**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. 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 Exceptions in C#?**
9. **What is the Logical Implementation in C# to Handle Exception?**
10. **Exception handling in C# using Try Catch implementation**
11. **Properties of Exception Class 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 as follows:

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 a compilation error (compile-time error). These errors occur due to syntactical mistakes in the program. That means these errors occur by typing the wrong syntax like missing double quotes in a string value and missing terminators in a statement, typing wrong spelling for keywords, assigning wrong data to a variable, trying to create an object for abstract class and interface, etc. So, whenever we compile the program, the compiler recognizes these errors and it will show us the list of errors.

So, in simple words, we can say that this type of error occurs due to a poor understanding of the programming language. These errors are identified by the compiler 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 runtime errors. These errors occurred at runtime due to various reasons such as when we are entering the 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, and missing required resources, etc. So, in simple words, we can say that the errors which are come while running the program are called runtime errors.

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.

**Note:** The compiler will never check the logic, the compiler will only check the syntaxes. So, the compiler will identify the syntax error, but not the logical error.

**Is Runtime Errors are Dangerous?**

Yes, runtime errors are dangerous. See, if you are transferring the money, then there are two updated statements. One update statement will deduct the money from the source account and another update statement add the money to the destination account. Suppose, the first update statement was executed successfully and before executing the second update statement, some runtime error occurred because of some reason. That means the money is deducted from your account but not added to the destination account.

Then what you will do? You might be contacted with the bank, you might be going to the nearest bank and investigating what happens, Why the money is deducted from my account, and why it is not added to the destination account. This is the problem and this very dangerous and this is because we are not handling the runtime errors in our application.

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

An Exception is a class in C# which is responsible for abnormal termination of the program when runtime errors occur while running the program.  So, these errors (runtime) are very dangerous because whenever the runtime errors occur in the programs, the program gets terminated abnormally on the same line where the error gets occurred without executing the next line of code.

**Note:** Most people are saying Runtime Errors are Exceptions which is not true. Exceptions are classes that are responsible for abnormal termination of the program when runtime errors occur.

**Who is Responsible for the Abnormal Termination of the Program whenever Runtime Errors occur?**

Objects of Exception classes are responsible for abnormal termination of the program whenever runtime errors 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 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 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.

**Note:** Exception class is the superclass of all Exception classes in C#.

**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?**

The CLR creates the exception class object that is associated with that logical mistake (exception) and terminates the program 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.

**Program Execution without Exception in C#**

The following C# 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.

using System;

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;

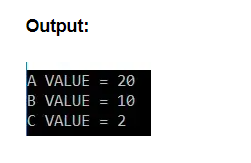
Console.WriteLine("C VALUE = " + c);

Console.ReadKey();

}

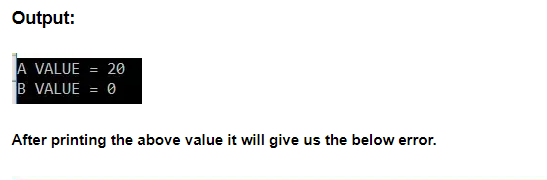
}

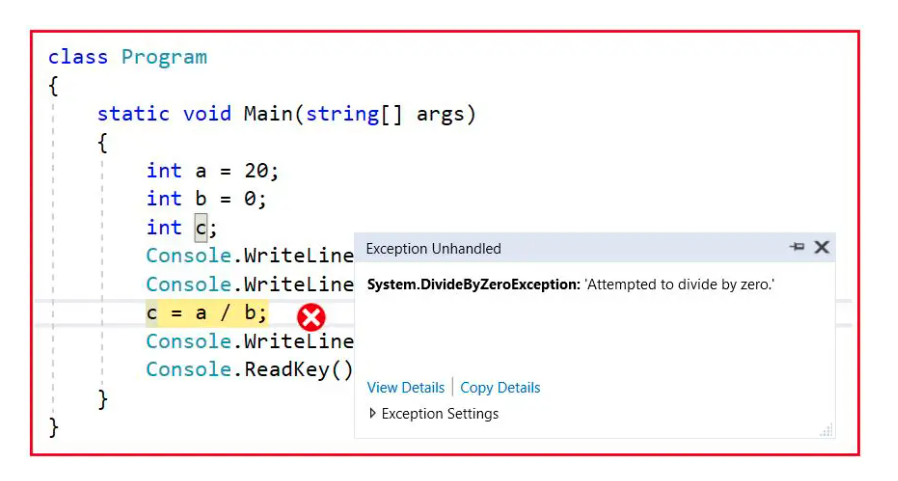
}



##### ****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 by 0 which is not possible in mathematics. So, it will throw the 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.





###### **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. So, what CLR will do in this case, first it will check what type of logical error is this. It will find that it will be a Divide By Zero Logical Error. So, then what CLR will do is, it will create an instance of **DivideByZeroException** class, and then it will throw that instance by using the throw statement like **throw** **new DivideByZeroException();** From the above program, we can define the exception technically as follows:

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 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 with 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 Exceptions 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 go for try-catch implementation.

##### ****Handling Exceptions 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 Number2 value. If it equals 0, then we are printing one message saying 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. Here, we are using IF-ELSE logical statement to handle the exception.

using System;

namespace ExceptionHandlingDemo

{

class Program

{

static void Main(string[] args)

{

int Number1, Number2, Result;

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

Number1 = int.Parse(Console.ReadLine());

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

Number2 = int.Parse(Console.ReadLine());

if (Number2 == 0)

{

Console.WriteLine("Second Number Should Not Be Zero");

}

else

{

Result = Number1 / Number2;

Console.WriteLine($"Result = {Result}");

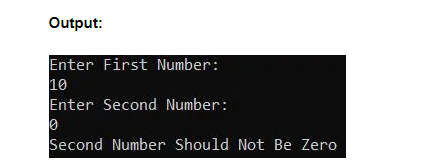
}

Console.ReadKey();

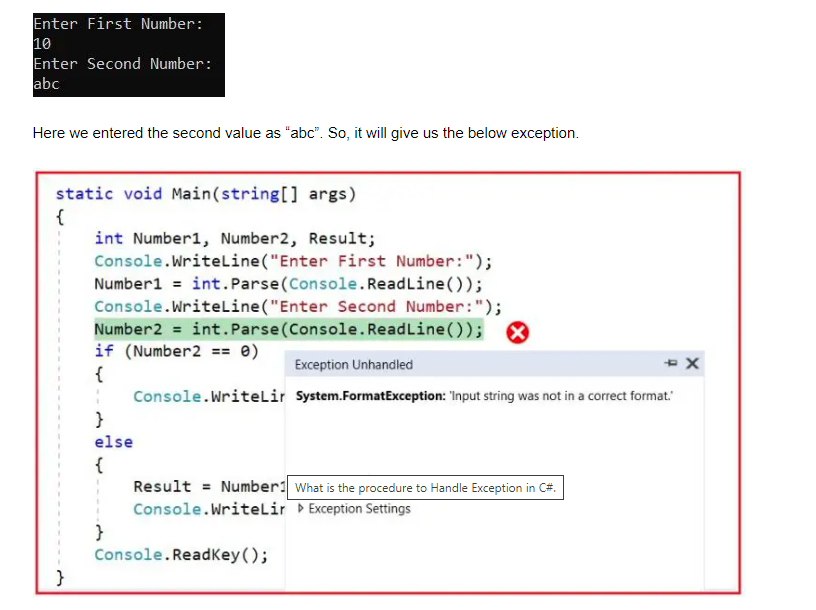
}

}

}



In the above case, 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 which is **FormatException** which is not handled in this program as shown below.



So to handle such types of exceptions in C# we need to go for Try catch implementation. So, the point that you need to remember is if we are unable to handle the exception using logical implementation, then only we need to go for try-catch implementation in C#.

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

To implement the try-catch implementation, the .NET framework provides three keywords. They are as follows:

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

**Try Block:**

The try keyword establishes a block in which we need to write the exception causing and its related statements. That means exception-causing statements and the related statements which we should not execute when an exception occurred must be placed in the try block. When the exception occurred, the CLR will create an instance of the Exception class based on the logical error and then throw that Exception object which is going to be handled by the corresponding catch block.

**Catch Block:**

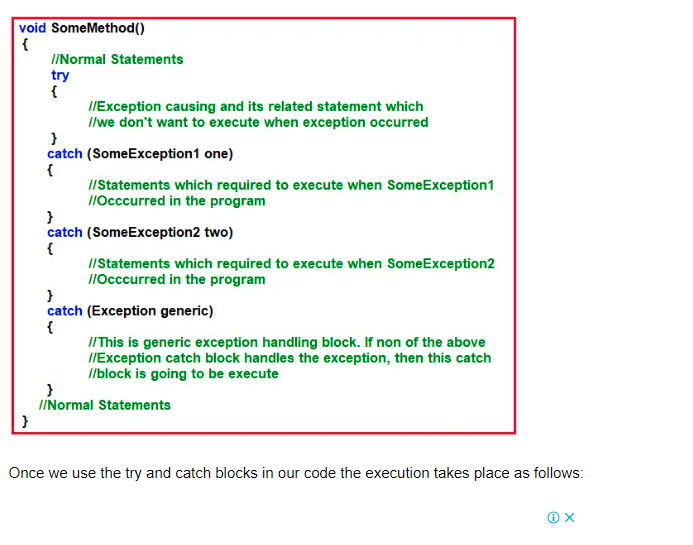
The catch block is used to catch the exception that is thrown from its corresponding try block. It has the logic to take necessary actions on that caught exception. The Catch block syntax in C# looks like a constructor. It does not take accessibility modifiers, normal modifiers, or return types. It takes only a single parameter of type Exception or any child class of the parent Exception class. Inside the catch block, we can write any statement which is legal in .NET including raising an exception. In this case, we can stop the abnormal termination of the program and we can also give user understandable error message so that the user can take necessary action to resolve the error.

**Finally Block:**

The keyword finally establishes a block that definitely executes the statements placed in it irrespective of whether any exception has occurred or not. That means the statements that are placed in finally block are always going to be executed irrespective of whether any exception is thrown or not, irrespective of whether the thrown exception is handled by the catch block or not.

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

The following image shows the syntax to use exception handling in C#. It starts with the try block, followed by the catch block, and writing the finally block is optional. 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.



1. If all the statements under the 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 the catch block (it means there is no runtime error in the code at all).
2. 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 block is not found, then the generic catch block is going to execute to handle the abnormal termination.
5. If you don’t have the generic catch block and if any of the catch blocks are unable to handle the exception, then again, the program execution terminates abnormally.

**Note:** Here, we are showing the try-and-catch block execution. Later in our upcoming videos, we will discuss the need and use of finally block in C#.

##### ****Example to Handle an Exception using Try-Catch Implementation with Generic Catch Block in C#****

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.

using System;

namespace ExceptionHandlingDemo

{

class Program

{

static void Main(string[] args)

{

int Number1, Number2, Result;

try

{

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

Number1 = int.Parse(Console.ReadLine());

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

Number2 = int.Parse(Console.ReadLine());

Result = Number1 / Number2;

Console.WriteLine($"Result = {Result}");

}

catch

{

Console.WriteLine("Some Error Occurred...");

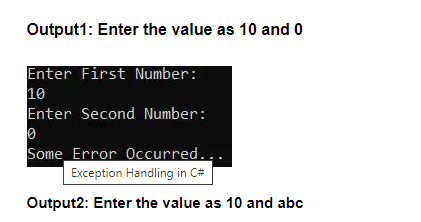
}

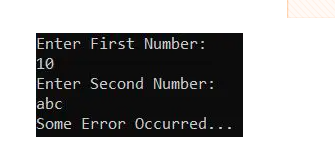
Console.ReadKey();

}

}

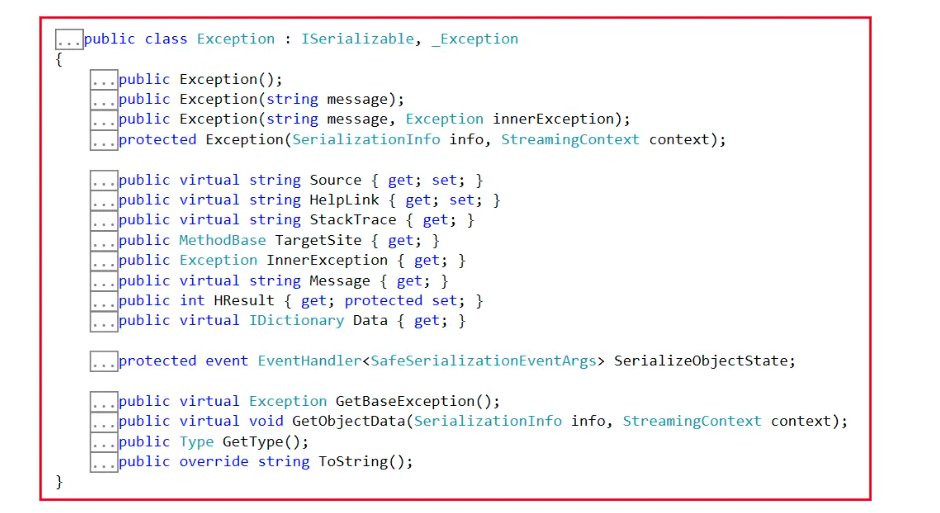
}





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

The Exception Class is the superclass of all Exception classes and this provides some virtual properties and methods which are re-implemented by the child classes. If you go to the definition of the Exception class, then you will see the following.



Some of the important properties of the Exception Class are properties 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. **HelpLink**: This is used to provide a link to any file /URL to give helpful information to the user about why this exception is raised.
4. **StackTrace**: This is used to provide more information about the Exception like the reason for the exception, at what method and what class the exception occurred, and at what line number the exception has occurred which helps us to resolve the issue.

**Exception Handling in C# using Try-Catch Implementation with Exception Catch Block**

In the below example, we have created a catch block that takes the Exception class as a parameter and within the catch block, we print the exception information using the Exception class properties i.e. Message, Source, StackTrace, 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.

using System;

namespace ExceptionHandlingDemo

{

class Program

{

static void Main(string[] args)

{

int Number1, Number2, Result;

try

{

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

Number1 = int.Parse(Console.ReadLine());

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

Number2 = int.Parse(Console.ReadLine());

Result = Number1 / Number2;

Console.WriteLine($"Result = {Result}");

}

catch (Exception ex)

{

Console.WriteLine($"Message: {ex.Message}");

Console.WriteLine($"Source: {ex.Source}");

Console.WriteLine($"HelpLink: {ex.HelpLink}");

Console.WriteLine($"StackTrace: {ex.StackTrace}");

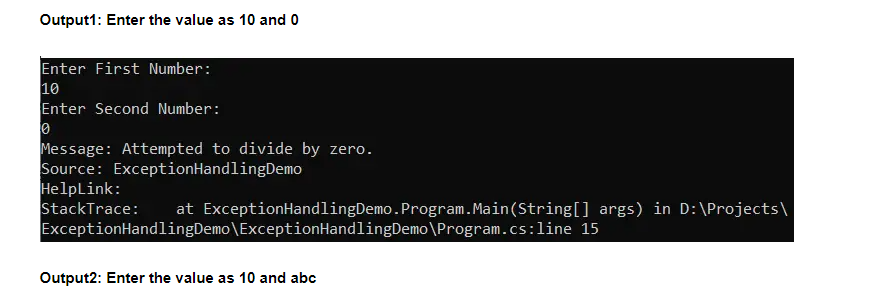
}

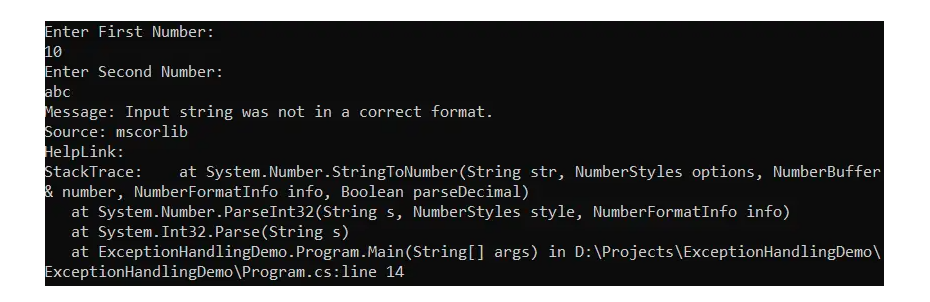
Console.ReadKey();

}

}

}





In the above example, the Exception superclass is used to handle all types of exceptions thrown from the corresponding try block. But if we use the super Exception class, when there is a relevant child exception class available, it will kill the execution performance of the program. So, in the next video, I will show you how to implement multiple catch blocks to handle different types of exceptions.

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# with Examples. Here, in this article, I try to explain **Exception handling in C#** with Examples. I hope you understood how to implement Exception Handling in C# with Examples.

**Multiple Catch Blocks 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 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?**

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

##### ****Example to Understand Multiple Catch Blocks in C#****

Let us see an example and try to understand how to implement multiple catch blocks for a given try block in C# and also try to 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 takes the DivideByZeroException class as the input parameter and the second catch block takes the FormatException class as the input parameter.

using System;

namespace ExceptionHandlingDemo

{

class Program

{

static void Main(string[] args)

{

int Number1, Number2, Result;

try

{

Console.WriteLine("Enter First Number");

Number1 = int.Parse(Console.ReadLine());

Console.WriteLine("Enter Second Number");

Number2 = int.Parse(Console.ReadLine());

Result = Number1 / Number2;

Console.WriteLine($"Result: {Result}");

}

catch (DivideByZeroException DBZE)

{

Console.WriteLine("Second Number Should Not Be Zero");

}

catch (FormatException FE)

{

Console.WriteLine("Enter Only Integer Numbers");

}

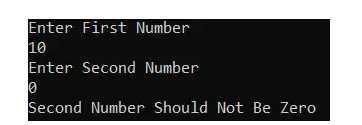
Console.ReadKey();

}

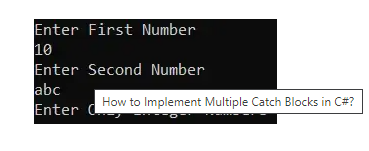
}

}

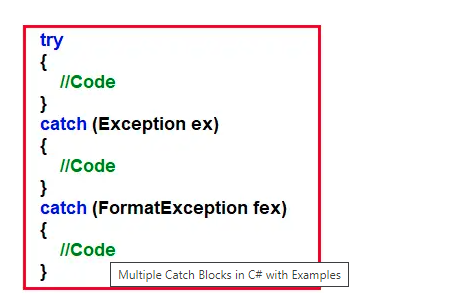
Run the above application and entered values like 10, and 0. It will give you the below output. In this case, when we entered the second number as 0, the CLR implicitly throws the **DivideByZeroException**instance and the catch block which takes as a parameter of DivideByZeroException will handle the exception, and hence the message Second Number Should Not Be Zero will be printed on the console as shown in the below image.



Again, run the application and entered the values as 10 and abc. This time it will give you the following output. In this case, when we entered the second value as abc, the CLR implicitly throws **FormatException**instance and the catch block which takes as a parameter of FormatException will handle the exception and hence, the message Enter Only Integer Numbers will be printed on the console as shown in the below image.



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 as it is the parent class of the child exception classes and it 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 Child 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. For a better understanding, please have a look at the following example.

using System;

namespace ExceptionHandlingDemo

{

class Program

{

static void Main(string[] args)

{

int Number1, Number2, Result;

try

{

Console.WriteLine("Enter First Number");

Number1 = int.Parse(Console.ReadLine());

Console.WriteLine("Enter Second Number");

Number2 = int.Parse(Console.ReadLine());

Result = Number1 / Number2;

Console.WriteLine($"Result: {Result}");

}

catch (Exception ex)

{

Console.WriteLine("Generic Catch Block...");

}

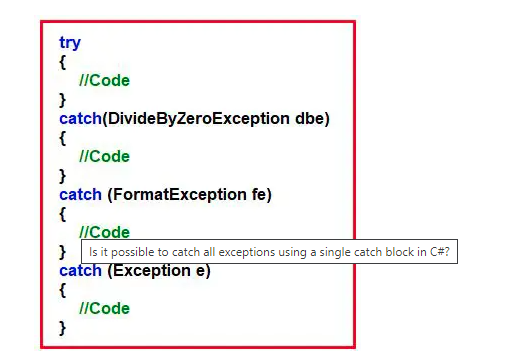
Console.ReadKey();

}

}

}

Now, in the above example, for any type of exception, the generic catch block is going to be executed. 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.



In the below example, we are writing the generic catch block as the last catch block and if any exception thrown in the try block is not handled by any specific catch block, then the generic block is going to be executed.

using System;

namespace ExceptionHandlingDemo

{

class Program

{

static void Main(string[] args)

{

int Number1, Number2, Result;

try

{

Console.WriteLine("Enter First Number");

Number1 = int.Parse(Console.ReadLine());

Console.WriteLine("Enter Second Number");

Number2 = int.Parse(Console.ReadLine());

Result = Number1 / Number2;

Console.WriteLine($"Result: {Result}");

}

catch (DivideByZeroException DBZE)

{

Console.WriteLine("Second Number Should Not Be Zero");

}

catch (FormatException FE)

{

Console.WriteLine("Enter Only Integer Numbers");

}

catch (Exception ex)

{

Console.WriteLine("Generic Catch Block...");

}

Console.ReadKey();

}

}

}

Now, run the above application and entered big integer values like 23456789234. It will give you the following output. In this case, when we entered the number 23456789234, the CLR implicitly throw the OverflowException instance and we don’t have any catch block which takes as a parameter of OverflowException**,**hence the generic catch block is going to execute and display the following message.



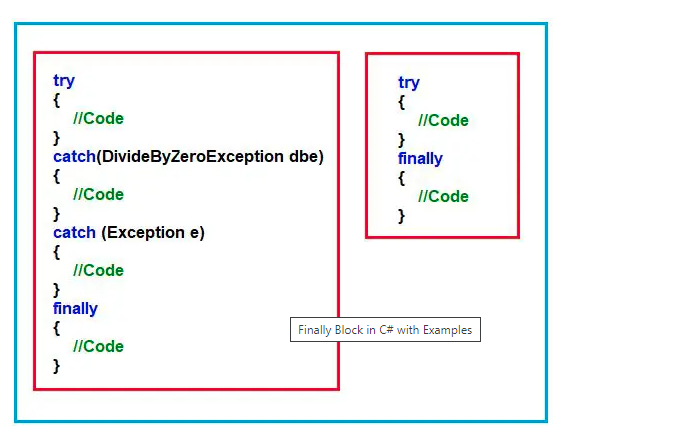
## ****Finally Block in C# with Examples****

In this article, I am going to discuss **Finally Block in C#** with examples. Please read our previous article where we discussed [**Multiple Catch Blocks in C#**](https://dotnettutorials.net/lesson/multiple-catch-blocks-csharp/) with examples. As part of this article, I am going to discuss the following pointers.

1. **What is finally block in C#?**
2. **Why do we need the finally block in the real-time project?**
3. **In how many ways we can use try-catch and finally block in C#?**

##### ****The Finally Block in C#****

The keyword finally establishes a block that definitely executes the statements placed in it irrespective of whether any exception has occurred or not. That means the statements that are placed in finally block are guaranteed to be going to be executed irrespective of whether any exception is thrown or not in the try block, irrespective of whether the thrown exception is handled by the catch block or not. 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 finally block in the Real-Time Project Development?****

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 it will be always executed irrespective of the exception raised or not raised.

##### ****Example to Understand the 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 finally block is 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.

using System;

namespace ExceptionHandlingDemo

{

class Program

{

static void Main(string[] args)

{

int Number1, Number2, Result;

try

{

Console.WriteLine("Enter First Number");

Number1 = int.Parse(Console.ReadLine());

Console.WriteLine("Enter Second Number");

Number2 = int.Parse(Console.ReadLine());

Result = Number1 / Number2;

Console.WriteLine($"Result: {Result}");

}

catch (DivideByZeroException DBZE)

{

Console.WriteLine("Second Number Should Not Be Zero");

}

catch (FormatException FE)

{

Console.WriteLine("Enter Only Integer Numbers");

}

finally

{

Console.WriteLine("Hello this is finally block...");

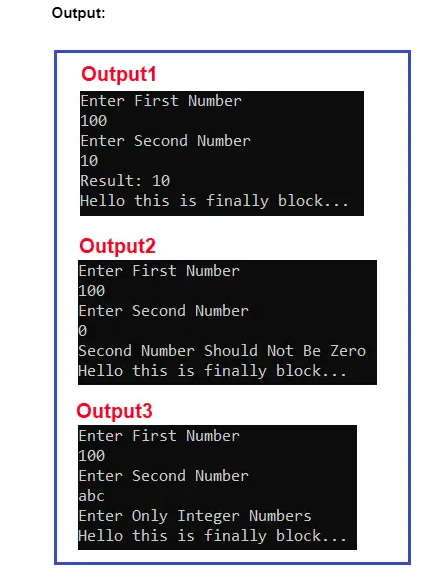
}

Console.ReadKey();

}

}

}



##### ****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.

##### ****Example to Understand Try-Finally Block without Catch Block:****

Let us see one example to understand the Try-Finally Block without Catch Block in C#. Please have a look at the following example. Inside the SomeMethod, we have written the logic in such a way that at runtime, this method is going to throw the Divide By Zero Exception and if you see we don’t handle that Exception using the catch block. Even though the catch block is not there, the finally block is going to be executed.

using System;

namespace ExceptionHandlingDemo

{

class Program

{

static void SomeMethod()

{

try

{

Console.WriteLine("Inside SomeMethod");

int num1 = 10, num2 = 0;

int result = num1 / num2; //Exception will be thrown here

Console.WriteLine($"Result: {result}");

}

finally

{

Console.WriteLine("SomeMethod finally Block");

}

}

static void Main(string[] args)

{

try

{

SomeMethod();

}

catch (Exception)

{

Console.WriteLine("Exception Caught");

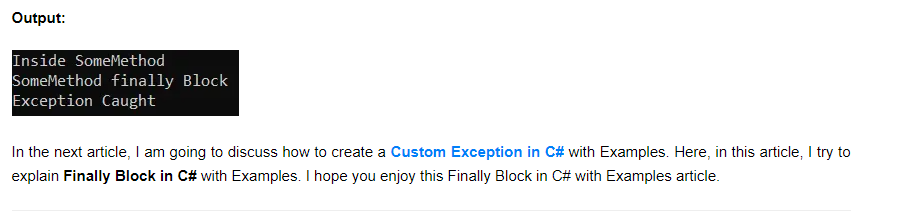
}

Console.ReadKey();

}

}

}

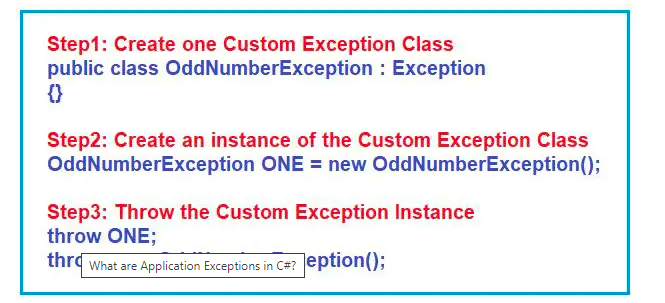


1. **DivideByZeroException**
2. **IndexOutOfRangeException**
3. **FormatException**
4. **SQLException**
5. **NullReferenceException, Etc.**

##### ****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 exceptions at any given point in time. For example, our requirement is that while performing the division operation, we need to check that if the second number is an odd number, then we need to throw an exception. This cannot be handled automatically by the CLR. Then as a user, we need to create our Custom Exception and we need to create an instance of our Custom Exception class and we need to throw that Custom Exception instance using the throw keyword explicitly based on our business requirement.

To raise an Application Exception in C#, we need to adopt the following process. First, we need to create a custom Exception class by inheriting it from the Parent Exception class and then we need to create an instance of the Custom Exception class and then we need to throw that instance.



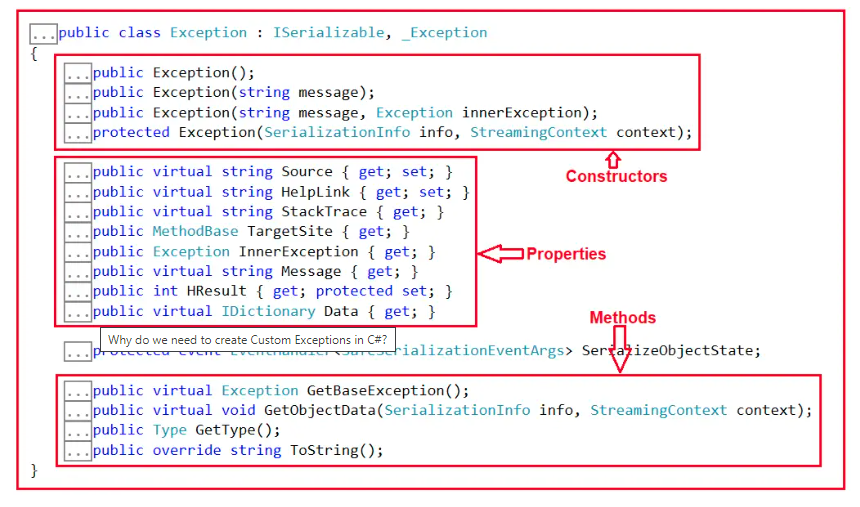
While creating and throwing an object of the Exception class manually, we should not use system exception classes like DivideByZeroException, FormatException, SQLException, etc. instead we should create our own Exception class and create and throw an instance of our own Exception class.

##### ****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#.

##### ****How to Create Own Custom Exception Class in C#?****

Before creating the Custom Exception class, we need to see the class definition of the Exception class as our Custom Exception class is going to be inherited from the parent Exception class. If you go to the definition of Exception class, then you will see the following.



As you can see, the Exception class has some constructors, some virtual and non-virtual properties, and some virtual and non-virtual methods. The virtual members you can override in the child of this Exception class and you can directly consume the non-virtual members using the child class instance.

Now, to create a Custom Exception class in C#, we need to follow the below steps.

1. **Step1:** Define a new class inheriting from the predefined Exception class so that the new class also acts as an Exception class.
2. **Step2:** Then as per your requirement, override the virtual members that are defined inside the Exception class like Message, Source, StackTrace, etc with the required error message.

Let us understand how to create a custom exception in C# with an example. Create a class file with the name OddNumberException.cs and then copy and paste the following code into it. Here, you can see that the OddNumberException class is inherited from the built-in Exception class and here we are re-implementing two virtual properties i.e. Message and HelpLink. Now, we can create an instance of OddNumberException class and if we invoke Message and HelpLink properties, then these two properties are going to be executed from this class only. But if you invoke the Source and StackTrace properties, then those properties are going to be executed from the Exception class only as we have not re-implemented these properties. This is the concept of [**Method Overriding in C#**](https://dotnettutorials.net/lesson/function-overriding-csharp/).

using System;

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";

}

}

//Overriding the HelpLink Property

public override string HelpLink

{

get

{

return "Get More Information from here: https://dotnettutorials.net/lesson/create-custom-exception-csharp/";

}

}

}

}

Now, as per our business logic, we can explicitly create an instance of the OddNumberException class and we can explicitly throw that instance from our application code. For a better understanding, please have a look at the following code. Here, inside the Main method, we are taking two numbers from the user and then checking if the second number is odd or not. If the second number i.e. divisor is odd, then we are creating an instance of the OddNumberException class and throwing that instance. And in the Catch block, we are handling that exception and we simply printing Message, StackTrace, Source, and HellpLink properties. Here, if OddNumberException occurred, then Message and HelpLink properties are going to execute from OddNumberException class and Source and StackTrace properties are going to be executed from the pre-defined parent Exception class.

using System;

namespace ExceptionHandlingDemo

{

class Program

{

static void Main(string[] args)

{

int Number1, Number2, Result;

try

{

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

Number1 = int.Parse(Console.ReadLine());

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

Number2 = int.Parse(Console.ReadLine());

if (Number2 % 2 > 0)

{

//OddNumberException ONE = new OddNumberException();

//throw ONE;

throw new OddNumberException();

}

Result = Number1 / Number2;

Console.WriteLine(Result);

}

catch (OddNumberException one)

{

Console.WriteLine($"Message: {one.Message}");

Console.WriteLine($"HelpLink: {one.HelpLink}");

Console.WriteLine($"Source: {one.Source}");

Console.WriteLine($"StackTrace: {one.StackTrace}");

}

Console.WriteLine("End of the Program");

Console.ReadKey();

}

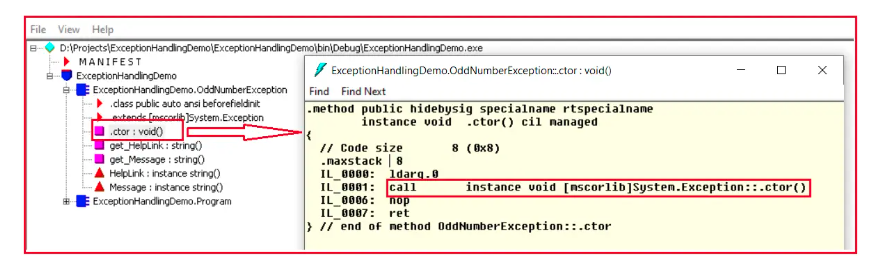
}

}

**Output:**



This fine. But if you see the Exception class, then you will see that the Exception class is having three public constructors. And in our example, we have used the overloaded constructor which does not take any parameter i.e. the default constructor. You might have one question. We have not used any constructor in our Custom Exception class, then how you can say, we are using the Exception class parameterless constructor? Yes, we are using it. This is implicit. And we have discussed this in our Constructor article. Whenever we are making the inheritance relationship the parent class constructor should be accessible to the child class otherwise inheritance is not possible. In our example, we have not defined any constrictor explicitly, so the compiler will provide one constructor and that constructor implicitly calls the Exception class parameterless constructor. To prove this, open the IL Code of our Custom Exception class using the ILDASM tool and you will see the following.



You can see in the above IL code, our compiler provided a default constructor calling the Exception class parameterless constructor.

According to [**Microsoft**](https://docs.microsoft.com/en-us/dotnet/standard/exceptions/how-to-create-user-defined-exceptions), When creating your own exceptions, end the class name of the user-defined exception with the word “Exception”, and implement the three common constructors. To do so, let us modify the OddNumberException class as follows to include three constructors which call the Exception class’s respective constructors. Here, I am not re-implementing the Message virtual property and the reason is, by using the constructor only, I am going to pass the error message while creating the instance.

using System;

namespace ExceptionHandlingDemo

{

//Creating our own Exception Class by inheriting Exception class

public class OddNumberException : Exception

{

public OddNumberException()

{

}

public OddNumberException(string message)

: base(message)

{

}

public OddNumberException(string message, Exception inner)

: base(message, inner)

{

}

//Overriding the HelpLink Property

public override string HelpLink

{

get

{

return "Get More Information from here: https://dotnettutorials.net/lesson/create-custom-exception-csharp/";

}

}

}

}

In our previous example, we used the default constructor while creating an instance of OddNumberException class, now you can use the other overloaded version of the constructor which takes a message as a parameter as shown in the below code.

using System;

namespace ExceptionHandlingDemo

{

class Program

{

static void Main(string[] args)

{

int Number1, Number2, Result;

try

{

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

Number1 = int.Parse(Console.ReadLine());

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

Number2 = int.Parse(Console.ReadLine());

if (Number2 % 2 > 0)

{

//OddNumberException ONE = new OddNumberException();

//throw ONE;

throw new OddNumberException("Odd Number Exception Occured Inside the Main Method of Program Class");

}

Result = Number1 / Number2;

Console.WriteLine(Result);

}

catch (OddNumberException one)

{

Console.WriteLine($"Message: {one.Message}");

Console.WriteLine($"HelpLink: {one.HelpLink}");

Console.WriteLine($"Source: {one.Source}");

Console.WriteLine($"StackTrace: {one.StackTrace}");

}

Console.WriteLine("End of the Program");

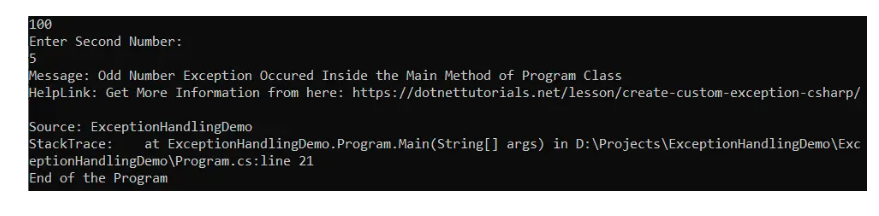
Console.ReadKey();

}

}

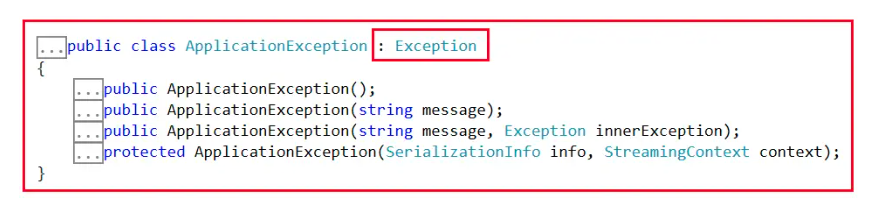
}

Now, run the application and pass the second number as an odd number and you will get the following output. Here, please observe the error message. Whatever message we are passing to the constrictor that we are getting as part of the Message property.



##### ****Can we create a Custom Exception class by Inheriting from ApplicationException class?****

Yes. This is also possible. Even though Microsoft does not recommend to use to ApplicationException class to create a Custom Exception class, but still, we can create it. This is because ApplicationException class is inherited from the Exception. And if you go to the definition of ApplicationException class, then you will also see that this class is having those constructors as Exception class as shown in the below image.



As this ApplicationException class is inherited from the Exception class, all the properties and methods available in Exception are also going to be available in the Custom Exception classes. Let us prove this. Simply modify the Custom class to be inherited from the ApplicationException instead of the Exception class as shown in the below code.

using System;

namespace ExceptionHandlingDemo

{

//Creating our own Exception Class by inheriting Exception class

public class OddNumberException : ApplicationException

{

public OddNumberException()

{

}

public OddNumberException(string message)

: base(message)

{

}

public OddNumberException(string message, Exception inner)

: base(message, inner)

{

}

//Overriding the HelpLink Property

public override string HelpLink

{

get

{

return "Get More Information from here: https://dotnettutorials.net/lesson/create-custom-exception-csharp/";

}

}

}

}

Now, with the above changes in place, run the application and you will also get the same output. So, these are the two ways to create Custom Exception Classes in C#.

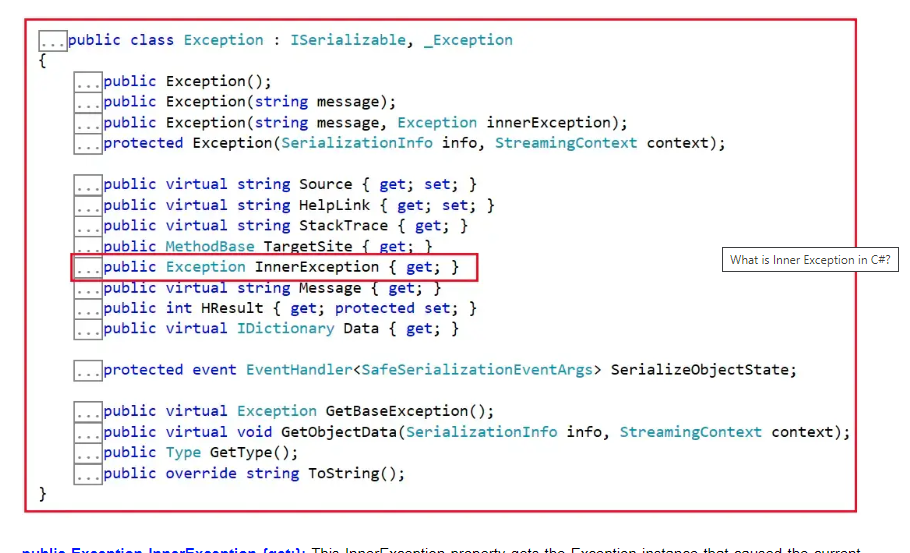
In the next article, I am going to discuss the [**Inner Exception in C#**](https://dotnettutorials.net/lesson/inner-exception-csharp/) with Examples. Here, in this article, I try to explain how to Create **Custom Exceptions in C#** with Examples. I hope you enjoy this How to Create Custom Exceptions in C# with Examples article.

## ****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 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 Inner Exception is and its need in C# with Examples.

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

The Inner Exception in C# is a property of the Exception class. If you go to the definition of the Exception class, then you will see that it is a read-only property i.e. having only get accessor as shown in the below image. As this property is defined in the parent Exception class, so this property is available to all the Child classes including the Custom Exception classes.



**public Exception InnerException {get;}:** This InnerException property gets the Exception instance that caused the current exception. It returns an object that describes the error that caused the current exception. The InnerException property returns the same value that passed into the constructor, or null if the inner exception value was not supplied to the constructor.

To simplify the above definition, we can say that, when there is a series of exceptions, then the **Most Current Exception Obtains the Previous Exception Details in the InnerException Property**. Suppose, in our application, from Method1, we are calling Method2. In Method2, we are getting one exception let’s say divide by zero exception, and then from Method1 we are getting another exception, let’s say Format exception. Then, in this case, the current exception or the latest exception is Format Exception and in the Format Exception InnerException property, you will get the previous exception details i.e. Divide By Zero Exception.

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 that 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 to Understand Inner Exception in C#:****

Let us understand the Inner Exception in C# with an example. Let us first see the complete Example Code first and then I will explain the code. The complete code to understand Inner Exception in C# is given below:

using System;

using System.IO;

using System.Text;

namespace ExceptionHandlingDemo

{

class Program

{

public static void Main()

{

//Outer Try

try

{

int FirstNumber, SecondNumber, Result;

//Inner Try

try

{

//Make sure to Cause Exception in the Try Block

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

FirstNumber = Convert.ToInt32(Console.ReadLine());

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

SecondNumber = Convert.ToInt32(Console.ReadLine());

Result = FirstNumber / SecondNumber;

Console.WriteLine($"Result = {Result}");

}

//Inner Catch

catch (Exception ex)

{

//Make sure this Path Does Not Exist

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

if (File.Exists(filePath))

{

StringBuilder stringBuilder = new StringBuilder();

stringBuilder.Append($"Message: {ex.Message} \n");

stringBuilder.Append($"Source: {ex.Source} \n");

stringBuilder.Append($"HelpLink: {ex.HelpLink} \n");

stringBuilder.Append($"StackTrace: {ex.StackTrace} \n");

stringBuilder.Append($"GetType(): {ex.GetType()} \n");

stringBuilder.Append($"GetType().Name: {ex.GetType().Name} \n");

StreamWriter streamWriter = new StreamWriter(filePath);

streamWriter.Write(stringBuilder.ToString());

streamWriter.Close();

Console.WriteLine("There is a Problem! Plese Try Later");

}

else

{

//To retain the Original Exception pass, this exceptiopm as a parameter

//to the constructor of the current exception

string Message = filePath + " Does Not Exist";

throw new FileNotFoundException(Message, ex);

}

}

}

//Outer Catch

catch (Exception exception)

{

//exception.Message will give the current exception message

//i.e. Message about File Not Found Exception

Console.WriteLine("\nCurrent Exception Details: ");

Console.WriteLine($"Current Exception Message: {exception.Message}");

Console.WriteLine($"Current Exception Source: {exception.Source}");

Console.WriteLine($"Current Exception StackTrace: {exception.StackTrace}");

//Check if InnerException is not null before accessing the InnerException properties

//else, you may get Null Reference Excception

if (exception.InnerException != null)

{

Console.WriteLine("\nInner Exception Details: ");

Console.WriteLine($"Inner Exception Message: {exception.InnerException.Message}");

Console.WriteLine($"Inner Exception Source: {exception.InnerException.Source}");

Console.WriteLine($"Inner Exception StackTrace: {exception.InnerException.StackTrace}");

}

}

Console.WriteLine("Main Method End");

Console.ReadLine();

}

}

}

**Point 1:**First we are asking the user to enter two numbers. In order to understand Inner Exception, we have to make sure this program causes an exception while running the application. To do that we have 3 options

1. You can enter a Character instead of a number which will cause a Format Exception.
2. Or, you can enter a very big number that an integer cannot hold which will cause Over Flow Exception.
3. Or, you can enter zero for the Second Number which will cause the application to throw the Divide By Zero Exception.

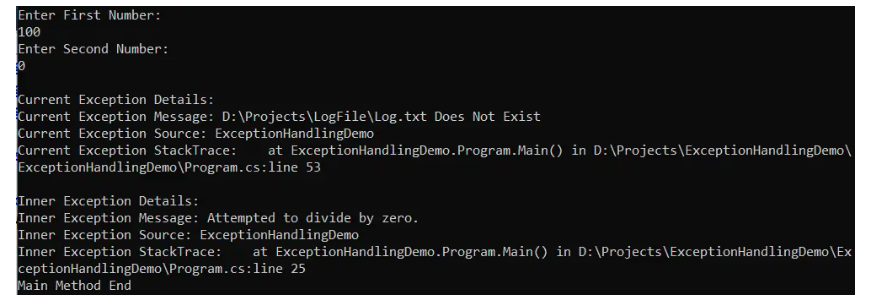
**Point 2:** Once you cause your application to throw an error from the inner try block, then that error is going to be handled by the Inner Catch block. This is because the inner catch is a generic catch block, taking the Exception class as a parameter that can capture any type of exceptions thrown from the corresponding try block.

**Point 3:** Once the catch block catches the exception, then we are trying to log the exception details into a text file. Here, if you are providing the correct file path, then the exception information will be logged into the text file. But to understand Inner Exception, make sure that the file path does not exist. If the File Path does not exist, then we are throwing a File Not Found Exception from the catch block and if you see, we are passing two parameters to the constructor of the File Not Found Exception class. The first parameter specifies the message and the second parameter is the exception (the exception that was thrown from the inner try block) and this exception information will be stored inside the InnerException property.

**Point 4:** Now, the outer catch block will catch the File Not Found Exception which is thrown by the inner catch block. Here, first, we are printing the current exception details and then we are printing the original or the old exception details i.e. the exception which is originally thrown from the Inner Try block. And we can access the old exception details from the Inner Exception property. But before accessing the InnnerExveption properties, please make sure that the InnnerExveption value is not null, else you might get a Null Reference Exception.

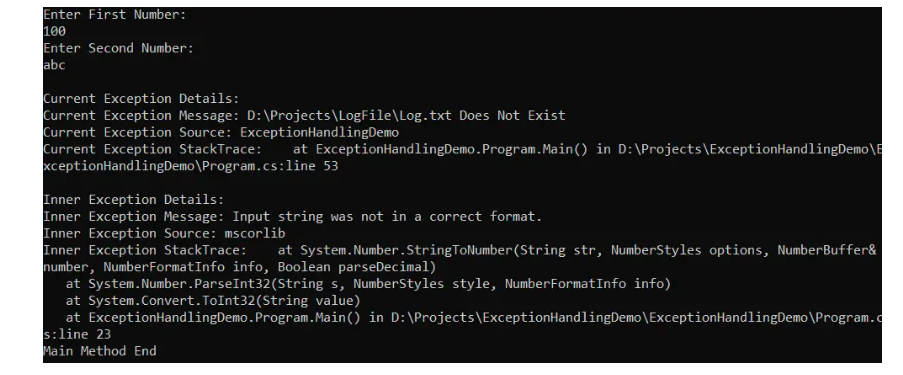
###### **Output1:**

Now, run the application and provide the two input numbers as 100 and 0, this time, the inner try block will throw Divide By Zero Exception and you will see that this Divide By Zero Exception details will be printed by Inner Exception as shown in the below image.



###### **Output2:**

Now, run the application and provide the second value as abc and this time, the inner try block will throw Format Exception and you will see that this Format Exception details will be printed by Inner Exception as shown in the below image



In the next article, I am going to discuss [**Exception Handling Abuse**](https://dotnettutorials.net/lesson/exception-handling-abuse-csharp/)in C#. Here, in this article, I try to explain the **Inner Exception in C#** with examples. I would like to have your feedback. Please post your feedback, question, or comments about this article.