Exception Handling in C#

**Exception Handling in C# with Examples**

In this article, I am going to discuss **Exception Handling in C#** with Examples. This is one of the most important concepts in C#. As a developer, while developing an application, it is your key responsibility to handle the exception. The C# Exception Handling is a procedure to handle the exception which occurred during the execution of a program. As part of this article, we are going to discuss the following pointers in detail.

1. **What are the different types of errors?**
2. **What is an Exception in C#?**
3. **Who is responsible for abnormal termination of the program whenever runtime errors occur in the program?**
4. **What happens if an exception is raised in the program?**
5. **What CLR does when an exception occurred in the program?**
6. **What is exception handling in C#?**
7. **Why do we need Exception Handling in C#?**
8. **What is the procedure to Handle Exception in C#?**

**Types of Errors in C#**

When we write and execute our code in the .NET framework then there is a possibility of two types of error occurrences they are

1. **Compilation errors**
2. **Runtime errors**

**Compilation Error in C#**

The error that occurs in a program at the time of compilation is known as compilation error (compile-time error). These errors occur due to the syntactical mistakes under the program. That means these errors occur by typing the wrong syntax like missing double quotes and terminators, typing wrong spelling for keywords, assigning wrong data to a variable, trying to create an object for abstract class and interface, etc.

So in simple words, we can say that this type of error occurs due to a poor understanding of the programming language. These errors can be identified by the programmer and can be rectified before the execution of the program only. So these errors do not cause any harm to the program execution.

**Runtime Error in C#**

The errors which are occurred at the time of program execution are called the runtime error. These errors occurred when we are entering wrong data into a variable, trying to open a file for which there is no permission, trying to connect to the database with the wrong user id and password, the wrong implementation of logic, missing required resources, etc.

Runtime errors are dangerous because whenever they occur in the program, the program terminates abnormally on the same line where the error gets occurred without executing the next line of code.

**What is an Exception in C#?**

A runtime error is known as an exception in C#. The exception will cause the abnormal termination of the program execution. So these errors (exceptions) are very dangerous because whenever the exception occurs in the programs, the program gets terminated abnormally on the same line where the error gets occurred without executing the next line of code.

**Who is responsible for abnormal termination of the program whenever runtime errors occur?**

Objects of exception classes are responsible for abnormal termination of the program whenever runtime errors (exceptions) occur. These exception classes are predefined under BCL (Base Class Libraries) where a separate class is provided for each and every different type of exception like

1. **IndexOutOfRangeException**
2. **FormatException**
3. **NullReferenceException**
4. **DivideByZeroException**
5. **FileNotFoundException**
6. **SQLException,**
7. **OverFlowException, etc.**

Each exception class provides a specific exception error message. All the above exception classes are responsible for abnormal termination of the program as well as after abnormal termination of the program they will be displaying an error message which specifies the reason for abnormal termination i.e. they provide an error message specific to that error.

So, whenever a runtime error (exception) occurs in a program, first the exception manager under the CLR (Common Language Runtime) identifies the type of error that occurs in the program, then creates an object of the exception class related to that error and throws that object which will immediately terminate the program abnormally on the line where error got occur and display the error message related to that class.

**What happens if an exception is raised in the program in C#?**

When an Exception is raised in C#, the program execution is terminated abnormally. That means the statements placed after the exception-causing statements are not executed but the statements placed before that exception-causing statement are executed by CLR.

**What CLR does when an exception occurred in the program?**

It creates the exception class object that is associated with that logical mistake (exception) and terminates the current method execution by throwing that exception object by using the “throw” keyword. So we can say an exception is an event that occurs during the execution of a program that disrupts the normal flow of instruction execution. Let’s understand this with an example.

**Example: Program Execution without Exception in C#**

The following example shows program execution without exception. This is a very simple program, We are just dividing two numbers and printing the result on the console.

**namespace** *ExceptionHandlingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int** a = 20;

**int** b = 10;

**int** c;

Console.WriteLine**(**"A VALUE = " + a**)**;

Console.WriteLine**(**"B VALUE = " + b**)**;

c = a / b;

Console.WriteLine**(**"C VALUE = " + c**)**;

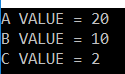
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Example: Program Execution with Exception in C#**

The following example shows program execution with an exception. As you can see, in the below code, we are dividing an integer number with 0 which is not possible in mathematics. So, it will through Divide By Zero Exception in this case. The statements which are present before the exception-causing statement i.e. before c = a / b; is executed and the statements which are present after the exception-causing statement will not be executed.

**namespace** *ExceptionHandlingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int** a = 20;

**int** b = 0;

**int** c;

Console.WriteLine**(**"A VALUE = " + a**)**;

Console.WriteLine**(**"B VALUE = " + b**)**;

c = a / b;

Console.WriteLine**(**"C VALUE = " + c**)**;

Console.ReadKey**()**;

**}**

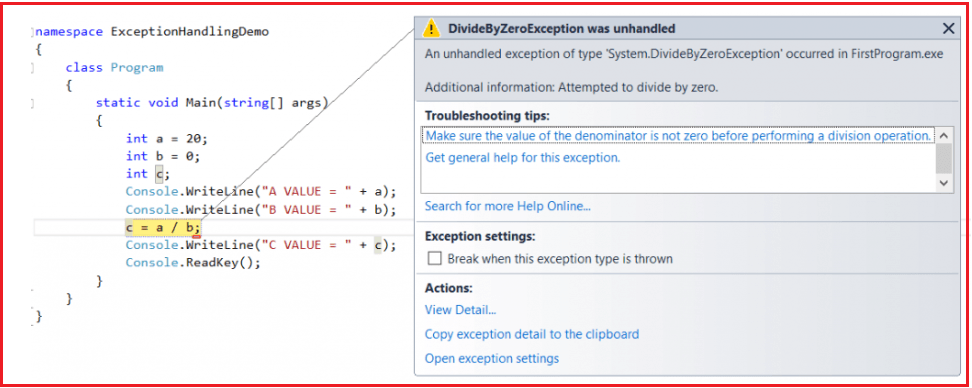
**}**

**}**

**OUTPUT:**

Who is responsible for abnormal termination of the program whenever runtime errors occur in the program?

**After printing the above value it will give us the below error.**



**Explanation:**

The CLR terminates the program execution by throwing **DivideByZeroException** because the logical mistake we committed here is dividing an integer number by integer zero. As we know it is not possible to divide an integer number by zero. From the above program, we can define the exception technically as

1. An exception is an event because when an exception is raised CLR internally executes some logic to prepare that exception-related messages.
2. The Exception is a signal because by looking into the exception message developer will take necessary actions against that exception.

**Is the above exception message is user understandable?**

Definitely, the answer is no. The user cannot understand the above exception message because they are .NET-based exception messages. So the user cannot take any decision alone to resolve the above problem. A developer should guide to solve the above problem.

**What is the solution to the above problem?**

It is the developer’s responsibility to convert .NET exception messages into user-understandable message formats. To solve this problem developer should handle the exception. Using the exception handling mechanism, the developer can catch the exception and can print and display user understandable messages.

**What is exception handling in C#?**

The process of catching the exception for converting the CLR given exception message to an end-user understandable message and for stopping the abnormal termination of the program whenever runtime errors are occurring is called Exception Handling in C#. Once we handle an exception under a program we will be getting the following advantages

1. We can stop the abnormal termination
2. We can perform any corrective action that may resolve the problem.
3. Displaying a user-friendly error message, so that the user can resolve the problem provided if it is under his control.

**Why do we need Exception Handling in C#?**

We need Exception Handling in C# because of the following two reasons.

1. To stop the abnormal termination of the program
2. To provide users understandable messages when an exception is raised. So that users can make a decision without the developer’s help.

Basically, by implementing Exception handling we are providing life to a program to talk to the user on behalf of a developer.

**What is the procedure to Handle Exception in C#?**

The Exception Handling in C# is a 4 steps procedure

1. Preparing the exception object that is appropriate to the current logical mistake.
2. Throwing that exception to the appropriate exception handler.
3. Catching that exception
4. Taking necessary actions against that exception

**How can we handle an exception in .NET?**

There are two methods to handle the exception in .NET

1. Logical Implementation
2. Try catch Implementation

**What is the logical implementation in C# to handle Exception?**

In logical Implementation, we need to handle the exception by using logical statements. In real-time programming, the first and foremost importance is always given to logical implementation only. If it is not possible to handle an exception using logical implementation then we need to try-catch implementation.

**Example: Handling Exception in C# using logical implementation**

The following example shows how to handle exceptions in C# using the logical Implementation. Here, we are checking the second number i.e. variable b value. If it equals 0, then we are printing one message that the second number should not be zero else if the second number is not zero then we are performing our division operation and showing the results on the console.

**namespace** *ExceptionHandlingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int** a, b, c;

Console.WriteLine**(**"ENTER ANY TWO NUBERS"**)**;

a = **int**.Parse**(**Console.ReadLine**())**;

b = **int**.Parse**(**Console.ReadLine**())**;

**if** **(**b == 0**)**

**{**

Console.WriteLine**(**"second number should not be zero"**)**;

**}**

**else**

**{**

c = a / b;

Console.WriteLine**(**"C VALUE = " + c**)**;

**}**

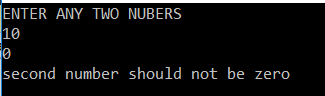
Console.ReadKey**()**;

**}**

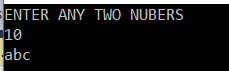
**}**

**}**

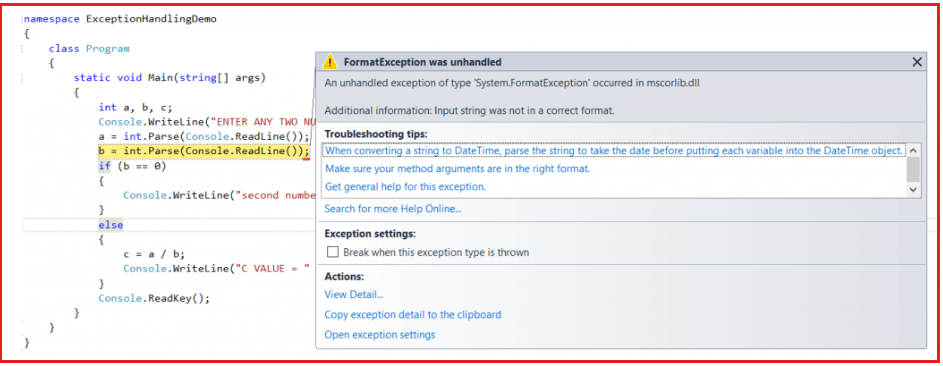
**Output:**



In the above example when the user entered the second number as zero exception will be raised and that is handled using the logical implementation in C#. But while we are entering two numbers instead of the number if we entered any character then it will give you one exception that is **FormatException** which is not handled in this program as shown below.



**Here we entered the second value as abc. So it will give us the below exception.**



So to handle such types of exceptions in C# we need to go for Try catch implementation.

**Exception handling in C# using the Try Catch implementation**

To implement the try-catch implementation .NET framework provides three keywords

1. **Try**
2. **Catch**
3. **finally**

**try:**

The try keyword establishes a block in which we need to write the exception causing and its related statements. That means exception-causing statements must be placed in the try block so that we can handle and catch that exception for stopping abnormal termination and to display end-user understandable messages.

**Catch:**

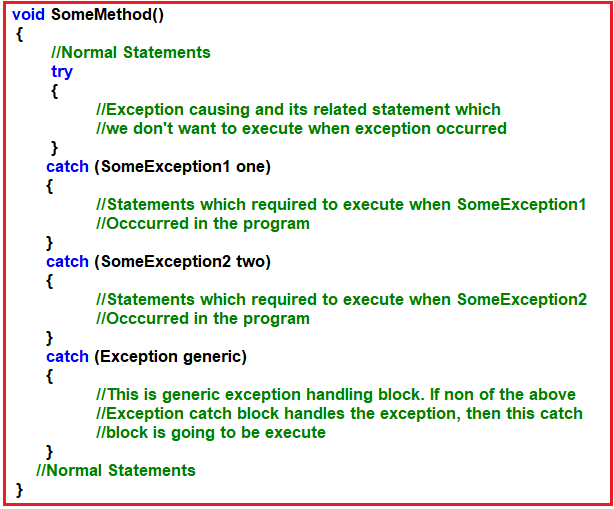
The catch block is used to catch the exception that is thrown from its corresponding try block. It has the logic to take necessary actions on that caught exception. The Catch block syntax in C# looks like a constructor. It does not take accessibility modifier, normal modifier, return type. It takes the only single parameter of type Exception. Inside catch block, we can write any statement which is legal in .NET including raising an exception.

**Finally:**

The keyword finally establishes a block that definitely executes statements placed in it. Statements that are placed in finally block are always going to be executed irrespective of the way the control is coming out from the try block either by completing normally or throwing an exception by catching or not catching.

**Syntax to use Exception Handling in C#:**

The following image shows the syntax to handle exceptions in C#. You can write any number of catch blocks for a given try block in C#. This will handle different types of exceptions thrown by the try block.



Once we use the try and catch blocks in our code the execution takes place as follows:

1. If all the statements under try block are executed successfully, from the last statement of the try block, the control directly jumps to the first statement that is present after the catch block (after all catch blocks) without executing catch block (it means there is no runtime error in the code at all ).
2. Then if any of the statements in the try block causes an error, from that statement without executing any other statements in the try block, the control directly jumps to the catch blocks which can handle that exception.
3. If a proper catch block is found that handles the exception thrown by the try block, then the abnormal termination stops there, executes the code under the catch block, and from there again it jumps to the first statement after all the catch blocks.
4. If a matching catch is not found then abnormal termination occurs.

**Note:** Here, we are showing the try and catch block execution. Later we will discuss the finally block.

**Example: Program to handle an exception using try-catch implementation with the generic catch**

The catch block without exception class is called a generic catch and the generic catch block in C# can handle any type of exception that is raised in the corresponding try block. For better understanding, please have a look at the below example. Here, we created the catch block without any Exception class.

**namespace** *ExceptionHandlingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int** a, b, c;

Console.WriteLine**(**"ENTER ANY TWO NUBERS"**)**;

**try**

**{**

a = **int**.Parse**(**Console.ReadLine**())**;

b = **int**.Parse**(**Console.ReadLine**())**;

c = a / b;

Console.WriteLine**(**"C VALUE = " + c**)**;

**}**

**catch**

**{**

Console.WriteLine**(**"error occured...."**)**;

**}**

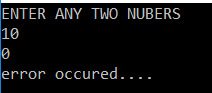
Console.ReadKey**()**;

**}**

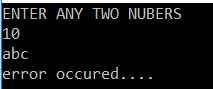
**}**

**}**

**Output1: Enter the value as 10 and 0**



**Output2: Enter the value as 10 and abc**



In the above example, there is no exception class used in the try block, so it is known as the generic catch block. The problem with the generic catch block is that, any kind of exception may occur the same message will be displayed to the end-user and the end-user cannot understand why the error has occurred; to overcome this, specific catch blocks are used. Using specific catch blocks it is possible to know more information about the exception.

**Properties of Exception Class in C#:**

The C# Exception Class has 3 properties are as follows:

1. **Message**: This property will store the reason why an exception has occurred.
2. **Source**: This property will store the name of the application from which the exception has been raised.
3. **Help link**: This is used to provide a link to any file /URL to give helpful information to the user when an exception is raised.

**Example:** **Exception Handling in C# using try-catch implementation with a specific catch block**

In the below example, we have created a catch block that taking the Exception class as a parameter and within the catch block we are print the exception information using the Exception class properties i.e. Message, Source, and Helplink. As you can see in the below code, we are using the super Exception class. This class is the superclass of all exception classes, so it will handle all types of exceptions raised in the try block.

**namespace** *ExceptionHandlingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int** a, b, c;

Console.WriteLine**(**"ENTER ANY TWO NUBERS"**)**;

**try**

**{**

a = **int**.Parse**(**Console.ReadLine**())**;

b = **int**.Parse**(**Console.ReadLine**())**;

c = a / b;

Console.WriteLine**(**"C VALUE = " + c**)**;

**}**

**catch** **(**Exception ex**)**

**{**

Console.WriteLine**(**ex.Message**)**;

Console.WriteLine**(**ex.Source**)**;

Console.WriteLine**(**ex.HelpLink**)**;

**}**

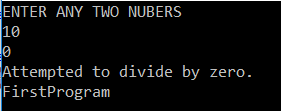
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



In the above example, the superclass exception is used to handle the exception. But if we use the super Exception class when there is any relevant class is available, it will kill the execution performance of the program.

In the next article, I am going to discuss how to use [**Multiple Catch Blocks and Finally Block**](https://dotnettutorials.net/lesson/multiple-catch-blocks-csharp/) in C#. Here, in this article, I try to explain the **Exception handling in C#** with examples. I hope you understood how to implement Exception Handling in C#.

Programs :

**using** System;

**class** MyClass

{

**public** **void** Method()

    {

**try**

        {

**int** x = 0;

**int** sum = 100 / x;

        }

**catch** (DivideByZeroException)

        {

**throw**;

        }

    }

}

**class** MyClient

{

**public** **static** **void** Main()

    {

        MyClass mc = **new** MyClass();

**try**

        {

            mc.Method();

        }

**catch** (Exception)

        {

            Console.WriteLine("Exception caught here");

        }

        Console.WriteLine("LAST STATEMENT");

    }

}

User defined exceptions :

**using** System;

**class** MyException : Exception

{

**public** MyException(**string** str)

    {

        Console.WriteLine("User defined exception");

    }

}

**class** MyClient

{

**public** **static** **void** Main()

    {

**try**

        {

**throw** **new** MyException("RAJESH");

        }

**catch** (Exception)

        {

            Console.WriteLine("Exception caught here" + e.ToString());

        }

        Console.WriteLine("LAST STATEMENT");

    }

}

Types of exceptions :

1. Built in exceptions
   1. IndexOutofRange
   2. Nullreference
2. User defined exception

The Anatomy of C# Exceptions

Exceptions allow an application to transfer control from one part of the code to another. When an exception is thrown, the current flow of the code is interrupted and handed back to a parent try catch block. C# exception handling is done with the follow keywords: try, catch, finally, and throw

* **try** – A try block is used to encapsulate a region of code. If any code throws an exception within that try block, the exception will be handled by the corresponding catch.
* **catch** – When an exception occurs, the Catch block of code is executed. This is where you are able to handle the exception, log it, or ignore it.
* **finally** – The finally block allows you to execute certain code if an exception is thrown or not. For example, disposing of an object that must be disposed of.
* **throw** – The throw keyword is used to actually create a new exception that is the bubbled up to a try catch finally block.