Exception Handling in C#

A runtime error is known as an **exception**. The exception will cause the abnormal termination of the program execution. So these errors (exceptions) are very dangerous because whenever the exception occurs in the programs, the program gets terminated abnormally on the same line where the error gets occurred without executing the next line of code.

Objects of exception classes are responsible for abnormal termination of the program whenever runtime errors (exceptions) occur. These exception classes are predefined under BCL (Base Class Libraries) where a separate class is provided for each and every different type of exception like IndexOutOfRangeException, FormatException, NullReferenceException, DivideByZeroException, FileNotFoundException, SQLException, OverFlowException, etc.

Each exception class provides a specific exception error message. All the exception classes are responsible for abnormal termination of the program as well as after abnormal termination of the program they will be displaying an error message which specifies the reason for abnormal termination i.e. they provide an error message specific to that error.

So, whenever a runtime error (exception) occurs in a program, first the exception manager under the CLR (Common Language Runtime) identifies the type of error that occur 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.

The process of catching the exception for converting the CLR given exception message to end-user understandable message or 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 following advantages

- We can stop the abnormal termination of application.
- We can perform any corrective action that may resolve the problem occurring due to abnormal termination.
- Displaying a user-friendly error message, so that the client can resolve the problem provided if it is under his control.

Exception handling uses the **`try`**, **`catch`**, and **`finally`** keywords to try actions that may not succeed, to handle failures when you decide that it's reasonable to do so, and to clean up resources afterward. Exceptions can be generated by the common language runtime (CLR), by .NET or third-party libraries, or by application code. Exceptions are created by using the **`throw`** keyword.

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.
- Don't 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.

When we implement multiple catch blocks in C#, then at any given point of time only one catch block going to be executed and other catch blocks will be ignored. We need to write multiple catch blocks in C# for a single try block to execute some logic specific to an exception.

Example:

```
static void CodeWithCleanup()
{
    int a, b, c;
    FileStream? file = null;
    FileInfo? fileInfo = null;
    string path = @"C:\Test\file.txt";
    try
    {
        Console.WriteLine("ENTER ANY TWO NUBERS");
        a = int.Parse(Console.ReadLine());
        b = int.Parse(Console.ReadLine());
        c = a / b;
        Console.WriteLine("C VALUE = " + c);

        fileInfo = new FileInfo(path);
        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    catch (UnauthorizedAccessException e)
    {
        Console.WriteLine(e.Message);
    }
    catch (System.IO.IOException e)
    {
        Console.WriteLine("Error reading from {0}. Message = {1}", path,
e.Message);
    }
    catch (DivideByZeroException dbe)
    {
        Console.WriteLine("2nd number should not be zero");
    }
    catch (FormatException fe)
```

```
    {
        Console.WriteLine("enter only integer number");
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
    finally
    {
        file?.Close();
    }
}
```

Because an exception can occur at any time within the try block before the `OpenWrite()` call, or the `OpenWrite()` call itself could fail, we aren't guaranteed that the `file` is open when we try to close it. The finally block adds a check to make sure that the `FileStream` object isn't null before you call the `Close()` method. Without the null check, the finally block could throw its own `NullReferenceException`, but throwing exceptions in `finally` blocks should be avoided if it's possible.

A database connection is another good candidate for being closed in a `finally` block. Because the number of connections allowed to a database server is sometimes limited, you should close database connections as quickly as possible. If an exception is thrown before you can close your connection, using the finally block is better than waiting for garbage collection.