CHAPTER 6

Collections of Objects

YOU LEARNED ABOUT the process of creating objects based on class definitions, a process known as instantiation, in Chapter 3. When we're only creating a few objects, we can afford to declare individualized reference variables for these objects: `Students` `s1`, `s2`, `s3`, perhaps, or `Professors` `profA`, `profB`, `profC`. But, at other times, individualized reference variables are impractical.

- Sometimes, there will be too many objects, as when creating `Course` objects to represent the hundreds of courses in a university's course catalog.

- Worse yet, we may not even ***know*** how many objects of a particular type there will be in advance. With our Student Registration System, for example, we may create a new `Student` object each time a new student logs on for the first time.

Fortunately, OOPLs solve this problem by providing a special category of object called a **collection** that is used to hold and organize other objects.

In this chapter, you'll learn about

- The properties and behaviors of some common collection types

- How collections enable us to model very sophisticated real-world concepts or situations

- How we can define our own collection types

What Are Collections?

We'd like a way to gather up objects as they are created so that we can manage them as a group and operate on them collectively, along with referring to them individually when necessary. For example:

- A `Professor` object may need to step through all `Student` objects registered for a particular `Course` that the professor is teaching in order to compute their grades.

- The Student Registration System (SRS) application as a whole may need to step through all of the `Course` objects in the current schedule of classes to determine which of them don't yet have any students registered for them, possibly to cancel these courses.

We use a special type of object called a **collection** to group other objects. A collection object can hold/contain multiple references to some other type of object. Think of a collection like an egg carton, and the objects it holds like the eggs! Both are objects, but with decidedly different properties.

Because collections are implemented as objects, this implies that

- Collections must be instantiated before they can first be used.
- Collections are defined by classes that in turn define methods for "getting" and "setting" their contents.
- By virtue of being objects, OO collections are encapsulated, and hence take full advantage of information hiding.

Let's discuss each of these three matters in turn.

Collections Must Be Instantiated Before They Can First Be Used

We can't merely declare a collection:

```
CollectionType c;
```

For example:

```
ArrayList c;
```

All this does is to declare a reference variable of type *`CollectionType`*. Until we "hand" `c` a *`CollectionType`* ***object*** to ***refer*** to, `c` is said to have the value `null`.

> `ArrayList` *is one of C#'s predefined collection types, defined by the .NET Framework Class Library (FCL). We'll introduce the* `ArrayList` *class in this chapter, and we'll then go into greater detail about* `ArrayList`*s, along with several other collection types, in Chapter 13.*

We have to take the distinct step of using the `new` operator to actually create an empty *`CollectionType`* object in memory, as follows:

```
c = new CollectionType();
```

For example:

```
c = new ArrayList();
```

Think of the newly created *`CollectionType`* object as an ***empty*** "egg carton," and the reference variable `c` as the handle that allows us to locate and access (reference) this egg carton whenever we'd like.

Then, as we instantiate objects ("eggs"), we'll place ***their references*** into the various egg carton compartments:

```
Student s = new Student();
// Pseudocode.
c.Add(s);
```

So, rather than thinking of the objects as eggs that are physically placed inside of the egg carton compartments, we should really think of the objects as balloons whose strings are tied inside the egg carton compartments, as illustrated in Figure 6-1. That is, the objects themselves live physically ***outside*** of the collection, but can be located through their references, which are stored ***within*** the collection.

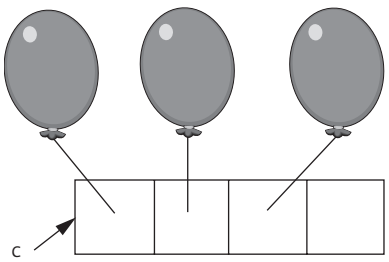


Figure 6-1. A collection organizes object references.

Thus, perhaps a more appropriate analogy than a collection as an egg carton would be that of a collection as an address book: we make an entry in the address book (collection) for each of the persons (objects) that we wish to contact, but the actual persons themselves are physically remote (see Figure 6-2).

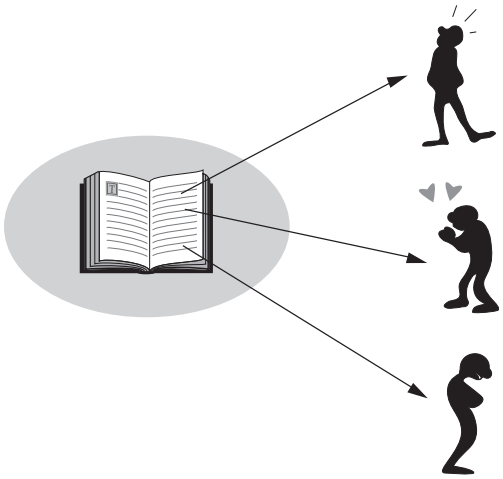


Figure 6-2. A collection **references** *objects, which live separately in memory.*

> *Note that C# collections can also hold references to value-type variables (*`int`*,* `float`*, etc.) because in the C# language, value-type variables are actually implemented as objects. This is in contrast to Java, where simple data types—*`int`*,* `float`*, etc.—are* **not** *objects, and hence cannot be stored "as is" in Java collections.*

Collections Are Defined by Classes

Here is a code snippet that illustrates the use of a collection in C#; we use a bit of pseudocode here to emphasize the common features of collections.

```
// Instantiate a collection object (pseudocode).
CollectionType x = new CollectionType();

// Create a few Student objects.
Student a = new Student();
Student b = new Student();
Student c = new Student();

// Store all three students in the collection by calling the appropriate
// method for adding objects to the collection ...
x.Add(a);
x.Add(b);
x.Add(c);

// ... and then retrieve the first one.
x.Retrieve(0); // we typically start counting at 0.
```

OO Collections Are Encapsulated

We don't need to know the private details of how object references are stored internally to a specific type of collection in order to use the collection properly; we only need to know a collection's public features—in particular, its method headers and properties—in order to choose an appropriate collection type for a particular situation and to use it effectively.

> *This is a tiny bit misleading: in the case of* **huge** *collections, it* **is** *helpful to know a little bit about the inner workings of various collection types so as to choose the one that is most efficient; we'll consider this matter further in Chapter 13.*

Virtually all collections, regardless of type and regardless of the programming language in which they are implemented, provide, at a minimum, methods for

- Adding object (reference)s
- Removing object (reference)s
- Retrieving specific individual object (reference)s
- Iterating through the object (reference)s in some predetermined order
- Getting a count of the number of object (reference)s in the container
- Answering a true/false question as to whether a particular object is referenced by the container or not

> *Throughout this chapter, we'll talk casually about manipulating* **objects** *in collections, but please remember that, with C#, what we really mean is that we're manipulating object* **references.**

Arrays As Simple Collections

One simple type of collection that you may already be familiar with from your work with other programming languages is the **array.** We can think of an array as a series of compartments, with each compartment sized appropriately for whatever data type the array as a whole is intended to hold. Arrays typically hold

items of like type: for example, int(eger)s, or char(acter)s, or, in an OO language, object references: Student objects, or Course objects, or Professor objects, etc.

Declaring and Instantiating Arrays

In C#, arrays are objects (as are all C# collections). The Array class of the System namespace is the basis for all C# arrays. The official C# syntax for declaring that a variable x will serve as a reference to an array containing items of a particular data type is as follows:

```
datatype[] x;
```

For example:

```
int[] x;
```

which is to be read "int(eger) array x" (or, alternatively, "x is an array of ints").

Because C# arrays are objects, they must be instantiated using the new operator; we also specify how many items the array is capable of holding, i.e., its size in terms of its number of compartments, when we first instantiate the array. Here is a code snippet that illustrates the somewhat unusual syntax for constructing an array; in this particular example, we're constructing an array designed to hold Student object references, depicted in Figure 6-3:

```
// Here, we are instantiating an array object that will be used to store 20
// Student object references, and are maintaining a handle on the array object
// via reference variable x.
Student[] x = new Student[20];
```

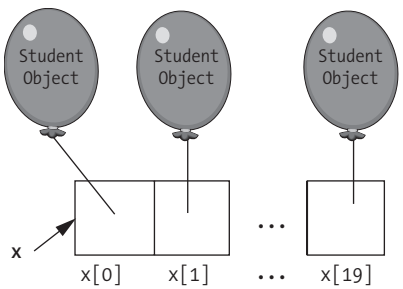


Figure 6-3. Array x *is designed to hold up to 20* Student *references.*

This use of the `new` operator is unusual, in that we don't see a typical constructor call (with optional arguments being passed in via parentheses) following the `new` keyword, the way we do when we're constructing other types of objects. Despite its unconventional appearance, however, this line of code is indeed instantiating a new `Array` object, just the same.

Accessing Individual Array Elements

Individual array elements are accessed by appending square brackets, enclosing the index of the element to be accessed, to the end of the array name. This syntax is known as an **element access expression**. Note that when we refer to individual items in an array based on their position, or **index,** relative to the beginning of the array, we start counting at 0. (As it turns out, the vast majority of collection types in C# as well as in other languages are **zero-based**.) So, the items stored in array `Student[] x` in our previous example would be individually referenced as `x[0]`, `x[1]`, . . . , `x[19]`.

Consider the following code snippet:

```
int[] data = new int[3];
data[0] = 4; // setting an element's value
int temp = data[1]; // getting an element's value
```

In the first line of code, we're declaring and instantiating an `int`(eger) array of size 3. In the second line of code, we're assigning the `int` value `4` to the "zeroeth" (***first***) element of the array. In the last line of code, we're obtaining the value of the ***second*** element of the array (element number 1) and assigning it to an `int` variable named `temp`.

In the next snippet, we're populating an array named `squareRoot` of type `double` to serve as a look-up table of square root values, where the value stored in cell `squareRoot[i]` represents the square root of `i`. (We declare the array to be one element larger than we need it to be so that we may skip over the zeroeth cell; for ease of look-up, we want the square root of 1 to be contained in cell 1 of the array, not in cell 0.)

```
double[] squareRoot = new double[11];  // we'll effectively ignore cell 0

// Note that we're skipping cell 0.
for (int i = 1; i <= 10; i++) {
  squareRoot[i] = Math.Sqrt(i);
}

Console.WriteLine("The square root of 5 is " + squareRoot[5]);
```

The `Math.Sqrt()` *method computes the square root of a* `double` *argument passed to the method; we're passing in an* `int` *in the preceding example, which automatically gets cast to a double. We'll revisit the* `Math` *class again in Chapter 7.*

Initializing Array Contents

Values can be assigned to individual elements of an array using indexes as shown earlier, or we can initialize an array with a complete set of values when the array is first instantiated. In the latter case, initial values are provided as a comma-separated list enclosed in braces. This syntax replaces the normal right-hand side of the array instantiation statement. For example, the following code instantiates and initializes a three-element `string` array:

```
string[] names = { "Lisa", "Jackson", "Zachary" };
```

Note that C# automatically counts the number of initial values that we're providing, and sizes the array appropriately. The preceding approach is much more concise than the equivalent alternative shown here:

```
string[] names = new string[3];
names[0] = "Lisa";
names[1] = "Jackson";
names[2] = "Zachary";
```

although the result in both cases is the same: the zeroeth (first) element of the array will reference the `string` "Lisa", the next element will reference "Jackson", and so on.

Note that it isn't possible to "bulk load" an array in this fashion ***after*** the array has been instantiated, as a separate line of code; that is, the following won't work:

```
string[] names = new string[5];
// This next line won't compile.
names = {"Steve", "Jacquie", "Chloe", "Shylow", "Baby Grode" };
```

If a set of comma-separated initial values aren't provided when an array is first instantiated, the elements of the array are automatically initialized to their zero-equivalent values. For example, `int[] data` as declared earlier would be initialized to contain 3 integer zeroes (`0`s), and `double[] squareRoot` as declared earlier

would be initialized to contain 11 floating point zeroes (`0.0`s). If we declare and instantiate an array intended to hold references to objects, as in

```
Student[] studentBody = new Student[100];
```

then we'd wind up with an `Array` object containing 100 `null` values (recall that `null`, a C# keyword, is the zero equivalent value for an object reference). If we think of an array as a simple type of collection, and we in turn think of a collection as an "egg carton," then we've just created an empty egg carton with 100 egg compartments, but no "eggs."

Manipulating Arrays of Objects

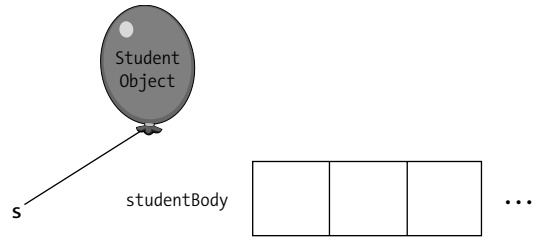
To fill our `Student` array with values other than `null`, we'd have to individually store `Student` object references in each cell of the array. For example, if we wanted to create brand-new `Student` objects to store in our array, we may write code as follows:

```
studentBody[0] = new Student();
studentBody[1] = new Student();
// etc.
```

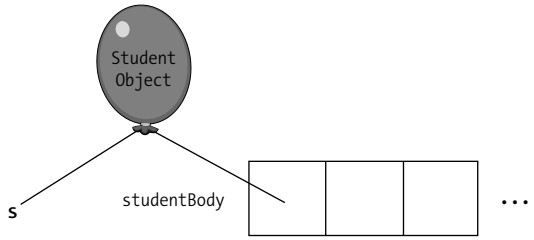
or alternatively:

```
Student s = new Student();
studentBody[0] = s;
// Reuse s!
s = new Student();
studentBody[1] = s;
```

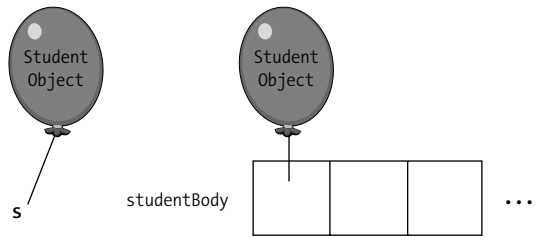
In the latter example, note that we're "recycling" the same reference variable, `s`, to create many different `Student` objects. This works because, after each instantiation, we store a handle on the newly created object in an array compartment, thus allowing `s` to let go of ***its*** handle on that same object, as depicted in Figure 6-4. This technique is used frequently, with ***all*** collection types, in virtually all OO programming languages.



A Student object is created and handed to s ...



... s hands the object's handle off to the array ...



... thus freeing up s to take hold of another new Student!

Figure 6-4. Handing new objects one by one into a collection

When we've created an array to hold objects, as we did for the `studentBody` array previously, then assuming we've populated the array with object references, an indexed reference to any populated compartment in the array represents an object reference, and can be used accordingly:

```
studentBody[0].GetName();  // We're using dot notation to call a method on
                           // studentBody[0], the first Student object
                           // reference in the array.
```

The syntax of this message may seem a bit peculiar at first, so let's study it a bit more carefully. Since `studentBody` is declared to be an array capable of holding `Student` object references, then `studentBody[`*n*`]` represents the contents of the *n*th

compartment of the array—namely, a reference to a `Student` object! So, the "dot" in the preceding message is separating an expression representing an object reference from the method call being made on that object, and is no different than any of the other dot notation messages that we've seen up until now.

By using a collection such as an array, we don't have to invent a different variable name for each `Student` object, which means we can step through them all quite easily using a `for` loop. Here is the syntax for doing so with an array:

```
// Step through all 100 compartments of the array.
for (int i = 0; i <= 99; i++) {
  // Access the "ith" element of the array -- a Student object (reference) -- so
  // that we may print each student's name in turn; in effect, we're printing
  // a student roster.
  Console.WriteLine(studentBody[i].GetName());
}
```

> *We'll examine a more sophisticated way of stepping through collections in general, using a special type of object called an* `IEnumerator`*, in Chapter 13.*

Note that we have to take care when stepping through an array to avoid "land mines" due to empty compartments. That is, if we're executing the preceding `for` loop, but the array isn't completely filled with `Student` objects, then our invocation of the `GetName` method will fail as soon as we hit the first empty/`null` compartment, because in essence we'd be trying to talk to an object that wasn't there! If we modify our code by inserting an `if` test to guard against null values, however, we'd be OK:

```
// Step through all 100 compartments of the array.
for (int i = 0; i <= 99; i++) {
  // Avoiding "land mines"!
  if (studentBody[i] != null) {
    Console.WriteLine(studentBody[i].GetName());
  }
}
```

> *We'll learn in Chapter 13 that this type of failure—namely, attempting to talk to a nonexistent object, or* ***null reference****—results in an* ***exception*** *being thrown.*

Other Array Considerations

Some other facts concerning C# arrays:

- ***Once the size of an array has been declared, its size can't be changed.*** This can become an issue in situations where we don't know what the size of an array should be at the time that we're declaring it. For this reason, arrays aren't always the best choice of collection type for a given application, as we'll discuss later in this chapter.

- ***We can't mix and match data types in an array;*** we're constrained to inserting values whose data type matches the type with which the array was first declared. For instance, we can't assign a `string` as an element of an integer array. The only exception is that if we can cast some value of type `A` into type `B`, then we can of course make such an assignment. For example, if we wish to assign the value of a `double` variable to an element of a `float` array, we would cast the variable into a `float` and then assign it to the array element as shown here:

```
float[] a = new float[10];
double d = 10.37;
a[0] = (float) d;  // note cast
```

Multidimensional Arrays

So far we've been discussing one-dimensional arrays. It's also possible to declare and use arrays of two or more dimensions. There are two types of **multidimensional arrays—rectangular** and **jagged.**

Rectangular Arrays

A rectangular array is one in which every row has the same number of columns:

```
3      2      37      8      4
7      4       9      0      3
1     11      99     13      5
```

This represents a three-row by five-column two-dimensional rectangular integer array.

The syntax for declaring and instantiating a two-dimensional rectangular array is as follows:

```
ArrayType[,] arrayName = new ArrayType[numRows, numCols];
```

For example:

```
double[,] values = new double[3, 5];  // three rows of five columns each
```

We place one comma between the brackets on the left-hand side to signal that the array will have two dimensions, and must then specify the size of each of the two dimensions, separated by a comma, on the right-hand side. For a three-dimensional rectangular array, we would use two commas between the brackets on the left-hand side, and would specify the sizes of three dimensions on the right-hand side:

```
ArrayType[,,] arrayName = new ArrayType[dim1, dim2, dim3];
```

and so forth.

To access the elements of a multidimensional rectangular array, we use indexes, but now must specify an index for each dimension of the element to be accessed. For example, consider the following code snippet, in which we instantiate a two-dimensional array of type `double`:

```
double[,] data = new double[2, 3];
data[0, 1] = 23.4;  // insert value into the FIRST row, SECOND column
// details omitted ...
double temp = data[1, 2]; // retrieve value from the SECOND row, THIRD column
```

The array has two rows and three columns. In the second line of code, the element in the ***first*** row and ***second*** column (`[0, 1]`—remember that we start counting from 0) of the array is assigned the value 23.4. In the third line, we're accessing the value of the element in the second row and third column of the array, and assigning that value to a `double` variable named `temp`.

The elements of a multidimensional rectangular array can be initialized at the time that the array is declared by placing the initial values in braces `{ ... }`, with one set of braces for each dimension of the array. For example, the following syntax would create and initialize a two-row, three-column array of integer values:

```
int[,] data = { {7, 22, 3},
                {48, 5, 10} };
```

Jagged Arrays

A jagged array is one where each row can have a different number of entries:

```
 14    3    85    2
100   11
  3   24   106
```

This represents a three-row, two-dimensional jagged integer array.

The syntax for declaring a jagged multidimensional array is different from that of a rectangular array.

- A separate set of empty braces is provided in the declaration for every dimension in the array: for a two-dimensional jagged array, we use two sets of braces; for a three-dimensional jagged array, we use three; etc. For example, the following syntax would declare a two-dimensional jagged `string` array:

```
string[][] names;  // note two sets of empty braces
```

- In the array instantiation statement, we only specify the size of the first dimension of such an array. For example, the following syntax would instantiate a two-dimensional jagged `string` array with three rows, allowing for a variable number of columns per row:

```
string[][] names = new string[3][];
```

In effect, when we create a two-dimensional jagged array, we have in essence created an array of one-dimensional arrays of varying sizes, as illustrated in Figure 6-5.

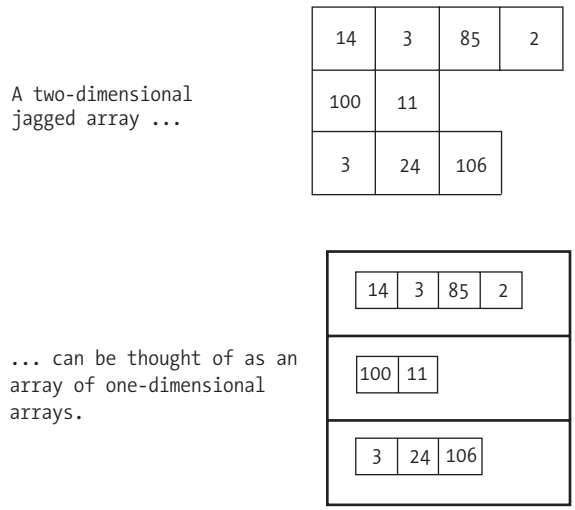


Figure 6-5. A jagged two-dimensional array as an array of one-dimensional arrays

The next step in the process is to initialize the length of each row in the array; for example:

```
names[0] = new string[4];  // first row has 4 columns (numbered 0 ... 3)
names[1] = new string[2];  // second row has 2 columns (numbered 0 ... 1)
names[2] = new string[3];  // third row has 3 columns (numbered 0 ... 2)
```

To access the elements of a multidimensional jagged array, we use indexes with a separate set of brackets to specify each dimension. For example, to assign a value to the element in the second row and second column of the `names` array, we'd use the following syntax:

```
names[1][1] = "Mel";
```

It's also possible to initialize the elements of a multidimensional jagged array when the array is first declared, but the syntax is a bit complicated. The elements of every one-dimensional array within the multidimensional array can be initialized by placing the initial values inside braces when the one-dimensional array is declared. The `new` keyword must be used, and the array type must also be specified. For example, the following code creates and initializes a two-dimensional jagged integer array that has three columns in its first row and four columns in the second row:

```
int[][] data = new int[2][];
data[0] = new int[] { 17, 3, 24 };
data[1] = new int[] { 6, 37, 108, 99 };
```

More Sophisticated Collection Types

In an OO language, there are typically many different types of collections available to us as programmers, arrays being arguably the most primitive. There are several problems with using an array to hold a collection of objects:

- It's often hard for us to predict in advance the number of objects that a collection will need to hold—e.g., how many students are going to enroll this semester? However, as mentioned earlier, arrays require that such a determination be made at the time they are first instantiated and, once sized, can't be expanded. So, to use an array in such a situation, we'd have to make it big enough to handle the worst-case scenario, which isn't very efficient. On the other hand, when we do know how many items we're going to need to store—say, the abbreviated names of the 12 months in a year—an array might be a fine choice.

- We also talked earlier about the "land mine" issues inherent in arrays.

Fortunately, OO languages provide a wide variety of collection types besides arrays for us to choose from, each of which has its own unique properties and

advantages. Let's talk about the general properties of three basic collection types found in most OO languages:

- Ordered lists
- Sets
- Dictionaries

Then, in Chapter 13, we'll illustrate some specific C# implementations of these collection types.

Ordered Lists

An ordered list is similar to an array, in that items can be placed in the collection in a particular order and later retrieved in that same order. Specific objects can also be retrieved based on their position in the list; e.g., retrieve the second item. One advantage of an ordered list over an array, however, is that its size doesn't have to be specified at the time that the collection object is first created; an ordered list will automatically grow in size as new items are added (see Figure 6-6). (In fact, virtually all collections besides arrays have this advantage!)

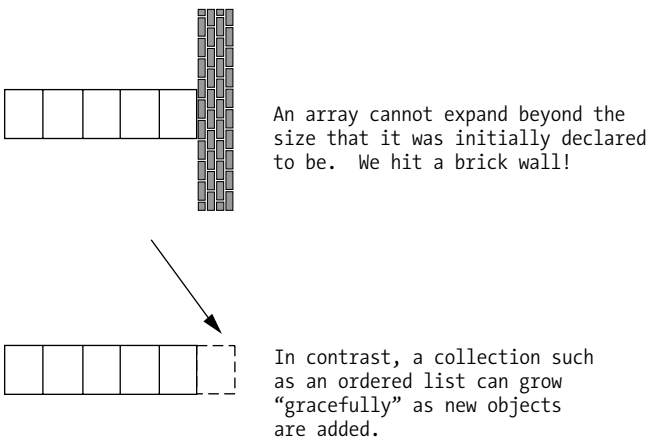


Figure 6-6. Collections other than arrays grow gracefully as needed.

By default, items are added at the end of an ordered list unless explicit instructions are given to insert a new item somewhere in the middle.

When an item is removed from an ordered list, the "hole" that would have been left behind is automatically closed up as shown in Figure 6-7. This is actually true of most collection types other than arrays, and so we don't generally speaking encounter the "land mine" problem with nonarray collections.

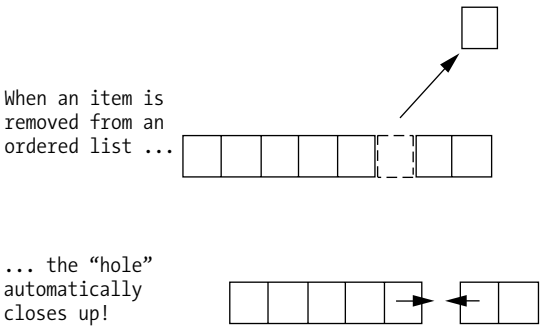


Figure 6-7. Collections other than arrays automatically shrink as items are removed.

An example of where we might use an ordered list in building our Student Registration System would be to manage a wait list for a course that had become full. Because the order with which `Student` objects are added to the list is preserved, we can be fair about selecting students from the wait list in first-come, first-served fashion should seats later become available in the course.

The C# `ArrayList` class is a specific example of an ordered list implementation.

Sorted Ordered Lists

A **sorted ordered list** is a special ,type of ordered list: when we add an object to a sorted ordered list, the list automatically inserts the object at the appropriate location in the list to maintain sorted order, instead of automatically adding the new object at the end of the list as with a generic ordered list.

With a sorted ordered list, we have to define the basis upon which the objects will be sorted, i.e., we must define a **sort key.** For example, we may wish to maintain a list of `Course` objects sorted by the value of each `Course`'s `courseNo` attribute for purposes of displaying the SRS course catalog.

Note that we could accomplish the same goal using a generic ordered list, but then the burden of keeping things sorted properly is on us, the programmers, instead of on the collection object! That is, we'd have to step through the (unsorted) list, comparing a newly added item's value to the value of each object already in the list until we found the correct insertion point, in order to preserve sorted order.

The C# `SortedList` class is a specific example of a sorted ordered list implementation.

Sets

A set is an ***unordered*** collection, which means that there is no way to ask for a particular item by number once it has been inserted. Using a set is like throwing an assortment of differently colored marbles into a bag: we can reach into the bag to pull the marbles out one by one, but there is no rhyme or reason as to the order with which we pull them out. Similarly, with a set, we can step through the entire collection of objects one-by-one to perform some operation on them; we just can't guarantee in what order the objects will be processed. We can also perform tests to determine whether a given specific object has been previously added to a set or not, just as we can answer the question "Is the blue marble in the bag?" (See Figure 6-8.)

Figure 6-8. A set is an unordered collection.

Note that duplicate object references aren't allowed in a set. If we were to create a set of `Student` object references, and a particular `Student` object reference had already been placed in that set, then the same `Student` object reference couldn't be added to the set a second time; the set would reject it. This isn't true of collections in general, however: if we wanted to, we could add a reference to the same `Student` object to an ordered list, for example, multiple times (see Figure 6-9).

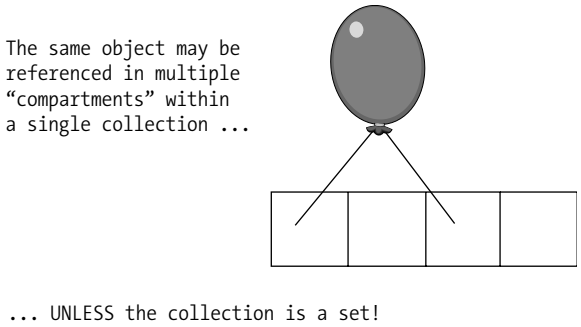


Figure 6-9. Collections other than sets accommodate mutiple references to the same object.

An example of where we might use sets in building our Student Registration System would be to group students according to the academic departments that they are majoring in. Then, if a particular course—say, Biology 216—requires that a student be a Biology major in order to register, it would be a trivial matter to determine if a particular student is a member of the Biology Department set or not.

Dictionaries

A dictionary provides a means for storing each object reference along with a unique look-up key that can later be used to retrieve the object (see Figure 6-10). The key is typically contrived based on one or more of the object's attribute values; for example, in our SRS, a `Student` object's ID number would make an excellent key, because it's inherently unique for each student. Items in a dictionary can then be quickly retrieved based on this key. Items can typically also be retrieved one by one from a dictionary type collection in ascending key order.

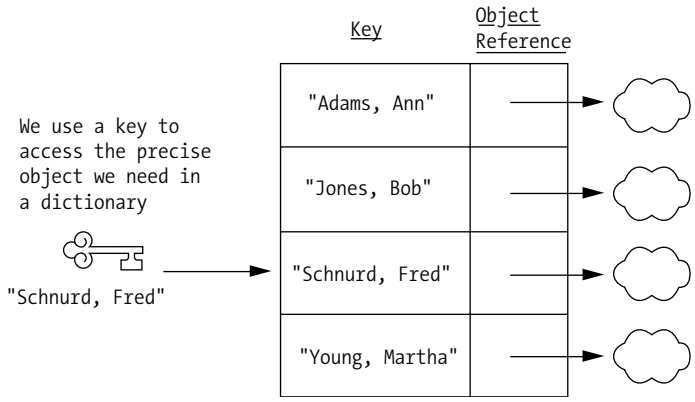


Figure 6-10. Dictionary collections accommodate direct access by key.

The SRS might use a dictionary, indexed on a unique combination of course number plus section number, to manage its course catalog. With so many courses to keep track of, being able to "pluck" the appropriate `Course` object from a collection directly (instead of having to step through an ordered list one-by-one to find it) adds greatly to the efficiency of the application.

The C# `Hashtable` *class is an example of a specific implementation of a dictionary.*

Referencing the Same Object Simultaneously from Multiple Collections

As we mentioned earlier, when we talk about inserting an object into a collection, what we really mean is that we're inserting a reference to the object, not the object itself. This implies that the same object can be referenced by multiple collections simultaneously. Think of a person as an object, and his or her telephone number as a handle for reaching that person. Now, as we proposed earlier in this chapter, think of an address book as a collection: it's easy to see that the same person's phone number (reference) can be recorded in many different address books (collections) simultaneously.

Now, for an example relevant to the SRS: given the students who are registered to attend a particular course, we may simultaneously maintain the following:

- An ordered list of these students for purposes of knowing who registered first for a follow-on course
- A dictionary that allows us to retrieve a given `Student` object based on his or her name
- Perhaps even a second SRS-wide dictionary that organizes ***all*** students at the university based on their student ID numbers

This is depicted conceptually in Figure 6-11.

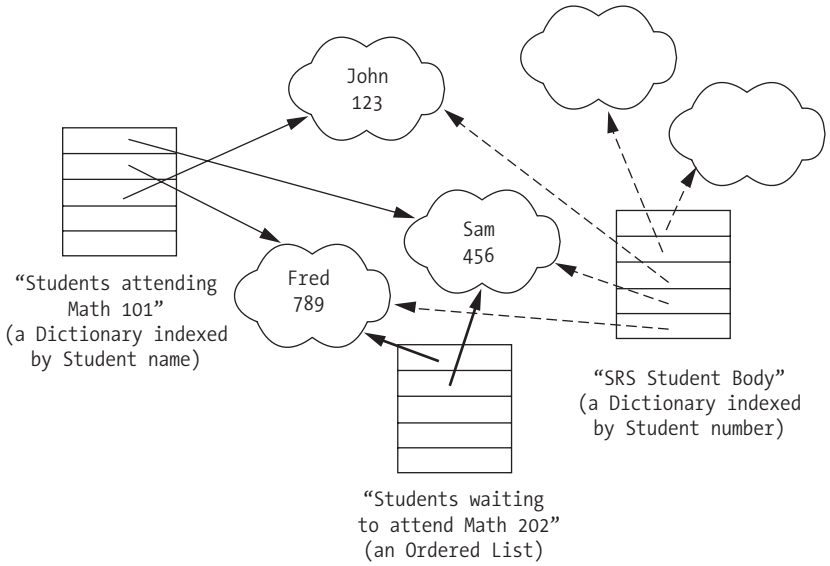


Figure 6-11. A given object may be referenced by multiple collections simultaneously.

Inventing Our Own Collection Types

As mentioned earlier, different types of collections have different properties and behaviors. You must therefore familiarize yourself with the various built-in collection types available for your OO language of choice, and choose the one that is the most appropriate for what you need in a given situation. Or, if none of them suit you, invent your own! This is where we start to get a real sense of the power of an OO language: since we have the ability to invent our own abstract data types, we have free rein to define our own collection types, because these, after all, are merely classes.

There are several ways to create one's own collection type:

- ***Approach #1:*** We can design a brand-new collection type from scratch.
- ***Approach #2:*** We can use the techniques that we learned in Chapter 5 to extend a predefined collection class.
- ***Approach #3:*** We can create a "wrapper" class that encapsulates one of the built-in collection types, to hide some of the details involved with manipulating the collection.

Let's discuss each of these approaches in turn.

Approach #1: Create a Brand-New Collection Type from Scratch

Creating a brand new collection class from scratch is typically quite a bit of work, and since most OO languages provide such a wide range of predefined collection types, it's almost always possible to find a preexisting collection type to use as a starting point, in which case one of the following two approaches would be preferred.

Approach #2: Extend a Predefined Collection Class

In the following example, we extend the built-in `ArrayList` class to create a collection class called `MyStringCollection`:

```
using System.Collections;

public class MyStringCollection : ArrayList
{
  // We inherit all of the attributes and methods of a standard ArrayList
  // "as-is," then define a few attributes and methods of our own.
  private string longestStringAddedSoFar;

  // Define a completely new method.
  public void AddAString(string s) {
    // Add the string to the collection using the Add() method that we've
    // inherited from ArrayList.
    Add(s);

    // Pseudocode.
    compare length of s to the length of the longest string inserted so far,
    as recorded by attribute longestStringAddedSoFar;

    // Pseudocode.
    if (s is longer than longestStringAddedSoFar) {
      // Remember this fact!
      longestStringAddedSoFar = s;
    }
  }
}
```

This approach works well if we simply intend to add a few completely ***new*** features (attributes, methods), inheriting the methods and attributes of the parent class "as is." If we plan on ***overriding*** any of the inherited methods, however,

we must remember to invoke the parent's version of the code first, via the `base` keyword, as we learned to do in Chapter 5:

```
using System.Collections;

public class MyStringCollection : ArrayList
{
  private string longestStringAddedSoFar;

   // In this case, we choose to override the Add() method of ArrayList.
   public override void Add(Object s) {
     // Add the string to the collection using our parent's version of the
     // Add() method.
     base.Add(s);

     // Pseudocode.
     compare length of s to the length of the longest string inserted so far
     as recorded by attribute longestStringAddedSoFar;

     // Pseudocode.
     if (s is longer than longestStringAddedSoFar) {
       // Remember this fact!
       longestStringAddedSoFar = s;
     }
   }
 }
```

By invoking `base.Add()` as the first step in our overridden `Add()` method, we're ensuring that we're doing everything that our parent `ArrayList` class does when adding an item to its internal collection, ***without having to know the details of what is happening behind the scenes,*** before we go on to do something extra: namely, to track the longest such `string` item in this particular case.

Approach #3: Create a "Wrapper" Class to Encapsulate a Predefined Collection Type

By creating such a wrapper class, we are able to hide some of the details involved with manipulating the collection. This is a nice compromise position, and we'll illustrate how one goes about doing this with a specific example.

Let's say we wanted to invent a new type of collection called an `EnrollmentCollection`, to be used by a `Course` object to manage all of its enrolled `Student` objects. We could take advantage of information hiding and encapsulation to "hide" a standard collection object—say, a C# `ArrayList`, which as mentioned

earlier is C#'s implementation of an ordered list collection—inside of our `EnrollmentCollection` class, as an attribute. We'd then provide

- `Enroll()` and `Drop()` methods for adding or removing a `Student` from our `EnrollmentCollection`
- An `IsEnrolled()` method, which will help us to determine if a particular `Student` object is already in the collection (we don't want the same `Student` to enroll twice in the same course)
- A `GetTotalEnrollment()` method to determine how many students are enrolled at any given time

These methods' logic can be as sophisticated as we wish for it to be—the method bodies are ours to control! The following example uses heavy doses of pseudocode to give you a sense of what we might actually program these methods to do for us; we'll see plenty of real C# collection manipulation (`ArrayLists` in particular) in the SRS code examples in Part Three of the book.

```
public class EnrollmentCollection
{
  // "Hide" a standard C# collection object inside as a private attribute.
  // We'll be storing Student objects in this collection.
  private ArrayList students;

  // Constructor.
  public EnrollmentCollection() {
    // Instantiate the encapsulated ArrayList instance.
    students = new ArrayList();
  }

  // Methods to add a student ...
  public bool Enroll(Student s) {
    // First, make sure that there is room in the class (pseudocode).
    if (adding this student will exceed course capacity) {
      return false;
    }

    // Next, make sure that this student isn't already enrolled
    // in this class (pseudocode).
    if (student is already enrolled) {
      return false;
    }
```

```
    // Verify that the student in question has met
    // all necessary prerequisites (pseudocode).
    if (some prerequisite not satisfied) {
      return false;
    }

    // If we made it to here, all is well!
    // Add the student to the ArrayList by calling its Add() method.
    // (This is an example of delegation, a concept that we discussed
    // in Chapter 4.)
    students.Add(s);
    return true;
  }

  // ... and to remove a student.
  public bool Drop(Student s) {
    // First make sure that the student in question
    // is actually enrolled (pseudocode).
    if (student is not enrolled) {
      return false;
    }

    // Remove the student from the ArrayList by calling the Remove()
    // method.  (Another example of delegation.)
    students.Remove(s);
    return true;
  }

  public int GetTotalEnrollment() {
    // Access and return the size of the ArrayList.  (Delegation yet again!)
    return students.Count;
  }

  public bool IsEnrolled(Student s) {
    // More delegation!
    if (students.Contains(s)) {
      return true;
    }
    else {
      return false;
    }
  }
}
```

By taking advantage of information hiding to "wrap" a standard collection type inside of one that we've invented, we've allowed for the flexibility to change the internal details of this implementation without disrupting the client code that takes advantage of this collection type. Down the road, we may wish to switch from using an `ArrayList` to a different predefined collection type, and because we've declared our `students` collection as a `private` attribute, we're free to do so, as long as we don't change the headers of our existing public methods.

Now, how do we use this collection class that we've invented? Let's show it in action in the `Course` class.

```
public class Course
{
  // We declare an attribute to be a collection of type
  // EnrollmentCollection, and will use it to manage
  // all of the students who register for this course.
  private EnrollmentCollection enrolledStudents;

  // Other simple attributes.
  string courseName;
  int credits;
  // etc.

  // Parameterless constructor.
  public Course() {
    enrolledStudents = new EnrollmentCollection();
    // Other details omitted.
  }

  // Other constructors' details omitted.

  public bool Enroll(Student s) {
    // All we have to do is to pass the Student reference
    // in to the collection's enroll method; the collection
    // does all of the hard work! This is another example of
    // delegation.
    enrolledStudents.Enroll(s);
  }

  public bool Drop(Student s) {
    // Ditto!
    enrolledStudents.Drop(s);
  }

  // etc.
}
```

Collections As Method Return Types

Collections provide a way to overcome the limitation that we noted in Chapter 4 about methods only being able to return a single result. If we define a method as having a return type that is a collection type, we can hand back an arbitrary sized collection of object references to the client code that invokes the method.

In the code snippet shown next for the `Course` class, we provide a `GetRegisteredStudents` method to enable client code to request a "handle" on the entire collection of `Student` objects that are registered for a particular course:

```
public class Course
{
  private EnrollmentCollection enrolledStudents;

  // Other details omitted ...

  // The following method returns a reference to an entire collection
  // containing however many students are registered for the course in question.
  EnrollmentCollection GetRegisteredStudents() {
    return enrolledStudents;
  }
}
```

An example of how client code would then use such a method is as follows:

```
// Instantiate a course and several students.
Course c = new Course();
Student s1 = new Student();
Student s2 = new Student();
Student s3 = new Student();

// Enroll the students in the course.
c.Enroll(s1);
c.Enroll(s2);
c.Enroll(s3);

// Now, ask the course to give us a handle on the collection of
// all of its registered students ...
EnrollmentCollection ec = c.GetRegisteredStudents();

// ... and iterate through the collection, printing out a grade report for
// each Student (pseudocode).
for (each Student in EnrollmentCollection) {
  s.PrintGradeReport();
}
```

Of course, if we return a direct handle on a collection such as `enrolledStudents` *to client code, we are giving client code the ability to modify that collection (e.g., removing a* `Student` *reference). Design considerations may warrant that we create a copy of the collection before returning it, so that the original collection is not modified:*

```
public class Course
{
  private EnrollmentCollection enrolledStudents;

  // Other details omitted ...

  // The following method returns a COPY of the Student's enrolledStudents
  // collection, so that client code can't modify the OFFICIAL version.
  EnrollmentCollection GetRegisteredStudents() {
    EnrollmentCollection temp = new EnrollmentCollection();

    // Pseudocode.
    copy contents of enrolledStudents to temp

    return temp;
  }
}
```

Another way to avoid the problem of allowing client code to modify a collection in the previous example would be to have the `GetRegisteredStudents` *method return an* ***enumeration*** *of the elements contained in the collection. We'll discuss how to use enumerators in Chapter 13.*

Collections of Supertypes

We said earlier that arrays, as simple collections, contain items that are all of the same type: all `int`(egers), for example, or all (references to) `Student` objects. As it turns out, this is true of collections in general: we'll typically want to constrain them to contain similarly typed objects. However, the power of inheritance steps in to make collections quite versatile.

It turns out that if we declare an array to hold objects of a given type—e.g., `Person`—then we're free to insert objects explicitly declared to be of type `Person` ***or of any types derived from*** `Person`—for example, `UndergraduateStudent`, `GraduateStudent`, and `Professor`. This is due to the "is a" nature of inheritance:

`UndergraduateStudent`, `GraduateStudent`, and `Professor` objects, as subclasses of `Person`, are simply special cases of `Person` objects. The C# compiler would therefore be perfectly happy to see code as follows:

```
Person[] people = new Person[100];  // of Person object references

Professor p = new Professor();
UndergraduateStudent s1 = new UndergraduateStudent();
GraduateStudent s2 = new GraduateStudent();

// Add a mixture of professors and students in random order to the array;
// as long as Professor, UndergraduateStudent, and GraduateStudent are
// all derived from Person, the compiler will be happy!
people[0] = s1;
people[1] = p;
people[2] = s2;
// etc.
```

As we'll see when we discuss C# collection types in more detail in Chapter 13, C# collections other than `Array`*s actually don't allow us to specify what type of object they will hold when we declare a collection, as illustrated here:*

```
ArrayList list = new ArrayList();  // No type designated!
                                   // ArrayLists hold generic Objects.
```

versus:

```
Student[] s = new Student[100];    // With Arrays, we DO specify a
                                   // type (Student, in this case).
```

Most C# collections are automatically designed to hold objects of type `Object`*, which as we learned in an earlier chapter is the superclass of all other classes in the C# language, user defined or otherwise. So, from the compiler's perspective we can put whatever types of object we wish into a collection in any combination. But, it's still important that you, as the programmer, know what the intended base type for a collection is going to be from an application design standpoint, so that you discipline yourself to only insert objects of the proper type (including derived types of that type) into the collection. This will be important when we subsequently attempt to iterate through and process all of the objects in the collection: we'll need to know what general class of object they are, so that we'll know what methods they can be called upon to perform. We'll talk about this in more detail when we discuss* polymorphism *in Chapter 7.*

Composite Classes, Revisited

You may recall that when we talked about the attributes of the `Student` class back in Chapter 3, we held off on assigning types to a few of the attributes, as shown in Table 6-1.

Table 6-1. Proposed Data Structure for the `Student` *Class*

Attribute Name	Data Type
name	string
studentID	string
birthdate	System.DateTime
address	string
major	string
gpa	double
advisor	Professor
courseLoad	**???**
transcript	**???**

Armed with what we now know about collections, we can go back and assign types to attributes `courseLoad` and `transcript`.

courseLoad

The `courseLoad` attribute is meant to represent a list of all `Course` objects that the `Student` is presently enrolled in. So, it makes perfect sense that this attribute be declared to be simply a standard collection of `Course` objects!

```
public class Student
{
  private string name;
  private string studentId;
  private CollectionType courseLoad; // of Course objects
  // etc.
```

transcript

The `transcript` attribute is a bit more challenging. What is a transcript, in real-world terms? It's a report of all of the courses that a student has taken since he or she was first admitted to this school, along with the semester in which each course was taken, the number of credit hours that each course was worth, and the letter grade that the student received for the course. If we think of each entry in this list as an object, we can define them via a `TranscriptEntry` class, representing an abstraction of a single line item on the transcript report, as follows:

```
public class TranscriptEntry
{
  // One TranscriptEntry represents a single line item on a transcript report.
  private Course courseTaken;
  private string semesterTaken;  // e.g., "Spring 2000"
  private string gradeReceived;  // e.g., "B+"

  // Other details omitted ...

  // Note how we "talk to" the courseTaken object via its methods
  // to retrieve some of this information (delegation once again!).
  public void PrintTranscriptEntry() {
    // Reminder:  \t is a tab character.
    Console.WriteLine(courseTaken.CourseNo + "\t" +
        courseTaken.Title + "\t" +
        courseTaken.CreditHours + "\t" +
        gradeReceived);
  }

  // etc.
}
```

Note that we're declaring one of the attributes of `TranscriptEntry` to be of type `Course`, which means that each `TranscriptEntry` object will maintain a handle on its corresponding `Course` object. By doing this, the `TranscriptEntry` object can avail itself of the `Course` object's title, course number, or credit hour value (needed for computing the GPA)—all privately encapsulated in the `Course` object as attributes—by accessing the appropriate properties on that `Course` object as needed.

Back in the `Student` class, we can now define the `Student`'s `transcript` attribute to be a collection of `TranscriptEntry` objects. We can then add a `PrintTranscript` method to the `Student` class, the code for which is highlighted in the following snippet:

```
public class Student
{
  private string name;
  private string studentId;
  // Pseudocode.
  private CollectionType transcript; // of TranscriptEntry objects
  // etc.

  // Details omitted ...

  public void PrintTranscript() {
    // Pseudocode.
    for (each TranscriptEntry t in transcript) {
      t.PrintTranscriptEntry();
    }
  }
}
```

transcript, Take 2

Alternatively, we could use the technique of creating a wrapper class called `Transcript` to encapsulate some standard collection type, as we did with `EnrollmentCollection` in an earlier example:

```
public class Transcript
{
    private ArrayList transcriptEntries; // of TranscriptEntry objects
    // other attributes omitted from this example

    // Pseudocode.
    public void AddTranscriptEntry(arglist) {
      insert new entry into the transcriptEntries ArrayList -- details omitted
    }

    // We've transferred the logic of the Student class's PrintTranscript
    // method into THIS class instead.
    public void PrintTranscript() {
      // Pseudocode.
      for (each TranscriptEntry t in transcriptEntries) {
        t.PrintTranscriptEntry();
      }
    }

    // etc.
  }
```

We then can go back to the `Student` class and change our declaration of the `transcript` attribute from being a standard collection type to being of type `Transcript`:

```
public class Student
{
  private string name;
  private string studentId;
  // etc.
  private Transcript transcript;  // an ENCAPSULATED collection of
                                  // TranscriptEntry objects
  // etc.
```

We can then turn around and simplify the `PrintTranscript` method of the `Student` class accordingly:

```
public class Student
{
  // Details omitted.

  public void PrintTranscript(string filename) {
    // We now delegate the work to the Transcript attribute!
    transcript.Print();
  }

  // etc.
```

This "take 2" approach of introducing ***two*** new classes/abstractions—`TranscriptEntry` and `Transcript`—is a bit more sophisticated than the first approach, where we only introduced `TranscriptEntry` as an abstraction. Also, this second approach is "truer" to the object paradigm, because the `Student` class needn't be complicated by the details of how `Transcripts` are represented or managed internally—those details are hidden inside of the `Transcript` class, as they should be.

Our Completed Student Data Structure

Table 6-2 illustrates how we've taken full advantage of collections to round out our `Student` class definition.

Table 6-2. Rounding Out the `Student` *Class's Data Structure with Collections*

Attribute Name	Data Type
`name`	`string`
`studentID`	`string`
`birthdate`	`System.DateTime`
`address`	`string`
`major`	`string`
`gpa`	`double`
`advisor`	`Professor`
`courseLoad`	Standard type collection of `Course` objects
`transcript`	Standard type collection of `TranscriptEntry` objects or (preferred) `Transcript`

Summary

In this chapter, you've learned

- That collections are special types of objects used to gather up and manage references to other objects
- That arrays, as simple collections, have some limitations, but that we have other more powerful collection types to draw upon with OO languages, such as
  - Ordered lists
  - Sets
  - Dictionaries
- That it's important to familiarize ourselves with the unique characteristics of whatever collection types are available for a particular OO language so as to make the most intelligent selection of which collection type to use for a particular circumstance
- That we can invent our own collection types by creating "wrapper classes" around any predefined collection classes

- How we can work around the limitation that a method can only return one result by having that result be a collection of objects
- How we can create very sophisticated composite classes through the use of collections as attributes

Exercises

1. Given the following abstraction:

   A book is a collection of chapters, which are each collections of pages.

   sketch out the code for the `Book`, `Chapter`, and `Page` classes.

   - Invent whatever attributes you think would be relevant, taking advantage of collections as attributes where appropriate.
   - Include methods on the `Chapter` class for adding pages, and for determining how many pages a chapter contains.
   - Include methods on the `Book` class for adding chapters, for determining how many chapters the book contains, for determining how many pages the book contains (hint: use delegation!), and for printing out a book's table of contents.

2. What generic type(s) of collection(s)—ordered list, sorted ordered list, set, dictionary—might you use to represent each of the following abstractions? Explain your choices.

   - A computer parts catalog
   - A poker hand
   - Trouble calls logged by a technical help desk

3. What collections do you think it would be important to maintain for the SRS, based on the requirements presented in the Introduction to this book?

4. What collections do you think it would be important to maintain for the Prescription Tracking System (PTS) described in Appendix B?

5. What collections do you think it would be important to maintain for the problem area that you described for exercise 3 in Chapter 2?