CSC-200

Module 3

Advanced Object-Oriented Programming with C#

1 Extending Classes

1.1 Extending a Class

One of the most important features that object-oriented programming provides is the ability to extend classes. There are many reasons why we want to extend a class. One reason is that we want to create a new version of a class but we do not wish to have to rewrite all the code. By extending the class, we can inherit all the existing data and code but we can also add in new data and code or replace inherited code. You do not need the source code to extend an existing class thus you can inherit code written by other developers. You can use the extend operator (**:**) to extend a class. The original class is called as the base class and the new class is called as the derived class. In the following example, we demonstrate how we can inherit code by extending a class but we can also add new code or replace inherited code. When you wish to replace an inherited method with a new method, you should use the **new** keyword to avoid a compiler warning. Not all classes are extendable as the **sealed** keyword can be used to prohibit a class to be extended.

Extending a class: Extend1

```
class ClassA {
      public void Proc1() { Console.WriteLine("ClassA.Proc1()"); }
      public void Proc2() { Console.WriteLine("ClassA.Proc2()"); }
}
class ClassB : ClassA {
      public new void Proc2() { Console.WriteLine("ClassB.Proc2()"); }
      public void Proc3() { Console.WriteLine("ClassB.Proc3()"); }
}
sealed class ClassC : ClassB {
      public void Proc4() { Console.WriteLine("ClassC.Proc4()"); }
}
```

Instancing and using objects

```
static void Main() {
      ClassA obj1 = new ClassA();
      ClassB obj2 = new ClassB();
      ClassC obj3 = new ClassC();
      obj1.Proc1(); obj1.Proc2();
      obj2.Proc1(); obj2.Proc2(); obj2.Proc3();
      obj3.Proc1(); obj3.Proc2(); obj3.Proc3(); obj3.Proc4();
}
```

Replacing inherited methods hides the inherited method from being accessed by external code directly. There is still a way that the original method can still be called internally and externally. If you use **new** keyword to replace inherited code, you are only shadowing the original method. Rather than replacing the entire method, you may wish to extend the method by also running the original code. You can call a shadowed member from within the class by using the **base** keyword.

Extending a method

```
class ClassB : ClassA {
      public new void Proc2() {
            base.Proc2();    // calling ClassA.Proc2();
            Console.WriteLine("ClassB.Proc2()");
      }
      public void Proc3() { Console.WriteLine("ClassB.Proc3()"); }
}
class Program {
      static void Main() {
            ClassB obj = new ClassB();
            obj.Proc2();     // call extended method
      }
```

It is also possible for a developer to bypass the new method and call back the original method even though this is not recommended since there must be a reason why the original method was replaced. Derived classes is compatible to the base class since the derived class has inherited all the members of the base class. An object that is created from a derived class can then be cast to the base class.

Calling base method externally

```
class Program {
      static void Main() {
            ClassB obj = new ClassB();
            obj.Proc2();     // call extended method
            (ClassA)obj.Proc2();  // call shadowed method
      }
```

1.2 Overriding Methods

Shadowing does not stop external code from calling back the original method as we have demonstrated above. There is another way to replace an inherited method that we call as overriding. This ensures that the right method will always be called base on the actual object and not the type of field or variable. However you cannot simply just override any inherited method, the original class has to allow you to override it using a **virtual** keyword.

Overriding virtual methods

```
using System;

class ClassA {
      public void Proc1() { Console.WriteLine("ClassA.Proc1()"); }
      public virtual void Proc2() {
            Console.WriteLine("ClassA.Proc2()"); }
}
class ClassB : ClassA {
      public override void Proc2() {
            base.Proc2();    // call ClassA.Proc2()
            Console.WriteLine("ClassB.Proc2()");
      }
      public void Proc3() { Console.WriteLine("ClassB.Proc3()"); }
}
```

You need to use the **override** keyword rather than the **new** keyword when you need to override the virtual method rather than shadow it. This guarantees that if an object is created from the derived class, the overriding method will always be called even if you cast the object to the base class.

You can force overriding using an **abstract** method. An abstract method does not contain any code and thus you cannot inherit any code from an abstract method. The class that contains any abstract method must also be marked as abstract. You cannot create an object from an abstract class since it is not complete. If the method has not been overridden by a derived class then the derived class will also be abstract.

Abstract class and method:

```
abstract class ClassA {
      public abstract void Proc3();
}
            :
class Program {
      static void Main() {
      //    ClassA obj1 = new ClassA();// this will not compile
            ClassA obj2 = new ClassB();
            obj2.Proc3();
      }
}
```

1.3 Class Compatibility

An advantage of extending a class is that objects created from derived classes will be compatible to the base class. Thus any methods that work with the base class would work with the derived classes. In the following example, we declare the variables as **ClassA** but we can still assign objects from other compatible classes. It is possible to any object for class compatibility using **is** operator.

Derived classes compatible to base class

```
ClassA obj1 = new ClassB();
ClassA obj2 = new ClassC();
ClassB obj3 = new ClassC();
obj1.Proc2();
obj2.Proc2();
obj3.Proc2();

object obj4 = new ClassC();
Console.WriteLine(obj4 is ClassC);
Console.WriteLine(obj4 is ClassB);
Console.WriteLine(obj4 is ClassA);
```

As mentioned above any method that is implemented for base classes will work also for all the base classes. In the following example, we implemented a method for the base class but will be eventually use for the derived classes.

Method implemented for base class

```
static void UseObject(ClassA obj) {
      obj.Proc1(); obj.Proc2();
}
```

Calling method with derived class objects

```
UseObject(new ClassB());
UseObject(new ClassC());
```

Using class compatibility, we can write methods that work with different types of objects as long as they are compatible. To be compatible, the classes have to derive from the same base classes. However, there is also another way to get compatibility between objects that are not related by using interfaces.

1.4 Interface Compatibility

Even though objects created from different classes are not related, they still may implement the same interface. Thus any method designed for the interface will then work for all the objects. In the following example, we implemented a method that can be used to make a copy of any object.

In the following code, we implemented a method that can accept any object as long as it implements **ICloneable** interface. Classes implemented the interface will definitely have a **Clone** method. Thus we can call this method with any object where its class has implemented the same interface.

Using an object thru an interface

```
static object Copy(ICloneable obj) {
      return obj.Clone();
}
```

Calling method with different object

```
object o1 = Copy("ABCDEF");      // string class implements ICloneable
object o2 = Copy(new ArrayList());    // ArrayList implements ICloneable
```

However if you pass in any object that does not support the interface, you may not be able to compile the program or a runtime error will occur when the objects are cast to the interface. You can make a more flexible method by allowing all objects and then determine during runtime whether the object supports the interface. If it supports we can then cast the object to the interface to call the methods.

Checking interface compatibility

```
static object Copy(object obj) {
      if (obj is ICloneable) {
            ICloneable item = (ICloneable)obj;
            return item.Clone();
      }
      return null;
}
```

You can perform type-checking and casting at the same time using **as** operator. If an object supports the interface, the reference to the object is returned otherwise **null** is returned instead. We can then test the reference before using the interface.

Checking and using interface

```
static object Copy(object obj) {
      ICloneable item = obj as ICloneable;
      return item == null ? null : item.Clone();
}
```

1.5 Protected Members

Even though you will inherit all data and code when you extend a class, this does not mean that you can access them. When fields, properties and methods are marked as **private**, they can only be accessed from within the same class. External and derived classes will not be able to access them. If you are building a class to be extended you can mark members with the **protected** modifier instead. Protected members can only be accessed from within the same classes or from derived classes.

Class with private and protected fields: Extend2

```
class ClassA {
      private int v1;
      protected int v2;
      public int Value1 { get { return v1; } }
      public int Value2 { get { return v2; } }
      public ClassA() { v1 = 1; v2 = 2; }
}
```

Accessing base and extended fields

```
class ClassB : ClassA {
      private int v3;
      public ClassB() { v3 = 3; }
      public void ShowValues() {
            Console.WriteLine(Value1); // cannot access directly
            Console.WriteLine(v2);     // can access protected member
            Console.WriteLine(v3);     // can access same class member
      }
}

class Program {
      static void Main() {
            ClassB obj = new ClassB();
            obj.ShowValues();
      }
}
```

1.6 Extending Constructors

Constructors are always extended from base constructors. That means that when you call a constructor in a derived class, it also call the base constructor. This will be proven by the following code. When you create an object from the derived class, not only is the constructor for the derived class called but also for the base class.

Constructors are always extended

```
public ClassA() {
      v1 = 1; v2 = 2; Console.WriteLine("ClassA() called!");
}
      :
public ClassB() {
      v3 = 3; Console.WriteLine("ClassB() called!");
}
```

It is possible that the base class may have more than one constructor but the default constructor will be the parameter-less constructor. Regardless of which constructor will be called in the derived class, the default constructor will always be called in the base class.

Multiple constructors in base and derived classes

```
public ClassA(int v) {
      v1 = v2 = v; Console.WriteLine("ClassA(int) called!");
}
      :
public ClassB(int v) {
      v3 = v; Console.WriteLine("ClassB(int) called!");
}
```

Default base constructor will be called

```
ClassB obj =
      new ClassB(123);      // ClassB(int) is called in derived class
obj.ShowValues();           // ClassA() is called in base class
```

You can choose to extend from different base constructor using extend operator in the derived constructor. The following example, we added a new constructor to the derived class that does not call the parameterless base constructor. To extend the base constructor, use the **base** keyword. You can alternatively use **this** keyword that will extend from another constructor in the same class.

Extending from a different base constructor

```
public ClassB(int a, int b) : base(a) {
      v3 = b; Console.WriteLine("ClassB(int,int) called!");
}
      :
static void Main() {
      ClassB obj =
            new ClassB(123,456);  // ClassB(int,int) called
      obj.ShowValues();           // ClassA(int) called in base class
}
```

2 Generics

2.1 Generic Methods

When implementing application framework libraries, your code will usually have to work with different types of objects. Even though you can use reflection as is shown in the previous chapter, you have to write complex and inefficient code regardless of whether the types are predefined or not. In the following example is a method implemented to swap the contents of two **int** variables.

Method to swap int variables: Generics1

```
static void Swap(ref int v1, ref int v2) {
      int vt = v1;
      v1 = v2;
      v2 = vt;
}

static void Main() {
      int n1 = 66;
      int n2 = 99;
      Swap(ref n1, ref n2);
      Console.WriteLine("{0},{1}", n1, n2);
}
```

The above method works fine but can only work for **int** variables. To swap other types of variables, we have to overload the method to support other types. This will force you to have one **Swap** method for each different type. This is clearly not efficient. One way to solve this is to use the **object** type. This type can be assigned a reference to any object or instantiated to store any value.

Method to swap object variables

```
static void Swap(ref object v1, ref object v2) {
      object vt = v1; v1 = v2; v2 = vt;
}
```

However since values are not objects, when you assign a value as an object, an object has to be physically instantiated to store the value. This is called *boxing*. To access the value back from the object, you need to use type-casting. It tells the compiler to generate code for extracting the value inside the object. This is called *unboxing*. Even if you are assigning an object, you still need to type-cast to access it back from an **object** variable which incurs a type-check operation as well.

Boxing, unboxing and type-checking

```
static void Main() {
      object o1 = 66; // boxing
      object o2 = 99; // boxing
      Swap(ref o1, ref o2);
      int n1 = (int)o1; // unboxing
      int n2 = (int)o2; // unboxing
      Console.WriteLine("{0},{1}", n1, n2);
      o1 = "Hello!";
      o2 = "Goodbye";
      Swap(ref o1, ref o2);
      string s1 = (string)o1; // type-check
      string s2 = (string)o2; // type-check
      Console.WriteLine("{0},{1}", s1, s2);
}
```

This problem can be solved by using generics which has been available since C# 2.0 to allow us to create generic types and methods that will be compiled as a template and then use to instantiate code when the actual types are known. Put one or more generic type parameters after the method name in angle brackets to create a generic method. The generic types can be used anywhere inside the code. The actual type can be supplied when the method is called or may also be detected by the compiler. C# supports type inference where it can detect types automatically from initializers or parameters. As shown in the following example below, no boxing, unboxing and type-casting operations are required for generic methods. Generic methods can be called across assemblies so you can re-use them across multiple applications if you compile them into a separate library assembly.

Generic method with a single generic type parameter

```
static void Swap<T>(ref T v1, ref T v2) {
      T vt = v1; v1 = v2;   v2 = vt;
}
```

Actual types identified during method call

```
static void Main() {
      int n1 = 66;
      int n2 = 99;
      Swap<int>(ref n1, ref n2); // Swap(ref n1, ref n2);
      Console.WriteLine("{0},{1}", n1, n2);
      string s1 = "Hello!";
      string s2 = "Goodbye!";
      Swap<string>(ref s1, ref s2);    // Swap(ref s1, ref s2);
      Console.WriteLine("{0},{1}", s1, s2);
}
```

You can have multiple generic type parameters for a single method if you need to use more than one generic type in the code. Following demonstrates the use of two generic type parameters. It is possible that both generic types reference the same type as well.

Supporting multiple generic types: Generics2

```
static void Proc1<A, B>(A v1, B v2) {
      Console.WriteLine("A={0},B={1}",
            v1.GetType().FullName, v2.GetType().FullName);
}

static void Main() {
      Proc1('A', 66);                  // Proc1<char,int>(…);
      Proc1(9.1, 199.99f);             // Proc1<double,float>(…);
      Proc1("Hello!", 9999.99m);       // Proc1<string,decimal>(…);
      Proc1(true, false);              // Proc1<bool,bool>(…);
}
```

2.2 Generic Type Constraints

You can put restrictions on what actual types can replace the generic types by assigning constraints to the generic type using the **where** keyword. It the type must be a value-type, you can apply the **struct** constraint. Use **class** instead to enforce the use of a reference-type. You may also apply type compatibility as a constraint. Applying type compatibility constraint will also allow you to perform more operations in the generic method. To be able to instantiate an object from a generic type in the code, you need to apply the parameter-less constructor constraint using the **new** keyword. If you have multiple constraints, this must be the last constraint.

Value-type & reference-type constraints

```
static void Proc2<T>(T v1) where T : struct {
      Console.WriteLine("{0} is a value!", v1);
}
static void Proc3<T>(T v1) where T : class {
      Console.WriteLine("{0} is an object!", v1);
}
```

Generic and non-generic type compatibility constraints

```
static int Proc4<T>(T v1, T v2) where T : IComparable {
      return v1.CompareTo(v2);
}
static int Proc5<T>(T v1, T v2) where T : IComparable<T> {
      return v1.CompareTo(v2);
}
```

Applying multiple type constraints

```
static T Proc6<T>(T v1) where T : ICloneable, IDisposable {
      T copy = (T)v1.Clone();
      v1.Dispose(); return copy;
}
```

Instancing an object from a generic type

```
static T Proc7<T>() where T : new() { return new T(); }
```

Only parameter-less constructors can be called in a generic method so custom constructors are not supported. Instead of instancing an object, you may only want to initialize a variable or field to the default value. You can use a **default** operator to obtain the default value for a generic type. Numerical-types returns 0, **bool** returns **false** and reference-types always return **null**.

Initializing to default value for generic type

```
static void Proc8<T>(ref T v1) { v1 = default(T); }
```

2.3 Generic Types

You can also implement generic types where the generic type parameters will be placed on a type rather than on a method. The type parameters can then be used anywhere in the type including fields, properties, constructors and also methods. In the following example, a **Valuables** generic class is implemented to be able to store a value of any type. The actual type has to be specified when the class is used and also when an object is instantiated. Type constraints can be placed on the type by using the **where** keyword.

Implementing a generic class: Generics3

```
public class Valueable<T> where T : struct {
      private T _value;
      public T Value {
            get { return _value; }
            set { if (!_value.Equals(value))
                        _value = value; }
      }
      public Valueable() { }
      public Valueable(T value) {
            _value = value;
      }
      public void Reset() {
            _value = default(T);
      }
}
```

Instancing and using generic classes

```
static void Main() {
      Valueable<int> v1 = new Valueable<int>(66);
      Valueable<double> v2 = new Valueable<double>(99.1);
      Valueable<bool> v3 = new Valueable<bool>(true);
      Console.WriteLine(v1.Value);
      Console.WriteLine(v2.Value);
      Console.WriteLine(v3.Value);
}
```

2.4 Using Nullable

Only reference types can contain nulls but not values types. We normally used a **null** to pass an optional parameter. If the parameter contains null, the method would probably use a default value. However, we would not be able to use this for value types unless it is boxed and passed as an object parameter. You can now use a generic **Nullable** type to hold an optional value. Use the **HasValue** property to determine whether there is a value. To access the actual value, use the **Value** property. The following shows how to use this generic type. To make this type easier to use, can use a "**?**" operator to indicate that a field, property, parameter, variable is nullable and you can use compare against **null** instead of checking the **HasValue** property. Use type-casting to extract the value that can then be assigned to non-nullable types. Since Nullable is a value-type, boxing and unboxing operations are not performed.

Using Nullable type: Nullable1

```
Nullable<int> n1 = new Nullable<int>(10);
Nullable<int> n2 = new Nullable<int>();
Console.WriteLine(n1.HasValue);
Console.WriteLine(n2.HasValue);
if (n1.HasValue) Console.WriteLine(n1);
if (n2.HasValue) Console.WriteLine(n2);
```

Using operators to simplify usage of Nullable type

```
int? n3 = 10;
int? n4 = null;
Console.WriteLine(n3 != null);
Console.WriteLine(n4 != null);
if (n3 != null) Console.WriteLine(n3);
if (n4 != null) Console.WriteLine(n4);
```

Use type-casting to convert Nullable to non-nullable (exception if null)

```
int n5 = (int)n3;
int n6 = (int)n4;
```

Check for null before type-casting to avoid exceptions

```
int n5 = (n3 != null) ? (int)n3 : 0;
int n6 = (n4 != null) ? (int)n4 : 0;
```

2.5 Extending Generic Types

Classes and interfaces can be extended regardless of whether they are generic or not. The new class or interface can also be generic or non-generic. Following examples show extending a generic and non-generic type from a generic type. **MyList** is still generic as it accepts a generic type parameter while **DecimalList** is closed and no longer generic. You can only store **decimal** values in the later class. Non-generic types can extend generic types and generic types can extend non-generic types.

Extending to a new generic type: Generics3

```
public class MyList<T> : List<T> {
      public void Display() {
            foreach (T value in this)
                  Console.WriteLine(value);
      }
}
```

Extending to a non-generic type

```
public class DecimalList : MyList<decimal> {
      public decimal Sum() {
            decimal total = 0;
            foreach (decimal value in this)
                  total = total + value;
            return total;
      }
}
```

Using extended types

```
MyList<int> l1 = new MyList<int>();
DecimalList l2 = new DecimalList();
l1.Add(100); l1.Add(200); l1.Add(300);
l2.Add(100.1m); l2.Add(200.2m); l2.Add(300.3m);
l1.Display(); l2.Display();
Console.WriteLine(l2.Sum());
```

2.6 Generic Interfaces

An interface allows you to write code that uses different objects that implement the same interface. This makes it easy to create compatibility between different types of objects. However this does not make the objects easier to use through the interface. For example, when objects using **ICloneable** interface the **Clone** method always return the copy as **object**. You would always need to typecast the result to original type even though you know what you are cloning.

Non-generic ICloneable interface

```
namespace System {
      public interface ICloneable {
            object Clone();
      }
}
```

Cloning a string

```
string s1 = (string)("abc".Clone());
```

There is no generic version of this interface but we can always create our own as shown below. The exact type to return can be chosen during implementation of the interface.

Generic ICloneable interface

```
namespace System {
      public interface ICloneable<T> {
            T Clone();
      }
}
```

Implementing a generic interface

```
public class Time : ICloneable, ICloneable<Time> {
            :
      public Time Clone() {
            return new Time(hour, minute, second);
      }
}
```

3 Delegates

3.1 Why Use Delegates?

Delegates are special classes build to contain a reference to method which can then be invoked through the delegate without knowing the name or the location of the method. It does not even matter if the method is static or object, public internal, protected, or private. As long as the reference is stored into a delegate object, you can then use it to call the method. We will commonly use delegates to implement notifications and callbacks.

Following example shows a program running a lengthy operation. Since the operation does not provide any feedback to the main program, the program will not be able to keep track of the progress of the operation. A user running the program would have to wait for it to complete without knowing what the state of the process is in.

Lengthly operation with no feedbacks: Delegates1

```
using System;
using System.Threading;

public class Job {
      public void Run() {
            // simulate 20 seconds of work
            Thread.Sleep(20000);
      }
}

class Program {
      static void Main() {
            Job job1 = new Job();
            job1.Run();
      }
}
```

We can improve the above program by changing the lengthy operation to show the current progress in the **Console** screen. The following example shows how to clear the console window and output to a fixed position on the screen.

Operation with progress report

```
public class Job {
      public void Run() {
            Console.Clear();
            for (int i = 1; i <= 100; i++) {
                  Thread.Sleep(200);
                  Console.SetCursorPosition(0, 0);
                  Console.WriteLine("{0,3}% completed.", i);
            }
      }
}
```

However by building progress reporting directly into the operation, the program can only run the operation locally and also must be a console application. If you run the operation remotely, it would display the progress report on the remote system not where your program is started. Even if you run the operation on the local system, the program must open a console window to get the result. To be able to perform the operation locally and remotely and in any application type, including console, window and web, we have to abstract the progress reporting away from the operation. We can also make progress reporting as optional. This can be done very easily by using delegates.

3.2 Using Delegates

As mentioned earlier we use a delegate object to call a method. Since methods do not always have the same return type or the same number of parameters, we need to declare a delegate type that matches the method we wish to call. In the following example, we can define a new delegate type to call a **void** method that accepts a single **int** parameter.

Declaring a delegate type:

```
public delegate void ProgressHandler(int status);
```

Since a delegate is an object, you need to have a variable or field to assign the object once it has been created. We will now declare a field in the **Job** class that can be used to assign a delegate object using the delegate type.

Declaring the delegate field

```
public class Job {
      public ProgressHandler OnProgressChanged;
                  :
}
```

Since there is no guarantee that the field has been assigned an object we will have to test the field to make sure a delegate object is available before we attempt to use it. To call the method using the delegate object, call the delegate as though it is the method. We do not have to know the name or location of the real method. In fact, the method may be even be implemented on a remote computer system and have to be called across a network.

Checking and using a delegate object

```
public void Run() {
      for (int i = 1; i <= 100; i++) {
            Thread.Sleep(200);
            if (OnProgressChanged != null)
                  OnProgressChanged(i);
      }
}
```

The application can now decide whether it wants to implement progress report or not. If it needs to display a progress report, it can implement a method that can be called using the delegate. The signature of the method must match the signature of the delegate as demonstrated below. However it does not matter if the method is static or not and what access modifier is applied.

Method to receive progress feedback from operation

```
class Program {
      static void ShowProgress1(int status) {
            Console.SetCursorPosition(0, 0);
            Console.WriteLine("{0,3}% completed!", status);
      }
                  :
```

All the application has to do is to create a delegate object to encapsulate a reference to the **ShowProgress1** method and assign it to the delegate field. It can then be used by the operation to call the method.

Creating and assigning the delegate object

```
static void Main() {
      Job job1 = new Job();
      job1.OnProgressChanged = new ProgressHandler(ShowProgress1);
      Console.Clear(); job1.Run();
}
```

In C# 2.0, you can now directly assign the method to the delegate field. The C# compiler would automatically create the delegate object to store the reference to the method. If for some reason this cannot work you can fallback to the old method of creating and assigning delegates.

Creating and assigning the delegate object in C# 2.0

```
static void Main() {
      Job job1 = new Job();
      job1.OnProgressChanged = ShowProgress1;
      Console.Clear(); job1.Run();
}
```

3.3 Multi-Cast Delegates

A delegate can be a single or a multi-cast delegate. A multi-cast delegate can be used to call more than one method. To use a multi-cast delegate replace the assignment operator with plus assignment (+=) operator. You can make use of this operator more than once to assign multiple methods. Following is another method that follows the same delegate signature. We can now use the plus assignment operator to assign multiple delegates to a single field. When the delegate is invoked, all methods will be called in the order of assignment. It is also possible to use (-=) assignment to remove individual delegates.

Another progress report method

```
static void ShowProgress2(int status) {
      Console.SetCursorPosition(0, 1);
      Console.WriteLine("{0,3}% work left!", 100 - status);
}

static void Main() {
      Job job1 = new Job();
      job1.OnProgressChanged += ShowProgress1;
      job1.OnProgressChanged += ShowProgress2;
      Console.Clear(); job1.Run();
}
```

Events are specialized delegates. Events are always multi-cast and the method to be called must always be a **void** method. You must use the plus assignment operator with an event even though you wish to call one method. Event to the delegate is like a property to a field. It generates an **add** and **remove** method to attach and remove event handlers from a delegate. Events can be declared in an interface while a delegate field cannot be accessed through interfaces. Visual Studio also recognizes events which will allow you to create and assign delegate objects visually rather than having to write programming code.

Marking a delegate as an event

```
public class Job {
      public event ProgressHandler OnProgressChanged;
                  :
}
```

3.4 Anonymous Methods

In the previous, you can only called named methods using delegates but in the new version a delegate can be used to execute a block of code in the content of another method. This can be used to reduce the number of methods in a class especially when they contain very little code. The following shows how to assign code to be executed by the event of a control and from another thread.

Using anonymous methods

```
Job job1 = new Job();
job1.OnProgressChanged += delegate(int status) {
      Console.SetCursorPosition(0, 0);
      Console.WriteLine("{0,3}% completed!", status);
};
job1.OnProgressChanged += delegate(int status) {
      Console.SetCursorPosition(0, 1);
      Console.WriteLine("{0,3}% work left!", 100 - status);
};
Console.Clear(); job1.Run();
```

In C# 3.0, calling anonymous methods is simplified using lambda expressions. The following shows simplification of assigning anonymous delegates using the lambda (**=>**) operator. It may even be shorter where brackets and semicolons can be removed if the anonymous method only has one statement. You can see this in later examples.

Using lambda expressions

```
job1.OnProgressChanged += status => {
      Console.SetCursorPosition(0, 0);
      Console.WriteLine("{0,3}% completed!", status);
};
job1.OnProgressChanged += status => {
      Console.SetCursorPosition(0, 1);
      Console.WriteLine("{0,3}% work left!", 100 - status);
};
```

3.5 Generic Delegates

In order to call three different methods where their parameter is of different types, you would normally need three separate delegate class. The following example shows three delegates calling methods that have different signatures. Using generic delegates, we can now define a single delegate to define calls to methods with the same number of parameters but of different types. Following shows how this can be done.

Different delegates for different methods: Delegates2

```
static void Proc1(int value) { }
static void Proc2(string value) { }
static void Proc3(bool value) { }

delegate void IntHandler(int v);
delegate void StringHandler(string v);
delegate void BoolHandler(bool v);

static void Main() {
      IntHandler d1 = Proc1;
      StringHandler d2 = Proc2;
      BoolHandler d3 = Proc3;
      d1(123);
      d2("Hello!");
      d3(true);
}
```

Using one generic delegate for all methods

```
delegate void Handler<T>(T value);

static void Main() {
      Handler<int> d1 = Proc1;
      Handler<string> d2 = Proc2;
      Handler<bool> d3 = Proc3;
                  :
}
```

In C# 3.0, there is no need for you do declare your own delegate classes since there are already a few groups of generic delegates that are already provided. The **Action** delegates are for **void** methods, The **Predicate** delegates for **bool** methods that have exactly one parameter, and **Func** delegates are for methods that return any type with any number of parameters. Following are examples of using these delegates together with lambda expressions.

Using Action delegates

```
Action a1 = () => Console.WriteLine("Hello!");
Action<string> a2 = x => Console.WriteLine(x);
Action<int, int, int> a3 = (x, y, z) => {
      DateTime dt = new DateTime(x, y, z);
      Console.WriteLine(dt);
};
a1();
a2("Goodbye!");
a2("Sayonara!");
a3(2011, 2, 14);
```

Using Func delegates

```
Func<int,int,int,DateTime> f1 = (x,y,z) => new DateTime(x,y,z);
Console.WriteLine(f1(1967,10,6));
Console.WriteLine(f1(2010,2,14));
```

Using Predicate and Action with collections

```
var values = new List<int> { 66, 91, 13, 80, 42, 72, 35 };
Predicate<int> Odd = x => (x % 2) == 1;
Predicate<int> Even = x => (x % 2) == 0;
var list1 = values.FindAll(Odd);
var list2 = values.FindAll(Even);
list1.ForEach(x => Console.WriteLine(x));   // Action<int>
list2.ForEach(x => Console.WriteLine(x));   // Action<int>
values.RemoveAll(Odd);
```