**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**)**;

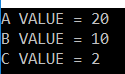
Console.ReadKey**()**;

**}**

**}**

**}**

**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**()**;

**}**

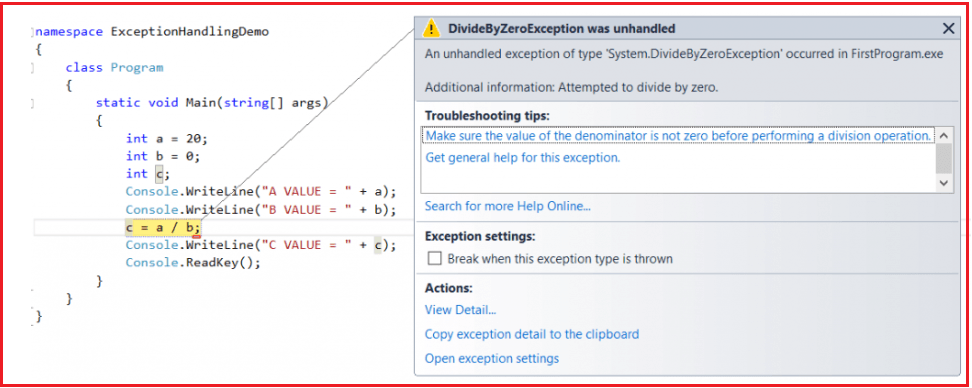
**}**

**}**

**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**)**;

**}**

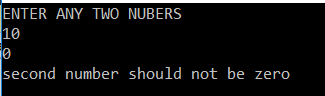
Console.ReadKey**()**;

**}**

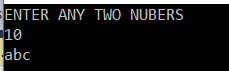
**}**

**}**

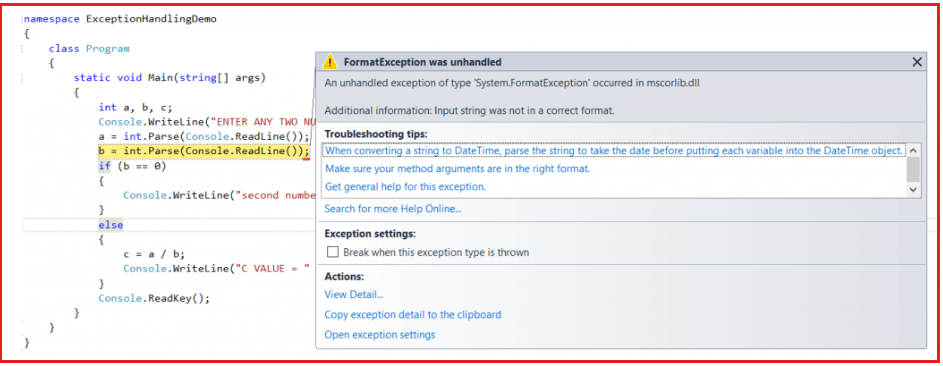
**Output:**



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:**

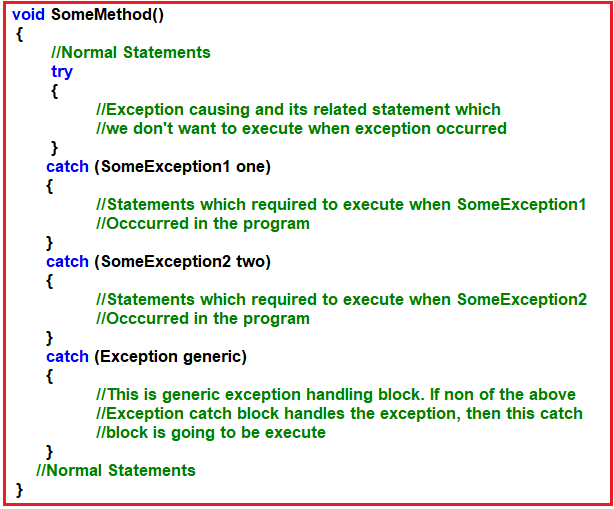
The catch block is used to catch the exception that is thrown from its corresponding try block. It has the logic to take necessary actions on that caught exception. The Catch block syntax in C# looks like a constructor. It does not take accessibility modifier, normal modifier, return type. It takes the 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...."**)**;

**}**

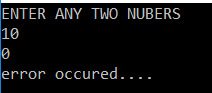
Console.ReadKey**()**;

**}**

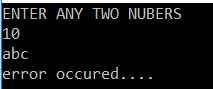
**}**

**}**

**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**)**;

**}**

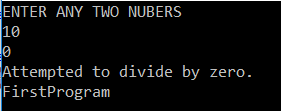
Console.ReadKey**()**;

**}**

**}**

**}**

**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.

**Multiple Catch Blocks and Finally Block in C#**

**Multiple Catch Blocks and Finally Block in C# with Examples**

In this article, I am going to discuss how to implement **Multiple Catch Blocks in C#** to handle different types of exceptions for a single try block as well as finally block in C# with examples. Please read our previous article before proceeding to this article where we discussed the basics of [**Exception Handling in C#**](https://dotnettutorials.net/lesson/exception-handling-csharp/) with examples. As part of this article, I am going to discuss the following pointers.

1. **How to implement multiple catch blocks in C#?**
2. **Is it possible to catch all exceptions using a single catch block?**
3. **When should we write multiple catch blocks in C# for a single try block?**
4. **What is finally block in C#?**
5. **Why do we need the finally block in the real-time project?**
6. **In how many ways we can use try-catch and finally block in C#?**

**How to Implement Multiple Catch Blocks in C#?**

It is possible in C#, to write multiple catch blocks for a given try block. When we implement multiple catch blocks in C#, then at any given point of time only one catch block going to be executed and other catch blocks will be ignored. With this keep in mind, let us proceed and see an example of how to implement Multiple Catch Blocks in C#.

**Example: Implementing Multiple Catch Blocks in C#.**

Let us see an example and understand how to implement multiple catch blocks for a given try block in C# and also understand the execution flow. Please have a look at the following example. As you can see, here, we created two catch blocks for the given try block. The first catch block taking the DivideByZeroException class as input parameter and the second catch block taking the FormatException class as the input parameter.

**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** **(**DivideByZeroException dbze**)**

**{**

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

**}**

**catch** **(**FormatException fe**)**

**{**

Console.WriteLine**(**"enter only integer numbers"**)**;

**}**

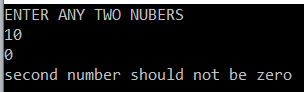
Console.ReadKey**()**;

**}**

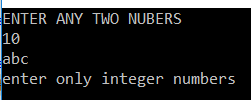
**}**

**}**

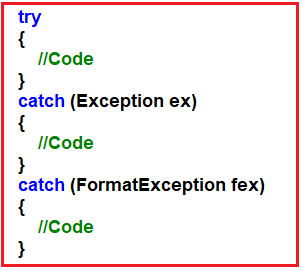
Run the above application and entered values like 10, 0. It will give you the below output



Again run the application and entered the values as 10 and abc. It will give you the below output



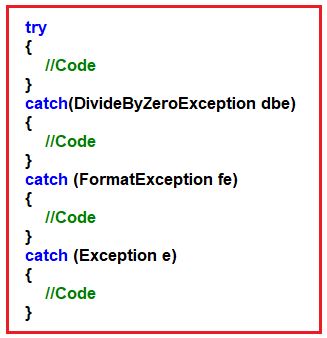
Whenever we implement Multiple Catch Blocks in C#, then it is not possible to write the catch blocks in the following manner, it raises to compilation error because the first catch block Exception can handle all the exceptions and does not make any sense to write the further catch blocks as they are never going to be executed.



**Is it possible to catch all exceptions using a single catch block in C#?**

Yes, it is possible. We can catch all exceptions with a single catch block with the parameter “**Exception**”. The Exception class is the superclass of all Exception classes and hence it can handle all types of exceptions thrown in the try block. We need to use this catch block only for stopping the abnormal termination irrespective of the exceptions thrown from its corresponding try block.

It is always recommended to write a catch block with the **Exception**parameter as the last catch block even though we are writing multiple catch blocks. It acts as a backup catch block. Following is the syntax to do the same.



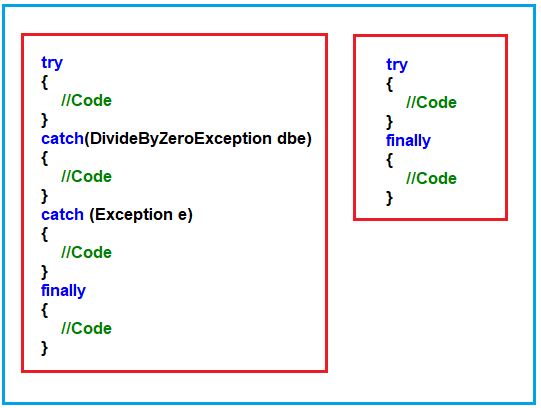
**When should we write Multiple Catch Blocks in C# for a single try block?**

We need to write multiple catch blocks in C# for a single try block because of the following reasons

1. To print messages specific to an exception or
2. To execute some logic specific to an exception

**The Finally Block in C#**

The keyword finally establishes a block that definitely executes statements placed in it irrespective of whether the exception has occurred or not, irrespective of whether the exception is handled or not in the catch block. That means in the simple word we can say that the statements which are placed in the finally block are always executed irrespective of the way the control is coming out from the try block either by completing normally or throwing the exception by catching or not catching. Following is the syntax to use finally block in C#,



As you can see, in two ways we can write the finally block in C#. They are as follows:

1. **Try, Catch, and Finally:**In this case, the exception will be handled, and stopping the abnormal termination along with the statements that are placed within the “finally” block gets executed at any cost.
2. **Try and Finally:**In this case, abnormal termination will not stop when a runtime error occurs because exceptions are not handled but even if an abnormal termination occurs, the finally blocks get executed.

**Why do we need the finally block in the real-time project?**

As per the industry coding standard, within the “**finally**” block we need to write the resource releasing logic or clean up the code. Resource releasing logic means un-referencing objects that are created in the try block. Since the statements written in the try and catch block are not guaranteed to be executed we must place them in finally block.

For example, if we want to close ADO.NET objects such as **Connection object, Command object,** etc. we must call the **close()** method in both the try as well as in the catch block to guarantee its execution. Instead of placing the same **close()** method call statements in multiple places if we write it in the finally block which will be always executed irrespective of the exception raised or not raised.

**Example: Use of finally block in C#.**

Let us see an example to understand the use of finally block in C#. In the below example, for the given try block, we have written two catch blocks and after the second catch block, we have written the finally block. The statements present inside the catch block are going to be executed irrespective of exception occurred or not, irrespective of the exception is handled or not. That means if we place something in the finally block, then those statements are definitely going to be executed.

**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** **(**DivideByZeroException dbe**)**

**{**

Console.WriteLine**(**"2nd number should not be zero"**)**;

**}**

**catch** **(**FormatException fe**)**

**{**

Console.WriteLine**(**"enter only integer number"**)**;

**}**

**finally**

**{**

Console.WriteLine**(**"hello this is finally block"**)**;

**}**

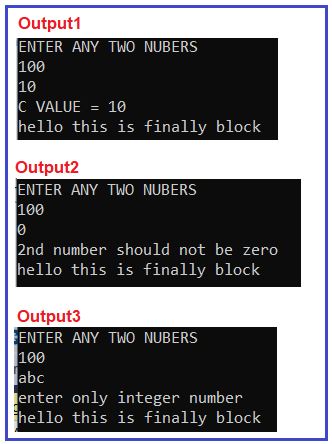
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**In how many ways we can use try-catch and finally block in C#?**

We can use try-catch-finally in three different ways. They are as follows:

1. **Try and Catch:**In this case, the exception will be handled and stop the abnormal termination.
2. **Try, Catch, and Finally:**In this case, the exception will be handled, and stopping the abnormal termination along with the statements that are placed within the “finally” block gets executed at any cost.
3. **Try and Finally:**In this case, abnormal will not stop when a runtime error occurs because exceptions are not handled but even if an abnormal termination occurs also finally blocks get executed.

**Custom Exception in C#**

**Creating Custom Exception in C# with examples**

In this article, I am going to discuss **how to create Custom Exception in C#** with Examples. Please read our previous article where we discussed how to implement [**Multiple Catch Blocks and Finally block in C#**](https://dotnettutorials.net/lesson/multiple-catch-blocks-csharp/) with Examples. As part of this article, we are going to discuss the following pointers in detail.

1. **What are System Exceptions and Application Exceptions in C#?**
2. **What are the different ways to create a Custom Exception in C#?**
3. **Why do we need to create Custom Exceptions in C#?**
4. **Examples of creating Custom Exceptions in C#.**

In C#, the exceptions are divided into two types such as

1. System exception
2. Application exception

**What are System Exceptions in C#?**

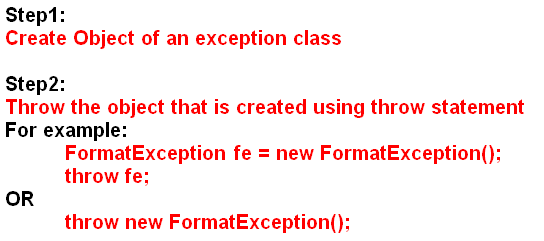
An exception that is raised implicitly under a program by the exception manager because of some logical mistakes (some predefined conditions) is known as a system exception. For example:

1. **DivideByZeroException**
2. **IndexOutOfRangeException**
3. **FormatExceptionetc**

An exception that is raised explicitly under a program based on our own condition (i.e. user-defined condition) is known as the application exception.

**What are Application Exceptions in C#?**

An exception that is raised explicitly under a program based on our own condition (i.e. user-defined condition) is known as an application exception. As a programmer, we can raise application exception at any given point of time. To raise an application exception we need to adopt the following process



While creating and throwing an object of the Exception class we should not use system exception classes like DivideByZeroException, FormatException, etc.

**What are the Different Ways to Create Custom Exception in C#?**

To create and throw an object of exception class by us, we have two different options.

1. Create the object of a predefined Exception class where we need to pass the error message as a parameter to its constructor and then throw that object so that whenever the exception occurs the given error message gets displayed.
2. Define a new class of type exception and throw that class object by creating it.

**To define an exception class of our own we have to follow two steps**

**Step1:** Define a new class inheriting from the predefined class Exception class so that the new class also acts as an Exception class.

**Step2:** Now override the virtual property message with the required error message.

Let us understand how to create a custom exception in C# with an example:

**Example: Creating and throwing a custom exception in C#**

**namespace** *ExceptionHandlingDemo*

**{**

//Creating our own Exception Class by inheriting Exception class

**public** **class** OddNumberException : Exception

**{**

//Overriding the Message property

**public** **override** string Message

**{**

**get**

**{**

**return** "divisor cannot be odd number";

**}**

**}**

**}**

**class** Program

**{**

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

**{**

**int** x, y, z;

Console.WriteLine**(**"ENTER TWO INTEGER NUMBERS:"**)**;

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

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

**try**

**{**

**if** **(**y % 2 **>** 0**)**

**{**

//OddNumberException ONE = new OddNumberException();

//throw ONE;

**throw** new OddNumberException**()**;

**}**

z = x / y;

Console.WriteLine**(**z**)**;

**}**

**catch** **(**OddNumberException one**)**

**{**

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

**}**

Console.WriteLine**(**"End of the program"**)**;

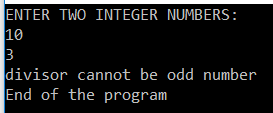
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Why do we need to create custom exceptions in C#?**

If none of the already existing .NET exception classes serve our purpose then we need to go for custom exceptions in C#. For example, I have an ASP.NET Web application. The application should allow the user to have only one logged-in session. If the user is already logged in, and if he opens another browser window and tries to log in again, the application should throw an error stating he is already logged in to another browser window.

Within the .NET framework, we do not have any exception class that handles this problem. So this scenario is one of the examples where we want to create a custom exception.

We know that an exception is a class. So to create a Custom exception, Create a class that derives from **System.Exception** class. As a convention, end the class name with Exception suffix. All .NET exceptions end with, exception suffix. If you don’t do so, you won’t get a compiler error, but you will be deviating from the guidelines for creating custom exceptions.

**public** **class** UserAlreadyLoggedInException : Exception

**{**

**}**

Provide a public constructor that takes a string parameter. This constructor simply passes the string parameter, to the base exception class constructor.

**public** UserAlreadyLoggedInException**(**string message**)** : **base(**message**)**

**{**

**}**

Using InnerExceptions, you can also track back the original exception. If you want to provide this capability for your custom exception class, then overload the constructor as shown below.

**public** UserAlreadyLoggedInException**(**string message, Exception innerException**)**

: **base(**message, innerException**)**

**{**

**}**

If you want your Exception class object to work across application domains, then the object must be serializable. To make your exception class serializable mark it with Serializable attribute and provide a constructor that invokes the base Exception class constructor that takes the SerializationInfo and StreamingContext objects as parameters.

**[**Serializable**]**

**public** **class** UserAlreadyLoggedInException : Exception

**{**

**public** UserAlreadyLoggedInException**(**SerializationInfo info, StreamingContext context**)**: **base(**info, context**)**

**{**

**}**

**}**

**Note:** It is also possible to provide your own custom serialization, which will discuss in a later session.

**Example: Create a custom Exception in C#:**

**using** *System;*

**using** *System.Runtime.Serialization;*

**namespace** *ExceptionHandlingDemo*

**{**

**[**Serializable**]**

**public** **class** UserAlreadyLoggedInException : Exception

**{**

**public** UserAlreadyLoggedInException**(**string message**)**

: **base(**message**)**

**{**

**}**

**public** UserAlreadyLoggedInException**(**string message, Exception innerException**)**

: **base(**message, innerException**)**

**{**

**}**

**public** UserAlreadyLoggedInException**(**SerializationInfo info, StreamingContext context**)**

: **base(**info, context**)**

**{**

**}**

**}**

**class** Program

**{**

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

**{**

**try**

**{**

**throw** new UserAlreadyLoggedInException**(**"User Already logged in"**)**;

**}**

**catch** **(**UserAlreadyLoggedInException ex**)**

**{**

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

**}**

Console.WriteLine**(**"End of the program"**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Inner Exception in C#**

**Inner Exception in C# with an Example**

In this article, I am going to discuss the **Inner Exception in C#** with Examples. Please read our previous article before proceeding to this article where we discussed [**how to create a Custom Exception in C#**](https://dotnettutorials.net/lesson/create-custom-exception-csharp/)with examples. At the end of this article, you will understand what is Inner Exception is and its need in C#.

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

The Inner Exception in C# is a property of an exception class. When there is a series of exceptions, then the most current exception obtains the previous exception details in the InnerException property. In order words, we can say that the InnerException property returns the original exception that caused the current exception. If this is not clear at the moment, then don’t worry we will discuss this with examples.

**Inner Exception Example in C#:**

Let us say we have an exception inside a try block which is throwing **DivideByZeroException** and the catch block catches that exception and then tries to write that exception to a file. However, if the file path is not found, then the catch block is also going to throw **FileNotFoundException**.

Let’s say the outside try block catches this **FileNotFoundException** exception, but how about the actual **DivideByZeroException** that was thrown? Is it lost? No, the **InnerException** property of the Exception class contains the actual exception.

**Example:**

Let us understand the Inner Exception with an example. In order to see the **inner exception**, we have to make this program cause an exception to fail. To do that we have 3 options

1. Enter a Character instead of a number (Causes Format Exception)
2. Or Enter a very big number that an integer cannot hold (Causes Over Flow Exception)
3. Or Enter Zero for Second Number (Causes Divide By Zero Exception)

**using** *System;*

**using** *System.IO;*

**namespace** *ExceptionHandlingDemo*

**{**

**class** Program

**{**

**public** **static** **void** Main**()**

**{**

**try**

**{**

**try**

**{**

//throw new ArgumentException();

Console.WriteLine**(**"Enter First Number"**)**;

**int** FN = Convert.ToInt32**(**Console.ReadLine**())**;

Console.WriteLine**(**"Enter Second Number"**)**;

**int** SN = Convert.ToInt32**(**Console.ReadLine**())**;

**int** Result = FN / SN;

Console.WriteLine**(**"Result = {0}", Result**)**;

**}**

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

**{**

//make sure this path does not exist

string filePath = @"C:\LogFile\Log.txt";

**if** **(**File.Exists**(**filePath**))**

**{**

StreamWriter sw = new StreamWriter**(**filePath**)**;

sw.Write**(**ex.GetType**()**.Name + ex.Message + ex.StackTrace**)**;

sw.Close**()**;

Console.WriteLine**(**"There is a problem! Plese try later"**)**;

**}**

**else**

**{**

//To retain the original exception pass it as a parameter

//to the constructor, of the current exception

**throw** new FileNotFoundException**(**filePath + " Does not Exist", ex**)**;

**}**

**}**

**}**

**catch** **(**Exception e**)**

**{**

//e.Message will give the current exception message

Console.WriteLine**(**"Current or Outer Exception = " + e.Message**)**;

//Check if inner exception is not null before accessing Message property

//else, you may get Null Reference Excception

**if** **(**e.InnerException != **null)**

**{**

Console.Write**(**"Inner Exception : "**)**;

Console.WriteLine**(**String.Concat**(**e.InnerException.StackTrace, e.InnerException.Message**))**;

**}**

**}**

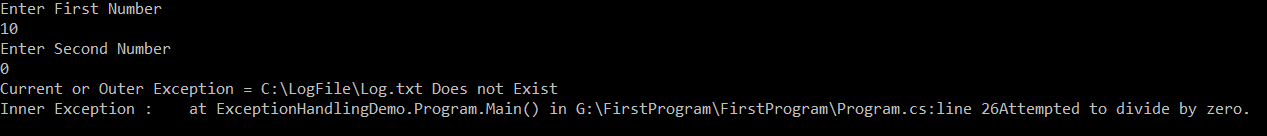
Console.ReadLine**()**;

**}**

**}**

**}**

When you run the above program it will give us the following output.



**Exception Handling Abuse in C#**

**Exception Handling Abuse in C# with an Example**

In this article, I am going to discuss the **Exception Handling Abuse in C#** and then we will see how to prevent Exception Handling Abuse in C# with one example. Please read our previous article where we discussed the [**Inner Exception in C#**](https://dotnettutorials.net/lesson/inner-exception-csharp/) with an example.

Exceptions are nothing but runtime errors that occur during the execution of a program. For example, when an application is executing a database query, due to some reason the database connection is lost, then we will get an SQLException runtime error. Exception handling is generally used to handle these scenarios.

But sometimes, as a programmer, we are using exception handling mechanisms to implement programming logic which is bad, and this is called as exception handling abuse.

**Example: Here we are using exception handling to implement logical flow:**

**namespace** *ExceptionHandlingDemo*

**{**

**class** Program

**{**

**public** **static** **void** Main**()**

**{**

**try**

**{**

//Convert.ToInt32() can throw FormatException, if the entered value

//cannot be converted to integer. So use int.TryParse() instead

Console.WriteLine**(**"Please enter First Number"**)**;

**int** FNO = Convert.ToInt32**(**Console.ReadLine**())**;

Console.WriteLine**(**"Please enter Second Number"**)**;

**int** SNO = Convert.ToInt32**(**Console.ReadLine**())**;

**int** Result = FNO / SNO;

Console.WriteLine**(**"Result = {0}", Result**)**;

**}**

**catch** **(**FormatException**)**

**{**

Console.WriteLine**(**"Only numbers are allowed!"**)**;

**}**

**catch** **(**OverflowException**)**

**{**

Console.WriteLine**(**"Only numbers between {0} & {1} are allowed",

Int32.MinValue, Int32.MaxValue**)**;

**}**

**catch** **(**DivideByZeroException**)**

**{**

Console.WriteLine**(**"Secoond Number cannot be zero"**)**;

**}**

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

**{**

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

**}**

Console.ReadLine**()**;

**}**

**}**

**}**

**Preventing exception handling abuse in C#:**

Let’s rewrite the same example that doesn’t use exception handling to control the program’s logical flow.

**namespace** *ExceptionHandlingDemo*

**{**

**class** Program

**{**

**public** **static** **void** Main**()**

**{**

**try**

**{**

Console.WriteLine**(**"Please enter First Number"**)**;

**int** FNO;

//int.TryParse() will not throw an exception, instead returns false

//if the entered value cannot be converted to integer

**bool** isValidFNO = **int**.TryParse**(**Console.ReadLine**()**, out FNO**)**;

**if** **(**isValidFNO**)**

**{**

Console.WriteLine**(**"Please enter Second Number"**)**;

**int** SNO;

**bool** isValidSNO = **int**.TryParse**(**Console.ReadLine**()**, out SNO**)**;

**if** **(**isValidSNO && SNO != 0**)**

**{**

**int** Result = FNO / SNO;

Console.WriteLine**(**"Result = {0}", Result**)**;

**}**

**else**

**{**

//Check if the second number is zero and print a friendly error

//message instead of allowing DivideByZeroException exception

//to be thrown and then printing error message to the user.

**if** **(**isValidSNO && SNO == 0**)**

**{**

Console.WriteLine**(**"Second Number cannot be zero"**)**;

**}**

**else**

**{**

Console.WriteLine**(**"Only numbers between {0} && {1} are allowed",

Int32.MinValue, Int32.MaxValue**)**;

**}**

**}**

**}**

**else**

**{**

Console.WriteLine**(**"Only numbers between {0} && {1} are allowed",

Int32.MinValue, Int32.MaxValue**)**;

**}**

**}**

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

**{**

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

**}**

Console.ReadLine**()**;

**}**

**}**

**}**

**Output:**

