CHAPTER 6

# Arrays


**6.1 INTRODUCTION TO ARRAYS** 306
Creating and Accessing Arrays 307
The `length` Instance Variable 311
Tip: Use `for` Loops with Arrays 311
Pitfall: Array Indices Always Start with Zero 312
Pitfall: Array Index Out of Bounds 313
Initializing Arrays 313
Pitfall: An Array of Characters Is Not a String 315

**6.2 ARRAYS AND REFERENCES** 316
Arrays Are Objects 316
Pitfall: Arrays with a Class Base Type 318
Arrays Parameters 319
Pitfall: Use of = and == with Arrays 323
Arguments for the Method `main` ✜ 326
Methods that Return an Array 327

**6.3 PROGRAMMING WITH ARRAYS** 329
Partially Filled Arrays 329
Example: A Class for Partially Filled Arrays 333
Tip: Accessor Methods Need Not Simply Return Instance Variables 339
Privacy Leaks with Array Instance Variables 340
A Preview of Vectors 342
Example: Sorting an Array 342

**6.4 MULTIDIMENSIONAL ARRAYS** 346
Multidimensional Array Basics 346
Using the `length` Instance Variable 350
Ragged Arrays ✜ 351
Multidimensional Array Parameters and Returned Values 352
Example: A Grade Book Class 353

**CHAPTER SUMMARY** 359
**ANSWERS TO SELF-TEST EXERCISES** 360
**PROGRAMMING PROJECTS** 367

# 6 Arrays

*Memory is necessary for all the operations of reason.*

Blaise Pascal, *Pensées*

### INTRODUCTION

An *array* is a data structure used to process a collection of data that is all of the same type, such as a list of numbers of type `double` or a list of strings. In this chapter we introduce you to the basics of defining and using arrays in Java.

### PREREQUISITES

Section 6.1 requires only Chapters 1 through 3 and Section 4.1 of Chapter 4. Indeed, much less than all of Section 4.1 is needed. All you really need from Section 4.1 is to have some idea of what an object is and what an instance variable is.

The remaining sections require Chapters 1 through 5 with the exception that Section 5.4 on packages and `javadoc` is not required.

## 6.1 Introduction to Arrays

*It is a capital mistake to theorize before one has data.*

Sir Arthur Conan Doyle,
*Scandal in Bohemia* (Sherlock Holmes)

Suppose we wish to write a program that reads in five test scores and performs some manipulations on these scores. For instance, the program might compute the highest test score and then output the amount by which each score falls short of the highest. The highest score is not known until all five scores are read in. Hence, all five scores must be retained in storage so that after the highest score is computed each score can be compared to it. To retain the five scores, we will need something equivalent to five variables of type `int`. We could use five individual variables of type `int`, but keeping track of five variables is hard, and we may later want to change our program to handle 100 scores; certainly, keeping track of 100 variables is impractical. An array is the perfect solution. An **array** behaves like a list of variables with a uniform naming mechanism that can be declared in a single line of simple code. For example, the names for the five individual variables we need might be `score[0]`,

`score[1]`, `score[2]`, `score[3]`, and `score[4]`. The part that does not change, in this case `score`, is the name of the array. The part that can change is the integer in the square brackets `[]`.

## CREATING AND ACCESSING ARRAYS

In Java, an array is a special kind of object, but it is often more useful to think of an array as a collection of variables all of the same type. For example, an array that behaves like a collection of five variables of type `double` can be created as follows:

```
double[] score = new double[5];
```

This is like declaring the following to be five variables of type `double`:

```
score[0], score[1], score[2], score[3], score[4]
```

These individual variables that together make up the array are referred to in a variety of different ways. We will call them **indexed variables**, though they are also sometimes called **subscripted variables** or **elements** of the array. The number in square brackets is called an **index** or a **subscript**. In Java, *indices are numbered starting with* `0`*, not starting with* `1` *or any number other than* `0`. The number of indexed variables in an array is called the **length** or **size** of the array. When an array is created, the length of the array is given in square brackets after the array base type. The indexed variables are then numbered (also using square brackets) starting with `0` and ending with the integer that is *one less than the length of the array.*

indexed variable
subscripted variable

index, subscript

size, length

The following

```
double[] score = new double[5];
```

is really shorthand for the following two statements:

```
double[] score;
score = new double[5];
```

The first statement declares the variable `score` to be of the array type `double[]`. The second statement creates an array with five indexed variables of type `double` and makes the variable `score` a name for the array. You may use any expression that evaluates to a nonnegative `int` value in place of the `5` in square brackets. In particular you can fill a variable with a value read from the keyboard and use the variable in place of the `5`. In this way the size of the array can be determined when the program is run.

An array can have indexed variables of any type, but they must all be of the same type. This type is called the **base type** of the array. In our example, the base type of the array `score` is `double`. To declare an array with base type `int`, simply use the type name `int` instead of `double` when the array is declared and created. The base type of an array can be any type. In particular it can be a class type.

base type

### Declaring and Creating an Array

You declare an array name and create an array in almost the same way that you create and name objects of classes. There is only a slight difference in the syntax.

**Syntax:**

```
Base_Type[] Array_Name = new Base_Type[Length];
```

The *Length* may be given as any expression that evaluates to a nonnegative integer. In particular *Length* can be an `int` variable.

**Examples:**

```
char[] line = new char[80];
double[] reading = new double[300];
Person[] specimen = new Person[100];
```

`Person` is a class.

Each of the five indexed variables of our example array `score` can be used just like any other variable of type `double`. For example, all of the following are allowed in Java:

```
score[3] = 32;
score[0] = score[3] + 10;
System.out.println(score[0]);
```

The five indexed variables of our sample array `score` are more than just five plain old variables of type `double`. That number in square brackets is part of the indexed variable's name. So, your program can compute the name of one of these variables. Instead of writing an integer constant in the square brackets, you can use any expression that evaluates to an integer that is at least `0` and at most `4`. So, the following is allowed:

```
System.out.println(score[index] + " is at position " + index);
```

where `index` is a variable of type `int` that has been given one of the values `0`, `1`, `2`, `3`, or `4`.

When we refer to these indexed variables grouped together into one collective item, we will call them an *array*. So, we can refer to the array named `score` (without using any square brackets).

The program in Display 6.1 shows an example of using our sample array `score` as five indexed variables, all of type `double`.

Note that the program can compute the name of an indexed variable by using a variable as the index, as in the following `for` loop:

```
for (index = 0; index < 5; index++)
    System.out.println(score[index] + " differs from max by "
                                    + (max − score[index]));
```

**Display 6.1 An Array Used in a Program *(Part 1 of 2)***

```
1   import java.io.BufferedReader;
2   import java.io.InputStreamReader;
3   import java.io.IOException;

4   public class ArrayOfScores
5   {
6       /**
7        Reads in 5 scores and shows how much each
8        score differs from the highest score.
9       */
10      public static void main(String[] args) throws IOException
11      {
12          BufferedReader keyboard =
13                     new BufferedReader(new InputStreamReader(System.in));
14          double[] score = new double[5];
15          int index;
16          double max;
17          String inputLine;

18          System.out.println("Enter 5 scores, one per line:");
19          score[0] = stringToDouble(keyboard.readLine());
20          max = score[0];
21          for (index = 1; index < 5; index++)
22          {
23              score[index] = stringToDouble(keyboard.readLine());
24              if (score[index] > max)
25                   max = score[index];
26          //max is the largest of the values score[0],..., score[index].
27          }

28          System.out.println("The highest score is " + max);
29          System.out.println("The scores are:");
30          for (index = 0; index < 5; index++)
31              System.out.println(score[index] + " differs from max by "
32                                               + (max - score[index]));
33      }
34      private static double stringToDouble(String stringObject)
35      {
36          return Double.parseDouble(stringObject.trim());
37      }
38  }
```

This form of input is covered in Chapter 2, Section 2.3.

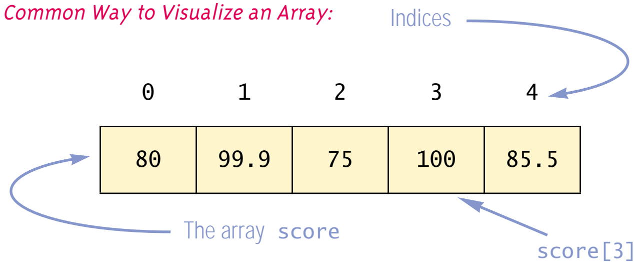
*The methods `parseDouble` and `trim` are discussed in Chapter 5 in the subsection entitled "Wrapper Classes."*

**Display 6.1 An Array Used in a Program *(Part 2 of 2)***

### Sample Dialogue

```
Enter 5 scores, one per line:
80
99.9
75
100
85.5
The highest score is 100
The scores are:
80.0 differs from max by 20
99.9 differs from max by 0.1
75.0 differs from max by 25
100.0 differs from max by 0.0
85.5 differs from max by 14.5
```

Due to imprecision in floating-point arithmetic, this value will probably only be a close approximation to `0.1`.




square brackets []

Do not confuse the three ways to use the square brackets `[]` with an array name. First, the square brackets can be used to create a type name, such as the `double[]` in the following:

```
double[] score;
```

Second, the square brackets can be used with an integer value as part of the special syntax Java uses to create a new array, as in

```
score = new double[5];
```

The third use of square brackets is to name an indexed variable of the array, such as `score[0]` or `score[3]`, as illustrated by the following line:

```
max = score[0];
```

As we mentioned previously, the integer inside the square brackets can be any expression that evaluates to a suitable integer, as illustrated by the following:

```
int next = 1;
score[next + 3] = 100;
System.out.println(
             "Score at position 4 is " + score[next + 3]);
```

Note that, in the preceding code, `score[next + 3]` and `score[4]` are the same indexed variable, because `next + 3` evaluates to `4`.

## THE length INSTANCE VARIABLE

In Java an array is considered to be an object, and, like other objects, it might have instance variables. As it turns out, an array has only one public instance variable, which is named `length`. The instance variable `length` is automatically set to the size of the array when the array is created. For example, if you create an array as follows,

```
double[] score = new double[5];
```

then `score.length` has a value of `5`.

The `length` instance variable can be used to make your program clearer by replacing an unnamed constant, such as `5`, whose meaning may not be obvious, with a meaningful name like `score.length`. In Display 6.2 we have rewritten the program in Display 6.1 using the `length` instance variable.

The `length` instance variable cannot be changed by your program (other than by creating a new array with another use of `new`).[1] For example, the following is illegal:

```
score.length = 10;//Illegal
```

**Tip**

### USE for LOOPS WITH ARRAYS

The second `for` loop in Display 6.2 illustrates a common way to step through an entire array using a `for` loop:

```
for (index = 0; index < score.length; index++)
    System.out.println(score[index] + " differs from max by "
                                    + (max − score[index]));
```

The `for` loop is ideally suited for performing array manipulations.

[1] The technical details are as follows: The instance variable `length` is created when the array is created and is declared to be `public final int`.

**Display 6.2 The length Instance Variable**

*This class also contains the method `stringToDouble`, even though it is not shown. The method `stringToDouble` is defined in Display 6.1.*

*The sample dialog is the same as in Display 6.1.*

```
1   import java.io.BufferedReader;
2   import java.io.InputStreamReader;
3   import java.io.IOException;

4   public class ArrayOfScores2
5   {
6       /**
7        Reads in 5 scores and shows how much each
8        score differs from the highest score.
9       */
10      public static void main(String[] args) throws IOException
11      {
12          BufferedReader keyboard =
13                        new BufferedReader(new InputStreamReader(System.in));
14          double[] score = new double[5];
15          int index;
16          double max;
17          String inputLine;

18          System.out.println("Enter " + score.length + " scores, one per line:");
19          score[0] = stringToDouble(keyboard.readLine());
20          max = score[0];
21          for (index = 1; index < score.length; index++)
22          {
23              score[index] = stringToDouble(keyboard.readLine());
24              if (score[index] > max)
25                  max = score[index];
26          //max is the largest of the values score[0],..., score[index].
27          }

28          System.out.println("The highest score is " + max);
29          System.out.println("The scores are:");
30          for (index = 0; index < score.length; index++)
31              System.out.println(score[index] + " differs from max by "
32                                               + (max - score[index]));
33      }
34  }
```

**Pitfall**

### Array Indices Always Start with Zero

The indices of an array always start with 0 and end with the integer that is one less than the size of the array.

**Pitfall**

### Array Index Out of Bounds

The most common programming error made when using arrays is attempting to use a nonexistent array index. For example, consider the following:

```
int[] a = new int[6];
```

When using the array a, every index expression must evaluate to one of the integers 0 through 5. For example, if your program contains the indexed variable a[i], the i must evaluate to one of the six integers 0, 1, 2, 3, 4, or 5. If i evaluates to anything else, that is an error. When an index expression evaluates to some value other than those allowed by the array declaration, the index is said to be **out of bounds**. If your program attempts to use an array index that is out of bounds, then your program will end with an error message.[2] Note that this is a run-time error message, not a compiler error message.

illegal array index

Array indices get out of bounds most commonly at the first or last iteration of a loop that processes the array. So, it pays to carefully check all array processing loops to be certain that they begin and end with legal array indices.

## INITIALIZING ARRAYS

An array can be initialized when it is declared. When initializing the array, the values for the various indexed variables are enclosed in braces and separated with commas. The expression with the braces is placed on the right-hand side of an assignment operator. For example:

```
int[] age = {2, 12, 1};
```

The array length (size) is automatically set to the number of values in the braces. So, this initializing declaration is equivalent to the following statements:

```
int[] age = new int[3];
age[0] = 2;
age[1] = 12;
age[2] = 1;
```

You can also initialize array elements using a `for` loop. For example:

```
double[] reading = new double[100];
int index;
for (index = 0; index < reading.length; index++)
    reading[index] = 42.0;
```

---

[2] Technically speaking, an `ArrayIndexOutOfBoundsException` is thrown. We will discuss exceptions in Chapter 9. Until you learn about handling exceptions, exceptions will simply appear as error conditions to you.

automatic initialization

If you do not initialize the elements of an array, they will automatically be initialized to a default value for the base type. The default values are the usual ones. For numeric types the default value is the zero of the type. (For base type `char` the default value is the nonprintable zeroth character `(char)0`, not the space character.) For the type `boolean` the default value is `false`. For class types, the default value is `null`. For example, if you do not initialize an array of `double`s, each element of the array will be initialized to `0.0`.

## Self-Test Exercises

1. In the array declaration

```
String[] word = new String[5];
```

what is

a. the array name?

b. the base type?

c. the length of the array?

d. the range of values an index accessing this array can have?

e. one of the indexed variables (or elements) of this array?

2. In the array:

```
double[] score = new double[10];
```

what is

a. the value of `score.length`?

b. the first index of `score`?

c. the last index of `score`?

3. What is the output of the following code?

```
char[] letter = {'a', 'b', 'c'};
for (int index = 0; index < letter.length; index++)
    System.out.print(letter[index] + ", ");
```

4. What is the output of the following code?

```
double[] a = {1.1, 2.2, 3.3};
System.out.println(a[0] + " " + a[1] + " " + a[2]);
a[1] = a[2];
System.out.println(a[0] + " " + a[1] + " " + a[2]);
```

5. What is wrong with the following piece of code?

```
int[] sampleArray = new int[10];
for (int index = 1; index <= sampleArray.length; index++)
    sampleArray[index] = 3*index;
```

6. Suppose we expect the elements of the array a to be ordered so that

```
a[0] ≤ a[1] ≤ a[2] ≤...
```

However, to be safe we want our program to test the array and issue a warning in case it turns out that some elements are out of order. The following code is supposed to output such a warning, but it contains a bug; what is the bug?

```
double[] a = new double[10];
    <Some code to fill the array a goes here.>
for (int index = 0; index < a.length; index++)
    if (a[index] > a[index + 1])
        System.out.println("Array elements " + index +
           " and " + (index + 1) + " are out of order.");
```

**Pitfall**

### An Array of Characters Is Not a String

An array of characters, such as the array a created below, is conceptually a list of characters and so is conceptually like a string:

```
char[] a = {'A', ' ', 'B', 'i', 'g', ' ', 'H', 'i', '!'};
```

However, an array of characters, like a, is not an object of the class `String`. In particular, the following is illegal in Java:

```
String s = a;
```

Similarly, you cannot normally use an array of characters, like a, as an argument for a parameter of type `String`.

It is, however, easy to convert an array of characters to an object of type `String`. The class `String` has a constructor that has a single parameter of type `char[]`. So, you can obtain a `String` value corresponding to an array of characters, like a, as follows:

```
String s = new String(a);
```

The object `s` will have the same sequence of characters as the array a. The object `s` is an independent copy; any changes made to a will have no effect on `s`. Note that this always uses the entire array a.

There is also a `String` constructor that allows you to specify a subrange of an array of characters *a*. For example,

```
String s2 = new String(a, 2, 3);
```

produces a `String` object with 3 characters from the array *a* starting at index 2. So, if *a* is as above, then

```
System.out.println(s2);
```

outputs

```
Big
```

Although an array of characters is not an object of the class `String`, it does have some things in common with `String` objects. For example, you can output an array of characters using `println`, as follows,

```
System.out.println(a);
```

which produces the output

```
A Big Hi!
```

provided *a* is as given above.

# 6.2 Arrays and References

*A little more than kin, and less than kind.*

William Shakespeare, *Hamlet*

Just like a variable of one of the class types you've seen, a variable of an array type holds a reference. In this section we explore the consequences of this fact, including a discussion of array parameters. We will see that arrays are objects and that array types can be considered class types, but somewhat different kinds of class types. Arrays and the kinds of classes we've seen before this chapter are *a little more than kin, and less than kind.*

## ARRAYS ARE OBJECTS

There are two ways to view an array: as a collection of indexed variables, and as a single item whose value is a collection of values of the base type. In Section 6.1 we discussed using arrays as a collection of indexed variables. We will now discuss arrays from the second point of view.

An array can be viewed as a single item whose value is a collection of values of the base type. An array variable (as opposed to an array indexed variable) names the array as a single item. For example, the following declares a variable of an array type:

```
double[] a;
```

This variable `a` can and will contain a single value. The expression

```
new double[10]
```

creates an array object and stores the object in memory. The following assignment statement places a reference to (the memory address of) this array object in the variable `a`:

```
a = new double[10];
```

Typically we combine all this into a single statement as follows:

```
double[] a = new double[10];
```

Notice that this is almost exactly the same as the way that we view objects of a class type. In Java, an array is considered an *object.* Whenever Java documentation says that something applies to objects, that means that it applies to arrays as well as objects of the class types we've seen up to now. You will eventually see examples of methods that can take arguments that may be objects of any kind. These methods will accept array objects as arguments as well as objects of an ordinary class type. Arrays are somewhat peculiar in how they relate to classes. Some authorities say array types are not classes and some authorities say they are classes. But, all authorities agree that the arrays themselves are objects. Given that arrays are objects, it seems that one should view array types as classes, and we will do so. However, although an array type `double[]` is a class, the syntax for creating an array object is a bit different. To create an array, you use the following syntax:

```
double a = new double[10];
```

You can view the expression `new double[10]` as an invocation of a constructor that uses a nonstandard syntax. (The nonstandard syntax was used to be consistent with the syntax used for arrays in older programming languages.)

As we have already seen, every array has an instance variable named `length`, which is a good example of viewing an array as an object. As with any other class type, array variables contain memory addresses, or, as they are usually called in Java, *references.* So array types are reference types.[3]

Since an array is an object, you might be tempted to think of the indexed variables of an array, such as `a[0]`, `a[1]`, and so forth, as being instance variables of the object.

---

[3] In many programming languages, such as C++, arrays are also reference types just as they are in Java. So, this detail about arrays is not peculiar to Java.

This is actually a pretty good analogy, but it is not literally true. Indexed variables are not instance variables of the array. Indexed variables are a special kind of variable peculiar to arrays. The only instance variable in an array is the `length` instance variable.

An array object is a collection of items of the base type. Viewed as such, an array is an object that can be assigned with the assignment operator and plugged in for a parameter of an array type. Since an array type is a reference type, the behaviors of arrays with respect to assignment =, ==, and parameter passing mechanisms are the same as what we have already described for classes. In the next few subsections we discuss these details about arrays.

### Arrays Are Objects

In Java, arrays are considered to be objects, and, although there is some disagreement on this point, you can safely view an array type as a class type.

### Array Types Are Reference Types

A variable of an array type holds the address of where the array object is stored in memory. This memory address is called a **reference** to the array object.

**Pitfall**

### Arrays with a Class Base Type

The base type of an array can be of any type, including a class type. For example, suppose `Date` is a class and consider the following:

```
Date[] holidayList = new Date[20];
```

This creates the 20 indexed variables `holidayList[0]`, `holidayList[1]`, ..., `holi-datList[19]`. It is important to note that this creates 20 indexed variables of type `Date`. This does not create 20 objects of type `Date`. (The index variables are automatically initialized to `null`, not to an object of the class `Date`.) Like any other variable of type `Date`, the indexed variables require an invocation of a constructor using `new` to create an object. One way to complete the initialization of the array `holidayList` is as follows:

```
Date[] holidayList = new Date[20];
for (int i = 0; i < holidayList.length; i++)
    holidayList[i] = new Date();
```

If you omit the `for` loop (and do not do something else more or less equivalent), then when you run your code, you will undoubtedly get an error message indicating a "null pointer exception." If you do not use new to create an object, an indexed variable like `holidayList[i]` is just a variable that names no object and hence cannot be used as the calling object for any method. Whenever you are using an array with a class base type and you get an error message referring to a "null pointer exception," it is likely that your indexed variables do not name any objects and you need to add something like the above `for` loop.

## ARRAY PARAMETERS

You can use both array indexed variables and entire arrays as arguments to methods, although they are different types of parameters. We first discuss array indexed variables as arguments to methods.

indexed variable arguments

An indexed variable can be an argument to a method in exactly the same way that any variable of the array base type can be an argument. For example, suppose a program contains the following declarations:

```
double n = 0;
double[] a = new double[10];
int i;
```

If `myMethod` takes one argument of type `double`, then the following is legal:

```
myMethod(n);
```

Since an indexed variable of the array `a` is also a variable of type `double`, just like `n`, the following is equally legal:

```
myMethod(a[3]);
```

There is one subtlety that does apply to indexed variables used as arguments. For example, consider the following method call:

```
myMethod(a[i]);
```

If the value of `i` is `3`, then the argument is `a[3]`. On the other hand, if the value of `i` is `0`, then this call is equivalent to the following:

```
myMethod(a[0]);
```

The indexed expression is evaluated to determine exactly which indexed variable is given as the argument.

entire array parameters

You can also define a method that has a formal parameter for an entire array so that when the method is called, the argument that is plugged in for this formal parameter is an entire array. Whenever you need to specify an array type, the type name has the

### Array Indexed Variables as Arguments

An array indexed variable can be used as an argument anyplace that a variable of the array's base type can be used. For example, suppose you have the following:

```
double[] a = new double[10];
```

Indexed variables such as `a[3]` and `a[index]` can then be used as arguments to any method that accepts a `double` as an argument.

form *Base_Type*[], so this is how you specify a parameter type for an entire array, For example, the method `doubleArrayElements`, given in what follows, will accept any array of `double` as its single argument:

```
public class SampleClass
{
    public static void doubleArrayElements(double[] a)
    {
        int i;
        for (i = 0; i < a.length; i++)
           a[i] = a[i]*2;
    }
    <The rest of the class definition goes here.>
}
```

To illustrate this, suppose you have the following in some method definition,

```
double[] a = new double[10];
double[] b = new double[30];
```

and suppose that the elements of the arrays `a` and `b` have been given values. Both of the following are then legal method invocations:

```
SampleClass.doubleArrayElements(a);
SampleClass.doubleArrayElements(b);
```

Note that no square brackets are used when you give an entire array as an argument to a method.

An array type is a reference type just as a class type is, so, as with a class type argument, a method can change the data in an array argument. To phrase it more precisely, a method can change the values stored in the indexed variables of an array argument. This is illustrated by the preceding method `doubleArrayElements`.

An array type determines the base type of an array argument that may be plugged in for the parameter, but it does not specify the length of the array. An array knows its

length and stores it in the `length` instance variable. The same array parameter can be replaced with array arguments of different lengths. Note that the preceding method `doubleArrayElements` can take an array of any length as an argument.

length of array arguments

## Array Parameters and Array Arguments

An argument to a method may be an entire array. Array arguments are like objects of a class, in that the method can change the data in an array argument; that is, a method can change the values stored in the indexed variables of an array argument. A method with an array parameter is defined and invoked as illustrated by the following examples. Note that the array parameter specifies the base type of the array, but not the length of the array.

### Examples (of array parameters):

```
public class AClass
{

    public static void listChars(char[] a)
    {
        int i;
        for (i = 0; i < a.length; i++)
            System.out.println(a[i] + " ");
    }

    public static void zeroAll(int[] anArray)
    {
        int i;
        for (i = 0; i < anArray.length; i++)
            anArray[i] = 0;
    }
        ...
}
```

### Examples (of array arguments):

```
char[] c = new char[10];
int[] a = new int[10];
int[] b = new int[20];

    <Some code to fill the arrays goes here.>

AClass.listChars(c);
AClass.zeroAll(a);
AClass.zeroAll(b);
```

*Note that arrays a and b have different lengths. Also note that no square brackets are used with array arguments.*

## Self-Test Exercises

7. Consider the following class definition:

```
public class SomeClass
{
    public static void doSomething(int n)
    {
        <Some code goes in here.>
    }
<The rest of the definition is irrelevant to this question.>
```

Which of the following are acceptable method calls?

```
int[] a = {4, 5, 6};
int number = 2;
SomeClass.doSomething(number);
SomeClass.doSomething(a[2]);
SomeClass.doSomething(a[3]);
SomeClass.doSomething(a[number]);
SomeClass.doSomething(a);
```

8. Write a method definition for a static `void` method called `oneMore`, which has a formal parameter for an array of integers and increases the value of each array element by one. (The definition will go in some class, but you need only give the method definition.)

9. Write a method named `outOfOrder` that takes as a parameter an array of `double` and returns a value of type `int`. This method will test the array for being out of order, meaning that the array violates the condition:

```
a[0] <= a[1] <= a[2] <= ...
```

The method returns `-1` if the elements are not out of order; otherwise, it returns the index of the first element of the array that is out of order. For example, consider the declaration

```
double[] a = {1.2, 2.1, 3.3, 2.5, 4.5,
                 7.9, 5.4, 8.7, 9.9, 1.0};
```

In the array above, `a[2]` and `a[3]` are the first pair out of order, and `a[3]` is the first element out of order, so the method returns `3`. If the array were sorted, the method would return –1.

10. What is wrong with the following method definition? It will compile but does not work as you might hope.

```
public static void doubleSize(int[] a)
{
    a = new int[a.length * 2];
}
```

**Pitfall**

### Use of = and == with Arrays

Array types are reference types; that is, an array variable contains the memory address of the array it names. The assignment operator copies this memory address. For example, consider the following code:

assignment with arrays

```
double[] a = new double[10];
double[] b = new double[10];
int i;
for (i = 0; i < a.length; i++)
    a[i] = i;
b = a;
System.out.println("a[2] = " + a[2] + " b[2] = " + b[2]);
a[2] = 42;
System.out.println("a[2] = " + a[2] + " b[2] = " + b[2]);
```

This will produce the following output:

```
a[2] = 2.0 b[2] = 2.0
a[2] = 42.0 b[2] = 42.0
```

The assignment statement `b = a;` copies the memory address from a to b so that the array variable b contains the same memory address as the array variable a. After the assignment statement, a and b are two different names for the same array. Thus, when we change the value of `a[2]`, we are also changing the value of `b[2]`.

Unless you want two array variables to be two names for the same array (and on rare occasions you do want this), you should not use the assignment operator with arrays. If you want the arrays a and b in the preceding code to be different arrays with the same values in each index position, then instead of the assignment statement

```
b = a;
```

you need to use something like the following:

```
int i;
for (i = 0; (i < a.length) && (i < b.length); i++)
    b[i] = a[i];
```

Note that the above code will not make b an exact copy of a, unless a and b have the same length.

The equality operator == does not test two arrays to see if they contain the same values. It tests two arrays to see if they are stored in the same location in the computer's memory. For example, consider the following code:

== with arrays

```
int[] c = new int[10];
int[] d = new int[10];
```

```
int i;
for (i = 0; i < c.length; i++)
    c[i] = i;
for (i = 0; i < d.length; i++)
    d[i] = i;

if (c == d)
    System.out.println("c and d are equal by ==.");
else
    System.out.println("c and d are not equal by ==.");
```

This produces the output

```
c and d are not equal by ==
```

even though c and d contain the same integers in the same indexed variables. A comparison using == will say they are not equal because == only checks the contents of the array variables c and d, which are memory addresses, and c and d contain different memory addresses.

If you want to test two arrays to see if they contain the same elements, then you can define an `equalArrays` method for the arrays, just as you defined an `equals` method for a class. Display 6.3 contains one possible definition of `equalArrays` for arrays in a small demonstration class.

**Display 6.3 Testing Arrays for Equality *(Part 1 of 2)***

```
1   public class DifferentEquals
2   {
3       /**
4        A demonstration to see how == and an equalArrays method are different.
5       */
6       public static void main(String[] args)
7       {
8           int[] c = new int[10];
9           int[] d = new int[10];

10          int i;
11          for (i = 0; i < c.length; i++)
12              c[i] = i;

13          for (i = 0; i < d.length; i++)
14              d[i] = i;
```

*The arrays c and d contain the same integers in each index position.*

**Display 6.3 Testing Arrays for Equality *(Part 2 of 2)***

```
15          if (c == d)
16              System.out.println("c and d are equal by ==.");
17          else
18              System.out.println("c and d are not equal by ==.");

19          System.out.println("== only tests memory addresses.");

20          if (equalArrays(c, d))
21              System.out.println(
22                     "c and d are equal by the equalArrays method.");
23          else
24              System.out.println(
25                    "c and d are not equal by the equalArrays method.");

26          System.out.println(
27                  "An equalArrays method is usually a more useful test.");

28      }

29      public static boolean equalArrays(int[] a, int[] b)
30      {
31          if (a.length != b.length)
32              return false;
33          else
34          {
35              int i = 0;
36              while (i < a.length)
37              {
38                  if (a[i] != b[i])
39                      return false;
40                  i++;
41              }
42          }

43          return true;
44      }

45  }
```

**Sample Dialogue**

```
c and d are not equal by ==.
== only tests memory addresses.
c and d are equal by the equalArrays method.
An equalArrays method is usually a more useful test.
```

## ARGUMENTS FOR THE METHOD main

The heading for the `main` method of a program looks as if it has a parameter for an array of base type of `String`:

```
public static void main(String[] args)
```

The identifier `args` is in fact a parameter of type `String[]`. Since `args` is a parameter, it could be replaced by any other non-keyword identifier. The identifier `args` is traditional, but it is perfectly legal to use some other identifier.

We have never given `main` an array argument, or any other kind of argument, when we ran any of our programs. So, what did Java use as an argument to plug in for `args`? If no argument is given when you run your program, then a default empty array of strings is automatically provided as a default argument to `main` when you run your program.

It is possible to run a Java program in a way that provides an argument to plug in for this array of `String` parameters. You do not provide it as an array. You provide any number of string arguments when you run the program, and those string arguments will automatically be made elements of the array argument that is plugged in for `args` (or whatever name you use for the parameter to `main`). This is normally done by running the program from the command line of the operating system, like so:

```
java YourProgram Do Be Do
```

This will set `args[0]` to `"Do"`, `args[1]` to `"Be"`, `args[2]` to `"Do"`, and `args.length` to `3`. These three indexed variables can be used in the method `main`, as in the following sample program:

```
public class YourProgram
{
    public static void main(String[] args)
    {
        System.out.println(args[1] + " " + args[0]
                                          + " " + args[1]);
    }
}
```

If the above program is run from the command line as follows,

```
java YourProgram Do Be Do
```

the output produced by the program will be

```
Be Do Be
```

Be sure to note that the argument to `main` is an array of *strings*. If you want numbers, you must convert the string representations of the numbers to values of a number type or types.

### THE METHOD main HAS AN ARRAY PARAMETER

The heading for the main method of a program is as follows:

```
public static void main(String[] args)
```

The identifier args is a parameter for an array of base type String. The details are explained in the text.

## METHODS THAT RETURN AN ARRAY

In Java, a method may return an array. You specify the return type for a method that returns an array in the same way that you specify a type for an array parameter. For example, the following is an example of a method that returns an array:

```
public static char[] upperCaseVersion(char[] a)
{
    char[] temp = new char[a.length];
    char i;
    for (i = 0; i < a.length; i++)
         temp[i] = Character.toUpperCase(a[i]);
    return temp;
}
```

## Self-Test Exercises

11. Give the definition of a method called halfArray that has a single parameter for an array of base type double and that returns another array of base type double that has the same length and in which each element has been divided by 2.0. Make it a static method. To test it, you can add it to any class or, better yet, write a class with a test program in the method main.

12. What is wrong with the following method definition? It is an alternate definition of the method by the same name defined in the previous subsection. It will compile.

```
public static char[] upperCaseVersion(char[] a)
{
    char i;
    for (i = 0; i < a.length; i++)
         a[i] = Character.toUpperCase(a[i]);
    return a;
}
```

### Returning an Array

A method can return an array. The details are basically the same as for a method that returns an object of a class type.

#### Syntax (for a typical way of returning an array):

```
public static Base_Type[] Method_Name(Parameter_List)
{
    Base_Type[] temp = new Base_Type[Array_Size]
    <Some code to fill temp goes here.>
    return temp;
}
```

The method need not be static and need not be public. You do not necessarily need to use a local array variable like temp.

#### Example (assumed to be in a class definition):

```
public static int[] incrementedArray(int[] a, int increment)
{
    int[] temp = new int[a.length];
    int i;
    for (i = 0; i < a.length; i++)
        temp[i] = a[i] + increment;
    return temp;
}
```

### Array Type Names

Whenever you need an array type name, whether for the type of an array variable declaration, the type of an array parameter, or the type for a method that returns an array, you specify the type name in the same way.

#### Syntax:

```
Base_Type[]
```

#### Examples:

```
double[] a = new double[10];
int[] giveIntArray(char[] arrayParameter)
{ ... }
```

# 6.3 Programming with Arrays

*Never trust to general impressions, my boy*
*but concentrate yourself upon details.*

Sir Arthur Conan Doyle,
*A Case of Identity* (Sherlock Holmes)

In this section we discuss partially filled arrays and discuss how to use arrays as class instance variables.

## PARTIALLY FILLED ARRAYS

Often the exact size needed for an array is not known when a program is written or the size may vary from one run of the program to another. One common and easy way to handle this situation is to declare the array to be of the largest size the program could possibly need. The program is then free to use as much or as little of the array as is needed.

partially filled array

Partially filled arrays require some care. The program must keep track of how much of the array is used and must not reference any indexed variable that has not been given a meaningful value. The program in Display 6.4 illustrates this point. The program reads in a list of golf scores and shows how much each score differs from the average. This program will work for lists as short as 1 score, as long as 10 scores, and of any length in between. The scores are stored in the array `score`, which has 10 indexed variables,

**Display 6.4 Partially Filled Array *(Part 1 of 4)***

*Contrary to normal practice, this allows fractional scores, like 71.5. However, this makes it a better example for our purposes. (Anyway, when I play golf, losing a ball is only half a stroke penalty. Try it sometime.)*

```
1  import java.io.BufferedReader;
2  import java.io.InputStreamReader;
3  import java.io.IOException;

4  public class GolfScores
5  {
6      public static final int MAX_NUMBER_SCORES = 10;

7      /**
8       Shows differences between each of a list of golf scores and their average.
9      */
10     public static void main(String[] args) throws IOException
11     {
12         double[] score = new double[MAX_NUMBER_SCORES];
13         int numberUsed = 0;

14         System.out.println("This program reads golf scores and shows");
15         System.out.println("how much each differs from the average.");
```

**Display 6.4 Partially Filled Array *(Part 2 of 4)***

```
16          System.out.println("Enter golf scores:");
17          numberUsed = fillArray(score);
18          showDifference(score, numberUsed);
19      }

20      /**
21       Reads values into the array a. Returns the number of values placed in the array a.
22      */
23      public static int fillArray(double[] a) throws IOException
24      {
25          System.out.println("Enter up to " + a.length
26                                 + " nonnegative numbers, one per line.");
27          System.out.println("Mark the end of the list with a negative number.");
28          BufferedReader keyboard =
29                                 new BufferedReader(new InputStreamReader(System.in));

30          double next;
31          int index = 0;
32          next = stringToDouble(keyboard.readLine());
33          while ((next >= 0) && (index < a.length))
34          {
35              a[index] = next;
36              index++;
37              next = stringToDouble(keyboard.readLine());
38             //index is the number of array indexed variables used so far.
39          }
40         //index is the total number of array indexed variables used.

41          if (next >= 0)
42               System.out.println("Could only read in "
43                                     + a.length + " input values.");

44          return index;
45      }

46      /**
47       Precondition: numberUsed <= a.length.
48                     a[0] through a[numberUsed-1] have values.
49       Returns the average of numbers a[0] through a[numberUsed-1].
50      */
```

The value of `index` is the number of values stored in the array.

**Display 6.4 Partially Filled Array *(Part 3 of 4)***

```
51      public static double computeAverage(double[] a, int numberUsed)
52      {
53          double total = 0;
54          for (int index = 0; index < numberUsed; index++)
55              total = total + a[index];
56          if (numberUsed > 0)
57          {
58              return (total/numberUsed);
59          }
60          else
61          {
62              System.out.println("ERROR: Trying to average 0 numbers.");
63              System.out.println("computeAverage returns 0.");
64              return 0;
65          }
66      }

67      private static double stringToDouble(String stringObject)
68      {
69          return Double.parseDouble(stringObject.trim());
70      }

71      /**
72       Precondition: numberUsed <= a.length.
73                    The first numberUsed indexed variables of a have values.
74        Postcondition: Gives screen output showing how much each of the first
75        numberUsed elements of the array a differ from their average.
76      */
77      public static void showDifference(double[] a, int numberUsed)
78      {
79          double average = computeAverage(a, numberUsed);
80          System.out.println("Average of the " + numberUsed
81                                                + " scores = " + average);
82          System.out.println("The scores are:");
83          for (int index = 0; index < numberUsed; index++)
84          System.out.println(a[index] + " differs from average by "
85                                      + (a[index] – average));
86      }
87  }
```

**Display 6.4 Partially Filled Array *(Part 4 of 4)***

### Sample Dialogue

```
This program reads golf scores and shows
how much each differs from the average.
Enter golf scores:
Enter up to 10 nonnegative numbers, one per line.
Mark the end of the list with a negative number.
69
74
68
-1
Average of the 3 scores = 70.3333
The scores are:
69.0 differs from average by -1.33333
74.0 differs from average by 3.66667
68.0 differs from average by -2.33333
```

but the program uses only as much of the array as it needs. The variable `numberUsed` keeps track of how many elements are stored in the array. The elements (that is, the scores) are stored in positions `score[0]` through `score[numberUsed - 1]`. The details are very similar to what they would be if `numberUsed` were `score.length` and the entire array were used. Note that the variable `numberUsed` usually must be an argument to any method that manipulates the partially filled array. For example, the methods `showDifference` and `computeAverage` use the argument `numberUsed` to ensure that only meaningful array indices are used.

## Self-Test Exercises

13. Complete the definition of the following method that could be added to the class `GolfScores` in Display 6.4:

```
/**
 Precondition: numberUsed <= argumentArray.length;
 the first numberUsed indexed variables of argumentArray
 have values.
 Returns an array of length numberUsed whose ith element
 is argumentArray[i] - adjustment.
*/
public static double[] differenceArray(
 double[] argumentArray, int numberUsed, double adjustment)
```

14. Rewrite the class `GolfScores` from Display 6.4 using the method `differenceArray` from exercise 13.

15. Rewrite the class `GolfScores` from Display 6.4 making the array of scores a static variable. Also, make the `int` variable `numberUsed` a static variable. Start with Display 6.4, not with the answer to exercise 14. Hint: All, or at least most, methods will have no parameters.

**Example**

### A Class for Partially Filled Arrays

If you are going to use some array in a disciplined way, such as using the array as a partially filled array, then it is often best to create a class that has the array as an instance variable and to have the constructors and methods of the class provide the needed operations as methods. For example, in Display 6.5 we have written a class for a partially filled array of doubles. In Display 6.6 we have rewritten the program in Display 6.4 using this class.

alternative coding

In Display 6.6 we have written the code to be exactly analogous to that of Display 6.4 so that you could see how one program mirrors the other. However, this resulted in occasionally recomputing a value several times. For example, the method `computeAverage` has the following expression three times:

```
a.getNumberOfElements()
```

Since the `PartiallyFilledArray` *a* is not changed in this method, these each return the same value. Some programmers advocate computing this value only once and saving the value in a variable. These programmers would use something like the following for the definition of `computeAverage` rather than what we used in Display 6.6. The variable `numberOfElementsIna` is used to save a value so it need not be recomputed.

```
public static double computeAverage(PartiallyFilledArray a)
{
    double total = 0;
    double numberOfElementsIna = a.getNumberOfElements();
    for (int index = 0; index < numberOfElementsIna; index++)
        total = total + a.getElement(index);
    if (numberOfElementsIna > 0)
    {
        return (total/numberOfElementsIna);
    }
    else
    {
        System.out.println(
                     "ERROR: Trying to average 0 numbers.");
        System.out.println("computeAverage returns 0.");
        return 0;
    }
}
```

This is not likely to produce a noticeable difference in the efficiency of the program in Display 6.6, but if the number of elements in the PartiallyFilledArray were large so that the for loop would be executed many times, it might make a difference in a situation where efficiency is critical.

**Display 6.5 Partially Filled Array Class *(Part 1 of 4)***

```
1   /**
2    Class for a partially filled array of doubles. The class enforces the
3    following invariant: All elements are at the beginning of the array in
4    locations 0, 1, 2, and so forth up to a highest index with no gaps.
5   */
6   public class PartiallyFilledArray
7   {
8       private int maxNumberElements; //Same as a.length
9       private double[] a;
10      private int numberUsed; //Number of indices currently in use

11      /**
12       Sets the maximum number of allowable elements to 10.
13      */
14      PartiallyFilledArray()
15      {
16          maxNumberElements = 10;
17          a = new double[maxNumberElements];
18          numberUsed = 0;
19      }

20      /**
21       Precondition arraySize > 0.
22      */
23      PartiallyFilledArray(int arraySize)
24      {
25          if (arraySize <= 0)
26          {
27              System.out.println("Error Array size zero or negative.");
28              System.exit(0);
29          }
30          maxNumberElements = arraySize;
31          a = new double[maxNumberElements];
32          numberUsed = 0;
33      }
```

**Display 6.5 Partially Filled Array Class *(Part 2 of 4)***

```
34      PartiallyFilledArray(PartiallyFilledArray original)
35      {
36          if (original == null)
37          {
38              System.out.println("Fatal Error: aborting program.");
39              System.exit(0);
40          }
41          maxNumberElements =
42                    original.maxNumberElements;
43          numberUsed = original.numberUsed;
44          a = new double[maxNumberElements];
45          for (int i = 0; i < numberUsed; i++)
46              a[i] = original.a[i];
47      }

48      /**
49       Adds newElement to the first unused array position.
50      */
51      public void add(double newElement)
52      {
53          if (numberUsed >= a.length)
54          {
55              System.out.println("Error: Adding to a full array.");
56              System.exit(0);
57          }
58          else
59          {
60              a[numberUsed] = newElement;
61              numberUsed++;
62          }
63      }

64      public double getElement(int index)
65      {
66          if (index < 0 || index >= numberUsed)
67          {
68              System.out.println("Error:Illegal or unused index.");
69              System.exit(0);
70          }

71          return a[index];
72      }
```

*Note that the instance variable a is a copy of `orignal.a`. The following would not be correct: `a = original.a;` This point is discussed in the subsection entitled "Privacy Leaks with Array Instance Variables."*

**Display 6.5 Partially Filled Array Class *(Part 3 of 4)***

```
73         /**
74          index must be an index in use or the first unused index.
75         */
76         public void resetElement(int index, double newValue)
77         {
78             if (index < 0 || index >= maxNumberElements)
79             {
80                 System.out.println("Error:Illegal index.");
81                 System.exit(0);
82             }
83             else if (index > numberUsed)
84             {
85                 System.out.println(
86                           "Error: Changing an index that is too large.");
87                 System.exit(0);
88             }
89             else
90                 a[index] = newValue;
91         }

92         public void deleteLast()
93         {
94             if (empty())
95             {
96                 System.out.println("Error:Deleting from an empty array.");
97                 System.exit(0);
98             }
99             else
100                numberUsed--;
101        }

102        /**
103         Deletes the element in position index. Moves down all elements with
104         indices higher than the deleted element.
105        */
106        public void delete(int index)
107        {
108            if (index < 0 || index >= numberUsed)
109            {
110                System.out.println("Error:Illegal or unused index.");
111                System.exit(0);
112            }

113            for (int i = index; i < numberUsed; i++)
114                 a[i] = a[i + 1];
115            numberUsed--;
116        }
```

**Display 6.5 Partially Filled Array Class *(Part 4 of 4)***

```
117     public boolean empty()
118     {
119         return (numberUsed == 0);
120     }

121     public boolean full()
122     {
123         return (numberUsed == maxNumberElements);
124     }

125     public int getMaxCapacity()
126     {
127         return maxNumberElements;
128     }
129
130     public int getNumberOfElements()
131     {
132         return numberUsed;
133     }
134 }
```

**Display 6.6 Display 6.4 Redone Using the Class PartiallyFilledArray *(Part 1 of 3)***

```
1   import java.io.BufferedReader;
2   import java.io.InputStreamReader;
3   import java.io.IOException;

4   /**
5    Demonstrates Using the class PartiallyFilledArray,
6   */
7   public class GolfScoresVersion2
8   {

9       public static final int MAX_NUMBER_SCORES = 10;

10      /**
11       Shows the differences between each of a list of golf scores and their average.
12      */
13      public static void main(String[] args) throws IOException
14      {
15          PartiallyFilledArray score =
16                               new PartiallyFilledArray(MAX_NUMBER_SCORES);
```

*Sample dialog is the same as in Display 6.4.*

**Display 6.6 Display 6.4 Redone Using the Class PartiallyFilledArray *(Part 2 of 3)***

```
17          System.out.println("This program reads golf scores and shows");
18          System.out.println("how much each differs from the average.");

19          System.out.println("Enter golf scores:");
20          fillArray(score);
21          showDifference(score);
22      }


23      /**
24       Reads values into the PartiallyFilledArray a.
25      */
26      public static void fillArray(PartiallyFilledArray a) throws IOException
27      {
28          System.out.println("Enter up to " + a.getMaxCapacity()
29                                 + " nonnegative numbers, one per line.");
30          System.out.println("Mark the end of the list with a negative number");
31          BufferedReader keyboard = new BufferedReader(
32                           new InputStreamReader(System.in));

33          double next = stringToDouble(keyboard.readLine());
34          while ((next >= 0) && (!a.full()))
35          {
36              a.add(next);
37              next = stringToDouble(keyboard.readLine());
38          }

39          if (next >= 0)
40               System.out.println("Could only read in "
41                                  + a.getMaxCapacity() + " input values.");
42      }

43      private static double stringToDouble(String stringObject)
44      {
45          return Double.parseDouble(stringObject.trim());
46      }

47      /**
48       Returns the average of numbers in the PartiallyFilledArray a.
49      */
50      public static double computeAverage(PartiallyFilledArray a)
51      {
52          double total = 0;
53          for (int index = 0; index < a.getNumberOfElements(); index++)
54              total = total + a.getElement(index);
```

**Display 6.6 Display 6.4 Redone Using the Class PartiallyFilledArray *(Part 3 of 3)***

```
55          if (a.getNumberOfElements() > 0)
56          {
57              return (total/a.getNumberOfElements());
58          }
59          else
60          {
61              System.out.println("ERROR: Trying to average 0 numbers.");
62              System.out.println("computeAverage returns 0.");
63              return 0;
64          }
65      }

66      /**
67       Gives screen output showing how much each of the
68       elements in the PartiallyFilledArray a differ from the average.
69      */
70      public static void showDifference(PartiallyFilledArray a)
71      {
72          double average = computeAverage(a);
73          System.out.println("Average of the " + a.getNumberOfElements()
74                                            + " scores = " + average);
75          System.out.println("The scores are:");
76          for (int index = 0; index < a.getNumberOfElements(); index++)
77              System.out.println(a.getElement(index) + " differs from average by "
78                                               + (a.getElement(index) - average));
79      }

80  }
```

**Tip**

### Accessor Methods Need Not Simply Return Instance Variables

Note that in the class `PartiallyFilledArray` in Display 6.5, there is no accessor method that returns a copy of the entire instance variable `a`. The reason that was not done is that, when the class is used as intended, a user of the class `PartiallyFilledArray` would have no need for the entire array `a`. That is an implementation detail. The other methods that start with `get` allow a programmer using the class to obtain all the data that he or she needs.

## PRIVACY LEAKS WITH ARRAY INSTANCE VARIABLES

In Chapter 5 we explained why it is a compromise of privacy for a class to have an accessor (or other method) that returns a reference to a private mutable object. As we noted there, an accessor method should instead return a reference to a *deep copy* of the private object. (See the Pitfall subsection of Chapter 5 entitled "Privacy Leaks.") At the time we had in mind returning the contents of a private instance variable of a class type. However, the lesson applies equally well to private instance variables of an array type.

For example, suppose that, despite what we said in the previous Programming Tip, you decide that you want an accessor method for the array instance variable in the class `PartiallyFilledArray` in Display 6.5. You might be tempted to define the accessor method as follows:

```
public double[] getInsideArray()// Problematic version
{
    return a;
}
```

As indicated in the comment, this definition has a problem. The problem is that this accessor method allows a programmer to change the array object named by the private instance variable `a` in ways that bypass the checks built into the mutator methods of the class `PartiallyFilledArray`. To see why this is true *suppose we had added this definition of the method* `getInsideArray` *to the class* `PartiallyFilledArray`, and consider the following code:

```
PartiallyFilledArray leakyArray =
                    new PartiallyFilledArray(10);
double[] arrayName = leakyArray.getInsideArray();
```

The variable `arrayName` and the private instance variable `a` now contain the same reference, so both `arrayName` and the private instance variable `a` name the same array. Using `arrayName` as a name for the array named by the private instance variable `a`, we can now fill the indexed variables of `a` in any order and need not fill the array starting at the first element. This violates the spirit of the `private` modifier for the array instance variable `a`. For this reason, the accessor method `getInsideArray` should return a deep copy of the array named by the private instance variable `a`. A safe definition of `getInsideArray` is the following:

```
public double[] getInsideArray()// Good version
{
    //Recall that maxNumberElements == a.length.
    double[] temp = new double[maxNumberElements];
    for (int i = 0; i < maxNumberElements; i++)
        temp[i] = a[i];
    return temp;
}
```

If a private instance variable is an array type that has a class as its base type, then you need to be sure to make copies of the class objects in the array when you make a copy of the array. This is illustrated by the toy class in Display 6.7.

Display 6.7 also includes a copy constructor. As illustrated in that display, the copy constructor should make a completely independent copy of the array instance variable (that is, a deep copy) in the same way that the accessor method does. This same point is also illustrated by the copy constructor in Display 6.5.

**Display 6.7 Accessor Method for an Array Instance Variable**

```
1   /**
2    Demonstrates the correct way to define an accessor
3    method to a private array of class objects.
4   */
5   public class ToyExample
6   {
7       private Date[] a;

8       public ToyExample(int arraySize)
9       {
10          a = new Date[arraySize];
11          for (int i = 0; i < arraySize; i++)
12              a[i] = new Date();
13      }

14      public ToyExample(ToyExample object)
15      {
16          int lengthOfArrays = object.a.length;
17          this.a = new Date[lengthOfArrays];
18          for (int i = 0; i < lengthOfArrays; i++)
19              this.a[i] = new Date(object.a[i]);
20      }

21      public Date[] getDateArray()
22      {
23          Date[] temp = new Date[a.length];
24          for (int i = 0; i < a.length; i++)
25              temp[i] = new Date(a[i]);
26          return temp;
27      }

        <There presumably are other methods that are not shown,
                    but they are irrelevant to the point at hand.>

28  }
```

*The class Date is defined in Display 4.11, but you do not need to know the details of the definition to understand the point of this example.*

Copy constructor for ToyExample (line 14)

Copy constructor for Date (line 19)

Accessor method (line 21)

Copy constructor for Date (line 25)

## A PREVIEW OF VECTORS

Java and many other programming languages have objects known as *vectors*, which are very much like objects of our class `PartiallyFilledArray` in Display 6.5. However, even after you learn about vectors, there will still be situations where the class `PartiallyFilledArray` is preferable to vectors. We will discuss vectors in Chapter 15.

## Self-Test Exercises

16. Define a method named `removeAll` that can be added to the class `PartialyFilledArray`. The method `removeAll` has no parameters. When invoked the method `removeAll` deletes all the elements in its calling object.

17. Define a method named `increaseCapacity` that can be added to the class `PartiallyFilledArray` in Display 6.5. The method has one `int` parameter named `newCapacity` that increases the capacity of the `PartialyFilledArray` so that it can hold up to `newCapacity` numbers. If `newCapacity` is less than or equal to `maxNumberOfElements`, then the method does nothing. If `newCapacity` is greater than `maxNumberOfElements`, then `maxNumberElements` is set equal to `newCapacity` and a new array of length `newCapacity` is created for the array instance variable `a`. The old values of the array instance variable are copied to the newly created array.

## Example

### Sorting an Array

In this example we define a method called `sort` that will sort a partially filled array of numbers so that they are ordered from smallest to largest.

The procedure `sort` has one array parameter `a`. The array `a` will be partially filled, so there is an additional formal parameter called `numberUsed`, which tells how many array positions are used. Thus, the heading for the method `sort` will be

```
public static void sort(double[] a, int numberUsed)
```

The method `sort` rearranges the elements in array `a` so that after the method call is completed, the elements are sorted as follows:

```
a[0] ≤ a[1] ≤ a[2] ≤ ... ≤ a[numberUsed - 1]
```

selection sort

The algorithm we use to do the sorting is called **selection sort**. It is one of the easiest of the sorting algorithms to understand.

One way to design an algorithm is to rely on the definition of the problem. In this case the problem is to sort an array *a* from smallest to largest. That means rearranging the values so that *a*[0] is the smallest, *a*[1] the next smallest, and so forth. That definition yields an outline for the **selection sort** algorithm:

```
for (int index = 0; index < numberUsed; index++)
      Place the indexth smallest element in a[index]
```

There are many ways to realize this general approach. The details could be developed by using two arrays and copying the elements from one array to the other in sorted order, but using one array should be both adequate and economical. Therefore, the method `sort` uses only the one array containing the values to be sorted. The method `sort` rearranges the values in the array *a* by interchanging pairs of values. Let us go through a concrete example so that you can see how the algorithm works.

Consider the array shown in Display 6.8. The selection sort algorithm will place the smallest value in *a*[0]. The smallest value is the value in *a*[4]. So, the algorithm interchanges the values of *a*[0] and *a*[4]. The algorithm then looks for the next smallest element. The value in *a*[0] is now the smallest element, so the next smallest element is the smallest of the remaining elements *a*[1], *a*[2], *a*[3],..., *a*[9]. In the example in Display 6.8 the next smallest element is in *a*[6], so the algorithm interchanges the values of *a*[1] and *a*[6]. This positioning of the second smallest element is illustrated in the fourth and fifth array pictures in Display 6.8. The algorithm then positions the third smallest element, and so forth. As the sorting proceeds, the beginning array elements are set equal to the correct sorted values. The sorted portion of the array grows by adding elements one after the other from the elements in the unsorted end of the array. Notice that the algorithm need not do anything with the value in the last indexed variable, *a*[9], because once the other elements are positioned correctly, *a*[9] must also have the correct value. After all, the correct value for *a*[9] is the smallest value left to be moved, and the only value left to be moved is the value that is already in *a*[9].

The definition of the method `sort`, included in a class, is given in Display 6.9. `sort` uses the method `indexOfSmallest` to find the index of the smallest element in the unsorted end of the array, then it does an interchange to move this next smallest element down into the sorted part of the array.

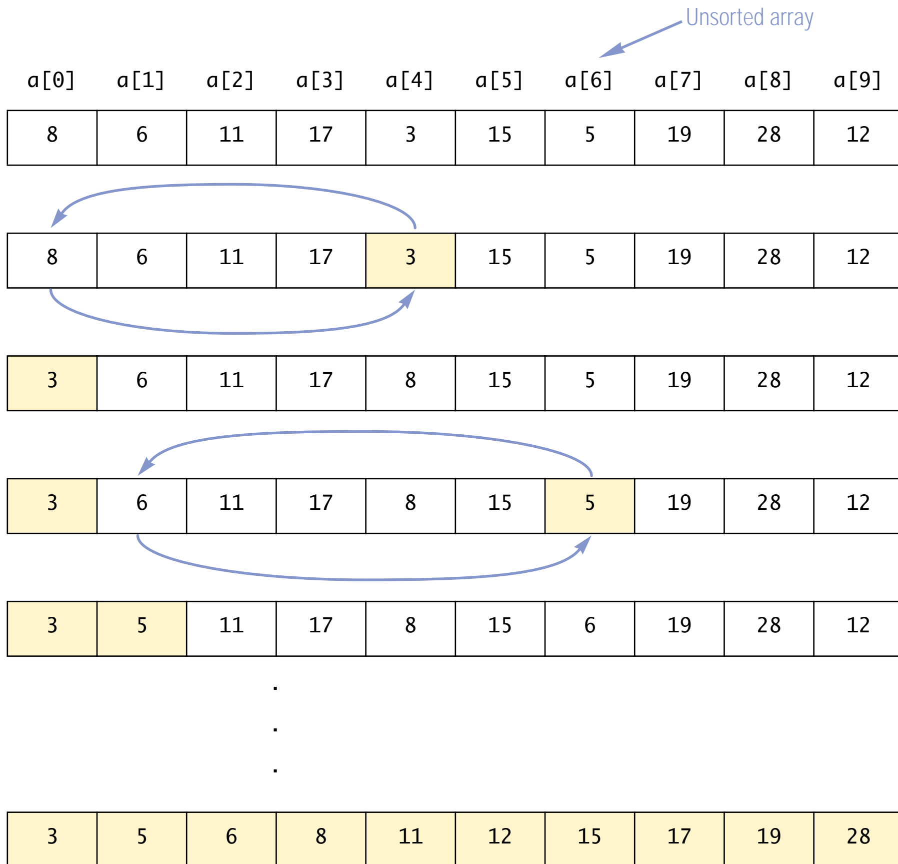
indexOf-Smallest

The method `interchange`, shown in Display 6.9, is used to interchange the values of indexed variables. For example, the following call will interchange the values of *a*[0] and *a*[4]:

```
interchange(0, 4, a);
```

A sample use of the `sort` method is given in Display 6.10.

**Display 6.8 Selection Sort**




## Self-Test Exercises

18. How would you need to change the method `sort` in Display 6.9 so that it can sort an array of values of type `double` into decreasing order, instead of increasing order?

19. If an array of `int` values has a value that occurs twice (like `b[0] == 42` and `b[7] == 42`) and you sort the array using the method `SelectionSort.sort`, will there be one or two copies of the repeated value after the array is sorted?

**Display 6.9 Selection Sort Class *(Part 1 of 2)***

```
1   public class SelectionSort
2   {
3       /**
4        Precondition: numberUsed <= a.length;
5                     The first numberUsed indexed variables have values.
6        Action: Sorts a so that a[0] <= a[1] <= ... <= a[numberUsed - 1].
7       */
8       public static void sort(double[] a, int numberUsed)
9       {
10          int index, indexOfNextSmallest;
11          for (index = 0; index < numberUsed - 1; index++)
12          {//Place the correct value in a[index]:
13              indexOfNextSmallest = indexOfSmallest(index, a, numberUsed);
14              interchange(index,indexOfNextSmallest, a);
15              //a[0] <= a[1] <=...<= a[index] and these are the smallest
16              //of the original array elements. The remaining positions
17              //contain the rest of the original array elements.
18          }
19      }

20      /**
21       Returns the index of the smallest value among
22       a[startIndex], a[startIndex+1], ... a[numberUsed - 1]
23      */
24      private static int indexOfSmallest(int startIndex,
25                                          double[] a, int numberUsed)
26      {
27          double min = a[startIndex];
28          int indexOfMin = startIndex;
29          int index;
30          for (index = startIndex + 1; index < numberUsed; index++)
31              if (a[index] < min)
32              {
33                  min = a[index];
34                  indexOfMin = index;
35                  //min is smallest of a[startIndex] through a[index]
36              }
37          return indexOfMin;
38      }
```

**Display 6.9 Selection Sort Class *(Part 2 of 2)***

```
    /**
     Precondition: i and j are legal indices for the array a.
     Postcondition: Values of a[i] and a[j] have been interchanged.
    */
    private static void interchange(int i, int j, double[] a)
    {
        double temp;
        temp = a[i];
        a[i] = a[j];
        a[j] = temp; //original value of a[i]
    }

}
```

# 6.4 Multidimensional Arrays

*Two indices are better than one.*

Anonymous

Java allows you to declare arrays with more than one index. In this section we describe these multidimensional arrays.

## MULTIDIMENSIONAL ARRAY BASICS

array declarations
indexed variables

It is sometimes useful to have an array with more than one index, and this is allowed in Java. The following creates an array of characters called `page`. The array `page` has two indices: the first index ranging from `0` to `29` and the second from `0` to `99`.

```
char[][] page = new char[30][100];
```

This is equivalent to the following two steps:

```
char[][] page;
page = new char[30][100];
```

The indexed variables for this array each have two indices. For example, `page[0][0]`, `page[15][32]`, and `page[29][99]` are three of the indexed variables for this array. Note that each index must be enclosed in its own set of square brackets. As was true of the one-dimensional arrays we have already seen, each indexed variable for a multidimensional array is a variable of the base type, in this case the type `char`.

**Display 6.10 Demonstration of the SelectionSort Class**

```
1  public class SelectionSortDemo
2  {
3      public static void main(String[] args)
4      {
5          double[] b = {7.7, 5.5, 11, 3, 16, 4.4, 20, 14, 13, 42};

6          System.out.println("Array contents before sorting:");
7          int i;
8          for (i = 0; i < b.length; i++)
9              System.out.print(b[i] + " ");
10         System.out.println();
11
12         SelectionSort.sort(b, b.length);

13         System.out.println("Sorted array values:");
14         for (i = 0; i < b.length; i++)
15             System.out.print(b[i] + " ");
16         System.out.println();
17     }
18 }
```

**SAMPLE DIALOGUE**

```
Array contents before sorting:
7.7 5.5 11.0 3.0 16.0 4.4 20.0 14.0 13.0 42.0
Sorted array values:
3.0 4.4 5.5 7.7 11.0 13.0 14.0 16.0 20.0 42.0
```

An array may have any number of indices, but perhaps the most common number of indices is two. A two-dimensional array can be visualized as a two-dimensional display with the first index giving the row and the second index giving the column. For example, the array indexed variables of the two-dimensional array a declared and created as

```
char[][] a = new char[5][12];
```

can be visualized as follows:

```
a[0][0], a[0][1], a[0][2], ..., a[0][11]
a[1][0], a[1][1], a[1][2], ..., a[1][11]
a[2][0], a[2][1], a[2][2], ..., a[2][11]
a[3][0], a[3][1], a[3][2], ..., a[3][11]
a[4][0], a[4][1], a[4][2], ..., a[4][11]
```

You might use the array `a` to store all the characters on a (very small) page of text that has five lines (numbered 0 through 4) and 12 characters on each line (numbered 0 through 11).

### Declaring and Creating a Multidimensional Array

You declare a multidimensional array variable and create a multidimensional array object in basically the same way that you create and name a one-dimensional array. You simply use as many square brackets as there are indices.

**SYNTAX:**

```
Base_Type[]...[] Variable_Name = new Base_Type[Length_1]...[Length_n];
```

**EXAMPLES:**

```
char[][] a = new char[5][12];
char[][] page = new char[30][100];
double[][] table = new double[100][10];
int[][][] figure = new int[10][20][30];
Person[][] entry = new Person[10][10];
```

`Person` is a class.

A multidimensional array is an array of arrays

In Java, a two-dimensional array, such as `a`, is actually an array of arrays. The above array `a` is actually a one-dimensional array of size `5`, whose base type is a one-dimensional array of characters of size `12`. This is diagrammed in Display 6.11. As shown in that display, the array variable `a` contains a reference to a one-dimensional array of length `5` and with a base type of `char[]`; that is, the base type of `a` is the type for an entire one-dimensional array of characters. Each indexed variable `a[0]`, `a[1]`, and so forth contains a reference to a one-dimensional array of characters.

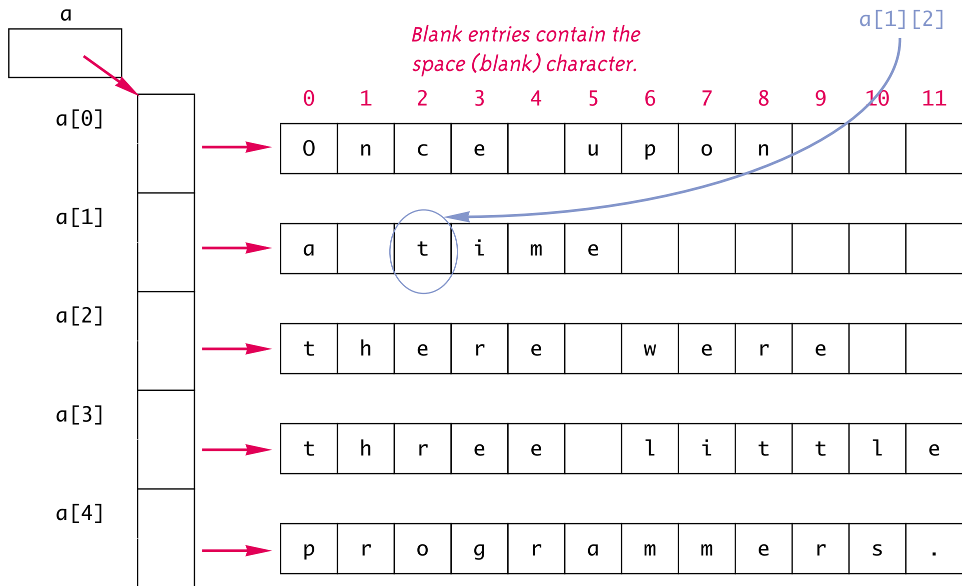
A three-dimensional array is an array of arrays of arrays, and so forth for higher dimensions.

Normally, the fact that a two-dimensional array is an array of arrays need not concern you, and you can usually act as if the array `a` is actually an array with two indices (rather than an array of arrays, which is harder to keep track of). There are, however, some situations where a two-dimensional array looks very much like an array of arrays. For example, you will see that when using the instance variable `length`, you must think of a two-dimensional array as an array of arrays.

**Display 6.11 Two-Dimensional Array as an Array of Arrays**

```
char[][] a = new char[5][12];
```

*Code that fills the array is not shown.*




We will see that these can and should be replaced with expressions involving the `length` instance variable.

```
int row, column;
for (row = 0; row < 5; row++)
{
    for (column = 0; column < 12; column++)
       System.out.print(a[row][column]);
    System.out.println();
}
```

*Produces the following output:*

```
Once upon
a time
there were
three little
programmers.
```

## USING THE length INSTANCE VARIABLE

Suppose you want to fill all the elements in the following two-dimensional array with `'Z'`:

```
char[][] page = new char[30][100];
```

You can use a nested `for` loop such as the following:

```
int row, column;
for (row = 0; row < page.length; row++)
   for (column = 0; column < page[row].length; column++)
     page[row][column] = 'Z';
```

Let's analyze this nested `for` loop in a bit more detail. The array `page` is actually a one-dimensional array of length `30`, and each of the 30 indexed variables `page[0]` through `page[29]` is a one-dimensional array with base type `char` and with a length of `100`. That is why the first `for` loop is terminated using `page.length`. For a two-dimensional array like `page`, the value of `length` is the number of first indices or, equivalently, the number of rows—in this case, `30`. Now let's consider the inside `for` loop.

The `0`th row in the two-dimensional array `page` is the one-dimensional array `page[0]`, and it has `page[0].length` entries. More generally, `page[row]` is a one-dimensional array of `char`s, and it has `page[row].length` entries. That is why the inner `for` loop is terminated using `page[row].length`. Of course, in this case, `page[0].length`, `page[1].length`, and so forth through to `page[29].length` are all equal and all equal to `100`. (If you read the optional section entitled "Ragged Arrays," you will see that these need not all be equal.)

## Self-Test Exercises

20. What is the output produced by the following code?

```
int[][] myArray = new int[4][4];
int index1, index2;
for (index1 = 0; index1 < myArray.length; index1++)
    for (index2 = 0;
              index2 < myArray[index1].length; index2++)
        myArray[index1][index2] = index2;
for (index1 = 0; index1 < myArray.length; index1++)
{
    for (index2 = 0;
              index2 < myArray[index1].length; index2++)
        System.out.print(myArray[index1][index2] + " ");
    System.out.println();
}
```

21. Write code that will fill the array `a` (declared and created below) with numbers typed in at the keyboard. The numbers will be input five per line, on four lines.

```
int[][] a = new int[4][5];
```

Hint: It will help to use the class `StringTokenizer`.

22. Write a method definition for a static `void` method called `echo` such that the following method call will echo the input described in exercise 21, and will echo it in the same format as we specified for the input (that is, four lines of five numbers per line):

```
Class_Name.echo(a);
```

## RAGGED ARRAYS

There is no need for each row in a two-dimensional array to have the same number of entries. Different rows can have different numbers of columns. These sorts of arrays are called **ragged arrays**.

To help explain the details, let's start with an ordinary, nonragged two-dimensional array, created as follows:

```
double[][] a = new double[3][5];
```

This is equivalent to the following:

```
double[][] a;
a = new double[3][];
a[0] = new double[5];
a[1] = new double[5];
a[2] = new double[5];
```

The line

```
a = new double[3][];
```

makes `a` the name of an array with room for `3` entries, each of which can be an array of `double`s that can be of any length. The next three lines each create an array of `double`s of length `5` to be named by `a[0]`, `a[1]`, and `a[2]`. The net result is a two-dimensional array of base type `double` with three rows and five columns.

If you want, you can make each of `a[0]`, `a[1]`, and `a[2]` a different length. The following code makes a ragged array `b` in which each row has a different length:

```
double[][] b;
b = new double[3][];
b[0] = new double[5];
b[1] = new double[10];
b[2] = new double[4];
```

There are situations in which you can profitably use ragged arrays, but most applications do not require them. However, if you understand ragged arrays, you will have a better understanding of how all multidimensional arrays work in Java.

## MULTIDIMENSIONAL ARRAY PARAMETERS AND RETURNED VALUES

array arguments

Methods may have multidimensional array parameters and may have a multidimensional array type as the type for the value returned. The situation is similar to that of the one-dimensional case, except that you use more square brackets when specifying the type name. For example, the following method will display a two-dimensional array in the usual way as rows and columns:[4]

```
public static void showMatrix(int[][] a)
{
    int row, column;
    for (row = 0; row < a.length; row++)
    {
        for (column = 0; column < a[row].length; column++)
            System.out.print(a[row][column] + "  ");
        System.out.println();
    }
}
```

returning an array

If you want to return a multidimensional array, you use the same kind of type specification as you use for a multidimensional array parameter. For example, the following method returns a two-dimensional array with base type `double`:

```
/**
 Precondition: Each dimension of a is at least
 the value of size.
 The array returned is the same as the size-by-size
 upper-left corner of the array a.
*/
public static double[][] corner(double[][] a, int size)
{
    double[][] temp = new double[size][size];
    int row, column;
    for (row = 0; row < size; row++)
        for (column = 0; column < size; column++)
            temp[row][column] = a[row][column];
    return temp;
}
```

[4] It is worth noting that this method works fine for ragged arrays.

**Example**

### A Grade Book Class

Display 6.12 contains a class for grade records in a class whose only recorded scores are quiz scores. An object of this class has three array instance variables. One is a two-dimensional array named `grade` that records the grade of each student on each quiz. For example, the score that student number 4 received on quiz number 1 is recorded in `grade[3][0]`. Since the student numbers and quiz numbers start with 1 and the array indices start with 0, we subtract one from the student number or quiz number to obtain the corresponding array index.

All the raw data is in the array `grade`, but two other arrays hold computed data. The array `studentAverage` is used to record the average quiz score for each of the students. For example, the program will set `studentAverage[0]` equal to the average of the quiz scores received by student 1, `studentAverage[1]` equal to the average of the quiz scores received by student 2, and so forth. The array `quizAverage` will be used to record the average score for each quiz. For example, the program will set `quizAverage[0]` equal to the average of all the student scores for quiz 1, `quizAverage[1]` will record the average score for quiz 2, and so forth. Display 6.13 illustrates the relationship between the arrays `grade`, `studentAverage`, and `quizAverage`. In that display, we have shown some sample data for the array `grade`. The data in `grade`, in turn, determine the values that are stored in `studentAverage` and in `quizAverage`. Display 6.13 also shows these computed values for `studentAverage` and `quizAverage`. The two arrays `studentAverage` and `quizAverage` are created and filled by the constructor that creates the `GradeBook` object. (The constructors do this by calling private helping methods.)

no-argument constructor

The no-argument constructor for the class `GradeBook` obtains the data for the array instance variable `grade` via a dialog with the user. Although this is not my favorite way to define a no-argument constructor, some programmers like it and you should see an example of it. Another alternative would be to have a no-argument constructor that essentially does nothing and then have an input method that sets all the instance variables including creating the array objects.

A very simple demonstration program along with the dialog it produces is given in Display 6.14.

**Self-Test Exercises**

23. Write a method definition for a method with the following heading. The method is to be added to the class `GradeBook` in Display 6.12.

```
/**
 Returns the grade that student numbered studentNumber
 received on quiz number quizNumber.
*/
public int getGrade(int studentNumber, int quizNumber)
```

**Display 6.12 A Grade Book Class *(Part 1 of 4)***

```
1   import java.io.BufferedReader;
2   import java.io.InputStreamReader;
3   import java.io.IOException;

4   public class GradeBook
5   {

6       private int numberOfStudents; // Same as studentAverage.length.
7       private int numberOfQuizzes; // Same as quizeAverage.length.

8       private int[][] grade; //numberOfStudents rows and numberOfQuizzes columns.
9       private double[] studentAverage;
10      private double[] quizAverage;

11      public GradeBook(int[][] a)
12      {
13          if (a.length == 0 || a[0].length == 0)
14          {
15              System.out.println("Empty grade records. Aborting.");
16              System.exit(0);
17          }

18          numberOfStudents = a.length;
19          numberOfQuizzes = a[0].length;
20          fillGrade(a);
21          fillStudentAverage();
22          fillQuizAverage();
23      }

24      public GradeBook(GradeBook book)
25      {
26          numberOfStudents = book.numberOfStudents;
27          numberOfQuizzes = book.numberOfQuizzes;
28          fillGrade(book.grade);
29          fillStudentAverage();
30          fillQuizAverage();
31      }

32      public GradeBook() throws IOException
33      {
34          BufferedReader keyboard = new BufferedReader(
35                                      new InputStreamReader(System.in));
```

**Display 6.12 A Grade Book Class *(Part 2 of 4)***

```
36          System.out.println("Enter number of students:");
37          numberOfStudents = stringToInt(keyboard.readLine());

38          System.out.println("Enter number of quizzes:");
39          numberOfQuizzes = stringToInt(keyboard.readLine());

40          grade = new int[numberOfStudents][numberOfQuizzes];

41          for (int studentNumber = 1;
42                       studentNumber <= numberOfStudents; studentNumber++)
43              for (int quizNumber = 1;
44                              quizNumber <= numberOfQuizzes; quizNumber++)
45              {
46                  System.out.println("Enter score for student number "
47                                                  + studentNumber);
48                  System.out.println("on quiz number " + quizNumber);
49                  grade[studentNumber - 1][quizNumber - 1] =
50                                              stringToInt(keyboard.readLine());
51              }

52          fillStudentAverage();
53          fillQuizAverage();
54      }

55      private static int stringToInt(String stringObject)
56      {
57          return Integer.parseInt(stringObject.trim());
58      }
```

*This class should have more accessor and mutator methods, but we have omitted them to save space. See Self-Test Exercises 23 through 26.*

```
59      private void fillGrade(int[][] a)
60      {
61          grade = new int[numberOfStudents][numberOfQuizzes];

62          for (int studentNumber = 1;
63                         studentNumber <= numberOfStudents; studentNumber++)
64          {
65              for (int quizNumber = 1;
66                            quizNumber <= numberOfQuizzes; quizNumber++)
67                  grade[studentNumber][quizNumber] =
68                                              a[studentNumber][quizNumber];
69          }
70      }
```

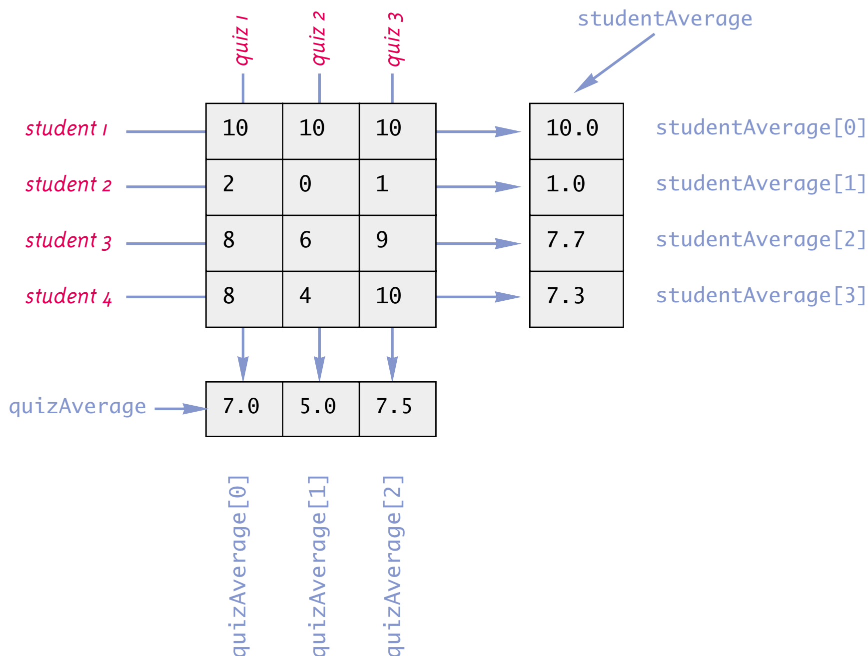
**Display 6.12 A Grade Book Class *(Part 3 of 4)***

```
71      /**
72        Fills the array studentAverage using the data from the array grade.
73      */
74      private void fillStudentAverage()
75      {
76          studentAverage = new double[numberOfStudents];

77          for (int studentNumber = 1;
78                          studentNumber <= numberOfStudents; studentNumber++)
79          {//Process one studentNumber:
80              double sum = 0;
81              for (int quizNumber = 1;
82                           quizNumber <= numberOfQuizzes; quizNumber++)
83                  sum = sum + grade[studentNumber − 1][quizNumber − 1];
84              //sum contains the sum of the quiz scores for student number studentNumber.
85              studentAverage[studentNumber − 1] = sum/numberOfQuizzes;
86              //Average for student studentNumber is studentAverage[studentNumber − 1]
87          }
88      }

89      /**
90        Fills the array quizAverage using the data from the array grade.
91      */
92      private void fillQuizAverage()
93      {
94          quizAverage = new double[numberOfQuizzes];

95          for (int quizNumber = 1; quizNumber <= numberOfQuizzes; quizNumber++)
96          {//Process one quiz (for all students):
97              double sum = 0;
98              for (int studentNumber = 1;
99                            studentNumber <= numberOfStudents; studentNumber++)
100                 sum = sum + grade[studentNumber − 1][quizNumber − 1];
101             //sum contains the sum of all student scores on quiz number quizNumber.
102             quizAverage[quizNumber − 1] = sum/numberOfStudents;
103             //Average for quiz quizNumber is the value of quizAverage[quizNumber − 1]
104         }
105     }

106     public void display()
107     {
108         for (int studentNumber = 1;
109                     studentNumber <= numberOfStudents; studentNumber++)
```

**Display 6.12 A Grade Book Class *(Part 4 of 4)***

```
110            {//Display for one studentNumber:
111                System.out.print("Student " + studentNumber + " Quizzes: ");
112                for (int quizNumber = 1;
113                                 quizNumber <= numberOfQuizzes; quizNumber++)
114                  System.out.print(grade[studentNumber − 1][quizNumber − 1] + " ");
115                System.out.println(" Ave = " + studentAverage[studentNumber − 1] );
116            }

117            System.out.println("Quiz averages: ");
118            for (int quizNumber = 1; quizNumber <= numberOfQuizzes; quizNumber++)
119                System.out.print("Quiz " + quizNumber
120                                 + " Ave = " + quizAverage[quizNumber − 1] + " ");
121            System.out.println();
122        }
123    }
```

**Display 6.13 The Two-Dimensional Array grade**



**Display 6.14 Demonstration of the Class GradeBook**

```
1  import java.io.IOException;
2
3  public class GradeBookDemo
4  {
5      public static void main(String[] args) throws IOException
6      {
7          GradeBook book = new GradeBook();
8          book.display();
9      }
10 }
```

### Sample Dialogue

```
Enter number of students:
4
Enter number of quizzes:
3
Enter score for student number 1
on quiz number 1
10
Enter score for student number 1
on quiz number 2
10

  <The rest of the input dialog is omitted to save space.>

Student 1 Quizzes: 10 10 10 Ave = 10.0
Student 2 Quizzes: 2 0 1 Ave = 1.0
Student 3 Quizzes: 8 6 9 Ave = 7.66666666667
Student 4 Quizzes: 8 4 10 Ave = 7.33333333333
Quiz averages:
Quiz 1 Ave = 7.0 Quiz 2 Ave = 5.0 Quiz 3 Ave = 7.5
```

24. Write a method definition for a method with the following heading. The method is to be added to the class `GradeBook` in Display 6.12.

```
/**
 Changes the grade for student number studentNumber
 on quiz number quizNumber to newGrade.
*/
public void changeGrade(int studentNumber,
                        int quizNumber, int newGrade))
```

25. Write a method definition for a method with the following heading. The method is to be added to the class GradeBook in Display 6.12.

```
/**
 Returns an array with the average quiz score for each student.
*/
public double[] getStudentAverages()
```

26. Write a method definition for a method with the following heading. The method is to be added to the class GradeBook in Display 6.12.

```
/**
 Returns an array with the average score for each quiz.
*/
public double[] getQuizAverages()
```

## Chapter Summary

- An array can be used to store and manipulate a collection of data that is all of the same type.
- The indexed variables of an array can be used just like any other variables of the base type of the array.
- Arrays are objects that are created with `new` just like the class objects we discussed before this chapter (although there is a slight difference in the syntax used).
- A `for` loop is a good way to step through the elements of an array and perform some program action on each indexed variable.
- The most common programming error made when using arrays is to attempt to access a nonexistent array index. Always check the first and last iterations of a loop that manipulates an array to make sure it does not use an index that is illegally small or illegally large.
- The indexed variables of an array can be used as an argument to be plugged in for a parameter of the array's base type.
- A method can have parameters of an array type. When the method is invoked, an entire array is plugged in for the array parameter.
- A method may return an array as the value returned by the method.
- When using a partially filled array, your program needs an additional variable of type `int` to keep track of how much of the array is being used.
- An instance variable of a class can be of an array type.
- If you need an array with more than one index, you can use a multidimensional array, which is actually an array of arrays.

## Answers to Self-Test Exercises

1. a. `word`

   b. `String`

   c. `5`

   d. `0` through `4` inclusive

   e. any of the following would be correct:
   `word[0]`, `word[1]`, `word[2]`, `word[3]`, `word[4]`

2. a. `10`

   b. `0`

   c. `9`

3. `a, b, c,`

4. 
```
1.1 2.2 3.3
1.1 3.3 3.3
```

5. The `for` loop uses indices `1` through `sampleArray.length`, but the correct indices are `0` through `sampleArray.length − 1`. The last index, `sampleArray.length`, is out of bounds. What was probably intended is the following:

```
int[] sampleArray = new int[10];
for (int index = 0; index < sampleArray.length; index++)
    sampleArray[index] = 3*index;
```

6. The last value of `index` is `a.length – 1`, which is the last index of the array. However, when `index` has the value `a.length – 1`, `a[index + 1]` has an index that is out of bounds since `index + 1` is one more than the largest array index. The `for` loop ending condition should instead be `index < a.length – 1`.

7. 
```
SomeClass.doSomething(number); //Legal.
SomeClass.doSomething(a[2]); //Legal.
SomeClass.doSomething(a[3]); //Illegal. Index out of bounds.
SomeClass.doSomething(a[number]); //Legal.
SomeClass.doSomething(a); //Illegal.
```

8. 
```
public static void oneMore(int[] a)
{
    for (int i = 0; i < a.length; i++)
        a[i] = a[i] + 1;
}
```

9. 
```
public static int outOfOrder(double[] a)
{
    for (int index = 0; index < a.length − 1; index++)
    if (a[index] > a[index + 1])
```

```
            return (index + 1);
        return – 1;
    }
```

10. This method is legal but pointless. When invoked, it has no effect on its argument. The parameter a is a local variable that contains a reference. The reference does indeed get changed to a reference to an array of double the size of the argument, but that reference goes away when the method ends. A method can change the values of the indexed variables of its argument, but it cannot change the reference in the array variable used as an argument.

11.
```
public static double[] halfArray(double[] a)
{
    double[] temp = new double[a.length];
    for (int i = 0; i < a.length; i++)
        temp[i] = a[i]/2.0;
    return temp;
}
```

12. The method will compile and run. However, it will change the values of its array argument. If you want to change the values in the array argument, a `void` method would make more sense. If you want to return an array, you should probably return a new array (as in the version in the previous subsection), not return a changed version of the argument array.

13.
```
/**
 Precondition: numberUsed <= argumentArray.length;
 the first numberUsed indexed variables of argumentArray
 have values.
 Returns an array of length numberUsed whose ith element
 is argumentArray[i] – adjustment.
*/
public static double[] differenceArray(
 double[] argumentArray, int numberUsed, double adjustment)
{
    double[] temp = new double[numberUsed];
    for (int i = 0; i < numberUsed; i++)
        temp[i] = argumentArray[i] – adjustment;
    return temp;
}
```

14. The only changes are to add the method `differenceArray` and to rewrite the method `showDifference` as follows (the complete class definition is in the file `GolfScoresExercise.java` on the accompanying CD):

extra code on CD

```
public static void showDifference(double[] a,
                                          int numberUsed)
{
    double average = computeAverage(a, numberUsed);
    System.out.println("Average of the " + numberUsed
                               + " scores = " + average);
```

```
    double[] difference =
                    differenceArray(a, numberUsed, average);
    System.out.println("The scores are:");
    for (int index = 0; index < numberUsed; index++)
          System.out.println(a[index] +
                             " differs from average by "
                                     + difference[index]);
}
```

15. The main differences are to remove parameters, replace the array name `a` by `score`, and make the method `fillArray` a `void` method. This code is in the file `GolfScoresStatic–Exercise.java` on the accompanying CD.

extra code on CD

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class GolfScoresStaticExercise
{
    public static final int MAX_NUMBER_SCORES = 10;
    private static double[] score =
                      new double[MAX_NUMBER_SCORES];
    private static int numberUsed = 0;

    /**
     Shows differences between each of a list of golf scores
     and their average.
    */
    public static void main(String[] args) throws IOException
    {
        System.out.println(
                  "This program reads golf scores and shows");
        System.out.println(
                  "how much each differs from the average.");

        System.out.println("Enter golf scores:");
        fillArray();
        showDifference();
    }

    /**
     Reads values into the array score.
    */
    public static void fillArray() throws IOException
    {
        System.out.println("Enter up to " + score.length
                      + " nonnegative numbers, one per line.");
```

```
    System.out.println(
       "Mark the end of the list with a negative number.");
    BufferedReader keyboard = new BufferedReader(
                     new InputStreamReader(System.in));

    double next;
    int index = 0;
    next = stringToDouble(keyboard.readLine());
    while ((next >= 0) && (index < score.length))
    {
        score[index] = next;
        index++;
        next = stringToDouble(keyboard.readLine());
       //index is the number of
        //array indexed variables used so far.
    }
   //index is the total number of array indexed variables used.

    if (next >= 0)
         System.out.println("Could only read in "
                       + score.length + " input values.");

    numberUsed = index;
}

/**
 Precondition: numberUsed <= score.length.
               score[0] through score[numberUsed-1] have values.
 Returns the average of numbers ascore[0] through
 score[numberUsed-1].
*/
public static double computeAverage()
{
    double total = 0;
    for (int index = 0; index < numberUsed; index++)
        total = total + score[index];
    if (numberUsed > 0)
    {
        return (total/numberUsed);
    }
    else
    {
        System.out.println(
                "ERROR: Trying to average 0 numbers.");
        System.out.println("computeAverage returns 0.");
        return 0;
    }
}
```

```
    private static double stringToDouble(String stringObject)
    {
        return Double.parseDouble(stringObject.trim());
    }

    /**
     Precondition: numberUsed <= score.length.
     The first numberUsed indexed variables of score have values.
     Postcondition: Gives screen output showing how much each of the
     first numberUsed elements of the array a differ from the average.
    */
    public static void showDifference()
    {
        double average = computeAverage();
        System.out.println("Average of the " + numberUsed
                                        + " scores = " + average);
        System.out.println("The scores are:");
        for (int index = 0; index < numberUsed; index++)
        System.out.println(score[index] +
                                   " differs from average by "
                                 + (score[index] - average));
    }
}
```

16.
```
public void removeAll()
{
     numberUsed = 0;
}
```

17.
```
public void increaseCapacity(int newCapacity)
{
     if (newCapacity > numberUsed)
     {
         maxNumberElements = newCapacity;
         double[] temp = new double[newCapacity];
         for (int i = 0; i < a.length; i++)
             temp[i] = a[i];
         a = temp;
     }//else do nothing.
}
```

18. All you need to do to make your code work for sorting into decreasing order is to replace the < with > in the following line of the definition of `indexOfSmallest`:

```
if (a[index] < min)
```

However, to make your code easy to read, you should also rename the method `indexOfSmallest` to `indexOfLargest`, rename the variable `min` to `max`, and rename the variable `indexOfMin` to `indexOfMax`. You should also rewrite some of the comments to reflect these changes.

19. If an array has a value that occurs more than once and you sort the array using the method `SelectionSort.sort`, then there will be as many copies of the repeated value after the array is sorted as there originally were in the array.

20.
```
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3
```

21. The answer is given along with the answer to exercise 22.

22. This program is on the CD that comes with this book.

extra code on CD

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.util.StringTokenizer;

public class DoubleExerciseAnswer
{
    public static final int ROWS = 4;
    public static final int COLUMNS = 5    ;

    public static void main(String[] args) throws IOException
    {
        //Answer to Self-Test Exercise 21:
        BufferedReader keyboard = new BufferedReader(
                     new InputStreamReader(System.in));
        int[][] a = new int[ROWS][COLUMNS];
        String inputLine;

        System.out.println("Enter " + ROWS*COLUMNS +
                  " numbers, " + COLUMNS + " per line, on "
                          + ROWS + " lines:");
        int row, column;
        for (row = 0; row < ROWS; row++)
        {
            inputLine = keyboard.readLine();
            StringTokenizer numberList =
                                new StringTokenizer(inputLine);
            for (column = 0; column < COLUMNS; column++)
                a[row][column] =
                    Integer.parseInt(numberList.nextToken());
        }

        System.out.println("You entered:");
        echo(a);
    }
```

```
    //Answer to Self-Test Exercise 22:
    public static void echo(int[][] a)
    {
        int row, column;
        for (row = 0; row < a.length; row++)
        {
            for (column = 0; column < a[row].length; column++)
                System.out.print(a[row][column] + " ");
            System.out.println();
        }
    }
}
```

23. If the array indices are out of bounds, then Java will halt the program with an error message, so no other checks on the parameters are needed.

```
/**
 Returns the grade that student numbered studentNumber
 received on quiz number quizNumber.
*/
public int getGrade(int studentNumber, int quizNumber)
{
    return grade[studentNumber][quizNumber];
}
```

24. If the array indices are out of bounds, then Java will halt the program with an error message, so no other checks on the parameters are needed.

```
/**
 Changes the grade for student number studentNumber
 on quiz number quizNumber to newGrade.
*/
public void changeGrade(int studentNumber,
                        int quizNumber, int newGrade)
{
    grade[studentNumber][quizNumber] = newGrade;
}
```

25.
```
/**
 Returns an array with the average quiz score for each student.
*/
public double[] getStudentAverages()
{
    int arraySize = studentAverage.length;
    double[] temp = new double[arraySize];
    for (int i = 0; i < arraySize; i++)
        temp[i] = studentAverage[i];
    return temp;
}
```

26.
```
/**
 Returns an array with the average score for each quiz.
*/
public double[] getQuizAverages()
{
    int arraySize = quizAverage.length;
    double[] temp = new double[arraySize];
    for (int i = 0; i < arraySize; i++)
        temp[i] = quizAverage[i];
    return temp;
}
```

## Programming Projects

1. Write a program that reads in the average monthly rainfall for a city for each month of the year and then reads in the actual monthly rainfall for each of the previous 12 months. The program then prints out a nicely formatted table showing the rainfall for each of the previous 12 months as well as how much above or below average the rainfall was for each month. The average monthly rainfall is given for the months January, February, and so forth, in order. To obtain the actual rainfall for the previous 12 months, the program first asks what the current month is and then asks for the rainfall figures for the previous 12 months. The output should correctly label the months. There are a variety of ways to deal with the month names. One straightforward method is to code the months as integers and then do a conversion to a string for the month name before doing the output. A large `switch` statement is acceptable in an output method. The month input can be handled in any manner you wish so long as it is relatively easy and pleasant for the user. Include a loop that allows the user to repeat this entire calculation until the user requests that the program end.

2. Write a static method called `deleteRepeats` that has a partially filled array of characters as a formal parameter and that deletes all repeated letters from the array. Since a partially filled array requires two arguments, the method will actually have two formal parameters: an array parameter and a formal parameter of type `int` that gives the number of array positions used. When a letter is deleted, the remaining letters are moved one position to fill in the gap. This will create empty positions at the end of the array so that less of the array is used. Since the formal parameter is a partially filled array, a second formal parameter of type `int` will tell how many array positions are filled. This second formal parameter cannot be changed by a Java method, so have the method return the new value for this parameter. For example, consider the following code:

```
char a[10];
a[0] = 'a';
a[1] = 'b';
a[2] = 'a';
```

```
a[3] = 'c';
int size = 4;
size = deleteRepeats(a, size);
```

After this code is executed, the value of `a[0]` is `'a'`, the value of `a[1]` is `'b'`, the value of `a[2]` is `'c'`, and the value of `size` is `3`. (The value of `a[3]` is no longer of any concern, since the partially filled array no longer uses this indexed variable.) You may assume that the partially filled array contains only lowercase letters. Write a suitable test program for your method.

3. The standard deviation of a list of numbers is a measure of how much the numbers deviate from the average. If the standard deviation is small, the numbers are clustered close to the average. If the standard deviation is large, the numbers are scattered far from the average. The standard deviation of a list of numbers $n_1$, $n_2$, $n_3$, and so forth is defined as the square root of the average of the following numbers:

   $(n_1 - a)^2$, $(n_2 - a)^2$, $(n_3 - a)^2$, and so forth.

   The number $a$ is the average of the numbers $n_1$, $n_2$, $n_3$, and so forth.

   Define a static method that takes a partially filled array of numbers as its argument and returns the standard deviation of the numbers in the partially filled array. Since a partially filled array requires two arguments, the method will actually have two formal parameters, an array parameter and a formal parameter of type `int` that gives the number of array positions used. The numbers in the array will be of type `double`. Write a suitable test program for your method.

4. Write a program that reads numbers from the keyboard into an array of type `int[]`. You may assume that there will be 50 or fewer entries in the array. Your program allows any number of numbers to be entered up to 50 numbers. The output is to be a two-column list. The first column is a list of the distinct array elements; the second column is the count of the number of occurrences of each element. The list should be sorted on entries in the first column, largest to smallest.

   For the array

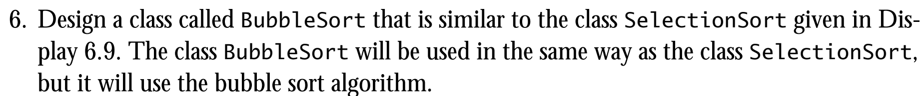
   ```
   –12 3 –12 4 1 1 –12 1 –1 1 2 3 4 2 3 –12
   ```

   the output should be

   ```
    N   Count
    4   2
    3   3
    2   2
    1   4
   –1   1
   –12  4
   ```

5. An array can be used to store large integers one digit at a time. For example, the integer `1234` could be stored in the array `a` by setting `a[0]` to `1`, `a[1]` to `2`, `a[2]` to `3`, and `a[3]` to

4. However, for this exercise you might find it more useful to store the digits backward; that is, place 4 in a[0], 3 in a[1], 2 in a[2], and 1 in a[3]. In this exercise you will write a program that reads in two positive integers that are 20 or fewer digits in length and then outputs the sum of the two numbers. Your program will read the digits as values of type char so that the number 1234 is read as the four characters '1', '2', '3', and '4'. After they are read into the program, the characters are changed to values of type int. The digits will be read into a partially filled array, and you might find it useful to reverse the order of the elements in the array after the array is filled with data from the keyboard. (Whether or not you reverse the order of the elements in the array is up to you. It can be done either way and each way has its advantages and disadvantages.) Your program will perform the addition by implementing the usual paper-and-pencil addition algorithm. The result of the addition is stored in an array of size 20 and the result is then written to the screen. If the result of the addition is an integer with more than the maximum number of digits (that is, more than 20 digits), then your program should issue a message saying that it has encountered "integer overflow." You should be able to change the maximum length of the integers by changing only one named constant. Include a loop that allows the user to continue to do more additions until the user says the program should end.



6. Design a class called BubbleSort that is similar to the class SelectionSort given in Display 6.9. The class BubbleSort will be used in the same way as the class SelectionSort, but it will use the bubble sort algorithm.

   The bubble sort algorithm checks all adjacent pairs of elements in the array from the beginning to the end and interchanges any two elements that are out of order. This process is repeated until the array is sorted. The algorithm is as follows:

   **Bubble Sort Algorithm to Sort an Array a**

   Repeat the following until the array a is sorted:

```
for (index = 0; index < a.length - 1; index++)
     if (a[index] > a[index + 1])
          Interchange the values of a[index] and a[index + 1].
```

   The bubble sort algorithm is good for sorting an array that is "almost sorted." It is not competitive to other sorting methods for most other situations.

7. Enhance the definition of the class PartiallyFilledArray (Display 6.5) in the following way: When the user attempts to add one additional element and there is no room in the array instance variable a, the user is allowed to add the element. The object creates a second array that is twice the size of the array a, copies values from the array a to the user's new array, makes this array (or more precisely its reference) the new value of a, and then adds the element to this new larger array a. Hence, this new class will have no limit (other than the physical size of the computer) to how many numbers it can hold. The instance variable maxNumberElements remains and the method getMaxCapacity is unchanged, but these now refer to the currently allocated memory and not to an absolute upper bound. Write a suitable test program.

8. Write a program that will allow two users to play tic-tac-toe. The program should ask for moves alternately from player X and player O. The program displays the game positions as follows:

```
1  2  3
4  5  6
7  8  9
```

The players enter their moves by entering the position number they wish to mark. After each move, the program displays the changed board. A sample board configuration is

```
X  X  O
4  5  6
O  8  9
```

9. Write a program to assign passengers seats in an airplane. Assume a small airplane with seat numberings as follows:

```
1   A B  C D
2   A B  C D
3   A B  C D
4   A B  C D
5   A B  C D
6   A B  C D
7   A B  C D
```

The program should display the seat pattern, with an `'X'` marking the seats already assigned. For example, after seats 1A, 2B, and 4C are taken, the display should look like:

```
1   X B  C D
2   A X  C D
3   A B  C D
4   A B  X D
5   A B  C D
6   A B  C D
7   A B  C D
```

After displaying the seats available, the program prompts for the seat desired, the user types in a seat, and then the display of available seats is updated. This continues until all seats are filled or until the user signals that the program should end. If the user types in a seat that is already assigned, the program should say that that seat is occupied and ask for another choice.