Chapter 4 Exception Handling

Exceptions:

An exception is a problem that arises during the execution of a program. A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C# exception handling is built upon four keywords: **try**, **catch**, **finally**, and **throw**.

- **try** − A try block identifies a block of code for which particular exceptions is activated. It is followed by one or more catch blocks.
- **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
- **finally** − The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.
- **throw** − A program throws an exception when a problem shows up. This is done using a throw keyword.

Syntax

Assuming a block raises an exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following −

```
try {
   // statements causing exception
} catch( ExceptionName e1 ) {
   // error handling code
} catch( ExceptionName e2 ) {
   // error handling code
} catch( ExceptionName eN ) {
   // error handling code
} finally {
   // statements to be executed
}
```

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

Exception Classes in C#:

C# exceptions are represented by classes. The exception classes in C# are mainly directly or indirectly derived from the **System.Exception** class. Some of the exception classes derived from the System.Exception class are the **System.ApplicationException** and **System.SystemException** classes.

The **System.ApplicationException** class supports exceptions generated by application programs. Hence the exceptions defined by the programmers should derive from this class.

The **System.SystemException** class is the base class for all predefined system exception.

The following table provides some of the predefined exception classes derived from the Sytem.SystemException class −

Sr.No.	Exception Class & Description
1	**System.IO.IOException** Handles I/O errors.
2	**System.IndexOutOfRangeException** Handles errors generated when a method refers to an array index out of range.
3	**System.ArrayTypeMismatchException** Handles errors generated when type is mismatched with the array type.
4	**System.NullReferenceException** Handles errors generated from referencing a null object.
5	**System.DivideByZeroException** Handles errors generated from dividing a dividend with zero.
6	**System.InvalidCastException** Handles errors generated during typecasting.
7	**System.OutOfMemoryException** Handles errors generated from insufficient free memory.
8	**System.StackOverflowException** Handles errors generated from stack overflow.

When executing C# code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C# will normally stop and generate an error message. The technical term for this is: C# will throw an **exception** (throw an error).

try and catch:

The `try` statement allows you to define a block of code to be tested for errors while it is being executed.

The `catch` statement allows you to define a block of code to be executed, if an error occurs in the try block.

The `try` and `catch` keywords come in pairs:

Syntax

```
try
{
  //  Block of code to try
}
```

```
catch (Exception e)
{
  //  Block of code to handle errors
}
```

Consider the following example, where we create an array of three integers:

This will generate an error, because **myNumbers[10]** does not exist.

```
int[] myNumbers = {1, 2, 3};
Console.WriteLine(myNumbers[10]); // error!
```

The error message will be something like this:

```
System.IndexOutOfRangeException: 'Index was outside the bounds of the array.'
```

If an error occurs, we can use `try...catch` to catch the error and execute some code to handle it.

In the following example, we use the variable inside the catch block (`e`) together with the built-in `Message` property, which outputs a message that describes the exception:

Example

```
try
{
  int[] myNumbers = {1, 2, 3};
  Console.WriteLine(myNumbers[10]);
}
catch (Exception e)
{
  Console.WriteLine(e.Message);
}
```

The output will be:

```
Index was outside the bounds of the array.
```

You can also output your own error message:

Example

```
try
{
  int[] myNumbers = {1, 2, 3};
  Console.WriteLine(myNumbers[10]);
}
catch (Exception e)
{
  Console.WriteLine("Something went wrong.");
}
```

The output will be:

```
Something went wrong.
```

Finally:

The `finally` statement lets you execute code, after `try...catch`, regardless of the result:

Example

```
try
{
  int[] myNumbers = {1, 2, 3};
  Console.WriteLine(myNumbers[10]);
}
catch (Exception e)
{
  Console.WriteLine("Something went wrong.");
}
finally
{
  Console.WriteLine("The 'try catch' is finished.");
}
```

The output will be:

```
Something went wrong.
The 'try catch' is finished.
```

The throw keyword:

The `throw` statement allows you to create a custom error.

The `throw` statement is used together with an **exception class**. There are many exception classes available in C#: `ArithmeticException`, `FileNotFoundException`, `IndexOutOfRangeException`, `TimeOutException`, etc:

Example

```
static void checkAge(int age)
{
  if (age < 18)
  {
    throw new ArithmeticException("Access denied - You must be at least 18 years old.");
  }
  else
  {
    Console.WriteLine("Access granted - You are old enough!");
  }
}

static void Main(string[] args)
{
  checkAge(15);
}
```

The error message displayed in the program will be:

```
System.ArithmeticException: 'Access denied - You must be at least 18 years old.'
```

Multiple Catch Blocks:

The main purpose of the catch block is to handle the exception raised in the try block. This block is only going to execute when the exception raised in the program.
In C#, You can use more than one catch block with the try block. Generally, multiple catch block is used

to handle different types of exceptions means each catch block is used to handle different type of exception. If you use multiple catch blocks for the same type of exception, then it will give you a compile-time error because **C# does not allow you to use multiple catch block for the same type of exception**. A catch block is always preceded by the try block.

In general, the catch block is checked within the order in which they have occurred in the program. If the given type of exception is matched with the first catch block, then first catch block executes and the remaining of the catch blocks are ignored. And if the starting catch block is not suitable for the exception type, then compiler search for the next catch block.

Syntax:

```
try {
// Your code
}
// 1st catch block
catch(Exception_Name) {
// Code
}
// 2nd catch block
catch(Exception_Name) {
// Code
}
```

Below given are some examples to understand the implementation in a better way:

Example 1: In the below example, try block generate two different types of exception i.e **DivideByZeroException** and **IndexOutOfRangeException**. Now we use two catch blocks to handle these exceptions that are associated with a single try block. Each catch block caught a different type of exception like catch block 1 is used to catch DivideByZeroException, catch block 2 is used to catch IndexOutOfRangeException.

```
// C# program to illustrate the
// use of multiple catch block
using System;
 class GFG {
     // Main Method
    static void Main()
    {
         // Here, number is greater than divisor
        int[] number = { 8, 17, 24, 5, 25 };
        int[] divisor = { 2, 0, 0, 5 };
         // --------- try block ---------
         for (int j = 0; j < number.Length; j++)
             // Here this block raises two different
           // types of exception, i.e. DivideByZeroException
           // and IndexOutOfRangeException
           try {
                 Console.WriteLine("Number: " + number[j]);
               Console.WriteLine("Divisor: " + divisor[j]);
               Console.WriteLine("Quotient: " + number[j] / divisor[j]);
           }
             // Catch block 1
             // This Catch block is for
           // handling DivideByZeroException
           catch (DivideByZeroException) {
                 Console.WriteLine("Not possible to Divide by zero");
```

```
            }
              // Catch block 2
              // This Catch block is for
            // handling IndexOutOfRangeException
            catch (IndexOutOfRangeException) {
                Console.WriteLine("Index is Out of Range");
            } }}
```

Output:

```
Number: 8
Divisor: 2
Quotient: 4
Number: 17
Divisor: 0
Not possible to Divide by zero
Number: 24
Divisor: 0
Not possible to Divide by zero
Number: 5
Divisor: 5
Quotient: 1
Number: 25
Index is Out of Range
```

Example 2: In the below example, try block raise an exception. So we will use three different type of catch blocks to handle the exception raised by the try block. Catch block 1 will handle **IndexOutOfRangeException**, catch block 2 will handle **FormatException**, and catch block 3 will handle **OverflowException**.

```
// C# program to illustrate the concept
// of multiple catch clause
using System;
class GFG {
    // Main method
    static void Main()
    {
        // This block raises an exception
        try {
            byte data = byte.Parse("a");
            Console.WriteLine(data);
        }
        // Catch block 1
        // This block is used to handle
        // IndexOutOfRangeException type exception
        catch (IndexOutOfRangeException) {
            Console.WriteLine("At least provide one Argument!");
        }
        // Catch block 2
        // This block is used to handle
        // FormatException type exception
        catch (FormatException) {
             Console.WriteLine("Entered value in not a number!");
       }
          // Catch block 3
          // This block is used to handle
        // OverflowException type exception
        catch (OverflowException) {
              Console.WriteLine("Data is out of Range!");
        } }}
```

Output:

```
Entered value in not a number!
```