Delegate in C#

- A **delegate** declaration defines a reference type that can be used to encapsulate a method with a specific signature. A delegate instance encapsulates a static or an instance method. Delegates are roughly similar to function pointers in C++; however, delegates are type-safe and secure.

- Why use delegates

When you need to call a series of methods, you can use a single delegate to call a series of methods, or you can call two methods the same in signature using delegate. Another use of delegate is that you can pass methods as arguments to other methods.

A delegate can be declared using the **delegate** keyword followed by a function signature

- **[access modifier] delegate [return type] [delegate name]([parameters])**

Ex: public delegate void MyDelegate(string msg);

Delegate example

```
namespace ConsoleApplication.Day_7.DelegatesInCSharp
{
    //A delegate is a type safe function pointer.

    public delegate void PrintHelloMethodDelegate(string message);

    public class Program
    {
        public static void Main(string[] args)
        {
            PrintHelloMethodDelegate helloMethodDelegate =
                                        new
PrintHelloMethodDelegate(DelegateDemo.PrintHello);
            helloMethodDelegate("This is delegate demo...");
            //DelegateDemo.PrintHello("This is delegate demo...");
            Console.ReadLine();
        }
    }
    public class DelegateDemo
    {
        public static void PrintHello(string message)
        {
            Console.WriteLine(message);
        }
    }
}
```

Multicast Delegate

a **delegate** that manages the references to multiple handler functions is known as a multicast delegate.

All of the functions that the multicast delegate references will be called when the delegate is called.

All method signatures should match if you want to use a delegate to call multiple methods.

Multicast Delegates have the capacity to hold and invoke multiple methods.

Multicast Delegates, also referred to as Combinable Delegates, are required to meet requirements such as the Delegate's return type having to be void.

Multicast Delegates, also referred to as Combinable Delegates, are required to meet requirements such as the Delegate's return type having to be void.

Multicast Delegate instance by joining two Delegates. Delegates are used in the order that they are added.

Points :

- Every method is called using the FIFO (First in, First out) principle.
- Methods are added to delegates using the + or += operator.
- Methods can be eliminated from the delegates list using the – or -= operator.

Example of multicast delegate

```
using System;
using System.CodeDom;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace ConsoleApplication.Day_7.MulticastDelegate
{
    public delegate void DelegateDemo(out int number);
    public class MulticastDelegate
    {
        public static void Main(string[] args)
        {
            int number1 = 10;
            int number2 = 10;
            string stringNo1 = "10";
            string stringNo2 = "10";

            Console.WriteLine(number1 + number2);
            Console.WriteLine(stringNo1 + stringNo2);

            //Approach 1 to create multicast delegates
            DelegateDemo del1 = new DelegateDemo(MethodOne);
            DelegateDemo del2 = new DelegateDemo(MethodTwo);
            DelegateDemo del3 = new DelegateDemo(MethodThree);

            DelegateDemo del4 = del1 + del2 + del3;



            DelegateDemo del = new DelegateDemo(MethodOne);
            del += MethodTwo;
            del += MethodThree;
            del -= MethodThree;

            int outValue = 1;
            del(out outValue);

            Console.WriteLine("Delegate returned value : " + outValue);
            Console.ReadLine();
        }

        public static void MethodOne(out int number)
        {
            number = 10;
        }
        public static void MethodTwo(out int number)
        {
            number = 20;
        }
        public static void MethodThree(out int number)
        {
            number = 30;
        }
    }
}
```

Exception Handling

- An exception is defined as an event that occurs during the execution of a program that is unexpected by the program code.

- The actions to be performed in case of occurrence of an exception is not known to the program. In such a case, we create an exception object and call the exception handler code.

- The execution of an exception handler so that the program code does not crash is called **exception handling**.

- Exception handling is important because it gracefully handles an unwanted event, an exception so that the program code still makes sense to the user.

Exceptions provide a way to transfer control from one part of a program to another. C# exception handling is built upon four keywords: **try**, **catch**, **finally**, and **throw**.

- **try** − A try block identifies a block of code for which particular exceptions is activated. It is followed by one or more catch blocks.
- **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
- **finally** − The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.
- **throw** − A program throws an exception when a problem shows up. This is done using a throw keyword.

Exception Classes in C#

- C# exceptions are represented by classes.
- The exception classes in C# are mainly directly or indirectly derived from the **System.Exception** class.
- Some of the exception classes derived from the System.Exception class are the **System.ApplicationException** and **System.SystemException** classes.
- The **System.ApplicationException** class supports exceptions generated by application programs. Hence the exceptions defined by the programmers should derive from this class.
- The **System.SystemException** class is the base class for all predefined system exception.

provides some of the predefined exception classes derived from the Sytem.SystemException class –

- **System.IO.IOException :** Handles I/O errors.
- **System.IndexOutOfRangeException** : Handles errors generated when a method refers to an array index out of range.
- **System.ArrayTypeMismatchException :** Handles errors generated when type is mismatched with the array type.
- **System.NullReferenceException :** Handles errors generated from referencing a null object.
- **System.DivideByZeroException :** Handles errors generated from dividing a dividend with zero.
- **System.InvalidCastException :** Handles errors generated during typecasting.
- **System.OutOfMemoryException:** Handles errors generated from insufficient free memory.
- **System.StackOverflowException :** Handles errors generated from stack overflow.

C# Exception Handling Keywords

- Try
- Catch
- Finally
- throw

Syntax of try catch

try {

// statements causing exception

} catch(ExceptionName e1) {

// error handling code

}

Example of try-catch

```
public class Program
{
       static void Main(string[] args)
        {


             try
              {
                    int[] myNumbers = { 1, 2, 3 };
                    Console.WriteLine(myNumbers[10]);
              }
              catch (Exception e)
              {
                      Console.WriteLine("Something went wrong.");
              }
                    Console.ReadLine();
       }
}
```

Finally:

The **finally** statement lets you execute code, after **try….catch** block.

```
public class Program
{
  static void Main(string[] args)
    {


     try
      {
        int[] myNumbers = { 1, 2, 3 };
        Console.WriteLine(myNumbers[10]);
      }
      catch (Exception e)
      {
         Console.WriteLine("Something went wrong.");
      }
      finally
      {
         Console.WriteLine("The 'try catch' is finished.");
      }
         Console.ReadLine();
     }
}
```

Throw keyword

- The throw statement allows you to create a custom error.
- The throw statement is used together with an **exception class**. There are many exception
  classes: ArithmeticException, FileNotFoundException, IndexOutOfRangeException, Time OutException.etc

DivideByZeroException exception example

```
using System;
namespace ErrorHandlingApplication
{
    class DivNumbers
    {
        int result;
        DivNumbers()
        {
            result = 0;
        }
        public void division(int num1, int num2)
        {
            try
            {
                result = num1 / num2;
            }
            catch (DivideByZeroException e)
            {
                Console.WriteLine("Exception caught: {0}", e);
            }
            finally
            {
                Console.WriteLine("Result: {0}", result);
            }
        }
        static void Main(string[] args)
        {
            DivNumbers d = new DivNumbers();
            d.division(25, 0);
            Console.ReadLine();
        }
    }
}
```

Custom Exception

- Create a class that derives from System.Exception class. As a convention, end the class name with Exception suffix. All .net exceptions end with Exception suffix.
- Provide a public constructor, that takes in a single string parameter. This constructor simply passes the string parameter, to the base exception class constructor.
- Using InnerExceptions, you can also track back the original exception. If you want to provide this capability for your custom exception class, then overload the constructor accordingly.
- If you want your Exception class object to work across application domain, then the object must be serializable. To make your exception class serializable mark it with Serializable attribute and provide a constructor that invokes the base Exception class constructor that takes in SerializationInfo and StreamingContext objects as parameters.
- It is also possible to provide your own custom serialization.

Example of Custom Exception

```
using System;
using System.Runtime.Serialization;

namespace ConsoleApplication.Day_7.CustomException
{
    [Serializable]
    public class UserIsAlreadyLoggedInException : Exception
    {
        public UserIsAlreadyLoggedInException() : base()
        {

        }
        public UserIsAlreadyLoggedInException(string message) : base(message)
        {

        }
        public UserIsAlreadyLoggedInException(string message, Exception
innerException) : base(message, innerException)
        {

        }
        public UserIsAlreadyLoggedInException(SerializationInfo info,
StreamingContext context) : base(info, context)
        {

        }
    }
    public class CustomException
    {
   public static void Main(string[] args)
{
    try
    {
        try
        {
            throw new UserIsAlreadyLoggedInException();
        }
        catch (UserIsAlreadyLoggedInException ex)
        {
            Console.WriteLine(ex.Message);
            throw ex;
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    finally
    {
        Console.WriteLine("Finally");
    }

    Console.ReadLine();
}
    }
}
```

Exception Handling Abuse

Exceptions are unforeseen errors that occur when a program is running. For example, when an application is executing a query, the database connection is lost. Exception handling is used to handle these scenarios.

Using exception handling to implement program logic flow is bad and is termed as exception handling abuse.

```
    public class DivideByZeroAbuse
    {
        public static void DivideByZeroAbuseMethod()
        {
            try
            {
                Console.WriteLine("Enter numerator : ");
                int numerator = Convert.ToInt32(Console.ReadLine());

                Console.WriteLine("Enter denominator : ");
                int denominator = Convert.ToInt32(Console.ReadLine());

                int result = numerator / denominator;
                Console.WriteLine($"Result : {result}");
            }
            catch (DivideByZeroException)
            {
                Console.WriteLine("Denominator should be non zero number");
            }
            catch (FormatException)
            {
                Console.WriteLine("Please enter a valid number");
            }
            catch (OverflowException)
            {
                Console.WriteLine($"Only number between {Int32.MinValue} and
{Int32.MaxValue} should be entered");
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            Console.ReadLine();
        }
    }
```

Exception Handling Abuse Solution

```
public class DivideByZeroAbuseResolved
{
    public static void Main(string[] args)
    {
        try
        {
            Console.WriteLine("Enter numerator : ");
            int numerator;
            bool isNumeratoraNumber = int.TryParse(Console.ReadLine(), out
numerator);
            if (isNumeratoraNumber)
            {
                Console.WriteLine("Enter denominator : ");
                int denominator;
                bool isDenominatorANumber = int.TryParse(Console.ReadLine(), out
denominator);
                if (isDenominatorANumber && denominator != 0)
                {
                    int result = numerator / denominator;
                    Console.WriteLine($"Result : {result}");
                }
                else
                {
                    if (denominator == 0)
                    {
                        Console.WriteLine("Denominator should be non zero number");
                    }
                    else
                    {
                        Console.WriteLine($"Only number between {Int32.MinValue} and
{Int32.MaxValue} should be entered");
                    }
                }
            }
            else
            {
                Console.WriteLine($"Only number between {Int32.MinValue} and
{Int32.MaxValue} should be entered");
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        Console.ReadLine();
    }
}
}
```