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