Chapter 14

Pentax 67 / SMC Takumar 150/2.8 / Ilford Delta 400

Jill & Ryan

Collections

Learning Objectives

- *Describe the purpose of a collection*
- *List and describe the classes, interfaces, and structures contained in the System.Collections, System.Collections.Generic, and System.Collections.ObjectModel namespaces*
- *State the general performance characteristics of arrays, lists, trees, and hash tables*
- *State the general operational characteristics of stacks and queues*
- *Choose a collection based on anticipated usage patterns*
- *Create custom collection classes by extending existing collection classes*
- *State the difference between non-generic and generic collections*
- *State the purpose of a class indexer member*
- *State the purpose of enumerators*
- *Use the foreach statement to iterate over a collection*
- *Utilize non-generic collections in your programs*
- *Utilize generic collections in your programs*

Introduction

When considering all functional assets of the .NET API, none possess the extraordinary potential to save you more time and hassle than its collections framework. Spanning four namespaces: System.Collections, System.Collections.Generic, System.Collections.ObjectModel, and System.Collections.Specialized, the interfaces, classes, and structures belonging to the collections framework provide you with a convenient way to manipulate collections of objects.

This chapter introduces you to the .NET collections framework and shows you how to employ its interfaces, classes, and structures in your programs. Along the way, you will learn about the functional characteristics of arrays, linked lists, hash tables, and red-black trees. Knowledge of these concepts sets the stage for understanding the inner workings of collections framework classes.

From the beginning, the .NET Framework API has undergone continuous evolutionary improvement. Nowhere is this more evident than in the collections framework. Each subsequent release of the .NET Framework introduces improved collections capability.

The differences between the .NET 2.0 collections framework and the previous version are significant. Version 2.0 introduced generics. These changes reduce and sometimes eliminate the need to utilize certain coding styles previously required to manipulate collections in earlier platform versions. For example, earlier non-generic versions of the collection classes stored object references. When the time came to access references stored in a collection, the object reference retrieved had to be cast to the required type. Such use of casting renders code hard to read and proves difficult for novice programmers to master.

I begin this chapter with a case study of a user-defined dynamic array class. The purpose of the case study is to provide a motivational context that demonstrates the need for a collections framework. I follow the case study with an overview of the collections framework, introducing you to its core interfaces and classes. I then show you how to manipulate collections using non-generic collection classes and interfaces. You will find this treatment handy primarily because you may find yourself faced with maintaining legacy .NET code. I then go on to discuss improvements made to the collections framework introduced with .NET 2.0 and later versions of the collections framework.

Case Study: Building A 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 14.1 along with a short test program shown in Example 14.2.

14.1 DynamicArray.cs

```
using System;

public class DynamicArray {
  private Object[] _object_array = null;
  private int _next_open_element = 0;
  private int _growth_increment = 10;
  private const int INITIAL_SIZE = 25;

  public int Count {
     get {  return _next_open_element; }
  }

  public object this[ int index] {
     get {
       if((index >= 0) && (index < _object_array.Length)){
          return _object_array[ index] ;
      } else return null;
     }
     set {
      if(_next_open_element < _object_array.Length){
              _object_array[ _next_open_element++]  = value;
          } else{
        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   }  // end DynamicArray class definition
```

Referring to Example 14.1 — 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 object array, and copies the elements to it from the temporary array.

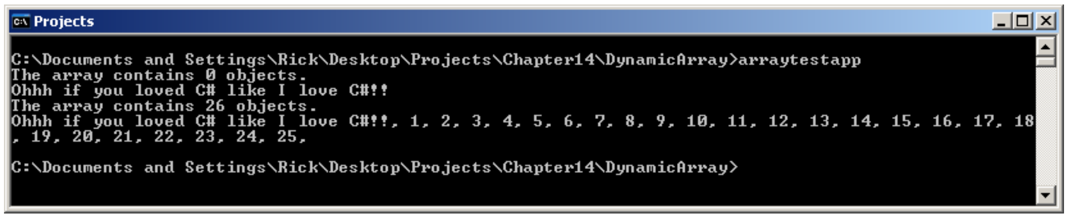
The indexer member allows you to access each element of the array. If the index argument falls out of bounds, the indexer returns null. The Count property simply returns the number of elements (references) contained in the array, which is the value of the _next_open_element variable.

Example 14.2 shows the DynamicArray class in action.

14.2 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          }
17       }
18       Console.WriteLine();
19     } //end Main() method
20   } // end ArrayTestApp class definition
```

Referring to Example 14.2 — 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 14-1 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\DynamicArray>arraytestapp
The array contains 0 objects.
Ohhh if you loved C# like I love C#!!
The array contains 26 objects.
Ohhh if you loved C# like I love C#!!, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18
, 19, 20, 21, 22, 23, 24, 25,

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\DynamicArray>
```

Figure 14-1: 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

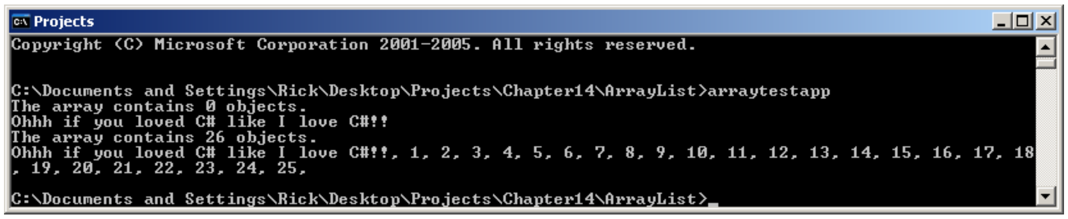
Let's re-write the ArrayTestApp program with the help of the IList interface and the ArrayList class, both of which belong to the .NET collections framework. Example 14.3 gives the code.

14.3 ArrayTestApp.cs (Mod 1)

```
1    using System;
2    using System.Collections;
3
4    public class ArrayTestApp {
5      public static void Main(){
6       IList 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         }
18      }
19      Console.WriteLine();
20     } //end Main() method
21   } // end ArrayTestApp class definition
22
```

Referring to Example 14.3 — 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 IList interface type, and 3) also on line 6, I created an instance of ArrayList instead of an instance of DynamicArray.

Figure 14-2 shows the results of running this program. If you compare figures 14-1 and 14-2, 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.



```
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\ArrayList>arraytestapp
The array contains 0 objects.
Ohhh if you loved C# like I love C#!!
The array contains 26 objects.
Ohhh if you loved C# like I love C#!!, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18
, 19, 20, 21, 22, 23, 24, 25,

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\ArrayList>_
```

Figure 14-2: Results of Running Example 17.3

You might be asking yourself, "Why does this code work?" As it turns out, I gamed the system. The DynamicArray class presented in Example 14.1 just happens to partially implement the IList interface. Later, in Example 14.3, when I changed the type of the da reference from DynamicArray to IList and used the ArrayList collection class, everything worked fine because the ArrayList class implements the IList interface as well. In fact, I could substitute for ArrayList any collection class that implements the IList interface. **Note:** Unfortunately, not many do, as you'll learn when you dive deeper into the Collections namespace.

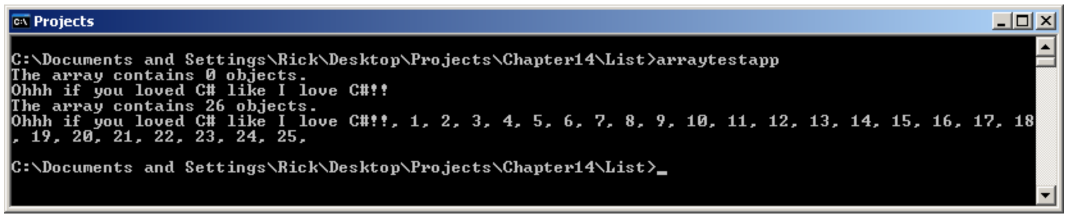
A Quick Peek At Generics

I can modify Example 14.1 once again to use a generic collection class. Example 14.4 gives the code.

14.4 ArrayTestApp.cs (Mod 2)

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

public class ArrayTestApp {
  public static void Main(){
    IList<Object> da = new List<Object>();
    Console.WriteLine("The array contains " + da.Count + " objects.");
    da.Add("Ohhh if you loved C# like I love C#!!");
    Console.WriteLine(da[0].ToString());
    for(int i = 1; i<26; i++){
      da.Add(i);
    }
    Console.WriteLine("The array contains " + da.Count + " objects.");
    for(int i=0; i<da.Count; i++){
      if(da[i] != null){
       Console.Write(da[i].ToString() + ", ");
       }
    }
    Console.WriteLine();
  }//end Main() method
}// end ArrayTestApp class definition
```

Referring to Example 14.4 — on line 6, I changed the da reference's type to IList<Object> and created an instance of the List<Object> generic collection class. (*i.e.,* List<T> where you substitute for T the type of objects you want the collection to contain.) Again, this code works because the IList<T> generic interface declares the same methods as the non-generic IList interface does. Figure 14-3 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\List>arraytestapp
The array contains 0 objects.
Ohhh if you loved C# like I love C#!!
The array contains 26 objects.
Ohhh if you loved C# like I love C#!!, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18
, 19, 20, 21, 22, 23, 24, 25,

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\List>_
```

Figure 14-3: Results of Running Example 14.4

Quick Review

The .NET collections framework can potentially save you a lot of time and hassle. It contains classes, structures, and interfaces designed to make it easy to manipulate collections of objects. The .NET 2.0 framework introduced generic collections and improved performance.

Data Structure Performance Characteristics

In this section, I want to introduce you to the performance characteristics of several different types of foundational data structures. These include the *array*, *linked list*, *hash table*, and *red-black binary tree*. Knowing a little bit about how these data structures work and behave will make it easier for you to select the .NET collection type that's best suited for your particular application.

Array Performance Characteristics

As you know already from reading Chapter 8, an array is a contiguous collection of homogeneous elements. You can have arrays of value types or arrays of references to objects. The general performance issues to be aware of regarding arrays concern inserting new elements into the array at some position prior to the last element, accessing elements, and searching for particular values within the array.

When a new element is inserted into an array at a position other than the end, room must be made at that index location for the insertion to take place by shifting the remaining references one element to the right. This series of events is depicted in figures 14-4 through 14-6.

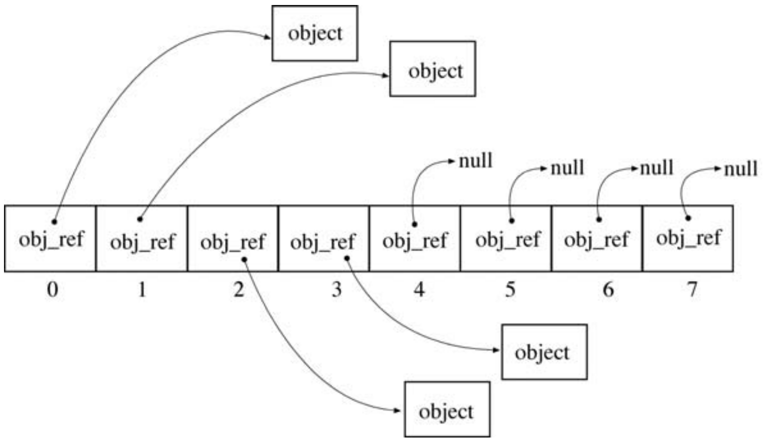


Figure 14-4: Array of Object References Before Insertion

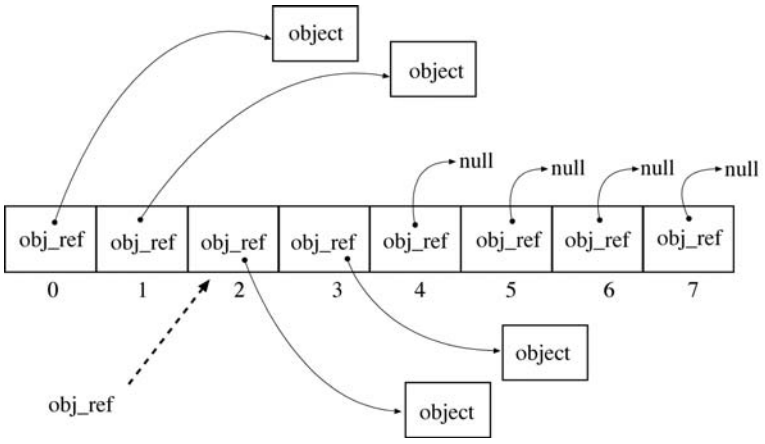


Figure 14-5: New Reference to be Inserted at Array Element 3 (index 2)

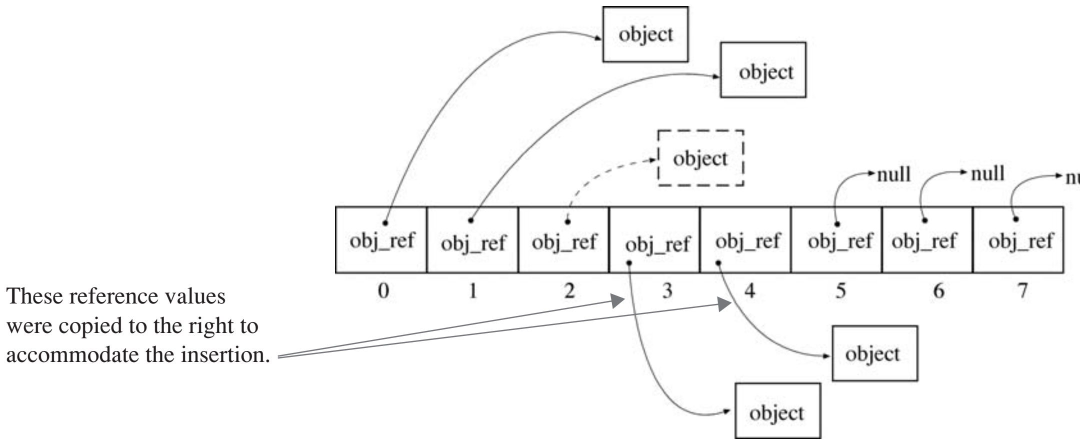


Figure 14-6: Array After New Reference Insertion

Referring to figures 14-4 through 14-6 — an array of object references contains references that may point to an object or to null. In this example, array elements 1 through 4 (index values 0 through 3) point to objects while the remaining array elements point to null.

A reference insertion is really just an assignment of the value of the reference being inserted to the reference residing at the target array element. To accommodate the insertion, the values contained in references located to the right of the target element must be reassigned one element to the right. (*i.e.,* They must be shifted to the right.) It is this shifting action that causes a performance hit when inserting elements into an array-based collection. If the insertion triggers the array growth mechanism, then you'll receive a double performance hit. The insertion performance penalty, measured in time, grows with the length of the array. Element retrieval, on the other hand, takes place fairly quickly because of the way array element addresses are computed. (*Refer to Chapter 8 — Arrays*)

Linked List Performance Characteristics

A linked list is a data structure whose elements stand alone in memory. (And may indeed be located anywhere in the heap!) Each element is linked to another by a reference. Unlike the elements of an array, which are ordinary references, each linked list node is a complex data structure that contains a reference to the *previous* node in the list, the *next* node in the list, and a reference to an object payload, as Figure 14-7 illustrates.

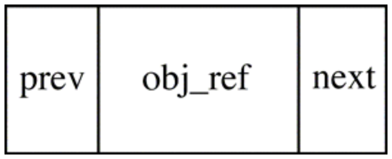


Figure 14-7: Linked List Node Organization

Whereas an array's elements are always located one right after the other in memory, and their memory addresses quickly calculated, a linked list's elements can be, and usually are, scattered in memory hither and yonder. The nice thing about linked lists is that element insertions take place fairly quickly because no element shifting is required. Figures 14-8 through 14-11 show the sequence of events for the insertion of a circular linked list node. Referring to figures 14-8 through 14-11 — a linked list contains one or more non-contiguous nodes. A node insertion requires reference rewiring. This entails setting the *previous* and *next* references on the new node in addition to resetting the affected references of its adjacent list nodes. If this looks complicated, guess what? It is! And if you take a data structures class you'll get the chance to create a linked list from scratch!

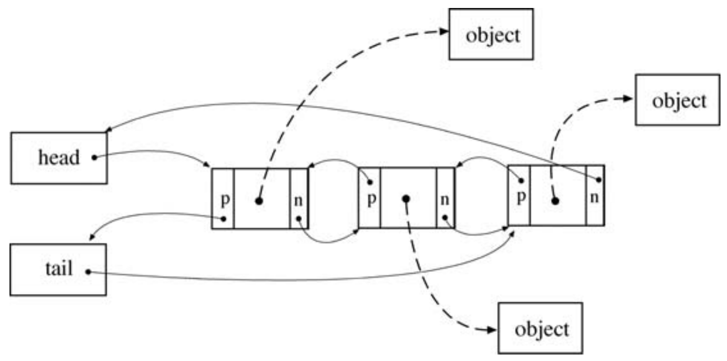


Figure 14-8: Linked List Before New Element Insertion

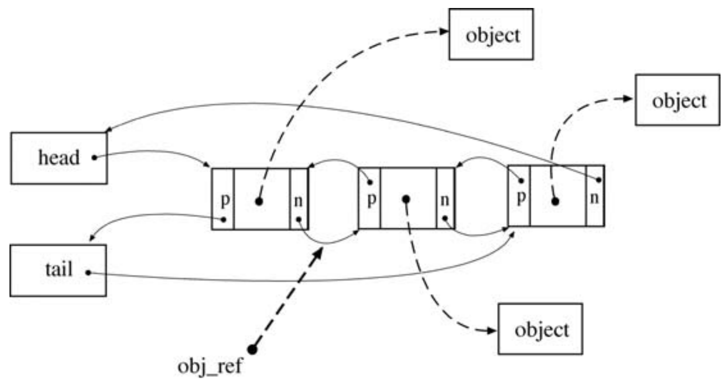


Figure 14-9: New Reference Being Inserted Into Second Element Position

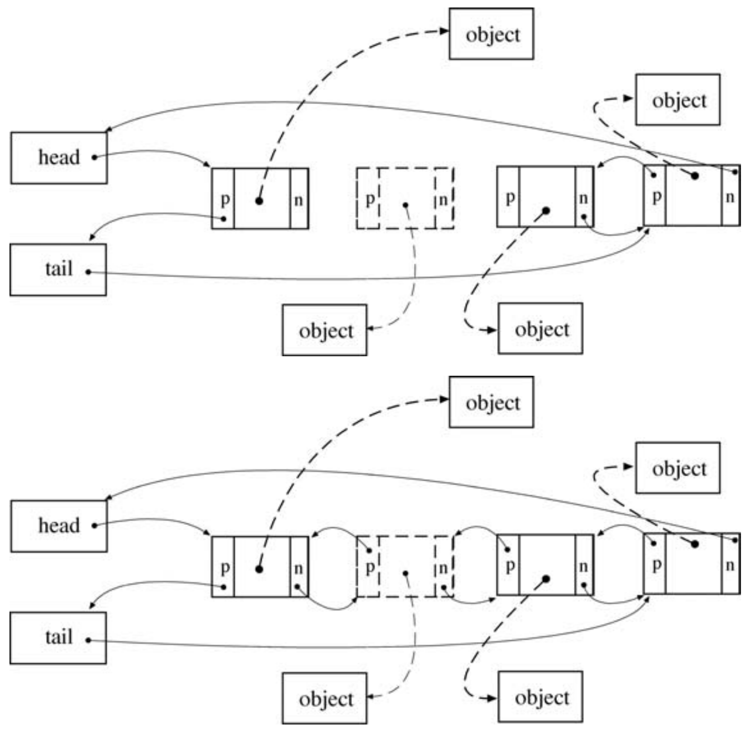


Figure 14-10: References of Previous, New, and Next List Elements must be Manipulated

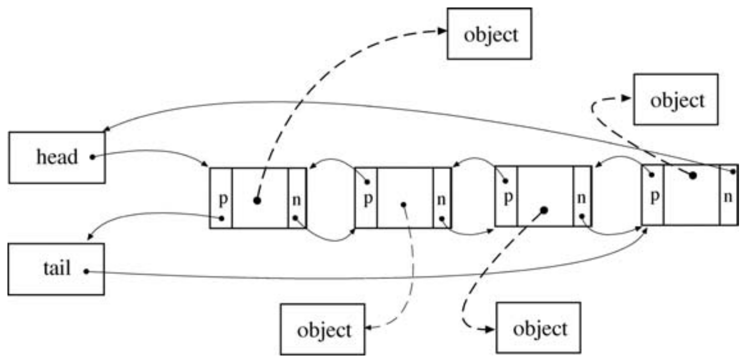


Figure 14-11: Linked List Insertion Complete

Hash Table Performance Characteristics

A hash table is an array whose elements can point to a series of nodes. Structurally, as you'll see, a hash table is a cross between an array and a one-way linked list. In an ordinary array, elements are inserted by index value. If there are potentially many elements to insert, the array space required to hold all the elements would be correspondingly large as well. This may result in wasted memory space. The hash table addresses this problem by reducing the size of the array used to point to its elements and assigning each element to an array location based on a *hash function* as Figure 14-12 illustrates.

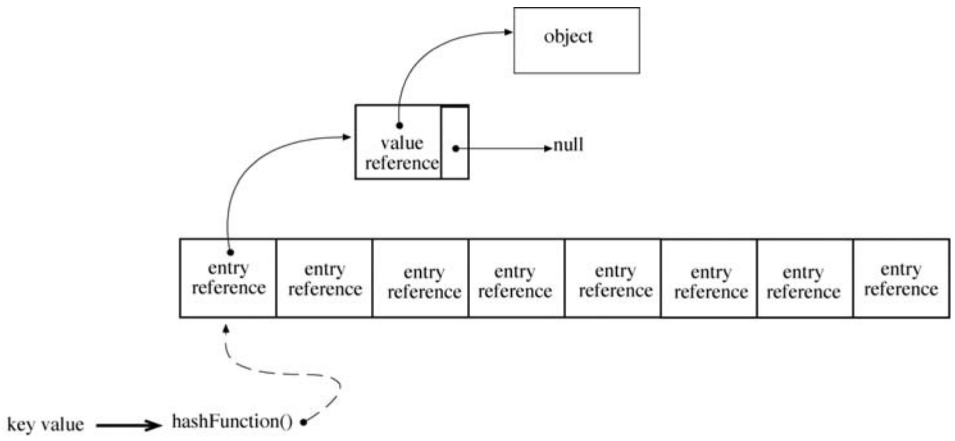


Figure 14-12: A Hash Function Transforms a Key Value into an Array Index

Referring to Figure 14-12 — the purpose of the hash function is to transform the key value into a unique array index value. However, sometimes two unique key values translate to the same index value. When this happens a *collision* is said to have occurred. The problem is resolved by chaining together nodes that share the same hash table index as is shown in Figure 14-13.

The benefits of a hash table include lower initial memory overhead and relatively fast element insertions. On the other hand, if too many insertion collisions occur, the linked elements must be traversed to insert new elements or to retrieve existing elements. List traversal extracts a performance penalty.

Chained Hash Table vs. Open-Address Hash Table

The hash table discussed above is referred to as a *chained hash table*. Another type of hash table, referred to as an *open-address hash table*, uses a somewhat larger array and replaces the linking mechanism with a *slot probe function* that searches for empty space when the table approaches capacity.

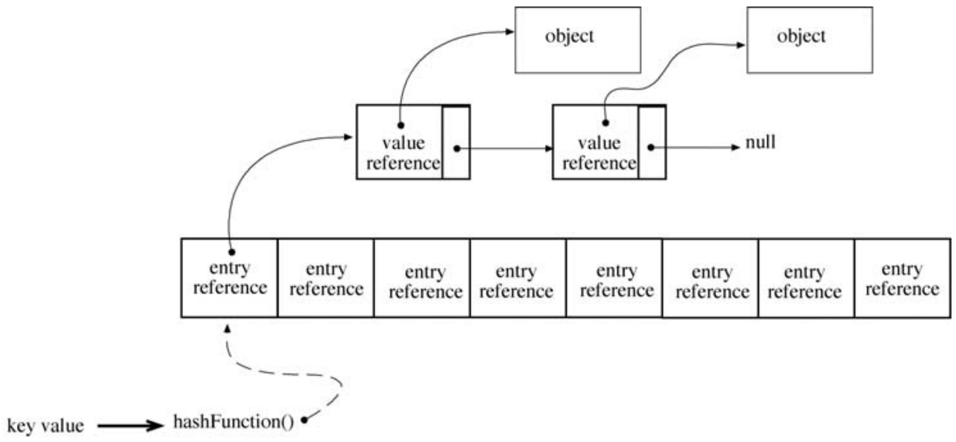


Figure 14-13: Hash Table Collisions are Resolved by Linking Nodes Together

Red-Black Tree Performance Characteristics

A *red-black tree* is a special type of *binary search tree* with a self-balancing characteristic. *Tree nodes* have an additional data element, *color*, that is set to either red or black. The data elements of a red-black tree node are shown in Figure 14-14.

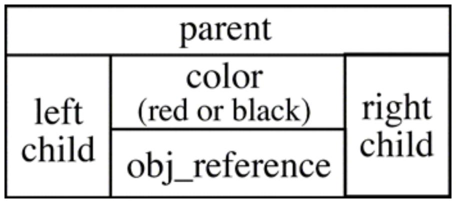


Figure 14-14: Red-Black Tree Node Data Elements

Insertions into a red-black tree are followed by a self-balancing operation. This ensures that all leaf nodes are the same number of black nodes away from the root node. Figure 14-15 shows the state of a red-black tree after inserting the integer values 1 through 9 in the following insertion order: 9, 3, 5, 6, 7, 2, 8, 4, 1. (Red nodes are shown lightly shaded.)

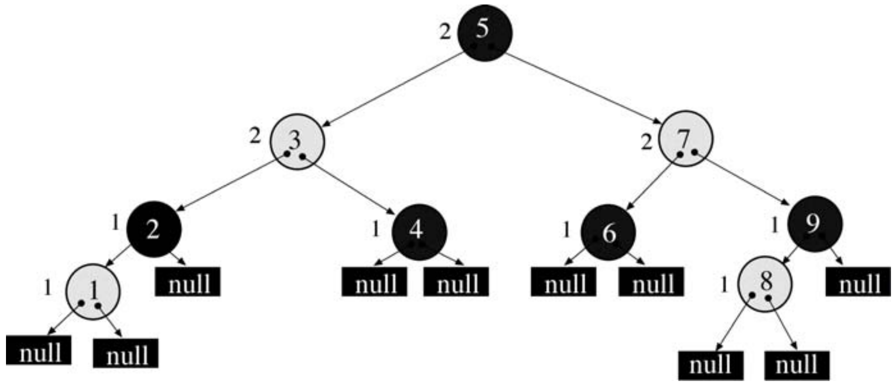


Figure 14-15: Red-Black Tree After Inserting Integer Values 9, 3, 5, 6, 7, 8, 4, 1

Referring to Figure 14-15 — the numbers appearing to the left of each node represent the height of the tree in black nodes. The primary benefit associated with a red-black tree is the generally overall good node search performance regardless of the number of nodes the tree contains. However, because the tree reorders itself with each insertion, an insertion into a tree that contains lots of nodes incurs a performance penalty.

Think of it in terms of a clean room versus a messy room. You can store things really fast in a messy room because you just throw your stuff anywhere. Finding things in a messy room takes some time. You may have to look high and low before finding what you're looking for. Storing things in a clean room, conversely, takes a little while longer, but when you need something, you can find if fast!

Stacks And Queues

Two additional data structures you'll encounter in the collections API are *stacks* and *queues*. A stack is a data structure that stores objects in a *last-in-first-out* (LIFO) basis. Objects are placed on the stack with a *push* operation and removed from the stack with a *pop* operation. A stack operates like a plate dispenser, where you put in a stack of plates and take plates off the stack one at a time. The last plate inserted into the plate dispenser is the first plate dispensed when someone needs a plate. Figure 14-16 shows the state of a stack after several pushes and pops.

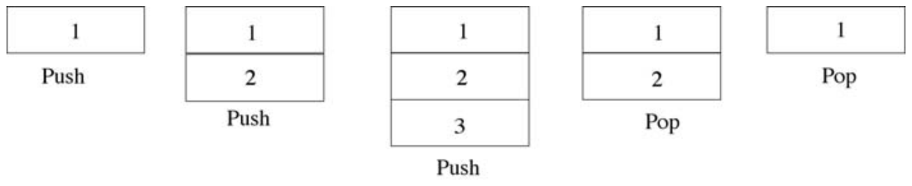


Figure 14-16: A Stack After Several Push and Pop Operations

A *queue* is a data structure that stores objects in a *first-in-first-out* (FIFO) basis. A queue operates like a line of people waiting for some type of service; the first person in line is the first person to be served. People arriving in line must wait for service until they reach the front of the line. Objects are added to a queue with an *enqueue* operation and removed with a *dequeue* operation. Figure 14-17 shows the state of a queue after several enqueues and dequeues.

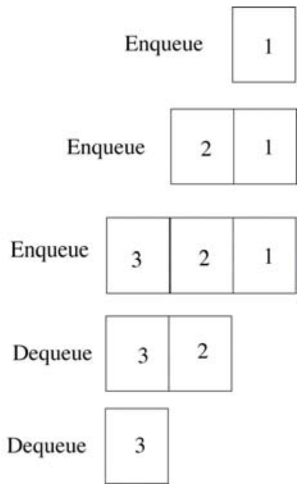


Figure 14-17: A Queue After Several Enqueue and Dequeue Operations

Quick Review

An *array* is a contiguous allocation of objects in memory. An array-based collection offers quick element access but slow element insertion, especially if the collection's underlying array must be resized and its contents shifted to accommodate the insertion.

A *linked list* consists of individual *nodes* linked to each other via references. To traverse a linked list, you must start at the beginning, or the end (head or tail) and follow each element to the next. Linked list-based collections can conserve memory space because memory need only be allocated on each object insertion. Insertions into linked list-based collections are relatively quick but element access is relatively slow due to the need to traverse the list.

A *chained hash table* is a cross between an array and a linked list and allows element insertion with key/value pairs. A *hash function* performed on the key determines the value's location in the hash table. A collision is said to occur when two keys produce the same hash code. When this happens, the values are chained together in a linked list-like structure. A hash function that produces a uniform distribution over all the keys is a critical feature of a hash table.

A red-black tree is a self-balancing binary tree. Insertions into a red-black tree take some time because of element ordering and balancing operations. Element access times for a red-black tree-based collection is fairly quick.

Navigating The .NET Collections API

To the uninitiated, the .NET collections API presents a bewildering assortment of interfaces, classes, and structures spread over four namespaces. In this section, I provide an overview of some of the things you'll find in each namespace. Afterward, I present a different type of organization that I believe you'll find more helpful.

One thing I will not do in this section is discuss every interface, class, or structure found in these namespaces. If I did that, you would fall asleep quick and kill yourself as your head slammed against the desk on its way down! Instead, I will only highlight the most important aspects of each namespace with an eye towards saving you time and frustration.

One maddening aspect of the .NET collections framework is in the way Microsoft chose to name their collection classes. For example, collection classes that contain the word List do not necessarily implement the IList or IList<T> interfaces. This means you can't substitute a LinkedList for an ArrayList without breaking your code. (In this regard I believe the Java collections framework is much more robust and logically organized.)

In concert with this section you should explore the collections API and see for yourself what lies within each namespace.

System.Collections

The System.Collections namespace contains non-generic versions of collection interfaces, classes, and structures. The contents of the System.Collections namespace represent the "old-school" way of collections programming. By this I mean that the collections defined here store only object references. You can insert any type of object into a collection like an ArrayList, Stack, etc., but, when you access an element in the collection and want to perform an operation on it specific to a particular type, you must first cast the object to a type that supports the operation. (By "performing an operation" I mean accessing an object member declared by its interface or class type.)

I recommend avoiding the System.Collections namespace altogether in favor of the generic versions of its members found in the System.Collections.Generic namespace. In most cases, you'll be trading cumbersome "old-school" style programming for more elegant code and improved performance offered by the newer collection classes.

System.Collections.Generic

.NET 2.0 brought with it generics and the collection classes found in the System.Collections.Generic namespace. In addition to providing generic versions of the "old-school" collections contained in the System.Collections namespace, the System.Collections.Generic namespace added several new collection types, one of them being LinkedList<T>.

Several collection classes within the System.Collections.Generic namespace can be used off-the-shelf, so to speak, to store and manipulate strongly-typed collections of objects. These include List<T>, LinkedList<T>, Queue<T>, Stack<T>.

Other classes such as Dictionary<TKey, TValue>, SortedDictionary<TKey, TValue>, and SortedList<TKey, TValue> store objects (values) in the collection based on the hash values of keys. Special rules must be followed when implementing a key class. These rules specify the types of interfaces a key class must implement in order to perform equality comparisons. They also offer suggestions regarding the performance of hashing functions to optimize insertion and retrieval. You can find these specialized instructions in the **Remarks** section of a collection class's API documentation page. I cover this topic in greater detail in Chapter 22 — Well Behaved Objects.

System.Collections.ObjectModel

The System.Collections.ObjectModel namespace contains classes that are meant to be used as the base classes for custom, user-defined collections. For example, if you want to create a specialized collection, you can extend the Collection<T> class. This namespace also includes the KeyedCollection<TKey, TItem>, ObservableCollection<T>, ReadOnlyCollection<T>, and ReadOnlyObservableCollection<T> classes.

The KeyedCollection<TKey, TItem> is an abstract class and is a cross between an IList and an IDictionary-based collection in that it is an indexed list of items. Each item in the list can also be accessed with an associated key. Collection elements are not key/value pairs as is the case in a Dictionary, rather, the element is the value and the key is extracted from the value upon insertion. The KeyedCollection<TKey, TItem> class must be extended and you must override its GetKeyForItem() method to properly extract keys from the items you insert into the collection.

The ObservableCollection<T> collection provides an event named CollectionChanged that you can register event handlers with to perform special processing when items are added or removed, or the collection is refreshed.

The ReadOnlyCollection<T> and ReadOnlyObservableCollection<T> classes implement read-only versions of the Collection<T> and ObservableCollection<T> classes.

System.Collections.Specialized

As its name implies, the System.Collections.Specialized namespace contains interfaces, classes, and structures that help you manage specialized types of collections. Some of these include the BitVector32 structure, the ListDictionary, which is a Dictionary implemented as a singly linked list intended for storing ten items or less, StringCollection, which is a collection of strings, and StringDictionary, which is a Dictionary whose key/value pairs are strongly typed to strings rather than objects.

Mapping Non-Generic To Generic Collections

In some cases, the System.Collection.Generic and System.Collections.ObjectModel namespaces provide a corresponding replacement for a collection class in the System.Collections namespace. But sometimes they do not. Table 14-1 lists the non-generic collection classes and their generic replacements, if any, and the underlying data structure implementation.

Non-Generic	Generic	Underlying Data Structure
ArrayList	List<T>	Array
BitArray	*No generic equivalent*	Array
CollectionBase	Collection<T>	Array
DictionaryBase	KeyedCollection<TKey, TItem>	Hash Table & Array
HashTable	Dictionary<TKey, TValue>	Hash Table
Queue	Queue<T>	Array
ReadOnlyCollectionBase	ReadOnlyCollection<T>	Array
SortedList	SortedList<TKey, TValue>	Red-Black Tree
Stack	Stack<T>	Array
No Non-Generic Equivalent	LinkedList<T>	Doubly Linked List

Table 14-1: Mapping Non-Generic Collections to Their Generic Counterparts

Non-Generic	Generic	Underlying Data Structure
No Non-Generic Equivalent	SortedDictionary<TKey, TValue>	Red-Black Tree
No Non-Generic Equivalent	SynchronizedCollection<T> †	Array
No Non-Generic Equivalent	SynchonizedKeyedCollection<TKey, TItem> †	Hash Table & Array
No Non-Generic Equivalent	SynchronizedReadOnlyCollection<T> †	Array
† Provides thread-safe operation		

Table 14-1: Mapping Non-Generic Collections to Their Generic Counterparts

Quick Review

"Old-school" style .NET collections classes store only object references and require casting when elements are retrieved. You should favor the use of generic collections as they offer strong element typing on insertion and retrieval and improved performance. The classes found in the System.Collections.ObjectModel namespace can serve as the basis for user-defined custom collections. The System.Collections.Specialized namespace contains classes and structures you will find helpful to manage unique collections.

Using Non-Generic Collection Classes - Pre .NET 2.0

In this section, I demonstrate the use of a non-generic collection class. Whereas earlier I used an ArrayList class to store various types of objects like strings and integers, here I use an ArrayList to store objects of type Person. I want to show you how to do two things in particular: 1) cast objects retrieved from a collection to a specified type, and 2) subclass ArrayList and override its methods to provide a strongly-typed collection. You'll find this section helpful if you're tasked with maintaining legacy .NET code based on 1.0 collection classes.

Example 14.5 gives the code for the Person class.

14.5 Person.cs

```
using System;

public class Person {

  //enumeration
  public enum Sex {MALE, FEMALE};

  // private instance fields
  private String   _firstName;
  private String   _middleName;
  private String   _lastName;
  private Sex      _gender;
  private DateTime _birthday;


  //private default constructor
  private Person(){}

  public Person(String firstName, String middleName, String lastName,
                Sex gender, DateTime birthday){
     FirstName = firstName;
     MiddleName = middleName;
     LastName = lastName;
     Gender = gender;
     BirthDay = birthday;
  }

  // public properties
  public String FirstName {
    get {  return _firstName; }
    set {  _firstName = value; }
  }

  public String MiddleName {
    get {  return _middleName; }
    set {  _middleName = value; }
  }

  public String LastName {
    get {  return _lastName; }
    set {  _lastName = value; }
  }

  public Sex Gender {
    get {  return _gender; }
    set {  _gender = value; }
  }

  public DateTime BirthDay {
    get {  return _birthday; }
    set {  _birthday = value; }
  }

   public int Age {
     get {
        int years = DateTime.Now.Year - _birthday.Year;
        int adjustment = 0;
        if((DateTime.Now.Month <= _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
          adjustment = 1;
       }
        return years - adjustment;
    }
   }

  public String FullName {
    get {  return FirstName + " " + MiddleName + " " + LastName; }
  }

  public String FullNameAndAge {
    get {  return FullName + " " + Age; }
  }

  public override String ToString(){
    return FullName + " is a " + Gender + " who is " + Age + " years old.";
  }
}  // end Person class
```

Objects In – Objects Out: Casting 101

“Old school” collections programming is characterized by the need to cast objects to an appropriate type when they are retrieved from a collection. Casting is required if you intend to perform an operation on an object other than those defined by the System.Object class. Example 14.6 gives the code for a short program that stores Person objects in an ArrayList.

14.6 MainApp.cs

```
using System;
using System.Collections;

public class MainApp {
   public static void Main(){
     ArrayList surrealists = new ArrayList();

     Person p1 = new Person("Rick", "", "Miller", Person.Sex.MALE, new DateTime(1961, 02, 04));
     Person p2 = new Person("Max", "", "Ernst", Person.Sex.MALE, new DateTime(1891, 04, 02));
     Person p3 = new Person("Andre", "", "Breton", Person.Sex.MALE, new DateTime(1896, 02, 19));
     Person p4 = new Person("Roland", "", "Penrose", Person.Sex.MALE, new DateTime(1900, 10, 14));
     Person p5 = new Person("Lee", "", "Miller", Person.Sex.FEMALE, new DateTime(1907, 04, 23));
     Person p6 = new Person("Henri-Robert-Marcel", "", "Duchamp", Person.Sex.MALE,
                             new DateTime(1887, 07, 28));

     surrealists.Add(p1);
     surrealists.Add(p2);
     surrealists.Add(p3);
     surrealists.Add(p4);
     surrealists.Add(p5);
     surrealists.Add(p6);

     foreach(object o in surrealists){
       Console.WriteLine(o.ToString());
     }

     Console.WriteLine("-----------------------------------------------------");
28
29        foreach(Person p in surrealists){
30          Console.WriteLine(p.FirstName);
31        }
32
33        Console.WriteLine("-----------------------------------------------------");
34
35        for(int i = 0; i<surrealists.Count; i++){
36          Console.WriteLine(((Person)surrealists[i]).FirstName + " " + ((Person)surrealists[i]).LastName);
37        }
38
39     } // end Main()
40   } // end MainApp class definition
```

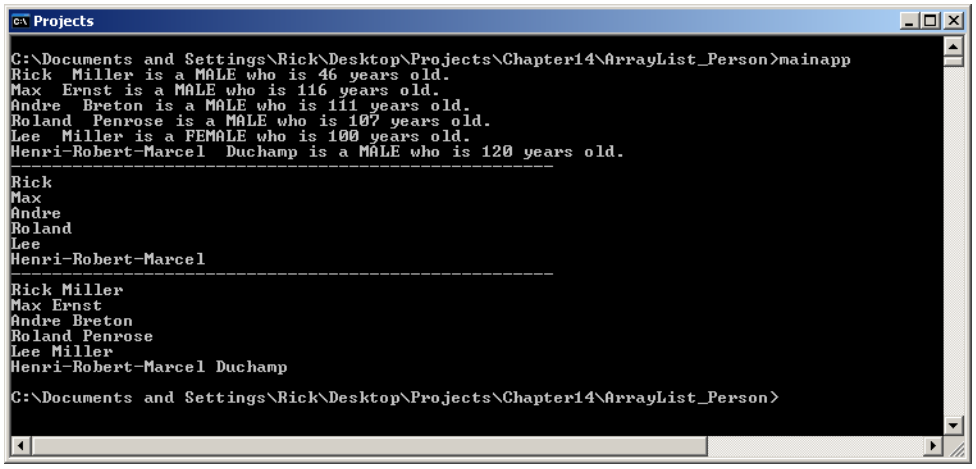
Referring to Example 14.6 — an ArrayList reference named surrealists is initialized on line 6 followed by the creation of six Person references on lines 8 through 13. Next, each Person reference is inserted into the collection using the ArrayList.Add() method.

The `foreach` statement on line 23 demonstrates how objects can be retrieved from the collection without the need for casting as long as you only intend to call methods defined by the System.Object class. In this case, the Person class overrides the ToString() method.

The `foreach` statement on line 29 demonstrates how to extract a particular type of object from the collection using an *implicit cast*. This works because all the objects in the ArrayList collection are indeed of type Person.

The `for` statement on line 35 iterates over the surrealists collection using array indexing. Each object in the collection must be explicitly cast to type Person before accessing its FirstName and LastName properties.

Figure 14-18 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\ArrayList_Person>mainapp
Rick  Miller is a MALE who is 46 years old.
Max  Ernst is a MALE who is 116 years old.
Andre  Breton is a MALE who is 111 years old.
Roland  Penrose is a MALE who is 107 years old.
Lee  Miller is a FEMALE who is 100 years old.
Henri-Robert-Marcel  Duchamp is a MALE who is 120 years old.
-----------------------------------------------------
Rick
Max
Andre
Roland
Lee
Henri-Robert-Marcel
-----------------------------------------------------
Rick Miller
Max Ernst
Andre Breton
Roland Penrose
Lee Miller
Henri-Robert-Marcel Duchamp

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\ArrayList_Person>
```

Figure 14-18: Results of Running Example 14.6

Extending ArrayList To Create A Strongly-Typed Collection

You can avoid the need to cast by creating a custom collection. In this section, I show you how to extend the ArrayList class to create a custom collection that stores Person objects. Example 14.7 gives the code for the custom collection named PersonArrayList.

14.7 PersonArrayList.cs

```
using System;
using System.Collections;

public class PersonArrayList : ArrayList {

  public PersonArrayList():base(){}

  public new Person this[int index]{
    get { return (Person) base[index];}

    set { base[index] = (Person) value; }
  }

  public override int Add(object o){
    return base.Add((Person)o);
  }
}// end PersonArrayList class definition
```

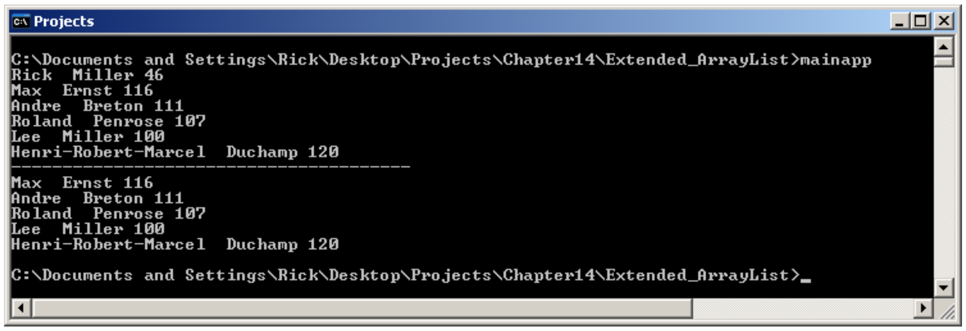
Referring to Example 14.7 — the PersonArrayList class extends ArrayList, overrides its Add() method, and provides a new implementation for its indexer. The PersonArrayList.Add() method accepts an object and casts it to Person before inserting it into the collection using the ArrayList.Add() method. The indexer casts incoming objects (*value*) to type Person before inserting them into the collection. It casts retrieved objects to type Person before their return from the indexer call. The indexer cannot simply be overridden in this case because an overridden method must return the same type as the base class method it overrides. Thus, a new indexer is declared with the help of the `new` keyword.

Example 14.8 shows this custom collection in action.

14.8 MainApp.cs

```
1    using System;
2
3    public class MainApp {
4       public static void Main(){
5          PersonArrayList surrealists = new PersonArrayList();
6
7          Person p1 = new Person("Rick", "", "Miller", Person.Sex.MALE, new DateTime(1961, 02, 04));
8          Person p2 = new Person("Max", "", "Ernst", Person.Sex.MALE, new DateTime(1891, 04, 02));
9          Person p3 = new Person("Andre", "", "Breton", Person.Sex.MALE, new DateTime(1896, 02, 19));
10         Person p4 = new Person("Roland", "", "Penrose", Person.Sex.MALE, new DateTime(1900, 10, 14));
11         Person p5 = new Person("Lee", "", "Miller", Person.Sex.FEMALE, new DateTime(1907, 04, 23));
12         Person p6 = new Person("Henri-Robert-Marcel", "", "Duchamp", Person.Sex.MALE,
13                                new DateTime(1887, 07, 28));
14
15         surrealists.Add(p1);
16         surrealists.Add(p2);
17         surrealists.Add(p3);
18         surrealists.Add(p4);
19         surrealists.Add(p5);
20         surrealists.Add(p6);
21
22         for(int i=0; i<surrealists.Count; i++){
23            Console.WriteLine(surrealists[i].FullNameAndAge);
24         }
25
26         surrealists.Remove(p1);
27         Console.WriteLine("---------------------------------------");
28
29         for(int i=0; i<surrealists.Count; i++){
30            Console.WriteLine(surrealists[i].FullNameAndAge);
31         }
32
33      } // end Main()
34   } // end MainApp
```

Referring to Example 14.8 — the important point to note here is there is no need to cast when using the PersonArrayList indexer in the bodies of the `for` statements that begin on lines 22 and 29. On line 26 the ArrayList.Remove() method is used to remove the object referenced by p1 from the collection. Figure 14-19 gives the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\Extended_ArrayList>mainapp
Rick  Miller 46
Max  Ernst 116
Andre  Breton 111
Roland  Penrose 107
Lee  Miller 100
Henri-Robert-Marcel  Duchamp 120
---------------------------------------
Max  Ernst 116
Andre  Breton 111
Roland  Penrose 107
Lee  Miller 100
Henri-Robert-Marcel  Duchamp 120

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\Extended_ArrayList>_
```

Figure 14-19: Results of Running Example 14.8

Using Generic Collection Classes – .NET 2.0 and Beyond

The release of the .NET 2.0 framework brought with it generics and generic collections. In this section I show you how generic collection classes can help you write cleaner code and eliminate the need to cast objects upon their retrieval from a collection. I start by showing you how to use the List<T> collection which, as you saw in Table 14-1, is the generic replacement for the ArrayList class. I will also show you how to implement a custom KeyedCollection class.

List<T>: Look Ma, No More Casting!

The List<T> generic collection class is the direct replacement for the ArrayList collection class. It's easy to use the List<T> generic collection. The T that appears between the left and right angle brackets represents the replacement type. You can think of the T as acting like a placeholder for a field in a mail-merge document. Anywhere the T appears, the type that's actually substituted for T will appear in its place when the collection object is created. For example, if you examine the List<T> class's MSDN documentation, you'll find that its Add() method signature is declared to be:

```
public void Add(T item)
```

The parameter named *item* is of type T. So, if you want to create a List collection object that stores Person objects you would do the following:

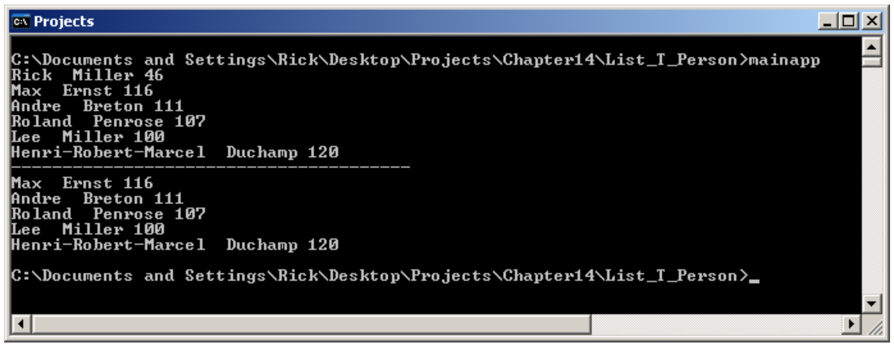
```
List<Person> person_list = new List<Person>();
```

The compiler substitutes the type Person everywhere in the List class the symbol T appears. Thus, its Add() method now accepts only Person-type objects. Let's see this in action. Example 14.9 gives the code for a program that uses the List<T> collection class to store Person objects.

14.9 MainApp.cs (List<T> version)

```
1    using System;
2    using System.Collections.Generic;
3
4    public class MainApp {
5       public static void Main(){
6          List<Person> surrealists = new List<Person>();
7
8          Person p1 = new Person("Rick", "", "Miller", Person.Sex.MALE, new DateTime(1961, 02, 04));
9          Person p2 = new Person("Max", "", "Ernst", Person.Sex.MALE, new DateTime(1891, 04, 02));
10         Person p3 = new Person("Andre", "", "Breton", Person.Sex.MALE, new DateTime(1896, 02, 19));
11         Person p4 = new Person("Roland", "", "Penrose", Person.Sex.MALE, new DateTime(1900, 10, 14));
12         Person p5 = new Person("Lee", "", "Miller", Person.Sex.FEMALE, new DateTime(1907, 04, 23));
13         Person p6 = new Person("Henri-Robert-Marcel", "", "Duchamp", Person.Sex.MALE,
14                                   new DateTime(1887, 07, 28));
15
16         surrealists.Add(p1);
17         surrealists.Add(p2);
18         surrealists.Add(p3);
19         surrealists.Add(p4);
20         surrealists.Add(p5);
21         surrealists.Add(p6);
22
23         for(int i=0; i<surrealists.Count; i++){
24            Console.WriteLine(surrealists[i].FullNameAndAge);
25         }
26
27         surrealists.Remove(p1);
28         Console.WriteLine("---------------------------------------");
29
30         for(int i=0; i<surrealists.Count; i++){
31            Console.WriteLine(surrealists[i].FullNameAndAge);
32         }
33
34      } // end Main()
35   } // end MainApp
```

Referring to Example 14.9 — note the only differences between this example and the code shown in Example 14.8 is the addition of the *using* directive on line 2 to include the System.Collections.Generic namespace. Also, I changed the type of the *surrealists* reference on line 6 from PersonArrayList to List<T>. Figure 14.20 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\List_T_Person>mainapp
Rick  Miller 46
Max  Ernst 116
Andre  Breton 111
Roland  Penrose 107
Lee  Miller 100
Henri-Robert-Marcel  Duchamp 120
--------------------------------------
Max  Ernst 116
Andre  Breton 111
Roland  Penrose 107
Lee  Miller 100
Henri-Robert-Marcel  Duchamp 120

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\List_T_Person>_
```

Figure 14-20: Results of Running Example 14.9

Implementing KeyedCollection<TKey, TItem>

You will find the KeyedCollection<TKey, TItem> in the System.Collections.ObjectModel namespace. In this section, I want to show you how to use this particular class because you can't just use it directly like you can List<T>. It is an abstract class and is meant to be used as the basis for a custom collection. It contains one abstract method, GetKeyForItem(), that you must override in order to properly extract the key from the item being inserted into the collection.

To briefly refiew, the KeyedCollection class functions like a cross between a list and a dictionary. When an item is inserted into the collection, it gets put into an array, just like items inserted into List<T>. When the item is inserted, a key is extracted from it and inserted into an internal dictionary collection. By default, the internal dictionary is created upon the insertion of the first item into the collection, however, by using another version of the KeyedCollection constructor, you can specify the number of items the collection must contain before the internal dictionary is created. This may increase search performance when the collection contains a small number of items.

You can access individual elements in a KeyedCollection in two ways: 1) by array indexing using an integer like you would normally do in an array or a collection that implements the IList<T> interface, or 2) by using an indexer that takes a key as an argument.

Example 14.10 gives the code for a custom KeyedCollection class named PersonKeyedCollection.

14.10 PersonKeyedCollection.cs

```
1    using System;
2    using System.Collections.ObjectModel;
3
4    public class PersonKeyedCollection : KeyedCollection<String, Person> {
5
6       public PersonKeyedCollection():base(){ }
7
8       protected override String GetKeyForItem(Person person){
9          return person.BirthDay.ToString();
10      }
11   }
```

Referring to Example 14.10 — you might be surprised at how little coding it takes to implement the custom KeyedCollection. Note the `using` directive on line 2 specifies the System.Collections.ObjectModel namespace. Also note on line 4 how the class is declared by specifying the actual types of the keys and items. In this example, keys are of type String and items are of type Person.

The overridden GetKeyForItem() method appears on line 8. It simply returns the string representation of a Person object's Birthday property. The GetKeyForItem() method is protected and as such is only used internally by the KeyedCollection class when items are inserted into the collection.

Example 14.11 demonstrates how to use a KeyedCollection.

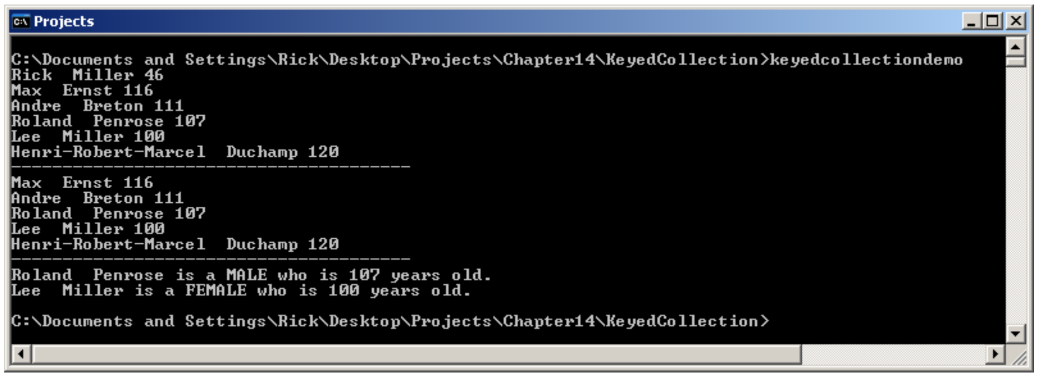
14.11 KeyedCollectionDemo.cs

```
using System;

public class KeyedCollectionDemo {
   public static void Main(){
      PersonKeyedCollection surrealists = new PersonKeyedCollection();

      Person p1 = new Person("Rick", "", "Miller", Person.Sex.MALE, new DateTime(1961, 02, 04));
      Person p2 = new Person("Max", "", "Ernst", Person.Sex.MALE, new DateTime(1891, 04, 02));
9         Person p3 = new Person("Andre", "", "Breton", Person.Sex.MALE, new DateTime(1896, 02, 19));
10        Person p4 = new Person("Roland", "", "Penrose", Person.Sex.MALE, new DateTime(1900, 10, 14));
11        Person p5 = new Person("Lee", "", "Miller", Person.Sex.FEMALE, new DateTime(1907, 04, 23));
12        Person p6 = new Person("Henri-Robert-Marcel", "", "Duchamp", Person.Sex.MALE,
13                               new DateTime(1887, 07, 28));
14
15        surrealists.Add(p1);
16        surrealists.Add(p2);
17        surrealists.Add(p3);
18        surrealists.Add(p4);
19        surrealists.Add(p5);
20        surrealists.Add(p6);
21
22        for(int i=0; i<surrealists.Count; i++){
23          Console.WriteLine(surrealists[ i] .FullNameAndAge);
24        }
25
26        surrealists.Remove(p1);
27        Console.WriteLine("----------------------------------------");
28
29        for(int i=0; i<surrealists.Count; i++){
30          Console.WriteLine(surrealists[ i] .FullNameAndAge);
31        }
32
33        Console.WriteLine("----------------------------------------");
34        Console.WriteLine(surrealists[ (new DateTime(1900, 10, 14)).ToString()] );
35        Console.WriteLine(surrealists[ (new DateTime(1907, 04, 23)).ToString()] );
36
37     }  // end Main()
38   }  // end KeyedCollectionDemo class
```

Referring to Example 14.11 — this codes looks a whole lot like that given in Example 14.9 with a few notable exceptions. The surrealists reference type is changed from List<T> to PersonKeyedCollection. Person objects are created and inserted into the collection using the Add() method as usual. Individual collection elements are accessed via an indexer in the fashion of a list.

Lines 34 and 35 demonstrate the use of the overloaded indexer to find elements by their key. In this case, we can find a particular surrealist by his or her birthday string.

Figure 14.21 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\KeyedCollection>keyedcollectiondemo
Rick  Miller 46
Max  Ernst 116
Andre  Breton 111
Roland  Penrose 107
Lee  Miller 100
Henri-Robert-Marcel  Duchamp 120
----------------------------------------
Max  Ernst 116
Andre  Breton 111
Roland  Penrose 107
Lee  Miller 100
Henri-Robert-Marcel  Duchamp 120
----------------------------------------
Roland  Penrose is a MALE who is 107 years old.
Lee  Miller is a FEMALE who is 100 years old.

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\KeyedCollection>
```

Figure 14-21: Results of Running Example 14.11

Quick Review

The List<T> generic collection class is the direct replacement for the ArrayList collection class. It's easy to use the List<T> generic collection. The T that appears between the left and right angle brackets represents the replacement type. You can think of the T as acting like a placeholder for a field in a mail-merge document. Anywhere the T appears, the type that's actually substituted for T will appear in its place when the collection object is created.

You can find the KeyedCollection<TKey, TItem> in the System.Collections.ObjectModel namespace. It is an abstract class and is meant to be used as the basis for a custom collection. It contains one abstract method, GetKeyForItem(), that you must override to properly extract the key from the item being inserted into the collection.

Special Operations On Collections

Collections exist to make it easier for you to manipulate the objects they contain. In this section, I show you how to sort a list using several different versions of its overloaded Sort() method. I then show you how to extract the contents of a collection into an array.

Sorting A List

The List<T> collection provides a Sort() method that makes sorting its contents easy. However, before you call the Sort() method, you need to be aware of what types of objects the list contains. You must do this because not all objects are naturally sortable.

To make an object sortable, its class or structure type must implement the IComparable<T> interface. If you don't own the source code for a particular class or structure and wish to sort them using List<T>'s Sort() method, then you can implement a Comparer<T> object that defines how such objects are to be compared with each other. I explain how to do this in the following sections.

Implementing System.IComparable<T>

Fundamental data types provided by the .NET Framework implement the System.IComparable or System.IComparable<T> interface and are therefore sortable. When you create a user-defined class or structure, you must ask yourself, as part of the design process, "Will objects of this type be compared with each other or with other types of objects?" If the answer is yes, then that class or structure should implement the IComparable<T> interface.

The IComparable<T> interface declares one method, CompareTo(). This method is automatically called during collection or array sort operations. If you want your user-defined types to sort correctly, you must provide an implementation for the CompareTo() method. How one object is compared against another is entirely up to you since you're the programmer. It's fun having all the power!

Example 14.12 shows the Person class modified to implement the IComparable<T> interface.

14.12 Person.cs (implementing IComparable<T>)

```
using System;

public class Person : IComparable<Person> {

  //enumeration
  public enum Sex {MALE, FEMALE};

  // private instance fields
  private String   _firstName;
  private String   _middleName;
  private String   _lastName;
  private Sex      _gender;
  private DateTime _birthday;


  //private default constructor
  private Person(){}

  public Person(String firstName, String middleName, String lastName,
                Sex gender, DateTime birthday){
     FirstName = firstName;
     MiddleName = middleName;
     LastName = lastName;
     Gender = gender;
     BirthDay = birthday;
  }

  // public properties
  public String FirstName {
    get { return _firstName; }
    set { _firstName = value; }
  }

  public String MiddleName {
    get { return _middleName; }
    set { _middleName = value; }
37     }
38
39     public String LastName {
40       get { return _lastName; }
41       set { _lastName = value; }
42     }
43
44     public Sex Gender {
45       get { return _gender; }
46       set { _gender = value; }
47     }
48
49     public DateTime BirthDay {
50       get { return _birthday; }
51       set { _birthday = value; }
52     }
53
54      public int Age {
55         get {
56            int years = DateTime.Now.Year - _birthday.Year;
57            int adjustment = 0;
58            if((DateTime.Now.Month <= _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
59               adjustment = 1;
60            }
61            return years - adjustment;
62       }
63      }
64
65     public String FullName {
66       get { return FirstName + " " + MiddleName + " " + LastName; }
67     }
68
69     public String FullNameAndAge {
70       get { return FullName + " " + Age; }
71     }
72
73     public override String ToString(){
74       return FullName + " is a " + Gender + " who is " + Age + " years old.";
75     }
76
77     public int CompareTo(Person other){
78        return this.BirthDay.CompareTo(other.BirthDay);
79      }
80
81   } // end Person class
```

Referring to Example 14.12 — the Person class now implements the IComparable<T> interface. The CompareTo() method, starting on line 77, defines how one person object is compared with another. In this case, I am comparing their BirthDay properties. Note that since the DateTime structure implements the IComparable<T> interface (*i.e.*, IComparable<DateTime>) one simply needs to call its version of the CompareTo() method to make the required comparison. But what's getting returned? Good question. I answer it in the next section.

Rules For Implementing The CompareTo(T other) Method

The rules for implementing the CompareTo(T other) method are laid out in Table 14-2.

Return Value	Returned When...
Less than Zero (-1)	This object is less than the *other* parameter
Zero (0)	This object is equal to the *other* parameter
Greater than Zero (1)	This object is greater than the *other* parameter, or, the *other* parameter is null

Table 14-2: Rules For Implementing IComparable<T>.CompareTo(T other) Method

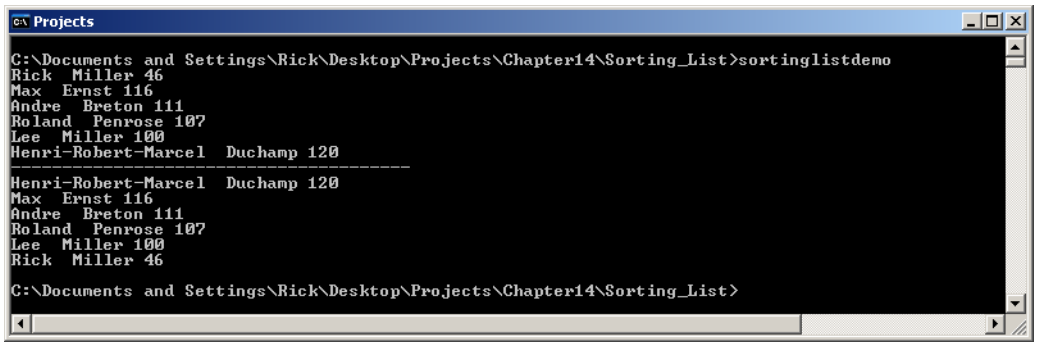
What property, exactly, you compare between objects is strictly dictated by the program's design. In the case of the Person class, I chose to compare BirthDays. In the next section, I'll show you how you would compare LastNames.

Now that the Person class implements the IComparable<T> interface, person objects can be compared against each other. Example 14.13 shows the List<T>.Sort() method in action.

14.13 SortingListDemo.cs

```
1    using System;
2    using System.Collections.Generic;
3
4    public class SortingListDemo {
5       public static void Main(){
6          List<Person> surrealists = new List<Person>();
7
8          Person p1 = new Person("Rick", "", "Miller", Person.Sex.MALE, new DateTime(1961, 02, 04));
9          Person p2 = new Person("Max", "", "Ernst", Person.Sex.MALE, new DateTime(1891, 04, 02));
10         Person p3 = new Person("Andre", "", "Breton", Person.Sex.MALE, new DateTime(1896, 02, 19));
11         Person p4 = new Person("Roland", "", "Penrose", Person.Sex.MALE, new DateTime(1900, 10, 14));
12         Person p5 = new Person("Lee", "", "Miller", Person.Sex.FEMALE, new DateTime(1907, 04, 23));
13         Person p6 = new Person("Henri-Robert-Marcel", "", "Duchamp", Person.Sex.MALE,
14                                new DateTime(1887, 07, 28));
15
16         surrealists.Add(p1);
17         surrealists.Add(p2);
18         surrealists.Add(p3);
19         surrealists.Add(p4);
20         surrealists.Add(p5);
21         surrealists.Add(p6);
22
23         for(int i=0; i<surrealists.Count; i++){
24            Console.WriteLine(surrealists[i].FullNameAndAge);
25         }
26
27         surrealists.Sort();
28         Console.WriteLine("----------------------------------------");
29
30         for(int i=0; i<surrealists.Count; i++){
31            Console.WriteLine(surrealists[i].FullNameAndAge);
32         }
33
34      } // end Main()
35   } // end SortingListDemo
```

Referring to Example 14.13 — the Sort() method is called on line 27. Figure 14-22 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\Sorting_List>sortinglistdemo
Rick  Miller 46
Max  Ernst 116
Andre  Breton 111
Roland  Penrose 107
Lee  Miller 100
Henri-Robert-Marcel  Duchamp 120
----------------------------------------
Henri-Robert-Marcel  Duchamp 120
Max  Ernst 116
Andre  Breton 111
Roland  Penrose 107
Lee  Miller 100
Rick  Miller 46

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\Sorting_List>
```

Figure 14-22: Results of Running Example 14.13

Extending Comparer<T>

What if you don't own the source code to the objects you want to compare? No problem. Simply implement a custom comparer object by extending the System.Collections.Generic.Comparer<T> class and providing an implementation for its Compare(T x, T y) method. Example 14.14 shows how a custom comparer might look. This particular example compares two Person objects.

14.14 PersonComparer.cs

```
using System.Collections.Generic;

public class PersonComparer : Comparer<Person> {

   /*********************************
      Return -1 if p1 < p2 or p1 == null
      Return  0 if p1 == p2
```

```
       Return +1 if p1 > p2 or p2 == null
    ***********************************/
    public override int Compare(Person p1, Person p2){
       if(p1 == null) return -1;
       if(p2 == null) return  1;

       return p1.LastName.CompareTo(p2.LastName);
    }
 }
```

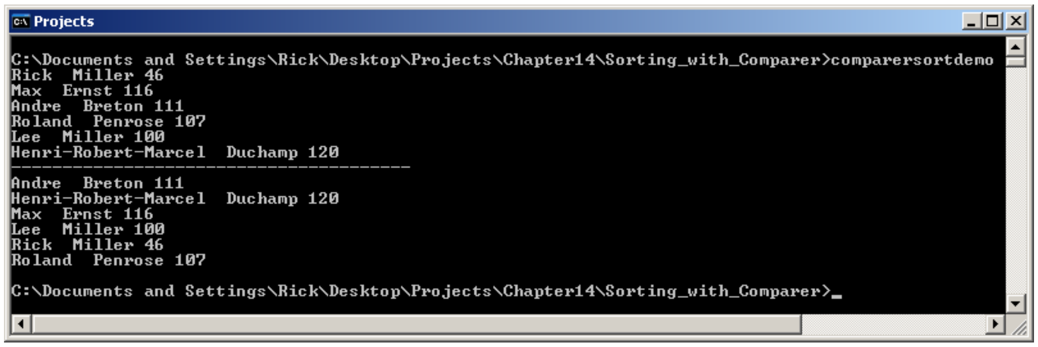
Referring to Example 14.14 — the PersonComparer class extends Comparer<T> (i.e., Comparer<Person>) and provides an overriding implementation for its Compare(T x, T y) method. In this example, I have renamed the parameters p1 and p2. Note that since I am comparing strings, I can simply call the String.CompareTo() method to actually perform the comparison.

Example 14.15 shows how the PersonComparer class is used to sort a list of Person objects.

14.15 ComparerSortDemo.cs

```
1    using System;
2    using System.Collections.Generic;
3
4    public class ComparerSortDemo {
5       public static void Main(){
6          List<Person> surrealists = new List<Person>();
7
8          Person p1 = new Person("Rick", "", "Miller", Person.Sex.MALE, new DateTime(1961, 02, 04));
9          Person p2 = new Person("Max", "", "Ernst", Person.Sex.MALE, new DateTime(1891, 04, 02));
10         Person p3 = new Person("Andre", "", "Breton", Person.Sex.MALE, new DateTime(1896, 02, 19));
11         Person p4 = new Person("Roland", "", "Penrose", Person.Sex.MALE, new DateTime(1900, 10, 14));
12         Person p5 = new Person("Lee", "", "Miller", Person.Sex.FEMALE, new DateTime(1907, 04, 23));
13         Person p6 = new Person("Henri-Robert-Marcel", "", "Duchamp", Person.Sex.MALE,
14                                new DateTime(1887, 07, 28));
15
16         surrealists.Add(p1);
17         surrealists.Add(p2);
18         surrealists.Add(p3);
19         surrealists.Add(p4);
20         surrealists.Add(p5);
21         surrealists.Add(p6);
22
23         for(int i=0; i<surrealists.Count; i++){
24           Console.WriteLine(surrealists[ i] .FullNameAndAge);
25         }
26
27         surrealists.Sort(new PersonComparer());
28         Console.WriteLine("----------------------------------------");
29
30         for(int i=0; i<surrealists.Count; i++){
31           Console.WriteLine(surrealists[ i] .FullNameAndAge);
32         }
33
34      }  // end Main()
35   }  // end SortingListDemo
```

Referring to Example 14.15 — an overloaded version of the List<T>.Sort() method is called on line 27. This version of the Sort() method takes a Comparer<T> object as an argument. Figure 14-23 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\Sorting_with_Comparer>comparersortdemo
Rick  Miller 46
Max  Ernst 116
Andre  Breton 111
Roland  Penrose 107
Lee  Miller 100
Henri-Robert-Marcel  Duchamp 120
----------------------------------------
Andre  Breton 111
Henri-Robert-Marcel  Duchamp 120
Max  Ernst 116
Lee  Miller 100
Rick  Miller 46
Roland  Penrose 107

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\Sorting_with_Comparer>_
```

Figure 14-23: Results of Running Example 14.15

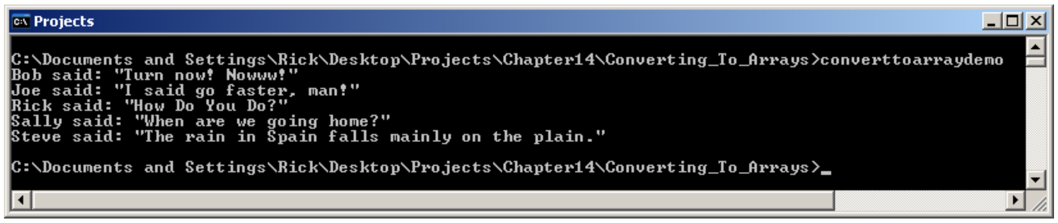
Converting A Collection Into An Array

It's often handy to convert the contents of a collection into an array. This holds especially true for collections that cannot be manipulated like lists. (*i.e.*, via indexers) Collection classes that implement the ICollection<T> interface provide a CopyTo() method that copies the contents of the collection into an existing array whose length is equal to that of the number of collection elements. Example 14.16 offers a short program that shows how to extract an array of KeyValuePair<TKey, TValue> values from a SortedDictionary<TKey, TValue> collection.

14.16 ConvertToArrayDemo.cs

```
1    using System;
2    using System.Collections.Generic;
3
4    public class ConvertToArrayDemo {
5       public static void Main(){
6          String[] names = { "Rick",
7                             "Sally",
8                             "Joe",
9                             "Bob",
10                            "Steve" };
11
12         SortedDictionary<String, String> quotes = new SortedDictionary<String, String>();
13
14         quotes.Add(names[0], "How Do You Do?");
15         quotes.Add(names[1], "When are we going home?");
16         quotes.Add(names[2], "I said go faster, man!");
17         quotes.Add(names[3], "Turn now! Nowww!");
18         quotes.Add(names[4], "The rain in Spain falls mainly on the plain.");
19
20         KeyValuePair<String, String>[] quote_array = new KeyValuePair<String, String>[quotes.Count];
21         quotes.CopyTo(quote_array, 0);
22
23         for(int i = 0; i<quote_array.Length; i++){
24            Console.WriteLine(quote_array[i].Key + " said: \"" + quote_array[i].Value + "\"" );
25         }
26
27      }// end Main()
28   } // end ConvertToArrayDemo class
```

Referring to Example 14.16 — first, the program creates an array of strings that's used to store five names. The SortedDictionary object is created on line 12. On lines 14 through 18, the names array provides the keys for each quote. Note that as each key/value pair is inserted into the SortedDictionary, it is inserted into its proper sorted order according to the key's value. (Ergo, classes used as keys must implement IComparable<T>) On lines 20 and 21, the array of sorted KeyValuePair elements is extracted into an array of KeyValuePairs equal in length to the collection's count. The quotes array is then used in the `for` statement beginning on line 23 to print the values of both the key and its associated value. In this case they are both strings. Figure 14-24 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\Converting_To_Arrays>converttoarraydemo
Bob said: "Turn now! Nowww!"
Joe said: "I said go faster, man!"
Rick said: "How Do You Do?"
Sally said: "When are we going home?"
Steve said: "The rain in Spain falls mainly on the plain."

C:\Documents and Settings\Rick\Desktop\Projects\Chapter14\Converting_To_Arrays>_
```

Figure 14-24: Results of Running Example 14.16

Quick Review

To make an object sortable, its class or structure type must implement the IComparable<T> interface. If you don't own the source code for a particular class or structure and wish to sort them using List<T>'s Sort() method, then you can implement a Comparer<T> object that defines how such objects are to be compared with each other.

It's often handy to convert the contents of a collection into an array. Collection classes that implement the ICollection<T> interface provide a CopyTo() method that is used to copy the contents of the collection into an existing array whose length is equal to that of the number of collection elements.

Summary

The .NET collections framework can potentially save you a lot of time and hassle. It contains classes, structures, and interfaces designed to make it easy to manipulate collections of objects. The .NET 2.0 framework introduced generic collections and improved performance.

An *array* is a contiguous allocation of objects in memory. An array-based collection offers quick element access but slow element insertion, especially if the collection's underlying array must be resized and its contents shifted to accommodate the insertion.

A *linked list* consists of individual *nodes* linked to each other via references. To traverse a linked list, you must start at the beginning, or the end (head or tail) and follow each element to the next. Linked list-based collections can conserve memory space because memory need only be allocated on each object insertion. Insertions into linked list-based collections are relatively quick, but element access is relatively slow due to the need to traverse the list.

A *chained hash table* is a cross between an array and a linked list and allows element insertion with key/value pairs. A hash function performed on the key determines the value's location in the hash table. A collision is said to occur when two keys produce the same hash code. When this happens, the values are chained together in a linked list-like structure. A hash function that produces a uniform distribution over all the keys is a critical feature of a hash table.

A *red-black tree* is a self-balancing binary tree. Insertions into a red-black tree take some time because of element ordering and balancing operations. Element access time for a red-black tree-based collection is fairly quick.

"Old-school" style .NET collections classes store only object references and require casting when elements are retrieved. You should favor the use of generic collections as they offer strong element typing on insertion and retrieval as well as improved performance. The classes found in the System.Collections.ObjectModel namespace serve as the basis for user-defined custom collections. The System.Collections.Specialized namespace contains classes and structures designed to manage unique collections.

The List<T> generic collection class is the direct replacement for the ArrayList collection class. It's easy to use the List<T> generic collection. The T that appears between the left and right angle brackets represents the replacement type. You can think of the T as acting like a placeholder for a field in a mail-merge document. Anywhere the T appears, the type that's actually substituted for T will appear in its place when the collection object is created.

You can find the KeyedCollection<TKey, TItem> in the System.Collections.ObjectModel namespace. It is an abstract class and is meant to be used as the basis for a custom collection. It contains one abstract method, GetKeyForItem(), that you must override to properly extract the key from the item being inserted into the collection.

To make an object sortable, its class or structure type must implement the IComparable<T> interface. If you don't own the source code for a particular class or structure and wish to sort them using List<T>'s Sort() method, then you can implement a Comparer<T> object that defines how such objects are to be compared with each other.

It's often handy to convert the contents of a collection into an array. Collection classes that implement the ICollection<T> interface provide a CopyTo() method that is used to copy the contents of the collection into an existing array whose length is equal to that of the number of collection elements.

Skill-Building Exercises

1. **API Drill:** Access the MSDN online documentation and explore all four .NET collections namespaces. List each class, structure, and interface you find there and write a brief description of its purpose and use.

2. **API Drill:** Research the ICollection<T> interface and answer the following question: What exception will be thrown if the number of collection elements exceeds the size of the array supplied via the CopyTo() method?

3. **API Drill:** What methods are available in the List<T> class that let you extract its elements into an array?

4. **Programming Drill:** Compile and run all the example programs presented in this chapter and note the results. Modify the programs as you see fit to utilize more of the methods provided by each collection class.

5. **Programming Drill:** Explore the System.Collections.Generic and System.Collections.Specialized namespaces, select several collections not discussed in this chapter and use each in a short program.

Suggested Projects

1. **Employee Management Application:** Create a GUI-based application that lets you create and manage Employees. Use the Person, IPayable, Employee, HourlyEmployee, and SalariedEmployee code originally presented in Chapter 11. Modify the Employee class to implement the IComparable<T> interface or, alternatively, create a Comparer<T> class that compares two Employee objects. Your program should allow you to create HourlyEmployee and SalariedEmployee objects and insert them into a generic collection that holds Employee type objects. Each time you create a new employee display a sorted list of employees.

2. **Collectables Application:** Are you a collector? Do you have lots of books, CDs, DVDs, or antique typewriters? Do you have a tiny little spoon from every state of the Union, or a book of matches from every restaurant or nightclub you've visited? Whatever your passion, write a program that helps you manage your collection. Create a class that captures the properties of your particular collectable. Create a GUI-based application that lets you enter information about each item in your collection. **Extra credit:** Add the capability to upload an image of the item and persist the collection object to disk so you don't lose the data you entered. **Note:** The serialization of objects to disk is discussed in Chapter 17 - File I/O.

Self-Test Questions

1. What must you do to elements retrieved from an "old-school" collection before accessing its members or performing an operation on it other than those defined by System.Object?

2. What are the general performance characteristics of an array-based collection?

3. What are the general performance characteristics of a linked list-based collection?

4. What are the general performance characteristics of a hash table-based collection?

5. What are the general performance characteristics of a red-black tree-based collection?

6. What does the T represent in a generic collection?

7. What interface must a class or structure implement before it can be sorted by the List<T>.Sort() method?

8. How can you sort objects using the List<T>.Sort() method if you don't have access to a class's source code?

9. Why would you want to use generic collections vs. earlier "old-school" collections?

10. What method do you use to extract a collection's elements into an array?

References

Microsoft Developer Network (MSDN) .NET Framework 3.0 Documentation [http://www.msdn.com]

John Franco. *Red-Black Tree Demonstration Applet*. [http://www.ececs.uc.edu/~franco/C321/html/RedBlack/redblack.html]

Thomas H. Cormen, et. al. *Introduction To Algorithms*. The MIT Press, Cambridge, MA. ISBN: 0-262-03141-8

Donald E. Knuth. *The Art of Computer Programming, Volume 1, Fundamental Algorithms*, *Third Edition*. Addison-Wesley, Reading, MA. ISBN: 0-201-89685-0

Donald E. Knuth. *The Art of Computer Programming, Volume 3, Sorting and Searching*, *Second Edition*. Addison-Wesley, Reading, MA. ISBN: 0-201-89685-0

Rick Miller, Raffi Kasparian. *Java For Artists: The Art, Philosophy, And Science Of Object-Oriented Programming*. Pulp Free Press, Falls Church, VA. ISBN: 1-932504-05-2

Notes