1) Write a small program where you need to implement a Try and Catch Block .

```
using System;
namespace assignment_advanced_csharp
{
   class Program
   {
      static void Main(string[] args)
      {
      try
     {
         throw new Exception("This is an exception");
      }
      catch (Exception e)
      {
         Console.WriteLine("An error occurred: " + e.Message);
      }
   }
}
```

```
IndexOutOfRangeException occurred: Index was outside the bounds of the array.
```

2) When should we write multiple catch blocks for a Single Try block?

Multiple catch blocks are used in a single try block when you want to handle different types of exceptions differently. Each catch block can handle a specific type of exception, allowing you to provide specific error handling for each type of exception.

```
using System;
namespace assignment_advanced_csharp
{
   class Program
   {
      static void Main(string[] args)
      {
          try
           {
              int[] numbers = { 1, 2, 3 };
              Console.WriteLine(numbers[3]); // This will throw an IndexOutOfRangeException
           }
           catch (IndexOutOfRangeException e)
```

```
            {
                Console.WriteLine("IndexOutOfRangeException occurred: " + e.Message);
            }
            catch (DivideByZeroException e)
            {
                Console.WriteLine("DivideByZeroException occurred: " + e.Message);
            }
            catch (Exception e)
            {
                Console.WriteLine("Some other exception occurred: " + e.Message);

            }
}
}
```

```
IndexOutOfRangeException occurred: Index was outside the bounds of the array.
```

3) How to define a delegate and call any method or event using it?

A delegate in C# is a type that represents references to methods with a particular parameter list and return type. It is similar to a function pointer in C or C++. Delegates are used to pass methods as arguments to other methods, or to define callback methods. Delegates can be used to define and encapsulate a method with a specific signature, and then call that method through the delegate instance.

GENERAL SYNTAX:delegate return_type DelegateName(parameter_list);

```
using System;
delegate void MyDelegate(string message);
namespace assignment_advanced_csharp
{
    class Program
    {
        static void Main(string[] args)
        {
            del += DisplayMessage; // Adding another method to the delegate
            del("Hello, world!"); // Calling the delegate with a message
            static void PrintMessage(string message)
            {
                Console.WriteLine("Printing message: " + message);
            }
            static void DisplayMessage(string message)
```

```
        {
            Console.WriteLine("Displaying message: " + message);
        }
}
}
}
```

```
Printing message: Hello, world!
Displaying message: Hello, world!
```

4) Try to use Func, Action and Predicate any program.

```
using System;
namespace assignment_advanced_csharp
{
   class Program
   {
      static void Main(string[] args)
      {
         Func<int, int, int> add = (a, b) => a + b;
         Console.WriteLine(add(2, 3)); // Output: 5

         Action<string> greet = name => Console.WriteLine("Hello, " + name);
         greet("Utkarsh"); // Output: Hello, Utkarsh

         Predicate<int> isEven = num => num % 2 == 0;
         Console.WriteLine(isEven(4)); // Output: True
}
}
}
```

```
5
Hello, Utkarsh
True

C:\Users\utkarsh.srivastava\Desktop\assignment_advanced_c#\assignment_advanced_csharp\bin\Debug\netcoreapp3.1\assignment
```

5) What will be the output of below code snipped :

```
static void Main()
{
  Func <string,string> output=delegate(string name)
{
  return "Hello" + name;
};
Console.Write(output("James"));
}
```

a) "HelloJames"

```
static void Main()
{
  Action <int> output = i=>Console.Write(i);
  output(10);
}
```

b) 10

6) Write a program to implement Async await with proper justification.

SIMPLE THREADING PROGRAM:

```
using System;
namespace assignment_advanced_csharp
{
    class Program
    {
        static void Main(string[] args)
        {
             Process1();
             Process2();
            static void Process1()
            {
                Console.WriteLine("Process 1 Started");
                System.Threading.Thread.Sleep(4000); // hold execution for 4 seconds
                Console.WriteLine("Process  1 Completed");
            }
            static void Process2()
            {
                Console.WriteLine("Process 2 Started");
            Console.WriteLine("Process  2 Completed");
            }
        }
    }
}
```

EXPLANATION:

1. MAIN is the entry point of the program process 1 and process 2.
When the program is executed, it will first call Process1 , which will print "Process 1 Started", then wait for 4 seconds, and finally print "Process 1 Completed". After that, it will call Process2, which will print "Process 2 Started" and "Process 2 Completed" one after the other..

THREADING PROGRAM USING ASYNC AND AWAIT:

```
using System;
using System.Threading.Tasks;
namespace assignment_advanced_csharp
{
   class Program
   {
      static async Task Main(string[] args)
      {
         {
            Console.WriteLine("Before calling the asynchronous method.");
            await DoSomethingAsync();
            Console.WriteLine("After calling the asynchronous method.");
         }
          static async Task DoSomethingAsync()
         {
            await Task.Delay(2000); // Simulate an asynchronous operation
            Console.WriteLine("Async method completed.");
         }
      }
   }
}
```

1.**MAIN** is the entry point of the program.
2. Inside the **MAIN** method, "Before calling the asynchronous method," is printed to the console.
3. The **AWAIT** keyword is used to call the **DoSomethingAsync** method. This keyword allows the program to asynchronously wait for the completion of the `**DoSomethingAsync**` method without blocking the thread.
4. The `**DoSomethingAsync**` method is marked as `**async**` and returns a `**Task**`. Inside this method, `**Task.Delay(2000)**` is used to simulate an asynchronous operation by delaying the execution for 2 seconds.
5. After the delay, "**Async method completed.**" is printed to the console.
6. Once the `**DoSomethingAsync**` method completes, "After calling the asynchronous method," is printed to the console in the **Main** method.