CHAPTER 1

An Introduction to Collections, Generics, and the Timing Class

This book discusses the development and implementation of data structures and algorithms using C#. The data structures we use in this book are found in the .NET Framework class library System.Collections. In this chapter, we develop the concept of a collection by first discussing the implementation of our own Collection class (using the array as the basis of our implementation) and then by covering the Collection classes in the .NET Framework.

An important addition to C# 2.0 is generics. Generics allow the C# programmer to write one version of a function, either independently or within a class, without having to overload the function many times to allow for different data types. C# 2.0 provides a special library, System.Collections.Generic, that implements generics for several of the System.Collections data structures. This chapter will introduce the reader to generic programming.

Finally, this chapter introduces a custom-built class, the Timing class, which we will use in several chapters to measure the performance of a data structure and/or algorithm. This class will take the place of Big O analysis, not because Big O analysis isn't important, but because this book takes a more practical approach to the study of data structures and algorithms.

COLLECTIONS DEFINED

A collection is a structured data type that stores data and provides operations for adding data to the collection, removing data from the collection, updating data in the collection, as well as operations for setting and returning the values of different attributes of the collection.

Collections can be broken down into two types: linear and nonlinear. A linear collection is a list of elements where one element follows the previous element. Elements in a linear collection are normally ordered by position (first, second, third, etc.). In the real world, a grocery list is a good example of a linear collection; in the computer world (which is also real), an array is designed as a linear collection.

Nonlinear collections hold elements that do not have positional order within the collection. An organizational chart is an example of a nonlinear collection, as is a rack of billiard balls. In the computer world, trees, heaps, graphs, and sets are nonlinear collections.

Collections, be they linear or nonlinear, have a defined set of properties that describe them and operations that can be performed on them. An example of a collection property is the collections Count, which holds the number of items in the collection. Collection operations, called methods, include Add (for adding a new element to a collection), Insert (for adding a new element to a collection at a specified index), Remove (for removing a specified element from a collection), Clear (for removing all the elements from a collection), Contains (for determining if a specified element is a member of a collection), and IndexOf (for determining the index of a specified element in a collection).

COLLECTIONS DESCRIBED

Within the two major categories of collections are several subcategories. Linear collections can be either direct access collections or sequential access collections, whereas nonlinear collections can be either hierarchical or grouped. This section describes each of these collection types.

Direct Access Collections

The most common example of a direct access collection is the array. We define an array as a collection of elements with the same data type that are directly accessed via an integer index, as illustrated in Figure 1.1.

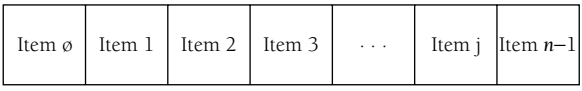

FIGURE 1.1. Array.

Arrays can be static so that the number of elements specified when the array is declared is fixed for the length of the program, or they can be dynamic, where the number of elements can be increased via the ReDim or ReDim Preserve statements.

In C#, arrays are not only a built-in data type, they are also a class. Later in this chapter, when we examine the use of arrays in more detail, we will discuss how arrays are used as class objects.

We can use an array to store a linear collection. Adding new elements to an array is easy since we simply place the new element in the first free position at the rear of the array. Inserting an element into an array is not as easy (or efficient), since we will have to move elements of the array down in order to make room for the inserted element. Deleting an element from the end of an array is also efficient, since we can simply remove the value from the last element. Deleting an element in any other position is less efficient because, just as with inserting, we will probably have to adjust many array elements up one position to keep the elements in the array contiguous. We will discuss these issues later in the chapter. The .NET Framework provides a specialized array class, ArrayList, for making linear collection programming easier. We will examine this class in Chapter 3.

Another type of direct access collection is the string. A string is a collection of characters that can be accessed based on their index, in the same manner we access the elements of an array. Strings are also implemented as class objects in C#. The class includes a large set of methods for performing standard operations on strings, such as concatenation, returning substrings, inserting characters, removing characters, and so forth. We examine the String class in Chapter 8.

C# strings are immutable, meaning once a string is initialized it cannot be changed. When you modify a string, a copy of the string is created instead of changing the original string. This behavior can lead to performance degradation in some cases, so the .NET Framework provides a StringBuilder class that enables you to work with mutable strings. We'll examine the StringBuilder in Chapter 8 as well.

The final direct access collection type is the struct (also called structures and records in other languages). A struct is a composite data type that holds data that may consist of many different data types. For example, an employee

record consists of employee' name (a string), salary (an integer), identification number (a string, or an integer), as well as other attributes. Since storing each of these data values in separate variables could become confusing very easily, the language provides the struct for storing data of this type.

A powerful addition to the C# struct is the ability to define methods for performing operations stored on the data in a struct. This makes a struct somewhat like a class, though you can't inherit or derive a new type from a structure. The following code demonstrates a simple use of a structure in C#:

```
using System;

  public struct Name {

  private string fname, mname, lname;

  public Name(string first, string middle, string last) {
    fname = first;
    mname = middle;
    lname = last;
  }

  public string firstName {
    get {
      return fname;
    }

    set {
      fname = firstName;
    }
  }
  public string middleName {
    get {
      return mname;
    }

    set {
      mname = middleName;
    }
  }

  public string lastName {
    get {
```

```
      return lname;
    }

    set {
      lname = lastName;
    }
  }

  public override string ToString() {
    return (String.Format("{0} {1} {2}", fname, mname,
            lname));
  }

  public string Initials() {
   return (String.Format("{0}{1}{2}", fname.Substring(0,1),
          mname.Substring(0,1), lname.Substring(0,1)));
  }
}

public class NameTest {
  static void Main() {
    Name myName = new Name("Michael", "Mason", "McMillan");
    string fullName, inits;
    fullName = myName.ToString();
    inits = myName.Initials();
    Console.WriteLine("My name is {0}.", fullName);
    Console.WriteLine("My initials are {0}.", inits);
  }
}
```

Although many of the elements in the .NET environment are implemented as classes (such as arrays and strings), several primary elements of the language are implemented as structures, such as the numeric data types. The Integer data type, for example, is implemented as the Int32 structure. One of the methods you can use with Int32 is the Parse method for converting the string representation of a number into an integer. Here's an example:

```
using System;

public class IntStruct {
  static void Main() {
```

```
        int num;
        string snum;
        Console.Write("Enter a number: ");
        snum = Console.ReadLine();
        num = Int32.Parse(snum);
        Console.WriteLine(num);
    }
    }
```

Sequential Access Collections

A sequential access collection is a list that stores its elements in sequential order. We call this type of collection a linear list. Linear lists are not limited by size when they are created, meaning they are able to expand and contract dynamically. Items in a linear list are not accessed directly; they are referenced by their position, as shown in Figure 1.2. The first element of a linear list is at the front of the list and the last element is at the rear of the list.

Because there is no direct access to the elements of a linear list, to access an element you have to traverse through the list until you arrive at the position of the element you are looking for. Linear list implementations usually allow two methods for traversing a list—in one direction from front to rear, and from both front to rear and rear to front.

A simple example of a linear list is a grocery list. The list is created by writing down one item after another until the list is complete. The items are removed from the list while shopping as each item is found.

Linear lists can be either ordered or unordered. An ordered list has values in order in respect to each other, as in:

Beata Bernica David Frank Jennifer Mike Raymond Terrill

An unordered list consists of elements in any order. The order of a list makes a big difference when performing searches on the data on the list, as you'll see in Chapter 2 when we explore the binary search algorithm versus a simple linear search.

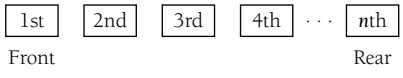

FIGURE 1.2. Linear List.

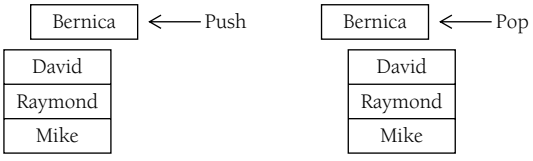

FIGURE 1.3. Stack Operations.

Some types of linear lists restrict access to their data elements. Examples of these types of lists are stacks and queues. A stack is a list where access is restricted to the beginning (or top) of the list. Items are placed on the list at the top and can only be removed from the top. For this reason, stacks are known as Last-in, First-out structures. When we add an item to a stack, we call the operation a push. When we remove an item from a stack, we call that operation a pop. These two stack operations are shown in Figure 1.3.

The stack is a very common data structure, especially in computer systems programming. Stacks are used for arithmetic expression evaluation and for balancing symbols, among its many applications.

A queue is a list where items are added at the rear of the list and removed from the front of the list. This type of list is known as a First-in, First-out structure. Adding an item to a queue is called an EnQueue, and removing an item from a queue is called a Dequeue. Queue operations are shown in Figure 1.4.

Queues are used in both systems programming, for scheduling operating system tasks, and for simulation studies. Queues make excellent structures for simulating waiting lines in every conceivable retail situation. A special type of queue, called a priority queue, allows the item in a queue with the highest priority to be removed from the queue first. Priority queues can be used to study the operations of a hospital emergency room, where patients with heart trouble need to be attended to before a patient with a broken arm, for example.

The last category of linear collections we'll examine are called generalized indexed collections. The first of these, called a hash table, stores a set of data

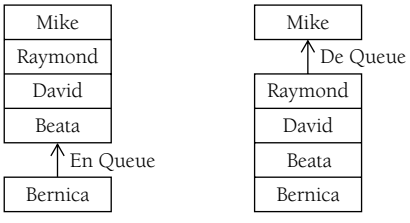

FIGURE 1.4. Queue Operations.

"Paul E. Spencer"
37500
5
"Information Systems"

FIGURE 1.5. A Record To Be Hashed.

values associated with a key. In a hash table, a special function, called a hash function, takes one data value and transforms the value (called the key) into an integer index that is used to retrieve the data. The index is then used to access the data record associated with the key. For example, an employee record may consist of a person's name, his or her salary, the number of years the employee has been with the company, and the department he or she works in. This structure is shown in Figure 1.5. The key to this data record is the employee's name. C# has a class, called HashTable, for storing data in a hash table. We explore this structure in Chapter 10.

Another generalized indexed collection is the dictionary. A dictionary is made up of a series of key–value pairs, called associations. This structure is analogous to a word dictionary, where a word is the key and the word's definition is the value associated with the key. The key is an index into the value associated with the key. Dictionaries are often called associative arrays because of this indexing scheme, though the index does not have to be an integer. We will examine several Dictionary classes that are part of the .NET Framework in Chapter 11.

Hierarchical Collections

Nonlinear collections are broken down into two major groups: hierarchical collections and group collections. A hierarchical collection is a group of items divided into levels. An item at one level can have successor items located at the next lower level.

One common hierarchical collection is the tree. A tree collection looks like an upside-down tree, with one data element as the root and the other data values hanging below the root as leaves. The elements of a tree are called nodes, and the elements that are below a particular node are called the node's children. A sample tree is shown in Figure 1.6.

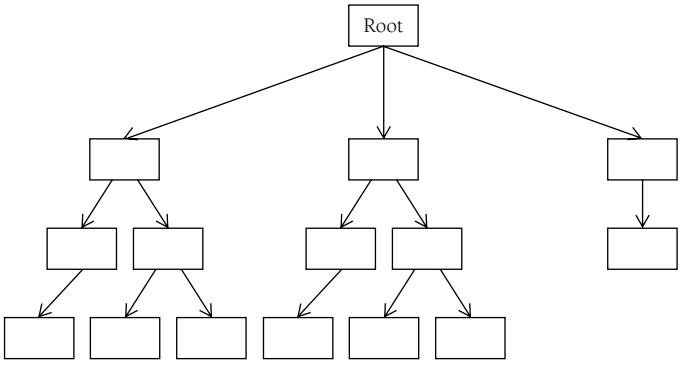

FIGURE 1.6. A Tree Collection.

Trees have applications in several different areas. The file systems of most modern operating systems are designed as a tree collection, with one directory as the root and other subdirectories as children of the root.

A binary tree is a special type of tree collection where each node has no more than two children. A binary tree can become a binary search tree, making searches for large amounts of data much more efficient. This is accomplished by placing nodes in such a way that the path from the root to a node where the data is stored is along the shortest path possible.

Yet another tree type, the heap, is organized so that the smallest data value is always placed in the root node. The root node is removed during a deletion, and insertions into and deletions from a heap always cause the heap to reorganize so that the smallest value is placed in the root. Heaps are often used for sorts, called a heap sort. Data elements stored in a heap can be kept sorted by repeatedly deleting the root node and reorganizing the heap.

Several different varieties of trees are discussed in Chapter 12.

Group Collections

A nonlinear collection of items that are unordered is called a group. The three major categories of group collections are sets, graphs, and networks.

A set is a collection of unordered data values where each value is unique. The list of students in a class is an example of a set, as is, of course, the integers. Operations that can be performed on sets include union and intersection. An example of set operations is shown in Figure 1.7.

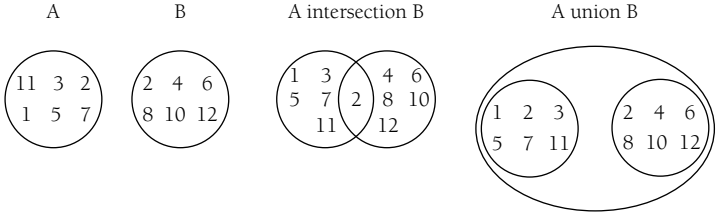

FIGURE 1.7. Set Collection Operations.

A graph is a set of nodes and a set of edges that connect the nodes. Graphs are used to model situations where each of the nodes in a graph must be visited, sometimes in a particular order, and the goal is to find the most efficient way to "traverse" the graph. Graphs are used in logistics and job scheduling and are well studied by computer scientists and mathematicians. You may have heard of the "Traveling Salesman" problem. This is a particular type of graph problem that involves determining which cities on a salesman's route should be traveled in order to most efficiently complete the route within the budget allowed for travel. A sample graph of this problem is shown in Figure 1.8.

This problem is part of a family of problems known as NP-complete problems. This means that for large problems of this type, an exact solution is not known. For example, to find the solution to the problem in Figure 1.8, 10 factorial tours, which equals 3,628,800 tours. If we expand the problem to 100 cities, we have to examine 100 factorial tours, which we currently cannot do with current methods. An approximate solution must be found instead.

A network is a special type of graph where each of the edges is assigned a weight. The weight is associated with a cost for using that edge to move from one node to another. Figure 1.9 depicts a network of cities where the weights are the miles between the cities (nodes).

We've now finished our tour of the different types of collections we are going to discuss in this book. Now we're ready to actually look at how collections

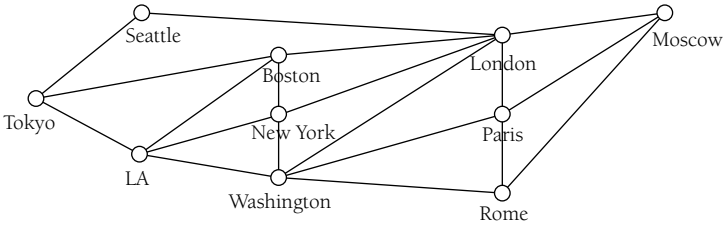

FIGURE 1.8. The Traveling Salesman Problem.