sqlConn = New-Object System.Data.SqlClient.SqlConnection
$sqlConn.ConnectionString = "Server=sqlsrv1;Database=testdb;Integrated
               Security=True"
$sqlConn.Open()

$sqlCmd = New-Object Sql.Data.SqlClient.SqlCommand
$sqlCmd.CommandText = "Select top 10 * FROM mytable"
$sqlCmd.Connection = $sqlConn

$sqlDataAdapter = New-Object System.data.SqlClient.SqlDataAdapter
$dataTable = New-Object System.Data.DataTable
$sqlDataAdapter.SelectCommand = $sqlCmd
$sqlDataAdapter.Fill($dataTable)
$sqlConn.Close()
```

Notice the two `Trap` code blocks at the beginning. The first `Trap` statement traps a specific kind of exception — namely, `System.Data.SqlClient.SqlException`, which you typically get if you encounter an error when using ADO.NET to connect to a SQL server. If an exception other than a SQL exception occurs, the second `Trap` statement, which catches everything else, takes over.

It's important to order your `Trap` statements from most specific to most general. Otherwise, the more general `Trap` statement will handle the exception first, because it'll be the first match.

This example also highlights one of the appealing qualities of exception handling. You see that the code that connects to the SQL database and queries some data from it is intact. You don't have a bunch of error checks after each potential error-producing statement, such as opening the SQL connection or filling the data table. Instead, you have all the code in there, assuming that everything will work perfectly; then you add the appropriate `Trap` statements to handle the errors outside the regular execution code. This technique separates functional code from error handling, making it much easier to make code changes without having to weed through error-checking code.

It doesn't matter whether you put a `Trap` statement before or after the code that might generate the exception. Windows PowerShell still looks for the trap when an exception occurs and matches the exception to the appropriate `Trap` statement, based on the scope in which the trap was defined.

Throwing Exceptions

You don't always have to be the recipient of bad news; you can create some yourself! Sure, almost all the .NET classes and PSH Cmdlets can generate exceptions, but you can also create `Exception` objects and throw them yourself. In fact, PSH makes this process super-easy. All you have to do is use the `Throw` keyword followed by the value or object you want to throw, like this:

```
Throw "Houston, we have a problem!"
```

This code works because PSH automatically wraps whatever object you provide the `throw` keyword into a special `RuntimeException` object. You're not limited to these simple `RuntimeException` objects. In fact, you can create instances of any of the built-in .NET `Exception` classes and throw them yourself. Here's an example that's a bit contrived, but you get the gist of it:

```
Function Divide([int]$a, [int]$b)
{
   if ($a -eq 0) {
      throw (New-Object System.DivideByZeroException)
   }
   Write-Host ($b/$a)
}
```

The code checks to make sure that the divisor is zero, and if it is, the code throws a `System.DivideByZeroException`. Now anyone who wants to use this very cool `Divide` function can create a trap to detect a `DivideByZeroException` and take some action, such as prompting for an alternative number to use as the divisor. By leveraging this ability to create and throw exceptions of your own, you can handle all your errors by using `Trap` statements.