

# try-catch (C# Reference)

07/20/2015 • 8 minutes to read •  +11

## In this article

[Exceptions in async methods](#)

[Example](#)

[Two catch blocks example](#)

[Async method example](#)

[Task.WhenAll example](#)


[C# language specification](#)

[See also](#)

The try-catch statement consists of a `try` block followed by one or more `catch` clauses, which specify handlers for different exceptions.

When an exception is thrown, the common language runtime (CLR) looks for the `catch` statement that handles this exception. If the currently executing method does not contain such a `catch` block, the CLR looks at the method that called the current method, and so on up the call stack. If no `catch` block is found, then the CLR displays an unhandled exception message to the user and stops execution of the program.

The `try` block contains the guarded code that may cause the exception. The block is executed until an exception is thrown or it is completed successfully. For example, the following attempt to cast a `null` object raises the [NullReferenceException](#) exception:

C#	 Copy
<pre>object o2 = null; try {     int i2 = (int)o2;    // Error }</pre>	

Although the `catch` clause can be used without arguments to catch any type of exception, this usage is not recommended. In general, you should only catch those exceptions that you know how to recover from. Therefore, you should always specify an object argument derived from [System.Exception](#). For example:

C#	 Copy
----	--

```
catch (InvalidCastException e)
{
}
```

It is possible to use more than one specific `catch` clause in the same try-catch statement. In this case, the order of the `catch` clauses is important because the `catch` clauses are examined in order. Catch the more specific exceptions before the less specific ones. The compiler produces an error if you order your catch blocks so that a later block can never be reached.

Using `catch` arguments is one way to filter for the exceptions you want to handle. You can also use an exception filter that further examines the exception to decide whether to handle it. If the exception filter returns false, then the search for a handler continues.

C#

 Copy

```
catch (ArgumentException e) when (e.ParamName == "...")
{
}
```

Exception filters are preferable to catching and rethrowing (explained below) because filters leave the stack unharmed. If a later handler dumps the stack, you can see where the exception originally came from, rather than just the last place it was rethrown. A common use of exception filter expressions is logging. You can create a filter that always returns false that also outputs to a log, you can log exceptions as they go by without having to handle them and rethrow.


A `throw` statement can be used in a `catch` block to re-throw the exception that is caught by the `catch` statement. The following example extracts source information from an [IOException](#) exception, and then throws the exception to the parent method.

C#


 Copy

```
catch (FileNotFoundException e)
{
    // FileNotFoundExceptions are handled here.
}
catch (IOException e)
{
    // Extract some information from this exception, and then
    // throw it to the parent method.
    if (e.Source != null)
        Console.WriteLine("IOException source: {0}", e.Source);
    throw;
}
```

You can catch one exception and throw a different exception. When you do this, specify the exception that you caught as the inner exception, as shown in the following example.


C#	 Copy
<pre>catch (InvalidCastException e) {     // Perform some action here, and then throw a new exception.     throw new YourCustomException("Put your error message here.", e); }</pre>	

You can also re-throw an exception when a specified condition is true, as shown in the following example.

C#	 Copy
<pre>catch (InvalidCastException e) {     if (e.Data == null)     {         throw;     }     else     {         // Take some action.     } }</pre>	


#### Note

It is also possible to use an exception filter to get a similar result in an often cleaner fashion (as well as not modifying the stack, as explained earlier in this document). The following example has a similar behavior for callers as the previous example. The function throws the `InvalidCastException` back to the caller when `e.Data` is `null`.

C#	 Copy
<pre>catch (InvalidCastException e) when (e.Data != null) {     // Take some action. }</pre>	

From inside a `try` block, initialize only variables that are declared therein. Otherwise, an exception can occur before the execution of the block is completed. For example, in the

following code example, the variable `n` is initialized inside the `try` block. An attempt to use this variable outside the `try` block in the `Write(n)` statement will generate a compiler error.

C#	 Copy
<pre>static void Main() {     int n;     try     {         // Do not initialize this variable here.         n = 123;     }     catch     {     }     // Error: Use of unassigned local variable 'n'.     Console.Write(n); }</pre>	

For more information about catch, see [try-catch-finally](#).

## Exceptions in async methods

An async method is marked by an `async` modifier and usually contains one or more await expressions or statements. An await expression applies the `await` operator to a `Task` or `Task<TResult>`.

When control reaches an `await` in the async method, progress in the method is suspended until the awaited task completes. When the task is complete, execution can resume in the method. For more information, see [Asynchronous programming with async and await](#).


The completed task to which `await` is applied might be in a faulted state because of an unhandled exception in the method that returns the task. Awaiting the task throws an exception. A task can also end up in a canceled state if the asynchronous process that returns it is canceled. Awaiting a canceled task throws an `OperationCanceledException`.

To catch the exception, await the task in a `try` block, and catch the exception in the associated `catch` block. For an example, see the [Async method example](#) section.

A task can be in a faulted state because multiple exceptions occurred in the awaited async method. For example, the task might be the result of a call to `Task.WhenAll`. When you await such a task, only one of the exceptions is caught, and you can't predict which exception will be caught. For an example, see the [Task.WhenAll example](#) section.

# Example

In the following example, the `try` block contains a call to the `ProcessString` method that may cause an exception. The `catch` clause contains the exception handler that just displays a message on the screen. When the `throw` statement is called from inside `ProcessString`, the system looks for the `catch` statement and displays the message `Exception caught`.

C#	 Copy
<pre>class TryFinallyTest {     static void ProcessString(string s)     {         if (s == null)         {             throw new ArgumentNullException();         }     }      public static void Main()     {         string s = null; // For demonstration purposes.          try         {             ProcessString(s);         }         catch (Exception e)         {             Console.WriteLine("{0} Exception caught.", e);         }     } } /* Output: System.ArgumentNullException: Value cannot be null.    at TryFinallyTest.Main() Exception caught. * */</pre>	

## Two catch blocks example

In the following example, two catch blocks are used, and the most specific exception, which comes first, is caught.

To catch the least specific exception, you can replace the `throw` statement in `ProcessString` with the following statement: `throw new Exception()`.

If you place the least-specific catch block first in the example, the following error message appears: A previous catch clause already catches all exceptions of this or a super type ('System.Exception').

C#  Copy

```
class ThrowTest3
{
    static void ProcessString(string s)
    {
        if (s == null)
        {
            throw new ArgumentNullException();
        }
    }

    public static void Main()
    {
        try
        {
            string s = null;
            ProcessString(s);
        }
        // Most specific:
        catch (ArgumentNullException e)
        {
            Console.WriteLine("{0} First exception caught.", e);
        }
        // Least specific:
        catch (Exception e)
        {
            Console.WriteLine("{0} Second exception caught.", e);
        }
    }
}
/*
Output:
System.ArgumentNullException: Value cannot be null.
at Test.ThrowTest3.ProcessString(String s) ... First exception caught.
*/
```

## Async method example

The following example illustrates exception handling for async methods. To catch an exception that an async task throws, place the `await` expression in a `try` block, and catch the exception in a `catch` block.

Uncomment the `throw new Exception` line in the example to demonstrate exception handling. The task's `IsFaulted` property is set to `True`, the task's

Exception.InnerException property is set to the exception, and the exception is caught in the catch block.

Uncomment the `throw new OperationCanceledException` line to demonstrate what happens when you cancel an asynchronous process. The task's `IsCanceled` property is set to `true`, and the exception is caught in the catch block. Under some conditions that don't apply to this example, the task's `IsFaulted` property is set to `true` and `IsCanceled` is set to `false`.

C#

 Copy

```
public async Task DoSomethingAsync()
{
    Task<string> theTask = DelayAsync();

    try
    {
        string result = await theTask;
        Debug.WriteLine("Result: " + result);
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Exception Message: " + ex.Message);
    }
    Debug.WriteLine("Task IsCanceled: " + theTask.IsCanceled);
    Debug.WriteLine("Task IsFaulted: " + theTask.IsFaulted);
    if (theTask.Exception != null)
    {
        Debug.WriteLine("Task Exception Message: "
            + theTask.Exception.Message);
        Debug.WriteLine("Task Inner Exception Message: "
            + theTask.Exception.InnerException.Message);
    }
}

private async Task<string> DelayAsync()
{
    await Task.Delay(100);

    // Uncomment each of the following lines to
    // demonstrate exception handling.

    //throw new OperationCanceledException("canceled");
    //throw new Exception("Something happened.");
    return "Done";
}

// Output when no exception is thrown in the awaited method:
//   Result: Done
//   Task IsCanceled: False
//   Task IsFaulted: False

// Output when an Exception is thrown in the awaited method:
//   Exception Message: Something happened.
```

```
// Task IsCanceled: False
// Task IsFaulted: True
// Task Exception Message: One or more errors occurred.
// Task Inner Exception Message: Something happened.

// Output when a OperationCanceledException or TaskCanceledException
// is thrown in the awaited method:
// Exception Message: canceled
// Task IsCanceled: True
// Task IsFaulted: False
```

## Task.WhenAll example

The following example illustrates exception handling where multiple tasks can result in multiple exceptions. The `try` block awaits the task that's returned by a call to [Task.WhenAll](#). The task is complete when the three tasks to which `WhenAll` is applied are complete.

Each of the three tasks causes an exception. The `catch` block iterates through the exceptions, which are found in the `Exception.InnerExceptions` property of the task that was returned by [Task.WhenAll](#).

C#

 Copy

```
public async Task DoMultipleAsync()
{
    Task theTask1 = ExcAsync(info: "First Task");
    Task theTask2 = ExcAsync(info: "Second Task");
    Task theTask3 = ExcAsync(info: "Third Task");

    Task allTasks = Task.WhenAll(theTask1, theTask2, theTask3);

    try
    {
        await allTasks;
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Exception: " + ex.Message);
        Debug.WriteLine("Task IsFaulted: " + allTasks.IsFaulted);
        foreach (var inEx in allTasks.Exception.InnerExceptions)
        {
            Debug.WriteLine("Task Inner Exception: " + inEx.Message);
        }
    }
}

private async Task ExcAsync(string info)
{
    await Task.Delay(100);
```



```
    throw new Exception("Error-" + info);  
}  
  
// Output:  
//   Exception: Error-First Task  
//   Task IsFaulted: True  
//   Task Inner Exception: Error-First Task  
//   Task Inner Exception: Error-Second Task  
//   Task Inner Exception: Error-Third Task
```

## C# language specification

For more information, see [The try statement](#) section of the [C# language specification](#).

## See also

- [C# Reference](#)
- [C# Programming Guide](#)
- [C# Keywords](#)
- [try, throw, and catch Statements \(C++\)](#)
- [throw](#)
- [try-finally](#)
- [How to: Explicitly Throw Exceptions](#)

---

## Is this page helpful?

 Yes  No

---