Contents

Exceptions and Exception Handling (C# Programming Guide)

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

The C# language's exception handling features help you deal with any unexpected or exceptional situations that occur when a program is running. Exception handling uses the `try`, `catch`, and `finally` keywords to try actions that may not succeed, to handle failures when you decide that it is reasonable to do so, and to clean up resources afterward. Exceptions can be generated by the common language runtime (CLR), by the .NET Framework or any third-party libraries, or by application code. Exceptions are created by using the `throw` keyword.

In many cases, an exception may be thrown not by a method that your code has called directly, but by another method further down in the call stack. When this happens, the CLR will unwind the stack, looking for a method with a `catch` block for the specific exception type, and it will execute the first such `catch` block that if finds. If it finds no appropriate `catch` block anywhere in the call stack, it will terminate the process and display a message to the user.

In this example, a method tests for division by zero and catches the error. Without the exception handling, this program would terminate with a **DivideByZeroException was unhandled** error.

```
class ExceptionTest
{
    static double SafeDivision(double x, double y)
    {
        if (y == 0)
            throw new System.DivideByZeroException();
        return x / y;
    }
    static void Main()
    {
        // Input for test purposes. Change the values to see
        // exception handling behavior.
        double a = 98, b = 0;
        double result = 0;

        try
        {
            result = SafeDivision(a, b);
            Console.WriteLine("{0} divided by {1} = {2}", a, b, result);
        }
        catch (DivideByZeroException e)
        {
            Console.WriteLine("Attempted divide by zero.");
        }
    }
}
```

Exceptions Overview

Exceptions have the following properties:

- Exceptions are types that all ultimately derive from `System.Exception`.
- Use a `try` block around the statements that might throw exceptions.
- Once an exception occurs in the `try` block, the flow of control jumps to the first associated exception

handler that is present anywhere in the call stack. In C#, the `catch` keyword is used to define an exception handler.

- If no exception handler for a given exception is present, the program stops executing with an error message.
- Do not catch an exception unless you can handle it and leave the application in a known state. If you catch `System.Exception`, rethrow it using the `throw` keyword at the end of the `catch` block.
- If a `catch` block defines an exception variable, you can use it to obtain more information about the type of exception that occurred.
- Exceptions can be explicitly generated by a program by using the `throw` keyword.
- Exception objects contain detailed information about the error, such as the state of the call stack and a text description of the error.
- Code in a `finally` block is executed even if an exception is thrown. Use a `finally` block to release resources, for example to close any streams or files that were opened in the `try` block.
- Managed exceptions in the .NET Framework are implemented on top of the Win32 structured exception handling mechanism. For more information, see Structured Exception Handling (C/C++) and A Crash Course on the Depths of Win32 Structured Exception Handling.

Related Sections

See the following topics for more information about exceptions and exception handling:

- Using Exceptions
- Exception Handling
- Creating and Throwing Exceptions
- Compiler-Generated Exceptions
- How to: Handle an Exception Using try/catch (C# Programming Guide)
- How to: Execute Cleanup Code Using finally

C# Language Specification

For more information, see Exceptions in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- SystemException
- C# Programming Guide
- C# Keywords
- throw
- try-catch
- try-finally
- try-catch-finally
- Exceptions

Using Exceptions (C# Programming Guide)

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

In C#, errors in the program at run time are propagated through the program by using a mechanism called exceptions. Exceptions are thrown by code that encounters an error and caught by code that can correct the error. Exceptions can be thrown by the .NET Framework common language runtime (CLR) or by code in a program. Once an exception is thrown, it propagates up the call stack until a `catch` statement for the exception is found. Uncaught exceptions are handled by a generic exception handler provided by the system that displays a dialog box.

Exceptions are represented by classes derived from Exception. This class identifies the type of exception and contains properties that have details about the exception. Throwing an exception involves creating an instance of an exception-derived class, optionally configuring properties of the exception, and then throwing the object by using the `throw` keyword. For example:

```
class CustomException : Exception
{
    public CustomException(string message)
    {

    }

}
private static void TestThrow()
{
    CustomException ex =
        new CustomException("Custom exception in TestThrow()");

    throw ex;
}
```

After an exception is thrown, the runtime checks the current statement to see whether it is within a `try` block. If it is, any `catch` blocks associated with the `try` block are checked to see whether they can catch the exception. `Catch` blocks typically specify exception types; if the type of the `catch` block is the same type as the exception, or a base class of the exception, the `catch` block can handle the method. For example:

```
static void TestCatch()
{
    try
    {
        TestThrow();
    }
    catch (CustomException ex)
    {
        System.Console.WriteLine(ex.ToString());
    }
}
```

If the statement that throws an exception is not within a `try` block or if the `try` block that encloses it has no matching `catch` block, the runtime checks the calling method for a `try` statement and `catch` blocks. The runtime continues up the calling stack, searching for a compatible `catch` block. After the `catch` block is found and executed, control is passed to the next statement after that `catch` block.

A `try` statement can contain more than one `catch` block. The first `catch` statement that can handle the

exception is executed; any following `catch` statements, even if they are compatible, are ignored. Therefore, catch blocks should always be ordered from most specific (or most-derived) to least specific. For example:

```
using System;
using System.IO;

public class ExceptionExample
{
    static void Main()
    {
        try
        {
            using (var sw = new StreamWriter(@"C:\test\test.txt"))
            {
                sw.WriteLine("Hello");
            }
        }
        // Put the more specific exceptions first.
        catch (DirectoryNotFoundException ex)
        {
            Console.WriteLine(ex);
        }
        catch (FileNotFoundException ex)
        {
            Console.WriteLine(ex);
        }
        // Put the least specific exception last.
        catch (IOException ex)
        {
            Console.WriteLine(ex);
        }

        Console.WriteLine("Done");
    }
}
```

Before the `catch` block is executed, the runtime checks for `finally` blocks. `Finally` blocks enable the programmer to clean up any ambiguous state that could be left over from an aborted `try` block, or to release any external resources (such as graphics handles, database connections or file streams) without waiting for the garbage collector in the runtime to finalize the objects. For example:

```
static void TestFinally()
{
    System.IO.FileStream file = null;
    //Change the path to something that works on your machine.
    System.IO.FileInfo fileInfo = new System.IO.FileInfo(@"C:\file.txt");

    try
    {
        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    finally
    {
        // Closing the file allows you to reopen it immediately - otherwise IOException is thrown.
        if (file != null)
        {
            file.Close();
        }
    }

    try
    {
        file = fileInfo.OpenWrite();
        System.Console.WriteLine("OpenWrite() succeeded");
    }
    catch (System.IO.IOException)
    {
        System.Console.WriteLine("OpenWrite() failed");
    }
}
```

If `WriteByte()` threw an exception, the code in the second `try` block that tries to reopen the file would fail if `file.Close()` is not called, and the file would remain locked. Because `finally` blocks are executed even if an exception is thrown, the `finally` block in the previous example allows for the file to be closed correctly and helps avoid an error.

If no compatible `catch` block is found on the call stack after an exception is thrown, one of three things occurs:

- If the exception is within a finalizer, the finalizer is aborted and the base finalizer, if any, is called.
- If the call stack contains a static constructor, or a static field initializer, aTypeInitializationException is thrown, with the original exception assigned to the InnerException property of the new exception.
- If the start of the thread is reached, the thread is terminated.

See also

- C# Programming Guide
- Exceptions and Exception Handling

Exception Handling (C# Programming Guide)

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

A try block is used by C# programmers to partition code that might be affected by an exception. Associated catch blocks are used to handle any resulting exceptions. A finally block contains code that is run regardless of whether or not an exception is thrown in the `try` block, such as releasing resources that are allocated in the `try` block. A `try` block requires one or more associated `catch` blocks, or a `finally` block, or both.

The following examples show a `try-catch` statement, a `try-finally` statement, and a `try-catch-finally` statement.

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
    // Only catch exceptions that you know how to handle.
    // Never catch base class System.Exception without
    // rethrowing it at the end of the catch block.
}
```

```
try
{
    // Code to try goes here.
}
finally
{
    // Code to execute after the try block goes here.
}
```

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
}
finally
{
    // Code to execute after the try (and possibly catch) blocks
    // goes here.
}
```

A `try` block without a `catch` or `finally` block causes a compiler error.

Catch Blocks

A `catch` block can specify the type of exception to catch. The type specification is called an *exception filter*. The exception type should be derived from Exception. In general, do not specify Exception as the exception filter unless either you know how to handle all exceptions that might be thrown in the `try` block, or you have included a throw statement at the end of your `catch` block.

Multiple `catch` blocks with different exception filters can be chained together. The `catch` blocks are evaluated from top to bottom in your code, but only one `catch` block is executed for each exception that is thrown. The first `catch` block that specifies the exact type or a base class of the thrown exception is executed. If no `catch` block specifies a matching exception filter, a `catch` block that does not have a filter is selected, if one is present in the statement. It is important to position `catch` blocks with the most specific (that is, the most derived) exception types first.

You should catch exceptions when the following conditions are true:

- You have a good understanding of why the exception might be thrown, and you can implement a specific recovery, such as prompting the user to enter a new file name when you catch a FileNotFoundException object.
- You can create and throw a new, more specific exception.

```
int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch(System.IndexOutOfRangeException e)
    {
        throw new System.ArgumentOutOfRangeException(
            "Parameter index is out of range.", e);
    }
}
```

- You want to partially handle an exception before passing it on for additional handling. In the following example, a `catch` block is used to add an entry to an error log before re-throwing the exception.

```
try
{
    // Try to access a resource.
}
catch (System.UnauthorizedAccessException e)
{
    // Call a custom error logging procedure.
    LogError(e);
    // Re-throw the error.
    throw;
}
```

Finally Blocks

A `finally` block enables you to clean up actions that are performed in a `try` block. If present, the `finally` block executes last, after the `try` block and any matched `catch` block. A `finally` block always runs, regardless of whether an exception is thrown or a `catch` block matching the exception type is found.

The `finally` block can be used to release resources such as file streams, database connections, and graphics handles without waiting for the garbage collector in the runtime to finalize the objects. See using Statement for more information.

In the following example, the `finally` block is used to close a file that is opened in the `try` block. Notice that the state of the file handle is checked before the file is closed. If the `try` block cannot open the file, the file handle still has the value `null` and the `finally` block does not try to close it. Alternatively, if the file is opened successfully in the `try` block, the `finally` block closes the open file.

```
System.IO.FileStream file = null;
System.IO.FileInfo fileinfo = new System.IO.FileInfo("C:\\file.txt");
try
{
    file = fileinfo.OpenWrite();
    file.WriteByte(0xF);
}
finally
{
    // Check for null because OpenWrite might have failed.
    if (file != null)
    {
        file.Close();
    }
}
```

C# Language Specification

For more information, see Exceptions and The try statement in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- Exceptions and Exception Handling
- try-catch
- try-finally
- try-catch-finally
- using Statement

Creating and Throwing Exceptions (C# Programming Guide)

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

Exceptions are used to indicate that an error has occurred while running the program. Exception objects that describe an error are created and then *thrown* with the throw keyword. The runtime then searches for the most compatible exception handler.

Programmers should throw exceptions when one or more of the following conditions are true:

- The method cannot complete its defined functionality.

  For example, if a parameter to a method has an invalid value:

```
static void CopyObject(SampleClass original)
{
    if (original == null)
    {
        throw new System.ArgumentException("Parameter cannot be null", "original");
    }

}
```

- An inappropriate call to an object is made, based on the object state.

  One example might be trying to write to a read-only file. In cases where an object state does not allow an operation, throw an instance of InvalidOperationException or an object based on a derivation of this class. This is an example of a method that throws an InvalidOperationException object:

```
class ProgramLog
{
    System.IO.FileStream logFile = null;
    void OpenLog(System.IO.FileInfo fileName, System.IO.FileMode mode) {}

    void WriteLog()
    {
        if (!this.logFile.CanWrite)
        {
            throw new System.InvalidOperationException("Logfile cannot be read-only");
        }
        // Else write data to the log and return.
    }
}
```

- When an argument to a method causes an exception.

  In this case, the original exception should be caught and an ArgumentException instance should be created. The original exception should be passed to the constructor of the ArgumentException as the InnerException parameter:

```
static int GetValueFromArray(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (System.IndexOutOfRangeException ex)
    {
        System.ArgumentException argEx = new System.ArgumentException("Index is out of range", "index",
ex);
        throw argEx;
    }
}
```

Exceptions contain a property named StackTrace. This string contains the name of the methods on the current call stack, together with the file name and line number where the exception was thrown for each method. A StackTrace object is created automatically by the common language runtime (CLR) from the point of the `throw` statement, so that exceptions must be thrown from the point where the stack trace should begin.

All exceptions contain a property named Message. This string should be set to explain the reason for the exception. Note that information that is sensitive to security should not be put in the message text. In addition to Message, ArgumentException contains a property named ParamName that should be set to the name of the argument that caused the exception to be thrown. In the case of a property setter, ParamName should be set to `value`.

Public and protected methods should throw exceptions whenever they cannot complete their intended functions. The exception class that is thrown should be the most specific exception available that fits the error conditions. These exceptions should be documented as part of the class functionality, and derived classes or updates to the original class should retain the same behavior for backward compatibility.

Things to Avoid When Throwing Exceptions

The following list identifies practices to avoid when throwing exceptions:

- Exceptions should not be used to change the flow of a program as part of ordinary execution. Exceptions should only be used to report and handle error conditions.
- Exceptions should not be returned as a return value or parameter instead of being thrown.
- Do not throw System.Exception, System.SystemException, System.NullReferenceException, or System.IndexOutOfRangeException intentionally from your own source code.
- Do not create exceptions that can be thrown in debug mode but not release mode. To identify run-time errors during the development phase, use Debug Assert instead.

Defining Exception Classes

Programs can throw a predefined exception class in the System namespace (except where previously noted), or create their own exception classes by deriving from Exception. The derived classes should define at least four constructors: one default constructor, one that sets the message property, and one that sets both the Message and InnerException properties. The fourth constructor is used to serialize the exception. New exception classes should be serializable. For example:

```
[Serializable()]
public class InvalidDepartmentException : System.Exception
{
    public InvalidDepartmentException() : base() { }
    public InvalidDepartmentException(string message) : base(message) { }
    public InvalidDepartmentException(string message, System.Exception inner) : base(message, inner) { }

    // A constructor is needed for serialization when an
    // exception propagates from a remoting server to the client.
    protected InvalidDepartmentException(System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context) : base(info, context) { }
}
```

New properties should only be added to the exception class when the data they provide is useful to resolving the exception. If new properties are added to the derived exception class, `ToString()` should be overridden to return the added information.

C# Language Specification

For more information, see Exceptions and The throw statement in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- Exceptions and Exception Handling
- Exception Hierarchy
- Exception Handling

Compiler-Generated Exceptions (C# Programming Guide)

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

Some exceptions are thrown automatically by the .NET Framework's common language runtime (CLR) when basic operations fail. These exceptions and their error conditions are listed in the following table.

EXCEPTION	DESCRIPTION
ArithmeticException	A base class for exceptions that occur during arithmetic operations, such as DivideByZeroException and OverflowException.
ArrayTypeMismatchException	Thrown when an array cannot store a given element because the actual type of the element is incompatible with the actual type of the array.
DivideByZeroException	Thrown when an attempt is made to divide an integral value by zero.
IndexOutOfRangeException	Thrown when an attempt is made to index an array when the index is less than zero or outside the bounds of the array.
InvalidCastException	Thrown when an explicit conversion from a base type to an interface or to a derived type fails at runtime.
NullReferenceException	Thrown when you attempt to reference an object whose value is null.
OutOfMemoryException	Thrown when an attempt to allocate memory using the new operator fails. This indicates that the memory available to the common language runtime has been exhausted.
OverflowException	Thrown when an arithmetic operation in a `checked` context overflows.
StackOverflowException	Thrown when the execution stack is exhausted by having too many pending method calls; usually indicates a very deep or infinite recursion.
TypeInitializationException	Thrown when a static constructor throws an exception and no compatible `catch` clause exists to catch it.

See also

- C# Programming Guide
- Exceptions and Exception Handling
- Exception Handling
- try-catch
- try-finally

- try-catch-finally

How to: Handle an Exception Using try/catch (C# Programming Guide)

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

The purpose of a try-catch block is to catch and handle an exception generated by working code. Some exceptions can be handled in a `catch` block and the problem solved without the exception being re-thrown; however, more often the only thing that you can do is make sure that the appropriate exception is thrown.

Example

In this example, IndexOutOfRangeException is not the most appropriate exception: ArgumentOutOfRangeException makes more sense for the method because the error is caused by the `index` argument passed in by the caller.

```
class TestTryCatch
{
    static int GetInt(int[] array, int index)
    {
        try
        {
            return array[index];
        }
        catch (System.IndexOutOfRangeException e)  // CS0168
        {
            System.Console.WriteLine(e.Message);
            // Set IndexOutOfRangeException to the new exception's InnerException.
            throw new System.ArgumentOutOfRangeException("index parameter is out of range.", e);
        }
    }
}
```

Comments

The code that causes an exception is enclosed in the `try` block. A `catch` statement is added immediately after to handle `IndexOutOfRangeException`, if it occurs. The `catch` block handles the `IndexOutOfRangeException` and throws the more appropriate `ArgumentOutOfRangeException` exception instead. In order to provide the caller with as much information as possible, consider specifying the original exception as the InnerException of the new exception. Because the InnerException property is readonly, you must assign it in the constructor of the new exception.

See also

- C# Programming Guide
- Exceptions and Exception Handling
- Exception Handling

How to: Execute Cleanup Code Using finally (C# Programming Guide)

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

The purpose of a `finally` statement is to ensure that the necessary cleanup of objects, usually objects that are holding external resources, occurs immediately, even if an exception is thrown. One example of such cleanup is calling Close on a FileStream immediately after use instead of waiting for the object to be garbage collected by the common language runtime, as follows:

```
static void CodeWithoutCleanup()
{
    System.IO.FileStream file = null;
    System.IO.FileInfo fileInfo = new System.IO.FileInfo("C:\\file.txt");

    file = fileInfo.OpenWrite();
    file.WriteByte(0xF);

    file.Close();
}
```

Example

To turn the previous code into a `try-catch-finally` statement, the cleanup code is separated from the working code, as follows.

```
static void CodeWithCleanup()
{
    System.IO.FileStream file = null;
    System.IO.FileInfo fileInfo = null;

    try
    {
        fileInfo = new System.IO.FileInfo("C:\\file.txt");

        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    catch(System.UnauthorizedAccessException e)
    {
        System.Console.WriteLine(e.Message);
    }
    finally
    {
        if (file != null)
        {
            file.Close();
        }
    }
}
```

Because an exception can occur at any time within the `try` block before the `OpenWrite()` call, or the `OpenWrite()` call itself could fail, we are not guaranteed that the file is open when we try to close it. The `finally` block adds a check to make sure that the FileStream object is not `null` before you call the Close method. Without the `null` check, the `finally` block could throw its own NullReferenceException, but throwing exceptions in `finally` blocks

should be avoided if it is possible.

A database connection is another good candidate for being closed in a `finally` block. Because the number of connections allowed to a database server is sometimes limited, you should close database connections as quickly as possible. If an exception is thrown before you can close your connection, this is another case where using the `finally` block is better than waiting for garbage collection.

See also

- C# Programming Guide
- Exceptions and Exception Handling
- Exception Handling
- using Statement
- try-catch
- try-finally
- try-catch-finally

How to: Catch a non-CLS Exception

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

Some .NET languages, including C++/CLI, allow objects to throw exceptions that do not derive from Exception. Such exceptions are called *non-CLS exceptions* or *non-Exceptions*. In C# you cannot throw non-CLS exceptions, but you can catch them in two ways:

- Within a `catch (RuntimeWrappedException e)` block.

  By default, a Visual C# assembly catches non-CLS exceptions as wrapped exceptions. Use this method if you need access to the original exception, which can be accessed through the RuntimeWrappedException.WrappedException property. The procedure later in this topic explains how to catch exceptions in this manner.

- Within a general catch block (a catch block without an exception type specified) that is put after all other `catch` blocks.

  Use this method when you want to perform some action (such as writing to a log file) in response to non-CLS exceptions, and you do not need access to the exception information. By default the common language runtime wraps all exceptions. To disable this behavior, add this assembly-level attribute to your code, typically in the AssemblyInfo.cs file: `[assembly: RuntimeCompatibilityAttribute(WrapNonExceptionThrows = false)]`.

To catch a non-CLS exception

Within a `catch(RuntimeWrappedException e)` block, access the original exception through the RuntimeWrappedException.WrappedException property.

Example

The following example shows how to catch a non-CLS exception that was thrown from a class library written in C++/CLI. Note that in this example, the C# client code knows in advance that the exception type being thrown is a System.String. You can cast the RuntimeWrappedException.WrappedException property back its original type as long as that type is accessible from your code.

```
// Class library written in C++/CLI.
var myClass = new ThrowNonCLS.Class1();

try
{
    // throws gcnew System::String(
    // "I do not derive from System.Exception!");
    myClass.TestThrow();
}
catch (RuntimeWrappedException e)
{
    String s = e.WrappedException as String;
    if (s != null)
    {
        Console.WriteLine(s);
    }
}
```

See also

- RuntimeWrappedException
- Exceptions and Exception Handling

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#

Contents

Interoperability (C# Programming Guide)

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

Interoperability enables you to preserve and take advantage of existing investments in unmanaged code. Code that runs under the control of the common language runtime (CLR) is called *managed code*, and code that runs outside the CLR is called *unmanaged code*. COM, COM+, C++ components, ActiveX components, and Microsoft Win32 API are examples of unmanaged code.

The .NET Framework enables interoperability with unmanaged code through platform invoke services, the System.Runtime.InteropServices namespace, C++ interoperability, and COM interoperability (COM interop).

In This Section

Interoperability Overview
Describes methods to interoperate between C# managed code and unmanaged code.

How to: Access Office Interop Objects by Using Visual C# Features
Describes features that are introduced in Visual C# to facilitate Office programming.

How to: Use Indexed Properties in COM Interop Programming
Describes how to use indexed properties to access COM properties that have parameters.

How to: Use Platform Invoke to Play a Wave File
Describes how to use platform invoke services to play a .wav sound file on the Windows operating system.

Walkthrough: Office Programming
Shows how to create an Excel workbook and a Word document that contains a link to the workbook.

Example COM Class
Demonstrates how to expose a C# class as a COM object.

C# Language Specification

For more information, see Basic concepts in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- Marshal.ReleaseComObject
- C# Programming Guide
- Interoperating with Unmanaged Code
- Walkthrough: Office Programming

Interoperability Overview (C# Programming Guide)

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

The topic describes methods to enable interoperability between C# managed code and unmanaged code.

Platform Invoke

Platform invoke is a service that enables managed code to call unmanaged functions that are implemented in dynamic link libraries (DLLs), such as those in the Microsoft Win32 API. It locates and invokes an exported function and marshals its arguments (integers, strings, arrays, structures, and so on) across the interoperation boundary as needed.

For more information, see Consuming Unmanaged DLL Functions and How to: Use Platform Invoke to Play a Wave File.

> **NOTE**
>
> The Common Language Runtime (CLR) manages access to system resources. Calling unmanaged code that is outside the CLR bypasses this security mechanism, and therefore presents a security risk. For example, unmanaged code might call resources in unmanaged code directly, bypassing CLR security mechanisms. For more information, see Security in .NET.

C++ Interop

You can use C++ interop, also known as It Just Works (IJW), to wrap a native C++ class so that it can be consumed by code that is authored in C# or another .NET Framework language. To do this, you write C++ code to wrap a native DLL or COM component. Unlike other .NET Framework languages, Visual C++ has interoperability support that enables managed and unmanaged code to be located in the same application and even in the same file. You then build the C++ code by using the **/clr** compiler switch to produce a managed assembly. Finally, you add a reference to the assembly in your C# project and use the wrapped objects just as you would use other managed classes.

Exposing COM Components to C#

You can consume a COM component from a C# project. The general steps are as follows:

1. Locate a COM component to use and register it. Use regsvr32.exe to register or un–register a COM DLL.
2. Add to the project a reference to the COM component or type library.

   When you add the reference, Visual Studio uses the Tlbimp.exe (Type Library Importer), which takes a type library as input, to output a .NET Framework interop assembly. The assembly, also named a runtime callable wrapper (RCW), contains managed classes and interfaces that wrap the COM classes and interfaces that are in the type library. Visual Studio adds to the project a reference to the generated assembly.
3. Create an instance of a class that is defined in the RCW. This, in turn, creates an instance of the COM object.
4. Use the object just as you use other managed objects. When the object is reclaimed by garbage collection, the instance of the COM object is also released from memory.

For more information, see Exposing COM Components to the .NET Framework.

Exposing C# to COM

COM clients can consume C# types that have been correctly exposed. The basic steps to expose C# types are as follows:

1. Add interop attributes in the C# project.

   You can make an assembly COM visible by modifying Visual C# project properties. For more information, see Assembly Information Dialog Box.

2. Generate a COM type library and register it for COM usage.

   You can modify Visual C# project properties to automatically register the C# assembly for COM interop. Visual Studio uses the Regasm.exe (Assembly Registration Tool), using the `/tlb` command-line switch, which takes a managed assembly as input, to generate a type library. This type library describes the `public` types in the assembly and adds registry entries so that COM clients can create managed classes.

For more information, see Exposing .NET Framework Components to COM and Example COM Class.

See also

- Improving Interop Performance
- Introduction to Interoperability between COM and .NET
- Introduction to COM Interop in Visual Basic
- Marshaling between Managed and Unmanaged Code
- Interoperating with Unmanaged Code
- C# Programming Guide

How to: Access Office Interop Objects by Using Visual C# Features (C# Programming Guide)

1/23/2019 • 12 minutes to read • Edit Online

Visual C# has features that simplify access to Office API objects. The new features include named and optional arguments, a new type called `dynamic`, and the ability to pass arguments to reference parameters in COM methods as if they were value parameters.

In this topic you will use the new features to write code that creates and displays a Microsoft Office Excel worksheet. You will then write code to add an Office Word document that contains an icon that is linked to the Excel worksheet.

To complete this walkthrough, you must have Microsoft Office Excel 2007 and Microsoft Office Word 2007, or later versions, installed on your computer.

If you are using an operating system that is older than Windows Vista, make sure that .NET Framework 2.0 is installed.

> **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 create a new console application

1. Start Visual Studio.
2. On the **File** menu, point to **New**, and then click **Project**. The **New Project** dialog box appears.
3. In the **Installed Templates** pane, expand **Visual C#**, and then click **Windows**.
4. Look at the top of the **New Project** dialog box to make sure that **.NET Framework 4** (or later version) is selected as a target framework.
5. In the **Templates** pane, click **Console Application**.
6. Type a name for your project in the **Name** field.
7. Click **OK**.

   The new project appears in **Solution Explorer**.

To add references

1. In **Solution Explorer**, right-click your project's name and then click **Add Reference**. The **Add Reference** dialog box appears.
2. On the **Assemblies** page, select **Microsoft.Office.Interop.Word** in the **Component Name** list, and then hold down the CTRL key and select **Microsoft.Office.Interop.Excel**. If you do not see the assemblies, you may need to ensure they are installed and displayed (see How to: Install Office Primary Interop Assemblies)
3. Click **OK**.

To add necessary using directives

1. In **Solution Explorer**, right-click the **Program.cs** file and then click **View Code**.
2. Add the following `using` directives to the top of the code file.

```
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

To create a list of bank accounts

1. Paste the following class definition into **Program.cs**, under the `Program` class.

```
public class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

2. Add the following code to the `Main` method to create a `bankAccounts` list that contains two accounts.

```
// Create a list of accounts.
var bankAccounts = new List<Account> {
    new Account {
                  ID = 345678,
                  Balance = 541.27
                },
    new Account {
                  ID = 1230221,
                  Balance = -127.44
                }
};
```

To declare a method that exports account information to Excel

1. Add the following method to the `Program` class to set up an Excel worksheet.

   Method Microsoft.Office.Interop.Excel.Workbooks.Add* has an optional parameter for specifying a particular template. Optional parameters, new in C# 4, enable you to omit the argument for that parameter if you want to use the parameter's default value. Because no argument is sent in the following code, `Add` uses the default template and creates a new workbook. The equivalent statement in earlier versions of C# requires a placeholder argument: `ExcelApp.Workbooks.Add(Type.Missing)`.

```
static void DisplayInExcel(IEnumerable<Account> accounts)
{
    var excelApp = new Excel.Application();
    // Make the object visible.
    excelApp.Visible = true;

    // Create a new, empty workbook and add it to the collection returned
    // by property Workbooks. The new workbook becomes the active workbook.
    // Add has an optional parameter for specifying a praticular template.
    // Because no argument is sent in this example, Add creates a new workbook.
    excelApp.Workbooks.Add();

    // This example uses a single workSheet. The explicit type casting is
    // removed in a later procedure.
    Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet;
}
```

2. Add the following code at the end of `DisplayInExcel`. The code inserts values into the first two columns of the first row of the worksheet.

```
// Establish column headings in cells A1 and B1.
workSheet.Cells[1, "A"] = "ID Number";
workSheet.Cells[1, "B"] = "Current Balance";
```

3. Add the following code at the end of `DisplayInExcel`. The `foreach` loop puts the information from the list of accounts into the first two columns of successive rows of the worksheet.

```
var row = 1;
foreach (var acct in accounts)
{
    row++;
    workSheet.Cells[row, "A"] = acct.ID;
    workSheet.Cells[row, "B"] = acct.Balance;
}
```

4. Add the following code at the end of `DisplayInExcel` to adjust the column widths to fit the content.

```
workSheet.Columns[1].AutoFit();
workSheet.Columns[2].AutoFit();
```

   Earlier versions of C# require explicit casting for these operations because `ExcelApp.Columns[1]` returns an `Object`, and `AutoFit` is an Excel Range method. The following lines show the casting.

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

   C# 4, and later versions, converts the returned `Object` to `dynamic` automatically if the assembly is referenced by the /link compiler option or, equivalently, if the Excel **Embed Interop Types** property is set to true. True is the default value for this property.

To run the project

1. Add the following line at the end of `Main`.

```
// Display the list in an Excel spreadsheet.
DisplayInExcel(bankAccounts);
```

2. Press CTRL+F5.

   An Excel worksheet appears that contains the data from the two accounts.

To add a Word document

1. To illustrate additional ways in which C# 4, and later versions, enhances Office programming, the following code opens a Word application and creates an icon that links to the Excel worksheet.

   Paste method `CreateIconInWordDoc`, provided later in this step, into the `Program` class. `CreateIconInWordDoc` uses named and optional arguments to reduce the complexity of the method calls to Microsoft.Office.Interop.Word.Documents.Add* and PasteSpecial. These calls incorporate two other new features introduced in C# 4 that simplify calls to COM methods that have reference parameters. First, you can send arguments to the reference parameters as if they were value parameters. That is, you can send values directly, without creating a variable for each reference parameter. The compiler generates temporary variables to hold the argument values, and discards the variables when you return from the call. Second, you can omit the `ref` keyword in the argument list.

   The `Add` method has four reference parameters, all of which are optional. In C# 4, or later versions, you can omit arguments for any or all of the parameters if you want to use their default values. In Visual C# 2008 and earlier versions, an argument must be provided for each parameter, and the argument must be a variable because the parameters are reference parameters.

   The `PasteSpecial` method inserts the contents of the Clipboard. The method has seven reference parameters, all of which are optional. The following code specifies arguments for two of them: `Link`, to create a link to the source of the Clipboard contents, and `DisplayAsIcon`, to display the link as an icon. In C# 4, you can use named arguments for those two and omit the others. Although these are reference parameters, you do not have to use the `ref` keyword, or to create variables to send in as arguments. You can send the values directly. In Visual C# 2008 and earlier versions, you must send a variable argument for each reference parameter.

```
static void CreateIconInWordDoc()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four reference parameters, all of which are
    // optional. Visual C# allows you to omit arguments for them if
    // the default values are what you want.
    wordApp.Documents.Add();

    // PasteSpecial has seven reference parameters, all of which are
    // optional. This example uses named arguments to specify values
    // for two of the parameters. Although these are reference
    // parameters, you do not need to use the ref keyword, or to create
    // variables to send in as arguments. You can send the values directly.
    wordApp.Selection.PasteSpecial( Link: true, DisplayAsIcon: true);
}
```

   In Visual C# 2008 or earlier versions of the language, the following more complex code is required.

```
static void CreateIconInWordDoc2008()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four parameters, all of which are optional.
    // In Visual C# 2008 and earlier versions, an argument has to be sent
    // for every parameter. Because the parameters are reference
    // parameters of type object, you have to create an object variable
    // for the arguments that represents 'no value'.

    object useDefaultValue = Type.Missing;

    wordApp.Documents.Add(ref useDefaultValue, ref useDefaultValue,
        ref useDefaultValue, ref useDefaultValue);

    // PasteSpecial has seven reference parameters, all of which are
    // optional. In this example, only two of the parameters require
    // specified values, but in Visual C# 2008 an argument must be sent
    // for each parameter. Because the parameters are reference parameters,
    // you have to contruct variables for the arguments.
    object link = true;
    object displayAsIcon = true;

    wordApp.Selection.PasteSpecial( ref useDefaultValue,
                                    ref link,
                                    ref useDefaultValue,
                                    ref displayAsIcon,
                                    ref useDefaultValue,
                                    ref useDefaultValue,
                                    ref useDefaultValue);
}
```

2. Add the following statement at the end of `Main`.

```
// Create a Word document that contains an icon that links to
// the spreadsheet.
CreateIconInWordDoc();
```

3. Add the following statement at the end of `DisplayInExcel`. The `Copy` method adds the worksheet to the Clipboard.

```
// Put the spreadsheet contents on the clipboard. The Copy method has one
// optional parameter for specifying a destination. Because no argument
// is sent, the destination is the Clipboard.
workSheet.Range["A1:B3"].Copy();
```

4. Press CTRL+F5.

   A Word document appears that contains an icon. Double-click the icon to bring the worksheet to the foreground.

To set the Embed Interop Types property

1. Additional enhancements are possible when you call a COM type that does not require a primary interop assembly (PIA) at run time. Removing the dependency on PIAs results in version independence and easier deployment. For more information about the advantages of programming without PIAs, see Walkthrough: Embedding Types from Managed Assemblies.

   In addition, programming is easier because the types that are required and returned by COM methods can

be represented by using the type `dynamic` instead of `Object`. Variables that have type `dynamic` are not evaluated until run time, which eliminates the need for explicit casting. For more information, see Using Type dynamic.

In C# 4, embedding type information instead of using PIAs is default behavior. Because of that default, several of the previous examples are simplified because explicit casting is not required. For example, the declaration of `worksheet` in `DisplayInExcel` is written as `Excel._Worksheet workSheet = excelApp.ActiveSheet` rather than `Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet`. The calls to `AutoFit` in the same method also would require explicit casting without the default, because `ExcelApp.Columns[1]` returns an `Object`, and `AutoFit` is an Excel method. The following code shows the casting.

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

2. To change the default and use PIAs instead of embedding type information, expand the **References** node in **Solution Explorer** and then select **Microsoft.Office.Interop.Excel** or **Microsoft.Office.Interop.Word**.

3. If you cannot see the **Properties** window, press **F4**.

4. Find **Embed Interop Types** in the list of properties, and change its value to **False**. Equivalently, you can compile by using the /reference compiler option instead of /link at a command prompt.

To add additional formatting to the table

1. Replace the two calls to `AutoFit` in `DisplayInExcel` with the following statement.

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

The AutoFormat method has seven value parameters, all of which are optional. Named and optional arguments enable you to provide arguments for none, some, or all of them. In the previous statement, an argument is supplied for only one of the parameters, `Format`. Because `Format` is the first parameter in the parameter list, you do not have to provide the parameter name. However, the statement might be easier to understand if the parameter name is included, as is shown in the following code.

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(Format:
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

2. Press CTRL+F5 to see the result. Other formats are listed in the XlRangeAutoFormat enumeration.

3. Compare the statement in step 1 with the following code, which shows the arguments that are required in Visual C# 2008 or earlier versions.

```
// The AutoFormat method has seven optional value parameters. The
// following call specifies a value for the first parameter, and uses
// the default values for the other six.

// Call to AutoFormat in Visual C# 2008. This code is not part of the
// current solution.
excelApp.get_Range("A1", "B4").AutoFormat(Excel.XlRangeAutoFormat.xlRangeAutoFormatTable3,
    Type.Missing, Type.Missing, Type.Missing, Type.Missing, Type.Missing,
    Type.Missing);
```

Example

The following code shows the complete example.

```
using System;
using System.Collections.Generic;
using System.Linq;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;


namespace OfficeProgramminWalkthruComplete
{
    class Walkthrough
    {
        static void Main(string[] args)
        {
            // Create a list of accounts.
            var bankAccounts = new List<Account>
            {
                new Account {
                              ID = 345678,
                              Balance = 541.27
                            },
                new Account {
                              ID = 1230221,
                              Balance = -127.44
                            }
            };

            // Display the list in an Excel spreadsheet.
            DisplayInExcel(bankAccounts);

            // Create a Word document that contains an icon that links to
            // the spreadsheet.
            CreateIconInWordDoc();
        }

        static void DisplayInExcel(IEnumerable<Account> accounts)
        {
            var excelApp = new Excel.Application();
            // Make the object visible.
            excelApp.Visible = true;

            // Create a new, empty workbook and add it to the collection returned
            // by property Workbooks. The new workbook becomes the active workbook.
            // Add has an optional parameter for specifying a praticular template.
            // Because no argument is sent in this example, Add creates a new workbook.
            excelApp.Workbooks.Add();

            // This example uses a single workSheet.
            Excel._Worksheet workSheet = excelApp.ActiveSheet;

            // Earlier versions of C# require explicit casting.
            //Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet;

            // Establish column headings in cells A1 and B1.
            workSheet.Cells[1, "A"] = "ID Number";
            workSheet.Cells[1, "B"] = "Current Balance";

            var row = 1;
            foreach (var acct in accounts)
            {
                row++;
                workSheet.Cells[row, "A"] = acct.ID;
                workSheet.Cells[row, "B"] = acct.Balance;
            }
```

```
            workSheet.Columns[1].AutoFit();
            workSheet.Columns[2].AutoFit();

            // Call to AutoFormat in Visual C#. This statement replaces the
            // two calls to AutoFit.
            workSheet.Range["A1", "B3"].AutoFormat(
                Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);

            // Put the spreadsheet contents on the clipboard. The Copy method has one
            // optional parameter for specifying a destination. Because no argument
            // is sent, the destination is the Clipboard.
            workSheet.Range["A1:B3"].Copy();
        }

        static void CreateIconInWordDoc()
        {
            var wordApp = new Word.Application();
            wordApp.Visible = true;

            // The Add method has four reference parameters, all of which are
            // optional. Visual C# allows you to omit arguments for them if
            // the default values are what you want.
            wordApp.Documents.Add();

            // PasteSpecial has seven reference parameters, all of which are
            // optional. This example uses named arguments to specify values
            // for two of the parameters. Although these are reference
            // parameters, you do not need to use the ref keyword, or to create
            // variables to send in as arguments. You can send the values directly.
            wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
        }
    }

    public class Account
    {
        public int ID { get; set; }
        public double Balance { get; set; }
    }
}
```

See also

- Type.Missing
- dynamic
- Using Type dynamic
- Named and Optional Arguments
- How to: Use Named and Optional Arguments in Office Programming

How to: Use Indexed Properties in COM Interop Programming (C# Programming Guide)

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

Indexed properties improve the way in which COM properties that have parameters are consumed in C# programming. Indexed properties work together with other features in Visual C#, such as named and optional arguments, a new type (dynamic), and embedded type information, to enhance Microsoft Office programming.

In earlier versions of C#, methods are accessible as properties only if the `get` method has no parameters and the `set` method has one and only one value parameter. However, not all COM properties meet those restrictions. For example, the Excel Range property has a `get` accessor that requires a parameter for the name of the range. In the past, because you could not access the `Range` property directly, you had to use the `get_Range` method instead, as shown in the following example.

```
// Visual C# 2008 and earlier.
var excelApp = new Excel.Application();
// . . .
Excel.Range targetRange = excelApp.get_Range("A1", Type.Missing);
```

Indexed properties enable you to write the following instead:

```
// Visual C# 2010.
var excelApp = new Excel.Application();
// . . .
Excel.Range targetRange = excelApp.Range["A1"];
```

NOTE

The previous example also uses the optional arguments feature, which enables you to omit `Type.Missing`.

Similarly to set the value of the `Value` property of a Range object in Visual C# 2008 and earlier, two arguments are required. One supplies an argument for an optional parameter that specifies the type of the range value. The other supplies the value for the `Value` property. The following examples illustrate these techniques. Both set the value of the A1 cell to `Name`.

```
// Visual C# 2008.
targetRange.set_Value(Type.Missing, "Name");
// Or
targetRange.Value2 = "Name";
```

Indexed properties enable you to write the following code instead.

```
// Visual C# 2010.
targetRange.Value = "Name";
```

You cannot create indexed properties of your own. The feature only supports consumption of existing indexed properties.

Example

The following code shows a complete example. For more information about how to set up a project that accesses the Office API, see How to: Access Office Interop Objects by Using Visual C# Features.

```
// You must add a reference to Microsoft.Office.Interop.Excel to run
// this example.
using System;
using Excel = Microsoft.Office.Interop.Excel;

namespace IndexedProperties
{
    class Program
    {
        static void Main(string[] args)
        {
            CSharp2010();
            //CSharp2008();
        }

        static void CSharp2010()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add();
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.Range["A1"];
            targetRange.Value = "Name";
        }

        static void CSharp2008()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add(Type.Missing);
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.get_Range("A1", Type.Missing);
            targetRange.set_Value(Type.Missing, "Name");
            // Or
            //targetRange.Value2 = "Name";

        }
    }
}
```

See also

- Named and Optional Arguments
- dynamic
- Using Type dynamic
- How to: Use Named and Optional Arguments in Office Programming
- How to: Access Office Interop Objects by Using Visual C# Features
- Walkthrough: Office Programming

How to: Use Platform Invoke to Play a Wave File (C# Programming Guide)

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

The following C# code example illustrates how to use platform invoke services to play a wave sound file on the Windows operating system.

Example

This example code uses `DllImport` to import `winmm.dll`'s `PlaySound` method entry point as `Form1 PlaySound()`. The example has a simple Windows Form with a button. Clicking the button opens a standard windows OpenFileDialog dialog box so that you can open a file to play. When a wave file is selected, it is played by using the `PlaySound()` method of the `winmm.dll` library. For more information about this method, see Using the PlaySound function with Waveform-Audio Files. Browse and select a file that has a .wav extension, and then click **Open** to play the wave file by using platform invoke. A text box shows the full path of the file selected.

The **Open Files** dialog box is filtered to show only files that have a .wav extension through the filter settings:

```
dialog1.Filter = "Wav Files (*.wav)|*.wav";
```

```
using System.Windows.Forms;
using System.Runtime.InteropServices;

namespace WinSound
{
    public partial class Form1 : Form
    {
        private TextBox textBox1;
        private Button button1;

        public Form1()  //constructor
        {
            InitializeComponent();
        }

        [System.Runtime.InteropServices.DllImport("winmm.DLL", EntryPoint = "PlaySound", SetLastError = true,
CharSet = CharSet.Unicode, ThrowOnUnmappableChar = true)]
        private static extern bool PlaySound(string szSound, System.IntPtr hMod, PlaySoundFlags flags);

        [System.Flags]
        public enum PlaySoundFlags : int
        {
            SND_SYNC = 0x0000,
            SND_ASYNC = 0x0001,
            SND_NODEFAULT = 0x0002,
            SND_LOOP = 0x0008,
            SND_NOSTOP = 0x0010,
            SND_NOWAIT = 0x00002000,
            SND_FILENAME = 0x00020000,
            SND_RESOURCE = 0x00040004
        }

        private void button1_Click (object sender, System.EventArgs e)
        {
            OpenFileDialog dialog1 = new OpenFileDialog();

            dialog1.Title = "Browse to find sound file to play";
            dialog1.InitialDirectory = @"c:\";
            dialog1.Filter = "Wav Files (*.wav)|*.wav";
            dialog1.FilterIndex = 2;
            dialog1.RestoreDirectory = true;

            if(dialog1.ShowDialog() == DialogResult.OK)
            {
                textBox1.Text = dialog1.FileName;
                PlaySound (dialog1.FileName, new System.IntPtr(), PlaySoundFlags.SND_SYNC);
            }
        }
    }
}
```

Compiling the Code

To compile the code

1. Create a new C# Windows Application project in Visual Studio and name it **WinSound**.
2. Copy the code above, and paste it over the contents of the `Form1.cs` file.
3. Copy the following code, and paste it in the `Form1.Designer.cs` file, in the `InitializeComponent()` method, after any existing code.

```
this.button1 = new System.Windows.Forms.Button();
this.textBox1 = new System.Windows.Forms.TextBox();
this.SuspendLayout();
//
// button1
//
this.button1.Location = new System.Drawing.Point(192, 40);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(88, 24);
this.button1.TabIndex = 0;
this.button1.Text = "Browse";
this.button1.Click += new System.EventHandler(this.button1_Click);
//
// textBox1
//
this.textBox1.Location = new System.Drawing.Point(8, 40);
this.textBox1.Name = "textBox1";
this.textBox1.Size = new System.Drawing.Size(168, 20);
this.textBox1.TabIndex = 1;
this.textBox1.Text = "FIle path";
//
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(5, 13);
this.ClientSize = new System.Drawing.Size(292, 266);
this.Controls.Add(this.textBox1);
this.Controls.Add(this.button1);
this.Name = "Form1";
this.Text = "Platform Invoke WinSound C#";
this.ResumeLayout(false);
this.PerformLayout();
```

4. Compile and run the code.

.NET Framework Security

For more information, see Security in .NET.

See also

- C# Programming Guide
- Interoperability Overview
- A Closer Look at Platform Invoke
- Marshaling Data with Platform Invoke

Walkthrough: Office Programming (C# and Visual Basic)

2/13/2019 • 11 minutes to read • Edit Online

Visual Studio offers features in C# and Visual Basic that improve Microsoft Office programming. Helpful C# features include named and optional arguments and return values of type `dynamic`. In COM programming, you can omit the `ref` keyword and gain access to indexed properties. Features in Visual Basic include auto-implemented properties, statements in lambda expressions, and collection initializers.

Both languages enable embedding of type information, which allows deployment of assemblies that interact with COM components without deploying primary interop assemblies (PIAs) to the user's computer. For more information, see Walkthrough: Embedding Types from Managed Assemblies.

This walkthrough demonstrates these features in the context of Office programming, but many of these features are also useful in general programming. In the walkthrough, you use an Excel Add-in application to create an Excel workbook. Next, you create a Word document that contains a link to the workbook. Finally, you see how to enable and disable the PIA dependency.

Prerequisites

You must have Microsoft Office Excel and Microsoft Office Word installed on your computer to complete this walkthrough.

If you are using an operating system that is older than Windows Vista, make sure that .NET Framework 2.0 is installed.

> **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 set up an Excel Add-in application

1. Start Visual Studio.
2. On the **File** menu, point to **New**, and then click **Project**.
3. In the **Installed Templates** pane, expand **Visual Basic** or **Visual C#**, expand **Office**, and then click the version year of the Office product.
4. In the **Templates** pane, click **Excel <version> Add-in**.
5. Look at the top of the **Templates** pane to make sure that **.NET Framework 4**, or a later version, appears in the **Target Framework** box.
6. Type a name for your project in the **Name** box, if you want to.
7. Click **OK**.
8. The new project appears in **Solution Explorer**.

To add references

1. In **Solution Explorer**, right-click your project's name and then click **Add Reference**. The **Add Reference** dialog box appears.

2. On the **Assemblies** tab, select **Microsoft.Office.Interop.Excel**, version `<version>.0.0.0` (for a key to the Office product version numbers, see Microsoft Versions), in the **Component Name** list, and then hold down the CTRL key and select **Microsoft.Office.Interop.Word**, `version <version>.0.0.0`. If you do not see the assemblies, you may need to ensure they are installed and displayed (see How to: Install Office Primary Interop Assemblies).

3. Click **OK**.

To add necessary Imports statements or using directives

1. In **Solution Explorer**, right-click the **ThisAddIn.vb** or **ThisAddIn.cs** file and then click **View Code**.

2. Add the following `Imports` statements (Visual Basic) or `using` directives (C#) to the top of the code file if they are not already present.

```
using System.Collections.Generic;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

```
Imports Microsoft.Office.Interop
```

To create a list of bank accounts

1. In **Solution Explorer**, right-click your project's name, click **Add**, and then click **Class**. Name the class Account.vb if you are using Visual Basic or Account.cs if you are using C#. Click **Add**.

2. Replace the definition of the `Account` class with the following code. The class definitions use *auto-implemented properties*. For more information, see Auto-Implemented Properties.

```
class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

```
Public Class Account
    Property ID As Integer = -1
    Property Balance As Double
End Class
```

3. To create a `bankAccounts` list that contains two accounts, add the following code to the `ThisAddIn_Startup` method in *ThisAddIn.vb* or *ThisAddIn.cs*. The list declarations use *collection initializers*. For more information, see Collection Initializers.

```
var bankAccounts = new List<Account>
{
    new Account
    {
        ID = 345,
        Balance = 541.27
    },
    new Account
    {
        ID = 123,
        Balance = -127.44
    }
};
```

```
Dim bankAccounts As New List(Of Account) From {
    New Account With {
                          .ID = 345,
                          .Balance = 541.27
                      },
    New Account With {
                          .ID = 123,
                          .Balance = -127.44
                      }
    }
```

To export data to Excel

1. In the same file, add the following method to the `ThisAddIn` class. The method sets up an Excel workbook and exports data to it.

```
void DisplayInExcel(IEnumerable<Account> accounts,
           Action<Account, Excel.Range> DisplayFunc)
{
    var excelApp = this.Application;
    // Add a new Excel workbook.
    excelApp.Workbooks.Add();
    excelApp.Visible = true;
    excelApp.Range["A1"].Value = "ID";
    excelApp.Range["B1"].Value = "Balance";
    excelApp.Range["A2"].Select();

    foreach (var ac in accounts)
    {
        DisplayFunc(ac, excelApp.ActiveCell);
        excelApp.ActiveCell.Offset[1, 0].Select();
    }
    // Copy the results to the Clipboard.
    excelApp.Range["A1:B3"].Copy();
}
```

```
Sub DisplayInExcel(ByVal accounts As IEnumerable(Of Account),
                   ByVal DisplayAction As Action(Of Account, Excel.Range))

    With Me.Application
        ' Add a new Excel workbook.
        .Workbooks.Add()
        .Visible = True
        .Range("A1").Value = "ID"
        .Range("B1").Value = "Balance"
        .Range("A2").Select()

        For Each ac In accounts
            DisplayAction(ac, .ActiveCell)
            .ActiveCell.Offset(1, 0).Select()
        Next

        ' Copy the results to the Clipboard.
        .Range("A1:B3").Copy()
    End With
End Sub
```

Two new C# features are used in this method. Both of these features already exist in Visual Basic.

- Method Add has an *optional parameter* for specifying a particular template. Optional parameters, new in C# 4, enable you to omit the argument for that parameter if you want to use the parameter's default value. Because no argument is sent in the previous example, `Add` uses the default template and creates a new workbook. The equivalent statement in earlier versions of C# requires a placeholder argument: `excelApp.Workbooks.Add(Type.Missing)`.

  For more information, see Named and Optional Arguments.

- The `Range` and `Offset` properties of the Range object use the *indexed properties* feature. This feature enables you to consume these properties from COM types by using the following typical C# syntax. Indexed properties also enable you to use the `Value` property of the `Range` object, eliminating the need to use the `Value2` property. The `Value` property is indexed, but the index is optional. Optional arguments and indexed properties work together in the following example.

  ```
  // Visual C# 2010 provides indexed properties for COM programming.
  excelApp.Range["A1"].Value = "ID";
  excelApp.ActiveCell.Offset[1, 0].Select();
  ```

  In earlier versions of the language, the following special syntax is required.

  ```
  // In Visual C# 2008, you cannot access the Range, Offset, and Value
  // properties directly.
  excelApp.get_Range("A1").Value2 = "ID";
  excelApp.ActiveCell.get_Offset(1, 0).Select();
  ```

  You cannot create indexed properties of your own. The feature only supports consumption of existing indexed properties.

  For more information, see How to: Use Indexed Properties in COM Interop Programming.

2. Add the following code at the end of `DisplayInExcel` to adjust the column widths to fit the content.

```
excelApp.Columns[1].AutoFit();
excelApp.Columns[2].AutoFit();
```

```
' Add the following two lines at the end of the With statement.
.Columns(1).AutoFit()
.Columns(2).AutoFit()
```

These additions demonstrate another feature in C#: treating `Object` values returned from COM hosts such as Office as if they have type dynamic. This happens automatically when **Embed Interop Types** is set to its default value, `True`, or, equivalently, when the assembly is referenced by the /link compiler option. Type `dynamic` allows late binding, already available in Visual Basic, and avoids the explicit casting required in Visual C# 2008 and earlier versions of the language.

For example, `excelApp.Columns[1]` returns an `Object`, and `AutoFit` is an Excel Range method. Without `dynamic`, you must cast the object returned by `excelApp.Columns[1]` as an instance of `Range` before calling method `AutoFit`.

```
// Casting is required in Visual C# 2008.
((Excel.Range)excelApp.Columns[1]).AutoFit();

// Casting is not required in Visual C# 2010.
excelApp.Columns[1].AutoFit();
```

For more information about embedding interop types, see procedures "To find the PIA reference" and "To restore the PIA dependency" later in this topic. For more information about `dynamic`, see dynamic or Using Type dynamic.

To invoke DisplayInExcel

1. Add the following code at the end of the `ThisAddIn_StartUp` method. The call to `DisplayInExcel` contains two arguments. The first argument is the name of the list of accounts to be processed. The second argument is a multiline lambda expression that defines how the data is to be processed. The `ID` and `balance` values for each account are displayed in adjacent cells, and the row is displayed in red if the balance is less than zero. For more information, see Lambda Expressions.

```
DisplayInExcel(bankAccounts, (account, cell) =>
// This multiline lambda expression sets custom processing rules
// for the bankAccounts.
{
    cell.Value = account.ID;
    cell.Offset[0, 1].Value = account.Balance;
    if (account.Balance < 0)
    {
        cell.Interior.Color = 255;
        cell.Offset[0, 1].Interior.Color = 255;
    }
});
```

```
DisplayInExcel(bankAccounts,
       Sub(account, cell)
           ' This multiline lambda expression sets custom
           ' processing rules for the bankAccounts.
           cell.Value = account.ID
           cell.Offset(0, 1).Value = account.Balance

           If account.Balance < 0 Then
               cell.Interior.Color = RGB(255, 0, 0)
               cell.Offset(0, 1).Interior.Color = RGB(255, 0, 0)
           End If
       End Sub)
```

2. To run the program, press F5. An Excel worksheet appears that contains the data from the accounts.

To add a Word document

1. Add the following code at the end of the `ThisAddIn_StartUp` method to create a Word document that contains a link to the Excel workbook.

```
var wordApp = new Word.Application();
wordApp.Visible = true;
wordApp.Documents.Add();
wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
```

```
Dim wordApp As New Word.Application
wordApp.Visible = True
wordApp.Documents.Add()
wordApp.Selection.PasteSpecial(Link:=True, DisplayAsIcon:=True)
```

   This code demonstrates several of the new features in C#: the ability to omit the `ref` keyword in COM programming, named arguments, and optional arguments. These features already exist in Visual Basic. The PasteSpecial method has seven parameters, all of which are defined as optional reference parameters. Named and optional arguments enable you to designate the parameters you want to access by name and to send arguments to only those parameters. In this example, arguments are sent to indicate that a link to the workbook on the Clipboard should be created (parameter `Link`) and that the link is to be displayed in the Word document as an icon (parameter `DisplayAsIcon`). Visual C# also enables you to omit the `ref` keyword for these arguments.

To run the application

1. Press F5 to run the application. Excel starts and displays a table that contains the information from the two accounts in `bankAccounts`. Then a Word document appears that contains a link to the Excel table.

To clean up the completed project

1. In Visual Studio, click **Clean Solution** on the **Build** menu. Otherwise, the add-in will run every time that you open Excel on your computer.

To find the PIA reference

1. Run the application again, but do not click **Clean Solution**.

2. Select the **Start**. Locate **Microsoft Visual Studio <version>** and open a developer command prompt.

3. Type `ildasm` in the Developer Command Prompt for Visual Studio window, and then press ENTER. The IL DASM window appears.

4. On the **File** menu in the IL DASM window, select **File** > **Open**. Double-click **Visual Studio <version>**, and then double-click **Projects**. Open the folder for your project, and look in the bin/Debug folder for *your project name*.dll. Double-click *your project name*.dll. A new window displays your project's attributes, in addition to references to other modules and assemblies. Note that namespaces `Microsoft.Office.Interop.Excel` and `Microsoft.Office.Interop.Word` are included in the assembly. By default in Visual Studio, the compiler imports the types you need from a referenced PIA into your assembly.

   For more information, see How to: View Assembly Contents.

5. Double-click the **MANIFEST** icon. A window appears that contains a list of assemblies that contain items referenced by the project. `Microsoft.Office.Interop.Excel` and `Microsoft.Office.Interop.Word` are not included in the list. Because the types your project needs have been imported into your assembly, references to a PIA are not required. This makes deployment easier. The PIAs do not have to be present on the user's computer, and because an application does not require deployment of a specific version of a PIA,

applications can be designed to work with multiple versions of Office, provided that the necessary APIs exist in all versions.

Because deployment of PIAs is no longer necessary, you can create an application in advanced scenarios that works with multiple versions of Office, including earlier versions. However, this works only if your code does not use any APIs that are not available in the version of Office you are working with. It is not always clear whether a particular API was available in an earlier version, and for that reason working with earlier versions of Office is not recommended.

> **NOTE**
> Office did not publish PIAs before Office 2003. Therefore, the only way to generate an interop assembly for Office 2002 or earlier versions is by importing the COM reference.

6. Close the manifest window and the assembly window.

To restore the PIA dependency

1. In **Solution Explorer**, click the **Show All Files** button. Expand the **References** folder and select **Microsoft.Office.Interop.Excel**. Press F4 to display the **Properties** window.
2. In the **Properties** window, change the **Embed Interop Types** property from **True** to **False**.
3. Repeat steps 1 and 2 in this procedure for `Microsoft.Office.Interop.Word`.
4. In C#, comment out the two calls to `Autofit` at the end of the `DisplayInExcel` method.
5. Press F5 to verify that the project still runs correctly.
6. Repeat steps 1-3 from the previous procedure to open the assembly window. Notice that `Microsoft.Office.Interop.Word` and `Microsoft.Office.Interop.Excel` are no longer in the list of embedded assemblies.
7. Double-click the **MANIFEST** icon and scroll through the list of referenced assemblies. Both `Microsoft.Office.Interop.Word` and `Microsoft.Office.Interop.Excel` are in the list. Because the application references the Excel and Word PIAs, and the **Embed Interop Types** property is set to **False**, both assemblies must exist on the end user's computer.
8. In Visual Studio, click **Clean Solution** on the **Build** menu to clean up the completed project.

See also

- Auto-Implemented Properties (Visual Basic)
- Auto-Implemented Properties (C#)
- Collection Initializers
- Object and Collection Initializers
- Optional Parameters
- Passing Arguments by Position and by Name
- Named and Optional Arguments
- Early and Late Binding
- dynamic
- Using Type dynamic
- Lambda Expressions (Visual Basic)
- Lambda Expressions (C#)
- How to: Use Indexed Properties in COM Interop Programming

- Walkthrough: Embedding Type Information from Microsoft Office Assemblies in Visual Studio (C#)
- Walkthrough: Embedding Types from Managed Assemblies
- Walkthrough: Creating Your First VSTO Add-in for Excel
- COM Interop
- Interoperability

Example COM Class (C# Programming Guide)

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

The following is an example of a class that you would expose as a COM object. After this code has been placed in a .cs file and added to your project, set the **Register for COM Interop** property to **True**. For more information, see How to: Register a Component for COM Interop.

Exposing Visual C# objects to COM requires declaring a class interface, an events interface if it is required, and the class itself. Class members must follow these rules to be visible to COM:

- The class must be public.
- Properties, methods, and events must be public.
- Properties and methods must be declared on the class interface.
- Events must be declared in the event interface.

Other public members in the class that are not declared in these interfaces will not be visible to COM, but they will be visible to other .NET Framework objects.

To expose properties and methods to COM, you must declare them on the class interface and mark them with a `DispId` attribute, and implement them in the class. The order in which the members are declared in the interface is the order used for the COM vtable.

To expose events from your class, you must declare them on the events interface and mark them with a `DispId` attribute. The class should not implement this interface.

The class implements the class interface; it can implement more than one interface, but the first implementation will be the default class interface. Implement the methods and properties exposed to COM here. They must be marked public and must match the declarations in the class interface. Also, declare the events raised by the class here. They must be marked public and must match the declarations in the events interface.

Example

```
using System.Runtime.InteropServices;

namespace project_name
{
    [Guid("EAA4976A-45C3-4BC5-BC0B-E474F4C3C83F")]
    public interface ComClass1Interface
    {
    }

    [Guid("7BD20046-DF8C-44A6-8F6B-687FAA26FA71"),
        InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
    public interface ComClass1Events
    {
    }

    [Guid("0D53A3E8-E51A-49C7-944E-E72A2064F938"),
        ClassInterface(ClassInterfaceType.None),
        ComSourceInterfaces(typeof(ComClass1Events))]
    public class ComClass1 : ComClass1Interface
    {
    }
}
```

See also

- C# Programming Guide
- Interoperability
- Build Page, Project Designer (C#)