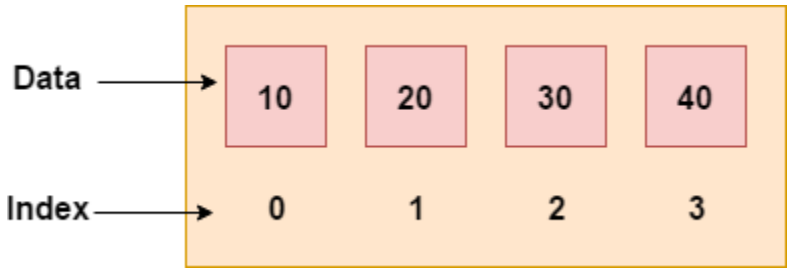
Chapter 3 Arrays and Strings, Properties, Indexers, Delegates & Events

C# Arrays

Like other programming languages, array in C# is a group of similar types of elements that have contiguous memory location. In C#, array is an object of base type System.Array. In C#, array index starts from 0. We can store only fixed set of elements in C# array



Advantages of C# Array

1. Code Optimization (less code)
2. Random Access
3. Easy to traverse data
4. Easy to manipulate data
5. Easy to sort data etc.
6. Fixed size

C# Array Types

There are 3 types of arrays in C# programming:

1. Single Dimensional Array
2. Multidimensional Array
3. Jagged Array

C# Single Dimensional Array

To create single dimensional array, you need to use square brackets [] after the type.

```
int[] arr = new int[5];//creating array
```

You cannot place square brackets after the identifier.

int arr[] = new int[5];//compile time error

Let's see a simple example of C# array, where we are going to declare, initialize and traverse array.

```
using System;

public class ArrayExample

{

    public static void Main(string[] args)

    {

        int[] arr = new int[5]; //creating array

        arr[0] = 10; //initializing array

        arr[2] = 20;

        arr[4] = 30;

                    //traversing array

        for (int i = 0; i < arr.Length; i++)

        {

        }   Console.WriteLine(arr[i]);

                }
```

Output:

10

0

20

0

30

C# Array Example: Declaration and Initialization at same time There are 3 ways to initialize array at the time of declaration. int[] arr = new int[5]{ 10, 20, 30, 40, 50 };

We can omit the size of array.

int[] arr = new int[]{ 10, 20, 30, 40, 50 };

We can omit the **new** operator also.

int[] arr = { 10, 20, 30, 40, 50 };

Let's see the example of array where we are declaring and initializing array at the same time.

```
using System;

public class ArrayExample

{

    public static void Main(string[] args)

    {

        int[] arr = { 10, 20, 30, 40, 50 };//Declaration and Initialization of
        array

            //traversing array

        for (int i = 0; i < arr.Length; i++)
        }          }
        {
                    }
                Console.WriteLine(arr[i]);
```

Output:

10

20

30

40

50

C# Array Example: Traversal using foreach loop

We can also traverse the array elements using foreach loop. It returns array element one by one.

```
using System;

public class ArrayExample

{
        public static void Main(string[] args)

        {

                int[] arr = { 10, 20, 30, 40, 50 };//creating and initializing array

                      //traversing array

                foreach (int i in arr)

                {

                        Console.WriteLine(i);

                }           }    }
```

Output:

10

20

30

40

50

Two Dimensional Array:-

In C#, a two-dimensional array (or 2D array) stores data in a grid-like format of rows and columns. It is also known as a rectangular array because all rows have the same number of columns.

Declaration and Initialization

You declare a 2D array by using a comma inside the square brackets [,].

```
int[,] matrix = { { 1, 2, 3 }, { 4, 5, 6 } }; // A 2x3 array

// Alternatively, specifying size explicitly
int[,] matrixWithSize = new int[2, 3] { { 1, 2, 3 }, { 4, 5, 6 } };
```

There are 3 ways to initialize a multidimensional array in C# while declaration:

1) int[,] arr = new int[3,3]= { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
2) //We can omit the array size.
   int[,] arr = new int[,]{ { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
3) //We can omit the new operator also.
4) int[,] arr = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };

t siz

Access & Change the Elements of an Array

```
// Getting values from the array
int valueAtRow1Column2 = arr[0, 1];
// Accessing the element at row 0, column 1 (indexing is zero-based)
Console.WriteLine("Value at row 0, column 1: " + valueAtRow1Column2); // Output: 2

// Setting values in the array
arr[1, 2] = 10; // Setting the value at row 1, column 2 to 10
```

printing array

```
for(int i=0;i<3;i++)
{
    for(int j=0;j<3;j++)
    {
        Console.Write(arr[i,j]+" ");
    }
    Console.WriteLine();//new line at each row
```

C# Jagged Arrays

In C#, jagged array is also known as "array of arrays" because its elements are arrays. The elemen of jagged array can be different.

Declaration of Jagged array

Let's see an example to declare jagged array that has two elements.

```
int[][] arr = new int[2][];
```

Initialization of Jagged array

Let's see an example to initialize jagged array. The size of elements can be different.

```
arr[0] = new int[4];


  arr[1] = new int[6];
```

Initialization and filling elements in Jagged array

Let's see an example to initialize and fill elements in jagged array.

```
arr[0] = new int[4] { 11, 21, 56, 78 };
arr[1] = new int[6] { 42, 61, 37, 41, 59, 63 };
```

Here, size of elements in jagged array is optional. So, you can write above code as given below:

```
arr[0] = new int[] { 11, 21, 56, 78 };

arr[1] = new int[] { 42, 61, 37, 41, 59, 63 };
```

C# Jagged Array Example

Let's see a simple example of jagged array in C# which declares, initializes and traverse jagged arrays.

```
public class JaggedArrayTest
{
   public static void Main()
   {
      int[][] arr = new int[2][];// Declare the array

      arr[0] = new int[] { 11, 21, 56, 78 };// Initialize the array
      arr[1] = new int[] { 42, 61, 37, 41, 59, 63 };

      // Traverse array elements
      for (int i = 0; i < arr.Length; i++)
      {
         for (int j = 0; j < arr[i].Length; j++)
         {
            System.Console.Write(arr[i][j]+" ");
         }
         System.Console.WriteLine();
      }
   }
}
```

Output:

```
11 21 56 78
42 61 37 41 59 63
```

Initialization of Jagged array upon Declaration

Let's see an example to initialize the jagged array while declaration.

```
int[][] arr = new int[3][]{
        new int[] { 11, 21, 56, 78 },
        new int[] { 2, 5, 6, 7, 98, 5 },
        new int[] { 2, 5 }
        };
```

C# Jagged Array Example 2

Let's see a simple example of jagged array which initializes the jagged arrays upon declaration.

```
public class JaggedArrayTest
{
    public static void Main()
    {
        int[][] arr = new int[3][]{
        new int[] { 11, 21, 56, 78 },
        new int[] { 2, 5, 6, 7, 98, 5 },
        new int[] { 2, 5 }
        };

        // Traverse array elements
        for (int i = 0; i < arr.Length; i++)
        {
            for (int j = 0; j < arr[i].Length; j++)
            {
                System.Console.Write(arr[i][j]+" ");
            }
            System.Console.WriteLine();
        }
    }
}
```

Output:

```
11 21 56 78
2 5 6 7 98 5
2 5
```

ArrayList:-

ArrayList represents an ordered collection of an object that can be indexed individually. It is basically an alternative to an array. It also allows dynamic memory allocation, adding, searching and sorting items inthe list.

Properties of ArrayList Class:

Elements can be added or removed from the Array List collection at any point in time.

The **ArrayList** is not guaranteed to be sorted.

The capacity of an **ArrayList** is the number of elements the ArrayList can hold.

Elements in this collection can be accessed using an integer index. Indexes in this collection are zero-based.

It also allows duplicate elements.

Using multidimensional arrays as elements in an ArrayList collection is not supported.

Constructors

Constructor	Description
ArrayList()	Initializes a new instance of the ArrayList class that is empty and has the default initial capacity.
ArrayList(ICollection)	Initializes a new instance of the ArrayList class that contains elements copied from the specified collection and that has the same initial capacity as the number of elements copied.
ArrayList(Int32)	Initializes a new instance of the ArrayList class that is empty and has the specified initial capacity.

```
// C# code to create an ArrayList

using System;

using System.Collections;

using System.Collections.Generic;

class BCATY {

        // Driver code

        {       // Creating an ArrayList
        public static void Main()
                ArrayList myList = new ArrayList();
```

```
            // Adding elements to ArrayList

            myList.Add("Hello");

            myList.Add("World");

            Console.WriteLine("Count : " + myList.Count); Console.WriteLine("Capacity

            : " + myList.Capacity);

    }}
```

Properties

Property	Description
Capacity	Gets or sets the number of elements that the ArrayList can contain.
Count	Gets the number of elements actually contained in the ArrayList.
IsFixedSize	Gets a value indicating whether the ArrayList has a fixed size.
IsReadOnly	Gets a value indicating whether the ArrayList is read-only.
IsSynchronized	Gets a value indicating whether access to the ArrayList is synchronized (thread safe).
Item[Int32]	Gets or sets the element at the specified index.

Example:

```
// C# program to illustrate the

// ArrayList Class Properties

using System;

using System.Collections;

using System.Collections.Generic;

    class BCATY {

        // Driver code

    public static void Main()

    {    ArrayList myList = new ArrayList();

          // Adding elements to ArrayList

        myList.Add("A");

        myList.Add("B");
        myList.Add("C");
        myList.Add("D");
        myList.Add("E");
```

```
            myList.Add("F");


            // To check if the ArrayList has fixed size or not

            Console.WriteLine(myList.IsFixedSize);

            // To check if the ArrayList is read-only or not

            Console.WriteLine(myList.IsReadOnly);

      }     }
```

Output
:False
False

Methods

Method	Description

Add(Object)	Adds an object to the end of the ArrayList.
	ArrayList list = new ArrayList();
	list.Add(10);
	list.Add(20);
	list.Add(30);

AddRange(ICollection)	Adds the elements of an ICollection to the end of the ArrayList. List.AddRange(33,22,55)
Clear()	Removes all elements from the ArrayList. List.Clear();
Contains(Object)	Determines whether an element is in the ArrayList.

CopyTo(Array)	Copies the entire ArrayList to a compatible one-dimensional Array, starting at the beginning of the target array.
	int[] arr = new int[3];
	list.CopyTo(arr);

CopyTo(Array, Int32)	Copies the entire ArrayList to a compatible one-dimensional Array, starting at the specified index of the target array. list.CopyTo(arr,2);
Equals(Object)	Determines whether the specified object is equal to the current object. list1.Equals(list2)
Insert(Int32, Object)	Inserts an element into the ArrayList at the specified index. list.Insert(1, 15);

Remove(Object)	Removes the first occurrence of a specific object from the ArrayList. list.Remove(20);
RemoveAt(Int32)	Removes the element at the specified index of the ArrayList. list.RemoveAt(2);
Reverse()	Reverses the order of the elements in the entire ArrayList. list.Reverse();
Sort()	Sorts the elements in the entire ArrayList. List.Sort();

Example 1:

```
// C# code to check if an element is contained in ArrayList or not

using System;

using System.Collections;

using System.Collections.Generic;

class BCATY {

        // Driver code

    public static void Main()

    {

            // Creating an ArrayList

        ArrayList myList = new ArrayList();
            // Adding elements to ArrayList

        myList.Add("A");

        myList.Add("B");

        myList.Add("C");

        myList.Add("D");

        myList.Add("E");

        myList.Add("F");

        myList.Add("G");

        myList.Add("H");


        if (myList.Contains("E"))

            Console.WriteLine("Yes, exists at index " + myList.IndexOf("E"));

        else

            Console.WriteLine("No, doesn't exists");

    } }
```

Output:

Yes, exists at index 4

Example 2:

```
// C# code to remove a range of elements from the ArrayList
using System;
using System.Collections;
using System.Collections.Generic;
   class BCATY {
      public static void Main()
      {     // Creating an ArrayList
            ArrayList myList = new ArrayList(10);
               // Adding elements to ArrayList
            myList.Add(2);
            myList.Add(4);
            myList.Add(6);
            myList.Add(8);
            myList.Add(10)
            ;
            myList.Add(12)
            ;
            myList.Add(14)
            ;
            myList.Add(16)
            ;
            myList.Add(18)
            ;
            myList.Add(20)
            ;
               // Displaying the elements in ArrayList
            Console.WriteLine("The initial ArrayList: ");
            foreach(int i in myList)
```

```
{
        Console.WriteLine(i);

}

// removing 4 elements starting from index 0

myList.RemoveRange(0, 4);
```

```
// Displaying the modified ArrayList
Console.WriteLine("The ArrayList after Removing
elements: ");
// Displaying the elements in ArrayList
foreach(int i in myList)
{
Console.WriteLine(i);

}
```

Output:

The initial ArrayList:

2

4

6

8

10

12

14

16

18

20

The ArrayList after Removing elements:

10

12

14

16

18

20

STRING Class

String manipulation is the most common part of many C# programs. Strings represent a sequence of characters. The easiest way to represent a sequence of characters in C# is by using a character array.

For example: char [ ] charArray = new char [4];

charArray [0) = 'N;

charArray [I) = 'a';
charArray [2] = 'm';

charArray [3) = 'e';

Although character arrays have the advantage of being able to query their length, by themselves they are not good enough to support the range of operations we may like to perform on strings. For example, copying one character array into another might require a lot of book keeping effort. Fortunately, C# is equipped with features that could handle these situations more efficiently.

C# supports two types of strings, namely, immutable strings and mutable strings .C# also supports a feature known as regular expressions that can be used for complex string manipulations and pattern matching.

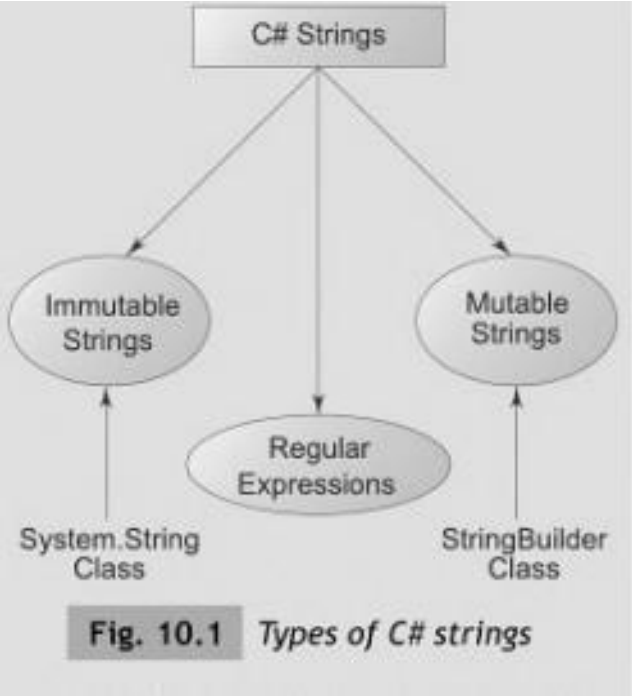


Fig. 10.1 *Types of C# strings*

CREATING STRINGS

C# supports a predefined reference type known as string. We can use string to declare string type objects. Remember, when we declare a string using string type, we are in fact declaring the object to be of type **System.String**, one of the built-in types provided by the .NET Framework.

A string type is therefore a System.String type as well. We can create immutable strings using string or String objects in a number of ways.

• Assigning string literals

• Copying from one object to another

• Concatenating two objects

• Reading from the keyboard

• Using ToString method

Assigning String Literals

The most common way to create a string is to assign a quoted string of characters known as string literal to a string object.

For example: string s1; ---//declaring a string object

si ="abc";-- //assigning string literal Both these statements may be combined into one as follows: string s1 = "abc"; //declaring and assigning

Copying Strings We can also create new copies of existing strings. This can be accomplished in two ways:

• Using the overloaded = operator

• Using the static Copy method

Example: string s2 = sl; //assigning string s2

• string.Copyis1); //copying Both these statements would accomplish the same thing, namely, copying the contents of sl into s2.

Concatenating Strings

We may also create new strings by concatenating existing strings.

There are a couple of ways to accomplish this.

• Using the overloaded + operator

• Using the static Concat method

Examples: string s3 = s1 + s2; //s1 and s2 exist already

string s3 = string.Concat(s1, s2)

If s I = 'abc' and s2 = 'xyz', then both the statements will store the string 'abcxyz' in s3.

Reading from the Keyboard It is possible to read a string value interactively from the keyboard and assign it to a string object.

string s = Console.ReadLine( ); On reaching this statement, the computer will wait for a string of characters to be entered from the keyboard. When the 'return key' is pressed, the string will be read and assigned to the string object s.

The ToString Method Another way of creating a string is to call the ToString method on an object and assign the result to a string variable.

```
int number =123;

string numStr = number.ToString( );
```

METHODS

String objects are immutable, meaning that we cannot modify the characters co However, since the string is an alias for the predefined System.String class Language Runtime (CLR), there are many built-in operations available that work operations produce a modified version of the string rather than modifying the string on is called.

METHOD OPERATION

string s1,s2

Compare ( )	Compares two strings `string.Compare(s1,s2)`
CompareTo ( )	Compares the current instance with another instance **Ex s1.CompareTo(s2)**
ConCat ( )	Concatenates two or more strings `string.Concat(s1,s2)`
Copy()	Creates a new string by copying another `string s2 = string.Copy(s1);`

CopyTo ( )	Copies a specified number of characters to an any of Unicode characters
	`string s1 = "Hello C#, How Are You?";`
	`char[] ch = new char[15];`
	`string s1 = "Hello C#, How Are You?"; char[] ch = new char[15]; s1.CopyTo(10,ch,0,12);`

EndsWith ( )	Determines whether a substring exists at the end of the string **s1.EndsWith("abc");**
Equals( )	Determines if two strings are equal **s1.Equals(s2)**
IndexOf ( )	Returns the position of the first occurrence of a substring **s1.IndexOf('e')**
Insert ( )	Returns a new string with a substring in second at a specified location **s1.Insert(5,"-")**
Join ( )	Joins an any of strings together `string s3 = string.Join("-",s1);`

LastIndex0f( )	Returns the position of the last occurrence of a
substring PadLett ( )	Left-aligns the strings in a field
PadRight ( )	Right-aligns the string in a field
Remove ( )	Deletes characters from the string `string s2 = s1.Remove(2);`
Replace	Replaces all instances of a character with a new character `string s2 = s1.Replace('F','C');`
StartsWith ( )	Determines whether a substring exists at the beginning of the string **s1.StatsWith("abc");**
ToLower ( )	Returns a lower-case version of the string
ToUpper ( )	Returns an upper-case version of the string

Property in C#

Property in C# is a class member that exposes the class' private fields.Internally, C# properties are special methods called accessors. A C# property has two accessors, a get property accessor or a **getter** and a set property accessor or a **setter**. A **get** accessor returns a property value,and a **set** accessor assigns a new value. The **value** keyword represents the value of a property. Properties can be read-write properties, read-only properties, or write-only properties.The read-write property implements both a get and a set accessor.

A write-only property implements a set accessor but no get accessor.

A read-only property implements a get accessor but no set accessor.

Why use Property

The members of a class are private then how another class in C# will be able to read, write, or compute the value of that field.

If the members of the class are public then another class may misuse that member.

```
class MyClass
{ private int x;
   public int X
   {
set
{ x = value;
   }
```

```
get
{ return x;
  }}}
class MyClient
{ public static void Main()
  {

MyClass m = new MyClass();
m.X = 10;
int xVal = m.X;
  Console.WriteLine(xVal);
  }}
```

```
using System;
  public class C1
  {
private int rn;
  private string
  name; public
  string Name
{ get
  {
return name;
  }
set
  {
name = value;
  }}
public int RN
  {
get
  {
return rn;
```

```
set
    {
rn= value;
     }}        }
public class C2
    {
    // Main Method
public static void Main(string[] args)
    {
classC1 obj = new C1();
    obj.RN = 10000;
obj.Name = null;
Console.WriteLine("Name: \nRoll No: ", obj.name, obj.rn);
    }
    }
```

Indexer:-

An indexer allows an instance of a class or struct to be indexed as an array. If the user will define an indexer for a class, then the class will behave like a virtual array. Array access operator i.e ([ ]) is used to access the instance of the class which uses an indexer. A user can retrieve or set the indexed value without pointing an instance or a type member. Indexers are almost similar to the **Properties**. *The main difference between Indexers and Properties* is that the accessors of the Indexers will take parameters.

Syntax:

```
[access_modifier] [return_type] this [argument_list]
{
 get
 {// get block code
 }
 set
 {// set block code      } }
```

In the above syntax:

access_modifier: It can be public, private, protected or internal.

return_type: It can be any valid C# type.

this: It is the keyword which points to the object of the current class.

argument_list: This specifies the parameter list of the indexer.

get{ } and set { }: These are the accessors.

Example:

```
class IndexerCreation
{
   private string[] val = new string[3];
    public string this[int index]
   {
      get
      {
          return val[index];
      }
      set
      {
         val[index] = value;
      }
   }
}

class main {

   public static void Main() {
      IndexerCreation ic = new IndexerCreation();

      ic[0] = "C";
      ic[1] = "CPP";
      ic[2] = "CSHARP";

      Console.Write("Printing values stored in objects used as arrays\n");

      Console.WriteLine("First value = {0}", ic[0]);
      Console.WriteLine("Second value = {0}", ic[1]);
      Console.WriteLine("Third value = {0}", ic[2]);
   }}
```

Important Points About Indexers:

- Indexers can be overloaded.
- These are different from Properties.
- This enables the object to be indexed in a similar way to arrays.
- A set accessor will always assign the value while the get accessor will return the value.“this” keyword is always used to declare an indexer.
- To define the value being assigned by the set indexer, ” value” keyword is used.Indexers are also known as the Smart Arrays or Parameterized Property in C#.
- Indexer can’t be a static member as it is an instance member of the class

C# Delegates

In C#, delegate is a *reference to the method*. It works like *function pointer* in C and C++. But it is objected-oriented, secured and type-safe than function pointer.

For static method, delegate encapsulates method only. But for instance method, it encapsulates method and instance both.

The best use of delegate is to use as event.

Internally a delegate declaration defines a class which is the derived class of **System.Delegate**.

Declaring Delegates

Delegate declaration determines the methods that can be referenced by the delegate. A delegate can refer to a method, which has the same signature as that of the delegate.

For example, consider a delegate −

```
delegate int Calculator(int n);//declaring delegate
```

The preceding delegate can be used to reference any method that has a single *string* parameter and returns an *int* type variable.

Syntax for delegate declaration is –

delegate <return type> <delegate-name> <parameter list>

Instantiating Delegates

Once a delegate type is declared, a delegate object must be created with the **new** keyword and be associated with a particular method. When creating a delegate, the argument passed to the **new** expression is written similar to a method call, but without the arguments to the method. For example –

```
Calculator c1 = new Calculator(add);//instantiating delegate
```

Calling Methods:-

```
c1(20);//calling method using delegate
```

```
c1.Invoke(20);
```

C# Delegate Example

Let's see a simple example of delegate in C# which calls add() and mul() methods.

```
using System;
delegate int Calculator(int n);//declaring delegate

public class DelegateExample
{
   static int number = 100;
   public static int add(int n)
   {
      number = number + n;
      return number;
   }
   public static int mul(int n)
   {
      number = number * n;
      return number;
   }
   public static int getNumber()
   {
      return number;
   }
   public static void Main(string[] args)
   {
      Calculator c1 = new Calculator(add);//instantiating delegate
```

```
        Calculator c2 = new Calculator(mul); c1(20);//calling
        method using delegate
        Console.WriteLine("After c1 delegate, Number is: " + getNumber()); c2(3);
        Console.WriteLine("After c2 delegate, Number is: " + getNumber());
    }
}
```

Output:

```
After c1 delegate, Number is: 120
After c2 delegate, Number is: 360
```

Multicasting of a Delegate

Delegate objects can be composed using the "+" operator. A composed delegate calls the two delegates it was composed from. Only delegates of the same type can be composed. The "-" operator can be used to remove a component delegate from a composed delegate.

Using this property of delegates you can create an invocation list of methods that will be called when a delegate is invoked. This is called **multicasting** of a delegate. The following program demonstrates multicasting of a delegate –

```
using System;
delegate int Calculator(int n);//declaring delegate

public class DelegateExample
{
    static int number = 100;
    public static int add(int n)
    {
        number = number + n;
        return number;
    }
    public static int mul(int n)
    {
        number = number * n;
        return number;
    }
    public static int getNumber()
    {
        return number;
    }
```

```
    public static void Main(string[] args)
    {
        Calculator c1 = new Calculator(add);//instantiating delegate C1+=
        new Calculator(mul);
        c1(20);//calling method using delegate
        Console.WriteLine("After c1 delegate, Number is: " + getNumber());

    }
}
```

Custom Events:-

Events are user actions such as key press, clicks, mouse movements, etc., or some occurrence such as system generated notifications. Applications need to respond to events when they occur. For example, interrupts. Events are used for inter-process communication.

Using Delegates with Events

The events are declared and raised in a class and associated with the event handlers using delegates within the same class or some other class. The class containing the event is used to publish the event. This is called the **publisher** class. Some other class that accepts this event is called the **subscriber** class. Events use the **publisher-subscriber** model.

A **publisher** is an object that contains the definition of the event and the delegate. The event-delegate association is also defined in this object. A publisher class object invokes the event and it is notified to other objects.

A **subscriber** is an object that accepts the event and provides an event handler. The delegate in the publisher class invokes the method (event handler) of the subscriber class.

Declaring Events

To declare an event inside a class, first of all, you must declare a delegate type for the even as:

public delegate string mydel(string str);

then, declare the event using the **event** keyword –

event mydel myevent;

The preceding code defines a delegate named *mydel* and an event named *myevent*, which invokes the delegate when it is raised.

Example

```
using System;
namespace SampleApp
 {
   public delegate string MyDel(string str);

   class EventProgram { event MyDel
      MyEvent;

      public EventProgram()
      {
        this.MyEvent += new MyDel(this.WelcomeUser);
      }
      public string WelcomeUser(string username)
      {return "Welcome ! " + username;
      }
      static void Main(string[] args)
      {
      EventProgram obj1 = new EventProgram();
      string result = obj1.MyEvent("Event fired");
        Console.WriteLine(result);
      }
   }
}
```

When the above code is compiled and executed, it produces the following result –

Welcome ! Event fired.