Chapter – 5
Exception Handling

Topics Covered:

- Ode to Errors, Bugs, and Exceptions,
- The Role of .NET Exception Handing,
- The System. Exception Base Class,
- Throwing a Generic Exception,
- Catching Exception,
- CLR System – Level Exception (System. System Exception),
- Custom Application- Level Exception (System. System Exception),
- Handling Multiple Exception,
- The Finally Block,
- The Last Chance Exception Dynamically Identifying Application – and System Level Exception Debugging System Exception Using VS. NET,

Ode to Errors, Bugs, and Exceptions [Meaning of errors, Bugs and Exceptions]:

- Errors are mistake or abnormal condition it occurs during the execution of the program.
- Errors also called as bug's causes the program to either run unexpectedly (shows unexpected result) or stop the execution of a program.
- The following different types of errors in programming language. They are:
    - Compile Time Error.
    - Runtime Error.
    - Logical Error.

Compile Time Error:

- Compile errors are those errors that occur at the time of compilation of the program. [If any error is generated at the time of compilation is known as compile time error, in general these are raised while break down the rules and regulation of programming language.]

- Compile Time errors may be further classified as two types:
    1. Syntax error.
    2. Semantic error.

Syntax error:

- Syntax errors occur during development, when you make type mistake in code. For example, instead of writing while, you write WHILE then it will be a syntax error since C# is a case sensitive language.
- Common examples of syntax errors are:
    1. Misspelled variable and function names
    2. Missing semicolons
    3. Improperly matches parentheses, square brackets, and curly braces
    4. Incorrect format in selection and loop statements

Example:

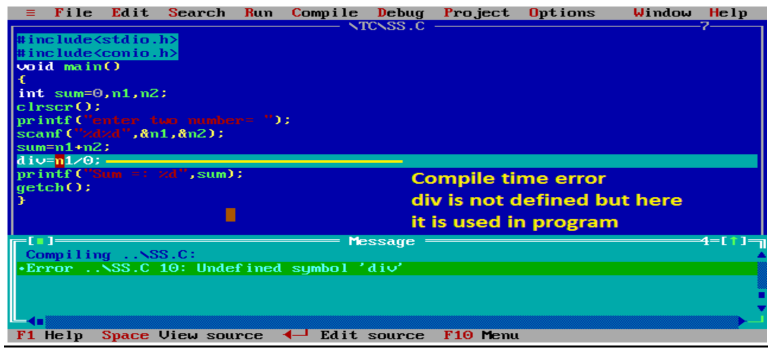
```
bool flag=true;
WHILE (flag)                        //syntax error, since c# is case sensitive
{
 //TO DO:
}
```

- When the rules of the c programming language are not followed, the compiler will show syntax errors.

Example:

Consider the statement **int a, b:**

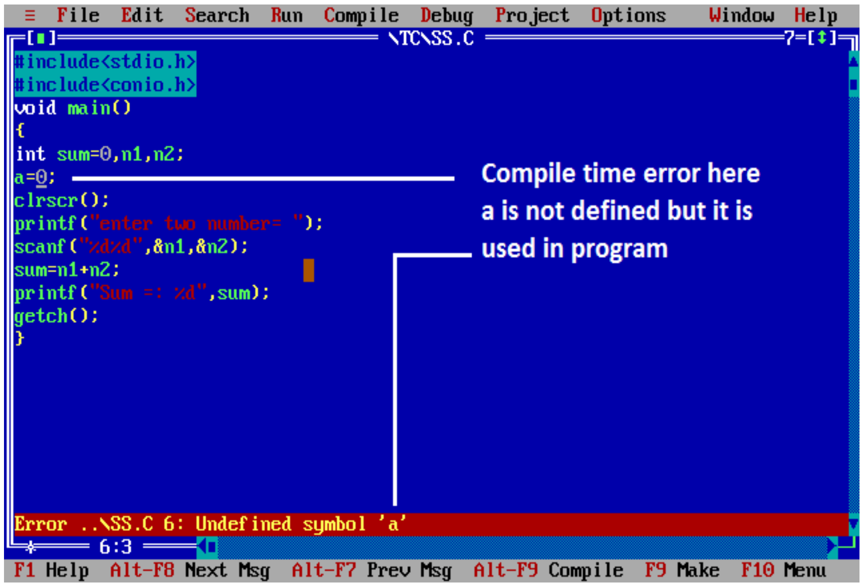
- The above statement will produce syntax error as the statement is terminated with**:** rather than ;.


≡ File Edit Search Run Compile Debug Project Options Window Help
\TC\SS.C
Message
F1 Help Space View source ◄┘ Edit source F10 Menu

```
#include<stdio.h>
#include<conio.h>
void main()
{
int sum=0,n1,n2;
clrscr();
printf("enter two number= ");
scanf("%d%d",&n1,&n2);
sum=n1+n2;
div=n1/0;
printf("Sum =: %d",sum);
getch();
}
```

Compile time error
div is not defined but here
it is used in program

```
Compiling ..\SS.C:
•Error ..\SS.C 10: Undefined symbol 'div'
```


≡ File Edit Search Run Compile Debug Project Options Window Help
\TC\SS.C
Compile time error here
a is not defined but it is
used in program
6:3
F1 Help Alt-F8 Next Msg Alt-F7 Prev Msg Alt-F9 Compile F9 Make F10 Menu

```
#include<stdio.h>
#include<conio.h>
void main()
{
int sum=0,n1,n2;
a=0;
clrscr();
printf("enter two number= ");
scanf("%d%d",&n1,&n2);
sum=n1+n2;
printf("Sum =: %d",sum);
getch();
}
```

```
Error ..\SS.C 6: Undefined symbol 'a'
```

Semantic Errors:

- Semantic errors are reported by the compiler when the statements written in the program are not meaningful to the compiler.
- May be using the wrong variable, the wrong operation, or operations in the wrong order.

Example:

Consider the statement b+c=a;

- In the above statement we are trying to assign value of a in the value obtained by summation of b and c which has no meaning in programming language. The correct statement will be a=b+c;

Logical Error:

- Program is compile and running successfully but we are not getting the expected output due to not written correct logic.
- Also, logical errors could not be detected by the compiler, and thus, programmers have to check the entire coding of a c program line by line.
- Example of Logical errors:
    1. Multiplying when you should be dividing
    2. Adding when you should be subtracting
    3. Opening and using data from the wrong file
    4. Displaying the wrong message

Run Time Error:

- Run Time Error is also called as Exception. This error occurs when application is running but some logic is failed according to CLR.
- Runtime errors occur during execution of the program. These are also called exceptions. This can be caused due to improper user inputs, improper design logic or system errors.
- Runtime errors are those errors that occur during the execution of a program and generally occur due to some illegal operation performed in the program.
- Examples of some illegal operations that may produce runtime errors are:
    1. Dividing a number by zero
    2. Trying to open a file which is not created
    3. Lack of free memory space

```
int a = 5, b = 0;
int result = a / b;                    // DivideByZeroException
```

What is error?

- An **error** is a term used to describe any issue that arises unexpectedly that causes a computer program to not function properly work. Computers can encounter either Software Errors or Hardware Errors.

What is bug?

- In the computer world, a bug is a coding error in a software program. It may cause a program to unexpectedly quit or behave in an accidental manner.
- For example, a small bug may cause a button within a program's interface not to respond when you click it. A more serious bug may cause the program to hang or crash due to an infinite calculation or memory leak.
- From a developer perspective, bugs can be syntax or logic errors within the source code of a program.

What is an Exception?

- An exception is a problem (unexpected errors) that arises during the execution of a program. [An *exception* is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.][An exception is an error that occurs during the execution of the program. The situation arises when an operation is not completed normally.]

- An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.
- An exception can occur for many different reasons. Following are some scenarios where an exception occurs.
    1. A user has entered an invalid data.
    2. A file that needs to be opened cannot be found.
    3. A network connection has been lost in the middle of communications

Common scenarios where exceptions may occur:

1) Scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;                                  //ArithmeticException
```

2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

```
String s=null;
System.out.println(s.length());              //NullPointerException
```

3) Scenario where NumberFormatException occurs

The wrong formatting of any value, may occur NumberFormatException. Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.

```
String s="abc";
int i=Integer.parseInt(s);                   //NumberFormatException
```

4) Scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

```
int a[]=new int[5];
a[10]=50;                                    //ArrayIndexOutOfBoundsException
```

The Role of .NET Exception Handling:

What is an Exception Handling?

The **exception handling** is one of the powerful mechanisms to handle the runtime errors so that normal flow of the application can be maintained.

Advantage of Exception Handling:

The core advantage of exception handling is **to maintain the normal flow of the application**. Exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

```
1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;          //exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;
```

Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform exception handling, rest of the statement will be executed.

- The .NET platform provides a standard technique to send and trap runtime errors: **S**tructured **E**xception **H**andling (SEH).
- Programmers have used this well-defined approach to error handling which is common to all .NET aware languages.
- C# includes built-in classes for every possible exception.
- Identical syntax used to throw & catch exceptions across assemblies, applications & machine boundaries.

- Rather than receiving a cryptic-numerical value that identifies the problem at hand, exceptions are objects that contain a human readable description of the problem.
- Objects also contain a detailed snapshot of the call-stack that eventually triggered the exception.
- The end-user can be provided with the help-link information. Help-link information point to a URL that provides detailed information regarding error at hand.

The Atoms of .NET Exception Handling:

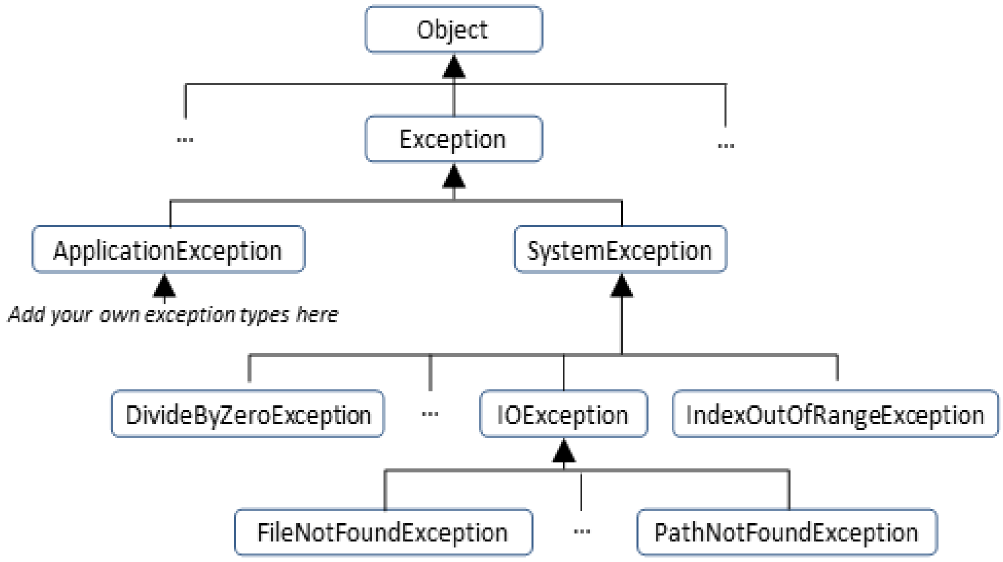
- Programming with structured exception handling involves the use of four interrelated entities:
    1. A **class** type that represents the details of the exception that occurred
    2. A **member** that throws an instance of the exception class to the caller
    3. A block of code on the caller’ s side that **invokes the exception-prone member**
    4. A block of code on the caller’s side that will **process (or catch) the exception** should it occur

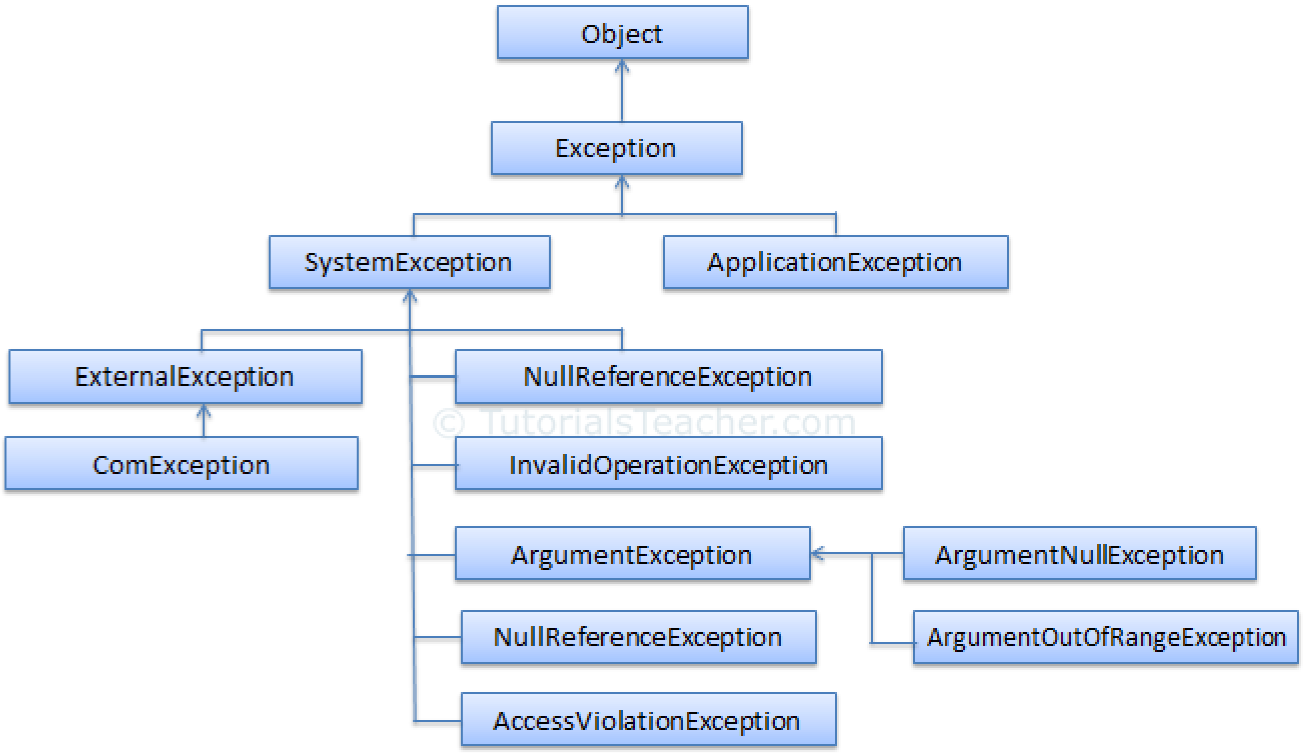
- **try:** try keyword is used for identifying code block which may cause exception at runtime.
- **catch:** catch keyword handle the exception if try block raises exception. The code under try block if raises runtime error, try block sends handler to catch block to handle error.
- **finally:** finally block executed whether exception is raised or not. It is used for cleaning resource and executing set of code.
- **throw:** throw keyword is used for creating user defined exception messages.

The System.Exception Base Class:

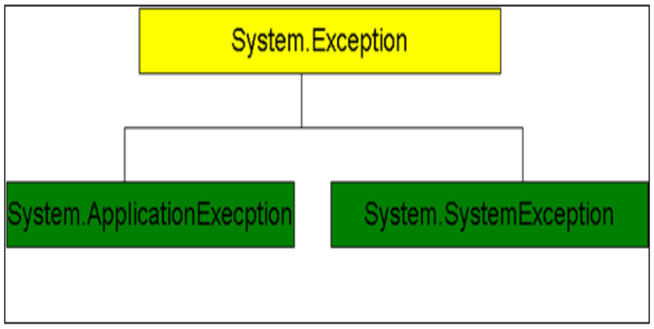
- An application may encounter an error during the execution. These error occurs, either CLR or program code throws an exception.
- Exception contains necessary information about the error.
- Two types of exceptions in .NET: Exceptions generated by the executing program and Exceptions generated by the CLR.

- C# includes built-in classes for every possible exception.
- All the exception classes are directly or indirectly derived from the Exception base class.





- There are two main classes for exceptions
    1. ***System.Exception*** class
    2. ***ApplicationException*** class



- System.Exception is a base class for all CLR generated errors.
- ApplicationException is a base class for all application related exceptions.
- System.Exception class is a base class for all the exception that can occurs during execution of the program.
- No other class derives ApplicationException class by default, because you as a programmer need to derive this class to create your own exceptions classes as per the business rules.

Some important exception classes available in .Net:

1. **ArgumentException:** Raised when a non-null argument that is passed to a method is invalid.
2. **ArgumentNullException:** Raised when null argument is passed to a method.
3. **ArgumentOutOfRangeException:** Raised when the value of an argument is outside the range of valid values.
4. **DivideByZeroException:** Raised when an integer value is divide by zero.
5. **FileNotFoundException**: Raised when a physical file does not exist at the specified location.
6. **IndexOutOfRangeException:** Raised when an array index is outside the lower or upper bounds of an array or collection.

- Every exception class in .Net is derived from the System.Exception base class.
- System.Exception contains important properties (core members).
- These members can be using to get information about the exception when you handle the exception.

Members (properties):

1. **Message:** This returns textual description of a given error. The error-message itself is set as a constructor-parameter.
2. **TargetSite:** This returns name of the method that threw the exception. Source This returns name of the assembly that threw the exception.

3. **HelpLink:** This returns a URL to a help-file which describes the error in detail.
4. **StackTree:** This returns a string that identifies the sequence of calls that triggered the exception.
5. **InnerException:** This is used to obtain information about the previous exceptions that causes the current exception to occur.

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.

Exception handling is done using four keywords,

1. try,
2. catch,
3. finally, and
4. throw

Try block:

- try block is used to enclose (surround) the code that might throw an exception. It must be used within the method.
- **For example**: if you are reading data from a file using the FileReader class it's expected that you handle the IOExceptions associated with using a FileReader object.

- Example: FileNotFoundException, IOException.
- To ensure this happens you can place the statements that deal with creating and using the FileReader object inside a try block:

We need to put code inside try block that might throw an exception. Try block must be within a method of a class.

try block must be followed by either catch or finally block or both.

Syntax of try block:

```
try
{
  //statements that may cause an exception
}
```

Catch block:

The catch block(s) provide a place to handle the exception thrown by the statements within a try block. [The catch block contains an exception handler and some statements used to overcome that exception]. The catch block is defined directly after the try block. It must specify the type of exception it is handling. For example, the FileReader object defined in the code above is capable of throwing a FileNotFoundException or an IOException. We can specify two catch blocks to handle both of those exceptions.

Syntax of catch Block:

```
catch(ExceptionType exOb)
{
  //Exception Handling Code
}
```

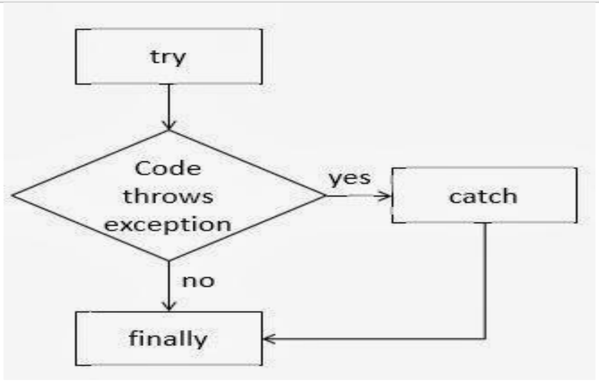
Where ExceptionType is the class of the object we are expecting the statements present in the *try* block to throw.

When the program is executing statements under the *try* block, it checks for any run-time errors. If any error occurred in *try* block then creates an object of that type and passes control to *catch* block by skipping any statements below the problematic statement. If no error occurs then it completely skips *catch* block.

Try Catch Block :

The try and catch keywords form the basis of complete exception handling.

```
try {
      //Statement 1;
      //Statement 2;
      //Statement 3;
    }
catch(Exception e)
    {
      //Exception Handling Code
    }
```



Example of try –catch Block:

```
public class ExceptionExample
{
    public static void main(String[] args)
{
       int number1 = 50;
       int number2 = 0;
       int result;
       try
{
  result = number1 / number2;     // not possible because universal rule is number is not
                                   possible divide by zero
 System.out.println("Result of Division : " + ans);
}
catch(ArithmeticException e)
{
System.out.println("Divide by Zero Error");
}
  }
}
```

Example -2

```
static void Main(string[] args)
{
    Student std = null;

    try
    {
        PrintStudentName(std);
    }
    catch(Exception ex)
    {
        Console.WriteLine(ex.Message );
    }

    Console.ReadKey();
}

private static void PrintStudentName( Student std)
{
    if (std  == null)
        throw new NullReferenceException("Student object is null.");

    Console.WriteLine(std.StudentName);
}
```

PrintStudentName() method raises nullReferenceException if Student object is null.

Example -3

```
using System;
namespace exceptionHandling_example4
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
        int total = 0;
        int[] arr = new int[10] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        try
        {

           for (int i = 0; i <= 10; i++)
           {
              total += arr[i];

           }
           throw new System.Exception();
        }
           catch(Exception ex)
        {
              Console.WriteLine("Unknown error");
              Console.ReadLine();
           }
        }
     }
  }
```

OR

//IndexOutOfRangeException

```
using System;
using System;

class Program
{
    static void Main()
    {
        int[] i = new int[6];
        i[6] = 1;                    // IndexOutOfRangeException is issued.

        Console.WriteLine("i[6] = ", i[6]);
        Console.Read();
    }
}
```

Example -4

```
Using system;
Namespace namespacefinally

Class program
{
Static void main(String[] args)
{
Try
{
```

```
Console.WriteLine(Divide(10,0));
Console.WriteLine(“Thaank you”);
}
Catch (exception ex)
{
Console.WriteLine(ex.Message);
}
Finally
{
Console.WriteLine(“Thank you”);
}
}
Private static int Divide(int num1, int num2)
{
Int num3=(num1/num2);
Return num3;
}
```

Throwing a Generic Exception:

- if any exception occurs, we can throw a specific exception such as:
    1. DivideByZeroException
    2. FileNotFoundException
    3. OutOfMemoryException
- We can also through a general exception directly using Exception class. [how to raise an exception manually.]
- An exception can be raised manually [or exception are send explicitly] by using the throw keyword.
- Any type of exceptions which is derived from *Exception* class can be raised using the throw keyword.

```
using System;
class Test
{
int Max = 100;
public void Fun(int d)
{
if (d > Max)
throw new Exception("crossed limit!!!");
else
Console.WriteLine("speed is ={0}", d);
}
```

```
public static void Main()
   {
Test ob = new Test();
Console.WriteLine("Enter a number:");
int d = int.Parse(Console.ReadLine());
ob.Fun(d);
}
}
```

Using throw keyword, program throws an exception when a problem

value d is greater than 100, then we throw an exception

created a new instance of the System.Exception class, then we have passed a message "crossed limit" to a Message property of Exception class

```
Enter a number: 50
speed is =50
Enter a number: 120
Unhandled Exception: System.Exception: crossed limit!!!
at Test.Fun(Int32 d)
at Test.Main()
```

```
static void Main(string[] args)
{
    Student std = null;

    try
    {
        PrintStudentName(std);
    }
    catch(Exception ex)
    {
        Console.WriteLine(ex.Message );
    }
    Console.ReadKey();
}
private static void PrintStudentName( Student std)
{
    if (std  == null)
        throw new NullReferenceException("Student object is null.");

    Console.WriteLine(std.StudentName);
}
```

Output:

```
Student object is null.
```

In the above example, PrintStudentName() method raises NullReferenceException if Student object is null.

Catching Exceptions:

- A catch block contains a code that will process the exception.
- When the program throws an exception, it can be caught using "**catch"** keyword.
- Once the exception is caught, the members of System.Exception class can be invoked.

- These members can be used to

    1) Display a message about the exception

    2) Store the information about the error in a file and

    3) Send a mail to administrator

```
using System;
class Test    {
int Max = 100;
public void Fun(int d)    {
Try    {
if (d > Max)
throw new Exception("crossed limit!!!");
}
catch (Exception e)    {
Console.WriteLine("Message:{0}", e.Message);
Console.WriteLine("Method:{0}", e.TargetSite);
}
//the error has been handled, continue with the flow of this application
Console.WriteLine("Speed is ={0}", d);
}
```

throwing an exception if *d>100*

Throwing an exception using the *throw new Exception ("Crossed Limit");*

once the exception thrown using the catch keyword to catch the exception and process the exception

```
public static void Main()
{
Test ob = new Test();
Console.WriteLine("Enter a number:");
int d = int.Parse(Console.ReadLine());
ob.Fun(d);
}
}
```

- Enter a number: 12
- Speed is =12

==

- Enter a number: 123
- Message: crossed limit!!!
- Method: Void Fun(Int32)
- Speed is=123

```
class Program
{
    static void Main(string[] args)
    {
        Student newStudent = null;

        try
        {
            newStudent = new Student();
            newStudent.StudentName = "James007";

            ValidateStudent(newStudent);
        }
        catch(InvalidStudentNameException ex)
        {
            Console.WriteLine(ex.Message );
        }

        Console.ReadKey();
    }

    private static void ValidateStudent(Student std)
    {
        Regex regex = new Regex("^[a-zA-Z]+$");

        if (regex.IsMatch(std.StudentName))
             throw new InvalidStudentNameException(std.StudentName);
         }
}
```

Example-3

```
static void Main(string[] args)
{
    int zero = 0;

    try
    {
        int result = 5/zero;  // this will throw an exception

    }
    catch(Exception ex)
    {
        Console.WriteLine("Inside catch block. Exception: {0}", ex.Message );
    }
    finally
    {
        Console.WriteLine("Inside finally block.");

    }
}
```

Example-4

```
class Program
{
    static void Main(string[] args)
    {
        Console.Write("Please enter two numbers: ");

        try
        {
            int num1 = int.Parse(Console.ReadLine());
            int num2 = int.Parse(Console.ReadLine());

            int result = num1 / num2;

            Console.WriteLine("{0} / {1} = {2}", num1, num2, result);
        }
        catch(DivideByZeroException ex)
        {
            LogError(ex);
            Console.Write("Cannot divide by zero. Please try again.");
        }
        catch(InvalidOperationException ex)
        {
            LogError(ex);
            Console.Write("Not a valid number. Please try again.");
        }
        catch(FormatException  ex)
        {
            LogError(ex);
            Console.Write("Not a valid number. Please try again.");
        }

        Console.ReadKey();
    }

}
```

Handling Multiple Exceptions:

- A try block can throw multiple exceptions that can be handled by multiple catch blocks. Remember that a more specialized catch block should come before a generalized one. Otherwise the compiler will show a compilation error.
- try block can follow multiple catch blocks. if there is a chance of getting multiple exceptions then we go for multiple catch blocks to handle all the exceptions at once.

```
try
{
statements..
}
catch()
{
statements..
}
catch()
{
statements..
}
```

Example-1

```
public class CatchDemo {

        public static void main(String[] args) {

                 try

                  {

                   int c[]={1,2,3};// taking an array of size 3

                   System.out.println(c[2]);// here we are printing the value

                   //c[3]=5;// if we uncomment this statement exception raises and cursor
jumps to catch block of ArrayIndexoutofboundsexception without executing the next line.

                   int a=10;//

                   System.out.println(a/0) ;// if we comment the above statement of
c[3]=5; this will execute and here an exception raises as ArithmeticException



                  }

                  catch(ArithmeticException ae)
```

```
                   {
                    System.out.println("number cannot divided by zero"+ae);
                   }
                   catch(ArrayIndexOutOfBoundsException aie)
                   {
                    System.out.println("array index out of bound exception"+aie);
                   }
                  catch(Exception e)
{
                          System.out.println(e);// this block will be executed when there
were no particular exception matches
                  }
                  System.out.println("After try catch block");
         }

}
```

Example-2

```
using System;
 namespace exceptionHandling_example3
{
    class Program
    {
        static void Main(string[] args)
        {
            double Operand1, Operand2;
            double Result = 0.00;
            char Operator;

            Console.WriteLine("This program allows you to perform an operation on two numbers");
             try
            {
```

```
        Console.Write("Enter a number: ");
        Operand1 = double.Parse(Console.ReadLine());
        Console.Write("Enter an operator: ");
        Operator = char.Parse(Console.ReadLine());
        Console.Write("Enter a number: ");
        Operand2 = double.Parse(Console.ReadLine());

        if (Operator != '+' &&
            Operator != '-' &&
            Operator != '*' &&
            Operator != '/')
            throw new Exception(Operator.ToString());

        if (Operator == '/') if (Operand2 == 0)
                throw new DivideByZeroException("Division by zero is not allowed");

        switch (Operator)
        {
            case '+':
                Result = Operand1 + Operand2;
                break;

            case '-':
                Result = Operand1 - Operand2;
                break;

            case '*':
                Result = Operand1 * Operand2;
                break;

            case '/':
                Result = Operand1 / Operand2;
                break;

            default:
                Console.WriteLine("Bad Operation");
                break;
        }
        Console.WriteLine("\n{0} {1} {2} = {3}", Operand1, Operator, Operand2, Result);
    }
    catch (DivideByZeroException ex)
    {
        Console.WriteLine(ex.Message);
    }
    catch (Exception ex)
    {
            Console.WriteLine("\nOperation Error: {0} is not a valid operator", ex.Message);
          }
          Console.ReadKey();
        }
    }
}
```

Example-3

```
class Excep
{
 public static void main(String[] args)
 {
  try
  {
   int arr[]={1,2};
   arr[2]=3/0;
  }
  catch(ArithmeticException ae)
  {
   System.out.println("divide by zero");
  }
  catch(ArrayIndexOutOfBoundsException e)
  {
   System.out.println("array index out of bound exception");
  }
 }
}
```

Example for Unreachable Catch block

While using multiple **catch** statements, it is important to remember that exception sub classes inside **catch** must come before any of their super classes otherwise it will lead to compile time error.

```
class Excep
{
 public static void main(String[] args)
 {
  try
  {
   int arr[]={1,2};
   arr[2]=3/0;
  }
  catch(Exception e)    //This block handles all Exception
  {
   System.out.println("Generic exception");
  }
  catch(ArrayIndexOutOfBoundsException e)    //This block is unreachable
  {
   System.out.println("array index out of bound exception");
  }
 }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Exception_Handling
{
    class Program
    {
        static void Main(string[] args)
        {

            try
            {
                int num1, num2;
                decimal result;

                Console.WriteLine("Divide Program. You Enter 2 number and we return result");
                Console.WriteLine("Enter 1st Number: ");
                num1 = Convert.ToInt32(Console.ReadLine());
                Console.WriteLine("Enter 2nd Number: ");
num2 = Convert.ToInt32(Console.ReadLine());

                result =(decimal)num1 / (decimal)num2;
                Console.WriteLine("Divide : " + result.ToString());
                Console.ReadLine();
            }
           //Multiple Catch block to handle exception

            catch (DivideByZeroException dex)
            {
                Console.WriteLine("You have Entered 0");
                Console.WriteLine("More Details about Error: \n\n" + dex.ToString() + "\n\n");
                goto label;
            }
            catch (FormatException fex)
            {
                Console.WriteLine("Invalid Input");
                Console.WriteLine("More Details about Error: \n\n" + fex.ToString() + "\n\n");
                goto label;
            }

            //Parent Exception: Catch all type of exception

            catch (Exception ex)
            {
                Console.WriteLine("Othe Exception raised" + ex.ToString() + "\n\n");
                goto label;
            }

           //Finally block: its always executed

            finally
            {
                Console.WriteLine("Finally Block: For Continue Press Enter and for Exit press Ctrl + c");
                Console.ReadLine();
            }
        }
    }
}
```

Generic catch Statements:

C# supports a generic catch block that does not explicitly define the type of exception. That is, we can write:

```
catch
{
Console.WriteLine("Some error has occurred");
 }
```

But, using this type of catch blocks indicates that the programmer is un-aware of the type of exception that is going to occur, which is not acceptable. Hence it is always advised to use specific type of exceptions.

Rethrowing Exceptions

To re-throw exception, simply make use of the "throw" keyword within a catch-block.

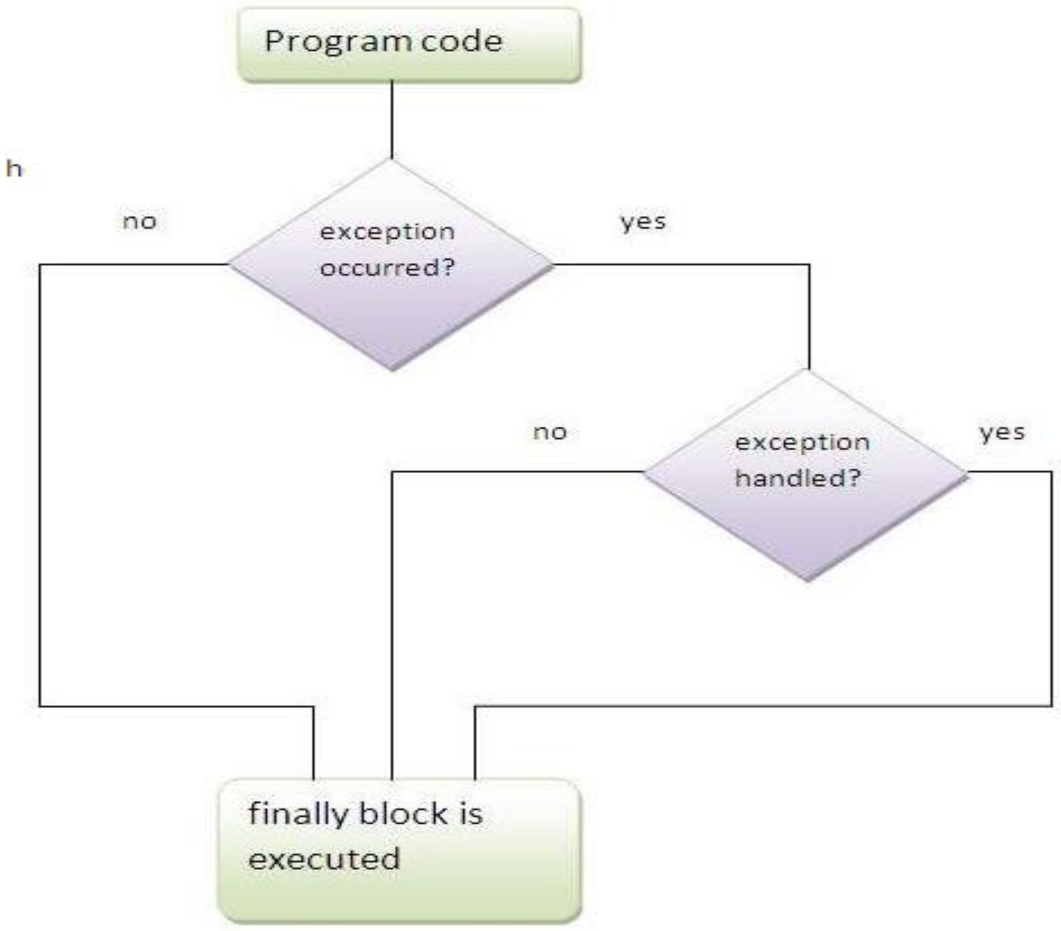
```
try
{ . . . }
catch(CarIsDeadException e)
{
```

throw e

}

The Finally Block:

- A **finally statement** must be linked with a **try statement**.
- It identifies a block of statements that needs to be executed regardless of whether or not an **exception occurs** within the try block.
- It means if you write a finally block , the code should execute after the execution of try block or catch block.
- After all other try-catch processing is complete, the **code inside the finally block executes**. It is not mandatory to include a finally block at all, but if you do, it will run regardless of whether an exception was thrown and handled by the try and catch parts of the block.



The finally block is used to:

1. Clean-up any allocated memory.
2. Close the database connection.
3. Close the file which is in use.

Syntax of Finally block:-

```
try
{
   //statements that may cause an exception
}
catch ( … )
{
  //error handling code
}
finally
{
  //statements to be executed
}
```

Example-1:

```
Class program  {
Static void main(String[] args)   {
Try  {
System.Console.WriteLine(Divide(10,0));
System.Console.WriteLine("Thank you");
}
Catch (exception ex)  {
System.Console.WriteLine(ex.Message);
}
}
Private static int Divide(int num1, int num2)  {
Int num3=(num1/num2);
Return num3;
}
```

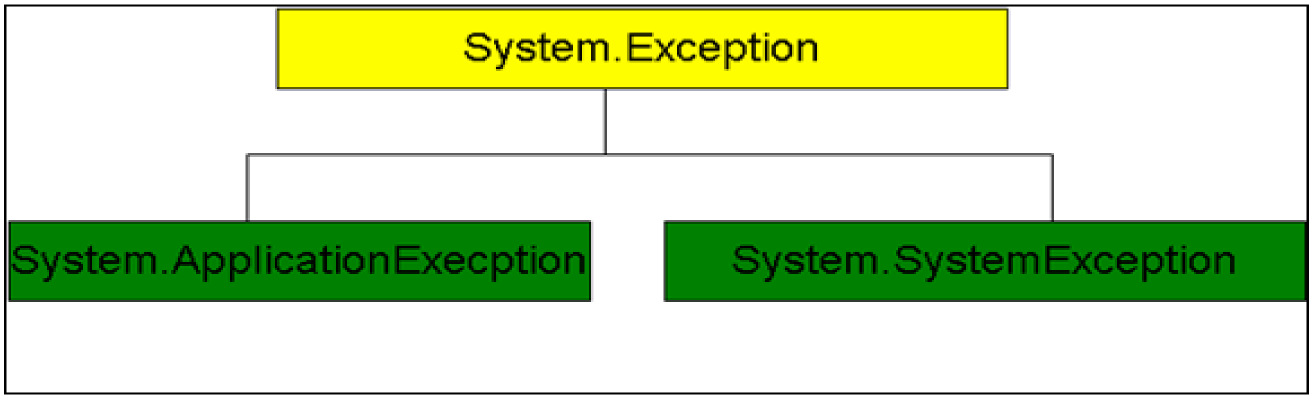
Example-2:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Exception_Handling
{
    class Program
    {
        static void Main(string[] args)
        {

            try
            {
                int num1, num2;
                decimal result;

                Console.WriteLine("Divide Program. You Enter 2 number and we return result");
                Console.WriteLine("Enter 1st Number: ");
                num1 = Convert.ToInt32(Console.ReadLine());
                Console.WriteLine("Enter 2nd Number: ");
num2 = Convert.ToInt32(Console.ReadLine());

                result =(decimal)num1 / (decimal)num2;
                Console.WriteLine("Divide : " + result.ToString());
                Console.ReadLine();
            }
           //Multiple Catch block to handle exception

            catch (DivideByZeroException dex)
            {
                Console.WriteLine("You have Entered 0");
                Console.WriteLine("More Details about Error: \n\n" + dex.ToString() + "\n\n");
                goto label;
            }

            catch (FormatException fex)
            {
                Console.WriteLine("Invalid Input");
                Console.WriteLine("More Details about Error: \n\n" + fex.ToString() + "\n\n");
                goto label;
            }

            //Parent Exception: Catch all type of exception

            catch (Exception ex)
            {
                Console.WriteLine("Othe Exception raised" + ex.ToString() + "\n\n");
                goto label;
        }

       //Finally block: its always executed

        finally
        {
           Console.WriteLine("Finally Block: For Continue Press Enter and for Exit press Ctrl + c");
           Console.ReadLine();
        }
     }
  }
}
```

CLR System-Level Exceptions (System.SystemException):

Exception Class can be classified into two types. They are:

1. System. system exception.
2. System. Application exception.



- C# includes built-in classes for every possible exception.
- Example:

  → DivideByZeroException

  → OutOfMemoryException

  → IndexOutOfRangeException etc.
- All these exception classes defined by CLR or BCL (Base Class Libraries)
- Exceptions thrown by methods in the BCL are called system-exceptions.
- **Main idea:,**When an exception-type derives from System.SystemException, it can be concluded that the exception was raised by .NET runtime.
- If we fail to handle a raised exception, the operating-system will trigger a "last chance exception".

System Exception: System Exception is predefined Exception class in C# that is ready to use in programming. Just choose which exception may occur in your code and use it in catch block.

System exceptions that are thrown by the CLR called as system exception. These exceptions are regarded as non-recoverable, fatal errors.

The System.SystemExecption class is the base class for all the predefined system exceptions.

The following table describes some of the classes derived from the System.SystemException class.

Exception Classes	Description
System.IO.IOException	It handles the I/O errors.
System.IndexOutOfRangeException	It handles the errors generated when a method refers to an array element, which is out of bound.
System.NullReferenceException	It handles errors generated during the process of dereferencing a null object.
System.DivideByZeroException	It handles errors generated during the process of dividing the dividend with zero.
System.InvalidCastException	It handles errors generated during typecasting.
System.OutOfMemoryException	It handles memory allocation to the application errors.

Custom Application-Level Exceptions (System.ApplicationException):

- If you want users to be able to programmatically distinguish between some error conditions, you can create your own user-defined exceptions.
- An exception that is raised explicitly by the programmer in the application on its own condition is an "Application Exception".
- It can also refer to as a "User Defined Exception".
- Use of t**hrow** keyword.
- While creating custom exception class it is best practice to keep class name with the end of Exception word.
- For example, OutofStockException, NumberNotFoundException, IamHeroException etc.
- The following procedure to develop a User Defined Exception.

How to develop our own User Defined Exception:

1. Inherit a class from the Exception class.

   For example:

   <Your class> : Exception

2. Override the string message from the exception class.

```
Public override string Message
{
 get
    {
       Return "Display the error Message when Exception Occur"
     }
   }
```

3. In the other class, if an exception occurs that throws an exception for the creation of an object of your own exception class then do something.

   Throws new <Own Exception name>

Example-1:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace user_defined_exception
{
    class Program
    {
        static void Main(string[] args)
        {
            int acceptorder;
            Console.WriteLine("Welcome to Shopping Site:\nHow many headphone you want to buy
(Total 10 in Stock):");
            acceptorder = Convert.ToInt32(Console.ReadLine());
            try
            {
                if (acceptorder == 10 || acceptorder < 10)
                {
                    Console.WriteLine("Congratulations! You have bought {0} headphones", acceptorder);
                    Console.ReadLine();
                }
                else
                {
                    throw (new OutofStockException("OutofStockException Generated: The number of item
you want to buy is out of stock. Please enter total item number within stock"));
                }
            }
            catch (OutofStockException oex)
            {
                Console.WriteLine(oex.Message.ToString());
                Console.ReadLine();
            }

        }
    }

    //Creating Custome Exception - OutofStockException
    public class OutofStockException : Exception
    {
        public OutofStockException(string message)
            : base(message)
        {
        }
    }
}
```

OUTPUT:

```
Welcome to Shopping Site:
How many headphone you want to buy (Total 10 in Stock):
4
Congratulations! You have bought 4 headphones
Welcome to Shopping Site:
How many headphone you want to buy (Total 10 in Stock):
12
OutofStockException Generated: The number of item you want to buy is out of stock. Please enter
total item number within stock
```

Explanation:

1. we have created a custom exception class **OutofStockException** which is catches by catch block when user enter larger number than the stock available. All the user defined exception class must be derived from its base class **Exception**.

```
public class OutofStockException : Exception
  {
     public OutofStockException(string message)
        : base(message)
     {
     }
  }
```

2. All the exception class must have a constructor. However, you may define numbers of constructors based on your requirement. Here we have created only one constructors to keep program simple and easy to understand. This constructors will be used later to throw a string message.

```
public class OutofStockException : Exception
  {
     public OutofStockException(string message)
        : base(message)
     {
     }
  }
```

3. Now in the main method, we asks user for buying a headphone and asks the total number of headphone he/she want to buy. If the number of buying headphone is equal or less than the stock a congratulations message appear but if the total number of buying headphone is larger than stock available it raise **OutofStockException**.

```
try
  {
    if (acceptorder == 10 || acceptorder < 10)
    {
    Console.WriteLine("Congratulations! You have bought {0} headphones", acceptorder);
    Console.ReadLine();
    }
    else
    {
    throw (new OutofStockException("OutofStockException Generated: The number of item
you want to buy is out of stock. Please enter total item number within stock"));
    }
  }
 catch (OutofStockException oex)
  {
    Console.WriteLine(oex.Message.ToString());
    Console.ReadLine();
  }
```

Example-2:

```
using System;
namespace userdefined_exception
{
    class Program
    {
        class billingException:ApplicationException
        {
            public billingException():base("value can not be less than 1000")
            {
            }
             public int calculate(int amount)
             {
                 int total;
                 if (amount < 1000)
            {
               throw new billingException();
            }
            else
            {
               total = amount - 1000;
               Console.WriteLine("amount greater than 1000 is:  "+total );
            }
            return total;

      }
   }

      static void Main(string[] args)
      {
         try
         {
            billingException obj = new billingException();
            Console.WriteLine("Enter  one amount value");
            int m = int.Parse(Console.ReadLine());
            obj.calculate(m);
         }
         catch (billingException ex)
         {
            Console.WriteLine(ex.Message);
            Console.ReadLine();
         }
         finally
            {
               Console.WriteLine("press enter for exit");
               Console.ReadLine();
            }

      }
   }
}
```

Example-3:

```
using System;
namespace MindstickCustomeExceptionDemo
{
public class MindstickEmployeeListNotFoundException : Exception
{
public MindstickEmployeeListNotFoundException()
{
}
public MindstickEmployeeListNotFoundException(string message):base(message)
{
}
}
class CustomException
{
static void Main(string[] args)
{
try
{
throw new MindstickEmployeeListNotFoundException("\n\n::Mindstick Employee List not found::\n\n");
}
catch (MindstickEmployeeListNotFoundException ce)
{
Console.WriteLine(ce.ToString());
}
25Console.ReadKey();
26}
27}
28}
```

Code Description: -

1. Using System (include the System namespace).

2. Create a namespace with MindstickCustomeExceptionDemo named.

4. Create a public class with MindstickEmployeeListNotFoundException named and inherit the Exception class.

6. Create a class constructor of MindstickEmployeeListNotFoundException without parameter.

9. Create a class constructor of MindstickEmployeeListNotFoundException with parameter.

13. Create a class with CustomException named.

15. Create Main method.

17. Start try block.

19. Thrown the exception with its exception class name.

21. This catch block.

23. Print the exception message.

Example-4:

```
using System;
public class InvalidAgeException : Exception
{
    public InvalidAgeException(String message)
        : base(message)
    {

    }
}
public class TestUserDefinedException
{
    static void validate(int age)
    {
        if (age < 18)
        {
            throw new InvalidAgeException("Sorry, Age must be greater than 18");
        }
    }
```

```
  public static void Main(string[] args)
  {
     try
     {
        validate(12);
     }
     catch (InvalidAgeException e) { Console.WriteLine(e); }
     Console.WriteLine("Rest of the code");
  }
}
```

Output:

```
InvalidAgeException: Sorry, Age must be greater than 18
Rest of the code
```

Example-5:

```
using System;
namespace UserDefinedException
{
  class TestTemperature
  {
    static void Main(string[] args)
    {
      Temperature temp = new Temperature();
      try
      {
        temp.showTemp();
      }
      catch(TempIsZeroException e)
      {
        Console.WriteLine("TempIsZeroException: {0}", e.Message);
      }
      Console.ReadKey();
    }
  }
}
public class TempIsZeroException: Exception
{
  public TempIsZeroException(string message): base(message)
  {
  }
}
public class Temperature
{
  int temperature = 0;
  public void showTemp()
  {
    if(temperature == 0)
    {
      throw (new TempIsZeroException("Zero Temperature found"));
    }
    else
    {
      Console.WriteLine("Temperature: {0}", temperature);
    }
  }
}
```

What is an exception?

An Exception can be anything which interrupts the normal flow of the program. When an exception occurs program processing gets terminated and doesn't continue further. In such cases we get a system generated error message. The good thing about exceptions is that they can be handled. We will cover the handling part later in this same tutorial.

When an exception can occur?
Exception can occur at runtime (known as runtime exceptions) as well as at compile-time (known Compile-time exceptions).

Reasons for Exceptions
There can be several reasons for an exception. For example, following situations can cause an exception – Opening a non-existing file, Network connection problem, Operands being manipulated are out of prescribed ranges, class file missing which was supposed to be loaded and so on.

Difference between error and exception

Errors indicate serious problems and abnormal conditions that most applications should not try to handle. Error defines problems that are not expected to be caught under normal circumstances by our program. For example memory error, hardware error, JVM error etc.
Exceptions are conditions within the code. A developer can handle such conditions and take necessary corrective actions. Few examples -

- DivideByZero exception
- NullPointerException
- ArithmeticException
- ArrayIndexOutOfBoundsException

Advantages of Exception Handling

- Exception handling allows us to control the normal flow of the program by using exception handling in program.
- It throws an exception whenever a calling method encounters an error providing that the calling method takes care of that error.
- It also gives us the scope of organizing and differentiating between different error types using a separate block of codes. This is done with the help of try-catch blocks.

Why to handle exception?
If an exception is raised, which has not been handled by programmer then program execution can get terminated and system prints a non user friendly error message.

An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

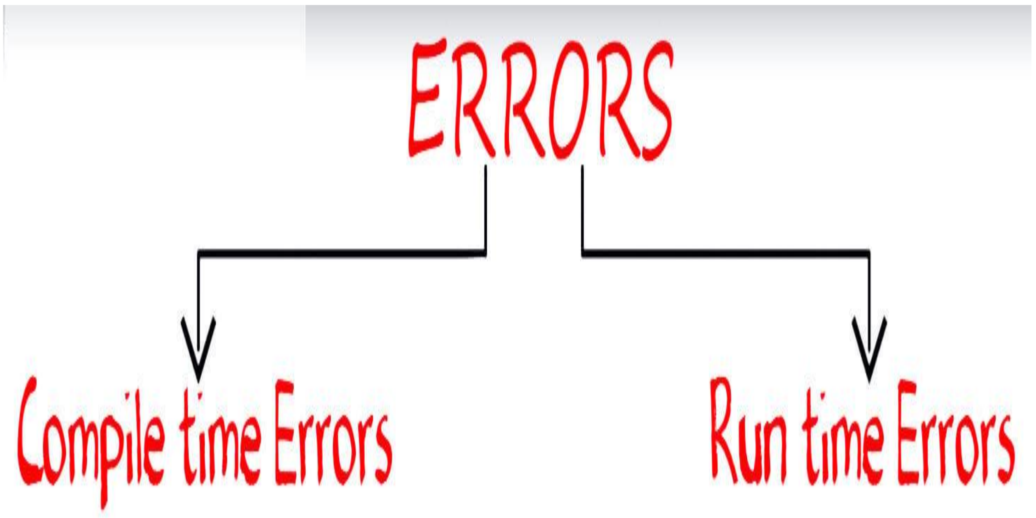
```
/*
 * C# Program to Demonstrate Multiple Exceptions
 */
using System;
class Exercise
{
    static void Main()
    {
        double Num1, Num2;
         double Result = 0.00;
         char op;
         try
         {
             Console.Write("Enter your First Number :  ");
```

```
      Num1 = double.Parse(Console.ReadLine());
      Console.Write("Enter an Operator  (+, -, * or /): ");
      op = char.Parse(Console.ReadLine());
      if (op != '+' && op != '-' &&
        op != '*' && op != '/')
        throw new Exception(op.ToString());
      Console.Write("Enter your Second Number :");
      Num2 = double.Parse(Console.ReadLine());
      if (op == '/')
        if (Num2 == 0)
          throw new DivideByZeroException("Division by zero is not allowed");
      Result = Calculator(Num1, Num2, op);
      Console.WriteLine("\n{0} {1} {2} = {3}", Num1, op, Num2, Result);
    }
    catch (FormatException)
    {
      Console.WriteLine("The number you typed is not valid");
    }
    catch (DivideByZeroException ex)
    {
      Console.WriteLine(ex.Message);
    }
    catch (Exception ex)
    {
      Console.WriteLine("Operation Error: {0} is not a valid op", ex.Message);
    }
    Console.Read();
  }

  static double Calculator(double v1, double v2, char op)
  {
    double Result = 0.00;

    switch (op)
    {
    case '+':
       Result = v1 + v2;
       break;
    case '-':
       Result = v1 - v2;
       break;
```

```
        case '*':
              Result = v1 * v2;
              break;
        case '/':
              Result = v1 / v2;
              break;
        }
        return Result;
    }
}
```

Here is the output of the C# Program:

```
Enter Your First Number : 10
Enter an Operator  (+, -, * or /) : ,
Operation Error : , is not a Valid Operator
```



```
static void ReadFile(string fileName)
{
    // Exceptions could be thrown in the code below
    try
    {
        TextReader reader = new StreamReader(fileName);
        string line = reader.ReadLine();
        Console.WriteLine(line);
        reader.Close();
    }
    catch (FileNotFoundException fnfe)
    {
        // Exception handler for FileNotFoundException
        // We just inform the user that there is no such file
        Console.WriteLine(
           "The file '{0}' is not found.", fileName);
    }
    catch (IOException ioe)
    {
        // Exception handler for other input/output exceptions
        // We just print the stack trace on the console
        Console.WriteLine(ioe.StackTrace);
    }
}


static void Main()
{
    try
    {
        string fileName = "WrongFileName.txt";
        ReadFile(fileName);
    }
    catch (Exception e)
    {
        throw new ApplicationException("Smth. bad happened", e);
    }
}
static void ReadFile(string fileName)
{
    TextReader reader = new StreamReader(fileName);
    string line = reader.ReadLine();
    Console.WriteLine(line);
    reader.Close();
}
```