## **try Statements and Exceptions**

In programming, unexpected situations can arise during execution, such as trying to divide by zero, accessing a file that doesn't exist, or running out of memory. These situations are called **exceptions**. C# provides a structured mechanism to deal with these exceptions using try statements, catch blocks, and finally blocks.

* A **try block** encloses the code that might throw an exception.
* A **catch block** handles specific types of exceptions that are thrown within the try block. You can have one or more catch blocks.
* A **finally block** contains cleanup code that *always* executes, regardless of whether an exception was thrown or caught.

A try statement must always be followed by at least one catch block or a finally block, or both.

The basic structure looks like this:

|  |
| --- |
| try {  // Code that might throw an exception } catch (ExceptionA ex) // Catches a specific type of exception (ExceptionA) {  // Handle ExceptionA } catch (ExceptionB ex) // Catches another specific type of exception (ExceptionB) {  // Handle ExceptionB } finally {  // Cleanup code (always executes) } |

### **Example: Handling Division by Zero**

Consider a simple Calc method that performs division:

|  |
| --- |
| int Calc(int x) => 10 / x;  // ... // int y = Calc(0); // This would cause a DivideByZeroException and crash the program // Console.WriteLine(y); |

To prevent the program from crashing, we can wrap the call in a try-catch block:

|  |
| --- |
| int Calc(int x) => 10 / x;  // ... try {  int y = Calc(0); // This line will throw DivideByZeroException  Console.WriteLine(y); } catch (DivideByZeroException ex) // Catching the specific exception type {  Console.WriteLine("Error: x cannot be zero.");  // 'ex' contains information about the exception } Console.WriteLine("Program completed."); |

**Output:**

|  |
| --- |
| Error: x cannot be zero. Program completed. |

In this example, instead of terminating, the program catches the DivideByZeroException, prints an error message, and then continues execution.

**Important Note:** While try-catch is powerful, it's generally preferable to prevent errors with explicit checks when possible. For instance, before calling Calc(x), you could check if (x == 0). Exceptions are relatively expensive in terms of performance, so they should be reserved for truly exceptional and unpredictable situations, not for expected validation.

### **How Exception Handling Works**

When an exception is thrown inside a try block:

1. The Common Language Runtime (CLR) looks for a compatible catch block immediately following the try block.
   * If a compatible catch block is found, execution jumps to that catch block. After the catch block finishes, the finally block (if present) executes, and then normal program flow resumes.
   * If no compatible catch block is found, the finally block (if present) executes. Then, the CLR walks up the call stack (to the method that called the current method, and so on) looking for other try blocks with compatible catch handlers. This process repeats until a handler is found.
2. If no function in the entire call stack takes responsibility for the exception (i.e., no compatible catch block is found anywhere), the program terminates.

## **The catch Clause**

A catch clause specifies the type of exception it can handle. This type must be System.Exception or a class derived from it.

* **Catching System.Exception:** This catches *all* possible errors. This is typically done in:  
  + **Recovery Scenarios:** If your program can genuinely recover regardless of the specific error.
  + **Logging and Rethrowing:** To log an error before rethrowing it (perhaps as a different exception type).
  + **Last Resort Handlers:** As a final safety net at the top level of your application to prevent crashes.
* **Catching Specific Exception Types:** More commonly, you catch specific exception types (e.g., DivideByZeroException, FormatException) to handle known error conditions precisely. This avoids catching unexpected errors that your handler isn't designed for (like OutOfMemoryException).

You can have **multiple catch clauses** to handle different exception types. The CLR evaluates them in order, from top to bottom. Only **one catch clause executes** for a given exception. Therefore, you must place more specific exception handlers *before* more general ones (e.g., DivideByZeroException before System.Exception).

|  |
| --- |
| static void Main(string[] args) {  try  {  byte b = byte.Parse(args[0]); // args[0] might not exist, or not be a number, or be too large  Console.WriteLine(b);  }  catch (IndexOutOfRangeException) // More specific  {  Console.WriteLine("Please provide at least one argument.");  }  catch (FormatException) // More specific  {  Console.WriteLine("That's not a number!");  }  catch (OverflowException) // More specific  {  Console.WriteLine("You've given me more than a byte!");  }  catch (Exception ex) // General fallback (must be last)  {  Console.WriteLine($"An unexpected error occurred: {ex.Message}");  } } |

You can also catch an exception **without specifying a variable** if you don't need to access its properties:

|  |
| --- |
| catch (OverflowException) { /\* ... \*/ } |

You can even **omit both the variable and the type** to catch all exceptions (this is equivalent to catch (System.Exception) without a variable):

|  |
| --- |
| catch { /\* ... \*/ } // Catches all exceptions |

### **Exception Filters (when clause)**

You can add an **exception filter** to a catch clause using the when keyword. This allows you to catch an exception *only if* a specified boolean condition is true.

|  |
| --- |
| catch (WebException ex) when (ex.Status == WebExceptionStatus.Timeout) {  Console.WriteLine("Web request timed out."); } catch (WebException ex) // This catch will only be considered if the above filter was false {  Console.WriteLine($"Another WebException occurred: {ex.Status}"); } |

If a WebException is thrown, the when condition is evaluated. If it's false, that catch block is skipped, and the CLR continues to look for other compatible catch clauses. The boolean expression in when can even have side effects, such as logging the exception.

## **The finally Block**

A finally block guarantees that its code will execute, irrespective of how the try block is exited. This makes it ideal for **cleanup tasks** that must always happen, such as closing file handles, database connections, or releasing resources.

A finally block executes in the following scenarios:

* After a catch block finishes (or throws a new exception).
* After the try block finishes normally.
* After the try block throws an exception for which there is no compatible catch block.
* If control leaves the try block due to a jump statement (e.g., return, break, goto).

The only situations that can prevent a finally block from executing are an infinite loop within the try or catch block, or the process being abruptly terminated (e.g., by the operating system).

**Example: Ensuring a File is Closed**

|  |
| --- |
| void ReadFile() {  StreamReader reader = null; // Declare outside try for finally block access  try  {  reader = File.OpenText("file.txt");  if (reader.EndOfStream) return; // Returns early, but finally still executes  Console.WriteLine(reader.ReadToEnd());  }  finally  {  if (reader != null)  reader.Dispose(); // Ensures the file stream is closed  } } |

In this example, reader.Dispose() is called even if an IOException occurs during ReadToEnd() or if the method returns early.

### **The using Statement and IDisposable**

Many classes that manage unmanaged resources (like file handles, network connections) implement the System.IDisposable interface, which has a single method: Dispose(). The using statement in C# provides a syntactic sugar for ensuring Dispose() is called on such objects within a finally block.

|  |
| --- |
| // The using statement: using (StreamReader reader = File.OpenText("file.txt")) // Reader is automatically disposed {  // Use reader here  Console.WriteLine(reader.ReadToEnd()); } // At the end of this block, reader.Dispose() is implicitly called |

This is precisely equivalent to the try-finally block shown above, making code cleaner and safer.

### **using Declarations (C# 8+)**

If you omit the curly braces and statement block after a using statement, it becomes a **using declaration**. The resource will be disposed when execution leaves the *enclosing scope* (the block in which the using declaration is made).

|  |
| --- |
| if (File.Exists("file.txt")) {  using var reader = File.OpenText("file.txt"); // reader will be disposed when leaving the 'if' block  Console.WriteLine(reader.ReadLine());  // ... more code using reader } // reader.Dispose() is called here |

## **Throwing Exceptions**

Exceptions can be thrown by the CLR (as seen with DivideByZeroException) or explicitly by your own code using the throw keyword.

|  |
| --- |
| void Display(string name) {  if (name == null)  throw new ArgumentNullException(nameof(name), "Name cannot be null."); // Throw an exception  Console.WriteLine(name); }  // ... try { Display(null); } catch (ArgumentNullException ex) {  Console.WriteLine($"Caught the exception: {ex.Message}"); } |

**ArgumentNullException.ThrowIfNull (from .NET 6):** For common null checks, a concise helper exists:

|  |
| --- |
| void Display(string name) {  ArgumentNullException.ThrowIfNull(name); // Throws ArgumentNullException if name is null  Console.WriteLine(name); } |

### **throw Expressions (C# 7+)**

The throw keyword can also be used as an expression, allowing it in places like expression-bodied members or ternary conditional expressions:

|  |
| --- |
| public string Foo() => throw new NotImplementedException(); // Throw as an expression  string ProperCase(string value) =>  value == null ? throw new ArgumentException("Value cannot be null.") : // Throw in ternary  value == "" ? "" :  char.ToUpper(value[0]) + value.Substring(1); |

### **Rethrowing an Exception**

You can catch an exception, perform some actions (like logging), and then **rethrow** the *original* exception.

|  |
| --- |
| try {  // Some risky operation } catch (Exception ex) {  // Log the error: ex.Message, ex.StackTrace, etc.  Console.WriteLine($"Logged error: {ex.Message}");  throw; // Rethrows the ORIGINAL exception, preserving its stack trace } |

**Important:** Using throw; (without ex) is crucial for preserving the original stack trace of the exception. If you use throw ex;, the stack trace will reset to the point where throw ex; was called, losing valuable debugging information about the original source of the error.

You can also rethrow a **new, more specific exception** while wrapping the original one:

|  |
| --- |
| try {  // Parse a DateTime from XML element data } catch (FormatException ex) {  // Wrap the original exception for debugging purposes  throw new XmlException("Invalid DateTime format in XML.", ex); } |

Passing the original ex to the new exception's constructor populates the InnerException property, forming a chain of exceptions that helps trace the root cause. This is a best practice.

## **Key Properties of System.Exception**

All exceptions in C# derive from System.Exception. Key properties for debugging and handling include:

* **StackTrace**: A string showing the sequence of method calls leading up to where the exception was thrown. Essential for debugging.
* **Message**: A string describing the error in detail.
* **InnerException**: If an exception was rethrown and wrapped another exception, this property holds the original, inner exception. This can form a chain of exceptions.

**Note:** C# has only **runtime exceptions** (unchecked exceptions). There is no equivalent to Java's compile-time "checked exceptions" that force you to declare what exceptions a method might throw.

## **Common Exception Types**

You'll frequently encounter and potentially throw these exception types:

* **System.ArgumentException**: General exception for an invalid argument passed to a method. Usually indicates a bug in the calling code.
* **System.ArgumentNullException**: A subclass of ArgumentException, specifically for null arguments where null is not allowed.
* **System.ArgumentOutOfRangeException**: Another subclass of ArgumentException, for when an argument (often numeric) is outside an acceptable range.
* **System.InvalidOperationException**: Thrown when an object's state is invalid for a particular operation (e.g., trying to read from a closed file). The arguments might be valid, but the object itself isn't ready.
* **System.NotSupportedException**: Indicates that a particular operation or feature is not implemented or supported (e.g., calling Add on a read-only collection).
* **System.NotImplementedException**: Indicates that a method or property has not yet been implemented (often used as a placeholder in development).
* **System.ObjectDisposedException**: Thrown when you try to use an object that has already been disposed.
* **System.NullReferenceException**: Thrown by the CLR when you attempt to access a member (field, property, method) of an object whose value is null. This nearly always indicates a programming error.

## **The TryXXX Method Pattern**

For operations where failure is a common or expected outcome (rather than an exceptional one), it's often useful to offer two versions of a method:

* A **throwing method** (e.g., Parse): Throws an exception on failure.
* A **TryXXX method** (e.g., TryParse): Returns a bool indicating success/failure and an out parameter for the result.

**Example: int.Parse vs. int.TryParse**

|  |
| --- |
| // Throws an exception if parsing fails int number1 = int.Parse("123"); // OK // int number2 = int.Parse("abc"); // Throws FormatException  // Returns true/false and provides result via 'out' parameter if (int.TryParse("123", out int result1)) {  Console.WriteLine($"Parsed: {result1}"); // Output: Parsed: 123 }  if (!int.TryParse("abc", out int result2)) {  Console.WriteLine("Failed to parse 'abc'."); // Output: Failed to parse 'abc'. } |

This pattern is ideal when failures are part of the normal program flow and can be handled without resorting to try-catch blocks, which are better for truly exceptional situations.