Exception Types

In C#, the exceptions are divided into two types: (1) System exception – Compiler-generated exceptions and (2) Application exception – User-defined exceptions

Compiler-generated Exceptions

Some exceptions are thrown automatically by the .NET runtime when basic operations fail. These exceptions and their error conditions are listed below:

1. **`ArithmeticException`**:
   A base class for exceptions that occur during arithmetic operations, such as **DivideByZeroException** and **OverflowException**.
2. **`ArrayTypeMismatchException`**:
   Thrown when an array can't store a given element because the actual type of the element is incompatible with the actual type of the array.
3. **`DivideByZeroException:`**
   Thrown when an attempt is made to divide an integral value by zero.
4. **`IndexOutOfRangeException`**:
   Thrown when an attempt is made to index an array when the index is less than zero or outside the bounds of the array.
5. **`InvalidCastException:`**
   Thrown when an explicit conversion from a base type to an interface or to a derived type fails at runtime.
6. **`NullReferenceException`**:
   Thrown when an attempt is made to reference an object whose value is null.
7. **`OutOfMemoryException`**:
   Thrown when an attempt to allocate memory using the new operator fails. This exception indicates that the memory available to the common language runtime has been exhausted.
8. **`OverflowException`**:
   Thrown when an arithmetic operation in a checked context overflows.
9. **`StackOverflowException`**:
   Thrown when the execution stack is exhausted by having too many pending method calls; usually indicates a very deep or infinite recursion.
10. **`TypeInitializationException`**:
    Thrown when a static constructor throws an exception and no compatible catch clause exists to catch it.

User-defined Exceptions

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 exception at

any given point of time. To create and throw an object of exception class by us, we have two different options.

- Create the object of a predefined **Exception** class where we need to pass the error message as a parameter to its constructor and then **throw** that object so that whenever the exception occurs the given error message gets displayed.
- Define a new class of type exception where we need to override Message property of the Exception class and throw that class object by creating it.

Example:

```
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";
            }
        }
    }
    //Creating our own Exception Class by inheriting Exception class and passing
    //necessary parameters to base class constructor
    [Serializable]
    public class UserAlreadyLoggedInException : Exception
    {
        public string UserName { get; }
        public UserAlreadyLoggedInException(string message) : base(message) { }
        public UserAlreadyLoggedInException(string message, Exception
                            innerException) : base(message, innerException) { }
        public UserAlreadyLoggedInException(SerializationInfo info,
                            StreamingContext context) : base(info, context) { }
        public UserAlreadyLoggedInException(string message, string name)
                                                                : this(message)
        {
            UserName = name;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                throw new UserAlreadyLoggedInException("User Already logged in",
"Dhaval");
            }
```

```
            catch (UserAlreadyLoggedInException ex)
            {
                Console.WriteLine(ex.Message);
            }

            int x, y, z;
            Console.WriteLine("ENTER TWO INTEGER NUMBERS:");
            x = int.Parse(Console.ReadLine());
            y = int.Parse(Console.ReadLine());
            try
            {
                if (y % 2 > 0)
                {
                    //OddNumberException ONE = new OddNumberException();
                    //throw ONE;
                    throw new OddNumberException();
                }
                z = x / y;
                Console.WriteLine(z);
            }
            catch (OddNumberException one)
            {
                Console.WriteLine(one.Message);
            }
            Console.WriteLine("End of the program");
            Console.ReadKey();
        }
    }
}
```

throw Statement

Exception objects that describe an error are created and then thrown with the `throw` keyword. The runtime then searches for the most compatible exception handler(`catch` block). Programmers should throw exceptions when one or more of the following conditions are true:

- The method can't complete its defined functionality. For example, if a parameter to a method has an invalid value:

  ```
  static void CopyObject(SampleClass original)
  {
      _ = original ?? throw new ArgumentException("Parameter cannot be null",
  nameof(original));
  }
  ```

- An inappropriate call to an object is made, based on the object state. One example might be trying to write to a read-only file. In cases where an object state doesn't allow an operation, throw an instance of `InvalidOperationException` or an object based on a derivation of this class. The following code is an example of a method that throws an `InvalidOperationException` object:

  ```
  public class ProgramLog
  ```

```
{
    FileStream logFile = null!;
    public void OpenLog(FileInfo fileName, FileMode mode) { }

    public void WriteLog()
    {
        if (!logFile.CanWrite)
        {
            throw new InvalidOperationException("Logfile cannot be read-
only");
        }
        // Else write data to the log and return.
    }
}
```

- When an argument to a method causes an exception. In this case, the original exception should be caught and an `ArgumentException` instance should be created. The original exception should be passed to the constructor of the `ArgumentException` as the `InnerException` parameter:

```
static int GetValueFromArray(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException ex)
    {
        throw new ArgumentException("Index is out of range", nameof(index),
ex);
    }
}
```

Starting with C# 7.0, throw can be used as an expression as well as a statement. This allows an exception to be thrown in contexts that were previously unsupported. These include the conditional operator, the null-coalescing operator and an expression-bodied lambda or method.

Exception Class Properties

Exceptions contain a property named **StackTrace**. This string contains the name of the methods on the current call stack, together with the file name and line number where the exception was thrown for each method. A StackTrace object is created automatically by the common language runtime (CLR) from the point of the throw statement, so that exceptions must be thrown from the point where the stack trace should begin.

All exceptions contain a property named **Message**. This string should be set to explain the reason for the exception. Information that is sensitive to security shouldn't be put in the message text. In addition to Message, **ArgumentException** contains a property named **ParamName** that should be set to the name of the argument that caused the exception to be thrown. In a property setter, ParamName should be set to value.