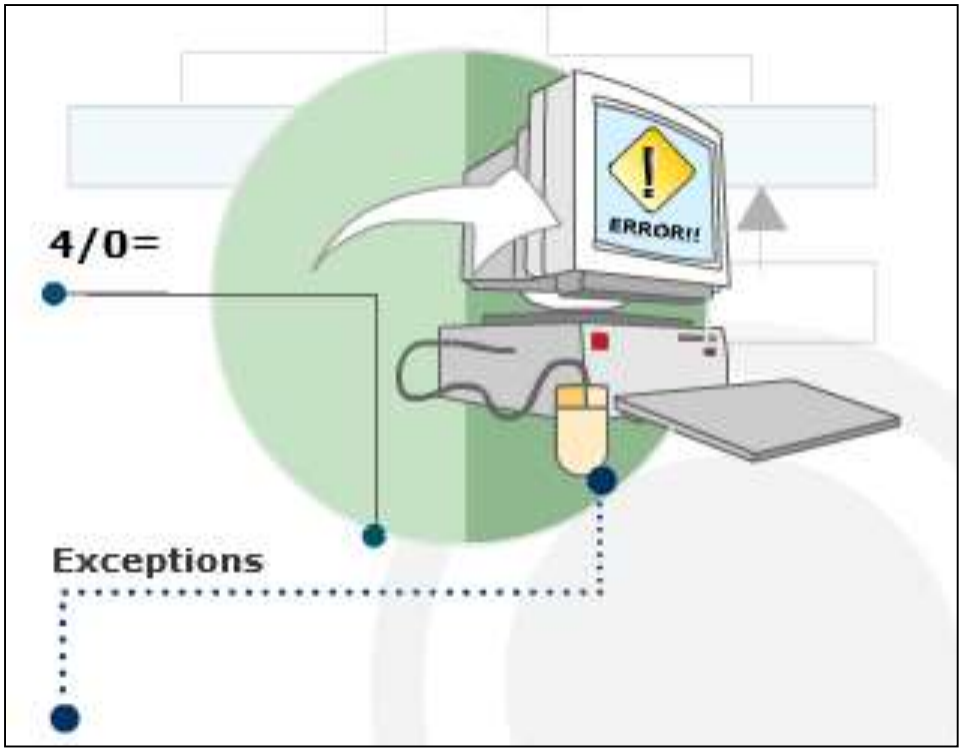
Session: 11

Exception Handling

Objectives

- Define and describe exceptions
- Explain the process of throwing and catching exceptions
- Explain nested **`try`** and multiple **`catch`** blocks
- Define and describe custom exceptions

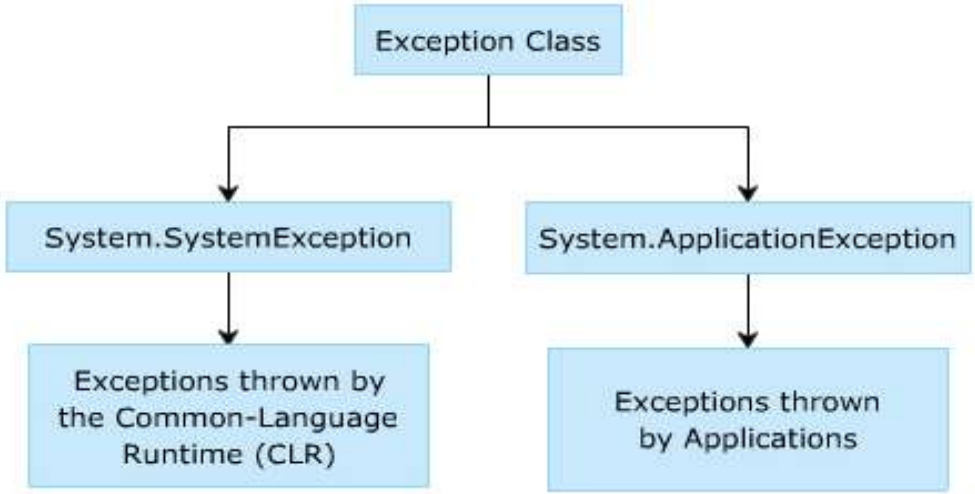
- An exception is an error (abnormal event) that occurs during program execution that prevent a certain task from being completed successfully



Types of Exceptions in C#

- **System-level Exceptions**:
  - are thrown by the system, by the CLR.
  - For example, exceptions are thrown due to failure in database connection or network connection.

- **Application-level Exceptions**:
  - are thrown by user-created applications.
  - For example, exceptions are thrown due to arithmetic operations or referencing any null object.



System.Exception Class

- is the base class that handles all exceptions.

Method	Description
Equals	Determines whether objects are equal
GetBaseException	Returns a type of Exception class when overridden in a derived class
GetHashCode	Returns a hash function for a particular type
GetObjectData	Stores information about serializing or deserializing a particular object with information about the exception when overridden in the inheriting class
GetType	Retrieves the type of the current instance
ToString	Returns a string representation of the thrown exception

Properties	Descriptions
Message	Is a message which indicates the reason for the exception.
Source	Provides the name of the application or the object that caused the exception.
StackTrace	Provides exception details on the stack at the time the exception was thrown.
InnerException	Returns the `Exception` instance that caused the current exception.

Snippet

```
class ExceptionProperties  {
    static void Main(string[] args) {
       byte numOne = 200;
       byte numTwo = 5;
       byte result = 0;
       try {
         result = checked((byte)(numOne * numTwo));
         Console.WriteLine("Result = {0}", result);
       }
       catch (OverflowException objEx) {
            Console.WriteLine("Message : {0}",objEx.Message);
            Console.WriteLine("Source : {0}", objEx.Source);
            Console.WriteLine("TargetSite : {0}",objEx.TargetSite);
            Console.WriteLine("StackTrace : {0}", objEx.StackTrace);
       }
       catch (Exception objEx) {
            Console.WriteLine("Error : {0}", objEx);
       }
    }
}
```

InvalidCastException Class

- is thrown when an explicit conversion from a base type to another type fails.

Snippet

```
class InvalidCastError {
   static void Main(string[] args) {
       try {
          string obj = "numOne";
          int result = (int)obj;
          Console.WriteLine("Value of numOne = {0}", result);
       }
       catch(InvalidCastException objEx) {
         Console.WriteLine("Message : {0}", objEx.Message);
         Console.WriteLine("Error : {0}", objEx);
       }
       catch (Exception objEx){
         Console.WriteLine("Error : {0}", objEx);
       }
   }
}
```

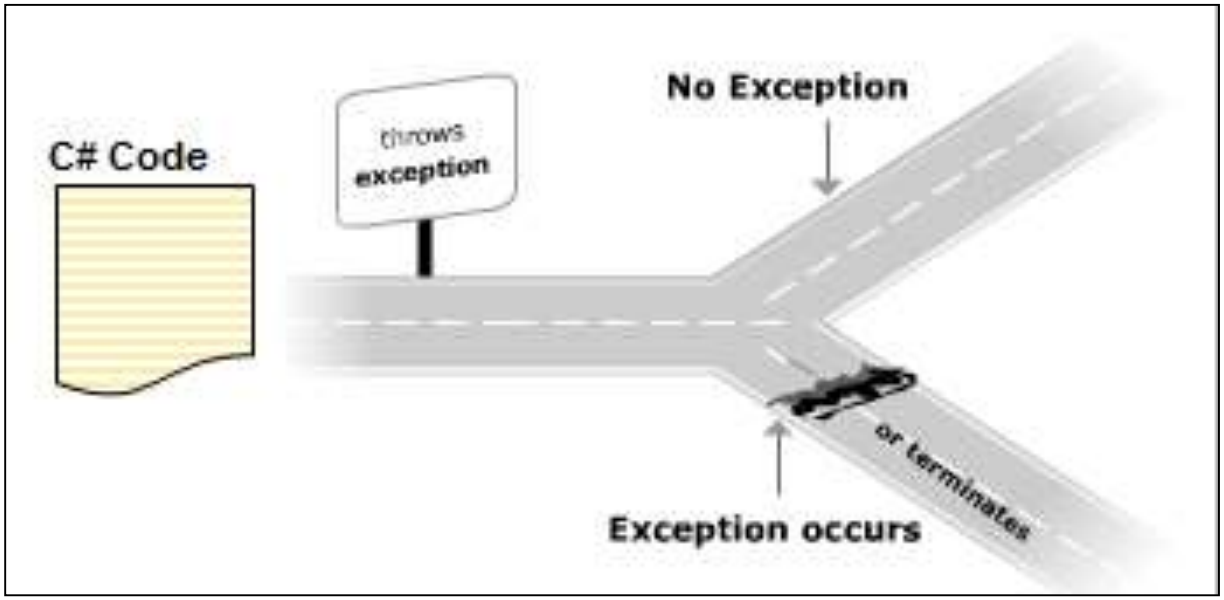
ArrayTypeMismatchException Class

- is thrown when the data type of the value being stored is incompatible with the data type of the array.

Snippet

```
class ArrayMisMatch {
    static void Main(string[] args){
      double[] salary = { 1000, 2000, 3000 };
      float[] bonus = new float[3];
      try {
        salary.CopyTo(bonus, 0);
      }
      catch (ArrayTypeMismatchException objType) {
        Console.WriteLine("Error: " + objType);
      }
      catch (Exception objEx) {
        Console.WriteLine("Error: " + objEx);
      }
    }
}
```

- An exception arises when an operation cannot be completed normally. In such situations, the system throws an error.



- The error is handled through the process of **exception handling**.

The throw Statement

- Exceptions also can be programmatically raised using the `throw` keyword.

Syntax

```
throw exceptionObject;
```

- where,
  - `throw`: Specifies that the exception is thrown programmatically.
  - `exceptionObject`: Is an instance of a particular exception class.

The `finally` Statement

- In general, if any of the statements in the `try` block raises an exception, the `catch` block is executed and the rest of the statements in the `try` block are ignored.
- Sometimes, it becomes mandatory to execute some statements irrespective of whether an exception is raised or not. In such cases, a `finally` block is used.
- The `finally` block is an optional block and it appears after the `catch` block. It encloses statements that are executed regardless of whether an exception occurs.

Syntax

```
finally
{
        // cleanup code;
}
```

Nested `try` and Multiple `catch` Blocks

- Exception handling code can be nested in a program. In nested exception handling, a `try` block can enclose another `try-catch` block.

- In addition, a single `try` block can have multiple `catch` blocks that are sequenced to catch and handle different type of exceptions raised in the `try` block.

- Features of the nested `try` block:

consists of multiple (inner) `try-catch` constructs that starts with a `try` block, which is called the outer `try` block.

If an exception is thrown by a inner nested `try` block, the control passes to its corresponding nested `catch` block.

However, if the inner `catch` block does not contain the appropriate error handler, the control is passed to the outer `catch` block.

Implementing Custom Exceptions

- Custom exceptions can be created by deriving from the `Exception` class, the `SystemException` or `ApplicationException` class.

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

public class CustomExceptionDemo {
    static void Main(string[] args) {
        try {
            throw new CustomError ("This illustrates creation and catching of
            custom exception");
        }
        catch(CustomError ex) {
            Console.WriteLine(ex.Message);
        }
    }
}
```

Summary

- Exceptions are errors that encountered at run-time.
- Exception-handling allows you to handle methods that are expected to generate exceptions.
- The `try` block should enclose statements that may generate exceptions while the `catch` block should catch these exceptions.
- The `finally` block is meant to enclose statements that need to be executed irrespective of whether or not an exception is thrown by the `try` block.
- Nested `try` blocks allow you to have a `try-catch-finally` construct within a `try` block.
- Multiple `catch` blocks can be implemented when a `try` block throws multiple types of exceptions.
- Custom exceptions are user-defined exceptions that allow users to handle system and application-specific runtime errors.