

Use exceptions


05/14/2021 • 3 minutes to read • 

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](#). 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
<pre>class CustomException : Exception { public CustomException(string message) { } } private static void TestThrow() { throw new CustomException("Custom exception in TestThrow()"); }</pre>	

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
<pre>try { TestThrow(); } catch (CustomException ex)</pre>	

```
{  
    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](#) 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.

Is this page helpful?

 Yes  No

Recommended content

[Methods - C# Guide](#)

Overview of methods, method parameters, and method return values

[this keyword - C# Reference](#)

this keyword (C# Reference)

[static modifier - C# Reference](#)

static modifier - C# Reference

[public keyword - C# Reference](#)

public keyword - C# Reference

Show more 