CHAPTER 14

Generics and Collections

You saw in Chapter 10 that arrays are useful for when you have a group of objects of the same type, and you need to treat them as a group—as a *collection*. Arrays are the least flexible of the five standard collections used in C# 2005:

- Array
- List
- Stack
- Queue
- Dictionary

This chapter will introduce each of the latter four collections, and will show how the new feature *generics* are used to make these collections type-safe (and why type-safety is important!).

You can also create classes that *act like* collections, and you can provide support for your collection classes so that they support some or all of the behavior expected of collections like the ability to be used in a `foreach` loop or to access their members using an indexer:

```
Employee joe = MyCompany[EmployeeID]
```

Generics

Until generics, all the collection classes (`Array`, `List`, `Stack`, and `Queue`) were defined to hold objects of type `Object` (the root class). Thus, you could add integers and strings to the same class, and when you took items out of the collection, you had to cast them to their "real" type. This was ugly, and it was error-prone (the compiler could not tell if you had a collection of integers and added a string).

With generics, the designer of the class (the person who creates the `Stack` class) can say, "This class will hold only one type, and that type will be defined by the developer who makes an instance of this class."

The user of the generic Stack class defines an instance of the Stack and the type it will hold, and the compiler can now use this type-safe Stack to ensure that only objects of the designated type are stored in the collection. Much better.

The designer adds a type placeholder (technically called a *type parameter*):

```
class Stack<T>
```

The user of the Stack class puts in the actual type when instantiating the class, like this:

```
class Stack<Employee> = new Stack<Employee>
```

Collection Interfaces

The .NET Framework provides a number of interfaces, such as IEnumerable and ICollection, that the designer of a collection must implement to provide full collection semantics. For example, ICollection allows your collection to be enumerated in a foreach loop.

Creating Your Own Collections

The goal in creating your own collections is to make them as similar to the standard .NET collections as possible. This reduces confusion, and makes for easier-to-use classes and easier-to-maintain code.

One feature you may want to provide is to allow users of your collection to add to or extract from the collection with an indexer, just as is done with arrays.

For example, suppose you create a ListBox control named myListBox that contains a list of strings stored in a one-dimensional array, a private member variable named myStrings. A ListBox control contains member properties and methods in addition to its array of strings, so the ListBox itself is not an array. However, it would be convenient to be able to access the ListBox array with an index, just as if the ListBox itself were an array.* For example, such a property would permit statements such as the following:

```
string theFirstString = myListBox[0];
string theLastString = myListBox[Length-1];
```

An *indexer* is a C# construct that allows you to treat a class as if it were an array. In the preceding example, you are treating the ListBox as if it were an array of strings, even though it is more than that. An indexer is a special kind of property, but like all properties, it includes get and set accessors to specify its behavior.

* The actual ListBox control provided by both Windows forms and ASP.NET has a collection called Items, and it is the Items collection that implements the indexer.

You declare an indexer property within a class using the following syntax:

```
type this [type argument]{get; set;}
```

The return type determines the type of object that will be returned by the indexer, while the type argument specifies what kind of argument will be used to index into the collection that contains the target objects. Although it is common to use integers as index values, you can index a collection on other types as well, including strings. You can even provide an indexer with multiple parameters to create a multidimensional array!

The `this` keyword is a reference to the object in which the indexer appears. As with a normal property, you also must define `get` and `set` accessors, which determine how the requested object is retrieved from or assigned to its collection.

Example 14-1 declares a `ListBox` control (`ListBoxTest`) that contains a simple array (`myStrings`) and a simple indexer for accessing its contents.

Example 14-1. Using a simple indexer

```
using System;

namespace SimpleIndexer
{
   // a simplified ListBox control
   public class ListBoxTest
   {
      private string[] strings;
      private int ctr = 0;

      // initialize the list box with strings
      public ListBoxTest( params string[] initialStrings )
      {
         // allocate space for the strings
         strings = new String[256];

         // copy the strings passed in to the constructor
         foreach ( string s in initialStrings )
         {
            strings[ctr++] = s;
         }
      }

      // add a single string to the end of the list box
      public void Add( string theString )
      {
         if ( ctr >= strings.Length )
         {
            // handle bad index
         }
         else
            strings[ctr++] = theString;
      }
```

Example 14-1. Using a simple indexer (continued)

```
    // allow array-like access

    public string this[int index]
    {
       get
       {
          if ( index < 0 || index >= strings.Length )
          {
             // handle bad index
          }
          return strings[index];
       }
       set
       {
          // add only through the add method
          if ( index >= ctr )
          {
             // handle error
          }
          else
             strings[index] = value;
       }
    }

    // publish how many strings you hold
    public int GetNumEntries( )
    {
       return ctr;
    }
}

public class Tester
{
    static void Main( )
    {
       // create a new list box and initialize
       ListBoxTest lbt =
          new ListBoxTest( "Hello", "World" );

       // add a few strings
       lbt.Add( "Proust" );
       lbt.Add( "Faulkner" );
       lbt.Add( "Mann" );
       lbt.Add( "Hugo" );

       // test the access
       string subst = "Universe";
       lbt[1] = subst;

       // access all the strings
       for ( int i = 0; i < lbt.GetNumEntries( ); i++ )
       {
```

Example 14-1. Using a simple indexer (continued)

```
            Console.WriteLine( "lbt[{0}]: {1}", i, lbt[i] );
        }
      }
   }
}
```

The output looks like this:

```
lbt[0]: Hello
lbt[1]: Universe
lbt[2]: Proust
lbt[3]: Faulkner
lbt[4]: Mann
lbt[5]: Hugo
```

To keep Example 14-1 simple, we strip the `ListBox` control down to the few features we care about. The listing ignores everything having to do with being a user control and focuses only on the list of strings the `ListBox` maintains, and methods for manipulating them. In a real application, of course, these are a small fraction of the total methods of a `ListBox`, whose principal job is to display the strings and enable user choice.

The first things to notice in this example are the two private members:

```
private string[] strings;
private int ctr = 0;
```

Our `ListBox` maintains a simple array of strings, cleverly named `strings`. The member variable `ctr` will keep track of how many strings have been added to this array.

Initialize the array in the constructor with the statement:

```
strings = new String[256];
```

The `Add( )` method of `ListBoxTest` does nothing more than append a new string to its internal array (`strings`), though a more complex object might write the strings to a database or other more complex data structure.

The key method of `ListBoxTest`, is the indexer. An indexer uses the `this` keyword:

```
public string this[int index]
```

The syntax of the indexer is very similar to that for properties. There is either a `get( )` method, a `set( )` method, or both. In the case shown, the `get( )` method endeavors to implement rudimentary bounds checking, and assuming the index requested is acceptable, it returns the value requested:

```
get
{
    if (index < 0 || index >= strings.Length)
    {
       // handle bad index
    }
    return strings[index];
}
```

How you handle a bad index is up to you. A common approach is to throw an exception; see the help files for information on the "ArgumentOutOfRange" exception.

The `set()` method checks to make sure that the index you are setting already has a value in the `ListBox`. If not, it treats the set as an error. (New elements can only be added using `Add` with this approach.) The `set` accessor takes advantage of the implicit parameter `value` that represents whatever is assigned using the index operator:

```
set
{
   if (index >= ctr )
   {
      // handle error
   }
   else
   {
      strings[index] = value;
   }
}
```

Thus, if you write:

```
lbt[5] = "Hello World"
```

the compiler will call the indexer `set()` method on your object and pass in the string `Hello World` as an implicit parameter named `value`.

Indexers and Assignment

In Example 14-1, you cannot assign to an index that does not have a value. Thus, if you write:

```
lbt[10] = "wow!";
```

you would trigger the error handler in the `set()` method, which would note that the index you've passed in (`10`) is larger than the counter (`6`).

This code is kept simple, and thus is not robust. There are any number of other checks you'll want to make on the value passed in (e.g., checking that you were not passed a negative index and that it does not exceed the size of the underlying `strings[]` array).

In `Main()`, you create an instance of the `ListBoxTest` class named `lbt` and pass in two strings as parameters:

```
ListBoxTest lbt = new ListBoxTest("Hello", "World");
```

Then call `Add()` to add four more strings:

```
// add a few strings
lbt.Add( "Proust" );
```

```
lbt.Add( "Faulkner" );
lbt.Add( "Mann" );
lbt.Add( "Hugo" );
```

Before examining the values, modify the second value (at index 1):

```
string subst = "Universe";
lbt[1] = subst;
```

Finally, display each value in a loop:

```
for (int i = 0;i<lbt.GetNumEntries( );i++)
{
    Console.WriteLine("lbt[{0}]: {1}",i,lbt[i]);
}
```

Indexing on Other Values

C# does not require that you always use an integer value as the index to a collection. When you create a custom collection class and create your indexer, you are free to create indexers that index on strings and other types. In fact, the index value can be overloaded so that a given collection can be indexed, for example, by an integer value and also by a string value, depending on the needs of the client.

Example 14-2 illustrates a string index. The indexer calls `FindString( )`, which is a helper method that returns a record based on the value of the string provided. Notice that the overloaded indexer and the indexer from Example 14-1 are able to coexist.

Example 14-2. Overloading an index

```
using System;

namespace OverloadedIndexer
{
   // a simplified ListBox control
   public class ListBoxTest
   {
      private string[] strings;
      private int ctr = 0;

      // initialize the list box with strings
      public ListBoxTest( params string[] initialStrings )
      {
         // allocate space for the strings
         strings = new String[256];

         // copy the strings passed in to the constructor
         foreach ( string s in initialStrings )
         {
            strings[ctr++] = s;
         }
      }
```

Example 14-2. Overloading an index (continued)

```
// add a single string to the end of the list box
public void Add( string theString )
{
   if ( ctr >= strings.Length )
   {
      // handle bad index
   }
   else
   {
      strings[ctr++] = theString;
   }
}


// allow array-like access
public string this[int index]
{
   get
   {
      if ( index < 0 || index >= strings.Length )
      {
         // handle bad index
      }
      return strings[index];
   }
   set
   {
      // add only through the add method
      if ( index >= ctr )
      {
         // handle error
      }
      else
         strings[index] = value;
   }
}

private int FindString( string searchString )
{
   for ( int i = 0; i < strings.Length; i++ )
   {
      if ( strings[i].StartsWith( searchString ) )
      {
         return i;
      }
   }
   return -1;
}

// index on string
public string this[string index]
{
```

Example 14-2. Overloading an index (continued)

```
        get
        {
           if ( index.Length == 0 )
           {
              // handle bad index
           }
           return this[FindString( index )];
        }
        set
        {
           // no need to check the index here because
           // find string will handle a bad index value
           strings[FindString( index )] = value;
        }
     }



     // publish how many strings you hold
     public int GetNumEntries( )
     {
        return ctr;
     }
  }

  public class Tester
  {
     static void Main( )
     {
        // create a new list box and initialize
        ListBoxTest lbt =
           new ListBoxTest( "Hello", "World" );

        // add a few strings
        lbt.Add( "Proust" );
        lbt.Add( "Faulkner" );
        lbt.Add( "Mann" );
        lbt.Add( "Hugo" );

        // test the access
        string subst = "Universe";
        lbt[1] = subst;
        lbt["Hel"] = "GoodBye";
        // lbt["xyz"] = "oops";

        // access all the strings
        for ( int i = 0; i < lbt.GetNumEntries( ); i++ )
        {
           Console.WriteLine( "lbt[{0}]: {1}", i, lbt[i] );
        }      // end for
     }         // end main
  }            // end tester
}
```

The output looks like this:

```
lbt[0]: GoodBye
lbt[1]: Universe
lbt[2]: Proust
lbt[3]: Faulkner
lbt[4]: Mann
lbt[5]: Hugo
```

Example 14-2 is identical to Example 14-1 except for the addition of an overloaded indexer, which can match a string, and the method `FindString`, created to support that index.

The `FindString` method simply iterates through the strings held in `myStrings` until it finds a string that starts with the target string used in the index. If found, it returns the index of that string; otherwise, it returns the value `-1`.

You can see in `Main( )` that the user passes in a string segment to the index, just as with an integer:

```
lbt["Hel"] = "GoodBye";
```

This calls the overloaded index, which does some rudimentary error-checking (in this case, making sure the string passed in has at least one letter) and then passes the value (`Hel`) to `FindString`. It gets back an index and uses that index to index into `myStrings`:

```
return this[FindString(index)];
```

The `set` value works in the same way:

```
myStrings[FindString(index)] = value;
```

The careful reader will note that if the string does not match, a value of `-1` is returned, which is then used as an index into `myStrings`. This action then generates an exception (`System.NullReferenceException`), as you can see by uncommenting the following line in `Main( )`:

```
lbt["xyz"] = "oops";
```

The proper handling of not finding a string is, as they say, left as an exercise for the reader. You might consider displaying an error message or otherwise allowing the user to recover from the error.

Generic Collection Interfaces

The .NET Framework provides standard interfaces for enumerating and comparing collections. These standard interfaces are type-safe, but the type is *generic*; that is, you can declare an `ICollection` of any type by substituting the actual type (`int`, `string`, or `Employee`) for the generic type in the interface declaration (`<T>`).

For example, if you were creating an interface called IStorable, but you didn't know what kind of objects would be stored, you'd declare the interface like this:

```
interface IStorable<T>
{
  // implementation here
}
```

Later on, if you wanted to create a class Document that implemented IStorable to store strings, you'd do it like this:

```
public class Document : IStorable<String>
```

Replacing T with the type you want to apply the interface to (in this case, string).

Shockingly, perhaps, that is all there is to generics. The creator of the class says, in essence, "This applies to some type <T> to be named later (when the interface or class is used) and the programmer using the interface or collection type replaces <T> with the actual type (e.g., int, string, Employee, etc.)."

The key generic collection interfaces are listed in Table 14-1.*

Table 14-1. Generic collection interfaces

Interface	Purpose
ICollection<T>	Base interface for generic collections
IEnumerator<T> IEnumerable<T>	Required for collections that will be enumerated with foreach
IComparer<T> IComparable<T>	Required for collections that will be sorted
IList<T>	Used by indexable collections (see the section "Generic Lists: List<T>")
IDictionary<K,V>	Used for key/value-based collections (see the section "Dictionaries")

The IEnumerable<T> Interface

You can support the foreach statement in ListBoxTest by implementing the IEnumerable<T> interface.

You read this as "IEnumerable of <T>" or "the generic interface IEnumerable."

* C# also provides nongeneric interfaces (ICollection, IEnumerator), but we will focus on the generic collections, which should be preferred whenever possible as they are type-safe.

IEnumerable has only one method, GetEnumerator( ), whose job is to return an implementation of IEnumerator<T>. The C# language provides special help in creating the enumerator, using the new keyword yield, as demonstrated in Example 14-3 and explained below.

Example 14-3. Making a ListBox an enumerable class

```
using System;
using System.Collections.Generic;  // for the generic classes

namespace Enumerable
{
   public class ListBoxTest : IEnumerable<String>
   {
      private string[] strings;
      private int ctr = 0;
      // Enumerable classes can return an enumerator
      public IEnumerator<string> GetEnumerator( )
      {
         foreach ( string s in strings )
         {
            yield return s;
         }
      }

      // required to fulfill IEnumerable
      System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator( )
      {
         throw new NotImplementedException( );
      }


      // initialize the list box with strings
      public ListBoxTest( params string[] initialStrings )
      {
         // allocate space for the strings
         strings = new String[256];

         // copy the strings passed in to the constructor
         foreach ( string s in initialStrings )
         {
            strings[ctr++] = s;
         }
      }

      // add a single string to the end of the list box
      public void Add( string theString )
      {
         strings[ctr] = theString;
         ctr++;
      }
```

Example 14-3. Making a ListBox an enumerable class (continued)

```
    // allow array-like access
    public string this[int index]
    {
       get
       {
          if ( index < 0 || index >= strings.Length )
          {
             // handle bad index
          }
          return strings[index];
       }
       set
       {
          strings[index] = value;
       }
    }

    // publish how many strings you hold
    public int GetNumEntries( )
    {
       return ctr;
    }

}

public class Tester
{
   static void Main( )
   {
      // create a new list box and initialize
      ListBoxTest lbt =
         new ListBoxTest( "Hello", "World" );

      // add a few strings
      lbt.Add( "Proust" );
      lbt.Add( "Faulkner" );
      lbt.Add( "Mann" );
      lbt.Add( "Hugo" );

      // test the access
      string subst = "Universe";
      lbt[1] = subst;

      // access all the strings
      foreach ( string s in lbt )
      {
         if ( s == null )
         {
            break;
         }
```

Example 14-3. Making a ListBox an enumerable class (continued)

```
            Console.WriteLine( "Value: {0}", s );
        }
    }
  }
}
```

The output looks like this:

```
Value: Hello
Value: Universe
Value: Proust
Value: Faulkner
Value: Mann
Value: Hugo
```

The program begins in `Main( )`, creating a new `ListBoxTest` object and passing two strings to the constructor. When the object is created, an array of `Strings` is created with enough room for 256 strings. Four more strings are added using the `Add` method, and the second string is updated, just as in the previous example.

The big change in this version of the program is that a `foreach` loop is called, retrieving each string in the `ListBox`. The `foreach` loop *automatically* uses the `IEnumerable<T>` interface, invoking `GetEnumerator( )`.

The `GetEnumerator` method is declared to return an `IEnumerator` of type `string`:

```
public IEnumerator<string> GetEnumerator( )
```

The implementation iterates through the array of strings, yielding each in turn:

```
foreach ( string s in strings )
{
  yield return s;
}
```

The new keyword `yield` is used here explicitly to return a value to the enumerator object. By using the `yield` keyword, all the bookkeeping for keeping track of which element is next, resetting the iterator, and so forth, is provided for you by the framework.

Note that our implementation includes an implementation of the non-generic `GetEnumerator` method. This is required by the definition of the generic `IEnumerable` and is typically defined to just throw an exception, since we don't expect to call it:

```
// required to fulfill IEnumerable
System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator( )
{
   throw new NotImplementedException( );
}
```

Framework Generic Collections

The .NET Framework provides four very useful generic collections, as enumerated earlier (`List`, `Stack`, `Queue`, and `Dictionary`). The most common case is that rather than writing your own collection, you'll use one of the collection classes provided for you. Each is described in turn in the next few sections.

Generic Lists: List<T>

The classic problem with the `Array` type is its fixed size. If you do not know in advance how many objects an array will hold, you run the risk of declaring either too small an array (and running out of room) or too large an array (and wasting memory).

The generic `List` class (which replaces the old non-generic `ArrayList`) is, essentially, an array whose size is dynamically increased as required. `Lists` provide a number of useful methods and properties for their manipulation. Some of the most important are shown in Table 14-2.

Table 14-2. List properties and methods

Method or property	Purpose
`Capacity`	Property to get or set the number of elements the `List` can contain. This value is increased automatically if count exceeds capacity. You might set this value to reduce the number of reallocations, and you may call `Trim( )` to reduce this value to the actual `Count`.
`Count`	Property to get the number of elements currently in the list.
`Item( )`	Gets or sets the element at the specified index. This is the indexer for the `List` class.[a]
`Add( )`	Public method to add an object to the `List`.
`AddRange( )`	Public method that adds the elements of an `ICollection` to the end of the `List`.
`BinarySearch( )`	Overloaded public method that uses a binary search to locate a specific element in a sorted `List`.
`Clear( )`	Removes all elements from the `List`.
`Contains( )`	Determines if an element is in the `List`.
`CopyTo( )`	Overloaded public method that copies a `List` to a one-dimensional array.
`Exists( )`	Determines if an element is in the `List`.
`Find( )`	Returns the first occurrence of the element in the `List`.
`FindAll( )`	Returns all the specified elements in the `List`.
`GetEnumerator( )`	Overloaded public method that returns an enumerator to iterate through a `List`.
`GetRange( )`	Copies a range of elements to a new `List`.
`IndexOf( )`	Overloaded public method that returns the index of the first occurrence of a value.
`Insert( )`	Inserts an element into `List`.
`InsertRange( )`	Inserts the elements of a collection into the `List`.
`LastIndexOf( )`	Overloaded public method that returns the index of the last occurrence of a value in the `List`.
`Remove( )`	Removes the first occurrence of a specific object.

Table 14-2. List properties and methods (continued)

Method or property	Purpose
`RemoveAt( )`	Removes the element at the specified index.
`RemoveRange( )`	Removes a range of elements.
`Reverse( )`	Reverses the order of elements in the `List`.
`Sort( )`	Sorts the `List`.
`ToArray( )`	Copies the elements of the `List` to a new array.
`TrimToSize( )`	Sets the capacity to the actual number of elements in the `List`.

[a] The idiom in the Framework Class Library is to provide an Item element for collection classes, which is implemented as an indexer in C#.

When you create a `List`, you do not define how many objects it will contain. You add to the `List` using the `Add( )` method, and the `List` takes care of its own internal bookkeeping, as illustrated in Example 14-4.

Example 14-4. Working with a List

```
using System;
using System.Collections.Generic;

namespace ListCollection
{
   // a simple class to store in the List
   public class Employee
   {
      private int empID;

      public Employee( int empID )
      {
         this.empID = empID;
      }
      public override string ToString( )
      {
         return empID.ToString( );
      }
      public int EmpID
      {
         get
         {
            return empID;
         }
         set
         {
            empID = value;
         }
      }
   }
   public class Tester
   {
      static void Main( )
      {
```

Example 14-4. Working with a List (continued)

```
        List<Employee> empList = new List<Employee>( );
        List<int> intList = new List<int>( );

        // populate the List
        for ( int i = 0; i < 5; i++ )
        {
           empList.Add( new Employee( i + 100 ) );
           intList.Add( i * 5 );
        }


        // print all the contents
        for ( int i = 0; i < intList.Count; i++ )
        {
           Console.Write( "{0} ", intList[i].ToString( ) );
        }

        Console.WriteLine( "\n" );

        // print all the contents of the Employee List
        for ( int i = 0; i < empList.Count; i++ )
        {
           Console.Write( "{0} ", empList[i].ToString( ) );
        }

        Console.WriteLine( "\n" );
        Console.WriteLine( "empList.Capacity: {0}", empList.Capacity );
     }
   }
}
```

The output looks like this:

```
0 5 10 15 20
100 101 102 103 104
empArray.Capacity: 8
```

The `List` class has a property, `Capacity`, which is the number of elements the `List` is capable of storing; however, this capacity is automatically doubled each time you reach the limit.

Creating objects that can be sorted by the generic list

The `List` implements the `Sort( )` method. You can sort any `List` that contains objects that implement `IComparable`. All the built-in types do implement this interface, so you can sort a `List<integer>` or a `List<string>`.

On the other hand, if you want to sort a `List<Employee>`, you must change the `Employee` class to implement `IComparable`:

```
public class Employee : IComparable<Employee>
```

As part of the IComparable interface contract, the Employee object must provide a CompareTo( ) method:

```
public int CompareTo(Employee rhs)
{
   return this.empID.CompareTo(rhs.empID);
}
```

The CompareTo( ) method takes an Employee as a parameter (we know this, because the interface is now generic and we can assume type safety). The Employee object must compare itself to this second Employee object and return -1 if it is smaller than the second Employee, 1 if it is greater, and 0 if the two Employee objects are equal to each other.

It is up to the designer of the Employee class to determine what *smaller than*, *greater than*, and *equal to* mean. In this example, you will delegate the comparison to the empId member. The empId member is an int and uses the default CompareTo( ) method for integer types, which will do an integer comparison of the two values.

To see if the sort is working, you'll add integers and Employee instances to their respective lists with random values. To create the random values, you'll instantiate an object of class Random. To cause your Random instance to generate the random values, you'll call its Next( ) method. One version of the Next( ) method allows you to specify the largest random number you want. In this case, you'll pass in the value 10 to generate a random number between 0 and 10:

```
Random r = new Random( );
r.Next(10);
```

Example 14-5 creates an integer array and an Employee array, populates them both with random numbers, and prints their values. It then sorts both arrays and prints the new values.

Example 14-5. Sorting an integer and an Employee array

```
using System;
using System.Collections.Generic;

namespace IComparable
{
   // a simple class to store in the array
   public class Employee : IComparable<Employee>
   {
      private int empID;

      public Employee( int empID )
      {
         this.empID = empID;
      }

      public override string ToString( )
      {
```

Example 14-5. Sorting an integer and an Employee array (continued)

```
        return empID.ToString( );
    }


    public bool Equals( Employee other )
    {
        if ( this.empID == other.empID )
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    // Comparer delegates back to Employee
    // Employee uses the integer's default
    // CompareTo method

    public int CompareTo( Employee rhs )
    {
        return this.empID.CompareTo( rhs.empID );
    }
}
public class Tester
{
    static void Main( )
    {
        List<Employee> empList = new List<Employee>( );
        List<Int32> intList = new List<Int32>( );

        // generate random numbers for
        // both the integers and the
        // employee id's
        Random r = new Random( );

        // populate the array
        for ( int i = 0; i < 5; i++ )
        {
            // add a random employee id
            empList.Add( new Employee( r.Next( 10 ) + 100 ) );

            // add a random integer
            intList.Add( r.Next( 10 ) );
        }


        // display all the contents of the int array
        Console.WriteLine("List<int> before sorting:");
        for ( int i = 0; i < intList.Count; i++ )
        {
```

Example 14-5. Sorting an integer and an Employee array (continued)

```
            Console.Write( "{0} ", intList[i].ToString( ) );
        }
        Console.WriteLine( "\n" );

        // display all the contents of the Employee array
        Console.WriteLine("List<Employee> before sorting:");
        for ( int i = 0; i < empList.Count; i++ )
        {
            Console.Write( "{0} ", empList[i].ToString( ) );
        }
        Console.WriteLine( "\n" );

        // sort and display the int array
        Console.WriteLine("List<int>after sorting:");
        intList.Sort( );
        for ( int i = 0; i < intList.Count; i++ )
        {
            Console.Write( "{0} ", intList[i].ToString( ) );
        }
        Console.WriteLine( "\n" );

        // sort and display the Employee array
        Console.WriteLine("List<Employee>after sorting:");
        //Employee.EmployeeComparer c = Employee.GetComparer( );
        //empList.Sort(c);

        empList.Sort( );


        // display all the contents of the Employee array
        for ( int i = 0; i < empList.Count; i++ )
        {
            Console.Write( "{0} ", empList[i].ToString( ) );
        }
        Console.WriteLine( "\n" );

    }
  }
}
```

The output looks like this:

```
List<int> before sorting:
6 9 8 3 6
List<Employee> before sorting:
108 103 107 102 109
List<int>after sorting:
3 6 6 8 9
List<Employee>after sorting:
102 103 107 108 109
```

The output shows that the lists of integers and Employees were generated with random numbers (and thus the numbers may be different each time you run the program). When sorted, the display shows the values have been ordered properly.

Random number generators do not, technically, create true random numbers; they create what computer scientists call pseudo-random numbers. Microsoft notes: "Pseudo-random numbers are chosen with equal probability from a finite set of numbers. The chosen numbers are not completely random because a definite mathematical algorithm is used to select them, but they are sufficiently random for practical purposes." This is certainly sufficient for our example.

Controlling how elements in a generic collection are sorted by implementing IComparer<T>

When you call Sort( ) on the List in Example 14-5, the default implementation of IComparer is called, which uses QuickSort to call the IComparable implementation of CompareTo( ) on each element in the List.

You are free, however, to create your own implementation of IComparer, which you might want to do if you need control over how the sort ordering is defined. In the next example, you will add a second field to Employee: yearsOfSvc. You want to be able to sort the Employee objects in the List either by ID or by years of service, and you want to make that decision at run time.

To accomplish this, you will create a custom implementation of IComparer, which you will pass to the Sort( ) method of List. In this IComparer class, EmployeeComparer will know how to sort Employees.

To simplify the programmer's ability to choose how a given set of Employees are sorted, we'll add a property WhichComparison, of type Employee.EmployeeComparer.ComparisonType (an enumeration):

```
public
Employee.EmployeeComparer.ComparisonType  WhichComparison
{
   get { return whichComparison; }
   set { whichComparison = value; }
}
```

ComparisonType is an enumeration with two values, empID or yearsOfSvc (indicating that you want to sort by employee ID or years of service, respectively):

```
public enum ComparisonType
{
   EmpID,
   YearsOfService
};
```

Before invoking Sort( ), you will create an instance of EmployeeComparer and set its ComparisonType property:

```
Employee.EmployeeComparer c = Employee.GetComparer( );
c.WhichComparison=Employee.EmployeeComparer.ComparisonType.EmpID;
empArray.Sort(c);
```

When you invoke Sort( ), the List will call the Compare( ) method on the EmployeeComparer, which in turn will delegate the comparison to the Employee.CompareTo( ) method, passing in its WhichComparison property:

```
public int Compare( Employee lhs, Employee rhs )
{
    return lhs.CompareTo( rhs, WhichComparison );
}
```

The Employee object must implement a custom version of CompareTo( ), which takes the comparison and compares the objects accordingly:

```
public int CompareTo
    (
    Employee rhs,
    Employee.EmployeeComparer.ComparisonType whichComparison
    )
{
   switch (whichComparison)
   {
      case Employee.EmployeeComparer.ComparisonType.EmpID:
         return this.empID.CompareTo(rhs.empID);
      case Employee.EmployeeComparer.ComparisonType.Yrs:
         return this.yearsOfSvc.CompareTo(rhs.yearsOfSvc);
   }
   return 0;
}
```

The complete source for this example is shown in Example 14-6. The integer array has been removed to simplify the example, and the output of the employee's ToString( ) method has been enhanced to enable you to see the effects of the sort.

Example 14-6. Sorting an array by employees' IDs and years of service

```
using System;
using System.Collections.Generic;

namespace IComparer
{
   public class Employee : IComparable<Employee>
   {
      private int empID;

      private int yearsOfSvc = 1;

      public Employee( int empID )
      {
```

Example 14-6. Sorting an array by employees' IDs and years of service (continued)

```
        this.empID = empID;
    }


    public Employee( int empID, int yearsOfSvc )
    {
        this.empID = empID;
        this.yearsOfSvc = yearsOfSvc;
    }

    public override string ToString( )
    {
        return "ID: " + empID.ToString( ) +
        ". Years of Svc: " + yearsOfSvc.ToString( );
    }

    public bool Equals( Employee other )
    {
        if ( this.empID == other.empID )
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    // static method to get a Comparer object
    public static EmployeeComparer GetComparer( )
    {
        return new Employee.EmployeeComparer( );
    }


    // Comparer delegates back to Employee
    // Employee uses the integer's default
    // CompareTo method
    public int CompareTo( Employee rhs )
    {
        return this.empID.CompareTo( rhs.empID );
    }


    // Special implementation to be called by custom comparer
    public int CompareTo(
        Employee rhs,
        Employee.EmployeeComparer.ComparisonType which )
    {
        switch ( which )
        {
            case Employee.EmployeeComparer.ComparisonType.EmpID:
```

Example 14-6. Sorting an array by employees' IDs and years of service (continued)

```
            return this.empID.CompareTo( rhs.empID );
         case Employee.EmployeeComparer.ComparisonType.Yrs:
            return this.yearsOfSvc.CompareTo( rhs.yearsOfSvc );
      }
      return 0;

   }

   // nested class which implements IComparer
   public class EmployeeComparer : IComparer<Employee>
   {
      // private state variable
      private Employee.EmployeeComparer.ComparisonType
         whichComparison;

      // enumeration of comparsion types
      public enum ComparisonType
      {
         EmpID,
         Yrs
      };

      public  bool Equals( Employee lhs, Employee rhs )
      {
         return this.Compare( lhs, rhs ) == 0;
      }


      public  int GetHashCode(Employee e)
      {
         return e.GetHashCode( );
      }

      // Tell the Employee objects to compare themselves
      public int Compare( Employee lhs, Employee rhs )
      {
          return lhs.CompareTo( rhs, WhichComparison );
      }

      public Employee.EmployeeComparer.ComparisonType
         WhichComparison
      {
         get{return whichComparison;}
         set{whichComparison = value;}
      }


   }
}
public class Tester
{
   static void Main( )
```

Example 14-6. Sorting an array by employees' IDs and years of service (continued)

```
{
   List<Employee> empArray = new List<Employee>( );

   // generate random numbers for
   // both the integers and the
   // employee id's
   Random r = new Random( );

   // populate the array
   for ( int i = 0; i < 5; i++ )
   {
      // add a random employee id

      empArray.Add(
         new Employee(
            r.Next( 10 ) + 100, r.Next( 20 )

         )
      );
   }

   // display all the contents of the Employee array
   for ( int i = 0; i < empArray.Count; i++ )
   {
      Console.Write( "\n{0} ", empArray[i].ToString( ) );
   }
   Console.WriteLine( "\n" );


   // sort and display the employee array
   Employee.EmployeeComparer c = Employee.GetComparer( );
   c.WhichComparison =
   Employee.EmployeeComparer.ComparisonType.EmpID;
   empArray.Sort( c );

   // display all the contents of the Employee array
   for ( int i = 0; i < empArray.Count; i++ )
   {
      Console.Write( "\n{0} ", empArray[i].ToString( ) );
   }
   Console.WriteLine( "\n" );


   c.WhichComparison = Employee.EmployeeComparer.ComparisonType.Yrs;
   empArray.Sort( c );

   for ( int i = 0; i < empArray.Count; i++ )
   {
      Console.Write( "\n{0} ", empArray[i].ToString( ) );
```

Example 14-6. Sorting an array by employees' IDs and years of service (continued)

```
        }
        Console.WriteLine( "\n" );


      }
   }
}
```

The output looks like this for one run of the program:

```
ID: 103. Years of Svc: 11
ID: 108. Years of Svc: 15
ID: 107. Years of Svc: 14
ID: 108. Years of Svc: 5
ID: 102. Years of Svc: 0

ID: 102. Years of Svc: 0
ID: 103. Years of Svc: 11
ID: 107. Years of Svc: 14
ID: 108. Years of Svc: 15
ID: 108. Years of Svc: 5

ID: 102. Years of Svc: 0
ID: 108. Years of Svc: 5
ID: 103. Years of Svc: 11
ID: 107. Years of Svc: 14
ID: 108. Years of Svc: 15
```

The first block of output shows the `Employee` objects as they are added to the `List`. The employee ID values and the years of service are in random order. The second block shows the results of sorting by the employee ID, and the third block shows the results of sorting by years of service.

Generic Queues

A *queue* represents a first-in, first-out (FIFO) collection. The classic analogy is to a line (or queue, if you are British) at a ticket window. The first person in line ought to be the first person to come off the line to buy a ticket.

A queue is a good collection to use when you are managing a limited resource. For example, you might want to send messages to a resource that can only handle one message at a time. You would then create a message queue so that you can say to your clients: "Your message is important to us. Messages are handled in the order in which they are received."

The `Queue` class has a number of member methods and properties, the most important of which are shown in Table 14-3.

Table 14-3. Queue methods and properties

Method or property	Purpose
`Count`	Public property that gets the number of elements in the `Queue`
`Clear( )`	Removes all objects from the `Queue`
`Contains( )`	Determines if an element is in the `Queue`
`CopyTo( )`	Copies the `Queue` elements to an existing one-dimensional array
`Dequeue( )`	Removes and returns the object at the beginning of the `Queue`
`Enqueue( )`	Adds an object to the end of the `Queue`
`GetEnumerator( )`	Returns an enumerator for the `Queue`
`Peek( )`	Returns the object at the beginning of the `Queue` without removing it
`ToArray( )`	Copies the elements to a new array

Add elements to your queue with the `Enqueue` command and take them off the queue with `Dequeue` or by using an enumerator, as shown in Example 14-7.

Example 14-7. Working with a queue

```
using System;
using System.Collections.Generic;

namespace Queue
{
   public class Tester
   {

      static void Main( )
      {

         Queue<Int32> intQueue = new Queue<Int32>( );


         // populate the array
         for ( int i = 0; i < 5; i++ )
         {

            intQueue.Enqueue( i * 5 );

         }

         // Display the Queue.
         Console.Write( "intQueue values:\t" );
         PrintValues( intQueue );

         // Remove an element from the Queue.
         Console.WriteLine(
            "\n(Dequeue)\t{0}", intQueue.Dequeue( ) );
```

Example 14-7. Working with a queue (continued)

```
        // Display the Queue.
        Console.Write( "intQueue values:\t" );
        PrintValues( intQueue );

        // Remove another element from the Queue.
        Console.WriteLine(
           "\n(Dequeue)\t{0}", intQueue.Dequeue( ) );

        // Display the Queue.
        Console.Write( "intQueue values:\t" );
        PrintValues( intQueue );

        // View the first element in the
        // Queue but do not remove.
        Console.WriteLine(
           "\n(Peek)   \t{0}", intQueue.Peek( ) );


        // Display the Queue.
        Console.Write( "intQueue values:\t" );
        PrintValues( intQueue );

     }

     public static void PrintValues(IEnumerable<Int32> myCollection)
     {
        IEnumerator<Int32> myEnumerator =
           myCollection.GetEnumerator( );
        while ( myEnumerator.MoveNext( ) )
           Console.Write( "{0} ", myEnumerator.Current );
        Console.WriteLine( );
     }

  }
}
```

The output looks like this:

```
intQueue values:       0 5 10 15 20

(Dequeue)       0
intQueue values:       5 10 15 20

(Dequeue)       5
intQueue values:       10 15 20

(Peek)          10
intQueue values:       10 15 20
```

I've dispensed with the `Employee` class to save room, but of course you can enqueue user-defined objects as well. The output shows that queuing objects adds them to the `Queue`, while `Dequeue( )` returns the object and also removes them from the `Queue`. The

Queue class also provides a Peek( ) method that allows you to see, but not remove, the next element.

Because the Queue class is enumerable, you can pass it to the PrintValues( ) method, which is provided as an IEnumerable interface. The conversion is implicit. In the PrintValues method, you call GetEnumerator, which you will remember is the single method of all IEnumerable classes. This returns an IEnumerator, which you then use to enumerate all the objects in the collection.

Generic Stacks

A *Stack* is a last-in, first-out (LIFO) collection, like a stack of dishes at a buffet table, or a stack of coins on your desk. A dish added on top, is the first dish you take off the stack.

The principal methods for adding to and removing from a stack are Push( ) and Pop( ); Stack also offers a Peek( ) method, very much like Queue. The significant methods and properties for Stack are shown in Table 14-4.

Table 14-4. Stack methods and properties

Method or property	Purpose
Count	Public property that gets the number of elements in the Stack
Clear( )	Removes all objects from the Stack
Contains( )	Determines if an element is in the Stack
CopyTo( )	Copies the Stack elements to an existing one-dimensional array
GetEnumerator( )	Returns an enumerator for the Stack
Peek( )	Returns the object at the top of the Stack without removing it
Pop( )	Removes and returns the object at the top of the Stack
Push( )	Inserts an object at the top of the Stack
ToArray( )	Copies the elements to a new array

The List, Queue, and Stack types contain multiple versions of the CopyTo( ) and ToArray( ) methods for copying their elements to an array. In the case of a Stack, the CopyTo( ) method will copy its elements to an existing one-dimensional array, overwriting the contents of the array beginning at the index you specify. The ToArray( ) method returns a new array with the contents of the Stack's elements. Example 14-8 illustrates several of the Stack methods.

Example 14-8. Working with a Stack

```
using System;
using System.Collections.Generic;

namespace Stack
{
```

Example 14-8. Working with a Stack (continued)

```
public class Tester
{
   static void Main( )
   {
      Stack<Int32> intStack = new Stack<Int32>( );

      // populate the array

      for ( int i = 0; i < 8; i++ )
      {
         intStack.Push( i * 5 );
      }


      // Display the Stack.
      Console.Write( "intStack values:\t" );
      PrintValues( intStack );

      // Remove an element from the Stack.
      Console.WriteLine( "\n(Pop)\t{0}",
      intStack.Pop( ) );

      // Display the Stack.
      Console.Write( "intStack values:\t" );
      PrintValues( intStack );

      // Remove another element from the Stack.
      Console.WriteLine( "\n(Pop)\t{0}",
         intStack.Pop( ) );

      // Display the Stack.
      Console.Write( "intStack values:\t" );
      PrintValues( intStack );

      // View the first element in the
      // Stack but do not remove.
      Console.WriteLine( "\n(Peek)   \t{0}",
         intStack.Peek( ) );


      // Display the Stack.
      Console.Write( "intStack values:\t" );
      PrintValues( intStack );

      // Declare an array object which will
      // hold 12 integers
      int[] targetArray = new int[12];

      for (int i = 0; i < targetArray.Length; i++)
      {
          targetArray[i] = i * 100 + 100;
      }
```

Example 14-8. Working with a Stack (continued)

```
        // Display the values of the target Array instance.
        Console.WriteLine( "\nTarget array:  " );
        PrintValues( targetArray );

        // Copy the entire source Stack to the
        // target Array instance, starting at index 6.
        intStack.CopyTo( targetArray, 6 );


        // Display the values of the target Array instance.
        Console.WriteLine( "\nTarget array after copy:  " );
        PrintValues( targetArray );


     }


     public static void PrintValues(
        IEnumerable<Int32> myCollection )
     {
        IEnumerator<Int32> enumerator =
           myCollection.GetEnumerator( );
        while ( enumerator.MoveNext( ) )
           Console.Write( "{0}  ", enumerator.Current );
        Console.WriteLine( );
     }
  }
}
```

The output looks like this:

```
intStack values:        35  30  25  20  15  10  5  0

(Pop)   35
intStack values:        30  25  20  15  10  5  0

(Pop)   30
intStack values:        25  20  15  10  5  0

(Peek)          25
intStack values:        25  20  15  10  5  0

Target array:
100  200  300  400  500  600  700  800  900  1000  1100  1200

Target array after copy:
100  200  300  400  500  600  25  20  15  10  5  0

The new  array:
25  20  15  10  5  0
```

The output reflects that the items pushed onto the Stack were popped in reverse order.

The effect of CopyTo( ) can be seen by examining the target array before and after calling CopyTo( ). The array elements are overwritten beginning with the index specified (6).

Dictionaries

A *dictionary* is a collection that associates a *key* to a *value*. A language dictionary, such as Webster's, associates a word (the key) with its definition (the value).

To see the value of dictionaries, start by imagining that you want to keep a list of the state capitals. One approach might be to put them in an array:

```
string[] stateCapitals = new string[50];
```

The stateCapitals array will hold 50 state capitals. Each capital is accessed as an offset into the array. For example, to access the capital for Arkansas, you need to know that Arkansas is the fourth state in alphabetical order:

```
string capitalOfArkansas = stateCapitals[3];
```

It is inconvenient, however, to access state capitals using array notation. After all, if I need the capital for Massachusetts, there is no easy way for me to determine that Massachusetts is the 21st state alphabetically.

It would be far more convenient to store the capital with the state name. A dictionary allows you to store a value (in this case, the capital) with a key (in this case, the name of the state).

A .NET Framework dictionary can associate any kind of key (string, integer, object) with any kind of value (string, integer, object). Typically, the key is fairly short and the value fairly complex, though in this case, we'll use short strings for both.

The most important attributes of a good dictionary are that it is easy to add values and it is quick to retrieve values. Table 14-5 lists some of the more important methods and properties of the dictionary.

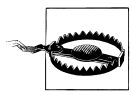
Table 14-5. Dictionary methods and properties

Method or property	Purpose
Count	Public property that gets the number of elements in the Dictionary.
Item( )	The indexer for the Dictionary.
Keys	Public property that gets a collection containing the keys in the Dictionary. (See also Values, later in this table.)
Values	Public property that gets a collection containing the values in the Dictionary. (See also Keys, earlier in this table.)
Add( )	Adds an entry with a specified Key and Value.
Clear( )	Removes all objects from the Dictionary.
ContainsKey( )	Determines whether the Dictionary has a specified key.

Table 14-5. Dictionary methods and properties (continued)

Method or property	Purpose
`ContainsValue( )`	Determines whether the `Dictionary` has a specified value.
`GetEnumerator( )`	Returns an enumerator for the `Dictionary`.
`Remove( )`	Removes the entry with the specified `Key`.

The key in a `Dictionary` can be a primitive type, or it can be an instance of a user-defined type (an object).



Objects used as keys for a `Dictionary` must implement `GetHashCode( )` as well as `Equals`. In most cases, you can simply use the inherited implementation from `Object`.

Dictionaries implement the `IDictionary<TKey,TValue>` interface (where `TKey` is the key type and `TValue>` is the value type). `IDictionary` provides a public property `Item`. The `Item` property retrieves a value with the specified key.

The `Item` property is implemented with the index operator (`[]`). Thus, you access items in any `Dictionary` object using the offset syntax, as you would with an array.

Example 14-9 demonstrates adding items to a `Dictionary` and then retrieving them with the indexer (which implicitly calls the `Dictionary`'s `Item` property).

Example 14-9. The Item property as offset operators

```
using System;
using System.Collections.Generic;

namespace Dictionary
{
   public class Tester
   {
      static void Main( )
      {
         // Create and initialize a new Dictionary.
         Dictionary<string, string> dict =
           new Dictionary<string, string>( );

      dict.Add("Alabama", "Montgomery");
      dict.Add("Alaska", "Juneau");
      dict.Add("Arizona", "Phoenix");
      dict.Add("Arkansas", "Little Rock");
      dict.Add("California", "Sacramento");
      dict.Add("Colorado", "Denver");
      dict.Add("Connecticut", "Hartford");
      dict.Add("Delaware", "Dover");
      dict.Add("Florida", "Tallahassee");
      dict.Add("Georgia", "Atlanta");
      dict.Add("Hawaii", "Honolulu");
```

Example 14-9. The Item property as offset operators (continued)

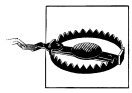
```
        dict.Add("Idaho", "Boise");
        dict.Add("Illinois", "Springfield");
        dict.Add("Indiana", "Indianapolis");
        dict.Add("Iowa", "Des Moines");
        dict.Add("Kansas", "Topeka");
        dict.Add("Kentucky", "Frankfort");
        dict.Add("Louisiana", "Baton Rouge");
        dict.Add("Maine", "Augusta");
        dict.Add("Maryland", "Annapolis");
        dict.Add("Massachusetts", "Boston");
        dict.Add("Michigan", "Lansing");
        dict.Add("Minnesota", "St. Paul");
        dict.Add("Mississippi", "Jackson");
        dict.Add("Missouri", "Jefferson City");
        dict.Add("Montana", "Helena");
        dict.Add("Nebraska", "Lincoln");
        dict.Add("Nevada", "Carson City");
        dict.Add("New Hampshire", "Concord");
        dict.Add("New Jersey", "Trenton");
        dict.Add("New Mexico", "Santa Fe");
        dict.Add("New York", "Albany");
        dict.Add("North Carolina", "Raleigh");
        dict.Add("North Dakota", "Bismarck");
        dict.Add("Ohio", "Columbus");
        dict.Add("Oklahoma", "Oklahoma City");
        dict.Add("Oregon", "Portland");
        dict.Add("Pennsylvania", "Harrisburg");
        dict.Add("Rhode Island", "Providence");
        dict.Add("South Carolina", "Columbia");
        dict.Add("South Dakota", "Pierre");
        dict.Add("Tennessee", "Nashville");
        dict.Add("Texas", "Austin");
        dict.Add("Utah", "Salt Lake City");
        dict.Add("Vermont", "Montpelier");
        dict.Add("Virginia", "Richmond");
        dict.Add("Washington", "Olympia");
        dict.Add("West Virginia", "Charleston");
        dict.Add("Wisconsin", "Madison");
        dict.Add("Wyoming", "Cheyenne");

        // access a state
        Console.WriteLine( "The capital of Massachusetts is {0}",
          dict["Massachusetts"] );
        }      // end main
    }         // end class
}             // end namespace
```

The output looks like this:

```
The capital of Massachusetts is Boston
```

Example 14-9 begins by instantiating a new `Dictionary` object with the type of the key and of the value declared to be string.

We add 50 key/value pairs. In this example, the state name is the key and the capital is the value (though, in a typical dictionary, the value is almost always larger than the key).



You must not change the value of the key object once you use it in a dictionary.

Summary

- The .NET Framework provides a number of type-safe (generic) collections, including the `list<t>`, `stack<t>`, `queue<t>`, and `dictionary<key><value>`.
- You are free to create your own generic collection types as well.
- Generics allow the collection designer to create a single collection without regard to the type of object it will hold, but to allow the collection *user* to define, at compile time, which type the object will hold. This enlists the compiler in finding bugs; if you add an object of the wrong type to a collection, it will be found at compile time, not run time. It also eliminates the need for casting and for boxing and unboxing.
- The .NET Framework provides a number of interfaces that collections must implement if they wish to act like the built-in collections (such as being iterated by a `foreach` loop).
- An indexer allows you to access or assign objects to and from a collection just as you do with an array (for example, `myCollection[5] = "hello"`).
- Indexers need not be restricted to integers. It is common to create indexers that take a string to assign or retrieve a value.
- All framework collections implement the `Sort( )` method. If you want to be able to sort a collection of objects of a user-defined type, however, the defining class must implement the `IComparable` interface.
- The generic list collection, `List<T>`, works like an array whose size is increased dynamically as you add elements.
- The `Queue<T>` class is a first-in, first-out collection.
- The `Stack<T>` class is a last-in, first-out collection.
- A `dictionary<k,v>` is a collection that associates a key with a value. Typically, the key is short, and the value is large.

Quiz

Question 14-1. What is the convention for naming an indexer?

Question 14-2. What types can be used in an indexer to index a collection?

Question 14-3. What are the preconditions for calling `Sort( )` on an array?

Question 14-4. What is the purpose of generics?

Question 14-5. What is the `IEnumerable<T>` interface?

Question 14-6. What is the principal difference between a `List<T>` and an array?

Question 14-7. What is the difference between a `List`, a `Stack`, a `Queue`, and a `Dictionary`?

Exercises

Exercise 14-1. Create an abstract `Animal` class that has private members weight and name, and abstract methods `Speak( )`, `Move( )`, and `ToString( )`. Derive from `Animal` a `Cat` and `Dog` class that override the methods appropriately. Create an `Animal` array, populate it with `Dogs` and `Cats`, and then call each member's overridden virtual method.

Exercise 14-2. Replace the array in Exercise 14-1 with a `List`. Sort the animals by size. You can simplify by just calling `ToString( )` before and after the sort. Remember that you'll need to implement `IComparable`.

Exercise 14-3. Replace the list from Exercise 14-2 with both a `Stack` and a `Queue`, and see the difference in the order in which the animals are returned.

Exercise 14-4. Rewrite Exercise 14-2 to allow `Animals` to be sorted either by weight or alphabetically by name.