Chapter: 7

C# Only : Delegates, Events, and Namespaces

Delegates, Anonymous methods, Events, Multicast events, Namespaces, The using directive

Delegates are, essentially, *objects* that can refer to *executable code*. ***Events*** are built on ***delegates***. An ***event*** is a *notification* that some *action* occurred.

C#_7.1 Delegates (Similar to function pointer in C/C++, Recall C/C++ 5.8)

A ***delegate*** is an ***object*** that can *refer to a method*. Thus, when you create a ***delegate***, you are creating an ***object*** that can *hold a reference* to a *method*. Furthermore, the ***method*** can be ***called*** through this ***reference***. Thus, a ***delegate*** can *invoke the method* to which it *refers*. A ***delegate*** in ***C#*** is similar to a ***function pointer*** in ***C/C++***.

❖ Once a ***delegate*** refers to a ***method***, the *method can be called through that delegate*. Furthermore, the ***same delegate*** can be used to call a ***different method*** by *simply changing the method* to which the delegate ***refers***. The principal advantage of a delegate is that it allows you to specify a ***call*** to a ***method***, but the ***method*** actually ***invoked*** is determined at ***runtime***, not at ***compile time***.

❒ A ***delegate type*** is declared using the keyword ***delegate***. The general form of a delegate declaration is: ***delegate ret-type name(parameter-list);***

☞ Here, ***ret-type*** is the *type of value* returned by the ***methods*** that the ***delegate*** will be calling. The *name of the delegate* is specified by ***name***. The *parameters* required by the methods called through the delegate are specified in the ***parameter-list***.

☞ Once created, a ***delegate instance*** can ***refer to*** and ***call*** only methods whose ***return type*** and ***parameter list*** match those specified by the ***delegate declaration***.

❒ A ***delegate*** can be used to ***call*** any method that agrees with its ***signature*** and ***return type***. Furthermore, the ***method*** can be specified at ***runtime*** by simply ***assigning*** to the ***delegate*** *a reference to a compatible method*. The method invoked can be an ***instance method*** associated with an ***object*** or a ***static method*** associated with a ***class***. All that matters is that the ***signature*** and ***return type*** of the ***method*** agree with that of the ***delegate***.

C#_Example 1: To see delegates in action, let's begin with the simple example shown here:

```
using System;
delegate string StrMod(string str); // Declare a delegate: A delegate called StrMod

class DelegateTest {
    // Replaces spaces with hyphens
        static string ReplaceSpaces(string a) {   Console.WriteLine("Replaces spaces with hyphens.");   return a.Replace(' ', '-'); }
    // Remove spaces.
        static string RemoveSpaces(string a) {    string temp = "";     int i;     Console.WriteLine("Removing spaces.");
                                                  for(i=0; i < a.Length; i++){ if(a[i] != ' ') temp += a[i]; }     return temp; }
    // Reverse a string.
        static string Reverse(string a) {         string temp = "";     int i, j;    Console.WriteLine("Reversing string.");
                                                  for(j=0, i=a.Length-1; i >= 0; i--, j++) {temp += a[i];}     return temp;  }
    static void Main() {  string str;
                StrMod strOp = ReplaceSpaces;                    // Construct a delegate instance
// Call methods through the delegate.
/* ReplaceSpaces already  referred */           str = strOp("This is a test."); Console.WriteLine("Resulting string: " + str); Console.WriteLine();
                strOp = RemoveSpaces;   str = strOp("This is a test."); Console.WriteLine("Resulting string: " + str); Console.WriteLine();
                strOp = Reverse;        str = strOp("This is a test."); Console.WriteLine("Resulting string: " + str);       }}
```

- The program declares a ***delegate*** called ***StrMod*** that takes one ***string parameter*** and ***returns*** a ***string***.
- In ***DelegateTest***, three ***static methods*** are declared, each with a ***matching signature***. These methods perform some type of ***string modification***. Notice that ***ReplaceSpaces( )*** uses one of string's methods, called ***Replace( )***, to replace ***spaces*** with ***hyphens***.
- In ***Main( )***, a ***StrMod*** reference called ***strOp*** is created and assigned a ***reference*** to ***ReplaceSpaces( )***. Pay close attention to: ***StrMod strOp = ReplaceSpaces;***
    - ▶ ***method group conversion:*** Notice how the method ***ReplaceSpaces( )*** is assigned to ***strOp***. *Only its* ***name*** *is used;* ***no parameters*** *are specified. [C# automatically provides a conversion from the method to the* ***delegate type****. This is called a* ***method group conversion****].* This conversion is really just *shorthand* for this *longer* form: ***StrMod strOp = new StrMod(ReplaceSpaces);***

      In this form, a *new delegate* is *explicitly instantiated* using ***new***.
    - ▶ With either approach, the ***method***'s ***signature*** must match that of the ***delegate***'s ***declaration***. If it doesn't, a ***compile-time*** error will result.
- Next, ***ReplaceSpaces( )*** is called through the *delegate instance* ***strOp***, as shown here: ***str = strOp("This is a test.");***
  Because ***strOp*** refers to ***ReplaceSpaces( )***, it is ***ReplaceSpaces( )*** that is ***invoked***.
- Next, ***strOp*** is assigned a reference to ***RemoveSpaces( )***, and then ***strOp*** is called again. This time, ***RemoveSpaces( )*** is invoked.
- Finally, ***strOp*** is assigned a reference to ***Reverse( )*** and ***strOp*** is called. This results in ***Reverse( )*** being called.

C#_7.2 Use Instance Methods as Delegates

C#_Example 2: A ***delegate*** can also refer to ***instance methods*** through an ***object reference***. For example, here is a rewrite of the previous example, which ***encapsulates*** the string operations inside a ***class*** called ***StringOps***:

```
using System;
delegate string StrMod(string str);                    // Declare a delegate type.

class StringOps {
   public string ReplaceSpaces(string a) { Console.WriteLine("Replaces spaces with hyphens."); return a.Replace(' ', '-'); }
   public string RemoveSpaces(string a) { string temp = "";   int i;   Console.WriteLine("Removing spaces.");   for(i=0; i < a.Length; i++) {if(a[i] != ' ') temp += a[i];}   return temp;}
   public string Reverse(string a) { string temp = "";    int i, j;    Console.WriteLine("Reversing string.");    for(j=0, i=a.Length-1; i >= 0; i--, j++) {temp += a[i];}    return temp; }
}

class DelegateTest { static void Main() {          string str;   StringOps so = new StringOps();
                                                   StrMod strOp = so.ReplaceSpaces;    // Construct a delegate: Create a delegate using an instance method.
// Call methods through a delegate.
            str = strOp("This is a test.");   Console.WriteLine("Resulting string: " + str); Console.WriteLine();    // ReplaceSpaces already  referred during  delegate construction
            strOp = so.RemoveSpaces; str = strOp("This is a test.");   Console.WriteLine("Resulting string: " + str);     Console.WriteLine();
            strOp = so.Reverse; str = strOp("This is a test.");          Console.WriteLine("Resulting string: " + str);        }}
```

- In this case, the ***delegate*** refers to *methods on an instance* of ***StringOps***.

C#_7.3 Multicasting through Delegates

Multicasting is the ability to create a ***chain of methods*** that will be called ***automatically*** when a ***delegate*** is ***invoked***. Such a chain is very easy to create. Simply ***instantiate*** a ***delegate***, and then use the "**+**" or "**+=**" operator to ***add methods*** to the ***chain***. To ***remove*** a method, use "**−**" or "**−=**".

- If the ***delegate*** returns a *value*, then the *value* returned by the ***last method*** in the ***list*** becomes the ***return value*** of the entire ***delegate invocation***. For this reason, a ***delegate*** that will make use of ***multicasting*** will often have a ***void return type***.

C#_Example 3: ***C#_Example 2*** by changing the *string manipulation method*'s *return type* to ***void*** & using a ***ref*** parameter to ***return*** the ***altered string*** to the caller.

```
using System;
delegate void StrMod(ref string str);                 // Declare a delegate type: notice "ref" is used.

/* methods are no more public. Notice "ref" is used */
class StringOps {
static void ReplaceSpaces(ref string a) { Console.WriteLine("Replaces spaces with hyphens.");    a = a.Replace(' ', '-'); }
static void RemoveSpaces(ref string a) { string temp = ""; int i;     Console.WriteLine("Removing spaces.");    for(i=0; i < a.Length; i++) { if(a[i] != ' ') temp += a[i]; }    a = temp; }
static void Reverse(ref string a) { string temp = ""; int i, j;    Console.WriteLine("Reversing string.");   for(j=0, i=a.Length-1; i >= 0; i--, j++) {temp += a[i]; }     a = temp; }
```

```
static void Main() {

// Construct a delegate.
    StrMod strOp;
    StrMod replaceSp = ReplaceSpaces;
    StrMod removeSp = RemoveSpaces;
    StrMod reverseStr = Reverse;
    string str = "This is a test";
```

```
/* Create a multicast: Replaces spaces with hyphens also reversing string. Add multipe methods*/
strOp = replaceSp;        strOp += reverseStr;
strOp(ref str);    // Invoke multicast.
Console.WriteLine("Resulting string: " + str);   Console.WriteLine();

/* Remove two or more method:  Remove replaceSp and add removeSp also reversing string. */
strOp -= replaceSp;        strOp += removeSp;
str = "This is a test.";    strOp(ref str);       //Reset string and Invoke multicast.
Console.WriteLine("Resulting string: " + str);   Console.WriteLine(); }}
```

```
OUTPUT:

Replaces spaces with hyphens.
Reversing string.
Resulting string: tset-a-si-sihT
Reversing string.
Removing spaces.
Resulting string: .tsetasisihT
```

- In **Main( )**, four **delegate instances** are created. One, **strOp**, is **null**. The other **three** refer to specific **string modification methods**. Next, a **multicast** is created that calls **ReplaceSpaces( )** and **Reverse( )**. This is accomplished via the following lines: ***`strOp = replaceSp; strOp += reverseStr;`***
    - First, **strOp** is assigned a **reference** to **replaceSp**. Next, using **+=**, **reverseStr** is added. When **strOp** is invoked, both methods are invoked, replacing **spaces** with **hyphens** and *reversing the string*, as the output illustrates.
- Next, **replaceSp** is removed from the chain using this line: ***`strOp -= replaceSp;`*** and **removeSp** is added using this line: ***`strOp += removeSp;`*** Then, **StrOp** is again **invoked**. This time, **spaces** are **removed** and the string is **reversed**.

Uses of delegates: ***delegates*** support ***events***. ***delegates*** give your program a way to ***execute*** methods at ***runtime*** without having to specify what that method is at ***compile time***. This ability is quite useful when you want to create a ***framework*** that allows components to be ***plugged*** in.

C#_7.4 Anonymous Methods

When working with **delegates**, you will often find that the **method referred to by a delegate** is used only for that purpose. i.e, the only reason for the method is so that it can be **invoked** via a **delegate**. The method is never called on its own. In such a case, you can avoid the need to create a separate method by the use of an **anonymous method**.

- An **anonymous method** is, essentially, a **block of code** that is passed to a **delegate constructor**. One advantage to using an anonymous method is simplicity. There is no need to declare a **separate method** whose only purpose is to be passed to a **delegate**.

```
using System;
delegate void CountIt();
class AnonMethDemo { static void Main() {
```

```
// Here, the code for counting is passed as an anonymous method. Notice the semicolon
CountIt count = delegate { for(int i=0; i <= 5; i++) Console.Write(i + " "); };  // This is the block of code passed to the delegate.
count();    Console.WriteLine();  }}
```

- This program first declares a delegate type called **CountIt** that has *no parameters* and returns **void**. Inside **Main( )**, a **CountIt** delegate called **count** is created, and it is passed the block of code that follows the **delegate** keyword. This block of code is the **anonymous method** that will be executed when **count( )** is called. Notice that the **block of code is followed by a semicolon**, which terminates the **declaration** statement.

- ***Parameterized anonymous method:*** To do so, **follow** the **`delegate`** keyword with a parenthesized **parameter list**. Then, pass the argument(s) to the **delegate instance** when it is **called**. For example, here is the preceding program rewritten so that the ending value for the count is passed:

```
delegate void CountIt(int end);
class AnonMethDemo2 { static void Main() {        CountIt count = delegate (int end) { for(int i=0; i <= end; i++) Console.Write(i + " "); };
                                                  count(3); count(5);   }}
```

- In this version, **CountIt** now takes an **integer argument**. Notice how the **parameter list** is specified after the **delegate** keyword when the **anonymous method** is created. In this case, the **only parameter** is **end**, which is an **int**. The *code inside the anonymous method has access to the parameter* **end** in just the same way it would if a “normal” method were being created.
- As you can see, the **value** of the argument to **count( )** is received by **end**, and this value is used as the stopping point for the **count**.

- ***Anonymous method returns a value:*** The value is returned by use of the **return** statement, which works the same in an **anonymous method** as it does in a **named method**. As you would expect, the **type** of the **return value** must be compatible with the **return type** specified by the **delegate**.

```
delegate int CountIt(int end);
class AnonMethDemo3 { static void Main() { int result;
CountIt count = delegate (int end) {   int sum = 0;
                                       for(int i=0; i <= end; i++) { Console.Write(i + " "); sum += i; }
                                       return sum; /* return a value from an anonymous method */  };
```

```
result = count(3); Console.WriteLine("\nSummation of 3 is " + result);
Console.WriteLine();
result = count(5); Console.WriteLine("\nSummation of 5 is " + result);
Console.WriteLine();      }}
```

- Here, the value of **sum** is returned by the **code block** that is associated with the **count delegate instance**. Notice that the **return** statement is used like **normal way**.

- ***Outer variables:*** A **local variable** or **parameter** whose **scope** includes an **anonymous method** is called a **outer variable**. *An anonymous method has access to* and can use these **outer variables**. When an **outer variable** is used by an **anonymous method**, that variable is said to be **captured**.
    - Even though a **local** variable will normally cease to exist when its **block** is **exited**, if that **local** variable is being used by an **anonymous method** (i.e. **captured**), then that variable will stay in **existence** at least until the **delegate** referring to that method is **destroyed**.

NOTE:
- **Lambda expression** improves on the concept of the **anonymous method**. Although **lambda expressions** are often a better option, they are not applicable to all situations. Also, **anonymous methods** are widely used in existing C# code.
- Perhaps the most important use of **anonymous methods** is with **events**. An anonymous method is the most efficient means of coding an **event handler**.

C#_7.5 Events

An **event** is, essentially, an **automatic notification** that some **action has occurred**. Events are used to represent things such as **keystrokes**, **mouse clicks**, **repaint requests**, and **incoming data**. Events are built upon the foundation of the **delegate**.

- ***Events work like this:*** *An object that has an interest in an event* **registers an event handler** for that *event*. When the ***event*** occurs, all ***registered handlers*** are called. ***Event handlers*** are represented by ***delegates***. The ***event handler*** responds to the ***event*** by taking appropriate action. For example, an ***event handler*** for ***keystrokes*** might respond by ***echoing*** the ***character*** to the ***screen***.
    - As a *general rule*, an *event should respond quickly and then return*. It should not maintain **control** of the **CPU** for an extended period of time.
    - **Events** are *members of a class* and are *declared* using the **event** keyword. Its *general form is*:

      `event event_delegate event_name;`

        - Here, ***event_delegate*** is the name of the ***delegate*** used to support the ***event***, and ***event_name*** is the ***name*** of the specific ***event*** being declared.

Simple Event Example

```
using System;
delegate void MyEventHandler();    //Create a delegate for the event

// Declare a class that contains an event.
class MyEvent {  public event MyEventHandler SomeEvent; // Declare an event.
                 public void Fire(){ if(SomeEvent != null) SomeEvent(); /* event Generator */ } }

class EventDemo {
      static void Handler() { Console.WriteLine("Event occurred"); }
      static void Main() {
                        MyEvent evt = new MyEvent();
/* Multicast*/          evt.SomeEvent += Handler;  // Add Handler() to the event chain.
                        evt.Fire();   /*Generate the event */   }}
```

- First a **delegate type** for the **event** handler is declared: `delegate void MyEventHandler();` All **events** are activated through a **delegate**.
  - The ***event delegate type*** defines the ***return type*** and ***signature*** for the ***event***. In this case, there are ***no parameters***, but *event parameters* are allowed.
- Next, an *event class*, called ***MyEvent***, is created. Inside the class, an ***event*** called ***SomeEvent*** is declared, using: *public event MyEventHandler SomeEvent;*
  - The keyword ***event*** tells the compiler than an ***event*** is being declared.
  - The method ***Fire()***, which is the method that a program will call to **signal** (or "***fire***") an ***event***. It calls an ***event handler*** through the ***SomeEvent*** delegate, as: ***if(SomeEvent != null) SomeEvent();***
- Notice that a **handler** is called *if and only if* **SomeEvent** is **not null**. Since other parts of your program **must register** an **interest** in an **event** in order to receive **event notifications**, it is possible that **Fire( )** could be called **before any event handler** has been **registered**. To prevent calling on a **null reference**, the **event delegate** must be **tested** to ensure that it is **not null**.
- Inside ***EventDemo***, an ***event handler*** called ***Handler()*** is created, which just displays a message.
- In ***Main()***, a ***MyEvent object*** is created and ***Handler()*** is *registered as a handler* for this **event**, as: ***MyEvent evt = new MyEvent();***
- ***Handler()*** is then added to the **event list/chain**: ***evt.SomeEvent += Handler;*** Notice that the handler is added using the **+=** operator. Events support only **+=** and **–** **=** for ***adding*** or ***removing handlers***.
- Finally, the **event** is **fired**, as: ***evt.Fire();*** Calling ***Fire()*** causes all ***registered event handlers*** to be called. In this case, there is ***only one*** registered handler.

C#_7.6 Multicasting Event

C#_Example 4 **(Example of a Multicast Event)** Events can be multicast. This enables multiple objects to respond to an event notification.

```
using System;
delegate void MyEventHandler();   //delegate for event

class MyEvent { public event MyEventHandler SomeEvent;   //event declaration
                public void Fire() { if(SomeEvent != null) SomeEvent(); }   /*Event generator*/   }

class X { public void Xhandler() { Console.WriteLine("Event received by X object"); }   }

class Y { public void Yhandler() { Console.WriteLine("Event received by Y object"); }   }

class EventDemo {
        static void Handler() { Console.WriteLine("Event received by EventDemo"); }
        static void Main() {     MyEvent evt = new MyEvent();
                                 X xOb = new X();          Y yOb = new Y();

// Add the handlers to the event list. Multicasting.
evt.SomeEvent += Handler;
evt.SomeEvent += xOb.Xhandler;
evt.SomeEvent += yOb.Yhandler;

evt.Fire();   // Fire or Generate the event.
Console.WriteLine();

// Remove a handler.
evt.SomeEvent -= xOb.Xhandler;
Console.WriteLine("After removing xOb.Xhandler");
evt.Fire();   /* Fire or Generate the event again */              }}
```

- This example creates two additional **classes**, called **X** and **Y**, which also define **event** handlers compatible with **MyEventHandler**. Thus, these handlers can also become part of the **event chain**.
- Notice that the ***handlers*** in ***X*** and ***Y*** are ***not static***. This means that ***objects*** of each ***must be created***, and the ***handler*** linked to an ***object instance*** is added to the ***event chain***. When the event is ***fired***, those handlers in the ***event chain*** are called.
  - Therefore, if the **contents** of the **event chain** change, **different handlers** are called. This is illustrated by the program when it removes ***xOb.Xhandler***.

C#_Example 5 **(Multicasting Event to different objects of a class)**: Understand that **events** are **sent** to specific **object instances**, not ***GENERICALLY*** to a **class**. Thus, ***each object*** of a ***class*** must ***register*** to receive an ***event notification***. Eg: following multicasts an event to **three objects** of type **X**:

```
using System;
delegate void MyEventHandler();

class MyEvent { public event MyEventHandler SomeEvent;
public void Fire() { if(SomeEvent != null) SomeEvent(); }      }

class X {  int id; public X(int x) { id = x; }
         public void Xhandler(){ Console.WriteLine("Event received by object" + id); } }

class EventDemo { static void Main() { MyEvent evt = new MyEvent();
                                       X o1 = new X(1);
                                       X o2 = new X(2);
                                       X o3 = new X(3);
    evt.SomeEvent += o1.Xhandler;   /* Each object that wants to receive an */
    evt.SomeEvent += o2.Xhandler;   /* event must add its own handler to the chain. */
    evt.SomeEvent += o3.Xhandler;                evt.Fire();   /* Fire the event. */  }}
```

C#_7.7 Anonymous Methods with Events

Anonymous methods are especially useful when working with **events** because an anonymous method can serve as an **event handler** *[Often, the event handler is not called by any code other than the event handling mechanism. Thus, there is usually no reason for a stand-alone method.]*. This eliminates the need to declare a **separate** method, which can significantly streamline **event-handling code**. The syntax for using an anonymous event handler is the same as that for using an anonymous method with any other type of delegate. Here is an example that uses an anonymous event handler:

```
delegate void MyEventHandler();
class MyEvent { public event MyEventHandler SomeEvent; public void Fire() { if(SomeEvent != null) SomeEvent(); } }
class AnonMethHandler {     static void Main() { MyEvent evt = new MyEvent();
                            evt.SomeEvent += delegate { Console.WriteLine("Event received."); }; // Use an anonymous method as an event handler.
                            evt.Fire(); evt.Fire();/* Fire the event twice */ }}
```

- Pay special attention to the way the anonymous event handler is added to the event by the following code sequence:

```
evt.SomeEvent += delegate { Console.WriteLine("Event received."); };
```

NOTE: Must use ***Special techniques*** to create events and ***event handlers*** that are ***compatible*** with the types of events used by the ***.NET Framework***.

- Although C# allows you to write any type of event that you desire, for component compatibility with the ***.NET Framework***, you must follow ***Microsoft's guidelines*** in this regard. At the core of these guidelines is the requirement that ***event handlers*** have two ***parameters***.
  - The ***first*** is a reference to the ***object that generated*** the event.
  - The ***second*** is a parameter of type ***EventArgs*** that contains any ***other information*** required by the ***handler***.
- For simple events in which the ***EventArgs*** parameter is unused, the delegate type can be ***EventHandler***. This is a predefined type that can be used to declare events that provide no extra information.

C#_7.8 Namespaces

A **namespace** defines a **declarative region** that provides a way to keep one **set of names** separate from **another**. Names declared in one namespace **will not conflict** with the same names declared in **another**. The **namespace** used by the **.NET Framework library** (which is the **C# library**) is **System**. This is why you have included **"using System;"**

❐ ***Namespace Declaration:*** A namespace is declared using the namespace keyword. The general form is: ***namespace name {*** */* members */* ***}***

- Here, **name** is the name of the namespace. A **namespace declaration** defines a **scope**. Anything declared immediately inside the **namespace** is in **scope** throughout the **namespace**. Within a **namespace**, you can declare **classes**, **structures**, **delegates**, **enumerations**, **interfaces**, or **another namespace**.

C#_Example 6*:** Following creates a namespace called ***Counter. It ***localizes*** the name used to implement a simple *countdown counter class* called ***CountDown***.

```
namespace Counter {
        class CountDown {      int val;
                public CountDown(int n) { val = n; }
                public void Reset(int n) { val = n; }
                public int Count() { if(val > 0) return val--; else return 0; }
                }
        }   // end of the Counter namespace.
```

- Here is a program that demonstrates the use of the ***Counter namespace***:

```
using System;
class NSDemo {          static void Main() { int i;
                        Counter.CountDown cd1 = new Counter.CountDown(10);

                        do { i = cd1.Count(); Console.Write(i + " "); } while(i > 0);
                        Console.WriteLine();    }}
```

- **CountDown** is declared within the **scope** defined by the **Counter namespace**.
- Put this code into a **file** called **Counter.cs**.
- Put this code into a file called **NSDemo.cs**
- use this **command** line to compile the program: `csc NSDemo.cs Counter.cs`
- To **compile** this program, you must include both the code of **NSDemo class** and the code contained in the **Counter namespace**.
  - Since **CountDown** is declared within the **Counter namespace**, when an object is created, **CountDown** must be qualified with **Counter**, as shown here:
    `Counter.CountDown cd1 = new Counter.CountDown(10);`
  - To use a **member** of a **namespace**, you must qualify it with the **namespace name**. If you don't, the *member of the namespace* won't be found by the **compiler**.
  - Once an ***object*** of ***type Counter*** has been created, it is ***not necessary*** to ***further qualify*** it or any of its **members** with the **namespace**. Thus, **`cd1.Count()`** can be called directly without **namespace qualification**, as this line shows: ***`i = cd1.Count();`***
  - Although this example used two separate files, one for **Counter namespace** and the other for **NSDemo program**, both could have been contained in the **same file**.
    - When a **named namespace** ends, the **outer namespace *resumes***, which in this case, is the **global namespace**. Furthermore, a **single file** can contain **two** or **more** named **namespaces**. *(Similar to the way that a file can contain **two** or more separate **classes**).* Each namespace defines its ***own*** declarative ***region***.

C#_7.10 USING directive

using can be used to bring **namespaces** that you create **into view**. There are **two forms** of the using **directive**. The first is shown here: ***`using name;`***

- Here, **name** specifies the *name of the namespace* you want to access.
  - All of the **members** defined within the **specified namespace** are brought into **view** and can be used **without qualification**.
  - A **using directive** must be specified at the **top of each file**, **prior** to any **other declarations**, or at the start of a **namespace body**. For example:

```
using Counter;
class NSDemo { static void Main() {CountDown cd1 = new CountDown(10);   // now, CountDown can be used directly.
/* same as C#_Example 6 */    }}
```

- Using **one namespace** does not override **another**. When you bring a ***namespace*** into view, it simply lets you use its *names without qualification*. Thus, in the example, both ***System*** and ***Counter*** have been brought into view.

A Second Form of using: The **using directive** has a **second form** which creates **another name**, called an **alias** *(name hides)*, for a **namespace** or a **type**. This form is:

`using alias = name;`

- Here, **alias** becomes ***another name** for the **type** (such as a **class type**)* or **namespace** specified by **name**. Once the **alias** has been created, it can be used in place of the **original name**. For Example consider the previous program: An **alias** for **Counter.CountDown** called **MyCounter** is created.

```
using MyCounter = Counter.CountDown;          // Create an alias for Counter.CountDown. Brings Counter's class Countdown into view.
class NSDemo { static void Main() { MyCounter cd1 = new MyCounter(10);     /* use Counter's class Countdown directly */
                                /* same as C#_Example 6 */          }}
```

- Here, ***`MyCounter`*** is an **alias** for the **type `Counter.CountDown`**. Once ***MyCounter*** has been specified as an ***alias***, it can be used to declare ***objects*** without any further ***namespace qualification***. For example, in the program, this line *MyCounter cd1 = new MyCounter(10);* creates a **CountDown** object.

C#_7.11 Namespaces: Advanced

Namespaces Are Additive: There can be **more than one** *namespace declaration* of the **same name**. This allows a **namespace** to be **split over several files** or even **separated** within the **same file**. For example, the following code also specifies the Counter namespace. It adds a class called CountUp, which counts up, rather than down.

```
namespace Counter {
        class CountDown {      int val;
                public CountDown(int n) { val = n; }
                public void Reset(int n) { val = n; }
                public int Count() { if(val > 0) return val--; else return 0; }
                }
}   // end of the Counter namespace.
```

```
// Here is another part of the Counter namespace.
namespace Counter { class CountUp { int val, target;
    /* Read-only property */    public int Target { get{ return target; } }
                                public CountUp(int n) { target = n; val = 0; }
                                public void Reset(int n) { target = n; val = 0; }
                                public int Count() { if(val < target) return val++;
                                                    else return target; }   }   }
```

- Here ***Target*** is a ***read-only property***. To follow along, put this **"another part"** into a file called **Counter2.cs**.

Following demonstrates the ***additive effect*** of ***namespaces*** by using both ***CountDown*** and ***CountUp***:

```
using System;
using Counter;      // Bring the entire Counter namespace into view: With both part.
class NSDemo { static void Main() {  CountDown cd = new CountDown(10);
                                     CountUp cu = new CountUp(8);
                                     int i;
    do { i = cd.Count(); Console.Write(i + " "); } while(i > 0);  Console.WriteLine();
    do { i = cu.Count(); Console.Write(i + " "); } while(i < cu.Target);  }}
```

- To ***compile*** this program, you must include the **NSDemo**'s code and **both files** that contain the **Counter namespace**.
- Assuming that you called the **NSDemo**'s code ***NSDemo.cs*** and that the **Counter namespace** files are called ***Counter.cs*** and ***Counter2.cs***, you can use this command line to compile the program:
  `csc NSDemo.cs Counter.cs Counter2.cs`

- The directive **using Counter;** brings into **view** the **entire contents** of the **Counter namespace**. Thus, both **CountDown** and **CountUp** can be referred to directly, without *namespace qualification*. It doesn't matter that the Counter namespace was **split** into **two parts**.

Namespaces Can Be Nested: One namespace can be nested within another. When referring to a nested namespace from code outside the nested namespaces, both the outer namespace and the inner namespace must be specified, with the two separated by a period. To understand the process, consider this program:

```
namespace NS1{          class ClassA{    public ClassA(){ Console.WriteLine("constructing ClassA"); }   }
/* a nested namespace */          namespace NS2 {      class ClassB {   public ClassB(){ Console.WriteLine("constructing ClassB"); }   }       }          }
class NestedNSDemo { static void Main() {       NS1.ClassA a= new NS1.ClassA();
                //                              NS2.ClassB b = new NS2.ClassB();    // Error!!! NS2 is not in view
                                                NS1.NS2.ClassB b = new NS1.NS2.ClassB();   /* this is right */         }}
```

- In the program, the *namespace* **NS2** is nested within **NS1**. Thus, to refer to **ClassB** from code that is outside both **NS1** and **NS2**, you must qualify it with both the **NS1** and **NS2** names. **NS2** by itself is **insufficient**.
- To refer to ***ClassB*** within ***Main( )***, you must use ***NS1.NS2.ClassB*** since, namespace names are separated by a ***period***.
- ***Other way:*** You can specify a **nested namespace** using a **single namespace statement** by separating each namespace with a **period**. For example:
  `namespace OuterNS { namespace InnerNS { /*...*/ }}` Can also be specified like: ***`namespace OuterNS.InnerNS { /*...*/ }`***

The Global Namespace: If you don't declare a namespace for your program, the default **global namespace** is used.