Chapter 3

Delegates , lambdas and events

Mrs. Swati Satpute

Delegates

- A delegate is a type that represents references to methods with a particular parameter list and return type.
- Delegate instance can **store references to any method** with a **compatible signature** and **return type.**
- Referenced **methods can be invoked through the delegate** instance.
- **Delegates** are **used to pass methods as arguments** to other methods.

Delegates

- A delegate in C# is **similar to a function pointer** in C or C++.

- Unlike function pointers in C or C++, **delegates are object-oriented**, **type-safe**, and **secure**.

- This **ability to refer** to **a method as a parameter makes delegates ideal for defining callback methods.**

- Delegates are **ideally suited for use as events** — notifications from one component to "listeners" about changes in that component.

Delegates

- **A delegate can reference** both **static and instance methods.**

- The members of a delegate are the members inherited from class **System.Delegate.**

Delegates

Examples:

```
public delegate void SimpleDelegate ()
```

This declaration defines a delegate named SimpleDelegate, which will **encapsulate any method that takes no parameters and returns no value.**

```
public delegate int ButtonClickHandler (object obj1, object obj2)
```

This declaration defines a delegate named ButtonClickHandler, which will **encapsulate any method that takes two objects as parameters and returns an int.**

Delegates

- There are three steps in defining and using delegates:

- Declaration
- Instantiation
- Invocation

Delegates – Static method

```
using System;
namespace BasicDelegate
{
    // Declaration
    public delegate void SimpleDelegate();

    class TestDelegate
    {
         public static void MyFunc()
         {
           Console.WriteLine("I was called by delegate ...");
    }

public static void Main()
{
    // Instantiation
    SimpleDelegate dlgObj = new SimpleDelegate(MyFunc);

    // Invocation
      dlgObj ();
}
```

Output:

```
I was called by delegate ...
```

Delegates

```
public delegate int dlg_calculation(int a, int b);
class mainclass
{
dlg_calculation calc_delegate;

public int add(int num1,int num2)
{
    return num1 + num2;
}

static void Main()
{
    int result;
    mainclass obj = new mainclass();
    obj.calc_delegate = new dlg_calculation(obj.add);
    result = obj.calc_delegate(50,70);
} //end of Main method
} //end of mainclass
```

Simple Delegate Application

Bubble Sorter Example – WO delegate

```
static class SimpleSort1
   {
       public static void BubbleSort (int[] items)
       {
           int I, j, temp;

           if (items == null)
           {
               return;
           }

           for (i = items.Length - 1; i >= 0; i--)
           {
               for (j = 1; j <= i; j++)
               {
                   if (items[j - 1] > items[j])
                   {
                       temp = items[j - 1];
                       items[j - 1] = items[j];
                       items[j] = temp;
                   }
```

Bubble Sorter with Delegates

```
delegate bool dlg_CompareObj (object lhs, object rhs);

public class BubbleSortClass
    {

        static public void Sort(object[] sortArray, dlg_CompareObj gt_Method)
        {
                for (int i = 0; i < sortArray.Length; i++)
                {
                        for (int j = 0; j < sortArray.Length; j++)
                        {
                                if(gt_Method(sortArray[j],sortArray[i]))
                                {
                                        object temp= sortArray[i];
                                        sortArray[i] = sortArray[j];
                                        sortArray[j]=temp;
                                }
                        }
                }
        }
    }
```

Students Class

```
class Student
{
            private string name;
            private int rollno;
            private int marks;


  // user defined function which is comparing two object and returning bool value

  public static bool RhsIsGreater(object lhs, object rhs)
 {
            Student stdLhs = (Student)lhs;
            Student stdRhs = (Student)rhs;
            return (stdRhs.marks > stdLhs.marks);
  }
}
```

```
public class Program
{
public delegate bool dlg_CompareObj (object lhs, object rhs);

static void Main(string[] args)
{
// Creating array of Student objects.
Student[] students = { new Student("Mark", 1, 799),  new Student ("David“, 2,545),
new Student ("Lavish", 3,999),  new Student ("Voora", 4,228)};

// Creating Delegate passing static method of Student class as argument
dlg_CompareObj StudentCompareOp = new dlg_CompareObj (Student.RhsIsGreater);

// Now calling static method of BubbleSortClass, passing Student object array and
delegate as argument

BubbleSortClass.Sort(students, StudentCompareOp);

}
```

Multicast Delegate

- A delegate containing more than 1 methods called as multicast delegate.

- For accepting multiple references, delegate signature's **return type must be void**

DoubleOp operations = new DoubleOp (MathsOp.MultiplyByTwo);

operations += new DoubleOp (MathsOp.Square)

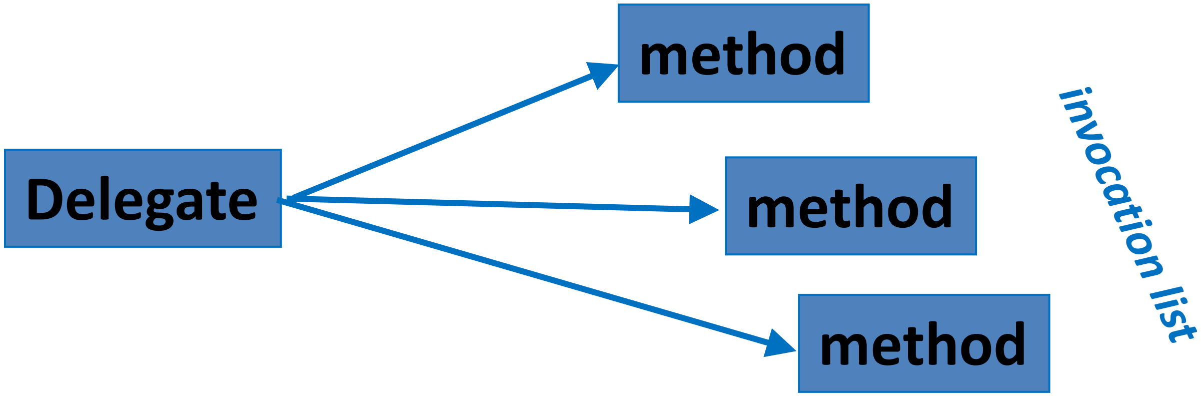
- += , -= operators are supported in multicast delegate
- **Order** in which methods will get executed **is** formally **undefined**

Delegates and Events

- A delegate in C# contains one or more method references: its **invocation list**.

- Invoking the delegate invokes the methods in its invocation list.

- Delegates are primarily used for event handling.



Basic delegate pattern in C#

```
delegate void MyDelegate(int x);
```

declare a delegate type

```
class Eater {
   static void EatAnInt(int x) {…}
   void ConsumeAnInt(int x) {…}
```

some methods for the invocation list

```
   static void Main() {
      Eater etr = new Eater();
      MyDelegate d =
            new MyDelegate(EatAnInt) +
            new MyDelegate(etr.ConsumeAnInt);
```

create a delegate

```
      d(5);
   }
}
```

invoke it; both EatAnInt(x) *and* etr.ConsumeAnInt(x) *will be invoked*

Delegates Summary

- Delegates are similar to C++ function pointers, but delegates are **fully object-oriented**
- Delegates encapsulate both an object instance and a method.
- Delegates allow methods to be passed as parameters.
- Delegates can be used to define callback methods.
- Delegates can be chained together; for example, multiple methods can be called on a single event.

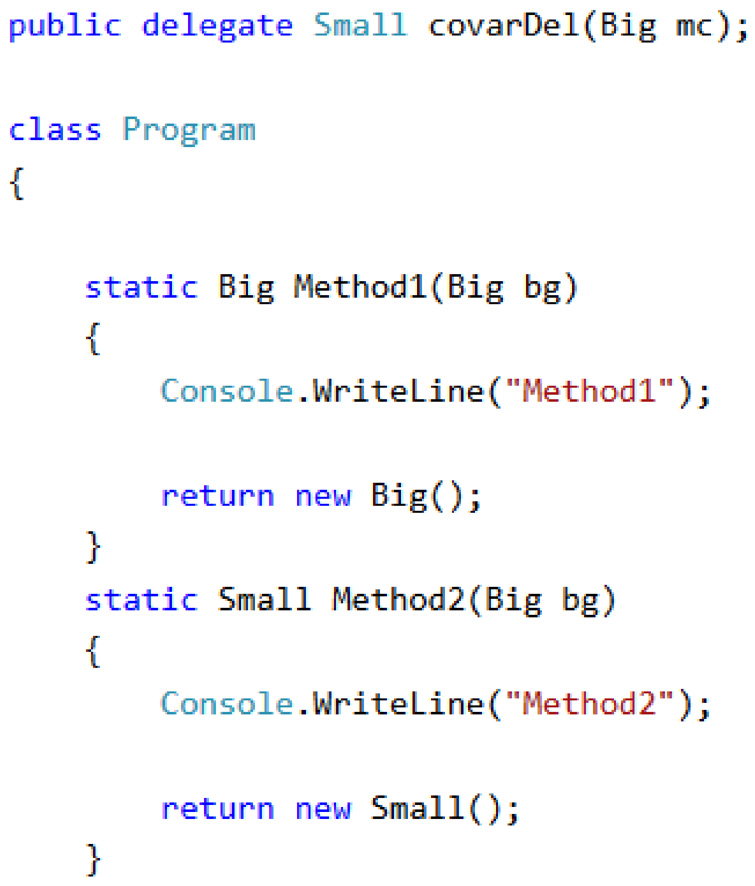
Variance in Delegates

- This means that one can assign to delegates not only methods that have matching signatures, but also
  - methods that **return *more* derived types** (**covariance**) or
  - that **accept parameters** that **have less derived types** (**contravariance**) than that specified by the delegate type.
- This includes both **generic and non-generic delegates.**

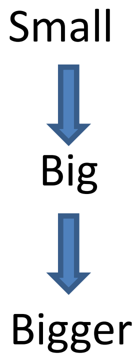
Covariance

- **Covariance** enables **to pass (return) a derived type where a base type is expected.**
- Covariance can be applied on delegate, generic, array, interface.
- Using covariance, one can assign any methods which return derived type instead of base type (as per delegates signature).

Covariance



```
public delegate Small covarDel(Big mc);

class Program
{

    static Big Method1(Big bg)
    {
        Console.WriteLine("Method1");

        return new Big();
    }
    static Small Method2(Big bg)
    {
        Console.WriteLine("Method2");

        return new Small();
    }
```



Simple Delegate

covarDel del= **Method2**;
Small sm1 = del(new Big());

Covaiance Delegate

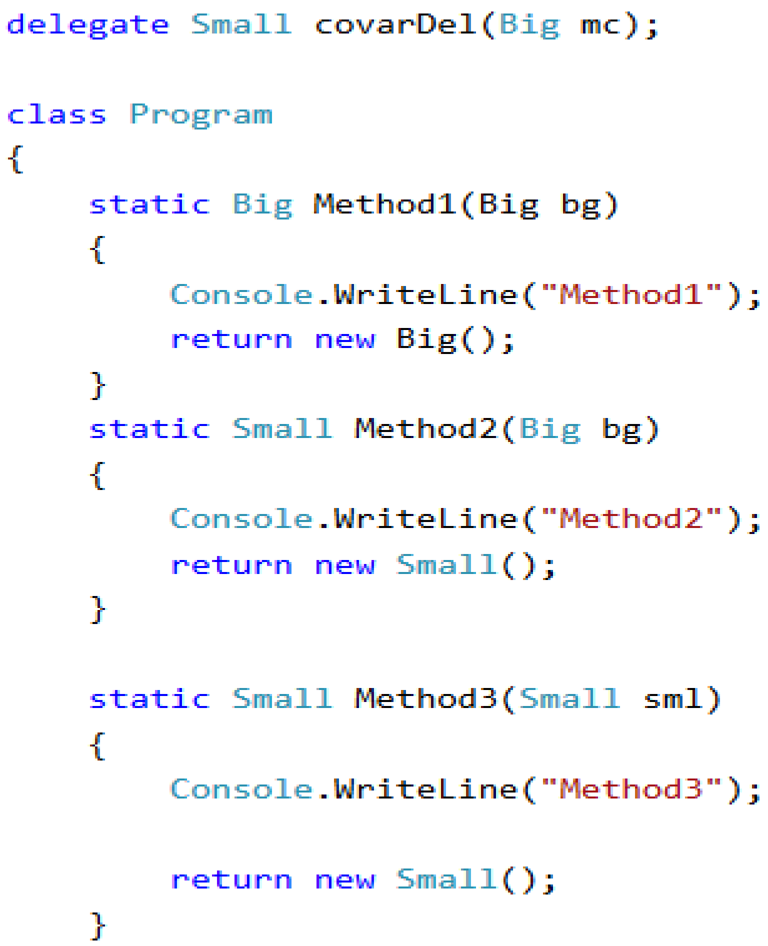
covarDel del = **Method1;**
Small sm1 = del(new Big());

Covariance focuses on return type i.e. allows derived type value to be returned

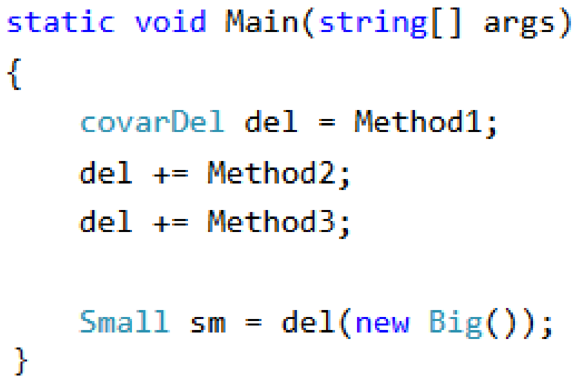
Contravariance

- Contravariance is applied to **parameters**.
- Contravariance allows a method with the **parameter of a base class to be assigned** to a delegate that **expects** the **parameter of a derived class.**

Contravariance



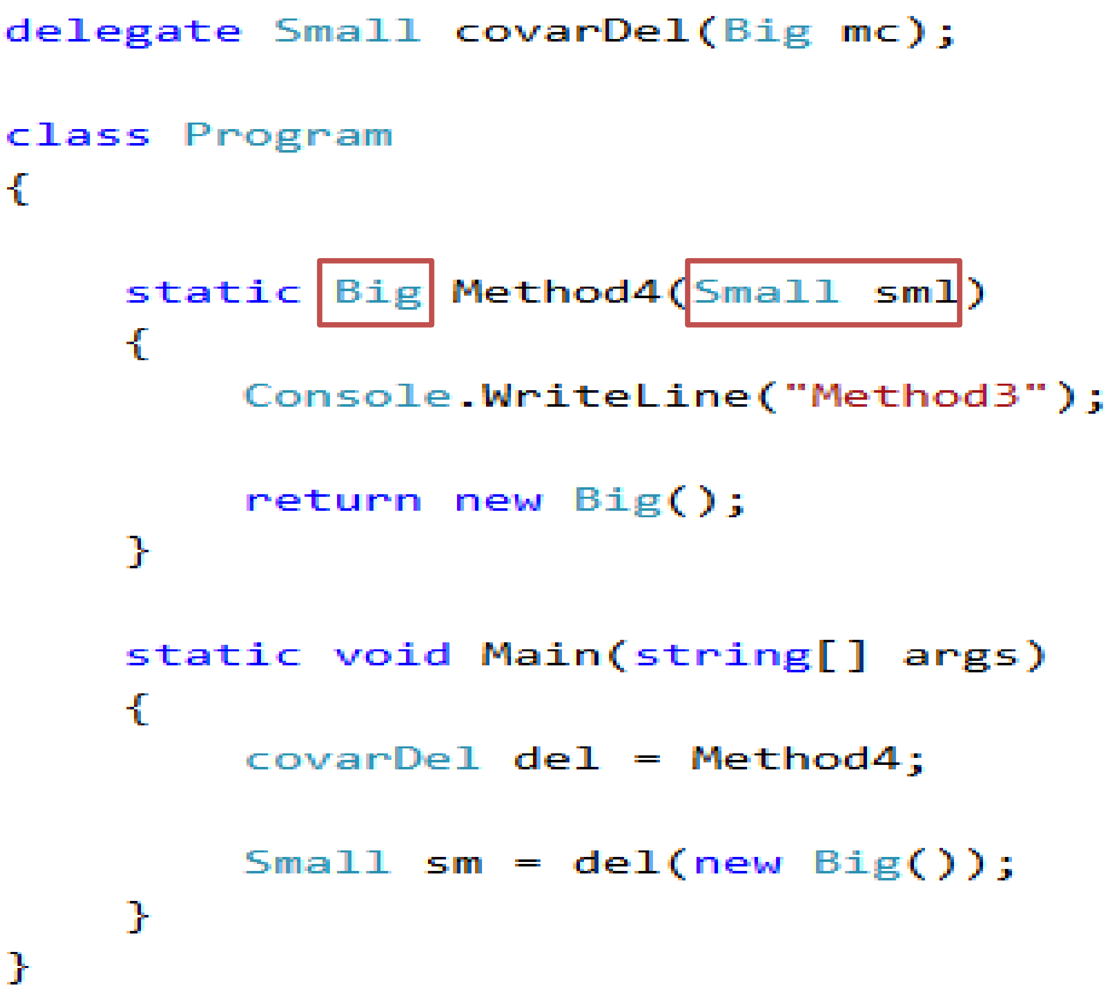
```
delegate Small covarDel(Big mc);

class Program
{
    static Big Method1(Big bg)
    {
        Console.WriteLine("Method1");
        return new Big();
    }
    static Small Method2(Big bg)
    {
        Console.WriteLine("Method2");
        return new Small();
    }

    static Small Method3(Small sml)
    {
        Console.WriteLine("Method3");

        return new Small();
    }
```



```
static void Main(string[] args)
{
    covarDel del = Method1;
    del += Method2;
    del += Method3;

    Small sm = del(new Big());
}
```

Method3 is expecting base class object where delegate expect derived class object

Covariance and Contravariance



```
delegate Small covarDel(Big mc);

class Program
{

    static Big Method4(Small sml)
    {
        Console.WriteLine("Method3");

        return new Big();
    }

    static void Main(string[] args)
    {
        covarDel del = Method4;

        Small sm = del(new Big());
    }
}
```

Covariance and ContraVariance

```
public class First { }
public class Second : First { }
public delegate First SampleDelegate(Second a);
public delegate R SampleGenericDelegate<A, R>(A a);


 // Matching signature.

 public static First ASecondRFirst(Second second)
 { return new First(); }

 // The return type is more derived.
 public static Second ASecondRSecond(Second second)
 { return new Second(); }
```

Demo

Summary:

covariant convert from a wider type to narrower type. (example, from double to float.)

Contravariant types convert from a narrower type to a wider type. (example, from short to int)

- Simple Delegate
- Multicast Delegate
- Covariance and Contra Variance Delegate

C# Event Processing

- Generally speaking, two logical components are required to implement the event processing model:
  - – 1) An event producer (or publisher)
  - – 2) An event consumer (or subscriber)

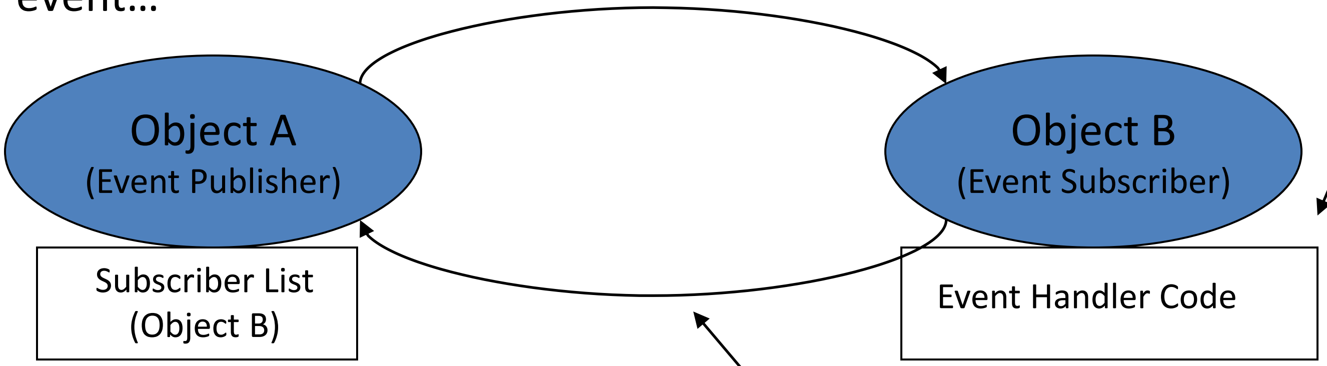
- Each logical components has assigned responsibilities

- Consider the following diagram

C# Event Processing

When an **Event** occurs **notification** is **sent** to all the subscribers on the list for that particular event…

Object B **processes** the event **notification** in its event handler code



Object A
(Event Publisher)

Subscriber List
(Object B)

Object B
(Event Subscriber)

Event Handler Code

Object A **maintains a list of subscribers** for each publishable event

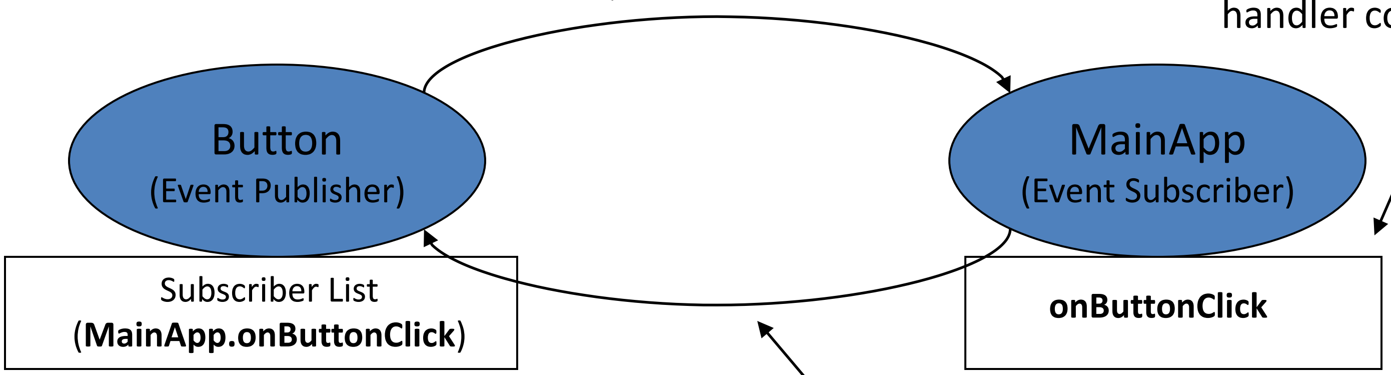
Object B subscribes to event
(or events) generated by Object A.

C# Event Processing

Let’s map Object A and Object B to some familiar object types…

When a Click event occurs notification is sent to all the subscribers on the list…

MainApp processes the Click notification in its onButtonClick event handler code



Button

(Event Publisher)

Subscriber List

(**MainApp.onButtonClick**)

MainApp

(Event Subscriber)

onButtonClick

Button maintains a list of subscribers of its Click event

MainApp subscribes to Button’s Click event

C# Event Processing

- The .Net API contains lots of classes that generate different types of events…
  - Most are GUI related
    - Can you think of a few?

- It also contains lots of Delegates
  - Can you name at least one?

Role of Each Component - **Delegate**

- Delegate
  - Delegate types represent references to methods with a particular parameter list and return type
    - Example
      - **EventHandler(Object sender, EventArgs e)**
      - Represents a method that has two parameters, the first one being of type Object and the second being of type EventArgs. Its **return type is void.**
      - Any method, so long as its signature matches that expected by the delegate, can be handled by that delegate.

Role of Each Component - **Delegate**

- The delegate object **contains the subscriber list**.
  - It is actually **implemented as a linked list** where each node of the list contains a pointer to a subscriber's event handler method

- Delegates are types – like classes
  - Except – you declare them with the **delegate keyword** and specify the types of methods they can reference

  e.g. public **delegate** void **ActionEventHandler**(object sender, ActionCancelEventArgs ev);

Role of Each Component - **Publisher**

- A publisher is any **class that can fire an event** and **send event notifications** to interested subscribers

- A **publisher class contains** the following critical elements:
  - An **event** field
    - This is what subscribers subscribe to…
  - An **event notification** method
    - This **activates the subscriber notification process** when the event occurs

Role of Each Component - **Event**

- An event is a **field in a class**
- Events are declared with the event keyword
- **Events** must be a **delegate type**
  - Delegates are objects that contain a list of pointers to subscriber methods that delegate can process
- An **event field will be null** until the first subscriber subscribes to that event

Role of Each Component - **Event Notification Method**

- In addition to an event field a publisher will have **a method whose job it is to start the subscriber notification process** when the event in question occurs
  - An event notification method is **just a normal method**
  - It usually has a parameter of EventArgs or a user-defined subtype of EventArgs.
    - But it can have any number and type of parameters you require

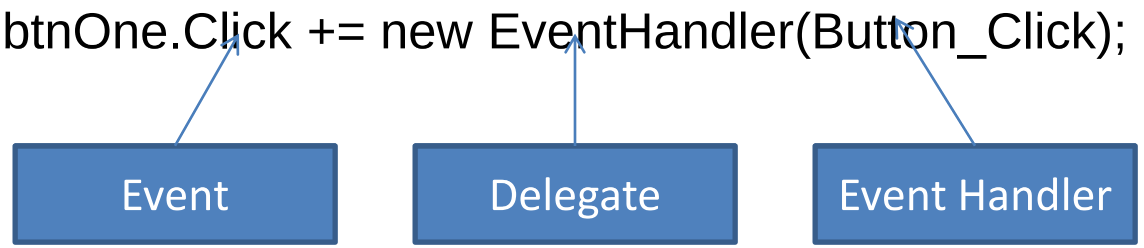
Role of Each Component - **Subscriber**

- A subscriber is a class that **registers its interest in a publisher's events**
- A subscriber class **contains one or more event handler methods**

Role of Each Component **- Event Handler Method**

- An event handler methods is an **ordinary method** that is registered with a publisher’s event.
- The **event handler method’s signature must match** the **signature** required by the **publisher’s event delegate.**

Receiving Event



```
btnOne.Click += new EventHandler(Button_Click);
```

Event

Delegate

Event Handler

EventHandler : Standard delegate defined by .NET Framework

```
private void Button_Click(Object sender, EventArgs e)
{

}
```

Rule of Event Handler

1. Method should return void
2. Parameter to Event Handler will always be 1st Object that raised event.
3. EventArgs object contain imp information about that event
    - On Mouse click, EventArgs contain which button was used, X- Y coordinates

Publishing Event

Event Publisher has following

- Define Event

  public static **event** ActionEventHandler **Action**;

- Define Delegate

  public **delegate** void **ActionEventHandler**(object sender, ActionCancelEventArgs ev);

- Define On*EventName* method

  This method must be named **On***EventName.* The **On***EventName* method raises the event by invoking the delegates.

```
protected void OnAction(object sender, EventArgs ev)
 {
     Action(sender, ev);  // Invoke event
}
```

Simple Event Handler

EventPublisher

```
// Declare a delegate for an event.
delegate void MyEventHandler();

// Declare an EventPublisher class.
class MyEvent
{
   public event MyEventHandler SomeEvent;

   // This is called to fire the event.
   public void OnSomeEvent()
   {
      if (SomeEvent != null)
         SomeEvent();
   } //end OnSomeEvent
} //end MyEvent
```

Event Subscriber

```
public class EventPublisher
   {
      // An event handler.
      static void handler()
      {
         Console.WriteLine("Event occurred");
       }
      public static void Main()
      {
         MyEvent evt = new MyEvent();
         // Add handler() to the event list.
         evt.SomeEvent += new MyEventHandler(handler);
         // Fire the event.
         evt.OnSomeEvent();
      } //end Main
   } //
```

Form Button Click Event Handler

```
public partial class Form1 : Form
   {
      public static event EventHandler FormButtonEvent;
      EventSubscriber mysubscriber = null;

      public Form1()
      {
         mysubscriber = new EventSubscriber();
         InitializeComponent();
      }
      private void button1_Click(object sender, EventArgs e)
      {
         if (FormButtonEvent != null)
            FormButtonEvent(sender, e);  // Invoke event
      }
   }
```

Form Button Click Event Handler

```
class EventSubscriber
  {
    public EventSubscriber()
    {
      Form1.FormButtonEvent += new EventHandler(Form1_FormButtonEvent);
     }

    void Form1_FormButtonEvent(object sender, EventArgs e)
    {
      MessageBox.Show("Received Event");
    }
   }
}// form
```

Evolution of Delegates in C#

- C# 1.0 - Use of **delegate** by explicitly initializing it with a method that was defined elsewhere in the code.

- C# 2.0 – Introduced **concept of anonymous methods** as a way to write unnamed inline statement blocks that **can be executed in a delegate invocation.**

- C# 3.0 introduced **lambda expressions**, which are similar in concept to anonymous methods but **more expressive and concise**.

- *Anonymous functions* = anonymous methods + lambda expressions

- Applications that target version 3.5 and later of the .NET Framework should use **lambda expressions**.

Anonymous Method

- An anonymous method is **a method without a name**.

- Anonymous methods in C# can be **defined using the delegate keyword** and

- **Can be assigned** to a **variable of delegate type**.

Anonymous Method

```
public delegate void Print(int value);

static void Main(string[] args)
{
    int i = 10;

    Print prnt = delegate(int val) {
        val  += i;
        Console.WriteLine("Anonymous method: {0}", val);
    };

    prnt(100);
}
```

Anonymous Method as Parameter

Anonymous methods can also be passed to a method that accepts the delegate as a parameter.

```
public delegate void Print(int value);

class DemoAnonymousMethod
{
    public static void PrintHelperMethod(Print printDel, int val)
    {
        val += 10;
        printDel(val);
    }

    static void Main(string[] args)
    {
        PrintHelperMethod(delegate(int val) { Console.WriteLine("Anonymous method: {0}", val); }, 100);
    }
}
```

```
Anonymous method: 110
```

Anonymous Method Limitations

- It **cannot contain jump statement** like goto, break or continue.
- It **cannot access** ref or out **parameter of an outer method.**
- It **cannot** have or access **unsafe code**.
- It **cannot** be used **on the left side** of the **is operator**

Lambda Expressions

- Similar to Anonymous function

- In Lambda expressions **no need to specify the type of the value that you input** thus making it **more flexible** to use.

- A *lambda expression can be of two type*
  - Expression lambda that has an expression as its body
    (input-parameters) => expression

  - Statement lambda that has a statement block as its body
    (input-parameters) => { <sequence-of-statements> }

Expression lambda

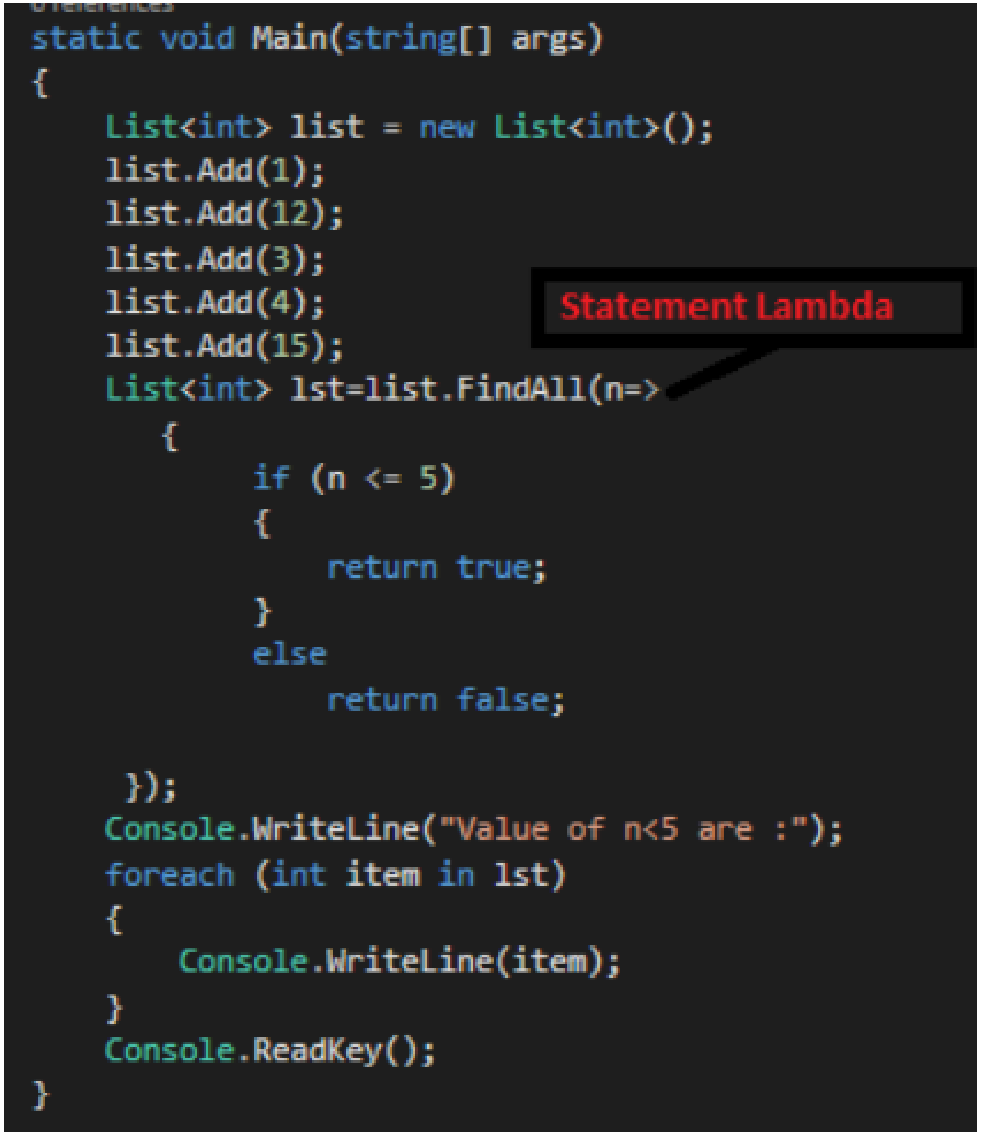
```
public static class demo
{
    public static void Main()
    {
        List<int> list = new List<int>() { 1, 2, 3, 4, 5, 6 };
        List<int> evenNumbers = list.FindAll(x => (x % 2) == 0);

        foreach (var num in evenNumbers)
        {
            Console.Write("{0} ", num);
        }
        Console.WriteLine();
        Console.Read();

    }
}
```

Statement lambda

```
class Dog
{
  public string Name { get; set; }
  public int Age { get; set; }
}
class demo{
  static void Main()
  {
    List<Dog> dogs = new List<Dog>() {
      new Dog { Name = "Rex", Age = 4 },
      new Dog { Name = "Sean", Age = 0 },
      new Dog { Name = "Stacy", Age = 3 }
    };
    var newDogsList = dogs.Select(x => new { Age = x.Age, FirstLetter = x.Name[0] });
    foreach (var item in newDogsList)
    {
      Console.WriteLine(item);
    }
    Console.Read();
  }
```

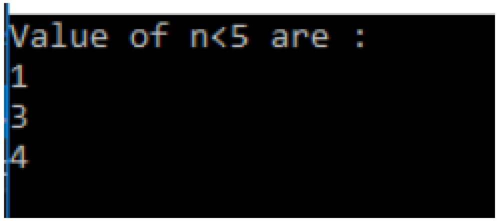
```
{ Age = 4, FirstLetter = R }
{ Age = 0, FirstLetter = S }
{ Age = 3, FirstLetter = S }
```

Statement lambda



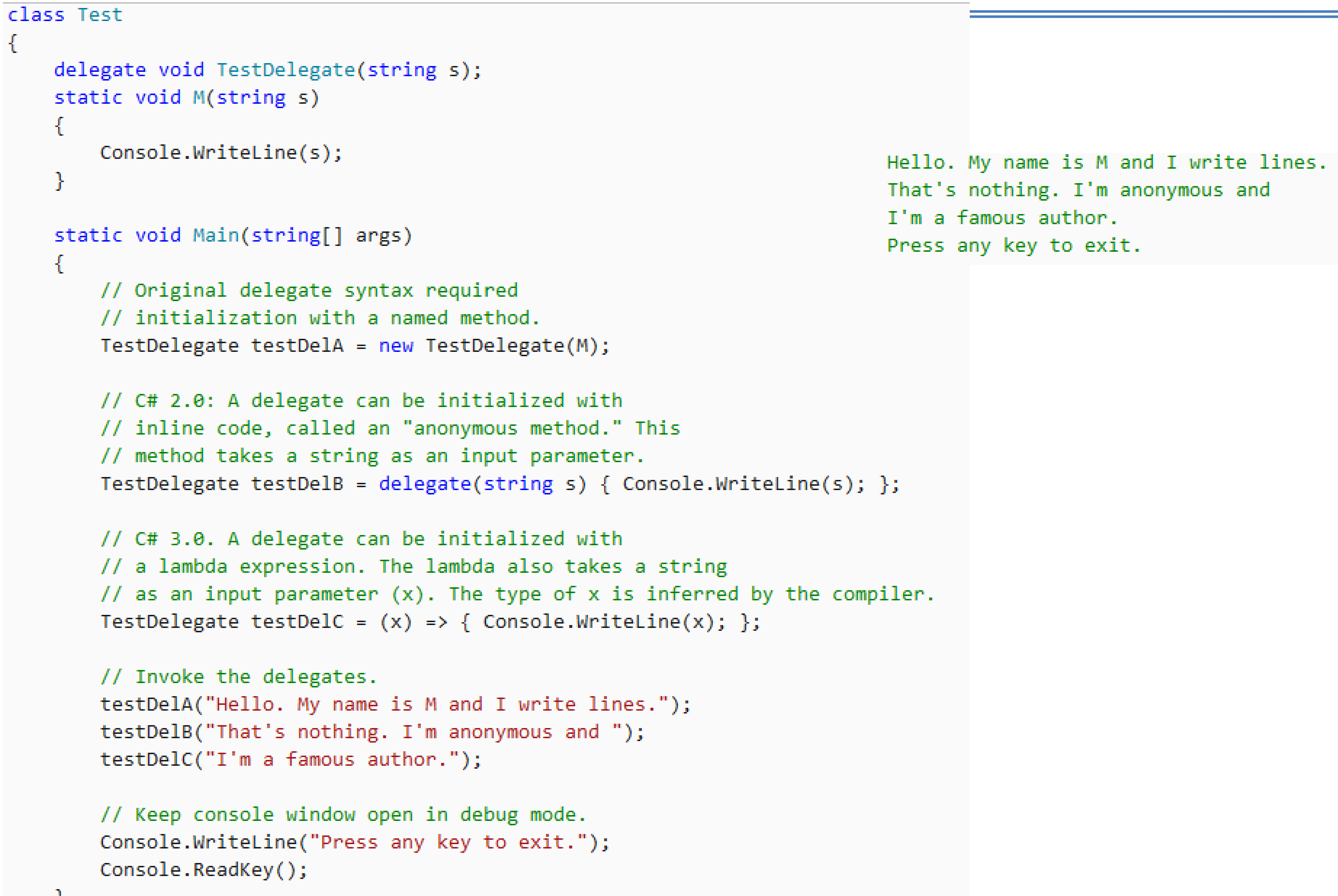
```
static void Main(string[] args)
{
    List<int> list = new List<int>();
    list.Add(1);
    list.Add(12);
    list.Add(3);
    list.Add(4);
    list.Add(15);
    List<int> lst=list.FindAll(n=>
       {
           if (n <= 5)
           {
               return true;
           }
           else
               return false;

     });
    Console.WriteLine("Value of n<5 are :");
    foreach (int item in lst)
    {
        Console.WriteLine(item);
    }
    Console.ReadKey();
}
```

Statement Lambda



```
Value of n<5 are :
1
3
4
```

Overview C# 1.0 to C# 3.0



```
class Test
{
    delegate void TestDelegate(string s);
    static void M(string s)
    {
        Console.WriteLine(s);
    }

    static void Main(string[] args)
    {
        // Original delegate syntax required
        // initialization with a named method.
        TestDelegate testDelA = new TestDelegate(M);

        // C# 2.0: A delegate can be initialized with
        // inline code, called an "anonymous method." This
        // method takes a string as an input parameter.
        TestDelegate testDelB = delegate(string s) { Console.WriteLine(s); };

        // C# 3.0. A delegate can be initialized with
        // a lambda expression. The lambda also takes a string
        // as an input parameter (x). The type of x is inferred by the compiler.
        TestDelegate testDelC = (x) => { Console.WriteLine(x); };

        // Invoke the delegates.
        testDelA("Hello. My name is M and I write lines.");
        testDelB("That's nothing. I'm anonymous and ");
        testDelC("I'm a famous author.");

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
```

```
Hello. My name is M and I write lines.
That's nothing. I'm anonymous and
I'm a famous author.
Press any key to exit.
```

Func<T>

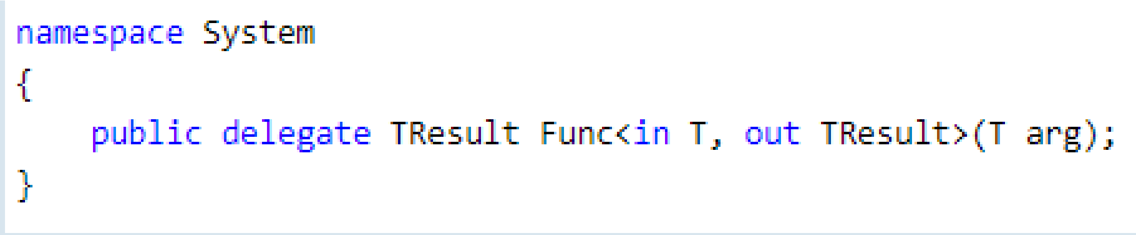
- C# 3.0 includes built-in **generic delegate types Func and Action**, so no need to define custom delegates like

  public delegate int SomeOperation(int i, int j);

- **Func** is a **generic delegate** included in the **System** namespace.

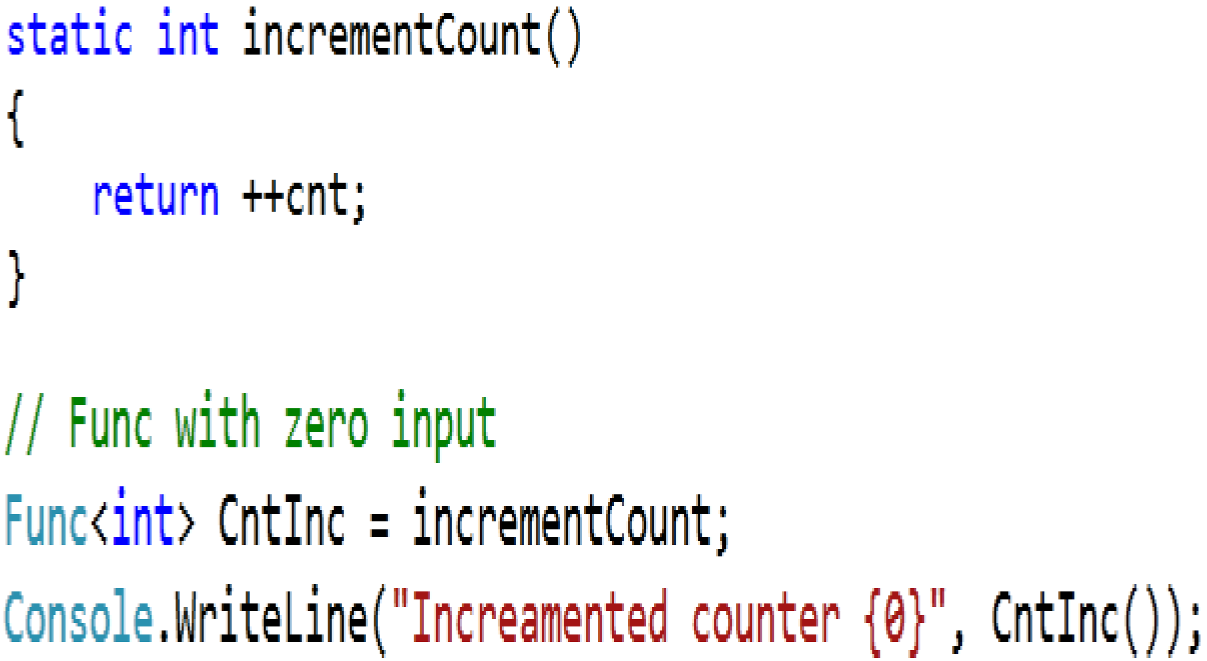
- It has **zero or more *input* parameters** and **one *out* parameter**.
- It **must include an out parameter** for the result

- Max 16 input parameters are allowed



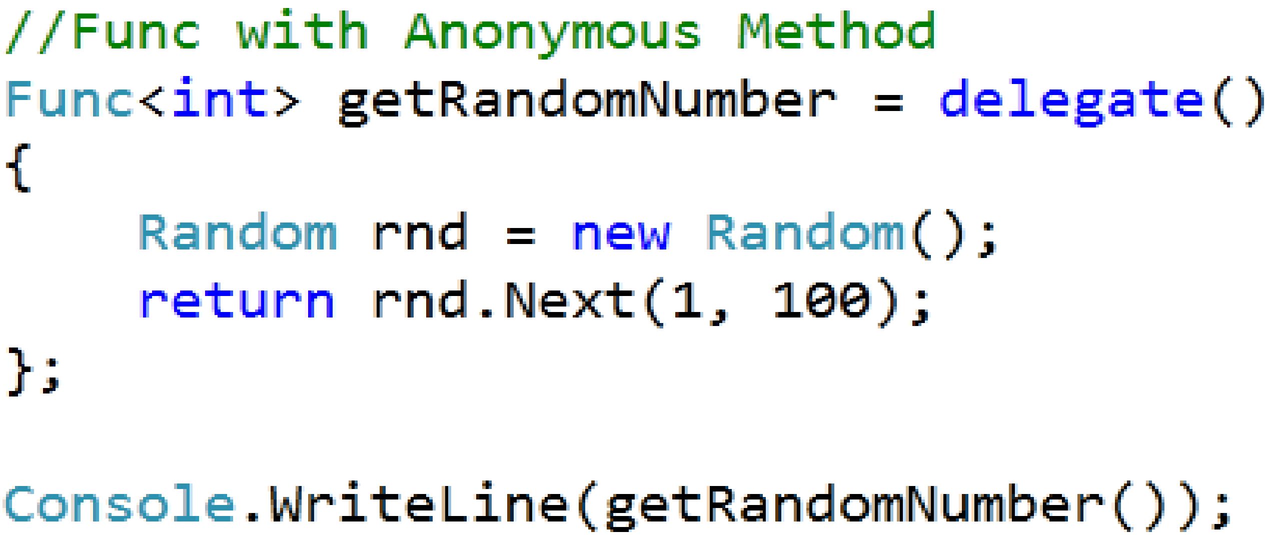
```
namespace System
{
    public delegate TResult Func<in T, out TResult>(T arg);
}
```

Func with Zero Parameter



```
static int incrementCount()
{
    return ++cnt;
}

// Func with zero input
Func<int> CntInc = incrementCount;
Console.WriteLine("Increamented counter {0}", CntInc());
```

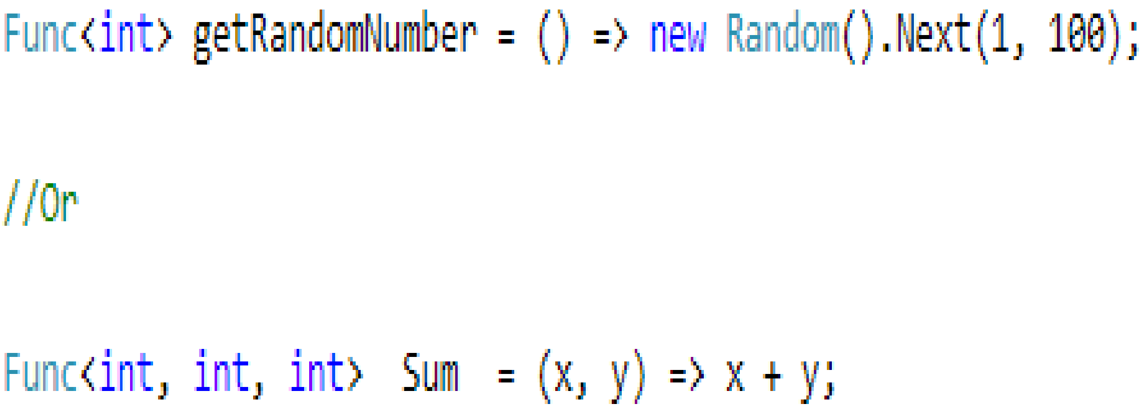
Func with Anonymous Method



```
//Func with Anonymous Method
Func<int> getRandomNumber = delegate()
{
    Random rnd = new Random();
    return rnd.Next(1, 100);
};

Console.WriteLine(getRandomNumber());
```

Demo

Func with lambda expression



```
Func<int> getRandomNumber = () => new Random().Next(1, 100);


//Or


Func<int, int, int>  Sum  = (x, y) => x + y;
```

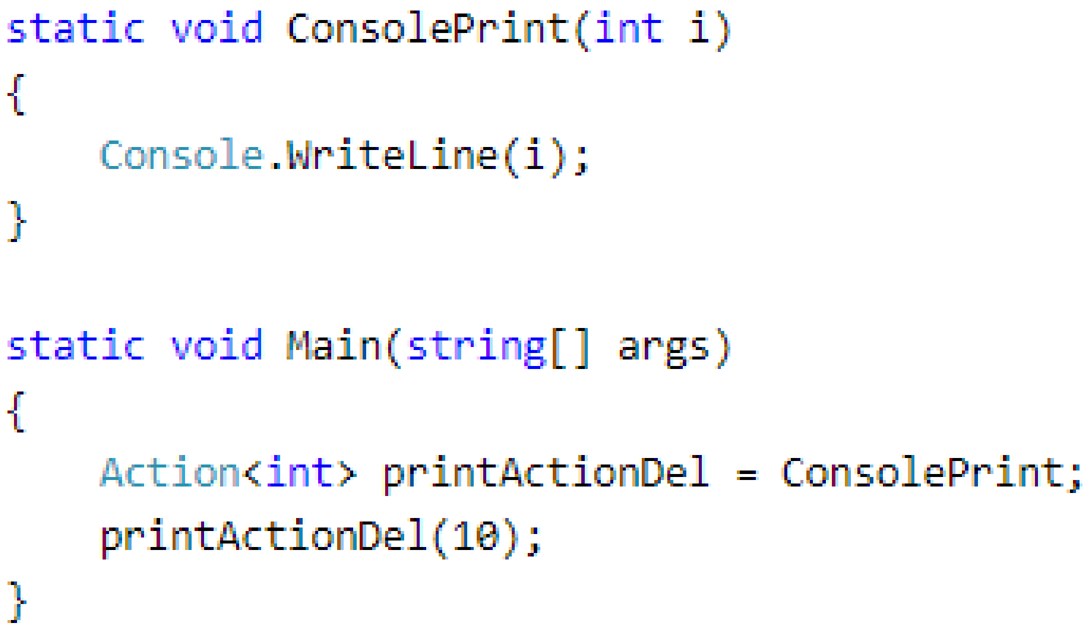
Func<> Summary

- Func is **built-in delegate type**.
- Func delegate type **must return a value**.
- Func delegate type can have **zero to 16 input** parameters.
- Func delegate does not allow ref and out parameters.
- Func delegate type can be used with an anonymous method or lambda expression

Action <T>

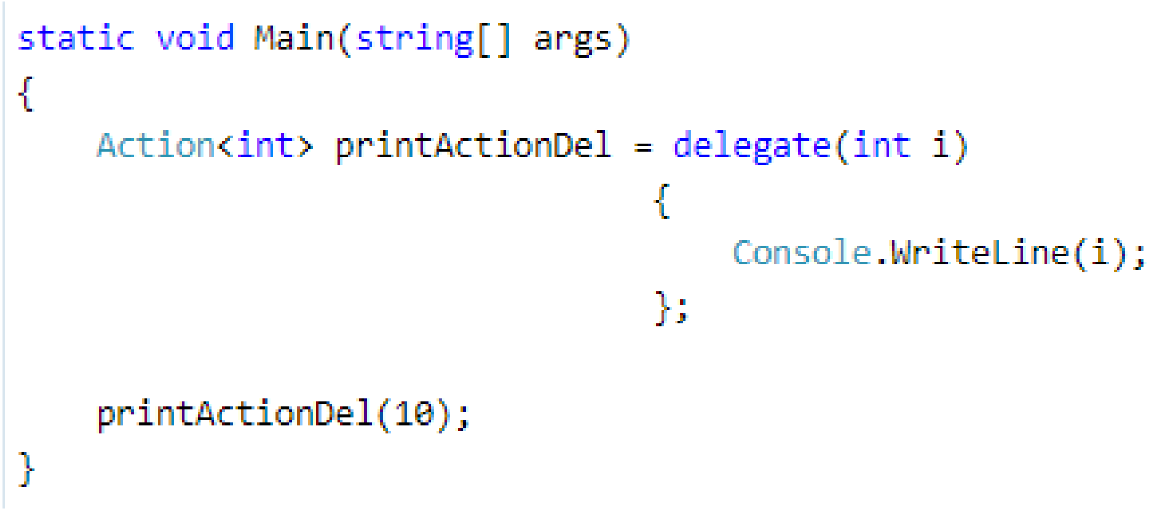
- An Action type delegate is the standard delegate
- Action delegate **doesn't return a value**
  - An Action delegate **can be used with a method that has a void return type**
- An Action delegate can take up to 16 input parameters of different types.
- An **Anonymous method** and **Lambda expression** can also be assigned to an Action delegate

Action <T>



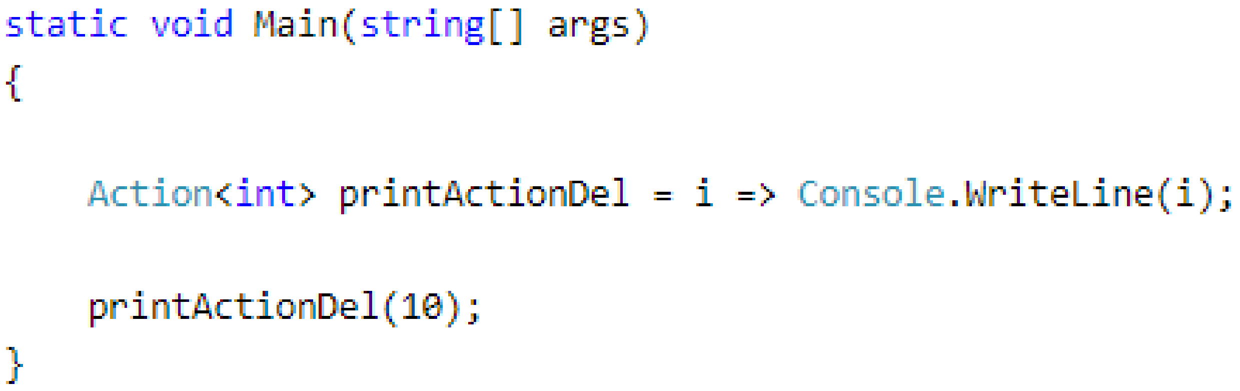
```
static void ConsolePrint(int i)
{
    Console.WriteLine(i);
}

static void Main(string[] args)
{
    Action<int> printActionDel = ConsolePrint;
    printActionDel(10);
}
```

Action <T> with Anonymous Method



```
static void Main(string[] args)
{
    Action<int> printActionDel = delegate(int i)
                                {
                                    Console.WriteLine(i);
                                };

    printActionDel(10);
}
```

Action <T> with lambda expression



```
static void Main(string[] args)
{

    Action<int> printActionDel = i => Console.WriteLine(i);

    printActionDel(10);
}
```

Advantages of Action and Func Delegate

- Easy and quick to define delegates.
- Makes code short.
- Compatible type throughout the application.