**Exceptions and Exception Handling**

* 10 minutes to read

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's reasonable to do so, and to clean up resources afterward. Exceptions can be generated by the common language runtime (CLR), by .NET or 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 an exception is thrown, 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.

C#Copy

public class ExceptionTest

{

static double SafeDivision(double x, double y)

{

if (y == 0)

throw new DivideByZeroException();

return x / y;

}

public static void Main()

{

// Input for test purposes. Change the values to see

// exception handling behavior.

double a = 98, b = 0;

double result;

try

{

result = SafeDivision(a, b);

Console.WriteLine("{0} divided by {1} = {2}", a, b, result);

}

catch (DivideByZeroException)

{

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.
* Don't 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 .NET are implemented on top of the Win32 structured exception handling mechanism. For more information, see [Structured Exception Handling (C/C++)](https://docs.microsoft.com/en-us/cpp/cpp/structured-exception-handling-c-cpp) and [A Crash Course on the Depths of Win32 Structured Exception Handling](http://bytepointer.com/resources/pietrek_crash_course_depths_of_win32_seh.htm).

**Use exceptions**

* 05/14/2021
* 3 minutes to read
  + [](https://github.com/BillWagner)

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 runtime 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](https://docs.microsoft.com/en-us/dotnet/api/system.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:

C#Copy

class CustomException : Exception

{

public CustomException(string message)

{

}

}

private static void TestThrow()

{

throw new CustomException("Custom exception in TestThrow()");

}

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:

C#Copy

try

{

TestThrow();

}

catch (CustomException ex)

{

System.Console.WriteLine(ex.ToString());

}

If the statement that throws an exception isn't 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're compatible, are ignored. Order catch blocks from most specific (or most-derived) to least specific. For example:

C#Copy

using System;

using System.IO;

namespace Exceptions

{

public class CatchOrder

{

public static void Main()

{

try

{

using (var sw = new StreamWriter("./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:

C#Copy

static void TestFinally()

{

FileStream? file = null;

//Change the path to something that works on your machine.

FileInfo fileInfo = new System.IO.FileInfo("./file.txt");

try

{

file = fileInfo.OpenWrite();

file.WriteByte(0xF);

}

finally

{

// Closing the file allows you to reopen it immediately - otherwise IOException is thrown.

file?.Close();

}

try

{

file = fileInfo.OpenWrite();

Console.WriteLine("OpenWrite() succeeded");

}

catch (IOException)

{

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() isn't 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, a [TypeInitializationException](https://docs.microsoft.com/en-us/dotnet/api/system.typeinitializationexception) is thrown, with the original exception assigned to the [InnerException](https://docs.microsoft.com/en-us/dotnet/api/system.exception.innerexception) property of the new exception.
* If the start of the thread is reached, the thread is terminated.

# Exception Handling (C# Programming Guide)

* 05/14/2021
* 4 minutes to read
  + [](https://github.com/BillWagner)

A [try](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-catch) block is used by C# programmers to partition code that might be affected by an exception. Associated [catch](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-catch) blocks are used to handle any resulting exceptions. A [finally](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-finally) block contains code that is run 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.

C#Copy

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.

}

C#Copy

try

{

// Code to try goes here.

}

finally

{

// Code to execute after the try block goes here.

}

C#Copy

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](https://docs.microsoft.com/en-us/dotnet/api/system.exception). In general, don't specify [Exception](https://docs.microsoft.com/en-us/dotnet/api/system.exception) as the exception filter unless either you know how to handle all exceptions that might be thrown in the try block, or you've included a [throw](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/throw) statement at the end of your catch block.

Multiple catch blocks with different exception classes 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 class, a catch block that doesn't have any type is selected, if one is present in the statement. It's important to position catch blocks with the most specific (that is, the most derived) exception classes first.

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](https://docs.microsoft.com/en-us/dotnet/api/system.io.filenotfoundexception) object.
* You can create and throw a new, more specific exception.

C#Copy

int GetInt(int[] array, int index)

{

try

{

return array[index];

}

catch (IndexOutOfRangeException e)

{

throw new ArgumentOutOfRangeException(

"Parameter index is out of range.", e);

}

}

* You want to partially handle an exception before passing it on for more handling. In the following example, a catch block is used to add an entry to an error log before rethrowing the exception.

C#Copy

try

{

// Try to access a resource.

}

catch (UnauthorizedAccessException e)

{

// Call a custom error logging procedure.

LogError(e);

// Re-throw the error.

throw;

}

You can also specify exception filters to add a boolean expression to a catch clause. Exception filters indicate that a specific catch clause matches only when that condition is true. In the following example, both catch clauses use the same exception class, but an extra condition is checked to create a different error message:

C#Copy

int GetInt(int[] array, int index)

{

try

{

return array[index];

}

catch (IndexOutOfRangeException e) when (index < 0)

{

throw new ArgumentOutOfRangeException(

"Parameter index cannot be negative.", e);

}

catch (IndexOutOfRangeException e)

{

throw new ArgumentOutOfRangeException(

"Parameter index cannot be greater than the array size.", e);

}

}

An exception filter that always returns false can be used to examine all exceptions but not process them. A typical use is to log exceptions:

C#Copy

public static void Main()

{

try

{

string? s = null;

Console.WriteLine(s.Length);

}

catch (Exception e) when (LogException(e))

{

}

Console.WriteLine("Exception must have been handled");

}

private static bool LogException(Exception e)

{

Console.WriteLine($"\tIn the log routine. Caught {e.GetType()}");

Console.WriteLine($"\tMessage: {e.Message}");

return false;

}

The LogException method always returns false, no catch clause using this exception filter matches. The catch clause can be general, using [System.Exception](https://docs.microsoft.com/en-us/dotnet/api/system.exception), and later clauses can process more specific exception classes.

## 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, 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. For more information See the [using Statement](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/using-statement).

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 can't open the file, the file handle still has the value null and the finally block doesn't try to close it. Instead, if the file is opened successfully in the try block, the finally block closes the open file.

C#Copy

FileStream? file = null;

FileInfo fileinfo = new System.IO.FileInfo("./file.txt");

try

{

file = fileinfo.OpenWrite();

file.WriteByte(0xF);

}

finally

{

// Check for null because OpenWrite might have failed.

file?.Close();

}

**C# Language Specification**

For more information, see [Exceptions](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/exceptions) in the [C# Language Specification](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/introduction). The language specification is the definitive source for C# syntax and usage.

**See also**

* [SystemException](https://docs.microsoft.com/en-us/dotnet/api/system.systemexception)
* [C# Keywords](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/)
* [throw](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/throw)
* [try-catch](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-catch)
* [try-finally](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-finally)
* [try-catch-finally](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-catch-finally)
* [Exceptions](https://docs.microsoft.com/en-us/dotnet/standard/exceptions/)

**Recommended content**

**[Access Modifiers - C# Programming Guide](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers)**

All types and type members in C# have an accessibility level which controls whether they can be used from other code. Review this list of access modifiers.

**[Inheritance in C#](https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/inheritance)**

Learn to use inheritance in C# libraries and applications.

**[Private Constructors - C# Programming Guide](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/private-constructors)**

A private constructor is a special instance constructor in C# used to restrict how an object can be created. They may be used with factory methods or other construction idioms.

**[Knowing When to Use Override and New Keywords - C# Programming Guide](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/knowing-when-to-use-override-and-new-keywords)**

Use the new and override keywords in C# to specify how methods with the same name in a base and derived class interact.

**[Exceptions and Exception Handling - C# Programming Guide](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/exceptions/)**

Learn about exceptions and exception handling. These C# features help deal with unexpected or exceptional situations that happen when a program is running.

**[Static Classes and Static Class Members - C# Programming Guide](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members)**

Static classes cannot be instantiated in C#. You access the members of a static class by using the class name itself.

**[override modifier - C# Reference](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/override)**

override modifier - C# Reference

**[abstract - C# Reference](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/abstract)**

abstract - C# Reference