Exception in C#

An application may encounter an error during the execution. When an error occurs, either CLR or program code throws an exception which contains necessary information about the error. There are two types of exceptions in .Net, exceptions generated by the

1 executing program

2 exceptions generated by the CLR.

C# includes built-in classes for every possible exception. All the exception classes are directly or indirectly derived from the **Exception** class. There are two main classes for exceptions - ***SystemException*** and ***ApplicationException***. SystemException is a base class for all CLR generated errors whereas ApplicationException serves as a base class for all application related exceptions, which you want to raise on business rule violation.

## Important Exception Classes:

The following table lists important exception classes available in .Net.

| Exception | Description |
| --- | --- |
| ArgumentException | Raised when a non-null argument that is passed to a method is invalid. |
| ArgumentNullException | Raised when null argument is passed to a method. |
| ArgumentOutOfRangeException | Raised when the value of an argument is outside the range of valid values. |
| DivideByZeroException | Raised when an integer value is divide by zero. |
| FileNotFoundException | Raised when a physical file does not exist at the specified location. |
| FormatException | Raised when a value is not in an appropriate format to be converted from a string by a conversion method such as Parse. |
| IndexOutOfRangeException | Raised when an array index is outside the lower or upper bounds of an array or collection. |
| InvalidOperationException | Raised when a method call is invalid in an object's current state. |
| InvalidCastException | Raised when incompitible types are being converted. |
| KeyNotFoundException | Raised when the specified key for accessing a member in a collection is not exists. |
| NotSupportedException | Raised when a method or operation is not supported. |
| NullReferenceException | Raised when program access members of null object. |
| OverflowException | Raised when an arithmetic, casting, or conversion operation results in an overflow. |
| OutOfMemoryException | Raised when a program does not get enough memory to execute the code. |
| StackOverflowException | Raised when a stack in memory overflows. |
| TimeoutException | The time interval allotted to an operation has expired. |

Every exception class in .Net is derived from the base Exception class. It includes the following important properties using which you can use to get information about the exception when you handle the exception.

| Property | Description |
| --- | --- |
| Message | Provides details about the cause of the exception. |
| StackTrace | Provides information about where the error occurred. |
| InnerException | Provides information about the series of exceptions that might have occurred. |
| HelpLink | This property can hold the help URL for a particular exception. |
| Data | This property can hold arbitrary data in key-value pairs. |
| TargetSite | Provides the name of the method where this exception was thrown. |

## try..catch..finally

C# provides three keywords try, catch and finally to implement exception handling. The try encloses the statements that might throw an exception whereas catch handles an exception if one exists. The finally can be used for any cleanup work that needs to be done.

Try..catch..finally block example:

1. **try**
2. {
3. // Statement which can cause an exception.
4. }
5. **catch**(Type x)
6. {
7. // Statements for handling the exception
8. }
9. **finally**
10. {
11. //Any cleanup code
12. }

If any exception occurs inside the try block, the control transfers to the appropriate catch block and later to the finally block.

But in C#, both catch and finally blocks are optional. The try block can exist either with one or more catch blocks or a finally block or with both catch and finally blocks.

If there is no exception occurred inside the try block, the control directly transfers to finally block. We can say that the statements inside the finally block is executed always. Note that it is an error to transfer control out of a finally block by using break, continue, return or goto.

In C#, exceptions are nothing but objects of the type Exception. The Exception is the ultimate base class for any exceptions in C#. The C# itself provides couple of standard exceptions. Or even the user can create their own exception classes, provided that this should inherit from either Exception class or one of the standard derived classes of Exception class like DivideByZeroExcpetion to ArgumentException etc.

## Uncaught Exceptions

The following program will compile but will show an error during execution. The division by zero is a runtime anomaly and program terminates with an error message. Any uncaught exceptions in the current context propagate to a higher context and looks for an appropriate catch block to handle it. If it can't find any suitable catch blocks, the default mechanism of the .NET runtime will terminate the execution of the entire program.

1. //C#: Exception Handling
2. //Author: rajeshvs@msn.com
3. **using** System;
4. **class** MyClient
5. {
6. **public** **static** **void** Main()
7. {
8. **int** x = 0;
9. **int** div = 100/x;
10. Console.WriteLine(div);
11. }
12. }

The modified form of the above program with exception handling mechanism is as follows. Here we are using the object of the standard exception class DivideByZeroException to handle the exception caused by division by zero.

1. //C#: Exception Handling
2. **using** System;
3. **class** MyClient
4. {
5. **public** **static** **void** Main()
6. {
7. **int** x = 0;
8. **int** div = 0;
9. **try**
10. {
11. div = 100 / x;
12. Console.WriteLine("This linein not executed");
13. }
14. **catch** (DivideByZeroException)
15. {
16. Console.WriteLine("Exception occured");
17. }
18. Console.WriteLine($"Result is {div}");
19. }
20. }

Result from above code is show below:



In the above case, the program do not terminate unexpectedly. Instead, the program control passes from the point where exception occurred inside the try block to the catch blocks. If it finds any suitable catch block, executes the statements inside that catch and continues with the normal execution of the program statements.

If a finally block is present, the code inside the finally block will get also be executed.

1. //C#: Exception Handling
2. **using** System;
3. **class** MyClient
4. {
5. **public** **static** **void** Main()
6. {
7. **int** x = 0;
8. **int** div = 0;
9. **try**
10. {
11. div = 100/x;
12. Console.WriteLine("Not executed line");
13. }
14. **catch**(DivideByZeroException)
15. {
16. Console.WriteLine("Exception occured");
17. }
18. **finally**
19. {
20. Console.WriteLine("Finally Block");
21. }
22. Console.WriteLine($"Result is {div}");
23. }
24. }

Remember that in C#, the catch block is optional. The following program is perfectly legal in C#.

1. //C#: Exception Handling
2. **using** System;
3. **class** MyClient
4. {
5. **public** **static** **void** Main()
6. {
7. **int** x = 0;
8. **int** div = 0;
9. **try**
10. {
11. div = 100/x;
12. Console.WriteLine("Not executed line");
13. }
14. **finally**
15. {
16. Console.WriteLine("Finally Block");
17. }
18. Console.WriteLine($"Result is {div}");
19. }
20. }

But in this case, since there is no exception handling catch block, the execution will get terminated. But before the termination of the program statements inside the finally block will get executed. In C#, a try block must be followed by either a catch or finally block.

## Multiple Catch Blocks

A try block can throw multiple exceptions, which can handle by using multiple catch blocks. Remember that more specialized catch block should come before a generalized one. Otherwise the compiler will show a compilation error.

1. //C#: Exception Handling: Multiple catch
2. **using** System;
3. **class** MyClient
4. {
5. **public** **static** **void** Main()
6. {
7. **int** x = 0;
8. **int** div = 0;
9. **try**
10. {
11. div = 100 / x;
12. Console.WriteLine("Not executed line");
13. }
14. **catch** (DivideByZeroException de)
15. {
16. Console.WriteLine("DivideByZeroException");
17. }
18. **catch** (Exception)
19. {
20. Console.WriteLine("Exception");
21. }
22. **finally**
23. {
24. Console.WriteLine("Finally Block");
25. }
26. Console.WriteLine($"Result is {div}");
27. }
28. }

## Catching all Exceptions

By providing a catch block without brackets or arguments, we can catch all exceptions occurred inside a try block. Even we can use a catch block with an Exception type parameter to catch all exceptions happened inside the try block since in C#, all exceptions are directly or indirectly inherited from the Exception class.

1. //C#: Exception Handling: Handling all exceptions
2. **using** System;
3. **class** MyClient
4. {
5. **public** **static** **void** Main()
6. {
7. **int** x = 0;
8. **int** div = 0;
9. **try**
10. {
11. div = 100 / x;
12. Console.WriteLine("Not executed line");
13. }
14. **catch**
15. {
16. Console.WriteLine("oException");
17. }
18. Console.WriteLine($"Result is {div}");
19. }
20. }

The following program handles all exception with Exception object.

1. //C#: Exception Handling: Handling all exceptions
2. **using** System;
3. **class** MyClient
4. {
5. **public** **static** **void** Main()
6. {
7. **int** x = 0;
8. **int** div = 0;
9. **try**
10. {
11. div = 100 / x;
12. Console.WriteLine("Not executed line");
13. }
14. **catch** (Exception)
15. {
16. Console.WriteLine("oException");
17. }
18. Console.WriteLine($"Result is {div}");
19. }
20. }

## Throwing an Exception

In C#, it is possible to throw an exception programmatically. The 'throw' keyword is used for this purpose. The general form of throwing an exception is as follows.

1. **throw** exception\_obj;

For example, the following statement throws an ArgumentException explicitly.

1. **throw** **new** ArgumentException("Exception");
3. //C#: Exception Handling:
4. **using** System;
5. **class** MyClient
6. {
7. **public** **static** **void** Main()
8. {
9. **try**
10. {
11. **throw** **new** DivideByZeroException("Invalid Division");
12. }
13. **catch** (DivideByZeroException)
14. {
15. Console.WriteLine("Exception");
16. }
17. Console.WriteLine("LAST STATEMENT");
18. }
19. }

## Re-throwing an Exception

The exceptions, which we caught inside a catch block, can re-throw to a higher context by using the keyword throw inside the catch block. The following program shows how to do this.

1. //C#: Exception Handling: Handling all exceptions
2. **using** System;
3. **class** MyClass
4. {
5. **public** **void** Method()
6. {
7. **try**
8. {
9. **int** x = 0;
10. **int** sum = 100 / x;
11. }
12. **catch** (DivideByZeroException)
13. {
14. **throw**;
15. }
16. }
17. }
18. **class** MyClient
19. {
20. **public** **static** **void** Main()
21. {
22. MyClass mc = **new** MyClass();
23. **try**
24. {
25. mc.Method();
26. }
27. **catch** (Exception)
28. {
29. Console.WriteLine("Exception caught here");
30. }
31. Console.WriteLine("LAST STATEMENT");
32. }
33. }

## Standard Exceptions

There are two types of exceptions: exceptions generated by an executing program and exceptions generated by the common language runtime. System.Exception is the base class for all exceptions in C#. Several exception classes inherit from this class including ApplicationException and SystemException. These two classes form the basis for most other runtime exceptions. Other exceptions that derive directly from System.Exception include IOException, WebException etc.

The common language runtime throws SystemException. The ApplicationException is thrown by a user program rather than the runtime. The SystemException includes the ExecutionEngineException, StaclOverFlowException etc. It is not recommended that we catch SystemExceptions nor is it good programming practice to throw SystemExceptions in our applications.

* System.OutOfMemoryException
* System.NullReferenceException
* Syste.InvalidCastException
* Syste.ArrayTypeMismatchException
* System.IndexOutOfRangeException
* System.ArithmeticException
* System.DevideByZeroException
* System.OverFlowException

## User-defined Exceptions

In C#, it is possible to create our own exception class. But Exception must be the ultimate base class for all exceptions in C#. So the user-defined exception classes must inherit from either Exception class or one of its standard derived classes.

1. //C#: Exception Handling: User defined exceptions
2. **using** System;
3. **class** MyException : Exception
4. {
5. **public** MyException(**string** str)
6. {
7. Console.WriteLine("User defined exception");
8. }
9. }
10. **class** MyClient
11. {
12. **public** **static** **void** Main()
13. {
14. **try**
15. {
16. **throw** **new** MyException("RAJESH");
17. }
18. **catch** (Exception)
19. {
20. Console.WriteLine("Exception caught here" + e.ToString());
21. }
22. Console.WriteLine("LAST STATEMENT");
23. }
24. }
25. Create Custom Exception Class in C#
26. We have seen built-in exception classes in the previous section. However, you often like to raise an exception when the business rule of your application gets violated. So, for this you can create a custom exception class by deriving Exception or ApplicationException class.
27. The .Net framework includes **ApplicationException** class since .Net v1.0. It was designed to use as a base class for the custom exception class. However, Microsoft now recommends **Exception** class to create a custom exception class.
28. For example, create InvalidStudentNameException class in a school application, which does not allow any special character or numeric value in a name of any of the students.
29. Example: ApplicationException
30. class Student
31. {
32. public int StudentID { get; set; }
33. public string StudentName { get; set; }
34. }
35. [Serializable]
36. class InvalidStudentNameException : Exception
37. {
38. public InvalidStudentNameException()
39. {
40. }
41. public InvalidStudentNameException(string name)
42. : base(String.Format("Invalid Student Name: {0}", name))
43. {
44. }
46. }
47. Now, you can raise InvalidStudentNameException in your program whenever the name contains special characters or numbers. Use the **throw** keyword to raise an exception.
48. Example: throw custom exception
49. class Program
50. {
51. static void Main(string[] args)
52. {
53. Student newStudent = null;
55. try
56. {
57. newStudent = new Student();
58. newStudent.StudentName = "James007";
60. ValidateStudent(newStudent);
61. }
62. catch(InvalidStudentNameException ex)
63. {
64. Console.WriteLine(ex.Message );
65. }
67. Console.ReadKey();
68. }
69. private static void ValidateStudent(Student std)
70. {
71. Regex regex = new Regex("^[a-zA-Z]+$");
72. if (!regex.IsMatch(std.StudentName))
73. **throw new InvalidStudentNameException(std.StudentName);**
75. }
76. }