Lab 1: PowerShell Batch Import

**Lab Time: 60 Minutes**

**Lab Directory: ECM401.PowerShellImport**

**Lab Overview:**

In this lab, you will create a **Windows PowerShell** cmdlet that will enable you to import batches of documents into SharePoint from a file share. Your cmdlet will optionally read a manifest file located in a given folder that specifies the metadata you want to be associated with files in that folder. In addition to explicit metadata specified in the manifest file, your cmdlet will also extract a subset of document properties from each document so that they are available as columns in the target SharePoint lists.

# Exercise 1: Install the Windows PowerShell Templates

1. In order to create a **Windows PowerShell** cmdlet, you first need to setup your development environment. In your **student/resources/tools** folder, there is a set of Visual Studio templates. If you have not done so already, run the appropriate **VSI** file for your language of choice (CS or VB). This will install the PowerShell project and item templates into Visual Studio.

Note: The basic structure of every PowerShell command is that you have two parts to the command, separated by a hyphen. The first part is a verb and the second part is a noun. In this exercise, you will create two commands. "SPImport-Document" will import a specific set of documents using a file mask, and "SPImport-Folder" will import a folder and its subfolders.

# Exercise 2: Create a Visual Studio Powershell Project

1. Start by creating a new Visual Studio project based on the **Windows PowerShell** project template, as shown in the following screenshot.
2. Select the **PSSnapin.cs** file and choose **View Code** from the context menu. Modify the **Name** property override so that it returns **"SPPowerShell"** instead of the default. You can leave the other overrides with their generated default values.

Note: You may also have to fix the generated class name if you included a '.' character in the project name. This is a bug in the PowerShell project template.

# Exercise 3: Create the Import Document CmdLet

1. Right-click the project node and select **Add New Item** . From the **Add New Item** dialog, select the **Windows PowerShell CmdLet** item.
2. Change the generated **VerbsCommon.Get** parameter to the **Cmdlet** attribute to **"SPImport"** and change the second parameter to **"Document"** . The resulting code should look like the following:

[Cmdlet("SPImport", "Document", SupportsShouldProcess = true)]  
public class SPImport\_Document : Cmdlet  
{  
 #region Parameters  
 /\*  
 [Parameter(Position = 0,  
 Mandatory = false,  
 ValueFromPipelineByPropertyName = true,  
 HelpMessage = "Help Text")]  
 [ValidateNotNullOrEmpty]  
 public string Name  
 {  
  
 }  
 \*/  
 #endregion  
  
 protected override void ProcessRecord()  
 {  
 try  
 {  
 throw new NotImplementedException();  
 }  
 catch (Exception)  
 {  
 }  
 }  
}

1. To allow the caller to specify which documents to upload and where to put them, parameters are needed. **Windows PowerShell** will automatically parse the command line based on attributes you use within the CmdLet. The first parameter accepts a wildcard file mask that will be expanded to a list of files that will then be uploaded to the website or document library.
2. Add the following code to your CmdLet.

[ Parameter(Position = 0,  
 Mandatory = true,  
 ValueFromPipeline = true,  
 ValueFromPipelineByPropertyName = true,  
 HelpMessage = "Enter the fully qualified path to the file(s) you wish to upload. You can enter a standard wildcard mask.")]  
  
[ ValidateNotNullOrEmpty ]  
public string FileMask { get; set; }

1. The second parameter specifies the url of the SharePoint site to which the document will be uploaded. If no document library is specified, then the file is uploaded directly into the site. If the site does not exist, an exception is thrown. Add the following code.

[ Parameter(Position = 1,  
 Mandatory = true,  
 ValueFromPipeline = true,  
 ValueFromPipelineByPropertyName = true,  
 HelpMessage = "Enter the url of the target SharePoint site.")]  
public string SiteUrl { get; set; }

1. The third parameter specifies the name of the target folder or document library within the specified site. If the destination folder or document library does not already exist, then it is created automatically if the ‘AutoCreate’ parameter is also set to true. Add the following code.

[Parameter(Position = 2,  
 Mandatory = false,  
 ValueFromPipeline = true,  
 ValueFromPipelineByPropertyName = true,  
 HelpMessage = "Enter the name of the folder or document library to which the file(s) will be added. To specify a root folder within the site, precede the name with a forward slash '/'.")]  
 public string Destination { get; set; }

1. PowerShell includes the notion of a **switch** parameter, which is converted to a Boolean value. The next parameter is declared as a **SwitchParameter** that specifies whether to create the target folder or document library if it does not already exist within the specified site. The final parameter is another **SwitchParameter** that controls whether existing fiels with the same name are overwritten. Add the following two parameters to your CmdLet class.

[ Parameter(  
 ValueFromPipeline=true,  
 ValueFromPipelineByPropertyName = true,  
 HelpMessage = "Specify whether to create the target library or folder if it does not already exist. Default=false.")]  
public SwitchParameter AutoCreate { get; set; }  
  
[ Parameter(  
 ValueFromPipelineByPropertyName=true,  
 HelpMessage="Specify whether to overwrite files that already exist in the target library or folder. Default=false.")]  
public SwitchParameter Overwrite { get; set; }

1. Finally, you will add the main processing method to your cmdlet. Replace the generated **ProcessRecord** method with the following code snippet.

Code Snippet: 'PowerShell Import Document'

/// <summary>  
/// This is the standard entry point for PowerShell commands.   
/// It is called after the command line arguments have been   
/// parsed and set into the corresponding properties of   
/// the cmdlet class.  
/// </summary>  
protected override void ProcessRecord()  
{  
 try  
 {  
 // Expand the file mask into a list of FileInfo objects.  
 FileInfo[] files = ExpandWildcards(FileMask);  
 if (files.Length == 0)  
 throw new Exception("No files specified.");  
  
 // Locate the target SharePoint site collection.  
 using (SPSite site = new SPSite(SiteUrl))  
 {  
 using (SPWeb web = site.OpenWeb())  
 {  
 // Check the destination to determine what to do.  
 if (string.IsNullOrEmpty(Destination))  
 {  
 // No destination provided, so copy   
 // the files into the site itself.  
 TransferFiles(web, files, null);  
 }  
 else  
 {  
 // Find or create the target folder   
 // and transfer the files.  
 SPFolder targetFolder = FindFolder(  
 web, Destination);  
 TransferFiles(web, files, targetFolder);  
 }  
 }  
 }  
 }  
 catch (Exception x)  
 {  
 Console.WriteLine("Unable to process files: ");  
 Console.WriteLine(x.Message);  
 }  
}

# Exercise 4: Add Helper Methods

1. Next, you will implement a set of helper methods that can be called from both cmdlets to perform the work of finding files, manipulating SharePoint folders and transferring files into the content database.
2. Add a new code file to your project named **"Helpers.cs"** and open it for editing.
3. The following helper method searches for a named folder within a SharePoint web site. If the folder does not exist, then the AutoCreate parameter controls whether the folder is created automatically. If the path includes multiple segments, then each segment is created with the specified name. If the path starts with a forward slash '/', then interpret the string as the fully qualified path of a folder within the site, otherwise treat the first segment as the name of a document library and the remainder as the path of a folder within the library. This gives the most flexibility when moving documents into the content database.

Code Snippet: 'PowerShell - FindFolder Method'

SPFolder FindFolder(SPWeb web, string path)  
{  
 SPFolder targetFolder = web.RootFolder;  
 string[] names = path.Split("/\\".ToCharArray());  
 if (path.StartsWith("/"))  
 {  
 // find or create the specified subfolder  
 foreach (string name in names)  
 if (!string.IsNullOrEmpty(name))  
 targetFolder = FindOrCreateSubfolder(  
 targetFolder, name);  
 }  
 else  
 {  
 // check if the first name matches a list in the site  
 SPList targetList = web.Lists[names[0]];  
  
 // no match, so check for autocreate  
 if (targetList == null)  
 {  
 if (AutoCreate)  
 {  
 // create a new document library in the site  
 Guid guid = web.Lists.Add(names[0],   
 "Created from PowerShell",   
 SPListTemplateType.DocumentLibrary);  
 targetList = web.Lists[guid];  
 }  
 }  
  
 // check again if the list exists  
 if (targetList == null)  
 throw new Exception(string.Format(  
 "Document library '{0}' not found.", path));  
  
 // the default target is the root folder of the list  
 targetFolder = targetList.RootFolder;  
  
 // process the remaining names as folder name.  
 for (int index = 1; index < names.Length; index++)  
 targetFolder = FindOrCreateSubfolder(targetFolder,   
 names[index]);  
 }  
 return targetFolder;  
}

1. Add the following helper method to transfer a collection of files into a folder within a SharePoint site. If the folder is null, then the files are copied directly into the site.

Code Snippet: 'PowerShell - TransferFiles Method'

void TransferFiles(SPWeb web, FileInfo[] files, SPFolder targetFolder)  
{  
 int nCopied = 0;  
 int nSkipped = 0;  
 SPList parentList = null;  
 SPDocumentLibrary docLib = null;  
 Guid parentListId = targetFolder.ParentListId;  
  
 if (parentListId != Guid.Empty)  
 {  
 parentList = web.Lists[parentListId];  
 if (targetFolder.ContainingDocumentLibrary != Guid.Empty &&  
 targetFolder.ContainingDocumentLibrary != parentListId)  
 {  
 docLib = web.Lists[targetFolder.ContainingDocumentLibrary  
 as SPDocumentLibrary;  
 }  
 }  
  
 Console.WriteLine("Transferring files to folder '{0}' in site '{1}'",   
 targetFolder.Name, web.Title);  
  
 foreach (FileInfo fileInfo in files)  
 {  
 bool okToCopy = true;  
  
 // Check if the file already exists in the target folder.  
 SPFile targetFile = FindFile(fileInfo, targetFolder);  
 if (targetFile != null)  
 {  
 if (this.Overwrite)  
 targetFile.Delete();  
 else  
 okToCopy = false;  
 }  
  
 // Get the file bits and add them to the target folder.  
 if (okToCopy)  
 {  
 nCopied++;  
 using (FileStream fs = fileInfo.Open(FileMode.Open))  
 targetFolder.Files.Add(fileInfo.Name, fs);  
 }  
 else  
 {  
 nSkipped++;  
 }  
 }  
  
 if (nCopied > 0)  
 Console.WriteLine("{0} files transferred.", nCopied);  
 else  
 Console.WriteLine("No files transferred.");  
  
 if (nSkipped > 0)  
 Console.WriteLine("{0} files skipped.", nSkipped);  
}

# Exercise 5: Test Your Work

1. To test your work, open a **Windows Powershell** command window. While building PowerShell cmdlets, it is most convenient to run the PowerShell command window from within the Visual Studio debugger. To set this up, change the **Start Action** setting on the **Debug** page of the project properties to the following command:

c:\windows\system32\windowspowershell\v1.0\powershell.exe

1. In order for **Windows PowerShell** to recognize your snap-in, it must be registered on the system. The best way to do that is to use the **InstallUtil** utility. You can run **InstallUtil** from the post-build events in the project. Open the project property pages and navigate to the **Build Events** tab. Enter the following command into the **Post-build event command line** section.

InstallUtil "$(TargetPath)"

1. Each time PowerShell runs, you will need to add the SPPowerShell snapin so that your commands are recognized. Type the following command at the PowerShell command prompt.

add-pssnapin SPPowerShell

1. To verify that the snapin was loaded successfully, enter the following command into the PowerShell command line.

get-command spimport\*

1. You should see your custom commands listed on the display.
2. Now you can enter a command to import a file into the SharePoint environment. Navigate to the **student/resources/documents** folder and enter the following command:

SPImport-Document Newsletter.doc "http://litwareinc.com"

# Exercise 6: Create the Import Folder CmdLet

1. This cmdlet provides a more powerful and comprehensive way to move documents into a SharePoint site collection by letting the user add metadata to each document as it is being transferred. The source documents are first arranged into physical folders on the file system, and a manifest file is optionally added to a given folder to specify the metadata and other rules that should apply to the documents in that folder. Rules in the manifest control whether nested folders inherit or override the settings in the parent folder. This is similar to the way that ASP.NET handles web.config files.
2. Add a new class and source file to the project named **SPImport-Folder.cs** , and insert the following code.

[ Cmdlet("SPImport","Folder",SupportsShouldProcess=true) ]  
public class SPImport\_Folder : Cmdlet  
{  
 #region Parameters  
 /// <summary>  
 /// This parameter accepts a wildcard file mask that will be expanded  
 /// to a list of folders to be imported to the SharePoint site.  
 ///  
 </summary>  
 [Parameter(Position = 0,  
 Mandatory = true,  
 ValueFromPipeline = true,  
 ValueFromPipelineByPropertyName = true,  
 HelpMessage = "Enter the fully qualified path to the folder you wish to import. You can enter a standard wildcard mask.")]  
  
 [ValidateNotNullOrEmpty]  
 [Alias("Folder", "Path", "SourceFolder")]  
 public string Source { get; set; }  
  
 /// <summary>  
 /// This parameter specifies the url of the SharePoint site to which the  
 /// documents will be imported. If no manifest is found or no document library is specified  
 /// in the manifest, then the files are imported to a new library called "Imported Documents".  
 /// If the specified site does not exist, an InvalidArgument exception is thrown.  
 ///  
 </summary>  
 [Parameter(Position = 1,  
 Mandatory = true,  
 ValueFromPipeline = true,  
 ValueFromPipelineByPropertyName = true,  
 HelpMessage = "Enter the url of the target SharePoint site.")]  
 [Alias("Site", "Url")]  
 public string SiteUrl { get; set; }  
  
 /// <summary>  
 /// This parameter controls whether existing files with the same name are overwritten.  
 ///  
 </summary>  
 [Parameter(  
 ValueFromPipelineByPropertyName = true,  
 HelpMessage = "Specify whether to overwrite files that already exist in the target library or folder. Default=false.")]  
 public SwitchParameter Overwrite { get; set; }  
  
 /// <summary>  
 /// This parameter controls whether subfolders are processed.  
 ///  
 </summary>  
 [Parameter(  
 ValueFromPipelineByPropertyName=true,  
 HelpMessage = "Specify whether to process subfolders of the source folder.")]  
 public SwitchParameter Recursive { get; set; }  
 #endregion  
}

1. Next, replace the generated **ProcessRecord** method with the following code snippet.

Code Snippet: 'PowerShell - Import Folder'

/// <summary>  
/// This is the standard entry point for PowerShell commands.   
/// It is called after the command line arguments have been parsed   
/// and set into the corresponding properties of the cmdlet class.  
/// </summary>  
protected override void ProcessRecord()  
{  
 try  
 {  
 // Reduce the source path to a DirectoryInfo object.  
 if (!this.Source.EndsWith("\\"))  
 this.Source += "\\";  
  
 string folderName = Path.GetDirectoryName(Source);  
 if (string.IsNullOrEmpty(folderName))  
 folderName = Environment.CurrentDirectory;  
 DirectoryInfo folder = new DirectoryInfo(  
 Path.GetFullPath(folderName));  
  
 // Locate the target SharePoint site collection.  
 using (SPSite site = new SPSite(SiteUrl))  
 using (SPWeb web = site.OpenWeb())  
 {  
 // Process the files in the folder   
 // and its subfolders.  
 Utilities.TransferFolderUsingManifest(  
 web, folder, Overwrite, Recursive);  
 }  
 }  
 catch (Exception x)  
 {  
 Console.WriteLine("Unable to process files: ");  
 Console.WriteLine(x.Message);  
 }  
}

1. To process the folder, you will need a new helper method that understands how to process the manifest file. The manifest file is an XML file based on a schema you will create in the next exercise that describes the different elements that can be processed.
2. Open the **helpers.cs** file and add the following code snippet.

Code Snippet: 'PowerShell - TransferFolderUsingManifest'

/// <summary>  
/// This method transfers all files in a given folder   
/// using an optional manifest file located within the folder itself.   
/// The manifest file name must be "\_manifest.xml". This file is  
/// deserialized into an ImportManifest object, which is then   
/// used to perform the actual transfer.  
/// </summary>  
/// <param name="web">the target website into   
/// which files will be transferred</param>  
/// <param name="folder">the source folder from   
/// which files will be transferred</param>  
/// <param name="overwrite">whether to overwrite   
/// existing files in the target library</param>  
/// <param name="recursive">whether to process subfolders</param>  
internal static void TransferFolderUsingManifest(SPWeb web,   
 DirectoryInfo folder, bool overwrite, bool recursive)  
{  
 // create a default manifest  
 ImportManifest manifest = new ImportManifest();  
 FileInfo[] files = folder.GetFiles("\_manifest.xml",   
 SearchOption.TopDirectoryOnly);  
  
 // search for the manifest file  
 if (files.Length > 0)  
 try  
 {  
 manifest = ImportManifest.Load(files[0].FullName);  
 }  
 catch (Exception x)  
 {  
 throw new Exception(  
 string.Format("Failed to load manifest file: {0}",   
 files[0].FullName), x);  
 }  
  
 // process the manifest  
 if (manifest != null)  
 try  
 {  
 manifest.ImportFiles(web, folder, overwrite);  
 }  
 catch (Exception x2)  
 {  
 throw new Exception(  
 string.Format("Failed to import files in folder: {0}",   
 folder.FullName), x2);  
 }  
  
 // process the subfolders  
 if (recursive)  
 {  
 foreach (DirectoryInfo subfolder in folder.GetDirectories())  
 TransferFolderUsingManifest(web, subfolder, overwrite, recursive);  
 }  
}

# Exercise 7: Create the Manifest XML Schema

1. Your **SPImport-Folder** cmdlet will optionally process a manifest file that can be placed in a given folder. You will use a schema to define the structure of this file and also to generate the code needed to process its elements.
2. Add a new **XML Schema** item to the project with the name **ImportManifest.xsd** Open the file for editing and replace the entire contents with the following code snippet.

XML Snippet: 'PowerShell - ImportManifest Schema'

<?xml version="1.0" encoding="utf-8"?>  
<xs:schema id="SPImportManifest"  
 targetNamespace="http://schemas.johnholliday.net/sharepoint/importmanifest.xsd"  
 elementFormDefault="qualified"  
 xmlns="http://schemas.johnholliday.net/sharepoint/importmanifest.xsd"  
 xmlns:mstns="http://schemas.johnholliday.net/sharepoint/importmanifest.xsd"  
 xmlns:xs="http://www.w3.org/2001/XMLSchema"  
>  
 <xs:element name="ImportManifest">  
 <xs:complexType>  
 <xs:sequence>  
 <xs:element name="Library" type="LibrarySpecification" minOccurs="0" maxOccurs="unbounded"/>  
 </xs:sequence>  
 </xs:complexType>  
 </xs:element>  
  
 <xs:complexType name="LibrarySpecification">  
 <xs:sequence>  
 <xs:element name="Columns" type="ColumnSet" minOccurs="1" maxOccurs="1"/>  
 </xs:sequence>  
 <xs:attribute name="Path" type="xs:string" use="required"/>  
 <xs:attribute name="RootWeb" type="xs:boolean" use="optional"/>  
 <xs:attribute name="InheritMetadata" type="xs:boolean" />  
 <xs:attribute name="AutoCreate" type="xs:boolean" use="optional" default="true"/>  
 <xs:attribute name="FileMask" type="xs:string" use="optional"/>  
 <xs:attribute name="Exclude" type="xs:string" use="optional"/>  
 </xs:complexType>  
  
 <xs:complexType name="ColumnSet">  
 <xs:sequence>  
 <xs:element name="Column" type="ColumnSpecification" minOccurs="0" maxOccurs="unbounded"/>  
 </xs:sequence>  
 </xs:complexType>  
  
 <xs:complexType name="ColumnSpecification">  
 <xs:attribute name="DocProperty" type="xs:string" use="required"/>  
 <xs:attribute name="DisplayName" type="xs:string" use="optional"/>  
 <xs:attribute name="Required" type="xs:boolean" use="optional"/>  
 <xs:attribute name="Type" type="xs:string" use="optional"/>  
 </xs:complexType>  
  
 <xs:complexType name="DocPropertyElement">  
 <xs:simpleContent>  
 <xs:extension base="xs:string">  
 </xs:extension>  
 </xs:simpleContent>  
 </xs:complexType>  
</xs:schema>

1. In order to simplify the parsing of manifest files, you will generate wrapper classes from the schema using the XSD.EXE tool. Select the **ImportManifest.xsd** file in the Visual Studio **Solution Explorer** window. From the **Properties** window, set the **Custom Tool** and **Custom Tool Namespace** values to match the following diagram.

Note: The **XsdClassGenerator** assembly is locted in your **Student/Resources/Tools** folder along with a batch file you can run to install the tool into your **Visual Studio** environment.

1. Next, you will extend the generated partial class with a static factory method to make it easy to load manifest files. Add a new class to the project named **ImportManifestEx** . Open the file for editing and add the following code snippet.

Code Snippet: 'PowerShell Import Manifest Extensions'

/// <summary>  
/// Extends the generated ImportManifest class to support file transfer  
/// operations into SharePoint sites.  
/// </summary>  
public partial class ImportManifest  
{  
 /// <summary>  
 /// Loads a manifest from a specified file.  
 /// </summary>  
 /// <param name="fileName"></param>  
 /// <returns></returns>  
 public static ImportManifest Load(string fileName)  
 {  
 ImportManifest manifest = null;  
 const string ns = "http://schemas.johnholliday.net/sharepoint/importmanifest.xsd";  
 XmlSerializer ser = new XmlSerializer(typeof(ImportManifest), ns);  
 using (FileStream fs = new FileStream(fileName, FileMode.Open))  
 {  
 manifest = ser.Deserialize(fs) as ImportManifest;  
 }  
 return manifest;  
 }  
  
 /// <summary>  
 /// Processes all files in the specified folder according to the rules  
 /// specified in this manifest.  
 /// </summary>  
 /// <param name="web"></param>  
 /// <param name="folder"></param>  
 public void ImportFiles(SPWeb web, DirectoryInfo folder, bool overwrite)  
 {  
 // The manifest contains one or more <Library> nodes that target a specific  
 // library. This means that the same files may be imported more than once.  
 foreach (LibrarySpecification library in this.Library)  
 library.ImportFiles(web, folder, overwrite);  
 }  
}

1. Finally, you will extend the generated **LibrarySpecification** and **ColumnSpecification** objects so that they understand how to transfer files into SharePoint and convert column specifications into SharePoint field references.
2. Add a new class to the project named **LibrarySpecification** . Open the **LibrarySpecification.cs** source file and replace the class declaration with the following code.

Code Snippet: 'PowerShell Library Specification'

/// <summary>  
/// Extends the generated LibrarySpecification class to support importing of files  
/// into SharePoint.  
/// </summary>  
public partial class LibrarySpecification  
{  
 public void ImportFiles(SPWeb web, DirectoryInfo folder, bool overwrite)  
 {  
 int nCopied = 0;  
 int nSkipped = 0;  
  
 // determine the target web.  
 SPWeb targetWeb = this.RootWeb ? web.Site.RootWeb : web;  
  
 // find or create the target folder within SharePoint  
 SPFolder targetFolder = Utilities.FindFolder(targetWeb, this.Path, this.AutoCreate);  
  
 if (targetFolder == null)  
 {  
 Console.WriteLine("Cannot find target folder at '{0}'", this.Path);  
 return;  
 }  
  
 if (targetFolder != null)  
 {  
 Console.WriteLine("Importing files from '{0}' => '{1}'", folder.FullName, targetFolder.Url);  
  
 // get the list of files matching the file mask(s)  
 Dictionary<string, FileInfo> files = new Dictionary<string, FileInfo>();  
 string maskSpec = string.IsNullOrEmpty(this.FileMask) ? "\*.\*" : this.FileMask;  
 string[] masks = maskSpec.Split(';');  
 foreach (string mask in masks)  
 {  
 FileInfo[] matchingFiles = folder.GetFiles(mask);  
  
 // add them to the list to be processed  
 foreach (FileInfo match in matchingFiles)  
 files.Add(match.Name, match);  
 }  
  
 // remove the list of excluded files  
 if (!string.IsNullOrEmpty(this.Exclude))  
 {  
 string[] excludePaths = this.Exclude.Split(';');  
 foreach (string exclusion in excludePaths)  
 {  
 FileInfo[] excludedFiles = folder.GetFiles(exclusion);  
 foreach (FileInfo exclude in excludedFiles)  
 if (files.ContainsKey(exclude.Name))  
 files.Remove(exclude.Name);  
 }  
 }  
  
 // process each file...  
 foreach (FileInfo file in files.Values)  
 {  
 bool okToCopy = true;  
 if (file.Name.Equals("\_manifest.xml"))  
 continue;  
  
 // check if the file already exists in the target folder  
 SPFile targetFile = Utilities.FindFile(file, targetFolder);  
 if (targetFile != null)  
 {  
 if (overwrite)  
 targetFile.Delete();  
 else  
 okToCopy = false;  
 }  
  
 if (!okToCopy)  
 nSkipped++;  
  
 if (okToCopy)  
 {  
 try  
 {  
 // if there are any <Column> nodes, map them to columns  
 // in the target document library so that SharePoint will promote  
 // any matching properties in the document.  
 foreach (ColumnSpecification colSpec in this.Columns)  
 colSpec.MapToLibrary(targetFolder);  
  
 // Get the file bits and add them to the target folder.  
 using (FileStream fs = file.Open(FileMode.Open))  
 targetFolder.Files.Add(file.Name, fs);  
 }  
 catch (Exception x)  
 {  
 Console.WriteLine("Failed to transfer file: '{0}' to '{1}", file.Name, targetFolder.Name);  
 Console.WriteLine("Exception occurred: {0}", x.ToString());  
 }  
 nCopied++;  
 }  
 }  
  
 if (nCopied > 0)  
 Console.WriteLine("{0} files transferred.", nCopied);  
 else  
 Console.WriteLine("No files transffered.");  
  
 if (nSkipped > 0)  
 Console.WriteLine("{0} files skipped.", nSkipped);  
 }  
 }  
}

1. Add a second class to the project named **ColumnSpecification** , open the code file and replace the class declaration with the following code.

Code Snippet: 'PowerShell - Column Specification'

/// <summary>  
/// Describes an individual column specification.  
/// </summary>  
public partial class ColumnSpecification  
{  
 /// <summary>  
 /// This method ensures that a column exists in the containing library of the  
 /// specified folder that matches a property by name in the document.  
 /// </summary>  
 /// <param name="targetFolder"></param>  
 public void MapToLibrary(SPFolder targetFolder)  
 {  
 SPList parentList = null;  
 Guid parentListId = targetFolder.ContainingDocumentLibrary;  
  
 if (parentListId == Guid.Empty)  
 parentListId = targetFolder.ParentListId;  
  
 if (parentListId != Guid.Empty)  
 parentList = targetFolder.ParentWeb.Lists[parentListId];  
  
 if (parentList != null)  
 {  
 SPField matchingField = null;  
 SPFieldType fieldType = GetFieldType(this.Type);  
 string fieldName = string.IsNullOrEmpty(this.DisplayName) ? this.DocProperty : this.DisplayName;  
  
 try  
 {  
 matchingField = parentList.Fields.GetField(fieldName);  
 }  
 catch { }  
 if (matchingField == null)  
 try  
 {  
 matchingField = parentList.Fields.GetFieldByInternalName(fieldName);  
 }  
 catch { }  
 if (matchingField == null)  
 {  
 // field was not found, so add a new column to the list  
 parentList.Fields.Add(fieldName, fieldType, this.RequiredSpecified ? this.Required : false);  
 parentList.Update();  
 }  
 }  
 }  
  
 /// <summary>  
 /// Determines the target field type based on a string.  
 /// </summary>  
 /// <param name="fieldType"></param>  
 /// <returns></returns>  
 SPFieldType GetFieldType(string fieldType)  
 {  
 SPFieldType type = SPFieldType.Text;  
 try  
 {  
 type = (SPFieldType)Enum.Parse(typeof(SPField), fieldType);  
 }  
 catch  
 {  
 }  
 return type;  
 }  
}

1. Now your helper methods can call the wrapper classes to interpret the manifest file when processing a folder.

# Exercise 8: (BONUS) Convert OLE Properties into SharePoint Metadata

1. When transferring documents from the file system into SharePoint, it is useful to have a way to automatically extract OLE properties and promote them to columns in the target list. This happens automatically for the custom properties, but not for the standard properties that most users are accustomed to.
2. If you would like to extend your cmdlet to handle the summary properties, you can use the following code to extract them.

Note: This code is a derivative work based on an article by Marcus Peters from the book "Inside Windows" published by Apress, May 4, 2008

Code Snippet: 'PowerShell - GetOleProperties'

namespace ECM401  
{  
 using DSOFile;  
  
 /// <summary>  
 /// Extract the OLE properties from a file using the DSOFile assembly.  
 /// </summary>  
 /// <remarks>  
 internal static StringDictionary GetOLEProperties(string fileName)  
 {  
 StringDictionary dictionary = new StringDictionary();  
 OleDocumentPropertiesClass oleDocument = null;  
 try  
 {  
 oleDocument = new OleDocumentPropertiesClass();  
 oleDocument.Open(fileName, true,   
 dsoFileOpenOptions.dsoOptionOpenReadOnlyIfNoWriteAccess);  
  
 // get the summary properties  
 SummaryProperties properties = oleDocument.SummaryProperties;  
 foreach (PropertyInfo property in   
 typeof(SummaryProperties).GetProperties(  
 BindingFlags.Public | BindingFlags.Instance))  
 {  
 try  
 {  
 object value = property.GetValue(properties, null);  
 if (value != null)  
 dictionary.Add(property.Name, value.ToString());  
 }  
 catch (Exception x)  
 {  
 Debug.WriteLine(x.ToString());  
 continue;  
 }  
 }  
  
 // get the custom properties  
 CustomProperties customProperties =   
 oleDocument.CustomProperties;  
  
 foreach (CustomProperty customProperty in customProperties)  
 {  
 string propertyValue = customProperty.get\_Value().ToString();  
 if (!dictionary.ContainsKey(customProperty.Name))  
 dictionary.Add(customProperty.Name, propertyValue);  
 else  
 {  
 string name = string.Format(  
 "Custom({0})", customProperty.Name);  
 dictionary.Add(name, propertyValue);  
 }  
 }  
 }  
 finally  
 {  
 // close the document  
 oleDocument.Close(false);  
 // release the unmanaged resource  
 Marshal.ReleaseComObject(oleDocument);  
  
 }  
 return dictionary;  
 }  
}

**This concludes the lab exercises.**