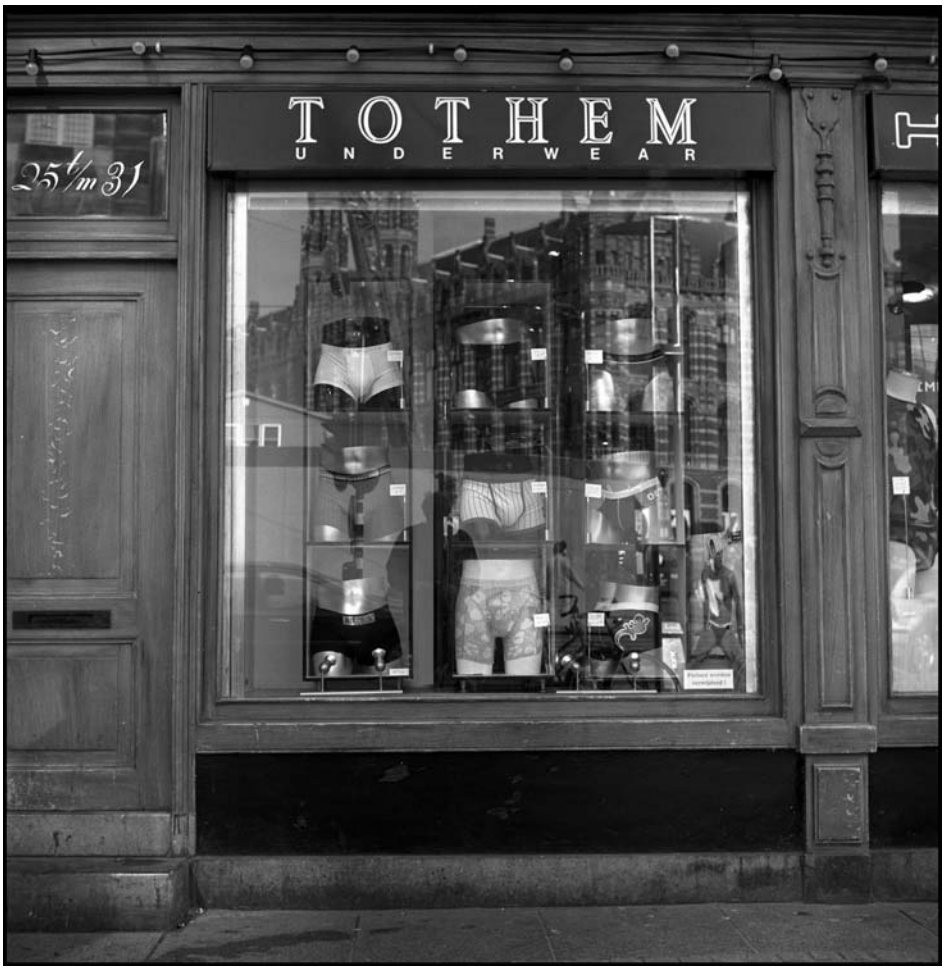
Chapter 6



Yashica Mat 124G

Underwear Window

Lists

Learning Objectives

- *Describe the features of the ArrayList class*
- *Describe the features of the List<T> class*
- *Describe the features of the LinkedList<T> class*
- *Add elements to a list using the Add() method*
- *Access individual elements of a list using array indexer notation*
- *Apply casting to objects retrieved from an ArrayList*
- *Use the façade software design pattern to make an ArrayList type safe*
- *Sort the contents of a list using the natural ordering of contained objects*
- *Reverse the contents of a list*
- *State the functionality provided by the IList, ICollection, and IEnumerable interfaces*
- *State the functionality provided by the IList<T>, ICollection<T>, and IEnumerable<T> interfaces*

Introduction

Lists are perhaps the most often used collection types of which there are two primary varieties: array-based lists, and linked lists.

As their name implies, lists store their contents in a sequentially accessible fashion, but there's a big difference between the behavior of an array-based list and a linked list. In this chapter I will explain the difference between these two list types and give you a glimpse into the inner workings of each.

I've already introduced you to the ArrayList and its generic relative, List<T>, in chapter 1. It's my intention here to dive deeper into the operations of lists and expand your repertoire by introducing you to the generic LinkedList<T> collection.

Many times I am asked, "How do array-based lists expand automatically to hold additional elements?" I will answer this question with a short demo program that shows how an array-based collection dynamically resizes itself to enlarge its capacity. I will also show you how linked lists operate, complete with sample code showing how individual list nodes are inserted and deleted, and how items on the list are located. In this regard, this chapter doubles as a short course in data structures.

Armed with a deeper understanding of these concepts, you'll be better able to select the collection class most appropriate to your particular programming situation.

Array-Based List Collections – How They Work

Arrays serve as the foundational data structure for array-based collections. In this section I will show you how an array-based collection automatically grows to accommodate the addition or insertion of more objects than its initial capacity allows. This dynamic growth capability is a primary performance characteristic of array-based lists that you must take into account when you use one in your code, especially if you plan to manipulate lists with large numbers of elements. Let's look at a home-grown, array-based collection class to demonstrate the dynamic resizing capability.

A Home-Grown Dynamic Array

Imagine for a moment that you are working on a project and you're deep into the code. You're in the flow and you don't want to stop to read no stinkin' API documentation. The problem at hand dictates the need for an array with special powers — one that can automatically grow itself when one too many elements are inserted. To solve your problem, you hastily crank out the code for a class named DynamicArray shown in Example 6.1.

6.1 DynamicArray.cs

```
1    using System;
2
3    public class DynamicArray {
4      private Object[] _object_array = null;
5      private int _next_open_element = 0;
6      private int _growth_increment = 10;
7      private const int INITIAL_SIZE = 25;
8
9      public int Count {
10        get { return _next_open_element; }
11     }
12
13     public object this[int index] {
14        get {
15          if((index >= 0) && (index < _object_array.Length)){
16             return _object_array[index];
17           }else throw new ArgumentOutOfRangeException();
18        }
19        set {
20         if(_next_open_element < _object_array.Length){
21                 _object_array[_next_open_element++] = value;
22              }else{
23            GrowArray();
24            _object_array[_next_open_element++] = value;
25        }
26        }
```

```
27      }
28
29     public DynamicArray(int size){
30       _object_array = new Object[size];
31     }
32
33     public DynamicArray():this(INITIAL_SIZE){ }
34
35     public void Add(Object o){
36       if(_next_open_element < _object_array.Length){
37          _object_array[_next_open_element++] = o;
38       }else{
39      GrowArray();
40      _object_array[_next_open_element++] = o;
41        }
42     } // end add() method;
43
44     private void GrowArray(){
45       Object[] temp_array = _object_array;
46       _object_array = new Object[_object_array.Length + _growth_increment];
47       for(int i=0, j=0; i<temp_array.Length; i++){
48         if(temp_array[i] != null){
49           _object_array[j++] = temp_array[i];
50          }
51         _next_open_element = j;
52        }
53        temp_array = null;
54     } // end growArray() method
55
56
57   } // end DynamicArray class definition
```

Referring to Example 6.1 — the data structure used as the basis for the DynamicArray class is an ordinary array of objects. Its initial size can be set via a constructor or, if the default constructor is called, the initial size is set to 25 elements. Its growth increment is 10 elements, meaning that when the time comes to grow the array, it will expand by 10 elements. In addition to its two constructors, the DynamicArray class has one property named Count, two additional methods named Add() and GrowArray(), and a class indexer member that starts on line 13. An indexer is a member that allows an object to be indexed the same way as an array.

The Add() method inserts an object reference into the next available array element pointed to by the _next_open_element variable. If the array is full, the GrowArray() method is called to grow the array. The GrowArray() method creates a temporary array of objects and copies each element to the temporary array. It then creates a new larger array and copies the elements into it from the temporary array.

The indexer member whose definition begins on line 13 allows you to access each element of the array. (**Note:** The array itself is private and therefore encapsulated, thus the need for the public indexer member to control access to the array.) If the index argument falls out of bounds, the indexer throws an ArgumentOutOfRangeException. The Count property simply returns the number of elements (references) contained in the array, which is the value of the _next_open_element variable. Example 6.2 shows the DynamicArray class in action.

6.2 ArrayTestApp.cs

```
1    using System;
2
3    public class ArrayTestApp {
4      public static void Main(){
5        DynamicArray da = new DynamicArray();
6        Console.WriteLine("The array contains " + da.Count + " objects.");
7        da.Add("Ohhh if you loved C# like I love C#!!");
8        Console.WriteLine(da[0].ToString());
9        for(int i = 1; i<26; i++){
10         da.Add(i);
11       }
12       Console.WriteLine("The array contains " + da.Count + " objects.");
13       for(int i=0; i<da.Count; i++){
14         if(da[i] != null){
15            Console.Write(da[i].ToString() + ", ");
16           if((i%20)==0){
17             Console.WriteLine();
18            }
19         }
20       }
21       Console.WriteLine();
22     }//end Main() method
23   }// end ArrayTestApp class definition
```

Referring to Example 6.2 — on line 5 an instance of DynamicArray is created using the default constructor. This results in an initial internal array length of 25 elements. Initially, its Count is zero because no references have yet

been inserted. On line 7 a string object is added to the array and then printed to the console on line 8. The `for` statement on line 9 inserts enough integers to test the array's growth capabilities. The `for` statement on line 13 prints all the non-null elements to the console. Figure 6-1 shows the results of running this program.

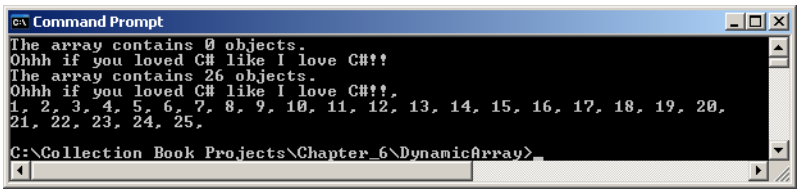


Figure 6-1: Results of Testing DynamicArray

Evaluating DynamicArray

The DynamicArray class works well enough for your immediate needs but it suffers several shortcomings that will cause serious problems should you try to use it in more demanding situations. For example, although you can access each element of the array, you cannot remove elements. You could add a method called Remove(), but what happens when the number of remaining elements falls below a certain threshold? You might want to shrink the array as well.

Another point to consider is how to insert references into specific element locations. When this happens, you must make room for the reference at the specified array index location and shift the remaining elements to the right. If you plan to frequently insert elements into your custom-built DynamicArray class, you will have a performance issue on your hands you did not foresee.

At this point, you would be well served to take a break from coding and dive into the API documentation to study up on the collections framework. There you will find that all this work, and more, is already done for you!

The ArrayList Class To The Rescue

Let's re-write the ArrayTestApp program with the help of the ArrayList class, which belongs to the .NET collections framework. Example 6.3 gives the code.

6.3 ArrayTestApp.cs (Mod 1)

```
1    using System;
2    using System.Collections;
3
4    public class ArrayTestApp {
5      public static void Main(){
6        ArrayList da = new ArrayList();
7        Console.WriteLine("The array contains " + da.Count + " objects.");
8        da.Add("Ohhh if you loved C# like I love C#!!");
9        Console.WriteLine(da[0].ToString());
10       for(int i = 1; i<26; i++){
11         da.Add(i);
12       }
13       Console.WriteLine("The array contains " + da.Count + " objects.");
14       for(int i=0; i<da.Count; i++){
15         if(da[i] != null){
16            Console.Write(da[i].ToString() + ", ");
17           if((i%20)==0){
18             Console.WriteLine();
19           }
20         }
21       }
22       Console.WriteLine();
23     }//end Main() method
24   }// end ArrayTestApp class definition
```

Referring to Example 6.3 — I made only three changes to the original ArrayTestApp program: 1) I added another `using` directive on line 2 to provide access to the System.Collections namespace, 2) I changed the `da` reference declared on line 6 from a DynamicArray type to an ArrayList type, and 3) also on line 6, I created an instance of ArrayList instead of an instance of DynamicArray. Figure 6-2 shows the results of running this program.

If you compare figures 6-1 and 6-2 you will see that the output produced with an ArrayList is exactly the same as that produced using the DynamicArray. However, the ArrayList class provides much more ready-made functionality.

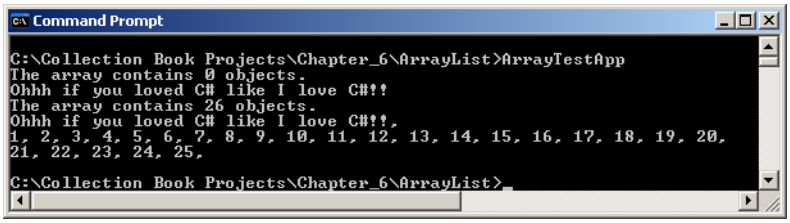


Figure 6-2: Results of Running Example 6.3

You might be asking yourself, "Why does this code work?" As it turns out, I gamed the system just a little bit. The DynamicArray class presented in Example 6.1 just happens to partially and informally implement the IList interface. In other words, my DynamicArray class defines a subset of the properties and methods defined by the IList interface, which is implemented by the ArrayList class. Later, in Example 6.3, when I changed the type of the `da` reference from DynamicArray to ArrayList and used the ArrayList collection class, everything worked fine.

Quick Review

Array-based lists, as their name implies, feature arrays as their fundamental data structure. Array-based lists are created with an initial capacity and can grow in size automatically to accommodate additional elements.

The Non-Generic ArrayList: Objects In – Objects Out

In this section I want to dive a bit deeper into the operation of an ArrayList collection class. I'll start with a discussion of the ArrayList's inheritance hierarchy and explain the functionality provided by each implemented interface. Following that, I'll show you how to provide a measure of type safety when using an ArrayList collection through the use of the façade software pattern.

ArrayList Inheritance Hierarchy

Figure 6-3 offers a class diagram showing the inheritance hierarchy of the ArrayList collection class.

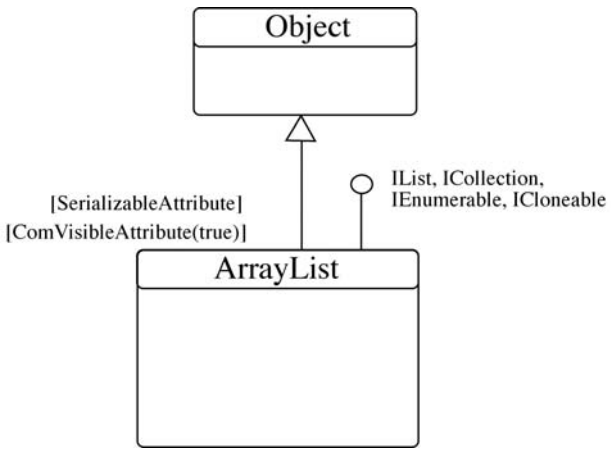


Figure 6-3: ArrayList Inheritance Hierarchy

Referring to figure 6-3 — the ArrayList class implicitly extends the Object class and implements the following interfaces: IList, ICollection, IEnumerable, and ICloneable. Additionally, the ArrayList class is tagged with two attributes: SerializableAttribute and ComVisibleAttribute(true).

An interface in C# may extend another interface and it will be helpful here to see the inheritance diagram for the ArrayList class drawn another way. Figure 6-4 offers an alternative UML class diagram of the ArrayList class.

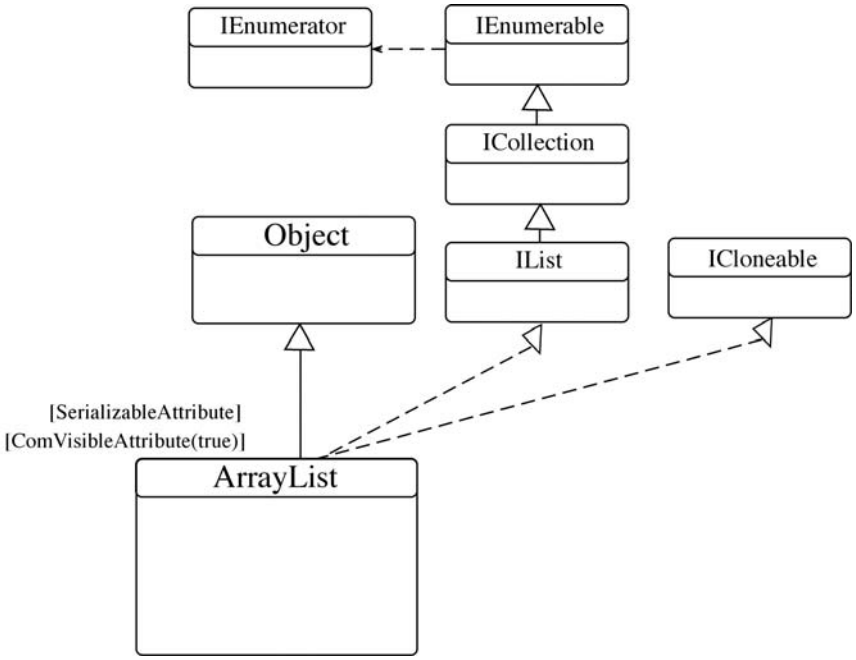


Figure 6-4: Expanded ArrayList Inheritance Hierarchy

Referring to figure 6-4 — the IList interface extends the ICollection interface, which in turn extends IEnumerable. Thus, any class that implements the IList interface is also implementing ICollection and IEnumerable. The IEnumerable interface has a dependency on the IEnumerator interface. The following sections explain the functionality provided by each of these interfaces and attributes.

Functionality Provided by the IEnumerable and IEnumerator Interfaces

Together, the IEnumerable and IEnumerator interfaces implement the *iterator* software design pattern in the .NET Framework. (See Eric Gamma, et. al., in the References section.) An iterator is an object that enables the standardized sequential traversal of the contents of a collection regardless of that collection's underlying structure. In other words, the iterator software pattern allows you to write polymorphic code that can step through the elements of a collection without you needing to worry about the messy details of how the collection is actually organized.

The IEnumerable interface declares one method named GetEnumerator() which returns the IEnumerator object for that particular collection. The IEnumerator object is then used to traverse the collection in a particular direction, with forwards, beginning with the first element, being the default implementation. However, you don't use the IEnumerator object directly; it's meant to be used with the `foreach` statement.

Collection classes are free to overload the GetEnumerator() method to provide additional ways of traversing the list. The ArrayList collection provides the GetEnumerator(int, int) method which allows the traversal of a range of elements contained in the list. When manually interacting with an IEnumerator object, you have access to two methods: MoveNext() and Reset(), and one property: Current.

When traversing a list, or any collection, with the help of an iterator, you generally say you're "iterating over the collection." For example, if a colleague were to stop by your office and find you writing a `foreach` statement, and he asked you what on earth you were doing, you'd reply, "Why, I'm iterating over a collection!"

Example 6.4 demonstrates three ways of iterating over a list.

6.4 EnumeratorDemo.cs

```
1    using System;
2    using System.Collections;
3
4    public class EnumeratorDemo {
5
6      public static void Main(){
7        ArrayList list = new ArrayList();
8        list.Add(1);
```



```
9         list.Add(2);
10        list.Add(3);
11        list.Add(4);
12
13        // Iterating over the list in the manner of old habits
14        for(int i = 0; i<list.Count; i++){
15          Console.Write(i + " ");
16        }
17        Console.WriteLine();
18
19        // Iterating over the list whilst ignoring the messy details
20        foreach(int i in list){
21          Console.Write(i + " ");
22        }
23        Console.WriteLine();
24
25        // Iterating over a list segment using overloaded GetEnumerator( ) method
26        // and directly manipulating the IEnumerator object via  IEnumerator.MoveNext()
27        IEnumerator e = list.GetEnumerator(1, 2);
28        while(e.MoveNext()){
29          Console.Write(e.Current + " ");
30        }
31      }
32    }
```

Referring to example 6.4 — a reference to an ArrayList is declared and initialized on line 7, followed by four consecutive calls to its Add() method. On line 14 a traditional `for` loop is used to iterate over the list and write each element to the console. I call this method "iterating over the list in the manner of old habits!" And quite frankly, when it comes to lists, it's a hard habit to break, but it does have its advantages, as you'll see later.

On line 19 I use the `foreach` statement to iterate over the list elements. The `foreach` uses the default implementation of the GetEnumerator() method, which allows the traversal of the collection in the forward direction. The only way to use the overloaded GetEnumerator(int, int) method to traverse the list is to directly manipulate the IEnumerator object using the MoveNext() method and accessing each element via the Current property as is demonstrated beginning on line 27.

Figure 6-5 shows the results of running example 6.4.

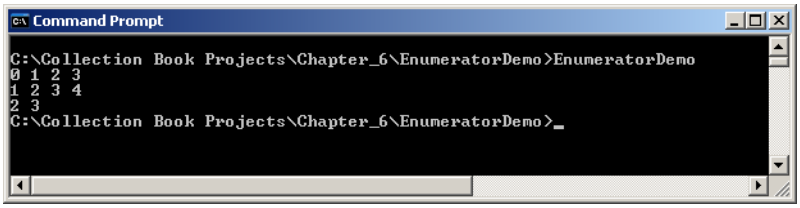


Figure 6-5: Results of Running Example 6.4

Collection Elements Cannot Be Modified When Using An Enumerator

Referring again to example 6.4 — the critically important difference between using the `for` loop vs. the `foreach` statement to iterate over the list is that you can modify the list elements in the body of the `for` loop but not in the body of the `foreach` loop nor in the body of the `while` loop when directly manipulating the enumerator. **The use of an enumerator to iterate over a collection results in a read-only sequence of elements.** Any attempt to modify the collection's elements, either by you or by another thread of execution, while stepping through the collection's elements with the enumerator, invalidates the enumerator and will result in an InvalidOperationException.

Where To Go From Here

For a detailed treatment of custom implementation of the IEnumerable and IEnumerator interfaces, Iterators, Iterator blocks, and named Iterators, please refer to Chapter 18: Creating Custom Collections. Collections and thread safety is covered in Chapter 14: Collections and Threads.

Functionality Provided by the ICollection Interface

The ICollection interface extends IEnumerable and serves as the base interface for all non-generic collection classes. (i.e., All collections classes found in the System.Collections namespace.)

In addition to those methods declared by the IEnumerable interface, the ICollection interface provides the CopyTo() method, which is used to copy the elements of the collection to a single-dimensional array, a feature you'll find to be quite handy on occasion. The ICollection interface also provides the Count, IsSynchronized, and SyncRoot properties. The Count property returns the number of elements contained in the collection. The IsSynchronized and SyncRoot properties are used to coordinate multithreaded access to the collection and are discussed in detail in Chapter 14: Collections and Threads.

Functionality Provided by the IList Interface

The IList interface adds numerous methods and properties but most importantly it adds an *indexer* that is used to access each element in the collection by an index, just like an array. In fact, as you learned in Chapter 4, arrays in the .NET Framework implement the IList interface. (**Note:** The indexer is listed in the Properties section of the MSDN documentation as Item.)

Functionality Provided by the ICloneable Interface

The ICloneable interface declares one method: Clone(). The Clone() method is used to create an exact copy of an existing collection. If you're creating a custom collection and you intend to implement the ICloneable interface, you'll need to be aware of the differences between a shallow copy and a deep copy. I discuss these concepts in detail in Chapter 18: Creating Custom Collections.

Functionality Provided by the SerializableAttribute

The SerializableAttribute informs the .NET runtime that the collection can be serialized. To serialize something means to convert it into a form that can be transmitted across a network or persisted to disk. Serialization is discussed in detail in Chapter 17: Collections and I/O.

The important thing to know about serialization at this point is that all objects contained within an ArrayList must be tagged as Serializable in order for a serialization operation on the list to succeed. If the list contains only types found in the .NET Framework then you're safe; it's with custom coded types you need most concern yourself.

Functionality Provided by the ComVisibleAttribute(true)

The ComVisibleAttribute is used to control the visibility of a class and its members to the Component Object Model (COM). By default, all public types and their public members are visible to COM. I do not cover COM programming in this book so that's the last you'll hear about COM. If you would like to learn more about COM programming, I recommend the excellent book *Essential COM* by Don Box. (See the References section.)

Extension Methods

C# 3.0 introduced numerous enhancement to the language, one of them being extension methods. An extension method is a static method that defines an operation on and extends the functionality of an existing type, without the need to formally extend the type you wish to enhance via normal inheritance. The new extension method can be used on the target type in the same way as an ordinary instance method.

The ArrayList has three extension methods: AsQueryable(), defined by the System.Linq.Queryable class, Cast<TResult>() and OfType<TResult>(), which are defined by the System.Linq.Enumerable class. I will discuss the use of these methods with the need arises. I discus the use and creation of extension methods in detail in Chapter 18: Creating Custom Collections.

Defensive Coding Using the Façade Software Pattern

The non-generic ArrayList collection, found in the System.Collections namespace, can contain any type of object: objects in — objects out, but his flexibility comes at a price. If you store a mixture of object types in an ArrayList collection, you must, if you're thinking about accessing interface members other than those specified by the Object class, keep track of such types so as not to throw an exception in your code. It's often desirable when programming with non-generic, old-school collections, to create a type-safe collection that only allows the insertion of a specific type of object.

There are several approaches you can take to creating a custom list collection that enforces type-safety. First, you could extend the ArrayList class and override all the public members it exposes. Note that you would need to override all such members because a failure to do so would expose the base ArrayList members not overridden. A second approach would be to extend the System.Collections.CollectionBase class. This is actually the approach recommended by Microsoft when you need to create a strongly-typed collection, but I will postpone its discussion until I formally cover custom collections in Chapter 18: Creating Custom Collections.

A third approach, and one that's fairly easy to implement, is to create a façade or wrapper class that contains an ArrayList and provides implementations for just those methods you need. Figure 6-6 gives the UML class diagram for a custom collection named DogList, which uses the façade software design pattern to encapsulate an ArrayList collection and provide a measure of type safety.

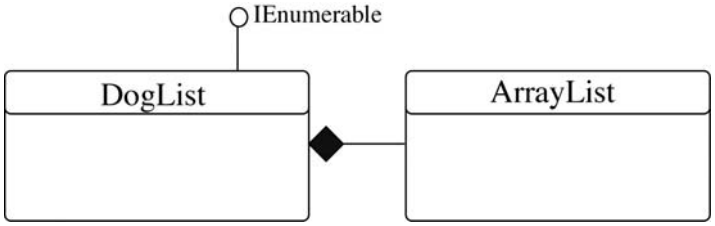


Figure 6-6: DogList Class Diagram

Referring to figure 6-6 — the DogList class contains an ArrayList by value and implements the IEnumerable interface. By implementing the IEnumerable interface, a DogList can be iterated via the `foreach` statement.

As its name implies, the DogList class will ensure that objects inserted into its ArrayList collection are of type Dog, the code for which is given in example 6.5.

6.5 Dog.cs

```
1    using System;
2
3    public class Dog : IComparable {
4      private string _first_name;
5      private string _last_name;
6      private string _breed;
7
8      public Dog(string breed, string f_name, string l_name){
9        _breed = breed;
10       _first_name = f_name;
11       _last_name = l_name;
12     }
13
14     public string FirstName {
15       get { return _first_name; }
16       set { _first_name = value; }
17     }
18
19     public string LastName {
20       get { return _last_name; }
21       set { _last_name = value; }
22     }
23
24     public string Breed {
25       get { return _breed; }
26       set { _breed = value; }
27     }
28
29     public string FullName {
30       get { return FirstName + " " + LastName; }
31     }
```



```
32
33     public string BreedAndFullName {
34       get { return Breed + ": " + FullName; }
35     }
36
37     public override string ToString(){
38       return this.BreedAndFullName;
39     }
40
41     public int CompareTo(object obj){
42       if(obj is Dog){
43         return this.LastName.CompareTo(((Dog)obj).LastName);
44       }else{
45         throw new ArgumentException("Object being compared is not a Dog!");
46       }
47     }
48   }
```

Referring to example 6.5 — the Dog class implements the IComparable interface so that instances of Dog can be compared to each other. This is required because I want to implement the Sort() method in the DogList class. The IComparable interface specifies a method named CompareTo(object obj) which I have implemented beginning on line 41. The first thing the CompareTo() method does is to check the type of incoming object to ensure it's a Dog. If the comparison succeeds, the current instance, represented by the `this` pointer, is compared with the obj parameter — I am comparing LastName properties in this case. If the incoming argument is not of type Dog, an ArgumentException is thrown.

I have also overridden the Object.ToString() method, on line 37, which simply returns the value of the BreedAndFullName property.

Example 6.6 gives the code for the DogList class.

6.6 DogList.cs

```
1    using System;
2    using System.Collections;
3
4    public class DogList : IEnumerable {
5       private ArrayList _list = null;
6
7
8       public DogList(int size){
9         _list = new ArrayList(size);
10      }
11
12      public DogList():this(25){  }
13
14      public int Count {
15        get { return _list.Count; }
16      }
17
18
19      public Dog this[int index] {
20        get {
21           return (Dog)_list[index];
22        }
23        set {
24          _list[index] = value;
25        }
26
27      }
28
29      public int Add(Dog d){
30        return _list.Add(d);
31      }
32
33      public void Remove(Dog d){
34        _list.Remove(d);
35      }
36
37      public void RemoveAt(int index){
38        _list.RemoveAt(index);
39      }
40
41      public void Reverse(){
42        _list.Reverse();
43      }
44
45      public IEnumerator GetEnumerator(){
46       return _list.GetEnumerator();
47      }
```



```
48
49      public void Sort(){
50        _list.Sort();
51      }
52   }
```

Referring to example 6.6 — the DogList class implements IEnumerable and encapsulates an ArrayList collection. I have provided implementations for just a small handful of the methods and properties found in the ArrayList class including the indexer, Add(), Remove(), RemoveAt(), Reverse(), and Sort(). Note how easy it is to implement the GetEnumerator() method with this particular approach. Example 6.7 shows the DogList class in action.

6.7 MainApp.cs

```
1    using System;
2
3    public class MainApp {
4      public static void Main(){
5        DogList list = new DogList();
6        list.Add(new Dog("Mutt", "Skippy", "Jones"));
7        list.Add(new Dog("French Poodle", "Bijou", "Jolie"));
8        list.Add(new Dog("Yellow Lab", "Schmoogle", "Miller"));
9        list.Add(new Dog("Mutt Lab", "Dippy", "Miller"));
10
11       for(int i = 0; i < list.Count; i++){
12         Console.WriteLine(list[i]);
13       }
14
15       Console.WriteLine("------------------------------");
16       list.Sort();
17
18       foreach(Dog d in list){
19         Console.WriteLine(d);
20       }
21
22       Console.WriteLine("------------------------------");
23       list.Reverse();
24
25       foreach(Dog d in list){
26         Console.WriteLine(d);
27       }
28     }
29   }
```

Referring to example 6.7 — a DogList instance is created on line 5, followed by the insertion of four Dog objects into the collection. On line 11 I demonstrate the DogList indexer by iterating over the list with a `for` loop. On line 16 I make a call to the Sort() method which sorts the contents of the list by last name. Next, I iterate over the list with the help of the enumerator and the `foreach` statement. Finally, on line 23, I call Reverse() to reverse the list elements, and again print the list contents to the console with the `foreach` statement. Figure 6-7 shows the results of running this program.

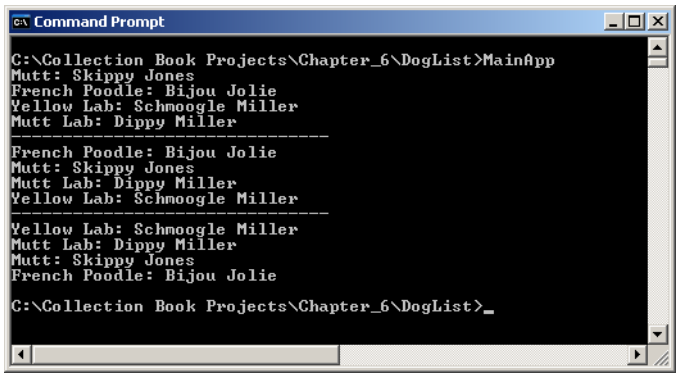


Figure 6-7: Results of Running Example 6.7

To be clear, the approaches described above would only apply if you were forced to use non-generic collections. This type of programming has been superseded by the introduction of generic collections.

Quick Review

The non-generic ArrayList collection can hold any type of object. It implements the IEnumerable, ICollection, IList, and ICloneable interfaces. The IEnumerable interface together with the IEnumerator interface enable the ArrayList elements to be iterated in a standardized, sequential fashion, beginning with the first element of the list and going forward. The ICollection interface extends IEnumerable and serves as the base interface for all non-generic collection classes. The IList interface adds numerous methods and properties but most importantly it adds an *indexer* that is used to access each element in the collection by an index, just like an array. The ICloneable interface declares one method: Clone(). The Clone() method is used to create an exact copy of an existing collection.

You can employ one of three approaches to create a type-safe list collection: 1) extend the ArrayList class and override all its public members, 2) extend the CollectionBase class, which is the recommended approach, or 3) use the façade design pattern and create a wrapper class that encapsulates an ArrayList collection and provides implementations for the most often-used interface methods. Note that all these approaches are superseded by the introduction of generic collection classes.

The Generic List<T> Collection

The generic List<T> collection, found in the System.Collections.Generic namespace, is the direct generic replacement for the non-generic ArrayList class. It is used to store and manipulate a collection of elements whose type is specified by the type parameter T.

In this section I will discuss the differences between the ArrayList and List<T> inheritance hierarchies and highlight the benefits derived from utilizing the generic List<T> class.

List<T> Inheritance Hierarchy

Figure 6-8 shows gives the UML class diagram for the expanded inheritance hierarchy of the List<T> collection class.

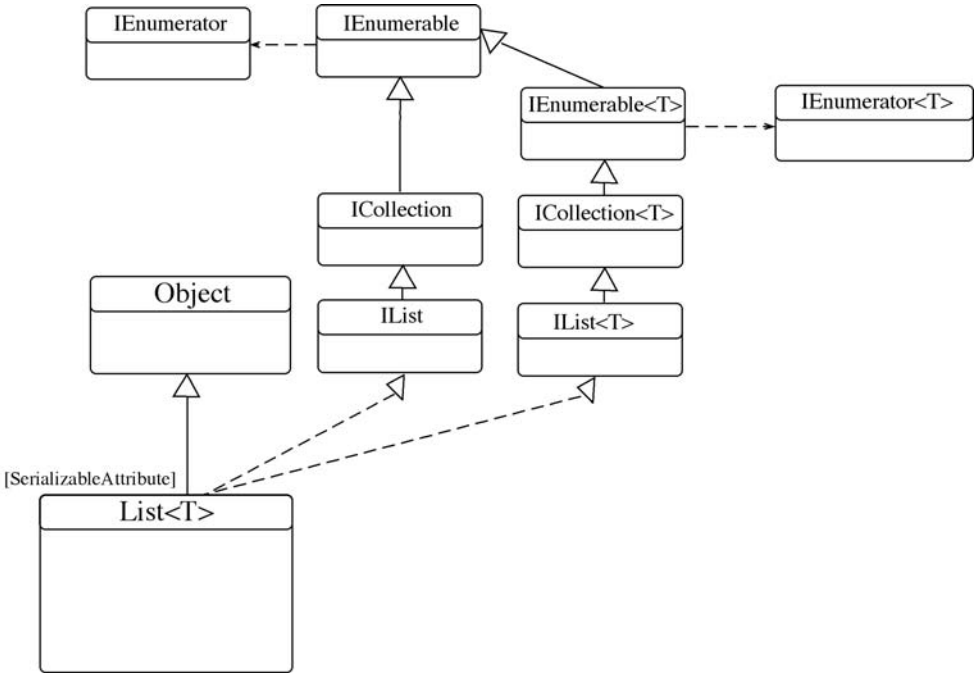


Figure 6-8: List<T> Inheritance Hierarchy

Referring to figure 6-8 — the List<T> class implements the same interfaces as the ArrayList class plus the following additional interfaces: IEnumerable<T>, ICollection<T>, and IList<T>. The IEnumerable<T> interface has a dependency on the IEnumerator<T> interface. Notice also that IEnumerable<T> extends IEnumerable. The following sections discuss the functionality provided by these interfaces.

Functionality Provided by the IEnumerable<T> and IEnumerator<T> Interfaces

The IEnumerable<T> and IEnumerator<T> interfaces together expose the enumerator for collections of a specified type. The IEnumerable<T> interface declares two overloaded versions of the GetEnumerator() method: One returns an IEnumerator object and the other returns and IEnumerator<T> object.

In addition to these two methods, the IEnumerable<T> sports an additional 43 extension methods, most of which are defined by the System.Linq.Enumerable class. (Compare this with the three extension methods defined for the non-generic IEnumerable interface.) As you can see, IEnumerable<T> offers a lot more functionality than IEnumerable.

Functionality Provided by the ICollection<T> Interface

The ICollection<T> interface extends IEnumerable<T> and serves as the base interface for all generic collection classes. It supplies the Add(), Clear(), Contains(), CopyTo(), GetEnumerator(), and Remove() methods in addition to the Count and IsReadOnly properties.

Functionality Provided by the IList<T> Interface

The IList<T> interface extends ICollection<T> and represents a strongly typed collection of elements that can be accessed by an index. Individual element access is provided by an indexer. (**Note:** It is the implementation of the IList or IList<T> interfaces that enable a collection to be accessed with an indexer like an ordinary array.)

Benefits of Using List<T> vs. ArrayList

The List<T> class provides two primary benefits over the non-generic ArrayList class. First, you get a significant performance improvement when manipulating large collections of value-type object. Value-type objects normally require a boxing and unboxing operation when used in a non-generic collection. But when a value-type is specified for the type parameter <T> in a generic collection, an optimized version of the collection is generated based on the specified type. This customization of the generated generic type eliminates the need to box and unbox value-type objects in the collection.

The second primary benefit to using the List<T> class is the wide array of extension methods you can use to manipulate list elements. These extension methods are simply not available via the ArrayList class. Example 6.8 demonstrates the use of several extension methods on a list of integers.

6.8 ListExtensionMethodsDemo.cs

```
1    using System;
2    using System.Collections.Generic;
3    using System.Linq;
4
5    public class ListExtensionMethodsDemo {
6
7      private static void PrintList(List<int> list){
8        for(int i=1; i<list.Count + 1; i++){
9          if((i%10) > 0){
10            Console.Write(list[i-1] + "\t");
11         }else{
12            Console.WriteLine(list[i-1] );
13          }
14        }
15       Console.WriteLine();
16     }
17
18     public static void Main(){
19       List<int> list = new List<int>();
20       Random random = new Random();
21
22       for(int i=0; i<50; i++){
```



```
23          list.Add(random.Next(0, 1000));
24        }
25
26        ListExtensionMethodsDemo.PrintList(list);
27        Console.WriteLine("---------------------------------");
28        Console.WriteLine("The sum of the list elements is: " + list.Sum());
29        Console.WriteLine("The average of the list elements is: " + list.Average());
30        Console.WriteLine("The maximum element value is: " + list.Max());
31        Console.WriteLine("The minimum element value is: " + list.Min());
32        Console.WriteLine("---------------------------------");
33        list.Sort();
34        ListExtensionMethodsDemo.PrintList(list);
35      }
36    }
```

Referring to example 6.8 — Note that for this example to work you must add the using directive on line 3 to access the System.Linq namespace. The Main() method begins on line 18. On line 19 I declare and instantiate a list of integers. On the following line I create a Random object and use it in the `for` loop beginning on line 22 to populate the list with integer values between 0 and 1000 inclusive. On line 26 a call to the static PrintList() method prints the elements in the list in a nice readable rectangular pattern.

On lines 28 through 31 I call the Sum(), Average(), Max(), and Min() extension methods respectively and print the results of each method call to the console. I then call the Sort() method to sort the list elements and print the results to the console. Figure 6-9 shows the results of running this program.

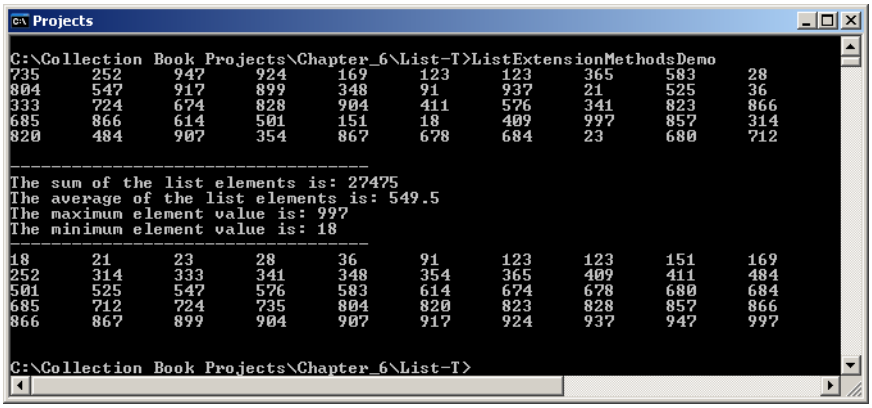


Figure 6-9: Results of Running Example 6.8

Quick Review

The generic List<T> collection, found in the System.Collections.Generic namespace, is the direct generic replacement for the non-generic ArrayList class. It is used to store and manipulate a collection of elements whose type is specified by the type parameter <T>.

The List<T> class offers several benefits over the non-generic ArrayList class: 1) faster performance when manipulating large collections of value-type objects, and 2) numerous extension methods defined by the System.Linq.Enumerable class.

Linked List Collections – How They Work

A linked list serves as the foundational data structure for the LinkedList<T> collection. In this section I will compare the performance characteristics of array-based lists to linked lists and show you how linked lists work under the covers.

Linked Lists vs. Array-Based Lists

Linked lists differ from array-based lists in many ways. An array-based list will always consume a certain amount of memory in the form of the array's initial capacity. In other words, when the list is created, its underlying array is created to hold a specific number of elements. In many cases these array elements contain no data, at least not initially, but the space is reserved just the same. In the case of an array of reference types, the size of each element is equal to the virtual machine's memory word size (32-bits for example). For value types, the size of each element will equal the size of the data structure. Integers are 32 bits, longs are 64 bits, etc., and user-defined value types can be and usually are larger. Thus, a large array of user-defined value types might tie up a significant amount of memory. However, at least in the case of array-based collections, the use of memory is conserved by starting the array size small and growing it larger only upon demand. Conversely, a linked list creates element nodes only when needed, therefore conserving memory at the outset.

Another big difference between the two data structures is how elements are inserted and retrieved. In an array-based collection, element retrieval via an indexer is lightening quick and constant across the entire array. However, if an element needs to be inserted in the middle of the array, elements must be shifted to accommodate the insertion. This may take some time if the list contains many elements. On the other hand, element insertions in a linked list take a constant amount of time, but the time it takes to retrieve an element from a linked list depends on where in the list it's located and the number of elements the list contains.

Linked List Operation - The Circular Linked List

A linked list is a data structure based on linked nodes. Unlike an array, where elements are grouped together contiguously, a linked list's nodes may be anywhere in memory. The only way you can locate a particular element in a linked list is to start at the first element (head) and search forward, or start at the last element (tail) and search backwards, moving from node to node via the link each node maintains to the next, as figure 6-10 illustrates.

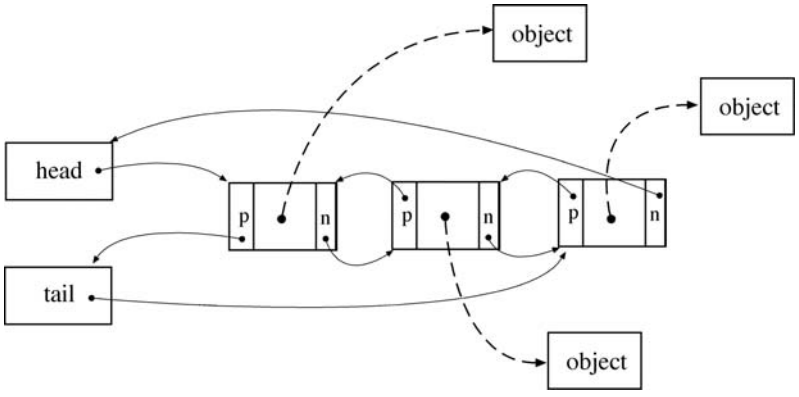


Figure 6-10: Circular, Doubly Linked List Data Structure

Referring to figure 6-10 — this type of linked list is referred to as a circular, doubly linked list. It's a doubly linked list because each node contains two references that maintain location information for the next node in the list and the previous node. It's circular because the first node in the list's previous reference is set to point to the tail, and the last node's next reference is set to point to the head. In this way one can "walk" the list from the first element to the last and return to the first element.

I'd like to illustrate these concepts better with some example code. Examples 6.9 through 6.11 demonstrate how a circular linked list might be constructed. Let's start by looking at the code for the node.

6.9 Node.cs

```
1    public class Node<T> {
2      private Node<T> _previous;
3      private Node<T> _next;
4      private T _value;
5
6      public Node<T> Previous {
7        get { return _previous; }
8        set { _previous = value; }
9      }
10
11     public Node<T> Next {
12       get { return _next; }
13       set { _next = value; }
14     }
15
16     public T Value {
17       get { return _value; }
18       set { _value = value; }
19     }
20   }
```

Referring to example 6.9 — the Node class contains three properties: Previous, Next, and Value, that correspond to the fields _previous, _next, and _value. Note how the fields _previous and _next are of type Node<T>. This is an example of when you are allowed to define a field to be the same type as the class you are defining.

Example 6.10 gives the code for the CircularLinkedList data structure.

6.10 CircularLinkedList.cs

```
1    using System;
2
3    public class CircularLinkedList<T> {
4      // private fields
5      private Node<T> _head = null;
6      private Node<T> _tail = null;
7      private int _count = 0;
8
9      // constructor
10     public CircularLinkedList(){ }
11
12     // read-only  properties
13     public int Count {
14       get { return _count; }
15     }
16
17     public Node<T> First {
18       get { return _head; }
19     }
20
21     public Node<T> Last {
22       get { return _tail; }
23     }
24
25     // methods
26     public Node<T> AddFirst(T value){
27        if(value == null) throw new ArgumentNullException();
28        Node<T> node = new Node<T>();
29        node.Value = value;
30        if(_head == null){ // list is empty
31          _head = node;
32          _tail = node;
33          node.Previous = _tail;
34          node.Next = _head;
35          _count++;
36        } else {
37            _head.Previous = node;
38            node.Next = _head;
39            node.Previous = _tail;
40            _tail.Next = node;
41            _head = node;
42            _count++;
43          }
44        return node;
45     }
46
47     public Node<T> AddLast(T value){
48       if(value == null) throw new ArgumentNullException();
49       if(_tail == null) { // list is empty
```



```
50          return this.AddFirst(value);
51        }
52        Node<T> node = new Node<T>();
53        node.Value = value;
54        node.Next = _tail.Next;
55        node.Previous = _tail;
56        _tail.Next = node;
57        _tail = node;
58        _count++;
59        return node;
60      }
61
62      public Node<T> AddBefore(Node<T> node, T value){
63        if((value == null) || (node == null)) throw new ArgumentNullException();
64        if(node == _head) {
65          return this.AddFirst(value);
66        }
67
68        Node<T> new_node = new Node<T>();
69        new_node.Value = value;
70        new_node.Previous = node.Previous;
71        node.Previous = new_node;
72        new_node.Next = node;
73        new_node.Previous.Next = new_node;
74        _count++;
75        return new_node;
76      }
77
78      public Node<T> AddAfter(Node<T> node, T value){
79        if((value == null) || (node == null)) throw new ArgumentNullException();
80        if(node == _tail) {
81          return this.AddLast(value);
82        }
83
84        Node<T> new_node = new Node<T>();
85        new_node.Value = value;
86        new_node.Previous = node;
87        new_node.Next = node.Next;
88        node.Next.Previous = new_node;
89        node.Next = new_node;
90        _count++;
91        return new_node;
92      }
93
94      public void Remove(Node<T> node){
95        if(node == null) throw new ArgumentNullException();
96        node.Next.Previous = node.Previous;
97        node.Previous.Next = node.Next;
98        if(node == _head){
99          _head = node.Next;
100       }
101       if(node == _tail){
102         _tail = node.Previous;
103       }
104       node.Next = null;
105       node.Previous = null;
106       _count--;
107     }
108
109     public Node<T> Find(T value){
110       if(value == null) throw new ArgumentNullException();
111       Node<T> current_node = _head;
112       for(int i = 0; i < _count; i++){
113         if(current_node.Value.Equals(value)){
114           return current_node;
115         } else {
116            current_node = current_node.Next;
117         }
118       }
119       return null; // return null if value not found in list
120     }
121   } // end CircularLinkedList<T> class definition
```

Referring to example 6.10 — the CircularLinkedList<T> class defines three private fields: _head, _tail, and _count, and three public properties: First, which corresponds to the _head field; Last, which corresponds to the _tail field, and Count, which of course corresponds to the _count field. I've also included six public methods: AddFirst(), AddLast(), AddBefore(), AddAfter(), Remove(), and Find(). (The property and method names I've used here match the property and method names of the System.Collections.Generic.LinkedList<T> class, which I discuss later in the chapter.)

Again referring to example 6.10 — let's take a closer look at the AddFirst() method. As its name implies, the AddFirst() method inserts the value as the first element in the list. The first thing I do is check the validity of the incoming value argument by making sure it's not null. If it is null, I throw an ArgumentNullException. Past that hurdle, I create a new Node<T> object and assign its Value property to equal the incoming value argument. Next, I check to see if the _head field is null. If it is, the list is currently empty and I make the insertion based on that assumption, else, the list contains at least one element, and the insertion must take into account the presence of an existing node.

When you do programming like this, you'll find yourself drawing diagrams on scrap pieces of paper to ensure you've accounted for all the references that must be set on each affected node, including the _head and _tail fields. Figures 6-11 and 6-12 show several pages from my engineering notebook when I wrote this code.

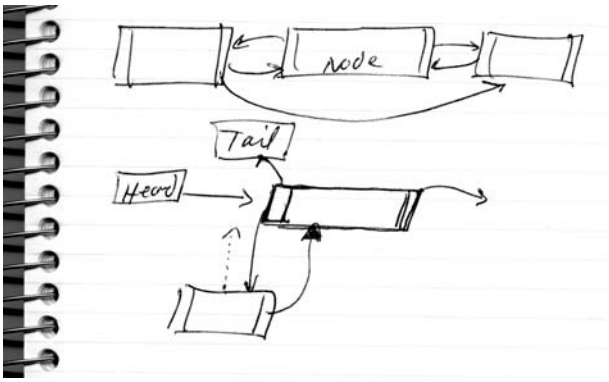


Figure 6-11: Linked List Insertion Notes

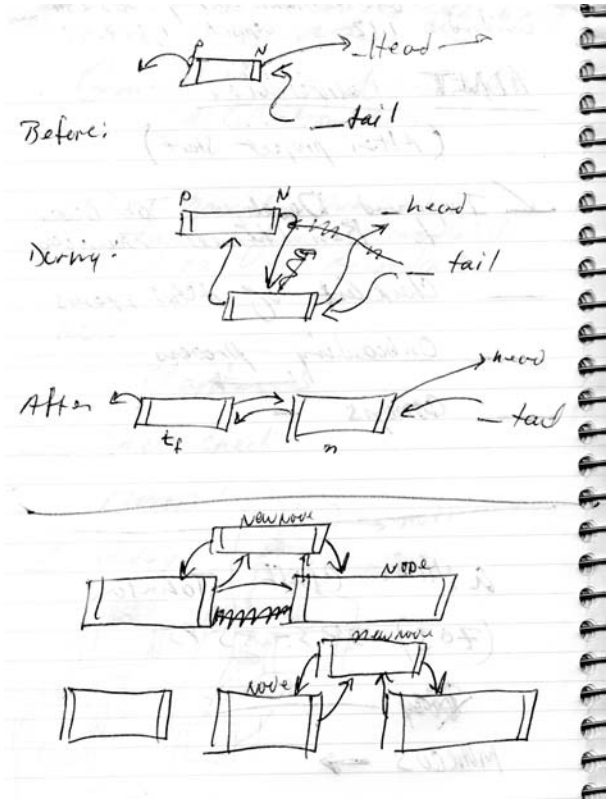


Figure 6-12: Linked List Insertion Notes

(Looking at those notes reminds me of learning to count and do math with the help of one's fingers!)

The AddLast() method works similar to the AddFirst() method, only it inserts the value as the last element in the list. The AddBefore(Node<T> node, T value) and AddAfter(Node<T> node, T value) methods take an existing node as their first argument and the value to be inserted as the second argument. These methods are used in conjunction with the Find() method, where the list must be searched for a particular value, and then the incoming value inserted into the list either before or after.

The Remove() method simply removes the indicated node from the list and subtracts one from the list count.

Example 6.11 shows the CircularLinkedList class in action.

6.11 MainApp.cs (Demonstrating CircularLinkedList)

```
1    using System;
2
3    public class MainApp {
4      private static CircularLinkedList<int> list = new CircularLinkedList<int>();
5      private static Node<int> current_node;
6
7      // utility method
8        private static void PrintListValues(){
9          current_node = list.First;
10         for(int i = 0; i < list.Count; i++){
11           Console.Write(current_node.Value + " ");
12           current_node = current_node.Next;
13         }
14         Console.WriteLine();
15       }
16
17     public static void Main(){
18
19       // Test AddFirst() method
20       MainApp.list.AddFirst(3);
21       MainApp.list.AddFirst(2);
22       MainApp.list.AddFirst(1);
23       MainApp.PrintListValues();
24
25       // Test Remove() method
26       MainApp.list.Remove(current_node.Previous);
27       MainApp.PrintListValues();
28
29       // Test AddBefore() method
30       MainApp.list.AddBefore(current_node.Previous, 4);
31       MainApp.PrintListValues();
32
33       // Test AddAfter() method
34       MainApp.list.AddAfter(MainApp.list.Find(4), 5);
35       MainApp.PrintListValues();
36
37       // Remove last element
38       MainApp.list.Remove(MainApp.list.Last);
39       MainApp.PrintListValues();
40
41       // Remove first element
42       MainApp.list.Remove(MainApp.list.First);
43       MainApp.PrintListValues();
44
45       // Test AddLast() method
46       MainApp.list.AddLast(6);
47       MainApp.PrintListValues();
48       Console.WriteLine("List has " + MainApp.list.Count + " items");
49
50       // Test AddAfter() method again
51       MainApp.list.AddAfter(MainApp.list.Find(6), 7);
52       MainApp.PrintListValues();
53
54       // Test AddBefore method again
55       MainApp.list.AddBefore(MainApp.list.Find(4), 3);
56       MainApp.PrintListValues();
57
58       // Let's step forward through the list
59       Console.Write("First element = " + MainApp.list.First.Value + " ");
60       Console.Write("Second element = " + MainApp.list.First.Next.Value + " ");
61       Console.Write("Third element = " + MainApp.list.First.Next.Next.Value + " ");
62       Console.Write("Forth element = " + MainApp.list.First.Next.Next.Next.Value + " ");
63       Console.Write("Fifth element = " + MainApp.list.First.Next.Next.Next.Next.Value + " ");
64       Console.Write("Next element = " + MainApp.list.First.Next.Next.Next.Next.Next.Value + " ");
65       Console.WriteLine();
66
67       // Now backwards
```



```
68        Console.Write("Last element = " + MainApp.list.Last.Value + " ");
69        Console.Write("Next element = " + MainApp.list.Last.Previous.Value + " ");
70        Console.Write("Next element = " + MainApp.list.Last.Previous.Previous.Value + " ");
71        Console.Write("Next element = " + MainApp.list.Last.Previous.Previous.Previous.Value + " ");
72        Console.Write("Next element = " + MainApp.list.Last.Previous.Previous.Previous.Previous.Value + " ");
73      }
74    }
```

Referring to example 6.11 — this MainApp class is structured slightly differently than previous MainApp examples. It contains two private fields and a method named PrintListValues(). The PrintListValues() method simply walks the list and prints each value to the console.

The Main() method begins on line 17. The first thing I do is exercise the AddFirst() method by inserting three integers into the list followed by a call to the PrintListValues() method. I follow this with a test of the various other methods defined by the CircularLinkedList class.

On lines 58 through 72 I demonstrate how to walk the list first going forward, via the Next references, then backwards using the Previous references.

Figure 6-13 shows the results of running example 6.11 demonstrating the use of the CircularLinkedList<T> class.

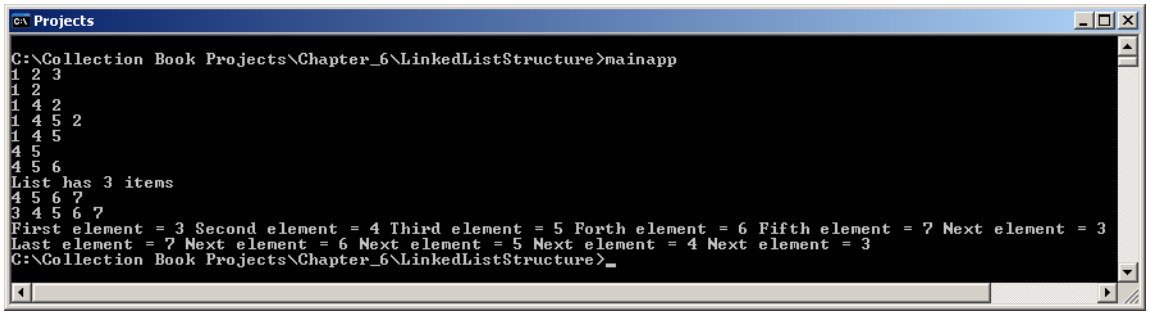


Figure 6-13: Results of Running Example 6.11 Testing the Circular Linked List

Quick Review

A linked list stores its elements in non-contiguous nodes which are linked together via Next and Previous references. The individual list nodes might be located anywhere in memory. To locate a specific node in a linked list, you must either start at the Head or First node and traverse the list forward, or start at the Tail or Last node and traverse the list backwards. A linked list conserves memory by storing data in individual nodes and grows the list one node at a time when needed.

The Generic LinkedList<T> Collection

Now that you understand how a linked list works, you'll understand how the generic LinkedList<T> class operates. As its name implies, the LinkedList<T> class stores elements in a linked list structure. Elements can be access sequentially beginning at the First element and walking the list forward, or starting at the Last element and walking the list backward.

The LinkedList<T> class provides a whole lot more functionality than the CircularLinkedList example presented in the previous section. Like most of the collections in the System.Collections.Generic namespace, the LinkedList<T> class can be manipulated with extension methods defined by the System.Linq.Enumerable class.

LinkedList<T> is Non-Circular!

An important feature to understand up front about the LinkedList<T> class is that it's not a circular list. In other words, the Next reference of the Last node points to null, and the Previous reference of the First node points to null as well.

The LinkedList<T> Inheritance Hierarchy

The LinkedList<T> class's inheritance hierarchy is shown in figure 6-14. Referring to figure 6-14 — the

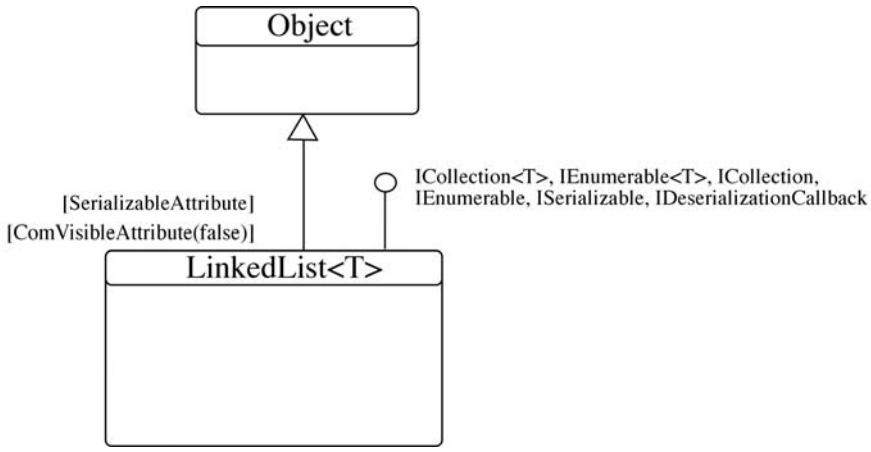


Figure 6-14: LinkedList<T> Inheritance Hierarchy

LinkedList<T> class implements the IEnumerable, IEnumerable<T>, ICollection, ICollection<T>, ISerializable, and IDeserializationCallback interfaces.

Functionality Provided By The IEnumerable and IEnumerable<T> Interfaces

The IEnumerable and IEnumerable<T> interfaces, along with their related IEnumerator and IEnumerator<T> interfaces expose the enumerator for the linked list structure. You can walk the list from the first to the last element on the list with the `foreach` statement. You can also access the enumerator directly by calling the GetEnumerator() method.

The thing to remember when using an enumerator with any collection class is that any attempt to modify the underlying collection while traversing the collection with the help of its enumerator will invalidate the enumerator and its behavior from that point forward cannot be predicted. If you need to control access to a collection while traversing its contents you must synchronize access to the collection object. This topic is discussed in detail in Chapter 14: Collections and Threads.

Functionality Provided by the ICollection and ICollection<T> Interfaces

The ICollection interface extends IEnumerable; the ICollection<T> interface extends IEnumerable<T>. Together the ICollection and ICollection<T> interfaces allow the LinkedList<T> class to be treated as a collection object, publishing the Add(), Clear(), Contains(), CopyTo(), GetEnumerator(), and Remove() methods.

Functionality Provided by the ISerializable and IDeserializationCallback Interfaces

The ISerializable and IDeserializationCallback interfaces indicate that the programmers of the LinkedList<T> class had to implement custom serialization routines over and above what the SerializableAttribute provided. I discuss custom serialization and deserialization in detail in Chapter 17: Collections and I/O.

LinkedList<T> Collection in Action

Example 6.12 demonstrates the use of the LinkedList<T> collection class. This example closely resembles the code used to demonstrate the CircularLinkedList in the previous section, however, I've modified it to work correctly with the non-circular LinkedList<T> class.

6.12 MainApp.cs (LinkedList<T> Demo)

```
1    using System;
2    using System.Collections.Generic;
3
4    public class MainApp {
5      private static LinkedList<int> list = new LinkedList<int>();
6      private static LinkedListNode<int> current_node;
7
8      // utility method
9        private static void PrintListValues(){
10         current_node = list.First;
11         for(int i = 0; i < list.Count; i++){
12           Console.Write(current_node.Value + " ");
13           current_node = current_node.Next;
14         }
15         Console.WriteLine();
16       }
17
18     public static void Main(){
19
20       // Test AddFirst() method
21       MainApp.list.AddFirst(3);
22       MainApp.list.AddFirst(2);
23       MainApp.list.AddFirst(1);
24       MainApp.PrintListValues();
25
26       // Test Remove() method
27       MainApp.list.Remove(MainApp.list.Last);
28       MainApp.PrintListValues();
29
30       // Test AddBefore() method
31       MainApp.list.AddBefore(MainApp.list.Last, 4);
32       MainApp.PrintListValues();
33
34       // Test AddAfter() method
35       MainApp.list.AddAfter(MainApp.list.Find(4), 5);
36       MainApp.PrintListValues();
37
38       // Remove last element
39       MainApp.list.Remove(MainApp.list.Last);
40       MainApp.PrintListValues();
41
42       // Remove first element
43       MainApp.list.Remove(MainApp.list.First);
44       MainApp.PrintListValues();
45
46       // Test AddLast() method
47       MainApp.list.AddLast(6);
48       MainApp.PrintListValues();
49       Console.WriteLine("List has " + MainApp.list.Count + " items");
50
51       // Test AddAfter() method again
52       MainApp.list.AddAfter(MainApp.list.Find(6), 7);
53       MainApp.PrintListValues();
54
55       // Test AddBefore method again
56       MainApp.list.AddBefore(MainApp.list.Find(4), 3);
57       MainApp.PrintListValues();
58
59       // Let's step forward through the list
60       Console.Write("First element = " + MainApp.list.First.Value + " ");
61       Console.Write("Second element = " + MainApp.list.First.Next.Value + " ");
62       Console.Write("Third element = " + MainApp.list.First.Next.Next.Value + " ");
63       Console.Write("Forth element = " + MainApp.list.First.Next.Next.Next.Value + " ");
64       Console.Write("Fifth element = " + MainApp.list.First.Next.Next.Next.Next.Value + " ");
65       Console.WriteLine();
66
67       // Now backwards
68       Console.Write("Last element = " + MainApp.list.Last.Value + " ");
69       Console.Write("Next element = " + MainApp.list.Last.Previous.Value + " ");
70       Console.Write("Next element = " + MainApp.list.Last.Previous.Previous.Value + " ");
71       Console.Write("Next element = " + MainApp.list.Last.Previous.Previous.Previous.Value + " ");
72       Console.Write("Next element = " + MainApp.list.Last.Previous.Previous.Previous.Previous.Value + " ");
73     }
74   }
```

Referring to example 6.12 — a LinkedList<int> references is declared and initialized on line 5. On line 6, an instance of LinkedListNode<int> is declared for used later in the program. In the body of the Main() method, I put the linked list through its paces. Figure 6-15 shows the results of running this program.

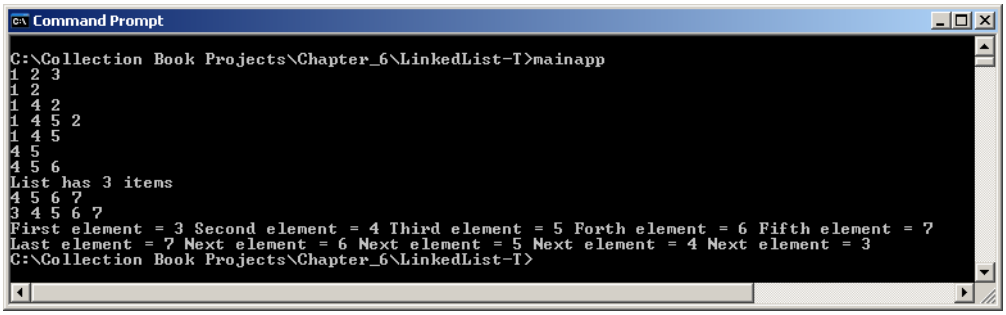


Figure 6-15: Results of Running Example 6.12

Quick Review

The generic LinkedList<T> class implements a doubly-linked, non-circular linked list collection. Elements of the LinkedList<T> class are stored as individual nodes of type LinkedListNode<T>. You may traverse the linked list structure manually, starting at the First node and moving forward, or at the Last node and moving backwards through the list, moving from node to node via the Next or Previous references as required. You can also traverse the list nodes automatically via its enumerator.

Summary

Array-based lists, as their name implies, feature arrays as their fundamental data structure. Array-based lists are created with an initial capacity and can grow in size automatically to accommodate additional elements.

The non-generic ArrayList collection can hold any type of object. It implements the IEnumerable, ICollection, IList, and ICloneable interfaces. The IEnumerable interface together with the IEnumerator interface enable the ArrayList elements to be iterated in a standardized, sequential fashion, beginning with the first element of the list and going forward. The ICollection interface extends IEnumerable and serves as the base interface for all non-generic collection classes. The IList interface adds numerous methods and properties but most importantly it adds an *indexer* that is used to access each element in the collection by an index, just like an array. The ICloneable interface declares one method: Clone(). The Clone() method is used to create an exact copy of an existing collection.

You can employ one of three approaches to create a type-safe list collection: 1) extend the ArrayList class and override all its public members, 2) extend the CollectionBase class, which is the recommended approach, or 3) use the façade design pattern and create a wrapper class that encapsulates an ArrayList collection and provides implementations for the most often-used interface methods. Note that all these approaches are superseded by the introduction of generic collection classes.

The generic List<T> collection, found in the System.Collections.Generic namespace, is the direct generic replacement for the non-generic ArrayList class. It is used to store and manipulate a collection of elements whose type is specified by the type parameter <T>.

The List<T> class offers several benefits over the non-generic ArrayList class: 1) faster performance when manipulating large collections of value-type objects, and 2) numerous extension methods defined by the System.Linq.Enumerable class.

A linked list stores its elements in non-contiguous nodes which are linked together via Next and Previous references. The individual list nodes might be located anywhere in memory. To locate a specific node in a linked list, you must either start at the Head or First node and traverse the list forward, or start at the Tail or Last node and traverse the list backwards. A linked list conserves memory by storing data in individual nodes and grows the list one node at a time when needed.

The generic LinkedList<T> class implements a doubly-linked, non-circular linked list collection. Elements of the LinkedList<T> class are stored as individual nodes of type LinkedListNode<T>. You may traverse the linked list structure manually, starting at the First node and moving forward, or at the Last node and moving backwards through

the list, moving from node to node via the Next or Previous references as required. You can also traverse the list nodes automatically via its enumerator.

References

Ryan Stephens, et. al., *C++ Cookbook*, O'Reilly Media, Inc., Sebastopol, CA. ISBN-13: 978-0-596-00761-4

Erich Gamma, et. al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Massachusetts. ISBN: 0-201-63361-2

Don Box. *Essential COM*, Addison-Wesley, Boston, Massachusetts. ISBN: 0-201-63446-5

Microsoft Developer Network (MSDN) [http://www.msdn.com]

Notes

