Chapter 18

Interfaces and the .NET Framework

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

Interfaces and the .NET Framework

Objectives

After completing this unit you will be able to:

- **Create and use interfaces in your C# programs.**
- **Use predefined interfaces in the .NET framework.**
- **Use C# collections to eliminate the need for custom coding of most common data structures.**
- **Explain the differences between *reference copy*, *shallow memberwise copy*, and *deep copy*.**

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

Overview

- **In the previous chapter we saw how useful interfaces can be in specifying contracts for our own classes.**
- **Interfaces can help us program at a higher level of abstraction, enabling us to see the essential features of our system without being bogged down in implementation details.**
- **Interfaces are a ubiquitous and important part of the .NET framework.**
- **Many of the standard classes implement specific interfaces, and we can call into the methods of these interfaces to obtain useful services.**
- **Collections are an example of classes in the .NET Framework that support a well-defined set of interfaces that provide useful functionality.**
- **In order to work with collections effectively, you need to override certain methods of the *object* base class.**
- **Besides calling into interfaces that are implemented by library classes, many .NET classes call standard interfaces.**
- **If we provide our own implementation of such interfaces, we can have .NET library code call our own code in appropriate ways, customizing the behavior of library code.**

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

Collections

- **The .NET Framework class library provides an extensive set of classes for working with collections of objects.**
- **These classes are all in the *System.Collections* namespace and implement a number of different kinds of collections, including lists, queues, stacks, arrays, and hashtables.**
  - The collections contain object instances.
  - Since all types derive ultimately from object, any built-in or user-defined type may be stored in a collection.
- **In this section we will look at a representative class in this namespace, *ArrayList*.**
  - We will examine the interfaces implemented by this class and see how to use array lists in our programs.
  - Part of our task in using arrays lists and similar collections is to properly implement our class whose instances are to be stored in the collection.
  - In particular, our class must generally override certain methods of **object**.

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

ArrayList Example

- **To get our bearings, let's begin with a simple example of using the *ArrayList* class.**
- **An array list, as the name suggests, is a list of items stored like an array.**
  - An array list can be dynamically sized and will grow as necessary to accommodate new elements being added.
- **As mentioned, collection classes are made up of instances of type *object*.**
  - We will illustrate creating and manipulating a collection of **string**.
  - We could also just as easily create a collection of any other built-in or user-defined type.
  - If our type were a value type, such as **int**, the instance would be boxed before being stored in the collection. When the object is extracted from the collection, it will be unboxed back to **int**.
- **Our example program is *StringList*.**
  - It initializes a list of strings, and then lets the user display the list, add strings, and remove strings.
  - A simple "help" method displays the commands that are available.
- **Please examine and run this program online.**

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

Count and Capacity

- **An array list has properties *Count* and *Capacity*.**
- **The *Count* is the current number of elements in the list, and *Capacity* is the number of available "slots."**
  - If you add a new element when the capacity has been reached, the **Capacity** will be automatically increased.
  - The default starting capacity is 16, but it can be adjusted by passing a starting size to the constructor.
  - The **Capacity** will double when it is necessary to increase it. The "count" command in the sample program displays the current values of **Count** and **Capacity**, and you can observe how these change by adding new elements.

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

foreach Loop

- **The *System.Collections.ArrayList* class implements the *IEnumerable* interface, as we will discuss later in the chapter, which means that you can use a *foreach* loop to iterate through it.**

```
private static void ShowList(ArrayList array)
{
   foreach (string str in array)
   {
      Console.WriteLine(str);
   }
}
```


Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

Array Notation

- ***ArrayList* implements the *IList* interface, which has the property *Item*.**
  - In C# this property is an indexer, so you can use array notation to access elements of an array list.
  - The "array" command demonstrates accessing the elements of the list using an index.

```
private static void ShowArray(ArrayList array)
{
   for (int i = 0; i < array.Count; i++)
   {
      Console.WriteLine("array[{0}] = {1}",
         i, array[i]);
   }
}
```


Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

Adding to the List

- **The *Add* method allows you to append an item to an array list.**
  - If you want to make sure you do not add a duplicate item, you can make use of the **Contains** method to check whether the proposed new item is already contained in the list.

```
private static void AddString(string str)
{
   if (list.Contains(str))
      throw new Exception("list contains "
         + str);
   list.Add(str);
}
```


Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

Remove Method

- **The *Remove* method allows you to remove an item from an array list.**
  - You can make use of the **Contains** method to check whether the item to be deleted is on the list.

```
private static void RemoveString(string str)
{
   if (list.Contains(str))
      list.Remove(str);
   else
      throw new Exception(str + " not on list");
}
```


Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

RemoveAt Method

- **The *RemoveAt* method allows you to remove an item at a specified integer index.**
  - If the index is out of range, an exception of type **ArgumentOutOfRangeException** will be thrown. (In our program we just let our normal test program exception handling pick up the exception.)

```
private static void RemoveAt(int index)
{
   list.RemoveAt(index);
}
```

  - The index is zero-based.

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

Collection Interfaces

- **The classes *ArrayList*, *Array*, and many other collection classes implement a set of four fundamental interfaces.**

```
public class ArrayList : IList, ICollection,
                         IEnumerable, ICloneable
```

  - In this section we will examine the first three interfaces. We will look at **ICloneable** later in the chapter.

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

IEnumerable and IEnumerator

- **The most basic interface is *IEnumerable*, which has a single method, *GetEnumerator*.**

```
interface IEnumerable
{
   IEnumerator GetEnumerator();
}
```

  - **GetEnumerator** returns an interface reference to **IEnumerator**, which is the interface used for iterating through a collection.
  - This interface has the property **Current** and the methods **MoveNext** and **Reset**.

```
interface IEnumerator
{
   object Current {get;}
   bool MoveNext();
   void Reset();
}
```

  - The enumerator is initially positioned before the first element in the collection and it must be advanced before it is used.

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

IEnumerable and IEnumerator Demo:

AccountList

- **The program *AccountList\Step0*, which we will discuss in detail later, illustrates using an enumerator to iterate through a list.**

```
private static void ShowEnum(ArrayList array)
{
   IEnumerator iter = array.GetEnumerator();
   bool more = iter.MoveNext();
   while (more)
   {
      Account acc = (Account) iter.Current;
      Console.WriteLine(acc.Info);
      more = iter.MoveNext();
   }
}
```

- **This pattern of using an enumerator to iterate through a list is so common that C# provides a special kind of loop, *foreach*, that can be used for iterating through the elements of any collection.**

- **Here is the comparable code using *foreach*.**

```
private static void ShowAccounts(ArrayList array)
   {
      foreach (Account acc in array)
      {
         Console.WriteLine(acc.Info);
      }
   }
```


Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

ICollection

- **The *ICollection* interface is derived from *IEnumerable* and adds a *Count* property and a *CopyTo* method.**

```
interface ICollection : IEnumerable
{
   int Count {get;}
   bool IsSynchronized {get;}
   object SyncRoot {get;}
   void CopyTo(Array array, int index);
}
```

- **There are also synchronization properties that can help you deal with thread safety issues.**
  - Threading is beyond the scope of this course. It is touched upon in the books *Introduction to C# Using .NET* and *Application Development Using C# and .NET*. It is also covered in Object Innovations course 412, .NET Framework Using C#.

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

IList

- **The *IList* interface is derived from *ICollection* and provides methods for adding an item to a list, removing an item, and so on.**
  - There is an indexer provided that enables array notation to be used.

```
interface IList : ICollection
{
   bool IsReadOnly {get;}
   bool IsReadOnly {get;}
   object this[int index] {get; set;}
   int Add(object value);
   void Clear();
   bool Contains(object value);
   int IndexOf(object value);
   void Insert(int index, object value);
   void Remove(object value);
   void RemoveAt(int index);
}
```

  - Our sample code illustrated using the indexer and the **Add**, **Contains**, **Remove**, and **RemoveAt** methods.

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

A Collection of User-Defined Objects

- **We will now look at an example of a collection of user-defined objects.**
  - The mechanics of calling the various collection properties and methods is very straightforward and is essentially identical to the usage for collections of built-in types.
  - What is different is that in your class you must override at least the **Equals** method (of the class Object) of the class you wish to use in a collection in order to obtain proper behavior in your collection.
  - For built-in types, you did not have to worry about this issue, because **Equals** is provided by the class library for you.
- **Our example program is *AccountList*, which comes in two steps. Step 0 illustrates a very simple *Account* class, with no methods of *object* overridden.**
- **The test program *AccountList.cs* contains code to initialize an array list of Account objects, show the initial accounts, and then perform a command loop.**
  - A simple help method gives a brief summary of the available commands:

```
The following commands are available:
        show     -- show all accounts
        enum     -- enumerate all accounts
        add      -- add an account (specify id)
```


Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

A Correction to AccountList (Step 1)

- **You can examine the code online.**
- **The salient point is that the "add" command is not protected against adding a duplicate element ("Bob").**
  - Our code is similar to what we used before in the **StringList** program, but now the **Contains** method does not work properly. The default implementation of **Equals** in the object root class is to check for reference equality, and the two "Bob" elements have the same data but different references.
- ***AccountList\Step1*** **contains corrected code for the** ***Account*** **class.**
  - In the test program we have code for both "add" and "remove," and everything behaves properly.
  - Our test for equality involves just the account ID.
  - For example, two people with the same name could have an account at the same bank, but their account IDs should be different.
  - Notice how easy it is to remove an element from an array list. Just construct an element that will test out "equal" to the element to be removed, and call the **Remove** method.

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

Bank Case Study: Step 7

- **It is now very easy to implement Step 7 of the bank case study, where we use an array list in place of an array to store the accounts.**
  - The basic idea is illustrated previously, only now we have the full blown account class hierarchy. We will briefly examine the three classes where there is change to our code.
  - **Bank**. We change **accounts** to be a reference to **ArrayList** in place of an array. We also define an interface **IBank**, and we implement a new method, **GetStatements**, which returns a report (in the form of a list of strings) showing monthly statements for all accounts in the bank. The **DeleteAccount** method now has a simpler implementation.
  - **Account**. We need to provide an override of the **Equals** method.
  - **TestBank**. A new command "month" exercises the **GetStatements** method of the **Bank** class.
- **As usual, the code can be found in the *CaseStudy* directory for the chapter.**
- **Please examine this code online.**

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

Copy Semantics And ICloneable

- **Many times in programming you have occasion to make a copy of a variable.**
  - When you program in C#, it is very important that you have a firm understanding of exactly what happens when you copy various kinds of data.
- **In this section we will look carefully at the copy semantics of C#.**
- **We will compare reference copy, shallow memberwise copy, and deep copy.**
  - The **ICloneable** interface enables *deep copy*.

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

Copy Semantics in C#

- **Recall that C# has value types and reference types. A value type contains all its own data, while a reference type refers to data stored somewhere else.**
  - If a reference variable gets copied to another reference variable, both will refer to the same object.
  - If the object referenced by the second variable is changed, the first variable will also reflect the new value.
- **As an example, consider what happens when you copy an array, which is a reference type. Consider the program *ArrayCopy*.**
  - When we make the assignment arr2 = arr1, we wind up not with two independent arrays, but rather two references to the same array.
  - When we make a change to an element of the first array, both arrays will wind up changed.

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

Shallow Copy and Deep Copy

- **A struct in C# automatically implements a "memberwise" copy, sometimes known as a "shallow copy."**
- **The *object* root class has a protected method, *MemberwiseClone*, which will perform a memberwise copy of members of a class.**
  - If one or more members of a class are of a reference type, this memberwise copy may not be good enough.
  - The result will be two references to the same data, not two independent copies of the data.
- **To actually copy the data itself and not merely the references, you will need to perform a "deep copy."**
  - Deep copy can be provided at either the language level or the library level.
  - In C++ deep copy is provided at the language level through a copy constructor.
- **In C# deep copy is provided by the .NET Framework through a special interface, *ICloneable*, which you can implement in your classes in order to enable them to perform deep copy.**

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

Example Program

- **We will illustrate all these ideas in the program *CopyDemo*.**
- **This program makes a copy of a Course.**
  - The Course class consists of a title and a collection of students.
  - The test program constructs a **Course** instance **c1** and then makes a copy **c2** by various methods.

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

Reference Copy

- **The first way the copy is performed is by the straight assignment *c2= c1*.**
  - Now we get two references to the same object, and if we make any change through the first reference, we will see the same change through the second reference.
  - The first part of the test program illustrates such an assignment.
  - We initialize with the title “Intro to C#” and two students.
  - We make the assignment **c2 = c1**, and then change the title and add another student for **c2**. We then show both **c1** and **c2**, and we see that both reflect both of these changes.

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

Memberwise Clone

- **The next way we will illustrate doing a copy is a memberwise copy, which can be accomplished using the *MemberwiseClone* method of *object*.**
  - Since this method is protected, we cannot call it directly from outside our **Course** class.
  - Instead, in **Course** we define a method, **ShallowCopy**, which is implemented using **MemberwiseClone**.
  - In the second part of the test program the **ShallowCopy** method is called. Again we change the title and a student in the second copy.
  - In the output of this second part of the program, the **Title** field has its own independent copy, but the **Roster** collection is just copied by reference, so each copy refers to the same collection of students.

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

Using ICloneable

- **The final version of copy relies on the fact that our *Course* class supports the *ICloneable* interface and implements the *Clone* method.**
  - To clone the **Roster** collection we use the fact that **ArrayList** implements the **ICloneable** interface, as discussed earlier in the chapter.
  - Note that the **Clone** method returns an object, so we must cast to **ArrayList** before assigning to the **Roster** field.
  - The third part of the test program calls the **Clone** method. Again we change the title and a student in the second copy.
  - In the output from the third part of the program we have completely independent instances of **Course**. Each has its own title and set of students.

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

Comparing Objects

- **We have quite exhaustively studied issues involved in *copying* objects.**
- **We will now examine the issues involved in *comparing* objects.**
- **In order to compare objects, the .NET Framework uses the interface *IComparable*.**
  - In this section we will examine the use of the interface **IComparable** through an example of sorting an array.

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

Sorting an Array

- **The *System.Array* class provides a static method, *Sort*, that can be used for sorting an array.**
- **The program *ArrayName\Step0* illustrates an attempt to apply this *Sort* method to an array of *Name* objects, where the *Name* class simply encapsulates a *string* through a read-only property *Text*.**

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

Anatomy of Array.Sort

- **What do you suppose will happen when you run this program? Here is the result:**

```
Exception occurred: System.ArgumentException: At
least one
object must implement IComparable.
```

- **The static method *Sort* of the *Array* class relies on some functionality of the objects in the array. The array objects must implement *IComparable*.**
- **Suppose we don't know whether the objects in our array support *IComparable*. Is there a way we can find out programmatically at runtime?**

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

Using the is Operator

- **There are in fact three ways we have seen so far to dynamically check if an interface is supported:**
  - Use exceptions.
  - Use the **as** operator.
  - Use the **is** operator.
- **In this case the most direct solution is to use the *is* operator (which is applied to an object, not to a class). See *ArrayName\Step1*.**
  - Here is the output from running the program. We're still not sorting the array, but at least we fail more gracefully.

```
Name does not implement IComparable
```


Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

The Use of Dynamic Type Checking

- **We can use dynamic type checking of object references to make our programs more robust.**
  - We can degrade gracefully rather than fail completely.
- **For example, in our array program the desired outcome is to print the array elements in sorted order.**
  - We could check whether the objects in the array support **IComparable**, and if not, we could go ahead and print out the array elements in unsorted order, obtaining at least some functionality.

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

Implementing IComparable

- **Consulting the documentation for *System*, we find the following specification for *IComparable*:**

```
public interface IComparable
{
    int CompareTo(object object);
}
```

- **We will implement *IComparable* in the class *Name*. See *ArrayName\Step2*. We also add a simple loop in *Main* to display the array elements after sorting.**

- **If we run the above program, we do not exactly get the desired output:**

```
Name
Name
Name
Name
Name
```

  – The first five lines of output are blank, and in place of the string in **Name**, we get the class name **Name** displayed.

  – The unassigned elements of the array are **null**, and they compare successfully with real elements, always being less than a real element.

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

Complete Solution

- **We should test for *null* before displaying.**
- **The most straightforward way to correct the issue of the strings in *Name* not displaying is to use the *Text* property.**
- **A more interesting solution is to override the *ToString* method in our *Name* class.**
- **The complete solution is in the directory *ArrayName\Step3*.**

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

Understanding Frameworks

- **Our example offers some insight into the workings of frameworks.**
  - A framework is **more** than a library.
  - In a typical library, you are concerned with your code calling library functions.
- **In a framework, you call into the framework and *the framework calls you*.**
- **Your program can be viewed as the middle layer of a sandwich.**
  - Your code calls the bottom layer.
  - The top layer calls your code.
- **The .NET Framework is an excellent example of such an architecture.**
  - There is rich functionality that you can call directly.
  - There are many interfaces, which you can optionally implement to make your program behave appropriately when called by the framework.

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved

Summary

- **Collections are an example of classes in the .NET Framework that support a well-defined set of interfaces that provide useful functionality.**
- **Collections support the interfaces *IEnumerable*, *ICollection*, *IList*, and *ICloneable*.**
- **In order to work with collections effectively, you need to override certain methods of the *object* base class, such as *Equals*.**
- **Comparison of objects are implemented through the *IComparable* interface.**
- **The .NET Framework class library is an excellent example of a rich framework, in which your code can be viewed as the middle layer of a sandwich.**

Copyright © 2003 Object Innovations, Inc.
All Rights Reserved