Exception Handling

An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

Exception handling is the process of responding to exceptions when a computer program runs. An exception occurs when an unexpected event happens that requires special processing.

Examples include a user providing abnormal input, a file system error being encountered when trying to read or write a file, or a program attempting to divide by zero.

Exception handling attempts to gracefully handle these situations so that a program (or worse, an entire system) does not crash.

Exception handling can be performed at both the software (as part of the program itself) and hardware levels (using mechanisms built into the design of the CPU).

Exceptions have the following properties:

- Exceptions are types that all ultimately derive from System.Exception.
- Use a try block around the statements that might throw exceptions.
- Once an exception occurs in the try block, the flow of control jumps to the first associated exception handler that is present anywhere in the call stack. In C#, the catch keyword is used to define an exception handler.
- If no exception handler for a given exception is present, the program stops executing with an error message.
- Do not catch an exception unless you can handle it and leave the application in a known state. If you catch System.Exception, rethrow it using the throw keyword at the end of the catch block.
- If a catch block defines an exception variable, you can use it to obtain more information about the type of exception that occurred.
- Exceptions can be explicitly generated by a program by using the throw keyword.
- Exception objects contain detailed information about the error, such as the state of the call stack and a text description of the error.
- Code in a finally block is executed even if an exception is thrown. Use a finally block to release resources, for example to close any streams or files that were opened in the try block.

Some important exception classes in C# and its base classes

EXCEPTION TYPE	BASE TYPE
Exception	Object
SystemException	Exception
IndexOutOfRangeException	SystemException
NullReferenceException	SystemException
AccessViolationException	SystemException
InvalidOperationException	SystemException
ArgumentException	SystemException
ArgumentNullException	ArgumentException
ArgumentOutOfRangeException	ArgumentException
ExternalException	SystemException
COMException	ExternalException
SEHException	ExternalException

Some frequently required exception types

SystemException

SystemException houses all exceptions related to, well, the system. These include all runtime-generated errors, such as System.IO.IOException, which is thrown when an I/O error occurs.

IndexOutOfRangeException

The IndexOutOfRangeException is thrown anytime an array or collection is accessed using an index outside its bounds.

NullReferenceException

If you attempt to use an object that is considered a null object, the NullReferenceException will be thrown.

AccessViolationException

An AccessViolationException is thrown when unmanaged code attempts to access memory which is unallocated. For many .NET applications, this will never occur, due to how .NET handles managed vs unmanaged code.

Managed code is code that .NET compiles and executes using the common language runtime. Conversely, unmanagedcode compiles into machine code, which is not executed within the safety of the CLR.

InvalidOperationException

It is typically thrown when the state of an object cannot support the particular method call being attempted for that object instance.

ArgumentException

As the name implies, ArgumentException is thrown when a call is made to a method using an invalid argument.

ArgumentNullException

ArgumentNullException is inherited from ArgumentException, but is thrown specifically when a method is called that doesn’t allow an argument to be null.

ArgumentOutOfRangeException

Another child of ArgumentException, the ArgumentOutOfRangeException error is thrown when a method expects argument values within a specified range, yet the provided argument falls outside those bounds.

Consider below applications to understand the concept of exceptions handling mechanism.

Application 1 :
Application which demonstrates exception prone code.

```
using System;
class Program
{
        static void Main(string[] args)
        {
       // Program which generates exception
       int no1 = 10;
       int no2 = 0;
       int ans1 = no1 / no2;

       Console.WriteLine("Code successfully executes");
        }
}
```

Application 2:

Application which handles the exception using try catch block.

```
using System;
class Program
{
    static void Main(string[] args)
    {
        // Program which handles the exception with try catch blck
        try
        {
            int no3 = 10;
            int no4 = 0;
            int ans2 = no3 / no4;
            Console.WriteLine("Hello");
        }
        catch (Exception e)
        {
            Console.WriteLine("Inside catch block");
            Console.WriteLine(e);
        }
        Console.WriteLine("Code succesfully executes");
        }
}
```

Application 3:

Application which handles the exception with try catch and finally block

```
using System;
class Program
{
    static void Main(string[] args)
    {
       try
        {
            int no5 = 10;
            int no6 = 0;
            int ans3 = no5 / no6;
        }
        catch (Exception e)
        {
            Console.WriteLine("Inside catch block");

            Console.WriteLine(e);
        }
        finally
        {
            Console.WriteLine("Inside finally block");
        }
        Console.WriteLine("Code succesfully executes");
          }
}
```

Application 4:

Application which generates the exception but there is no matching catch block

```
using System;
class Program
{
    static void Main(string[] args)
    {
        try
        {
            int no7 = 10;
            int no8 = 0;
            int ans4 = no7 / no8;
        }
        catch (NullReferenceException e)
        {
            Console.WriteLine("Inside catch block");
            Console.WriteLine(e);
        }
        finally
        {
            Console.WriteLine("Finally block is executed");
        }
        Console.WriteLine("Rest of the code");
         }
}
```