Contents

File System and the Registry (C# Programming Guide)

12/11/2018 • 2 minutes to read • Edit Online

The following topics show how to use C# and the .NET Framework to perform various basic operations on files, folders, and the Registry.

In This Section

TITLE	DESCRIPTION
How to: Iterate Through a Directory Tree	Shows how to manually iterate through a directory tree.
How to: Get Information About Files, Folders, and Drives	Shows how to retrieve information such as creation times and size, about files, folders and drives.
How to: Create a File or Folder	Shows how to create a new file or folder.
How to: Copy, Delete, and Move Files and Folders (C# Programming Guide)	Shows how to copy, delete and move files and folders.
How to: Provide a Progress Dialog Box for File Operations	Shows how to display a standard Windows progress dialog for certain file operations.
How to: Write to a Text File	Shows how to write to a text file.
How to: Read From a Text File	Shows how to read from a text file.
How to: Read a Text File One Line at a Time	Shows how to retrieve text from a file one line at a time.
How to: Create a Key In the Registry	Shows how to write a key to the system registry.

Related Sections

File and Stream I/O

How to: Copy, Delete, and Move Files and Folders (C# Programming Guide)

C# Programming Guide

Files, Folders and Drives

System.IO

How to: Iterate Through a Directory Tree (C# Programming Guide)

1/23/2019 • 7 minutes to read • Edit Online

The phrase "iterate a directory tree" means to access each file in each nested subdirectory under a specified root folder, to any depth. You do not necessarily have to open each file. You can just retrieve the name of the file or subdirectory as a `string`, or you can retrieve additional information in the form of a System.IO.FileInfo or System.IO.DirectoryInfo object.

> **NOTE**
> In Windows, the terms "directory" and "folder" are used interchangeably. Most documentation and user interface text uses the term "folder," but the .NET Framework class library uses the term "directory."

In the simplest case, in which you know for certain that you have access permissions for all directories under a specified root, you can use the `System.IO.SearchOption.AllDirectories` flag. This flag returns all the nested subdirectories that match the specified pattern. The following example shows how to use this flag.

```
root.GetDirectories("*.*", System.IO.SearchOption.AllDirectories);
```

The weakness in this approach is that if any one of the subdirectories under the specified root causes a DirectoryNotFoundException or UnauthorizedAccessException, the whole method fails and returns no directories. The same is true when you use the GetFiles method. If you have to handle these exceptions on specific subfolders, you must manually walk the directory tree, as shown in the following examples.

When you manually walk a directory tree, you can handle the subdirectories first (*pre-order traversal*), or the files first (*post-order traversal*). If you perform a pre-order traversal, you walk the whole tree under the current folder before iterating through the files that are directly in that folder itself. The examples later in this document perform post-order traversal, but you can easily modify them to perform pre-order traversal.

Another option is whether to use recursion or a stack-based traversal. The examples later in this document show both approaches.

If you have to perform a variety of operations on files and folders, you can modularize these examples by refactoring the operation into separate functions that you can invoke by using a single delegate.

> **NOTE**
> NTFS file systems can contain *reparse points* in the form of *junction points*, *symbolic links*, and *hard links*. The .NET Framework methods such as GetFiles and GetDirectories will not return any subdirectories under a reparse point. This behavior guards against the risk of entering into an infinite loop when two reparse points refer to each other. In general, you should use extreme caution when you deal with reparse points to ensure that you do not unintentionally modify or delete files. If you require precise control over reparse points, use platform invoke or native code to call the appropriate Win32 file system methods directly.

Example

The following example shows how to walk a directory tree by using recursion. The recursive approach is elegant

but has the potential to cause a stack overflow exception if the directory tree is large and deeply nested.

The particular exceptions that are handled, and the particular actions that are performed on each file or folder, are provided as examples only. You should modify this code to meet your specific requirements. See the comments in the code for more information.

```
public class RecursiveFileSearch
{
    static System.Collections.Specialized.StringCollection log = new
System.Collections.Specialized.StringCollection();

    static void Main()
    {
        // Start with drives if you have to search the entire computer.
        string[] drives = System.Environment.GetLogicalDrives();

        foreach (string dr in drives)
        {
            System.IO.DriveInfo di = new System.IO.DriveInfo(dr);

            // Here we skip the drive if it is not ready to be read. This
            // is not necessarily the appropriate action in all scenarios.
            if (!di.IsReady)
            {
                Console.WriteLine("The drive {0} could not be read", di.Name);
                continue;
            }
            System.IO.DirectoryInfo rootDir = di.RootDirectory;
            WalkDirectoryTree(rootDir);
        }

        // Write out all the files that could not be processed.
        Console.WriteLine("Files with restricted access:");
        foreach (string s in log)
        {
            Console.WriteLine(s);
        }
        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key");
        Console.ReadKey();
    }

    static void WalkDirectoryTree(System.IO.DirectoryInfo root)
    {
        System.IO.FileInfo[] files = null;
        System.IO.DirectoryInfo[] subDirs = null;

        // First, process all the files directly under this folder
        try
        {
            files = root.GetFiles("*.*");
        }
        // This is thrown if even one of the files requires permissions greater
        // than the application provides.
        catch (UnauthorizedAccessException e)
        {
            // This code just writes out the message and continues to recurse.
            // You may decide to do something different here. For example, you
            // can try to elevate your privileges and access the file again.
            log.Add(e.Message);
        }

        catch (System.IO.DirectoryNotFoundException e)
        {
            Console.WriteLine(e.Message);
        }
```

```
        if (files != null)
        {
            foreach (System.IO.FileInfo fi in files)
            {
                // In this example, we only access the existing FileInfo object. If we
                // want to open, delete or modify the file, then
                // a try-catch block is required here to handle the case
                // where the file has been deleted since the call to TraverseTree().
                Console.WriteLine(fi.FullName);
            }

            // Now find all the subdirectories under this directory.
            subDirs = root.GetDirectories();

            foreach (System.IO.DirectoryInfo dirInfo in subDirs)
            {
                // Resursive call for each subdirectory.
                WalkDirectoryTree(dirInfo);
            }
        }
    }
}
```

Example

The following example shows how to iterate through files and folders in a directory tree without using recursion. This technique uses the generic Stack<T> collection type, which is a last in first out (LIFO) stack.

The particular exceptions that are handled, and the particular actions that are performed on each file or folder, are provided as examples only. You should modify this code to meet your specific requirements. See the comments in the code for more information.

```
public class StackBasedIteration
{
    static void Main(string[] args)
    {
        // Specify the starting folder on the command line, or in
        // Visual Studio in the Project > Properties > Debug pane.
        TraverseTree(args[0]);

        Console.WriteLine("Press any key");
        Console.ReadKey();
    }

    public static void TraverseTree(string root)
    {
        // Data structure to hold names of subfolders to be
        // examined for files.
        Stack<string> dirs = new Stack<string>(20);

        if (!System.IO.Directory.Exists(root))
        {
            throw new ArgumentException();
        }
        dirs.Push(root);

        while (dirs.Count > 0)
        {
            string currentDir = dirs.Pop();
            string[] subDirs;
            try
            {
                subDirs = System.IO.Directory.GetDirectories(currentDir);
            }
            // An UnauthorizedAccessException exception will be thrown if we do not have
```

```
            // discovery permission on a folder or file. It may or may not be acceptable
            // to ignore the exception and continue enumerating the remaining files and
            // folders. It is also possible (but unlikely) that a DirectoryNotFound exception
            // will be raised. This will happen if currentDir has been deleted by
            // another application or thread after our call to Directory.Exists. The
            // choice of which exceptions to catch depends entirely on the specific task
            // you are intending to perform and also on how much you know with certainty
            // about the systems on which this code will run.
            catch (UnauthorizedAccessException e)
            {
                Console.WriteLine(e.Message);
                continue;
            }
            catch (System.IO.DirectoryNotFoundException e)
            {
                Console.WriteLine(e.Message);
                continue;
            }

            string[] files = null;
            try
            {
                files = System.IO.Directory.GetFiles(currentDir);
            }

            catch (UnauthorizedAccessException e)
            {

                Console.WriteLine(e.Message);
                continue;
            }

            catch (System.IO.DirectoryNotFoundException e)
            {
                Console.WriteLine(e.Message);
                continue;
            }
            // Perform the required action on each file here.
            // Modify this block to perform your required task.
            foreach (string file in files)
            {
                try
                {
                    // Perform whatever action is required in your scenario.
                    System.IO.FileInfo fi = new System.IO.FileInfo(file);
                    Console.WriteLine("{0}: {1}, {2}", fi.Name, fi.Length, fi.CreationTime);
                }
                catch (System.IO.FileNotFoundException e)
                {
                    // If file was deleted by a separate application
                    //  or thread since the call to TraverseTree()
                    // then just continue.
                    Console.WriteLine(e.Message);
                    continue;
                }
            }

            // Push the subdirectories onto the stack for traversal.
            // This could also be done before handing the files.
            foreach (string str in subDirs)
                dirs.Push(str);
        }
    }
}
```

It is generally too time-consuming to test every folder to determine whether your application has permission to open it. Therefore, the code example just encloses that part of the operation in a `try/catch` block. You can modify

the `catch` block so that when you are denied access to a folder, you try to elevate your permissions and then access it again. As a rule, only catch those exceptions that you can handle without leaving your application in an unknown state.

If you must store the contents of a directory tree, either in memory or on disk, the best option is to store only the FullName property (of type `string`) for each file. You can then use this string to create a new FileInfo or DirectoryInfo object as necessary, or open any file that requires additional processing.

Robust Programming

Robust file iteration code must take into account many complexities of the file system. For more information on the Windows file system, see NTFS overview.

See also

- System.IO
- LINQ and File Directories
- File System and the Registry (C# Programming Guide)

How to: Get Information About Files, Folders, and Drives (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

In the .NET Framework, you can access file system information by using the following classes:

- System.IO.FileInfo
- System.IO.DirectoryInfo
- System.IO.DriveInfo
- System.IO.Directory
- System.IO.File

The FileInfo and DirectoryInfo classes represent a file or directory and contain properties that expose many of the file attributes that are supported by the NTFS file system. They also contain methods for opening, closing, moving, and deleting files and folders. You can create instances of these classes by passing a string that represents the name of the file, folder, or drive in to the constructor:

```
System.IO.DriveInfo di = new System.IO.DriveInfo(@"C:\");
```

You can also obtain the names of files, folders, or drives by using calls to DirectoryInfo.GetDirectories, DirectoryInfo.GetFiles, and DriveInfo.RootDirectory.

The System.IO.Directory and System.IO.File classes provide static methods for retrieving information about directories and files.

Example

The following example shows various ways to access information about files and folders.

```
class FileSysInfo
{
    static void Main()
    {
        // You can also use System.Environment.GetLogicalDrives to
        // obtain names of all logical drives on the computer.
        System.IO.DriveInfo di = new System.IO.DriveInfo(@"C:\");
        Console.WriteLine(di.TotalFreeSpace);
        Console.WriteLine(di.VolumeLabel);

        // Get the root directory and print out some information about it.
        System.IO.DirectoryInfo dirInfo = di.RootDirectory;
        Console.WriteLine(dirInfo.Attributes.ToString());

        // Get the files in the directory and print out some information about them.
        System.IO.FileInfo[] fileNames = dirInfo.GetFiles("*.*");


        foreach (System.IO.FileInfo fi in fileNames)
        {
            Console.WriteLine("{0}: {1}: {2}", fi.Name, fi.LastAccessTime, fi.Length);
        }
```

```
        // Get the subdirectories directly that is under the root.
        // See "How to: Iterate Through a Directory Tree" for an example of how to
        // iterate through an entire tree.
        System.IO.DirectoryInfo[] dirInfos = dirInfo.GetDirectories("*.*");

        foreach (System.IO.DirectoryInfo d in dirInfos)
        {
            Console.WriteLine(d.Name);
        }

        // The Directory and File classes provide several static methods
        // for accessing files and directories.

        // Get the current application directory.
        string currentDirName = System.IO.Directory.GetCurrentDirectory();
        Console.WriteLine(currentDirName);

        // Get an array of file names as strings rather than FileInfo objects.
        // Use this method when storage space is an issue, and when you might
        // hold on to the file name reference for a while before you try to access
        // the file.
        string[] files = System.IO.Directory.GetFiles(currentDirName, "*.txt");

        foreach (string s in files)
        {
            // Create the FileInfo object only when needed to ensure
            // the information is as current as possible.
            System.IO.FileInfo fi = null;
            try
            {
                 fi = new System.IO.FileInfo(s);
            }
            catch (System.IO.FileNotFoundException e)
            {
                // To inform the user and continue is
                // sufficient for this demonstration.
                // Your application may require different behavior.
                Console.WriteLine(e.Message);
                continue;
            }
            Console.WriteLine("{0} : {1}",fi.Name, fi.Directory);
        }

        // Change the directory. In this case, first check to see
        // whether it already exists, and create it if it does not.
        // If this is not appropriate for your application, you can
        // handle the System.IO.IOException that will be raised if the
        // directory cannot be found.
        if (!System.IO.Directory.Exists(@"C:\Users\Public\TestFolder\"))
        {
            System.IO.Directory.CreateDirectory(@"C:\Users\Public\TestFolder\");
        }

        System.IO.Directory.SetCurrentDirectory(@"C:\Users\Public\TestFolder\");

        currentDirName = System.IO.Directory.GetCurrentDirectory();
        Console.WriteLine(currentDirName);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

Robust Programming

When you process user-specified path strings, you should also handle exceptions for the following conditions:

- The file name is malformed. For example, it contains invalid characters or only white space.
- The file name is null.
- The file name is longer than the system-defined maximum length.
- The file name contains a colon (:).

If the application does not have sufficient permissions to read the specified file, the `Exists` method returns `false` regardless of whether a path exists; the method does not throw an exception.

See also

- System.IO
- C# Programming Guide
- File System and the Registry (C# Programming Guide)

How to: Create a File or Folder (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

You can programmatically create a folder on your computer, create a subfolder, create a file in the subfolder, and write data to the file.

Example

```
public class CreateFileOrFolder
{
    static void Main()
    {
        // Specify a name for your top-level folder.
        string folderName = @"c:\Top-Level Folder";

        // To create a string that specifies the path to a subfolder under your
        // top-level folder, add a name for the subfolder to folderName.
        string pathString = System.IO.Path.Combine(folderName, "SubFolder");

        // You can write out the path name directly instead of using the Combine
        // method. Combine just makes the process easier.
        string pathString2 = @"c:\Top-Level Folder\SubFolder2";

        // You can extend the depth of your path if you want to.
        //pathString = System.IO.Path.Combine(pathString, "SubSubFolder");

        // Create the subfolder. You can verify in File Explorer that you have this
        // structure in the C: drive.
        //    Local Disk (C:)
        //        Top-Level Folder
        //            SubFolder
        System.IO.Directory.CreateDirectory(pathString);

        // Create a file name for the file you want to create.
        string fileName = System.IO.Path.GetRandomFileName();

        // This example uses a random string for the name, but you also can specify
        // a particular name.
        //string fileName = "MyNewFile.txt";

        // Use Combine again to add the file name to the path.
        pathString = System.IO.Path.Combine(pathString, fileName);

        // Verify the path that you have constructed.
        Console.WriteLine("Path to my file: {0}\n", pathString);

        // Check that the file doesn't already exist. If it doesn't exist, create
        // the file and write integers 0 - 99 to it.
        // DANGER: System.IO.File.Create will overwrite the file if it already exists.
        // This could happen even with random file names, although it is unlikely.
        if (!System.IO.File.Exists(pathString))
        {
            using (System.IO.FileStream fs = System.IO.File.Create(pathString))
            {
                for (byte i = 0; i < 100; i++)
                {
                    fs.WriteByte(i);
                }
            }
```

```
            }
        }
        else
        {
            Console.WriteLine("File \"{0}\" already exists.", fileName);
            return;
        }

        // Read and display the data from your file.
        try
        {
            byte[] readBuffer = System.IO.File.ReadAllBytes(pathString);
            foreach (byte b in readBuffer)
            {
                Console.Write(b + " ");
            }
            Console.WriteLine();
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
    // Sample output:

    // Path to my file: c:\Top-Level Folder\SubFolder\ttxvauxe.vv0

    //0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
    //30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
    // 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 8
    //3 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
}
```

If the folder already exists, CreateDirectory does nothing, and no exception is thrown. However, File.Create replaces an existing file with a new file. The example uses an `if` - `else` statement to prevent an existing file from being replaced.

By making the following changes in the example, you can specify different outcomes based on whether a file with a certain name already exists. If such a file doesn't exist, the code creates one. If such a file exists, the code appends data to that file.

- Specify a non-random file name.

```
// Comment out the following line.
//string fileName = System.IO.Path.GetRandomFileName();

// Replace that line with the following assignment.
string fileName = "MyNewFile.txt";
```

- Replace the `if` - `else` statement with the `using` statement in the following code.

```
using (System.IO.FileStream fs = new System.IO.FileStream(pathString, FileMode.Append))
{
    for (byte i = 0; i < 100; i++)
    {
        fs.WriteByte(i);
    }
}
```

Run the example several times to verify that data is added to the file each time.

For more `FileMode` values that you can try, see FileMode.

The following conditions may cause an exception:

- The folder name is malformed. For example, it contains illegal characters or is only white space (ArgumentException class). Use the Path class to create valid path names.
- The parent folder of the folder to be created is read-only (IOException class).
- The folder name is `null` (ArgumentNullException class).
- The folder name is too long (PathTooLongException class).
- The folder name is only a colon, ":" (PathTooLongException class).

.NET Framework Security

An instance of the SecurityException class may be thrown in partial-trust situations.

If you don't have permission to create the folder, the example throws an instance of the UnauthorizedAccessException class.

See also

- System.IO
- C# Programming Guide
- File System and the Registry (C# Programming Guide)

How to: Copy, Delete, and Move Files and Folders (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

The following examples show how to copy, move, and delete files and folders in a synchronous manner by using the System.IO.File, System.IO.Directory, System.IO.FileInfo, and System.IO.DirectoryInfo classes from the System.IO namespace. These examples do not provide a progress bar or any other user interface. If you want to provide a standard progress dialog box, see How to: Provide a Progress Dialog Box for File Operations.

Use System.IO.FileSystemWatcher to provide events that will enable you to calculate the progress when operating on multiple files. Another approach is to use platform invoke to call the relevant file-related methods in the Windows Shell. For information about how to perform these file operations asynchronously, see Asynchronous File I/O.

Example

The following example shows how to copy files and directories.

```
// Simple synchronous file copy operations with no user interface.
// To run this sample, first create the following directories and files:
// C:\Users\Public\TestFolder
// C:\Users\Public\TestFolder\test.txt
// C:\Users\Public\TestFolder\SubDir\test.txt
public class SimpleFileCopy
{
    static void Main()
    {
        string fileName = "test.txt";
        string sourcePath = @"C:\Users\Public\TestFolder";
        string targetPath =  @"C:\Users\Public\TestFolder\SubDir";

        // Use Path class to manipulate file and directory paths.
        string sourceFile = System.IO.Path.Combine(sourcePath, fileName);
        string destFile = System.IO.Path.Combine(targetPath, fileName);

        // To copy a folder's contents to a new location:
        // Create a new target folder, if necessary.
        if (!System.IO.Directory.Exists(targetPath))
        {
            System.IO.Directory.CreateDirectory(targetPath);
        }

        // To copy a file to another location and
        // overwrite the destination file if it already exists.
        System.IO.File.Copy(sourceFile, destFile, true);

        // To copy all the files in one directory to another directory.
        // Get the files in the source folder. (To recursively iterate through
        // all subfolders under the current directory, see
        // "How to: Iterate Through a Directory Tree.")
        // Note: Check for target path was performed previously
        //       in this code example.
        if (System.IO.Directory.Exists(sourcePath))
        {
            string[] files = System.IO.Directory.GetFiles(sourcePath);

            // Copy the files and overwrite destination files if they already exist.
            foreach (string s in files)
            {
                // Use static Path methods to extract only the file name from the path.
                fileName = System.IO.Path.GetFileName(s);
                destFile = System.IO.Path.Combine(targetPath, fileName);
                System.IO.File.Copy(s, destFile, true);
            }
        }
        else
        {
            Console.WriteLine("Source path does not exist!");
        }

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

Example

The following example shows how to move files and directories.

```
// Simple synchronous file move operations with no user interface.
public class SimpleFileMove
{
    static void Main()
    {
        string sourceFile = @"C:\Users\Public\public\test.txt";
        string destinationFile = @"C:\Users\Public\private\test.txt";

        // To move a file or folder to a new location:
        System.IO.File.Move(sourceFile, destinationFile);

        // To move an entire directory. To programmatically modify or combine
        // path strings, use the System.IO.Path class.
        System.IO.Directory.Move(@"C:\Users\Public\public\test\", @"C:\Users\Public\private");
    }
}
```

Example

The following example shows how to delete files and directories.

```
// Simple synchronous file deletion operations with no user interface.
// To run this sample, create the following files on your drive:
// C:\Users\Public\DeleteTest\test1.txt
// C:\Users\Public\DeleteTest\test2.txt
// C:\Users\Public\DeleteTest\SubDir\test2.txt

public class SimpleFileDelete
{
    static void Main()
    {
        // Delete a file by using File class static method...
        if(System.IO.File.Exists(@"C:\Users\Public\DeleteTest\test.txt"))
        {
            // Use a try block to catch IOExceptions, to
            // handle the case of the file already being
            // opened by another process.
            try
            {
                System.IO.File.Delete(@"C:\Users\Public\DeleteTest\test.txt");
            }
            catch (System.IO.IOException e)
            {
                Console.WriteLine(e.Message);
                return;
            }
        }

        // ...or by using FileInfo instance method.
        System.IO.FileInfo fi = new System.IO.FileInfo(@"C:\Users\Public\DeleteTest\test2.txt");
        try
        {
            fi.Delete();
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }

        // Delete a directory. Must be writable or empty.
        try
        {
            System.IO.Directory.Delete(@"C:\Users\Public\DeleteTest");
        }
        catch (System.IO.IOException e)
```

```
        {
            Console.WriteLine(e.Message);
        }
        // Delete a directory and all subdirectories with Directory static method...
        if(System.IO.Directory.Exists(@"C:\Users\Public\DeleteTest"))
        {
            try
            {
                System.IO.Directory.Delete(@"C:\Users\Public\DeleteTest", true);
            }

            catch (System.IO.IOException e)
            {
                Console.WriteLine(e.Message);
            }
        }

        // ...or with DirectoryInfo instance method.
        System.IO.DirectoryInfo di = new System.IO.DirectoryInfo(@"C:\Users\Public\public");
        // Delete this dir and all subdirs.
        try
        {
            di.Delete(true);
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }

    }
}
```

See also

- System.IO
- C# Programming Guide
- File System and the Registry (C# Programming Guide)
- How to: Provide a Progress Dialog Box for File Operations
- File and Stream I/O
- Common I/O Tasks

How to: Provide a Progress Dialog Box for File Operations (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

You can provide a standard dialog box that shows progress on file operations in Windows if you use the CopyFile(String, String, UIOption) method in the Microsoft.VisualBasic namespace.

> **NOTE**
>
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

To add a reference in Visual Studio

1. On the menu bar, choose **Project**, **Add Reference**.

   The **Reference Manager** dialog box appears.

2. In the **Assemblies** area, choose **Framework** if it isn't already chosen.

3. In the list of names, select the **Microsoft.VisualBasic** check box, and then choose the **OK** button to close the dialog box.

Example

The following code copies the directory that `sourcePath` specifies into the directory that `destinationPath` specifies. This code also provides a standard dialog box that shows the estimated amount of time remaining before the operation finishes.

```
// The following using directive requires a project reference to Microsoft.VisualBasic.
using Microsoft.VisualBasic.FileIO;

class FileProgress
{
    static void Main()
    {
        // Specify the path to a folder that you want to copy. If the folder is small,
        // you won't have time to see the progress dialog box.
        string sourcePath = @"C:\Windows\symbols\";
        // Choose a destination for the copied files.
        string destinationPath = @"C:\TestFolder";

        FileSystem.CopyDirectory(sourcePath, destinationPath,
            UIOption.AllDialogs);
    }
}
```

See also

- File System and the Registry (C# Programming Guide)

How to: Write to a Text File (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

These examples show various ways to write text to a file. The first two examples use static convenience methods on the System.IO.File class to write each element of any `IEnumerable<string>` and a string to a text file. Example 3 shows how to add text to a file when you have to process each line individually as you write to the file. Examples 1-3 overwrite all existing content in the file, but example 4 shows you how to append text to an existing file.

These examples all write string literals to files. If you want to format text written to a file, use the Format method or C# string interpolation feature.

Example

```
class WriteTextFile
{
    static void Main()
    {

        // These examples assume a "C:\Users\Public\TestFolder" folder on your machine.
        // You can modify the path if necessary.


        // Example #1: Write an array of strings to a file.
        // Create a string array that consists of three lines.
        string[] lines = { "First line", "Second line", "Third line" };
        // WriteAllLines creates a file, writes a collection of strings to the file,
        // and then closes the file.  You do NOT need to call Flush() or Close().
        System.IO.File.WriteAllLines(@"C:\Users\Public\TestFolder\WriteLines.txt", lines);


        // Example #2: Write one string to a text file.
        string text = "A class is the most powerful data type in C#. Like a structure, " +
                      "a class defines the data and behavior of the data type. ";
        // WriteAllText creates a file, writes the specified string to the file,
        // and then closes the file.    You do NOT need to call Flush() or Close().
        System.IO.File.WriteAllText(@"C:\Users\Public\TestFolder\WriteText.txt", text);

        // Example #3: Write only some strings in an array to a file.
        // The using statement automatically flushes AND CLOSES the stream and calls
        // IDisposable.Dispose on the stream object.
        // NOTE: do not use FileStream for text files because it writes bytes, but StreamWriter
        // encodes the output as text.
        using (System.IO.StreamWriter file =
            new System.IO.StreamWriter(@"C:\Users\Public\TestFolder\WriteLines2.txt"))
        {
            foreach (string line in lines)
            {
                // If the line doesn't contain the word 'Second', write the line to the file.
                if (!line.Contains("Second"))
                {
                    file.WriteLine(line);
                }
            }
        }

        // Example #4: Append new text to an existing file.
        // The using statement automatically flushes AND CLOSES the stream and calls
        // IDisposable.Dispose on the stream object.
        using (System.IO.StreamWriter file =
            new System.IO.StreamWriter(@"C:\Users\Public\TestFolder\WriteLines2.txt", true))
```

```
        {
            file.WriteLine("Fourth line");
        }
    }
}
 //Output (to WriteLines.txt):
 //   First line
 //   Second line
 //   Third line

 //Output (to WriteText.txt):
 //   A class is the most powerful data type in C#. Like a structure, a class defines the data and behavior of
the data type.

 //Output to WriteLines2.txt after Example #3:
 //   First line
 //   Third line

 //Output to WriteLines2.txt after Example #4:
 //   First line
 //   Third line
 //   Fourth line
```

Robust Programming

The following conditions may cause an exception:

- The file exists and is read-only.
- The path name may be too long.
- The disk may be full.

See also

- C# Programming Guide
- File System and the Registry (C# Programming Guide)
- Sample: Save a collection to Application Storage

How to: Read From a Text File (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

This example reads the contents of a text file by using the static methods ReadAllText and ReadAllLines from the System.IO.File class.

For an example that uses StreamReader, see How to: Read a Text File One Line at a Time.

> **NOTE**
> The files that are used in this example are created in the topic How to: Write to a Text File.

Example

```
class ReadFromFile
{
    static void Main()
    {
        // The files used in this example are created in the topic
        // How to: Write to a Text File. You can change the path and
        // file name to substitute text files of your own.

        // Example #1
        // Read the file as one string.
        string text = System.IO.File.ReadAllText(@"C:\Users\Public\TestFolder\WriteText.txt");

        // Display the file contents to the console. Variable text is a string.
        System.Console.WriteLine("Contents of WriteText.txt = {0}", text);

        // Example #2
        // Read each line of the file into a string array. Each element
        // of the array is one line of the file.
        string[] lines = System.IO.File.ReadAllLines(@"C:\Users\Public\TestFolder\WriteLines2.txt");

        // Display the file contents by using a foreach loop.
        System.Console.WriteLine("Contents of WriteLines2.txt = ");
        foreach (string line in lines)
        {
            // Use a tab to indent each line of the file.
            Console.WriteLine("\t" + line);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

Compiling the Code

Copy the code and paste it into a C# console application.

If you are not using the text files from How to: Write to a Text File, replace the argument to `ReadAllText` and `ReadAllLines` with the appropriate path and file name on your computer.

Robust Programming

The following conditions may cause an exception:

- The file doesn't exist or doesn't exist at the specified location. Check the path and the spelling of the file name.

.NET Framework Security

Do not rely on the name of a file to determine the contents of the file. For example, the file `myFile.cs` might not be a C# source file.

See also

- System.IO
- C# Programming Guide
- File System and the Registry (C# Programming Guide)

How to: Read a Text File One Line at a Time (Visual C#)

1/23/2019 • 2 minutes to read • Edit Online

This example reads the contents of a text file, one line at a time, into a string using the `ReadLine` method of the `StreamReader` class. Each text line is stored into the string `line` and displayed on the screen.

Example

```
int counter = 0;
string line;

// Read the file and display it line by line.
System.IO.StreamReader file =
    new System.IO.StreamReader(@"c:\test.txt");
while((line = file.ReadLine()) != null)
{
    System.Console.WriteLine(line);
    counter++;
}

file.Close();
System.Console.WriteLine("There were {0} lines.", counter);
// Suspend the screen.
System.Console.ReadLine();
```

Compiling the Code

Copy the code and paste it into the `Main` method of a console application.

Replace `"c:\test.txt"` with the actual file name.

Robust Programming

The following conditions may cause an exception:

- The file may not exist.

.NET Framework Security

Do not make decisions about the contents of the file based on the name of the file. For example, the file `myFile.cs` may not be a C# source file.

See also

- System.IO
- C# Programming Guide
- File System and the Registry (C# Programming Guide)

How to: Create a Key In the Registry (Visual C#)

1/23/2019 • 2 minutes to read • Edit Online

This example adds the value pair, "Name" and "Isabella", to the current user's registry, under the key "Names".

Example

```
Microsoft.Win32.RegistryKey key;
key = Microsoft.Win32.Registry.CurrentUser.CreateSubKey("Names");
key.SetValue("Name", "Isabella");
key.Close();
```

Compiling the Code

- Copy the code and paste it into the `Main` method of a console application.
- Replace the `Names` parameter with the name of a key that exists directly under the HKEY_CURRENT_USER node of the registry.
- Replace the `Name` parameter with the name of a value that exists directly under the Names node.

Robust Programming

Examine the registry structure to find a suitable location for your key. For example, you might want to open the Software key of the current user, and create a key with your company's name. Then add the registry values to your company's key.

The following conditions might cause an exception:

- The name of the key is null.
- The user does not have permissions to create registry keys.
- The key name exceeds the 255-character limit.
- The key is closed.
- The registry key is read-only.

.NET Framework Security

It is more secure to write data to the user folder — `Microsoft.Win32.Registry.CurrentUser` — rather than to the local computer — `Microsoft.Win32.Registry.LocalMachine`.

When you create a registry value, you need to decide what to do if that value already exists. Another process, perhaps a malicious one, may have already created the value and have access to it. When you put data in the registry value, the data is available to the other process. To prevent this, use the. `Overload:Microsoft.Win32.RegistryKey.GetValue` method. It returns null if the key does not already exist.

It is not secure to store secrets, such as passwords, in the registry as plain text, even if the registry key is protected by access control lists (ACL).

See also

- System.IO
- C# Programming Guide
- File System and the Registry (C# Programming Guide)
- Read, write and delete from the registry with C#