

**FOR PUBLIC
RELEASE**

E I G H T E E N

Interfaces and the .NET Framework

*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. In this chapter we will examine the role of interfaces in the .NET Framework, where they are ubiquitous. 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. We will provide a new implementation of our case study, using a collection of accounts in place of an array of accounts.*

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. We will look at examples, including object cloning and comparison of objects. This behavior of your program being called into has traditionally been provided by “callback” functions. In C# there is a type-safe, object-oriented kind of callback known as a delegate, a topic we will examine in Chapter 19.

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**.

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:

```
The following commands are available:
    show      -- show all strings
    array     -- show strings via array loop
    add       -- add a string
    remove    -- remove a string
    removeat  -- remove a string at index
    count     -- show count and capacity
    quit      -- exit the program
```

Here is the code of our example program:

```
// StringList.cs

using System;
using System.Collections;
```

```
public class StringList
{
    private static ArrayList list;
    public static void Main()
    {
        // Initialize strings and show starting state
        list = new ArrayList(4);
        ShowCount();
        AddString("Bob");
        AddString("Mary");
        AddString("Charlie");
        ShowList(list);
        ShowCount();
        // Command processing loop
        InputWrapper iw = new InputWrapper();
        string cmd;
        Console.WriteLine("Enter command, quit to exit");
        cmd = iw.getString("> ");
        while (! cmd.Equals("quit"))
        {
            try
            {
                if (cmd.Equals("show"))
                    ShowList(list);
                if (cmd.Equals("array"))
                    ShowArray(list);
                else if (cmd.Equals("add"))
                {
                    string str = iw.getString("string: ");
                    AddString(str);
                }
                else if (cmd.Equals("remove"))
                {
                    string str = iw.getString("string: ");
                    RemoveString(str);
                }
                else if (cmd.Equals("removeat"))
                {
                    int index = iw.getInt("index: ");
                    RemoveAt(index);
                }
                else if (cmd.Equals("count"))
                    ShowCount();
                else
                    help();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
                if (e.InnerException != null)

```

```
        {
            Console.WriteLine(e.InnerException.Message);
        }
    }
    cmd = iw.getString("> ");
}
}
private static void ShowList(ArrayList array)
{
    foreach (string str in array)
    {
        Console.WriteLine(str);
    }
}
private static void ShowArray(ArrayList array)
{
    for (int i = 0; i < array.Count; i++)
    {
        Console.WriteLine("array[{0}] = {1}", i, array[i]);
    }
}
private static void ShowCount()
{
    Console.WriteLine("list.Count = {0}", list.Count);
    Console.WriteLine("list.Capacity = {0}",
        list.Capacity);
}
private static void AddString(string str)
{
    if (list.Contains(str))
        throw new Exception("list contains " + str);
    list.Add(str);
}
private static void RemoveString(string str)
{
    if (list.Contains(str))
        list.Remove(str);
    else
        throw new Exception(str + " not on list");
}
private static void RemoveAt(int index)
{
    list.RemoveAt(index);
}
private static void help()
{
    ...
}
}
```

Here is a sample run of the program:

```
list.Count = 0
list.Capacity = 4
Bob
Mary
Charlie
list.Count = 3
list.Capacity = 4
Enter command, quit to exit
> add
string: David
> add
string: Ellen
> add
string: Bob
list contains Bob
> count
list.Count = 5
list.Capacity = 8
> array
array[0] = Bob
array[1] = Mary
array[2] = Charlie
array[3] = David
array[4] = Ellen
> remove
string: Charlie
> array
array[0] = Bob
array[1] = Mary
array[2] = David
array[3] = Ellen
> removeat
index: 2
> array
array[0] = Bob
array[1] = Mary
array[2] = Ellen
> remove
string: David
David not on list
> removeat
index: 3
Index was out of range. Must be non-negative and less than
size. Parameter name: index
```

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.

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);
    }
}
```

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]);
    }
}
```

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);
}
```

REMOVE METHOD

The **Remove** method allows you to remove an item from an array list. Again 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");
}
```

REMOVEAT METHOD

The **RemoveAt** method allows you to remove an item at a specified integer index. If the item is not found, 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);
}
```

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.

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. 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);
    }
}
```

ICOLLECTION

The **ICollection** interface is derived from **IEnumerable** and adds a **Count** property and a **CopyTo** method. There are also synchronization properties that can help you deal with thread safety issues, a topic we will touch on in Chapter 20.

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

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
{
    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.

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 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.

```
// Account.cs - Step 0

using System;

public class Account
{
    private decimal balance;
    private string owner;
    private int id;
    public Account(decimal balance, string owner, int id)
    {
        this.balance = balance;
        this.owner = owner;
        this.id = id;
    }
    public string Info
    {
        get
        {
            return id.ToString().PadRight(4)
                + owner.PadRight(12)
                + string.Format("{0:C}", balance);
        }
    }
}
```

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)
```

The code is very straightforward, so we won't give a listing. You can examine the code online. We gave the implementation of the "enum" command as an example of explicitly using an enumerator. Here is a sample run of the program:

```
accounts.Count = 0
accounts.Capacity = 16
1  Bob          $100.00
2  Mary         $200.00
3  Charlie      $300.00
```

```

accounts.Count = 3
accounts.Capacity = 16
Enter command, quit to exit
> enum
1  Bob          $100.00
2  Mary         $200.00
3  Charlie      $300.00
> add
balance: 100
owner: Bob
id: 1
> show
1  Bob          $100.00
2  Mary         $200.00
3  Charlie      $300.00
1  Bob          $100.00

```

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. Here is the code added to **Account.cs**.

```

// Account.cs - Step 1

using System;

public class Account
{
    ...
    public override bool Equals(object obj)
    {
        Account acc = (Account) obj;
        return (acc.id == this.id);
    }
}

```

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.

Here is the code for implementing the “add” and “remove” commands.

```

private static void AddAccount(decimal bal, string owner,
                               int id)
{

```

```

// low-level method that lets user specify the id
Account acc = new Account(bal, owner, id);
if (accounts.Contains(acc))
{
    Console.WriteLine("Account with id {0}", id);
    Console.WriteLine(
        "is already contained in the collection");
    return;
}
accounts.Add(acc);
}
private static void RemoveAccount(int id)
{
    Account acc = new Account(0, "", id);
    if (accounts.Contains(acc))
        accounts.Remove(acc);
    else
        throw new Exception("Account " + id + " not on list");
}

```

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.

Here is a sample run illustrating that “add” and “remove” function correctly. Note that the error message for an illegal “remove” comes from normal exception handling in the test program.

```

accounts.Count = 0
accounts.Capacity = 16
1   Bob           $100.00
2   Mary          $200.00
3   Charlie       $300.00
accounts.Count = 3
accounts.Capacity = 16
Enter command, quit to exit
> add
balance: 5
owner: Bobby
id: 1
Account with id 1
is already contained in the collection
> add
balance: 5
owner: Bobby
id: 7
> enum
1   Bob           $100.00
2   Mary          $200.00
3   Charlie       $300.00
7   Bobby         $5.00

```

```
> remove
id: 1
> show
2   Mary           $200.00
3   Charlie        $300.00
7   Bobby          $5.00
> remove
id: 1
Account 1 not on list
```

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.

Bank

```
// Bank.cs - Step 7

using System;
using System.Collections;

public enum AccountType
{
    Checking,
    Savings,
    Invalid
}

interface IBank
```

```
{
    int AddAccount(AccountType type, decimal bal,
                  string owner);
    ArrayList GetAccounts();
    void DeleteAccount(int id);
    Account FindAccount(int id);
    ArrayList GetStatements();
}

public class Bank : IBank
{
    private ArrayList accounts;
    private int nextid = 1;
    public Bank()
    {
        accounts = new ArrayList();
        AddAccount(AccountType.Checking, 100, "Bob");
        AddAccount(AccountType.Savings, 200, "Mary");
        AddAccount(AccountType.Checking, 300, "Charlie");
    }
    public int AddAccount(AccountType type, decimal bal,
                        string owner)
    {
        Account acc;
        int id = nextid++;
        switch(type)
        {
            case AccountType.Checking:
                acc = new CheckingAccount(bal, owner, id);
                break;
            case AccountType.Savings:
                acc = new SavingsAccount(bal, owner, id);
                break;
            default:
                Console.WriteLine("Unexpected AccountType");
                return -1;
        }
        accounts.Add(acc);
        return id;
    }
    public ArrayList GetAccounts()
    {
        return accounts;
    }
    public void DeleteAccount(int id)
    {
        CheckingAccount acc = new CheckingAccount(0m, "", id);
        if (accounts.Contains(acc))
            accounts.Remove(acc);
        else
```

```

        throw new Exception(
            "Account " + id + " not on list");
public Account FindAccount(int id)
{
    foreach (Account acc in accounts)
    {
        if (acc.Id == id)
            return acc;
    }
    return null;
}
public ArrayList GetStatements()
{
    ArrayList array = new ArrayList(accounts.Count);
    foreach (Account acc in accounts)
    {
        acc.Post();
        string str = acc.GetStatement();
        acc.MonthEnd();
        array.Add(str);
        str = "-----";
        array.Add(str);
    }
    return array;
}
}

```

As discussed in the introduction to this section, we replace the array of accounts by an array list. This change simplifies the code, notably in the **DeleteAccount** method. We don't need the **FindIndex** helper method any longer, and we don't have to code moving elements around in the array. One little nuance is that when we construct an account object to use for matching when we call **Remove**, we cannot create an **Account** instance, because **Account** is an abstract class. So we just pick one kind of account, **CheckingAccount**.

The new **GetStatements** method uses **foreach** to iterate through all the accounts. We exploit polymorphism on the calls to **Post**, **GetStatement**, and **MonthEnd** to get the proper behavior for each account type. We call **Post** before **GetStatement**, so the balances will be updated to reflect fees and interest. We then call **MonthEnd** to initialize for the next month.

The **GetAccounts** method now returns a copy of the array list itself. The client program can now do what it needs to do, providing more flexibility than our previous approach, which returned strings. For example, a GUI client could provide a totally different user interface. We will introduce GUI programming in Chapter 22.

Account

```
// Account.cs - Step 7

using System;

abstract public class Account
{
    ...
    public override bool Equals(object obj)
    {
        Account acc = (Account) obj;
        return (acc.Id == this.Id);
    }
}
```

TestBank

```
// TestBank.cs - Step 7

using System;
using System.Collections;

public class TestBank
{
    public static void Main()
    {
        Bank bank = new Bank();
        InputWrapper iw = new InputWrapper();
        string cmd;
        Console.WriteLine("Enter command, quit to exit");
        cmd = iw.getString("> ");
        while (! cmd.Equals("quit"))
        {
            ...
            else if (cmd.Equals("show"))
                ShowAccounts (bank.GetAccounts());
            else if (cmd.Equals("account"))
            {
                int id = iw.getInt("account id: ");
                Account acc = bank.FindAccount(id);
                Atm.ProcessAccount(acc);
            }
            else if (cmd.Equals("month"))
                ShowStringList (bank.GetStatements());
            else
                help();
            cmd = iw.getString("> ");
        }
    }
}
```

```
}
private static void ShowAccounts(ArrayList array)
{
    foreach (Account acc in array)
    {
        string owner = acc.Owner.PadRight(12);
        string stype = acc.Prompt;
        string sid = acc.Id.ToString();
        string sbal = string.Format("{0:C}", acc.Balance);
        string str = owner + "\t" + stype + sid + "\t" +
            sbal;
        Console.WriteLine(str);
    }
}
private static void ShowStringList(ArrayList array)
{
    foreach (string str in array)
        Console.WriteLine(str);
}
...
}
```

Here is a sample run:

```
Enter command, quit to exit
> show
Bob                C: 1    $100.00
Mary               S: 2    $200.00
Charlie            C: 3    $300.00
> account
account id: 1
balance = $100.00
Enter command, quit to exit
C: deposit
amount: 50
balance = $150.00
C: deposit
amount: 50
balance = $200.00
C: withdraw
amount: 95
balance = $105.00
C: show
Statement for Bob id = 1
3 transactions, balance = $105.00, fee = $5.00
C: quit
> show
Bob                C: 1    $105.00
Mary               S: 2    $200.00
Charlie            C: 3    $300.00
> month
```

```

Statement for Bob id = 1
3 transactions, balance = $100.00, fee = $5.00
-----
Statement for Mary id = 2
0 transactions, balance = $201.00, interest = $1.00
-----
Statement for Charlie id = 3
0 transactions, balance = $300.00, fee = $0.00
-----
> show
Bob           C: 1      $100.00
Mary          S: 2      $201.00
Charlie       C: 3      $300.00

```

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. We will see that by implementing the **ICloneable** interface in your class, you can enable deep copy.

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**.

```

// ArrayCopy.cs

using System;

public class ArrayCopy
{
    public static int Main(string[] args)
    {
        int [] arr1 = {1, 4, 9};
        int [] arr2 = arr1;
        show(arr1, "first array");
    }
}

```

```

        show(arr2, "second array");
        arr1[1] = 444;           // this will change BOTH arrays!
        show(arr1, "first array");
        show(arr2, "second array");
        return 0;
    }
    public static void show (int [] arr, string caption)
    {
        Console.WriteLine("----{0}----", caption);
        for (int i = 0; i < arr.Length; i++)
        {
            Console.Write("{0} ", arr[i]);
        }
        Console.WriteLine();
    }
}

```

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. Here is the output:

```

----first array----
1 4 9
----second array----
1 4 9
----first array----
1 444 9
----second array----
1 444 9

```

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.

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.

```
// Course.cs

using System;
using System.Collections;

public class Course : ICloneable
{
    public string Title;
    public ArrayList Roster;
    public Course(string title)
    {
        Title = title;
        Roster = new ArrayList();
    }
    public void AddStudent(string name)
    {
        Roster.Add(name);
    }
    public void Show(string caption)
    {
        Console.WriteLine("-----{0}-----", caption);
        Console.WriteLine("Course : {0} with {1} students",
            Title, Roster.Count);
        foreach (string name in Roster)
        {
            Console.WriteLine(name);
        }
    }
    public Course ShallowCopy()
    {
        return (Course) this.MemberwiseClone();
    }
    public object Clone()
    {
        Course course = new Course(Title);
        course.Roster = (ArrayList) Roster.Clone();
        return course;
    }
}
```

The test program constructs a **Course** instance **c1** and then makes a copy **c2** by various methods.

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.

```
// CopyDemo.cs

using System;
using System.Collections;

public class CopyDemo
{
    private static Course c1, c2;
    public static void Main()
    {
        Console.WriteLine("Copy is done via c2 = c1");
        InitializeCourse();
        c1.Show("original");
        c2 = c1;
        c2.Show("copy");
        c2.Title = ".NET Programming";
        c2.AddStudent("Charlie");
        c2.Show("copy with changed title and new student");
        c1.Show("original");

        ...
    }
    private static void InitializeCourse()
    {
        c1 = new Course("Intro to C#");
        c1.AddStudent("John");
        c1.AddStudent("Mary");
    }
}
```

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. Here is the output from this first part of the program:

```
Copy is done via c2 = c1
-----original-----
Course : Intro to C# with 2 students
John
Mary
-----copy-----
Course : Intro to C# with 2 students
John
```

```

Mary
-----copy with changed title and new student-----
Course : .NET Programming with 3 students
John
Mary
Charlie
-----original-----
Course : .NET Programming with 3 students
John
Mary
Charlie

```

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**.

```

// Course.cs

using System;
using System.Collections;

public class Course : ICloneable
{
    ...
    public Course ShallowCopy()
    {
        return (Course) this.MemberwiseClone();
    }
    ...
}

```

Here is the second part of the test program, which calls the **ShallowCopy** method. Again we change the title and a student in the second copy.

```

// CopyDemo.cs

using System;
using System.Collections;

public class CopyDemo
{
    ...
    Console.WriteLine(
        "\nCopy is done via c2 = c1.ShallowCopy()");
}

```

```

InitializeCourse();
c2 = c1.ShallowCopy();
c2.Title = ".NET Programming";
c2.AddStudent("Charlie");
c2.Show("copy with changed title and new student");
c1.Show("original");
...

```

Here is the output of this second part of the program. Now 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.

```

Copy is done via c2 = c1.ShallowCopy()
-----copy with changed title and new student-----
Course : .NET Programming with 3 students
John
Mary
Charlie
-----original-----
Course : Intro to C# with 3 students
John
Mary
Charlie

```

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.

```

// Course.cs

using System;
using System.Collections;

public class Course : ICloneable
{
    ...
    public object Clone()
    {
        Course course = new Course(Title);
        course.Roster = (ArrayList) Roster.Clone();
        return course;
    }
}

```

Here is the third part of the test program, which calls the **Clone** method. Again we change the title and a student in the second copy.

```
// CopyDemo.cs

using System;
using System.Collections;

public class CopyDemo
{
    ...
    Console.WriteLine(
        "\nCopy is done via c2 = c1.Clone()");
    InitializeCourse();
    c2 = (Course) c1.Clone();
    c2.Title = ".NET Programming";
    c2.AddStudent("Charlie");
    c2.Show("copy with changed title and new student");
    c1.Show("original");
    ...
}
```

Here is the output from the third part of the program. Now we have completely independent instances of **Course**. Each has its own title and set of students.

```
Copy is done via c2 = c1.Clone()
-----copy with changed title and new student-----
Course : .NET Programming with 3 students
John
Mary
Charlie
-----original-----
Course : Intro to C# with 2 students
John
Mary
```

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.

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**.

```
// ArrayName.cs - Step 0

using System;

public class Name
{
    private string text;
    public Name(string text)
    {
        this.text = text;
    }
    public string Text
    {
        get
        {
            return text;
        }
    }
}

public class ArrayName
{
    public static int Main(string[] args)
    {
        Name[] array = new Name[10];
        array[0] = new Name("Michael");
        array[1] = new Name("Charlie");
        array[2] = new Name("Peter");
        array[3] = new Name("Dana");
        array[4] = new Name("Bob");
        Array.Sort(array);
        return 0;
    }
}
```

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?

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**.

```
// ArrayName.cs - Step 1

...
public class ArrayName
{
    public static int Main(string[] args)
    {
        Name[] array = new Name[10];
        array[0] = new Name("Michael");
        array[1] = new Name("Charlie");
        array[2] = new Name("Peter");
        array[3] = new Name("Dana");
        array[4] = new Name("Bob");
        if (array[0] is IComparable)
            Array.Sort(array);
        else
            Console.WriteLine(
                "Name does not implement IComparable");
        return 0;
    }
}
```

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
```

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.

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.

```
// ArrayName.cs - Step 2

using System;

public class Name : IComparable
{
    private string text;
    public Name(string text)
    {
        this.text = text;
    }
    public string Text
    {
        get
        {
            return text;
        }
    }
    public int CompareTo(object obj)
    {
        string s1 = this.Text;
        string s2 = ((Name) obj).Text;
        return String.Compare(s1, s2);
    }
}

public class ArrayName
{
    public static int Main(string[] args)
    {
        ...
        foreach (Name name in array)
            Console.WriteLine(name);
        return 0;
    }
}
```

```
    }  
}
```

AN INCOMPLETE SOLUTION

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.

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. Here is the complete solution, in the directory **ArrayName\Step3**.

```
// ArrayName.cs - Step 3  
  
using System;  
  
public class Name : IComparable  
{  
    private string text;  
    public Name(string text)  
    {  
        this.text = text;  
    }  
    public string Text  
    {  
        get  
        {  
            return text;  
        }  
    }  
    public int CompareTo(object obj)  
    {
```

```
        string s1 = this.Text;
        string s2 = ((Name) obj).Text;
        return String.Compare(s1, s2);
    }
    override public string ToString()
    {
        return text;
    }
}

public class ArrayName
{
    public static int Main(string[] args)
    {
        Name[] array = new Name[10];
        array[0] = new Name("Michael");
        array[1] = new Name("Charlie");
        array[2] = new Name("Peter");
        array[3] = new Name("Dana");
        array[4] = new Name("Bob");
        if (array[0] is IComparable)
            Array.Sort(array);
        else
            Console.WriteLine(
                "Name does not implement IComparable");
        foreach (Name name in array)
        {
            if (name != null)
                Console.WriteLine(name);
        }
        return 0;
    }
}
```

Here is the output:

```
Bob
Charlie
Dana
Michael
Peter
```

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.

SUMMARY

In this chapter we examined the ubiquitous role of interfaces in 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. Collections support the interfaces **IEnumerable**, **ICollection**, **IList**, and **ICloneable**. The first three interfaces provide the standard methods for iterating the elements of a list, obtaining a count of the number of elements, adding and removing elements, and so on. The **ICloneable** interface is used to implement a deep copy of a class. In order to work with collections effectively, you need to override certain methods of the **object** base class, such as **Equals**. We also looked at comparison of objects, which 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. There is rich functionality that you can call, and there are many interfaces that you can optionally implement to make your program behave properly when called by the framework.

In the next chapter we will look at another variety of another program calling into your code. We will look at *delegates*, which can be viewed as object-oriented, type-safe callback functions. We will also look at *events*, which are a higher level construct built on top of delegates.