Chapter 7

Arrays

Introduction

The sequential nature of files severely limits the number of interesting things that you can do easily with them. The algorithms we have examined so far have all been sequential algorithms: algorithms that can be performed by examining each data item once, in sequence. An entirely different class of algorithms can be performed when you can access the data items multiple times and in an arbitrary order.

This chapter examines a new object called an array that provides this more flexible kind of access. The concept of arrays is not complex, but it can take a while for a novice to learn all of the different ways that an array can be used. The chapter begins with a general discussion of arrays and then moves into a discussion of common array manipulations as well as advanced array techniques. The chapter also includes a discussion of special rules known as reference semantics that apply only to objects like arrays and strings.

7.1 Array Basics

An *array* is a flexible structure for storing a sequence of values that are all of the same type.

Array

An indexed structure that holds multiple values of the same type.

The values stored in an array are called *elements*. The individual elements are accessed using an integer *index*.

Index

An integer indicating the position of a particular value in a data structure.

As an analogy, consider post office boxes. The boxes are indexed with numbers, so you can refer to an individual box by using a description like “P.O. Box 884.” You already have experience using an index to indicate positions within a `String`; recall the methods `charAt` and `substring`. Like `String` indexes, array indexes start with 0. This is a convention known as *zero-based indexing*.

Zero-Based Indexing

A numbering scheme used throughout Java in which a sequence of values is indexed starting with 0 (element 0, element 1, element 2, and so on).

It might seem more natural to start indexes with 1 instead of 0, but Java uses the same indexing scheme that is used in C and C++.

Constructing and Traversing an Array

Suppose you want to store some different temperature readings. You could keep them in a series of variables:

```
double temperature1;
double temperature2;
double temperature3;
```

This isn’t a bad solution if you have just 3 temperatures, but suppose you need to store 3000 temperatures. Then you would want a more flexible way to store the values. You can instead store the temperatures in an array.

When you use an array, you first need to declare a variable for it, so you have to know what type to use. The type will depend on the type of elements you want to have in your array. To indicate that you are creating an array, follow the type name with a set of square brackets: `[]`. If you are storing temperature values, you want a sequence of values of type `double`, so you use the type `double[]`. Thus, you can declare a variable for storing your array as follows:

```
double[] temperature;
```

Arrays are objects, which means that they must be constructed. Simply declaring a variable isn't enough to bring the object into existence. In this case you want an array of three `double` values, which you can construct as follows:

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

This is a slightly different syntax than you've used previously to create a new object. It is a special syntax for arrays only. Notice that on the left-hand side you don't put anything inside the square brackets, because you're describing a type. The variable `temperature` can refer to any array of `double` values, no matter how many elements it has. On the right-hand side, however, you have to mention a specific number of elements because you are asking Java to construct an actual array object and it needs to know how many elements to include.

The general syntax for declaring and constructing an array is as follows:

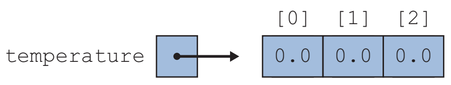
```
<element type>[] <name> = new <element type>[<length>];
```

You can use any type as the element type, although the left and right sides of this statement have to match. For example, any of the following lines of code would be legal ways to construct an array:

```
int[] numbers = new int[10];       // an array of 10 ints
char[] letters = new char[20];     // an array of 20 chars
boolean[] flags = new boolean[5];  // an array of 5 booleans
String[] names = new String[100];  // an array of 100 Strings
Color[] colors = new Color[50];    // an array of 50 Colors
```

Some special rules apply when you construct an array of objects such as an array of `Strings` or an array of `Colors`, but we'll discuss those later in the chapter.

When it executes the line of code to construct the array of temperatures, Java will construct an array of three `double` values, and the variable `temperature` will refer to the array:



As you can see, the variable `temperature` is not itself the array. Instead, it stores a reference to the array. The array indexes are indicated in square brackets. To refer to an individual element of the array, you combine the name of the variable that refers

Table 7.1 Zero-Equivalent Auto-Initialization Values

Type	Value
`int`	`0`
`double`	`0.0`
`char`	`'\0'`
`boolean`	`false`
objects	`null`

to the array (`temperature`) with a specific index (`[0]`, `[1]`, or `[2]`). So, there is an element known as `temperature[0]`, an element known as `temperature[1]`, and an element known as `temperature[2]`.

In the `temperature` array diagram, each of the array elements has the value `0.0`. This is a guaranteed outcome when an array is constructed. Each element is initialized to a default value, a process known as *auto-initialization.*

Auto-Initialization

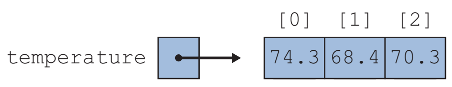
The initialization of variables to a default value, such as on an array's elements when it is constructed.

When Java performs auto-initialization, it always initializes to the zero-equivalent for the type. Table 7.1 indicates the zero-equivalent values for various types. The special value `null` will be explained later in this chapter.

Notice that the zero-equivalent for type `double` is `0.0`, which is why the array elements were initialized to that value. Using the indexes, you can store the specific temperature values that are relevant to this problem:

```
temperature[0] = 74.3;
temperature[1] = 68.4;
temperature[2] = 70.3;
```

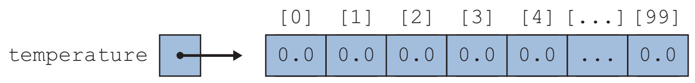
This code modifies the array to have the following values:



Obviously an array isn't particularly helpful when you have just three values to store, but you can request a much larger array. For example, you could request an array of 100 temperatures by writing the following line of code:

```
double[] temperature = new double[100];
```

This is almost the same line of code you executed before. The variable is still declared to be of type `double[]`, but in constructing the array, you requested 100 elements instead of 3, which constructs a much larger array:



Notice that the highest index is 99 rather than 100 because of zero-based indexing.

You are not restricted to using simple literal values inside the brackets. You can use any integer expression. This flexibility allows you to combine arrays with loops, which greatly simplifies the code you write. For example, suppose you want to read a series of temperatures from a `Scanner`. You could read each value individually:

```
temperature[0] = input.nextDouble();
temperature[1] = input.nextDouble();
temperature[2] = input.nextDouble();
...
temperature[99] = input.nextDouble();
```

But since the only thing that changes from one statement to the next is the index, you can capture this pattern in a `for` loop with a control variable that takes on the values `0` to `99`:

```
for (int i = 0; i < 100; i++) {
    temperature[i] = input.nextDouble();
}
```

This is a very concise way to initialize all the elements of the array. The preceding code works when the array has a length of 100, but you can change this to accommodate an array of a different length. Java provides a useful mechanism for making this code more general. Each array keeps track of its own length. You're using the variable `temperature` to refer to your array, which means you can ask for `temperature.length` to find out the length of the array. By using `temperature.length` in the `for` loop test instead of the specific value 100, you make your code more general:

```
for (int i = 0; i < temperature.length; i++) {
    temperature[i] = input.nextDouble();
}
```

Notice that the array convention is different from the `String` convention. When you are working with a `String` variable `s`, you ask for the length of the `String` by referring to `s.length()`. When you are working with an array variable, you don't include the parentheses after the word "length." This is another one of those unfortunate inconsistencies that Java programmers just have to memorize.

The previous code provides a pattern that you will see often with array-processing code: a `for` loop that starts at 0 and that continues while the loop variable is less than the length of the array, doing something with element `[i]` in the body of the loop. The program goes through each array element sequentially, which we refer to as *traversing* the array.

Array Traversal

Processing each array element sequentially from the first to the last.

This pattern is so useful that it is worth including it in a more general form:

```
for (int i = 0; i < <array>.length; i++) {
    <do something with array[i]>;
}
```

We will see this traversal pattern repeatedly as we explore common array algorithms.

Accessing an Array

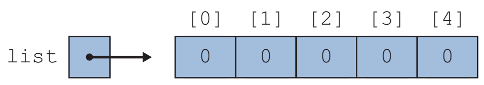
As we discussed in the last section, we refer to array elements by combining the name of the variable that refers to the array with an integer index inside square brackets:

```
<array variable>[<integer expression>]
```

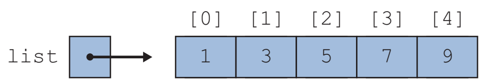
Notice in this syntax description that the index can be an arbitrary integer expression. To explore this feature, let's examine how we would access particular values in an array of integers. Suppose that we construct an array of length 5 and fill it up with the first five odd integers:

```
int[] list = new int[5];
for (int i = 0; i < list.length; i++) {
    list[i] = 2 * i + 1;
}
```

The first line of code declares a variable `list` of type `int[]` that refers to an array of length 5. The array elements are auto-initialized to `0`:



Then the code uses the standard traversing loop to fill in the array with successive odd numbers:



Suppose that we want to report the first, middle, and last values in the list. From an examination of the preceding diagram, we can see that these values occur at indexes 0, 2, and 4, which means we could write the following code:

```
// works only for an array of length 5
System.out.println("first = " + list[0]);
System.out.println("middle = " + list[2]);
System.out.println("last = " + list[4]);
```

This technique works when the array is of length 5, but suppose that we use an array of a different length? If the array has a length of 10, for example, this code will report the wrong values. We need to modify it to incorporate `list.length`, just as we modified the standard traversing loop.

The first element of the array will always be at index 0, so the first line of code doesn't need to change. You might at first think that we could fix the third line of code by replacing the `4` with `list.length`:

```
// doesn't work
System.out.println("last = " + list[list.length]);
```

However, this code doesn't work. The culprit is zero-based indexing. In our example, the last value is stored at index 4, not index 5, when `list.length` is 5. More generally, the last value will be at index `list.length – 1`. We can use this expression directly in our `println` statement:

```
// this one works
System.out.println("last = " + list[list.length – 1]);
```

Notice that what appears inside the square brackets is an integer expression (the result of subtracting 1 from `list.length`).

A simple approach to finding the middle value is to divide the length of the list in half:

```
// is this right?
System.out.println("middle = " + list[list.length / 2]);
```

When `list.length` is 5, this expression evaluates to `2`, which prints the correct value. But what about when `list.length` is 10? In that case the expression evaluates to `5`, and we would print `list[5]`. But when the list has an even length, there are actually two values in the middle. For a list of length 10, the two values are at `list[4]` and `list[5]`. In general, the preceding expression always returns the second of the two values in the middle when the list is of even length.

If we wanted the code to return the first of the two values in the middle instead, we could subtract 1 from the length before dividing it in half. Here is a complete set of `println` statements that follows this approach:

```
System.out.println("first = " + list[0]);
System.out.println("middle = " + list[(list.length - 1) / 2]);
System.out.println("last = " + list[list.length - 1]);
```

As you learn how to use arrays, you will find yourself wondering what types of operations you can perform on an array element that you are accessing. For example, for the array of integers called `list`, what exactly can you do with `list[i]`? The answer is that you can do anything with `list[i]` that you would normally do with any variable of type `int`. For example, if you have a variable called `x` of type `int`, any of the following expressions are valid:

```
x = 3;
x++;
x *= 2;
x--;
```

That means that the same expressions are valid for `list[i]` if `list` is an array of integers:

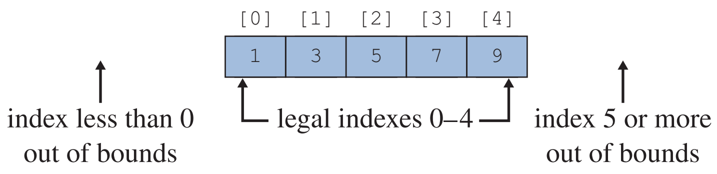
```
list[i] = 3;
list[i]++;
list[i] *= 2;
list[i]--;
```

From Java's point of view, because `list` is declared to be of type `int[]`, an array element like `list[i]` is of type `int` and can be manipulated as such. For example, to increment every value in the array, you could use the standard traversing loop:

```
for (int i = 0; i < list.length; i++) {
    list[i]++;
}
```

This code would increment each value in the array, turning the array of odd numbers into an array of even numbers.

It is possible to refer to an illegal index of an array, in which case Java throws an exception. For example, for an array of length 5, the legal indexes are from 0 to 4. Any number less than 0 or greater than 4 is outside the bounds of the array:


[0] [1] [2] [3] [4]
1 3 5 7 9
index less than 0 out of bounds
legal indexes 0–4
index 5 or more out of bounds

When you are working with this sample array, if you attempt to refer to `list[-1]` or `list[5]`, you are attempting to access an array element that does not exist. If your code makes such an illegal reference, Java will halt your program with an `ArrayIndexOutOfBoundsException`.

Initializing Arrays

Java has a special syntax for initializing an array when you know exactly what you want to put into it. For example, you could write the following code to initialize an array of integers to keep track of the number of days that are in each month ("Thirty days hath September . . .") and an array of `Strings` to keep track of the abbreviations for the days of the week:

```
int[] daysIn = new int[12];
daysIn[0] = 31;
daysIn[1] = 28;
daysIn[2] = 31;
daysIn[3] = 30;
daysIn[4] = 31;
daysIn[5] = 30;
daysIn[6] = 31;
daysIn[7] = 31;
daysIn[8] = 30;
daysIn[9] = 31;
daysIn[10] = 30;
daysIn[11] = 31;
String[] dayNames = new String[7];
dayNames[0] = "Mon";
dayNames[1] = "Tue";
dayNames[2] = "Wed";
dayNames[3] = "Thu";
dayNames[4] = "Fri";
dayNames[5] = "Sat";
dayNames[6] = "Sun";
```

This code works, but it's a rather tedious way to declare these arrays. Java provides a shorthand:

```
int[] daysIn = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
String[] dayNames = {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"};
```

The general syntax for array initialization is as follows:

```
<element type>[] <name> = {<value>, <value>, ..., <value>};
```

You use the curly braces to enclose a series of values that will be stored in the array. The order of the values is important. The first value will go into index 0, the second value will go into index 1, and so on. Java counts how many values you include and constructs an array that is just the right size. It then stores the various values into the appropriate spots in the array.

This is one of only two examples we have seen in which Java will construct an object without the `new` keyword. The other place we saw this was with `String` literals, in which Java constructs `String` objects without your having to call `new`. Both of these techniques are conveniences for programmers. These tasks are so common that the designers of the language wanted to make it easy to do them.

Declaring and manipulating arrays in JShell is a good way to try out the syntax and learn about how arrays behave. The concise array initialization syntax makes it easier to create and examine an array containing a given set of elements.

```
jshell> int[] list = {1, 3, 5, 7, 9};
list ==> int[5] { 1, 3, 5, 7, 9 }

jshell> list[0]
$2 ==> 1

jshell> list[4]
$3 ==> 9

jshell> list.length
$4 ==> 5

jshell> list[list.length - 1]
$5 ==> 9

jshell> list[list.length / 2]
$6 ==> 5

jshell> list[5]
|  java.lang.ArrayIndexOutOfBoundsException thrown: 5
|        at (#7:1)
```

A Complete Array Program

Let’s look at a program in which an array allows you to solve a problem that you couldn’t solve before. If you tune in to any local news broadcast at night, you’ll hear them report the high temperature for that day. It is usually reported as an integer, as in, “It got up to 78 today.”

Suppose you want to examine a series of daily high temperatures, compute the average high temperature, and count how many days were above that average temperature. You've been using `Scanners` to solve problems like this, and you can almost solve the problem that way. If you just wanted to know the average, you could use a `Scanner` and write a cumulative sum loop to find it:

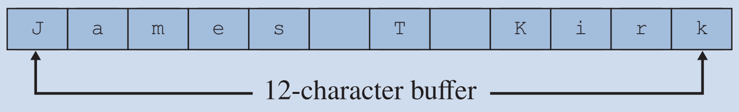
```java
// Reads a series of high temperatures and reports the average.

import java.util.*;

public class Temperature1 {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        System.out.print("How many days' temperatures? ");
        int numDays = console.nextInt();
        int sum = 0;
        for (int i = 1; i <= numDays; i++) {
            System.out.print("Day " + i + "'s high temp: ");
            int next = console.nextInt();
            sum += next;
        }
        double average = (double) sum / numDays;
        System.out.println();
        System.out.println("Average = " + average);
    }
}
```

Did You Know?

Buffer Overruns

One of the earliest and still most common sources of computer security problems is a *buffer overrun* (also known as a *buffer overflow*). A buffer overrun is similar to an array index out of bounds exception. It occurs when a program writes data beyond the bounds of the buffer that is set aside for that data.

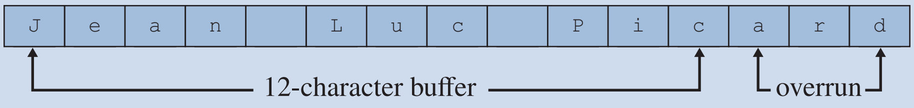
For example, you might have space allocated for the `String` "James T Kirk", which is 12 characters long, counting the spaces:



Continued on next page

Continued from previous page

Suppose that you tell the computer to overwrite this buffer with the `String` "Jean Luc Picard". There are 15 letters in Picard's name, so if you write all of those characters into the buffer, you "overrun" it by writing three extra characters:



The last three letters of Picard's name ("ard") are being written to a part of memory that is beyond the end of the buffer. This is a very dangerous situation, because it will overwrite any data that is already there. An analogy would be a fellow student grabbing three sheets of paper from you and erasing anything you had written on them. You are likely to have had useful information written on those sheets of paper, so the overrun is likely to cause a problem.

When a buffer overrun happens accidentally, the program usually halts with some kind of error condition. However, buffer overruns are particularly dangerous when they are done on purpose by a malicious program. If the attacker can figure out just the right memory location to overwrite, the attacking software can take over your computer and instruct it to do things you haven't asked it to do.

Three of the most famous Internet worms were built on buffer overruns: the 1988 Morris worm, the 2001 Code Red worm, and the 2003 SQL Slammer worm.

Buffer overruns are often written as array code. You might wonder how such a malicious program could be written if the computer checks the bounds when you access an array. The answer is that older programming languages like C and C++ do not check bounds when you access an array. By the time Java was designed in the early 1990s, the danger of buffer overruns was clear and the designers of the language decided to include array-bounds checking so that Java would be more secure. Microsoft included similar bounds checking when it designed the language C# in the late 1990s.

The preceding program does a pretty good job. Here is a sample execution:

```
How many days' temperatures? 5
Day 1's high temp: 78
Day 2's high temp: 81
Day 3's high temp: 75
Day 4's high temp: 79
Day 5's high temp: 71
Average = 76.8
```

But how do you count how many days were above average? You could try to incorporate a comparison to the average temperature into the loop, but that won't work. The problem is that you can't figure out the average until you've gone through all of the data. That means you'll need to make a second pass through the data to figure out how many days were above average. You can't do that with a `Scanner`, because a `Scanner` has no "reset" option that allows you to see the data a second time. You'd have to prompt the user to enter the temperature data a second time, which would be silly.

Fortunately, you can solve the problem with an array. As you read numbers in and compute the cumulative sum, you can fill up an array that stores the temperatures. Then you can use the array to make the second pass through the data.

In the previous temperature example you used an array of `double` values, but here you want an array of `int` values. So, instead of declaring a variable of type `double[]`, declare a variable of type `int[]`. You're asking the user how many days of temperature data to include, so you can construct the array right after you've read that information:

```
int numDays = console.nextInt();
int[] temps = new int[numDays];
```

Here is the old loop:

```
for (int i = 1; i <= numDays; i++) {
    System.out.print("Day " + i + "'s high temp: ");
    int next = console.nextInt();
    sum += next;
}
```

Because you're using an array, you'll want to change this to a loop that starts at 0 to match the array indexing. But just because you're using zero-based indexing inside the program doesn't mean that you have to confuse the user by asking for "Day 0's high temp." You can modify the `println` to prompt for day `(i + 1)`. Furthermore, you no longer need the variable `next` because you'll be storing the values in the array instead. So, the loop code becomes

```
for (int i = 0; i < numDays; i++) {
    System.out.print("Day " + (i + 1) + "'s high temp: ");
    temps[i] = console.nextInt();
    sum += temps[i];
}
```

Notice that you're now testing whether the index is strictly less than `numDays`. After this loop executes, you compute the average as we did before. Then you write a new loop that counts how many days were above average using our standard traversing loop:

```
int above = 0;
for (int i = 0; i < temps.length; i++) {
    if (temps[i] > average) {
        above++;
    }
}
```

In this loop the test involves `temps.length`. You could instead have tested whether the variable is less than `numDays`; either choice works in this program because they should be equal to each other.

If you put these various code fragments together and include code to report the number of days that had an above-average temperature, you get the following complete program:

```
// Reads a series of high temperatures and reports the
// average and the number of days above average.

import java.util.*;

public class Temperature2 {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        System.out.print("How many days' temperatures? ");
        int numDays = console.nextInt();
        int[] temps = new int[numDays];

        // record temperatures and find average
        int sum = 0;
        for (int i = 0; i < numDays; i++) {
            System.out.print("Day " + (i + 1) + "'s high temp: ");
            temps[i] = console.nextInt();
            sum += temps[i];
        }
        double average = (double) sum / numDays;

        // count days above average
        int above = 0;
        for (int i = 0; i < temps.length; i++) {
            if (temps[i] > average) {
                above++;
            }
        }

        // report results
        System.out.println();
```

```
            System.out.println("Average = " + average);
            System.out.println(above + " days above average");
        }
    }
```

Here is a sample execution of the program:

```
How many days' temperatures? 9
Day 1's high temp: 75
Day 2's high temp: 78
Day 3's high temp: 85
Day 4's high temp: 71
Day 5's high temp: 69
Day 6's high temp: 82
Day 7's high temp: 74
Day 8's high temp: 80
Day 9's high temp: 87

Average = 77.88888888888889
5 days above average
```

Random Access

Most of the algorithms we have seen so far have involved *sequential access*.

Sequential Access

Manipulating values in a sequential manner from first to last.

A `Scanner` object is often all you need for a sequential algorithm, because it allows you to access data by moving forward from the first element to the last. But as we have seen, there is no way to reset a `Scanner` back to the beginning. The sample program we just studied uses an array to allow a second pass through the data, but even this is fundamentally a sequential approach because it involves two forward passes through the data.

An array is a powerful data structure that allows a more flexible kind of access known as *random access:*

Random Access

Manipulating values in any order whatsoever to allow quick access to each value.

An array can provide random access because it is allocated as a contiguous block of memory. The computer can quickly compute exactly where a particular value will be stored, because it knows how much space each element takes up in memory and it knows that all the elements are allocated right next to one another in the array.

When you work with arrays, you can jump around in the array without worrying about how much time it will take. For example, suppose that you have constructed an array of temperature readings that has 10,000 elements and you find yourself wanting to print a particular subset of the readings with code like the following:

```
System.out.println("#1394 = " + temps[1394]);
System.out.println("#6793 = " + temps[6793]);
System.out.println("#72 = " + temps[72]);
```

This code will execute quickly even though you are asking for array elements that are far apart from one another. Notice also that you don't have to ask for them in order. You can jump to element 1394, then jump ahead to element 6793, and then jump back to element 72. You can access elements in an array in any order that you like, and you will get fast access.

Later in the chapter we will explore several algorithms that would be difficult to implement without fast random access.

Common Programming Error

Off-by-One Bug

When you converted the `Temperature1` program to one that uses an array, you modified the `for` loop to start with an index of 0 instead of 1. The original `for` loop was written the following way:

```
for (int i = 1; i <= numDays; i++) {
    System.out.print("Day " + i + "'s high temp: ");
    int next = console.nextInt();
    sum += next;
}
```

Because you were storing the values into an array rather than reading them into a variable called `next`, you replaced `next` with `temps[i]`:

```
// wrong loop bounds
for (int i = 1; i <= numDays; i++) {
    System.out.print("Day " + i + "'s high temp: ");
    temps[i] = console.nextInt();
    sum += temps[i];
}
```

Continued on next page

Continued from previous page

Because the array is indexed starting at 0, you changed the bounds of the `for` loop to start at `0` and adjusted the `print` statement. Suppose those were the only changes you made:

```
// still wrong loop bounds
for (int i = 0; i <= numDays; i++) {
    System.out.print("Day " + (i + 1) + "'s high temp: ");
    temps[i] = console.nextInt();
    sum += temps[i];
}
```

This loop generates an error when you run the program. The loop asks for an extra day's worth of data and then throws an exception. Here's a sample execution:

```
How many days' temperatures? 5
Day 1's high temp: 82
Day 2's high temp: 80
Day 3's high temp: 79
Day 4's high temp: 71
Day 5's high temp: 75
Day 6's high temp: 83
Exception in thread "main"
    java.lang.ArrayIndexOutOfBoundsException: 5
        at Temperature2.main(Temperature2.java:18)
```

The problem is that if you're going to start the `for` loop variable at `0`, you need to do a test to ensure that it is strictly less than the number of iterations you want. You changed the `1` to a `0` but left the <= test. As a result, the loop is performing an extra iteration and trying to make a reference to an array element `temps[5]` that doesn't exist.

This is a classic off-by-one error. The fix is to change the loop bounds to use a strictly less-than test:

```
// correct bounds
for (int i = 0; i < numDays; i++) {
    System.out.print("Day " + (i + 1) + "'s high temp: ");
    temps[i] = console.nextInt();
    sum += temps[i];
}
```

Arrays and Methods

VideoNote

You will find that when you pass an array as a parameter to a method, the method has the ability to change the contents of the array. We'll examine in detail later in the chapter why this occurs, but for now, the important point is simply to understand that methods can alter the contents of arrays that are passed to them as parameters.

Let's explore a specific example to better understand how to use arrays as parameters and return values for a method. Earlier in the chapter, we saw the following code for constructing an array of odd numbers and incrementing each array element:

```
int[] list = new int[5];
for (int i = 0; i < list.length; i++) {
    list[i] = 2 * i + 1;
}
for (int i = 0; i < list.length; i++) {
    list[i]++;
}
```

Let's see what happens when we move the incrementing loop into a method. It will need to take the array as a parameter. We'll rename it `data` instead of `list` to make it easier to distinguish it from the original array variable. Remember that the array is of type `int[]`, so we would write the method as follows:

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

You might think this method will have no effect whatsoever, or that we have to return the array to cause the change to be remembered. But when we use an array as a parameter, this approach actually works. We can replace the incrementing loop in the original code with a call on our method:

```
int[] list = new int[5];
for (int i = 0; i < list.length; i++) {
    list[i] = 2 * i + 1;
}
incrementAll(list);
```

This code produces the same result as the original.

The key lesson to draw from this is that when we pass an array as a parameter to a method, that method has the ability to change the contents of the array. We don't need to return the array to allow this to happen.

To continue with this example, let's define a method for the initializing code that fills the array with odd numbers. We can accomplish this by moving the initializing loop into a method that takes the array as a parameter:

```
public static void fillWithOdds(int[] data) {
    for (int i = 0; i < data.length; i++) {
        data[i] = 2 * i + 1;
    }
}
```

We would then change our `main` method to call this `fillWithOdds` method:

```
int[] list = new int[5];
fillWithOdds(list);
incrementAll(list);
```

Like the `incrementAll` method, this method would change the array even though it does not return it. But this isn't the best approach to use in this situation. It seems odd that the `fillWithOdds` method requires you to construct an array and pass it as a parameter. Why doesn't `fillWithOdds` construct the array itself? That would simplify the call to the method, particularly if we ended up calling it multiple times.

If `fillWithOdds` is going to construct the array, it will have to return a reference to it. Otherwise, only the method will have a reference to the newly constructed array. In its current form, the `fillWithOdds` method assumes that the array has already been constructed, which is why we wrote the following two lines of code in `main`:

```
int[] list = new int[5];
fillWithOdds(list);
```

If the method is going to construct the array, it doesn't have to be passed as a parameter, but it will have to be returned by the method. Thus, we can rewrite these two lines of code from `main` as a single line:

```
int[] list = fillWithOdds();
```

Now, however, we have a misleading method name. The method isn't just filling an existing array, it is constructing one. Also notice that we can make the method more flexible by telling it how large to make the array. So if we rename it and pass the size as a parameter, then we'd call it this way:

```
int[] list = buildOddArray(5);
```

We can then rewrite the `fillWithOdds` method so that it constructs and returns the array:

```
public static int[] buildOddArray(int size) {
    int[] data = new int[size];
    for (int i = 0; i < data.length; i++) {
        data[i] = 2 * i + 1;
    }
    return data;
}
```

Pay close attention to the header of the preceding method. It no longer has the array as a parameter, and its return type is `int[]` rather than `void`. It also ends with a `return` statement that returns a reference to the array that it constructs.

Putting this all together along with some code to print the contents of the array, we end up with the following complete program:

```
// Sample program with arrays passed as parameters

public class IncrementOdds {
    public static void main(String[] args) {
        int[] list = buildOddArray(5);
        incrementAll(list);
        for (int i = 0; i < list.length; i++) {
            System.out.print(list[i] + " ");
        }
        System.out.println();
    }

    // returns array of given size composed of consecutive odds
    public static int[] buildOddArray(int size) {
        int[] data = new int[size];
        for (int i = 0; i < data.length; i++) {
            data[i] = 2 * i + 1;
        }
        return data;
    }

    // adds one to each array element
    public static void incrementAll(int[] data) {
        for (int i = 0; i < data.length; i++) {
            data[i]++;
        }
    }
}
```

The program produces the following output:

```
2 4 6 8 10
```

The For-Each Loop

Java has a loop construct that simplifies certain array loops. It is known as the enhanced `for` loop, or the for-each loop. You can use it whenever you want to examine each value in an array. For example, the program `Temperature2` had an array variable called `temps` and the following loop:

```
for (int i = 0; i < temps.length; i++) {
    if (temps[i] > average) {
        above++;
    }
}
```

We can rewrite this as a for-each loop:

```
for (int n : temps) {
    if (n > average) {
        above++;
    }
}
```

This loop is normally read as, "For each `int n` in `temps`. . . ." The basic syntax of the for-each loop is

```
for (<type> <name> : <array>) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

There is nothing special about the variable name, as long as you keep it consistent within the body of the loop. For example, the previous loop could be written with the variable `x` instead of the variable `n`:

```
for (int x : temps) {
    if (x > average) {
        above++;
    }
}
```

The for-each loop is most useful when you simply want to examine each value in sequence. There are many situations in which a for-each loop is not appropriate. For example, the following loop would double every value in an array called `list`:

```
for (int i = 0; i < list.length; i++) {
    list[i] *= 2;
}
```

Because the loop is changing the array, you can't replace it with a for-each loop:

```
for (int n : list) {
    n *= 2; // changes only n, not the array
}
```

As the comment indicates, the preceding loop doubles the variable `n` without changing the array elements.

In some cases, the for-each loop isn't the most convenient choice even when the code involves examining each array element in sequence. Consider, for example, the following loop that prints each array index along with the array value separated by a tab character:

```
for (int i = 0; i < data.length; i++) {
    System.out.println(i + "\t" + data[i]);
}
```

A for-each loop could be used to replace the array access:

```
for (int n : data) {
    System.out.println(i + "\t" + n); // not quite legal
}
```

However, this loop would cause a problem. We want to print the value of `i`, but we eliminated `i` when we converted the array access to a for-each loop. We would have to add extra code to keep track of the value of `i`:

```
// legal but clumsy
int i = 0;
for (int n : data) {
    System.out.println(i + "\t" + n);
    i++;
}
```

In this case, the for-each loop doesn't really simplify things, and the original version is probably clearer.

The `Arrays` Class

Arrays have some important limitations that you should understand. Over the years Java has attempted to remedy these limitations by providing various utility methods in a class called `Arrays`. This class provides many methods that make it easier to work with arrays. The `Arrays` class is part of the `java.util` package, so you would have to include an `import` declaration in any program that uses it.

The first limitation you should be aware of is that you can't change the size of an array in the middle of program execution. Remember that arrays are allocated as a contiguous block of memory, so it is not easy for the computer to expand the array. If you find that you need a larger array, you should construct a new array and copy the values from the old array to the new array. The method `Arrays.copyOf` provides exactly this functionality. For example, if you have an array called `data`, you can create a copy that is twice as large with the following line of code:

```
int[] newData = Arrays.copyOf(data, 2 * data.length);
```

If you want to copy only a portion of an array, there is a similar method called `Arrays.copyOfRange` that accepts an array, a starting index, and an ending index as parameters.

The second limitation is that you can't print an array using a simple `print` or `println` statement. You will get odd output when you do so. JShell understands how to display arrays in a nice way, but in general arrays do not know how to print themselves in a useful format.

The `Arrays` class once again offers a solution: The method `Arrays.toString` returns a conveniently formatted version of an array. Consider, for example, the following three lines of code:

```
int[] primes = {2, 3, 5, 7, 11, 13, 17, 19, 23};
System.out.println(primes);
System.out.println(Arrays.toString(primes));
```

It produces the following output:

```
[I@fee4648
[2, 3, 5, 7, 11, 13, 17, 19, 23]
```

Notice that the first line of output is not at all helpful. The second line, however, allows us to see the list of prime numbers in the array because we called `Arrays.toString` to format the array before printing it.

The third limitation is that you can't compare arrays for equality using a simple == test. We saw that this was true of `Strings` as well. If you want to know whether two arrays contain the same set of values, you should call the `Arrays.equals` method:

```
int[] data1 = {1, 1, 2, 3, 5, 8, 13, 21};
int[] data2 = {1, 1, 2, 3, 5, 8, 13, 21};
if (Arrays.equals(data1, data2)) {
    System.out.println("They store the same data");
}
```

Table 7.2 Useful Methods of the `Arrays` Class

Method	Description
`copyOf(array, newSize)`	returns a copy of the array with the given size
`copyOfRange(array, startIndex, endIndex)`	returns a copy of the given subportion of the given array from `startIndex` (inclusive) to `endIndex` (exclusive)
`equals(array1, array2)`	returns `true` if the arrays contain the same elements
`fill(array, value)`	sets every element of the array to be the given value
`sort(array)`	rearranges the elements so that they appear in sorted (nondecreasing) order
`toString(array)`	returns a `String` representation of the array, as in `[3, 5, 7]`

This code prints the message that the arrays store the same data. It would not do so if we used a direct comparison with ==.

The `Arrays` class provides other useful methods as well, including methods for sorting the array and for filling it up with a specific value. Table 7.2 contains a list of some of the most useful methods in the `Arrays` class.

7.2 Array-Traversal Algorithms

VideoNote

The previous section presented two standard patterns for manipulating an array. The first is the traversing loop, which uses a variable of type `int` to index each array value:

```
for (int i = 0; i < <array>.length; i++) {
    <do something with array[i]>;
}
```

The second is the for-each loop:

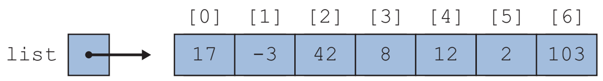
```
for (<type> <name> : <array>) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

In this section we will explore some common array algorithms that can be implemented with these patterns. Of course, not all array operations can be implemented this way—the section ends with an example that requires a modified version of the standard code.

We will implement each operation as a method. Java does not allow you to write generic array code, so we have to pick a specific type. We'll assume that you are operating on an array of `int` values. If you are writing a program to manipulate a different kind of array, you'll have to modify the code for the type you are using (e.g., changing `int[]` to `double[]` if you are manipulating an array of `double` values).

Printing an Array

Suppose you have an array of `int` values like the following:



How would you go about printing the values in the array? For other types of data, you can use a `println` statement:

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

Unfortunately, as mentioned in the `Arrays` class section of this chapter, with an array the `println` statement produces strange output like the following:

```
[I@6caf43
```

This is not helpful output, and it tells us nothing about the contents of the array. We saw that Java provides a solution to this problem in the form of a method called `Arrays.toString` that converts the array into a convenient text form. You can rewrite the `println` as follows to include a call on `Arrays.toString`:

```
System.out.println(Arrays.toString(list));
```

This line of code produces the following output:

```
[17, -3, 42, 8, 12, 2, 103]
```

This is a reasonable way to show the contents of the array, and in many situations it will be sufficient. However, for situations in which you want something different, you can write your own method.

Suppose that you want to write each number on a line by itself. In that case, you can use a for-each loop that does a `println` for each value:

```
public static void print(int[] list) {
    for (int n : list) {
        System.out.println(n);
    }
}
```

You can then call this method with the variable `list`:

```
print(list);
```

This call produces the following output:

```
17
-3
42
8
12
2
103
```

In some cases, the for-each loop doesn't get you quite what you want, though. For example, consider how the `Arrays.toString` method must be written. It produces a list of values that are separated by commas, which is a classic fencepost problem (e.g., seven values separated by six commas). To solve the fencepost problem, you'd want to use an indexing loop instead of a for-each loop so that you can print the first value before the loop:

```
System.out.print(list[0]);
for (int i = 1; i < list.length; i++) {
    System.out.print(", " + list[i]);
}
System.out.println();
```

Notice that `i` is initialized to `1` instead of `0` because `list[0]` is printed before the loop. This code produces the following output for the preceding sample array:

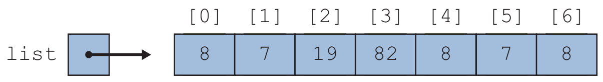
```
17, -3, 42, 8, 12, 2, 103
```

Even this code is not correct, though, because it assumes that there is a `list[0]` to print. It is possible for arrays to be empty, with a length of 0, in which case this code will generate an `ArrayIndexOutOfBoundsException`. The version of the method that follows produces output that matches the `String` produced by `Arrays.toString`. The printing statements just before and just after the loop have been modified to include square brackets, and a special case has been included for empty arrays:

```
public static void print(int[] list) {
    if (list.length == 0) {
        System.out.println("[]");
    } else {
        System.out.print("[" + list[0]);
        for (int i = 1; i < list.length; i++) {
            System.out.print(", " + list[i]);
        }
        System.out.println("]");
    }
}
```

Searching and Replacing

Often you'll want to search for a specific value in an array. For example, you might want to count how many times a particular value appears in an array. Suppose you have an array of `int` values like the following:



Counting occurrences is the simplest search task, because you always examine each value in the array and you don't need to change the contents of the array. You can accomplish this task with a for-each loop that keeps a count of the number of occurrences of the value for which you're searching:

```
public static int count(int[] list, int target) {
    int count = 0;
    for (int n : list) {
        if (n == target) {
            count++;
        }
    }
    return count;
}
```

You can use this method in the following call to figure out how many 8s are in the list:

```
int number = count(list, 8);
```

This call would set `number` to `3` for the sample array, because there are three occurrences of `8` in the list. If you instead made the call

```
int number = count(list, 2);
```

`number` would be set to `0`, because there are no occurrences of `2` in the list.

Sometimes you want to find out where a value is in a list. You can accomplish this task by writing a method that will return the index of the first occurrence of the value in the list. Because you don't know exactly where you'll find the value, you might try using a `while` loop, as in the following pseudocode:

```
int i = 0;
while (we haven´t found it yet) {
    i++;
}
```

However, there is a simpler approach. Because you're writing a method that returns a value, you can return the appropriate index as soon as you find a match. That means you can use the standard traversal loop to solve this problem:

```
for (int i = 0; i < list.length; i++) {
    if (list[i] == target) {
        return i;
    }
}
```

Remember that a `return` statement terminates a method, so you'll break out of this loop as soon as the target value is found. But what if the value isn't found? What if you traverse the entire array and find no matches? In that case, the `for` loop will finish executing without ever returning a value.

There are many things you can do if the value is not found. The convention used throughout the Java class libraries is to return the value `−1` to indicate that the value is not anywhere in the list. So you can add an extra `return` statement after the loop that will be executed only when the target value is not found. Putting all this together, you get the following method:

```
public static int indexOf(int[] list, int target) {
    for (int i = 0; i < list.length; i++) {
        if (list[i] == target) {
            return i;
        }
    }
    return -1;
}
```

You can use this method in the following call to find the first occurrence of the value `7` in the list:

```
int position = indexOf(list, 7);
```

This call would set `position` to `1` for the sample array, because the first occurrence of `7` is at index 1. There is another occurrence of `7` later in the array, at index 5, but this code terminates as soon as it finds the first match.

If you instead made the call

```
int position = indexOf(list, 42);
```

position would be set to -1 because there are no occurrences of 42 in the list.

As a final variation, consider the problem of replacing all the occurrences of a value with some new value. This is similar to the counting task. You'll want to traverse the array looking for a particular value and replace the value with something new when you find it. You can't accomplish that task with a for-each loop, because changing the loop variable has no effect on the array. Instead, use a standard traversing loop:

```
public static void replaceAll(int[] list, int target, int replacement) {
    for (int i = 0; i < list.length; i++) {
        if (list[i] == target) {
            list[i] = replacement;
        }
    }
}
```

Notice that even though the method is changing the contents of the array, you don't need to return it in order to have that change take place.

As we noted at the beginning of this section, these examples involve an array of integers, and you would have to change the type if you were to manipulate an array of a different type (for example, changing int[] to double[] if you had an array of double values). But the change isn't quite so simple if you have an array of objects, such as Strings. In order to compare String values, you must make a call on the equals method rather than using a simple == comparison. Here is a modified version of the replaceAll method that would be appropriate for an array of Strings:

```
public static void replaceAll(String[] list, String target,
                              String replacement) {
    for (int i = 0; i < list.length; i++) {
        if (list[i].equals(target)) {
            list[i] = replacement;
        }
    }
}
```

Testing for Equality

Because arrays are objects, testing them for equality is more complex than testing primitive values like integers and doubles for equality. Two arrays are equivalent in value if they have the same length and store the same sequence of values. The method Arrays.equals performs this test:

```
if (Arrays.equals(list1, list2)) {
    System.out.println("The arrays are equal");
}
```

Like the `Arrays.toString` method, often the `Arrays.equals` method will be all you need. But sometimes you'll want slight variations, so it's worth exploring how to write the method yourself.

The method will take two arrays as parameters and will return a `boolean` result indicating whether or not the two arrays are equal. So, the method will look like this:

```
public static boolean equals(int[] list1, int[] list2) {
    ...
}
```

When you sit down to write a method like this, you probably think in terms of defining equality: "The two arrays are equal if their lengths are equal and they store the same sequence of values." But this isn't the easiest approach. For example, you could begin by testing that the lengths are equal, but what would you do next?

```
public static boolean equals(int[] list1, int[] list2) {
    if (list1.length == list2.length) {
        // what do we do?
        ...
    }
    ...
}
```

Methods like this one are generally easier to write if you think in terms of the opposite condition: What would make the two arrays *unequal?* Instead of testing for the lengths being equal, start by testing whether the lengths are unequal. In that case, you know exactly what to do. If the lengths are not equal, the method should return a value of `false`, and you'll know that the arrays are not equal to each other:

```
public static boolean equals(int[] list1, int[] list2) {
    if (list1.length != list2.length) {
        return false;
    }
    ...
}
```

If you get past the `if` statement, you know that the arrays are of equal length. Then you'll want to check whether they store the same sequence of values. Again, test for inequality rather than equality, returning `false` if there's a difference:

```
public static boolean equals(int[] list1, int[] list2) {
    if (list1.length != list2.length) {
        return false;
    }
    for (int i = 0; i < list1.length; i++) {
        if (list1[i] != list2[i]) {
            return false;
        }
    }
    ...
}
```

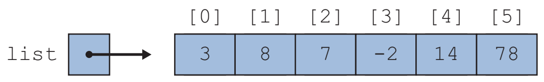
If you get past the `for` loop, you'll know that the two arrays are of equal length and that they store exactly the same sequence of values. In that case, you'll want to return the value `true` to indicate that the arrays are equal. This addition completes the method:

```
public static boolean equals(int[] list1, int[] list2) {
    if (list1.length != list2.length) {
        return false;
    }
    for (int i = 0; i < list1.length; i++) {
        if (list1[i] != list2[i]) {
            return false;
        }
    }
    return true;
}
```

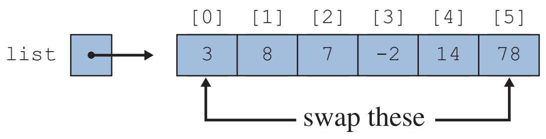
This is a common pattern for a method like `equals`: You test all of the ways that the two objects might not be equal, returning `false` if you find any differences, and returning `true` at the very end so that if all the tests are passed the two objects are declared to be equal.

Reversing an Array

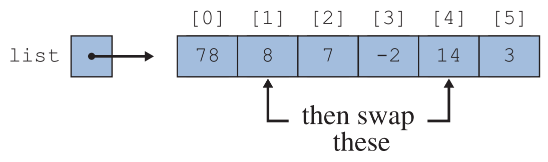
As a final example of common operations, let's consider the task of reversing the order of the elements stored in an array. For example, suppose you have an array that stores the following values:



One approach would be to create a new array and to store the values from the first array into the second array in reverse order. Although that approach would be reasonable, you should be able to solve the problem without constructing a second array. Another approach is to conduct a series of exchanges or swaps. For example, the value `3` at the front of the list and the value `78` at the end of the list need to be swapped:



After swapping that pair, you can swap the next pair in (the values at indexes 1 and 4):



You can continue swapping until the entire list has been reversed. Before we look at the code that will perform this reversal, let's consider the general problem of swapping two values.

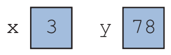
Suppose you have two integer variables `x` and `y` that have the values `3` and `78`:

```
int x = 3;
int y = 78;
```

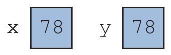
How would you swap these values? A naive approach is to simply assign the values to one another:

```
// will not swap properly
x = y;
y = x;
```

Unfortunately, this doesn't work. You start out with the following:



When the first assignment statement is executed, you copy the value of `y` into `x`:

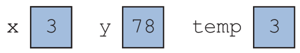


You want `x` to eventually become equal to `78`, but if you attempt to solve the problem this way, you lose the old value of `x` as soon as you assign the value of `y` to it. The second assignment statement then copies the new value of `x`, `78`, back into `y`, which leaves you with two variables equal to `78`.

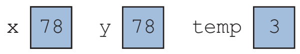
The standard solution is to introduce a temporary variable that you can use to store the old value of `x` while you're giving `x` its new value. You can then copy the old value of `x` from the temporary variable into `y` to complete the swap:

```
int temp = x;
x = y;
y = temp;
```

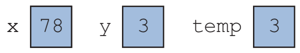
You start by copying the old value of `x` into `temp`:



Then you put the value of `y` into `x`:



Next, you copy the old value of `x` from `temp` to `y`:



At this point you have successfully swapped the values of `x` and `y`, so you don't need `temp` anymore.

In some programming languages, you can define this as a `swap` method that can be used to exchange two `int` values:

```
// this method won't work
public static void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}
```

As you've seen, this kind of method won't work in Java because the `x` and `y` that are swapped will be copies of any integer values passed to them. But because arrays are stored as objects, you can write a variation of this method that takes an array and two indexes as parameters and swaps the values at those indexes:

```
public static void swap(int[] list, int i, int j) {
    int temp = list[i];
    list[i] = list[j];
    list[j] = temp;
}
```

The code in this method matches the code in the previous method, but instead of using `x` and `y` it uses `list[i]` and `list[j]`. This method will work because, instead of changing simple `int` variables, the method is changing the contents of the array.

Given this `swap` method, you can fairly easily write a reversing method. You just have to think about what combinations of values to swap. Start by swapping the first and last values. The sample array has a length of 6, which means that you will be swapping the values at indexes 0 and 5. But you want to write the code so that it works for an array of any length. In general, the first swap you'll want to perform is to swap element 0 with element `(list.length - 1)`:

```
swap(list, 0, list.length - 1);
```

Then you'll want to swap the second value with the second-to-last value:

```
swap(list, 1, list.length - 2);
```

Then you'll swap the third value with the third-to-last value:

```
swap(list, 2, list.length - 3);
```

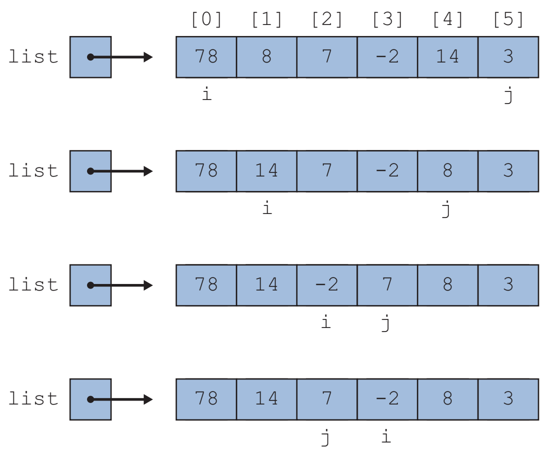
There is a pattern to these swaps that you can capture with a loop. If you use a variable `i` for the first parameter of the call on `swap` and introduce a local variable `j` to store an expression for the second parameter to `swap`, each of these calls will take the following form:

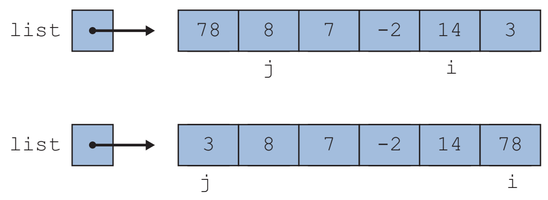
```
int j = list.length - i - 1;
swap(list, i, j);
```

To implement the reversal, you could put the method inside the standard traversal loop:

```
// doesn't quite work
for (int i = 0; i < list.length; i++) {
    int j = list.length - i - 1;
    swap(list, i, j);
}
```

If you were to test this code, though, you'd find that it seems to have no effect whatsoever. The list stores the same values after executing this code as it stores initially. The problem is that this loop does too much swapping. Here is a trace of the six swaps that are performed on the list [3, 8, 7, −2, 14, 78], with an indication of the values of `i` and `j` for each step:





The values of `i` and `j` cross halfway through this process. As a result, the first three swaps successfully reverse the array, and then the three swaps that follow undo the work of the first three. To fix this problem, you need to stop it halfway through the process. This task is easily accomplished by changing the test:

```
for (int i = 0; i < list.length / 2; i++) {
    int j = list.length - i - 1;
    swap(list, i, j);
}
```

In the sample array, `list.length` is `6`. Half of that is `3`, which means that this loop will execute exactly three times. That is just what you want in this case (the first three swaps), but you should be careful to consider other possibilities. For example, what if `list.length` were `7`? Half of that is also `3`, because of truncating division. Is three the correct number of swaps for an odd-length list? The answer is yes. If there are an odd number of elements, the value in the middle of the list does not need to be swapped. So, in this case, a simple division by 2 turns out to be the right approach.

Including this code in a method, you end up with the following overall solution:

```
public static void reverse(int[] list) {
    for (int i = 0; i < list.length / 2; i++) {
        int j = list.length - i - 1;
        swap(list, i, j);
    }
}
```

String Traversal Algorithms

In Java we often think of a string as a chunk of text, but you can also think of it as a sequence of individual characters. Viewed in this light, a string is a lot like an array. Recall that the individual elements of a string are of type `char` and that you can access the individual character values by calling the `charAt` method.

The same techniques we have used to write array traversal algorithms can be used to write string traversal algorithms. The syntax is slightly different, but the logic is the same. Our array traversal template looks like this:

```
for (int i = 0; i < <array>.length; i++) {
    <do something with array[i]>;
}
```

The corresponding string algorithm template looks like this:

```
for (int i = 0; i < <string>.length(); i++) {
    <do something with string.charAt(i)>;
}
```

Notice that with arrays you refer to `length` without using parentheses, but with a string you do use parentheses. Notice also that the array square bracket notation is replaced with a call on the `charAt` method.

For example, you can count the number of occurrences of the letter “i” in “Mississippi” with this code:

```
String s = "Mississippi";
int count = 0;
for (int i = 0; i < s.length(); i++) {
    if (s.charAt(i) == 'i') {
        count++;
    }
}
```

This code would correctly compute that there are four occurrences of “i” in the string. For another example, consider the task of computing the reverse of a string. You can traverse the string building up a new version that has the letters in reverse order by putting each new character at the front of the string you are building up. Here is a complete method that uses this approach:

```
public static String reverse(String text) {
    String result = "";
    for (int i = 0; i < text.length(); i++) {
        result = text.charAt(i) + result;
    }
    return result;
}
```

If you make the call `reverse("Mississippi")`, the method returns `"ippississiM"`.

Functional Approach

Chapter 19 describes a different approach to manipulating arrays that leads to code that looks quite different than the examples in this section. It relies on features added to the Java programming language starting with version 8 that allow you to manipulate arrays and other data structures in a more declarative manner. Instead of specifying exactly how to traverse an array, you can instead tell Java what you want to do with the array elements and allow Java to figure out how to do the traversal. The addition of the for-each loop starting with version 5 of Java was an initial move in this direction, but the new features go much further.

Suppose, for example, that you have an array of values defined as follows:

```
int[] numbers = {8, 3, 2, 17};
```

Let's look at the code you would write for two simple tasks: finding the sum and printing the values. Using the standard traversal loops, you would write the following code.

```
// sum an array of numbers and print them (for loop)
int sum = 0;
for (int i = 0; i < numbers.length; i++) {
    sum += numbers[i];
}
System.out.println("sum = " + sum);
for (int i = 0; i < numbers.length; i++) {
    System.out.println(numbers[i]);
}
```

This code produces the following output.

```
sum = 30
8
3
2
17
```

The for-each loop simplifies this code by specifying that you want to manipulate each of the different values in the array in sequence, but it doesn't require you to include an indexing variable to say exactly how that is done.

```
// sum an array of numbers and print them (for-each loop)
int sum = 0;
for (int n : numbers) {
    sum += n;
}
System.out.println("sum = " + sum);
for (int n : numbers) {
    System.out.println(n);
}
```

With the new Java 8 features, this becomes even simpler. The task of finding the sum of a sequence of values is so common that there is a built-in method that does it for you. And the task of printing each value with a call on the `println` method of `System.out` can also be expressed in a very concise manner.

```
// sum an array of numbers and print them (functional)
int sum = Arrays.stream(numbers).sum();
System.out.println("sum = " + sum);
Arrays.stream(numbers).forEach(System.out::println);
```

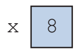
This code doesn't at all describe how the traversal is to be performed. Instead, you tell Java the operations you want to have performed on the values in the array and leave it up to Java to perform the traversal. See Chapter 19 for a more complete explanation of this approach.

7.3 Reference Semantics

In Java, arrays are objects. We have been using objects since Chapter 3 but we haven't yet discussed in detail how they are stored. It's about time that we explored the details. Objects are stored in the computer's memory in a different way than primitive data are stored. For example, when we declare the integer variable

```
int x = 8;
```

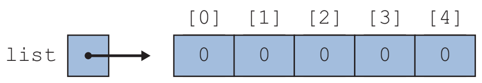
the variable stores the actual data. So, we've drawn pictures like the following:



The situation is different for arrays and other objects. With regard to objects, the variable doesn't store the actual data. Instead, the data are stored in an object and the variable stores a reference to the location at which the object is stored. So, we have two different elements in the computer's memory: the variable and the object. Thus, when we construct an array object such as

```
int[] list = new int[5];
```

we end up with the following:



As the diagram indicates, two different values are stored in memory: the array itself, which appears on the right side of the diagram, and a variable called `list`, which stores a reference to the array (represented in this picture as an arrow). We say that `list` *refers* to the array.

It may take some time for you to get used to the two different approaches to storing data, but these approaches are so common that computer scientists have technical terms to describe them. The system for the primitive types like `int` is known as *value semantics*, and those types are often referred to as *value types.* The system for arrays and other objects is known as *reference semantics*, and those types are often referred to as *reference types.*

Value Semantics (Value Types)

A system in which values are stored directly and copying is achieved by creating independent copies of values. Types that use value semantics are called value types.

Reference Semantics (Reference Types)

A system in which references to values are stored and copying is achieved by copying these references. Types that use reference semantics are called reference types.

It will take us a while to explore all of the implications of this difference. The key thing to remember is that when you are working with objects, you are always working with references to data rather than the data itself.

At this point you are probably wondering why Java has two different systems. Java was designed for object-oriented programming, so the first question to consider is why Sun decided that objects should have reference semantics. There are two primary reasons:

- **Efficiency.** Objects can be complex, which means that they can take up a lot of space in memory. If we made copies of such objects, we would quickly run out of memory. A `String` object that stores a large number of characters might take up a lot of space in memory. But even if the `String` object is very large, a reference to it can be fairly small, in the same way that even a mansion has a simple street address. As another analogy, think how we use cell phones to communicate with people. The phones can be very tiny and easy to transport because cell phone numbers don't take up much space. Imagine that, instead of carrying around a set of cell phone numbers, you tried to carry around the actual people!
- **Sharing.** Often, having a copy of something is not good enough. Suppose that your instructor tells all of the students in the class to put their tests into a certain box. Imagine how pointless and confusing it would be if each student made a copy of the box. The obvious intent is that all of the students use the same box. Reference semantics allows you to have many references to a single object, which allows different parts of your program to share a certain object.

Without reference semantics, Java programs would be more difficult to write. Then why did Sun also decide to include primitive types that have value semantics? The reasons are primarily historical. Sun wanted to leverage the popularity of C and C++, which had similar types, and to guarantee that Java programs would run quickly, which was easier to accomplish with the more traditional primitive types. If Java's designers had a chance to redesign Java today, the company might well get rid of the primitive types and use a consistent object model with just reference semantics.

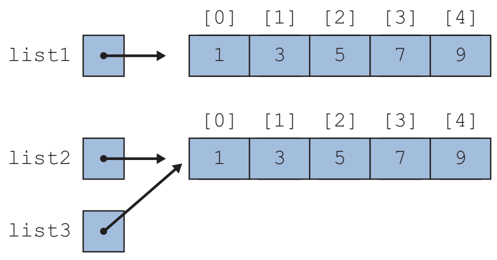
Multiple Objects

In the previous section, you saw how to manipulate a single array object. In this section, we will delve deeper into the implications of reference semantics by considering what happens when there are multiple objects and multiple references to the same object.

Consider the following code:

```
int[] list1 = new int[5];
int[] list2 = new int[5];
for (int i = 0; i < list1.length; i++) {
    list1[i] = 2 * i + 1;
    list2[i] = 2 * i + 1;
}
int[] list3 = list2;
```

Each call on `new` constructs a new object and this code has two calls on `new`, so that means we have two different objects. The code is written in such a way that `list2` will always have the exact same length and sequence of values as `list1`. After the two arrays are initialized, we define a third array variable that is assigned to `list2`. This step creates a new reference but not a new object. After the computer executes the code, memory would look like this:



We have three variables but only two objects. The variables `list2` and `list3` both refer to the same array object. Using the cell phone analogy, you can think of this as two people who both have the cell phone number for the same person. That means that either one of them can call the person. Or, as another analogy, suppose that both you and a friend of yours know how to access your bank information online. That means that you both have access to the same account and that either one of you can make changes to the account.

The implication of this method is that `list2` and `list3` are in some sense both equally able to modify the array to which they refer. The line of code

```
list2[2]++;
```

will have exactly the same effect as the line

```
list3[2]++;
```

Since both variables refer to the same array object, you can access the array through either one.

Reference semantics help us to understand why a simple == test does not give us what we might expect. When this test is applied to objects, it determines whether two *references* are the same (not whether the objects to which they refer are somehow equivalent). In other words, when we test whether two references are equal, we are testing whether they refer to exactly the same object.

The variables `list2` and `list3` both refer to the same array object. As a result, if we ask whether `list2 == list3`, the answer will be yes (the expression evaluates to `true`). But if we ask whether `list1 == list2`, the answer will be no (the expression evaluates to `false`) even though we think of the two arrays as somehow being equivalent.

Sometimes you want to know whether two variables refer to exactly the same object, and for those situations, the simple == comparison will be appropriate. But you'll also want to know whether two objects are somehow equivalent in value, in which case you should call methods like `Arrays.equals` or the string `equals` method.

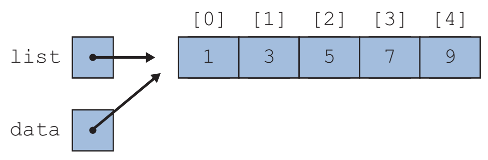
Understanding reference semantics also allows you to understand why a method is able to change the contents of an array that is passed to it as a parameter. Remember that earlier in the chapter we considered the following method:

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

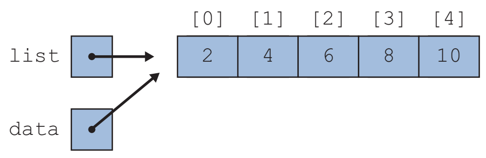
We saw that when our variable `list` was initialized to an array of odd numbers, we could increment all of the values in the array by means of the following line:

```
incrementAll(list);
```

When the method is called, we make a copy of the variable `list`. But the variable `list` is not itself the array; rather, it stores a reference to the array. So, when we make a copy of that reference, we end up with two references to the same object:



Because `data` and `list` both refer to the same object, when we change `data` by saying `data[i]++`, we end up changing the object to which `list` refers. That's why, after the loop increments each element of data, we end up with the following:



The key lesson to draw from this discussion is that when we pass an array as a parameter to a method, that method has the ability to change the contents of the array.

Before we leave the subject of reference semantics, we should describe in more detail the concept of the special value `null`. It is a special keyword in Java that is used to represent "no object".

`null`
A Java keyword signifying no object.

The concept of null doesn't have any meaning for value types like `int` and `double` that store actual values. But it can make sense to set a variable that stores a reference to `null`. This is a way of telling the computer that you want to have the variable, but you haven't yet come up with an object to which it should refer. So you can use `null` for variables of any object type, such as a `String` or array:

```
String s = null;
int[] list = null;
```

There is a difference between setting a variable to an empty string and setting it to `null`. When you set a variable to an empty string, there is an actual object to which your variable refers (although not a very interesting object). When you set a variable to `null`, the variable doesn't yet refer to an actual object. If you try to use the variable to access the object when it has been set to `null`, Java will throw a `NullPointerException`.

7.4 Advanced Array Techniques

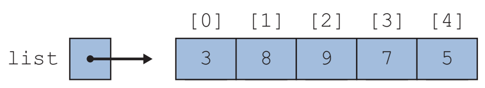
VideoNote

In this section we'll discuss some advanced uses of arrays, such as algorithms that cannot be solved with straightforward traversals. we'll also see how to create arrays that store objects instead of primitive values.

Shifting Values in an Array

You'll often want to move a series of values in an array. For example, suppose you have an array of integers that stores the sequence of values [3, 8, 9, 7, 5] and you want to send the value at the front of the list to the back and keep the order of the other values the same. In other words, you want to move the 3 to the back, yielding the list [8, 9, 7, 5, 3]. Let's explore how to write code to perform that action.

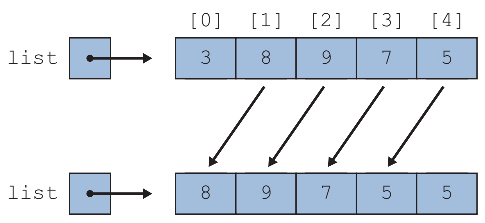
Suppose you have a variable of type `int[]` called `list` of length 5 that stores the values [3, 8, 9, 7, 5]:



The shifting operation is similar to the swap operation discussed in the previous section, and you'll find that it is useful to use a temporary variable here as well. The 3 at the front of the list is supposed to go to the back of the list, and the other values are supposed to rotate forward. You can make the task easier by storing the value at the front of the list (3, in this example) into a local variable:

```
int first = list[0];
```

With that value safely tucked away, you now have to shift the other four values to the left by one position:



The overall task breaks down into four different shifting operations, each of which is a simple assignment statement:

```
list[0] = list[1];
list[1] = list[2];
list[2] = list[3];
list[3] = list[4];
```

Obviously you'd want to write this as a loop rather than writing a series of individual assignment statements. Each of the preceding statements is of the form

```
list[i] = list[i + 1];
```

You'll replace list element `[i]` with the value currently stored in list element `[i + 1]`, which shifts that value to the left. You can put this line of code inside a standard traversing loop:

```
for (int i = 0; i < list.length; i++) {
    list[i] = list[i + 1];
}
```

This loop is almost the right answer, but it has an off-by-one bug. This loop will execute five times for the sample array, but you only want to shift four values (you want to do the assignment for `i` equal to `0`, `1`, `2`, and `3`, but not for `i` equal to `4`). So, this loop goes one too many times. On the last iteration of the loop, when `i` is equal to `4`, the loop executes the following line of code:

```
list[i] = list[i + 1];
```

This line becomes:

```
list[4] = list[5];
```

There is no value `list[5]` because the array has only five elements, with indexes 0 through 4. So, this code generates an `ArrayIndexOutOfBoundsException`. To fix the problem, alter the loop so that it stops one iteration early:

```
for (int i = 0; i < list.length − 1; i++) {
    list[i] = list[i + 1];
}
```

In place of the usual `list.length`, use `(list.length − 1)`. You can think of the minus one in this expression as offsetting the plus one in the assignment statement.

Of course, there is one more detail you must address. After shifting the values to the left, you've made room at the end of the list for the value that used to be at the front of the list (which is currently stored in a local variable called `first`). When the loop has finished executing, you have to place this value at index 4:

```
list[list.length − 1] = first;
```

Here is the final method:

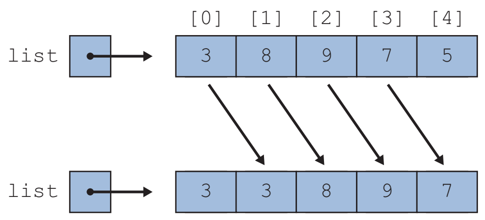
```
public static void rotateLeft(int[] list) {
    int first = list[0];
    for (int i = 0; i < list.length − 1; i++) {
        list[i] = list[i + 1];
    }
    list[list.length − 1] = first;
}
```

An interesting variation on this method is to rotate the values to the right instead of rotating them to the left. To perform this inverse operation, you want to take the value that is currently at the end of the list and bring it to the front, shifting the remaining values to the right. So, if a variable called `list` initially stores the values [3, 8, 9, 7, 5], it should bring the 5 to the front and store the values [5, 3, 8, 9, 7].

Begin by tucking away the value that is being rotated into a temporary variable:

```
int last = list[list.length - 1];
```

Then shift the other values to the right:



In this case, the four individual assignment statements would be the following:

```
list[1] = list[0];
list[2] = list[1];
list[3] = list[2];
list[4] = list[3];
```

A more general way to write this is the following line of code:

```
list[i] = list[i - 1];
```

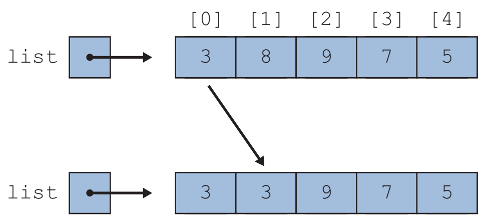
If you put this code inside the standard `for` loop, you get the following:

```
// doesn't work
for (int i = 0; i < list.length; i++) {
    list[i] = list[i - 1];
}
```

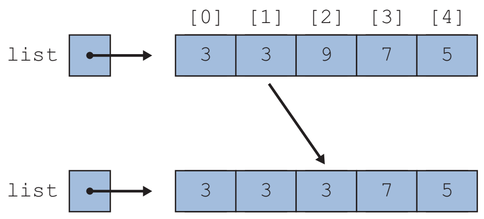
There are two problems with this code. First, there is another off-by-one bug. The first assignment statement you want to perform would set `list[1]` to contain the value that is currently in `list[0]`, but this loop sets `list[0]` to `list[-1]`. Java generates an `ArrayIndexOutOfBoundsException` because there is no value `list[-1]`. You want to start `i` at `1`, not `0`:

```
// still doesn't work
for (int i = 1; i < list.length; i++) {
    list[i] = list[i - 1];
}
```

However, this version of the code doesn't work either. It avoids the exception, but think about what it does. The first time through the loop it assigns `list[1]` to what is in `list[0]`:

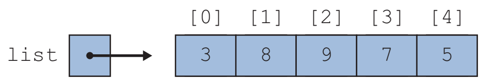


What happened to the value `8`? It's overwritten with the value `3`. The next time through the loop `list[2]` is set to be `list[1]`:

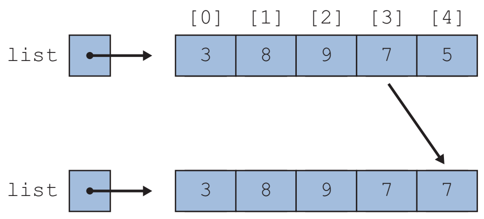


You might say, "Wait a minute . . . `list[1]` isn't a `3`, it's an `8`." It was an `8` when you started, but the first iteration of the loop replaced the `8` with a `3`, and now the `3` has been copied into the spot where `9` used to be.

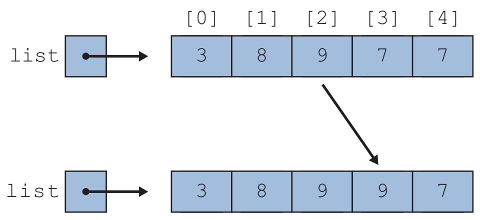
The loop continues in this way, putting `3` into every cell of the array. Obviously, that's not what you want. To make this code work, you have to run the loop in reverse order (from right to left instead of left to right). So let's back up to where we started:



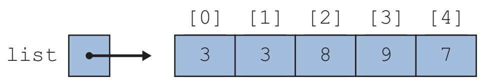
We tucked away the final value of the list into a local variable. That frees up the final array position. Now, assign `list[4]` to be what is in `list[3]`:



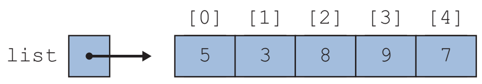
This wipes out the `5` that was at the end of the list, but that value is safely stored away in a local variable. And once you've performed this assignment statement, you free up `list[3]`, which means you can now set `list[3]` to be what is currently in `list[2]`:



The process continues in this manner, copying the `8` from index 1 to index 2 and copying the `3` from index 0 to index 1, leaving you with the following:



At this point, the only thing left to do is to put the `5` stored in the local variable at the front of the list:



You can reverse the `for` loop by changing the `i++` to `i--` and adjusting the initialization and test. The final method is as follows:

```
public static void rotateRight(int[] list) {
    int last = list[list.length - 1];
    for (int i = list.length - 1; i >= 1; i--) {
        list[i] = list[i - 1];
    }
    list[0] = last;
}
```

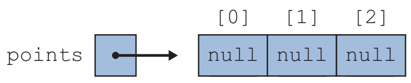
Arrays of Objects

All of the arrays we have looked at so far have stored primitive values like simple `int` values, but you can have arrays of any Java type. Arrays of objects behave slightly differently, though, because objects are stored as references rather than as data values. Constructing an array of objects is usually a two-step process, because you normally have to construct both the array and the individual objects.

As an example, Java has a `Point` class as part of its `java.awt` package. Each `Point` object is used for storing the (*x*, *y*) coordinates of a point in two-dimensional space. (We will discuss this class in more detail in the next chapter, but for now we will just construct a few objects from it.) Suppose that you want to construct an array of `Point` objects. Consider the following statement:

```
Point[] points = new Point[3];
```

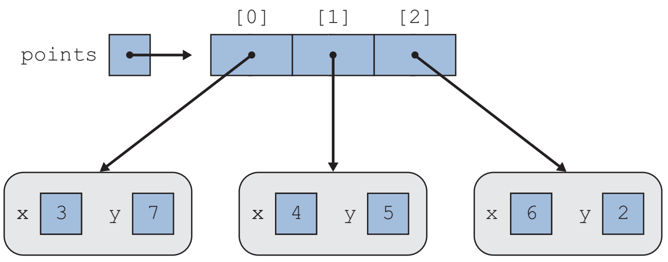
This statement declares a variable called `points` that refers to an array of length 3 that stores references to `Point` objects. Using the new keyword to construct the array doesn't construct any actual `Point` objects. Instead it constructs an array of length 3, each element of which can store a reference to a `Point`. When Java constructs the array, it auto-initializes these array elements to the zero-equivalent for the type. The zero-equivalent for all reference types is the special value `null`, which indicates "no object":



The actual `Point` objects must be constructed separately with the `new` keyword, as in the following code:

```
Point[] points = new Point[3];
points[0] = new Point(3, 7);
points[1] = new Point(4, 5);
points[2] = new Point(6, 2);
```

After these lines of code execute, your program will have created individual `Point` objects referred to by the various array elements:



Notice that the new keyword is required in four different places, because there are four objects to be constructed: the array itself and the three individual `Point` objects. You could also use the curly brace notation for initializing the array, in which case you don't need the new keyword to construct the array itself:

```
Point[] points = {new Point(3, 7), new Point(4, 5), new Point(6, 2)};
```

Command-Line Arguments

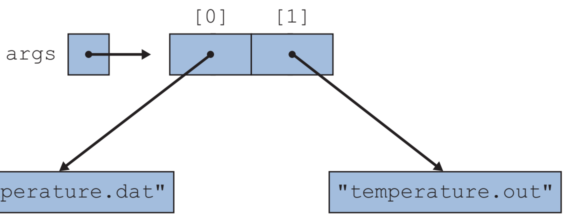
As you've seen since Chapter 1, whenever you define a `main` method, you're required to include as its parameter `String[] args`, which is an array of `String` objects. Java itself initializes this array if the user provides what are known as *command-line* *arguments* when invoking Java. For example, the user could execute a Java class called `DoSomething` from a command prompt or terminal by using a command like:

```
java DoSomething
```

The user has the option to type extra arguments, as in the following:

```
java DoSomething temperature.dat temperature.out
```

In this case the user has specified two extra arguments that are file names that the program should use (e.g., the names of an input and output file). If the user types these extra arguments when starting up Java, the `String[] args` parameter to `main` will be initialized to an array of length 2 that stores these two `strings`:



Nested Loop Algorithms

All of the algorithms we have seen have been written with a single loop. But many computations require nested loops. For example, suppose that you were asked to print all inversions in an array of integers. An inversion is defined as a pair of numbers in which the first number in the list is greater than the second number.

In a sorted list such as [1, 2, 3, 4], there are no inversions at all and there is nothing to print. But if the numbers appear instead in reverse order, [4, 3, 2, 1], then there are many inversions to print. We would expect output like the following:

```
(4, 3)
(4, 2)
(4, 1)
(3, 2)
(3, 1)
(2, 1)
```

Notice that any given number (e.g., 4 in the list above) can produce several different inversions, because it might be followed by several smaller numbers (1, 2, and 3 in the example). For a list that is partially sorted, as in [3, 1, 4, 2], there are only a few inversions, so you would produce output like this:

```
(3, 1)
(3, 2)
(4, 2)
```

This problem can't be solved with a single traversal because we are looking for pairs of numbers. There are many possible first values in the pair and many possible second values in the pair. Let's develop a solution using pseudocode.

We can't produce all pairs with a single loop, but we can use a single loop to consider all possible first values:

```
for (every possible first value) {
    print all inversions that involve this first value.
}
```

Now we just need to write the code to find all the inversions for a given first value. That requires us to write a second, nested loop:

```
for (every possible first value) {
    for (every possible second value) {
        if (first value > second value) {
            print(first, second).
        }
    }
}
```

This problem is fairly easy to turn into Java code, although the loop bounds turn out to be a bit tricky. For now, let's use our standard traversal loop for each:

```
for (int i = 0; i < data.length; i++) {
    for (int j = 0; j < data.length; j++) {
        if (data[i] > data[j]) {
            System.out.println("(" + data[i] + ", " + data[j] + ")");
        }
    }
}
```

The preceding code isn't quite right. Remember that for an inversion, the second value has to appear *after* the first value in the list. In this case, we are computing all possible combinations of a first and second value. To consider only values that come after the given first value, we have to start the second loop at `i + 1` instead of starting at `0`. We can also make a slight improvement by recognizing that because an inversion requires a pair of values, there is no reason to include the last number of the list as a possible first value. So the outer loop involving `i` can end one iteration earlier:

```
for (int i = 0; i < data.length - 1; i++) {
    for (int j = i + 1; j < data.length; j++) {
            if (data[i] > data[j]) {
                System.out.println("(" + data[i] + ", " + data[j] + ")");
            }
        }
    }
```

When you write nested loops like these, it is a common convention to use `i` for the outer loop, `j` for the loop inside the outer loop, and `k` if there is a loop inside the `j` loop.

7.5 Multidimensional Arrays

The array examples in the previous sections all involved what are known as one-dimensional arrays (a single row or a single column of data). Often, you'll want to store data in a multidimensional way. For example, you might want to store a two-dimensional grid of data that has both rows and columns. Fortunately, you can form arrays of arbitrarily many dimensions:

- `double`: one `double`
- `double[]`: a one-dimensional array of `doubles`
- `double[][]`: a two-dimensional grid of `doubles`
- `double[][][]`: a three-dimensional collection of `doubles`
- ...

Arrays of more than one dimension are called *multidimensional arrays.*

Multidimensional Array

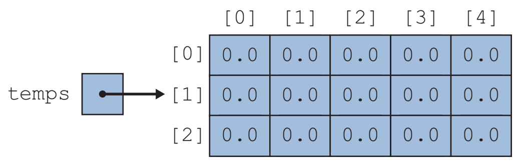
An array of arrays, the elements of which are accessed with multiple integer indexes.

Rectangular Two-Dimensional Arrays

The most common use of a multidimensional array is a two-dimensional array of a certain width and height. For example, suppose that on three separate days you took a series of five temperature readings. You can define a two-dimensional array that has three rows and five columns as follows:

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

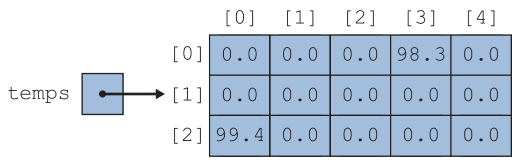
Notice that on both the left and right sides of this assignment statement, you have to use a double set of square brackets. When you are describing the type on the left, you have to make it clear that this is not just a one-dimensional sequence of values, which would be of type `double[]`, but instead a two-dimensional grid of values, which is of type `double[][]`. On the right, when you construct the array, you must specify the dimensions of the grid. The normal convention is to list the row first followed by the column. The resulting array would look like this:



As with one-dimensional arrays, the values are initialized to `0.0` and the indexes start with 0 for both rows and columns. Once you've created such an array, you can refer to individual elements by providing specific row and column numbers (in that order). For example, to set the fourth value of the first row to `98.3` and to set the first value of the third row to `99.4`, you would write the following code:

```
temps[0][3] = 98.3;  // fourth value of first row
temps[2][0] = 99.4;  // first value of third row
```

After the program executes these lines of code, the array would look like this:



It is helpful to think of referring to individual elements in a stepwise fashion, starting with the name of the array. For example, if you want to refer to the first value of the third row, you obtain it through the following steps:

`temps`	the entire grid
`temps[2]`	the entire third row
`temps[2][0]`	the first element of the third row

You can pass multidimensional arrays as parameters just as you pass one-dimensional arrays. You need to be careful about the type, though. To pass the temperature grid, you would have to use a parameter of type `double[][]` (with both sets of brackets). For example, here is a method that prints the grid:

```
public static void print(double[][] grid) {
    for (int i = 0; i < grid.length; i++) {
        for (int j = 0; j < grid[i].length; j++) {
            System.out.print(grid[i][j] + " ");
        }
        System.out.println();
    }
}
```

Notice that to ask for the number of rows you ask for `grid.length` and to ask for the number of columns you ask for `grid[i].length`.

The `Arrays.toString` method mentioned earlier in this chapter does work on multidimensional arrays, but it produces a poor result. When used with the preceding array `temps`, it produces output such as the following:

```
[[D@14b081b, [D@1015a9e, [D@1e45a5c]
```

This poor output is because `Arrays.toString` works by concatenating the `String` representations of the array's elements. In this case the elements are arrays themselves, so they do not convert into `Strings` properly. To correct the problem you can use a different method called `Arrays.deepToString` that will return better results for multidimensional arrays:

```
System.out.println(Arrays.deepToString(temps));
```

The call produces the following output:

```
[[0.0, 0.0, 0.0, 98.3, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0],
[99.4, 0.0, 0.0, 0.0, 0.0]]
```

Arrays can have as many dimensions as you want. For example, if you want a three-dimensional 4 by 4 by 4 cube of integers, you would write the following line of code:

```
int[][][] numbers = new int[4][4][4];
```

The normal convention for the order of values is the plane number, followed by the row number, followed by the column number, although you can use any convention you want as long as your code is written consistently.

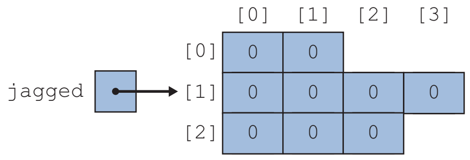
Jagged Arrays

The previous examples have involved rectangular grids that have a fixed number of rows and columns. It is also possible to create a jagged array in which the number of columns varies from row to row.

To construct a jagged array, divide the construction into two steps: Construct the array for holding rows first, and then construct each individual row. For example, to construct an array that has two elements in the first row, four elements in the second row, and three elements in the third row, you can write the following lines of code:

```
int[][] jagged = new int[3][];
jagged[0] = new int[2];
jagged[1] = new int[4];
jagged[2] = new int[3];
```

This code would construct an array that looks like this:



We can explore this technique by writing a program that produces the rows of what is known as *Pascal's triangle.* The numbers in the triangle have many useful mathematical properties. For example, row n of Pascal's triangle contains the coefficients obtained when you expand the equation:

$(x + y)^n$

Here are the results for n between 0 and 4:

$$
\begin{aligned}
&(x + y)^0 = 1\\
&(x + y)^1 = x + y\\
&(x + y)^2 = x^2 + 2xy + y^2\\
&(x + y)^3 = x^3 + 3x^2y + 3xy^2 + y^3\\
&(x + y)^4 = x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4
\end{aligned}
$$

If you pull out just the coefficients, you get the following values:

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
```

These rows of numbers form a five-row Pascal's triangle. One of the properties of the triangle is that if you are given any row, you can use it to compute the next row. For example, let's start with the last row from the preceding triangle:

```
1 4 6 4 1
```

We can compute the next row by adding adjacent pairs of values together. So, we add together the first pair of numbers $(1 + 4)$, then the second pair of numbers $(4 + 6)$, and so on:

```
5 10 10 5
```

$$\underbrace{(1\ +\ 4)}_{5}\ \underbrace{(4\ +\ 6)}_{10}\ \underbrace{(6\ +\ 4)}_{10}\ \underbrace{(4\ +\ 1)}_{5}$$

Then we put a 1 at the front and back of this list of numbers, and we end up with the next row of the triangle:

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
1   5   10  10  5   1
```

This property of the triangle provides a technique for computing it. We can construct it row by row, computing each new row from the values in the previous row. In other words, we write the following loop (assuming that we have a two-dimensional array called `triangle` in which to store the answer):

```
for (int i = 0; i < triangle.length; i++) {
    construct triangle[i] using triangle[i – 1].
}
```

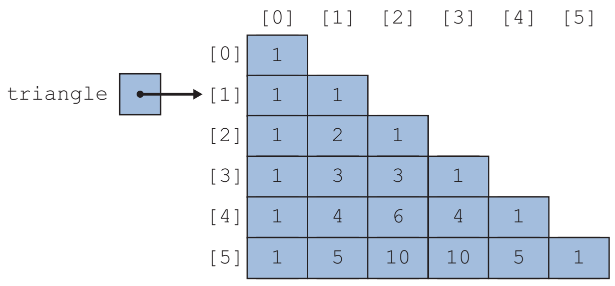
We just need to flesh out the details of how a new row is constructed. This is a jagged array because each row has a different number of elements. Looking at the triangle, you'll see that the first row (row 0) has one value in it, the second row (row 1) has two values in it, and so on. In general, row `i` has `(i + 1)` values, so we can refine our pseudocode as follows:

```
for (int i = 0; i < triangle.length; i++) {
    triangle[i] = new int[i + 1];
    fill in triangle[i] using triangle[i – 1].
}
```

We know that the first and last values in each row should be `1`:

```
for (int i = 0; i < triangle.length; i++) {
    triangle[i] = new int[i + 1];
    triangle[i][0] = 1;
    triangle[i][i] = 1;
    fill in the middle of triangle[i] using triangle[i – 1].
}
```

And we know that the middle values come from the previous row. To figure out how to compute them, let's draw a picture of the array we are attempting to build:



We have already written code to fill in the `1` that appears at the beginning and end of each row. We now need to write code to fill in the middle values. Look at row 5 for an example. The value `5` in column 1 comes from the sum of the values `1` in column 0 and `4` in column 1 in the previous row. The value `10` in column 2 comes from the sum of the values in columns 1 and 2 in the previous row.

More generally, each of these middle values is the sum of the two values from the previous row that appear just above and to the left of it. In other words, for column `j` the values are computed as follows:

```
triangle[i][j] = (value above and left) + (value above).
```

We can turn this into actual code by using the appropriate array indexes:

```
triangle[i][j] = triangle[i - 1][j - 1] + triangle[i - 1][j];
```

We need to include this statement in a `for` loop so that it assigns all of the middle values. The for loop is the final step in converting our pseudocode into actual code:

```
for (int i = 0; i < triangle.length; i++) {
    triangle[i] = new int[i + 1];
    triangle[i][0] = 1;
    triangle[i][i] = 1;
    for (int j = 1; j < i; j++) {
        triangle[i][j] = triangle[i - 1][j - 1] + triangle[i - 1][j];
    }
}
```

If we include this code in a method along with a printing method similar to the grid-printing method described earlier, we end up with the following complete program:

```
// This program constructs a jagged two-dimensional array
// that stores Pascal's Triangle. It takes advantage of the
// fact that each value other than the 1s that appear at the
// beginning and end of each row is the sum of two values
```

```java
// from the previous row.

public class PascalsTriangle {
    public static void main(String[] args) {
        int[][] triangle = new int[11][];
        fillIn(triangle);
        print(triangle);
    }

    public static void fillIn(int[][] triangle) {
        for (int i = 0; i < triangle.length; i++) {
            triangle[i] = new int[i + 1];
            triangle[i][0] = 1;
            triangle[i][i] = 1;
            for (int j = 1; j < i; j++) {
                triangle[i][j] = triangle[i - 1][j - 1]
                                 + triangle[i - 1][j];
            }
        }
    }

    public static void print(int[][] triangle) {
        for (int i = 0; i < triangle.length; i++) {
            for (int j = 0; j < triangle[i].length; j++) {
                System.out.print(triangle[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

This program produces the following output:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1
```

7.6 Arrays of Pixels

Recall from Supplement 3G that images are stored on computers as a two-dimensional grid of colored dots known as *pixels*. One of the most common applications of two-dimensional (2D) arrays is for manipulating the pixels of an image. Popular apps like Instagram provide filters and options for modifying images by applying algorithms to their pixels; for example, you can make an image black-and-white, sharpen it, enhance the colors and contrast, or make it look like an old faded photograph. The two-dimensional rectangular nature of an image makes a 2D array a natural way to represent its pixel data.

Supplement 3G introduced the `DrawingPanel` class that we use to represent a window for drawing 2D shapes and colors. Recall that an image is composed of pixels whose locations are specified with integer coordinates starting from the top-left corner of the image at (0, 0). The various drawing commands of the panel's `Graphics` object, such as `drawRect` and `fillOval`, change the color of regions of pixels. Colors are usually specified by `Color` objects, but the full range of colors comes from mixtures of red, green, and blue elements specified by integers that range from 0 to 255 inclusive. Each combination of three integers specifies a particular color and is known as an *RGB value*.

The `DrawingPanel` includes several methods for getting and setting the color of pixels, listed in Table 7.3. You can interact with a single pixel, or you can grab all of the pixels of the image as a 2D array and manipulate the entire array. The array is in row-major order; that is, the first index of the array is the *y*-coordinate and the second is the *x*-coordinate. For example, `a[`*r*`][`*c*`]` represents the pixel at position ($x=c$, $y=r$). For efficiency it is generally recommended to use the array-based versions of the methods; the individual-pixel methods run slowly when applied repeatedly over all pixels of a large image.

The following `DrawPurpleTriangle` example program uses `getPixels` and `setPixels` to fill a triangular region of the panel with a purple color. Figure 7.1 shows the graphical output of the program. Notice that you must call `setPixels` at the end to see the updated image; changing the array will not produce any effect on the screen until you tell the panel to update itself using the new contents of the array.

Table 7.3 `DrawingPanel` methods related to pixels

Method	Description
`getPixel(x, y)`	returns a pixel's color as a `Color` object
`getPixels()`	returns all pixels' colors as a 2D array of `Color` objects, in row-major order *(first index is row or y, second index is column or x)*
`setPixel(x, y, color)`	sets a pixel's color to the given `Color` object's color
`setPixels(pixels)`	sets all pixels' colors from given 2D array of `Color` objects, resizing the panel if necessary to match the array's dimensions

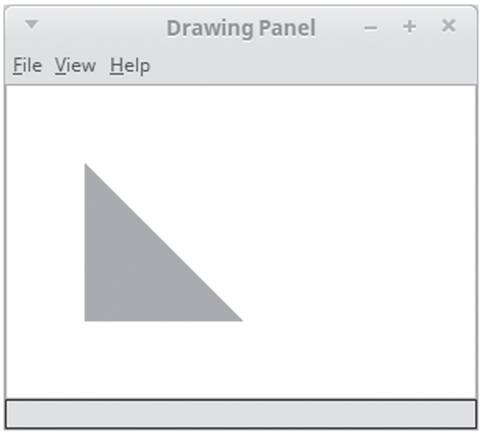


Figure 7.1 Output of `DrawPurpleTriangle`

```java
// This program demonstrates the DrawingPanel's
// getPixels and setPixels methods for
// manipulating pixels of an image.

import java.awt.*;

public class DrawPurpleTriangle {
    public static void main(String[] args) {
        DrawingPanel panel = new DrawingPanel(300, 200);
        Color[][] pixels = panel.getPixels();
        for (int row = 50; row <= 150; row++) {
            for (int col = 50; col <= row; col++) {
                pixels[row][col] = Color.MAGENTA;
            }
        }
        panel.setPixels(pixels);
    }
}
```

You can use `getPixels` and `setPixels` to draw a shape like our purple triangle, but a more typical usage of these methods would be to grab the panel's existing state and alter it in some interesting way. The following `Mirror` program demonstrates the use of a 2D array of `Color` objects. The program's `mirror` method accepts a `DrawingPanel` parameter and flips the pixel contents horizontally, swapping each pixel's color with the one at the opposite horizontal location. The code uses the dimensions of the array to represent the size of the image; `pixels.length` is its height and `pixels[0].length` (the length of the first row of the 2D array) is its width. Figure 7.2 shows the program's graphical output before and after `mirror` is called.

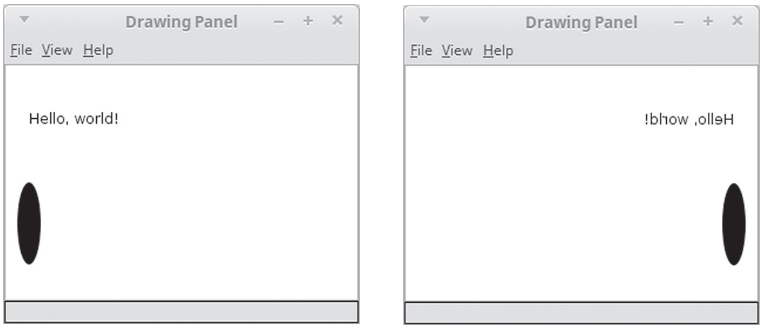


Figure 7.2 Output of `Mirror` before and after mirroring

```java
// This program contains a mirror method that flips the appearance
// of a DrawingPanel horizontally pixel-by-pixel.

import java.awt.*;

public class Mirror {
    public static void main(String[] args) {
        DrawingPanel panel = new DrawingPanel(300, 200);
        Graphics g = panel.getGraphics();
        g.drawString("Hello, world!", 20, 50);
        g.fillOval(10, 100, 20, 70);
        mirror(panel);
    }

    // Flips the pixels of the given drawing panel horizontally.
    public static void mirror(DrawingPanel panel) {
        Color[][] pixels = panel.getPixels();
        for (int row = 0; row < pixels.length; row++) {
            for (int col = 0; col < pixels[0].length / 2; col++) {
                // swap with pixel at "mirrored" location
                int opposite = pixels[0].length - 1 - col;
                Color px = pixels[row][col];
                pixels[row][col] = pixels[row][opposite];
                pixels[row][opposite] = px;
            }
        }
        panel.setPixels(pixels);
    }
}
```

Often you'll want to extract the individual red, green, and blue components of a color to manipulate them. Each pixel's `Color` object has methods to help you do this.

Table 7.4 `Color` methods related to pixel RGB components

Method	Description
`getRed()`	returns the red component from 0-255
`getGreen()`	returns the green component from 0-255
`getBlue()`	returns the blue component from 0-255

The `getRed`, `getGreen`, and `getBlue` methods extract the relevant components out of an RGB integer. Table 7.4 lists the relevant methods.

The following code shows a method that computes the negative of an image, which is found by taking the opposite of each color's RGB values. For example, the opposite of (red = 255, green = 100, blue = 35) is (red = 0, green = 155, blue = 220). The simplest way to compute the negative is to subtract the pixel's RGB values from the maximum color value of 255. Figure 7.3 shows an example output.

```
// Produces the negative of the given image by inverting all color
// values in the panel.
public static void negative(DrawingPanel panel) {
    Color[][] pixels = panel.getPixels();
    for (int row = 0; row < pixels.length; row++) {
        for (int col = 0; col < pixels[0].length; col++) {
            // extract red/green/blue components from 0-255
            int r = 255 - pixels[row][col].getRed();
            int g = 255 - pixels[row][col].getGreen();
            int b = 255 - pixels[row][col].getBlue();

            // update the pixel array with the new color value
            pixels[row][col] = new Color(r, g, b);
        }
    }
    panel.setPixels(pixels);
}
```

All of the previous examples have involved making changes to a 2D pixel array in place. But sometimes you want to create an image with different dimensions, or want to set each pixel based on the values of pixels around it, and therefore you need to create a new pixel array. The following example shows a `stretch` method that widens the contents of a `DrawingPanel` to twice their current width. To do so, it creates an array `newPixels` that is twice as wide as the existing one. (Remember that the first index of the 2D array is y and the second is x, so to widen the array, the code must double the array's second dimension.) The `setPixels` method will resize the panel if necessary to accommodate our new larger array of pixels.

The loop to fill the new array sets the value at each index to the value at half as large an x-index in the original array. So, for example, the original array's pixel value

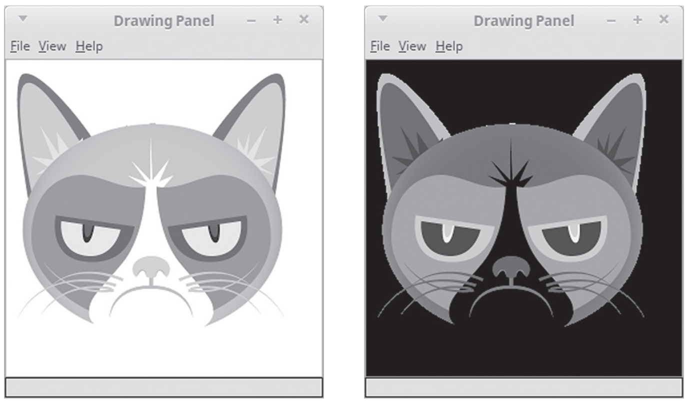


Figure 7.3 Negative of an image (before and after)

at (52, 34) is used to fill the new array's pixels at (104, 68) and (105, 68). Figure 7.4 shows the graphical output of the stretched image.

```
// Stretches the given panel to be twice as wide.
// Any shapes and colors drawn on the panel are stretched to fit.
public static void stretch(DrawingPanel panel) {
    Color[][] pixels = panel.getPixels();
    Color[][] newPixels = new Color[pixels.length][2 * pixels[0].length];
    for (int row = 0; row < pixels.length; row++) {
        for (int col = 0; col < 2 * pixels[0].length; col++) {
            newPixels[row][col] = pixels[row][col / 2];
        }
    }
    panel.setPixels(newPixels);
}
```

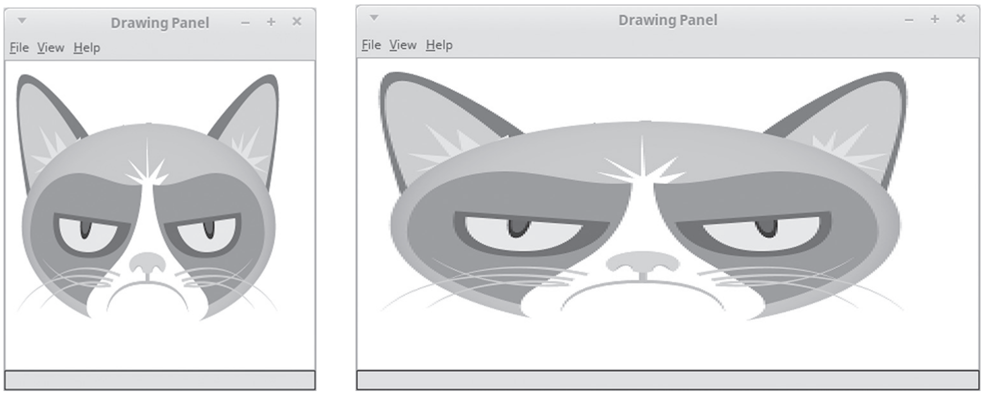


Figure 7.4 Horizontally stretched image (before and after)

The pixel-based methods shown in this section are somewhat inefficient because they create large arrays of `Color` objects, which takes a lot of time and memory. These methods aren't efficient enough for an animation or a game. The `DrawingPanel` provides some additional methods like `getPixelsRGB` that use specially packed integers to represent red, green, and blue color information instead of `Color` objects to improve the speed and memory usage at the cost of a bit of code complexity. If you are interested, you can read about these additional methods in the online `DrawingPanel` documentation at buildingjavaprograms.com.

7.7 Case Study: Benford's Law

Let's look at a more complex program example that involves using arrays. When you study real-world data you will often come across a curious result that is known as *Benford's Law*, named after a physicist named Frank Benford who stated it in 1938.

Benford's Law involves looking at the first digit of a series of numbers. For example, suppose that you were to use a random number generator to generate integers in the range of 100 to 999 and you looked at how often the number begins with 1, how often it begins with 2, and so on. Any decent random number generator would spread the answers out evenly among the nine different regions, so we'd expect to see each digit about one-ninth of the time (11.1%). But with a lot of real-world data, we see a very different distribution.

When we examine data that matches the Benford distribution, we see a first digit of 1 over 30% of the time (almost one third) and, at the other extreme, a first digit of 9 only about 4.6% of the time (less than one in twenty cases). Table 7.5 shows the expected distribution for data that follows Benford's Law.

Why would the distribution turn out this way? Why so many 1s? Why so few 9s? The answer is that exponential sequences have different properties than simple linear sequences. In particular, exponential sequences have a lot more numbers that begin with 1.

Table 7.5 Expected Distribution under Benford's Law

First Digit	Frequency
1	30.1%
2	17.6%
3	12.5%
4	9.7%
5	7.9%
6	6.7%
7	5.8%
8	5.1%
9	4.6%

To explore this phenomenon, let's look at two different sequences of numbers: one that grows linearly and one that grows exponentially. If you start with the number 1 and add 0.2 to it over and over, you get the following linear sequence:

1, 1.2, 1.4, 1.6, 1.8, 2, 2.2, 2.4, 2.6, 2.8, 3, 3.2, 3.4, 3.6, 3.8, 4, 4.2, 4.4, 4.6, 4.8,
5, 5.2, 5.4, 5.6, 5.8, 6, 6.2, 6.4, 6.6, 6.8, 7, 7.2, 7.4, 7.6, 7.8, 8, 8.2, 8.4, 8.6, 8.8,
9, 9.2, 9.4, 9.6, 9.8, 10

In this sequence there are five numbers that begin with 1, five numbers that begin with 2, five numbers that begin with 3, and so on. For each digit, there are five numbers that begin with that digit. That's what we expect to see with data that goes up by a constant amount each time.

But consider what happens when we make it an exponential sequence instead. Let's again start with 1 and continue until we get to 10, but this time let's multiply each successive number by 1.05 (we'll limit ourselves to displaying just two digits after the decimal, but the actual sequence takes into account all of the digits):

1.00, 1.05, 1.10, 1.16, 1.22, 1.28, 1.34, 1.41, 1.48, 1.55, 1.63, 1.71, 1.80, 1.89,
1.98, 2.08, 2.18, 2.29, 2.41, 2.53, 2.65, 2.79, 2.93, 3.07, 3.23, 3.39, 3.56, 3.73,
3.92, 4.12, 4.32, 4.54, 4.76, 5.00, 5.25, 5.52, 5.79, 6.08, 6.39, 6.70, 7.04, 7.39,
7.76, 8.15, 8.56, 8.99, 9.43, 9.91, 10.40

In this sequence there are 15 numbers that begin with 1 (31.25%), 8 numbers that begin with 2 (16.7%), and so on. There are only 2 numbers that begin with 9 (4.2%). In fact, the distribution of digits is almost exactly what you see in the table for Benford's Law.

There are many real-world phenomena that exhibit an exponential character. For example, population tends to grow exponentially in most regions. There are many other data sets that also seem to exhibit the Benford pattern, including sunspots, salaries, investments, heights of buildings, and so on. Benford's Law has been used to try to detect accounting fraud under the theory that when someone is making up data, they are likely to use a more random process that won't yield a Benford style distribution.

For our purposes, let's write a program that reads a file of integers and that shows the distribution of the leading digit. We'll read the data from a file and will run it on several sample inputs. First, though, let's consider the general problem of tallying.

Tallying Values

VideoNote

In programming we often find ourselves wanting to count the number of occurrences of some set of values. For example, we might want to know how many people got a 100 on an exam, how many got a 99, how many got a 98, and so on. Or we might want to know how many days the temperature in a city was above 100 degrees, how many days it was in the 90s, how many days it was in the 80s, and so on. The approach is very nearly the same for each of these tallying tasks. Let's look at a small tallying task in which there are only five values to tally.

Suppose that a teacher scores quizzes on a scale of 0 to 4 and wants to know the distribution of quiz scores. In other words, the teacher wants to know how many scores of 0 there are, how many scores of 1, how many scores of 2, how many scores of 3, and how many scores of 4. Suppose that the teacher has included all of the scores in a data file like the following:

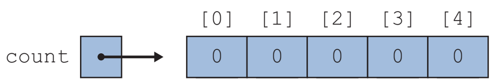
```
1 4 1 0 3 2 1 4 2 0
3 0 2 3 0 4 3 3 4 1
2 4 1 3 1 4 3 3 2 4
2 3 0 4 1 4 4 1 4 1
```

The teacher could hand-count the scores, but it would be much easier to use a computer to do the counting. How can you solve the problem? First you have to recognize that you are doing five separate counting tasks: You are counting the occurrences of the number 0, the number 1, the number 2, the number 3, and the number 4. You will need five counters to solve this problem, which means that an array is a great way to store the data. In general, whenever you find yourself thinking that you need *n* of some kind of data, you should think about using an array of length *n*.

Each counter will be an `int`, so you want an array of five `int` values:

```
int[] count = new int[5];
```

This line of code will allocate the array of five integers and will auto-initialize each to `0`:



You're reading from a file, so you'll need a `Scanner` and a loop that reads scores until there are no more scores to read:

```
Scanner input = new Scanner(new File("tally.dat"));
while (input.hasNextInt()) {
    int next = input.nextInt();
    // process next
}
```

To complete this code, you need to figure out how to process each value. You know that `next` will be one of five different values: `0`, `1`, `2`, `3`, or `4`. If it is `0`, you want to increment the counter for `0`, which is `count[0]`; if it is `1`, you want to increment the counter for `1`, which is `count[1]`, and so on. We have been solving problems like this one with nested `if/else` statements:

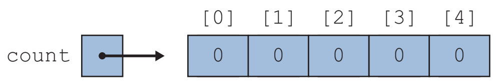
```
if (next == 0) {
    count[0]++;
} else if (next == 1) {
    count[1]++;
} else if (next == 2) {
    count[2]++;
} else if (next == 3) {
    count[3]++;
} else { // next == 4
    count[4]++;
}
```

But with an array, you can solve this problem much more directly:

```
count[next]++;
```

This line of code is so short compared to the nested `if/else` construct that you might not realize at first that it does the same thing. Let's simulate exactly what happens as various values are read from the file.

When the array is constructed, all of the counters are initialized to `0`:



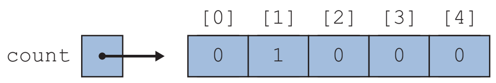
The first value in the input file is a `1`, so the program reads that into `next`. Then it executes this line of code:

```
count[next]++;
```

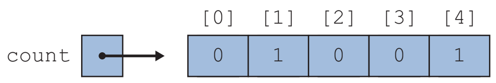
Because `next` is `1`, this line of code becomes

```
count[1]++;
```

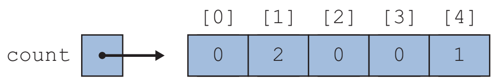
So the counter at index `[1]` is incremented:



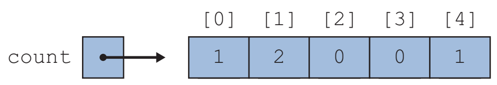
Then a `4` is read from the input file, which means `count[4]` is incremented:



Next, another `1` is read from the input file, which increments `count[1]`:

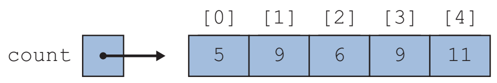


Then a `0` is read from the input file, which increments `count[0]`:



Notice that in just this short set of data you've jumped from index 1 to index 4, then back down to index 1, then to index 0. The program continues executing in this manner, jumping from counter to counter as it reads values from the file. This ability to jump around in the data structure is what's meant by random access.

After processing all of the data, the array ends up looking like this:



After this loop finishes executing, you can report the total for each score by using the standard traversing loop with a `println`:

```
for (int i = 0; i < count.length; i++) {
    System.out.println(i + "\t" + count[i]);
}
```

With the addition of a header for the output, the complete program is as follows:

```
// Reads a series of values and reports the frequency of
// occurrence of each value.

import java.io.*;
import java.util.*;

public class Tally {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner input = new Scanner(new File("tally.dat"));
        int[] count = new int[5];
        while (input.hasNextInt()) {
            int next = input.nextInt();
            count[next]++;
        }
        System.out.println("Value\tOccurrences");
        for (int i = 0; i < count.length; i++) {
            System.out.println(i + "\t" + count[i]);
        }
    }
}
```

Given the sample input file shown earlier, this program produces the following output:

```
Value   Occurrences
0       5
1       9
2       6
3       9
4       11
```

It is important to realize that a program written with an array is much more flexible than programs written with simple variables and `if/else` statements. For example, suppose you wanted to adapt this program to process an input file with exam scores that range from 0 to 100. The only change you would have to make would be to allocate a larger array:

```
int[] count = new int[101];
```

If you had written the program with an `if/else` approach, you would have to add 96 new branches to account for the new range of values. When you use an array solution, you just have to modify the overall size of the array. Notice that the array size is one more than the highest score (101 rather than 100) because the array is zero-based and because you can actually get 101 different scores on the test, including 0 as a possibility.

Completing the Program

Now that we've explored the basic approach to tallying, we can fairly easily adapt it to the problem of analyzing a data file to find the distribution of leading digits. As we stated earlier, we're assuming that we have a file of integers. To count the leading digits, we will need to be able to get the leading digit of each. This task is specialized enough that it deserves to be in its own method.

So let's first write a method called `firstDigit` that returns the first digit of an integer. If the number is a one-digit number, then the number itself will be the answer. If the number is not a one-digit number, then we can chop off its last digit because we don't need it. If we do the chopping in a loop, then eventually we'll get down to a one-digit number (the first digit). This leads us to write the following loop:

```
while (result >= 10) {
    result = result / 10;
}
```

We don't expect to get any negative numbers, but it's not a bad idea to make sure we don't have any negatives. So putting this into a method that also handles negatives, we get the following code:

```
public static int firstDigit(int n) {
    int result = Math.abs(n);
    while (result >= 10) {
        result = result / 10;
    }
    return result;
}
```

In the previous section we explored the general approach to tallying. In this case we want to tally the digits 0 through 9, so we want an array of length 10. Otherwise the solution is nearly identical to what we did in the last section. We can put the tallying code into a method that constructs an array and returns the tally:

```
public static int[] countDigits(Scanner input) {
    int[] count = new int[10];
    while (input.hasNextInt()) {
        int n = input.nextInt();
        count[firstDigit(n)]++;
    }
    return count;
}
```

Notice that instead of tallying `n` in the body of the loop, we are instead tallying `firstDigit(n)` (just the first digit, not the entire number).

The value `0` presents a potential problem for us. Benford's Law is meant to apply to data that comes from an exponential sequence. But even if you are increasing exponentially, if you start with `0`, you never get beyond `0`. As a result, it is best to eliminate the `0` values from the calculation. Often they won't occur at all.

When reporting results, then, let's begin by reporting the excluded zeros if they exist:

```
if (count[0] > 0) {
    System.out.println("excluding " + count[0] + " zeros");
}
```

For the other digits, we want to report the number of occurrences of each and also the percentage of each. To figure the percentage, we'll need to know the sum of the values. This is a good place to introduce a method that finds the sum of an array of integers. It's a fairly straightforward array traversal problem that can be solved with a for-each loop:

```
public static int sum(int[] data) {
    int sum = 0;
    for (int n : data) {
        sum += n;
    }
    return sum;
}
```

Now we can compute the total number of digits by calling the method and subtracting the number of `0`s:

```
int total = sum(count) - count[0];
```

And once we have the total number of digits, we can write a loop to report each of the percentages. To compute the percentages, we multiply each count by 100 and divide by the total number of digits. We have to be careful to multiply by 100.0 rather than 100 to make sure that we are computing the result using `double` values. Otherwise we'll get truncated integer division and won't get any digits after the decimal point:

```
for (int i = 1; i < count.length; i++) {
    double pct = count[i] * 100.0 / total;
    System.out.println(i + " " + count[i] + " " + pct);
}
```

Notice that the loop starts at `1` instead of `0` because we have excluded the zeros from our reporting.

Here is a complete program that puts these pieces together. It also uses `printf` statements to format the output and includes a header for the table and a total afterward:

```
// This program finds the distribution of leading digits in a set
// of positive integers.  The program is useful for exploring the
// phenomenon known as Benford's Law.

import java.io.*;
import java.util.*;

public class Benford {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner console = new Scanner(System.in);
        System.out.println("Let's count those leading digits...");
        System.out.print("input file name? ");
        String name = console.nextLine();
        Scanner input = new Scanner(new File(name));
        int[] count = countDigits(input);
        reportResults(count);
    }

    // Reads integers from input, computing an array of counts
    // for the occurrences of each leading digit (0-9).
    public static int[] countDigits(Scanner input) {
        int[] count = new int[10];
        while (input.hasNextInt()) {
            int n = input.nextInt();
            count[firstDigit(n)]++;
        }
        return count;
    }

    // Reports percentages for each leading digit, excluding zeros
    public static void reportResults(int[] count) {
        System.out.println();
        if (count[0] > 0) {
            System.out.println("excluding " + count[0] + " zeros");
        }
        int total = sum(count) - count[0];
        System.out.println("Digit Count Percent");
        for (int i = 1; i < count.length; i++) {
            double pct = count[i] * 100.0 / total;
            System.out.printf("%5d %5d %6.2f\n", i, count[i], pct);
        }
        System.out.printf("Total %5d %6.2f\n", total, 100.0);
    }

    // returns the sum of the integers in the given array
    public static int sum(int[] data) {
        int sum = 0;
        for (int n : data) {
            sum += n;
        }
        return sum;
    }

    // returns the first digit of the given number
    public static int firstDigit(int n) {
        int result = Math.abs(n);
        while (result >= 10) {
            result = result / 10;
        }
        return result;
    }
}
```

Now that we have a complete program, let's see what we get when we analyze various data sets. The Benford distribution shows up with population data because population tends to grow exponentially. Let's use data from the web page http://www.census.gov/popest/ which contains population estimates for various U.S. counties. The data set has information on 3000 different counties with populations varying from 100 individuals to over 9 million for the census year 2000. Here is a sample output of our program using these data:

```
Let's count those leading digits...
input file name? county.txt

Digit   Count  Percent
    1     970   30.90
    2     564   17.97
    3     399   12.71
    4     306    9.75
    5     206    6.56
    6     208    6.63
    7     170    5.24
    8     172    5.48
    9     144    4.59
Total    3139  100.00
```

These percentages are almost exactly the numbers predicted by Benford's Law.

Data that obey Benford's Law have an interesting property. It doesn't matter what scale you use for the data. So if you are measuring heights, for example, it doesn't matter whether you measure in feet, inches, meters, or furlongs. In our case, we counted the number of people in each U.S. county. If we instead count the number of human hands in each county, then we have to double each number. Look at the preceding output and see if you can predict the result when you double each number. Here is the actual result:

```
Let's count those leading digits...
input file name? county2.txt

Digit   Count  Percent
    1     900   28.67
    2     555   17.68
    3     415   13.22
    4     322   10.26
    5     242    7.71
    6     209    6.66
    7     190    6.05
    8     173    5.51
    9     133    4.24
Total    3139  100.00
```

Notice that there is very little change. Doubling the numbers has little effect because if the original data is exponential in nature, then the same will be true of the doubled numbers. Here is another sample run that triples the county population numbers:

```
Let's count those leading digits...
input file name? county3.txt

Digit   Count  Percent
    1     926   29.50
    2     549   17.49
    3     385   12.27
    4     327   10.42
    5     258    8.22
    6     228    7.26
    7     193    6.15
    8     143    4.56
    9     130    4.14
Total    3139  100.00
```

Another data set that shows Benford characteristics is the count of sunspots that occur on any given day. Robin McQuinn maintains a web page at http://sidc.oma.be/html/sunspot.html that has daily counts of sunspots going back to 1818. Here is a sample execution using these data:

```
Let's count those leading digits...
input file name? sunspot.txt

excluding 4144 zeros
Digit   Count  Percent
    1    5405   31.24
    2    1809   10.46
    3    2127   12.29
    4    1690    9.77
    5    1702    9.84
    6    1357    7.84
    7    1364    7.88
    8     966    5.58
    9     882    5.10
Total   17302  100.00
```

Notice that on this execution the program reports the exclusion of some `0` values.

Chapter Summary

An array is an object that groups multiple primitive values or objects of the same type under one name. Each individual value, called an element, is accessed with an integer index that ranges from 0 to one less than the array's length.

Attempting to access an array element with an index of less than 0 or one that is greater than or equal to the array's length will cause the program to crash with an `ArrayIndexOutOfBoundsException`.

Arrays are often traversed using `for` loops. The length of an array is found by accessing its length field, so the loop over an array can process indexes from 0 to `length - 1`. Array elements can also be accessed in order using a type of loop called a for-each loop.

Arrays have several limitations, such as fixed size and lack of support for common operations like == and `println`. To perform these operations, you must either use the `Arrays` class or write `for` loops that process each element of the array.

Several common array algorithms, such as printing an array or comparing two arrays to each other for equality, are implemented by traversing the elements and examining or modifying each one.

Java arrays are objects and use reference semantics, in which variables store references to values rather than to the actual values themselves. This means that two variables can refer to the same array or object. If the array is modified through one of its references, the modification will also be seen in the other.

Arrays of objects are actually arrays of references to objects. A newly declared and initialized array of objects actually stores `null` in all of its element indexes, so each element must be initialized individually or in a loop to store an actual object.

A multidimensional array is an array of arrays. These are often used to store two-dimensional data, such as data in rows and columns or *xy* data in a two-dimensional space.

Self-Check Problems

Section 7.1: Array Basics

1. Which of the following is the correct syntax to declare an array of ten integers?

a. `int a[10] = new int[10];`

b. `int[10] a = new int[10];`

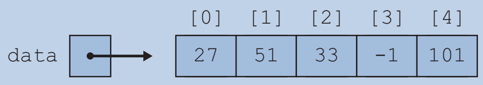
c. `[]int a = [10]int;`

d. `int a[10];`

e. `int[] a = new int[10];`

2. What expression should be used to access the first element of an array of integers called `numbers`? What expression should be used to access the last element of `numbers`, assuming it contains 10 elements? What expression can be used to access its last element, regardless of its length?

3. Write code that creates an array of integers named `data` of size 5 with the following contents:



4. Write code that stores all odd numbers between −6 and 38 into an array using a loop. Make the array's size exactly large enough to store the numbers.

Then, try generalizing your code so that it will work for any minimum and maximum values, not just −6 and 38.

5. What elements does the array `numbers` contain after the following code is executed?

```
int[] numbers = new int[8];
numbers[1] = 4;
numbers[4] = 99;
numbers[7] = 2;

int x = numbers[1];
numbers[x] = 44;
numbers[numbers[7]] = 11; // uses numbers[7] as index
```

6. What elements does the array `data` contain after the following code is executed?

```
int[] data = new int[8];
data[0] = 3;
data[7] = -18;
data[4] = 5;
data[1] = data[0];

int x = data[4];
data[4] = 6;
data[x] = data[0] * data[1];
```

7. What is wrong with the following code?

```
int[] first = new int[2];
first[0] = 3;
first[1] = 7;
int[] second = new int[2];
second[0] = 3;
second[1] = 7;

// print the array elements
System.out.println(first);
System.out.println(second);

// see if the elements are the same
if (first == second) {
    System.out.println("They contain the same elements.");
} else {
    System.out.println("The elements are different.");
}
```

8. Which of the following is the correct syntax to declare an array of the given six integer values?
   a. `int[] a = {17, -3, 42, 5, 9, 28};`
   b. `int a {17, -3, 42, 5, 9, 28};`
   c. `int[] a = new int[6] {17, -3, 42, 5, 9, 28};`
   d. `int[6] a = {17, -3, 42, 5, 9, 28};`
   e. `int[] a = int [17, -3, 42, 5, 9, 28] {6};`

9. Write a piece of code that declares an array called `data` with the elements `7`, `-1`, `13`, `24`, and `6`. Use only one statement to initialize the array.

10. Write a piece of code that examines an array of integers and reports the maximum value in the array. Consider putting your code into a method called `max` that accepts the array as a parameter and returns the maximum value. Assume that the array contains at least one element.

11. Write a method called `average` that computes the average (arithmetic mean) of all elements in an array of integers and returns the answer as a `double`. For example, if the array passed contains the values `[1, -2, 4, -4, 9, -6, 16, -8, 25, -10]`, the calculated average should be `2.5`. Your method accepts an array of integers as its parameter and returns the average.

Section 7.2: Array-Traversal Algorithms

12. What is an array traversal? Give an example of a problem that can be solved by traversing an array.

13. Write code that uses a `for` loop to print each element of an array named `data` that contains five integers:

```
element [0] is 14
element [1] is 5
element [2] is 27
element [3] is -3
element [4] is 2598
```

Consider generalizing your code so that it will work on an array of any size.

14. What elements does the array `list` contain after the following code is executed?

```
int[] list = {2, 18, 6, -4, 5, 1};
for (int i = 0; i < list.length; i++) {
    list[i] = list[i] + (list[i] / list[0]);
}
```

15. Write a piece of code that prints an array of integers in reverse order, in the same format as the `print` method from Section 7.2. Consider putting your code into a method called `printBackwards` that accepts the array as a parameter.

16. Describe the modifications that would be necessary to change the `count` and `equals` methods developed in Section 7.2 to process arrays of `String`s instead of arrays of integers.

17. Write a method called `allLess` that accepts two arrays of integers and returns `true` if each element in the first array is less than the element at the same index in the second array. Your method should return `false` if the arrays are not the same length.

Section 7.3: Reference Semantics

18. Why does a method to swap two array elements work correctly when a method to swap two integer values does not?

19. What is the output of the following program?

```
public class ReferenceMystery1 {
    public static void main(String[] args) {
        int x = 0;
        int[] a = new int[4];
        x = x + 1;
        mystery(x, a);
        System.out.println(x + " " + Arrays.toString(a));
        x = x + 1;
        mystery(x, a);
        System.out.println(x + " " + Arrays.toString(a));
    }
    public static void mystery(int x, int[] a) {
        x = x + 1;
        a[x] = a[x] + 1;
        System.out.println(x + " " + Arrays.toString(a));
    }
}
```

20. What is the output of the following program?

```
public class ReferenceMystery2 {
    public static void main(String[] args) {
        int x = 1;
        int[] a = new int[2];
        mystery(x, a);
        System.out.println(x + " " + Arrays.toString(a));
        x--;
        a[1] = a.length;
        mystery(x, a);
        System.out.println(x + " " + Arrays.toString(a));
    }

    public static void mystery(int x, int[] list) {
        list[x]++;
        x++;
        System.out.println(x + " " + Arrays.toString(list));
    }
}
```

21. Write a method called `swapPairs` that accepts an array of integers and swaps the elements at adjacent indexes. That is, elements 0 and 1 are swapped, elements 2 and 3 are swapped, and so on. If the array has an odd length, the final element should be left unmodified. For example, the list `[10, 20, 30, 40, 50]` should become `[20, 10, 40, 30, 50]` after a call to your method.

Section 7.4: Advanced Array Techniques

22. What are the values of the elements in the array `numbers` after the following code is executed?

```
int[] numbers = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
for (int i = 0; i < 9; i++) {
    numbers[i] = numbers[i + 1];
}
```

23. What are the values of the elements in the array `numbers` after the following code is executed?

```
int[] numbers = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
for (int i = 1; i < 10; i++) {
    numbers[i] = numbers[i - 1];
}
```

24. Consider the following method, `mystery`:

```
public static void mystery(int[] a, int[] b) {
    for (int i = 0; i < a.length; i++) {
        a[i] += b[b.length - 1 - i];
    }
}
```

What are the values of the elements in array `a1` after the following code executes?

```
int[] a1 = {1, 3, 5, 7, 9};
int[] a2 = {1, 4, 9, 16, 25};
mystery(a1, a2);
```

25. Consider the following method, `mystery2`:

```
public static void mystery2(int[] a, int[] b) {
    for (int i = 0; i < a.length; i++) {
        a[i] = a[2 * i % a.length] - b[3 * i % b.length];
    }
}
```

What are the values of the elements in array `a1` after the following code executes?

```
int[] a1 = {2, 4, 6, 8, 10, 12, 14, 16};
int[] a2 = {1, 1, 2, 3, 5, 8, 13, 21};
mystery2(a1, a2);
```

26. Consider the following method, `mystery3`:

```
public static void mystery3(int[] data, int x, int y) {
    data[data[x]] = data[y];
    data[y] = x;
}
```

What are the values of the elements in the array `numbers` after the following code executes?

```
int[] numbers = {3, 7, 1, 0, 25, 4, 18, -1, 5};
mystery3(numbers, 3, 1);
```

```
mystery3(numbers, 5, 6);
mystery3(numbers, 8, 4);
```

27. Consider the following method:

```
public static int mystery4(int[] list) {
    int x = 0;
    for (int i = 1; i < list.length; i++) {
        int y = list[i] - list[0];
        if (y > x) {
            x = y;
        }
    }
    return x;
}
```

What value does the method return when passed each of the following arrays?

a. `{5}`

b. `{3, 12}`

c. `{4, 2, 10, 8}`

d. `{1, 9, 3, 5, 7}`

e. `{8, 2, 10, 4, 10, 9}`

28. Consider the following method:

```
public static void mystery5(int[] nums) {
    for (int i = 0; i < nums.length - 1; i++) {
        if (nums[i] > nums[i + 1]) {
            nums[i + 1]++;
        }
    }
}
```

What are the final contents of each of the following arrays if each is passed to the above method?

a. `{8}`

b. `{14, 7}`

c. `{7, 1, 3, 2, 0, 4}`

d. `{10, 8, 9, 5, 5}`

e. `{12, 11, 10, 10, 8, 7}`

29. Write a piece of code that computes the average `String` length of the elements of an array of `Strings`. For example, if the array contains `{"belt", "hat", "jelly", "bubble gum"}`, the average length is `5.5`.

30. Write code that accepts an array of `String`s as its parameter and indicates whether that array is a palindrome—that is, whether it reads the same forward as backward. For example, the array `{"alpha", "beta", "gamma", "delta", "gamma", "beta", "alpha"}` is a palindrome.

Section 7.5: Multidimensional Arrays

31. What elements does the array `numbers` contain after the following code is executed?

```
int[][] numbers = new int[3][4];
for (int r = 0; r < numbers.length; r++) {
    for (int c = 0; c < numbers[0].length; c++) {
        numbers[r][c] = r + c;
    }
}
```

32. Assume that a two-dimensional rectangular array of integers called `data` has been declared with four rows and seven columns. Write a loop to initialize the third row of data to store the numbers 1 through 7.

33. Write a piece of code that constructs a two-dimensional array of integers with 5 rows and 10 columns. Fill the array with a multiplication table, so that array element `[i][j]` contains the value `i * j`. Use nested `for` loops to build the array.

34. Assume that a two-dimensional rectangular array of integers called `matrix` has been declared with six rows and eight columns. Write a loop to copy the contents of the second column into the fifth column.

35. Consider the following method:

```
public static void mystery2d(int[][] a) {
    for (int r = 0; r < a.length; r++) {
        for (int c = 0; c < a[0].length - 1; c++) {
            if (a[r][c + 1] > a[r][c]) {
                a[r][c] = a[r][c + 1];
            }
        }
    }
}
```

If a two-dimensional array `numbers` is initialized to store the following integers, what are its contents after the call shown?

```
int[][] numbers = {{3, 4, 5, 6},
                   {4, 5, 6, 7},
                   {5, 6, 7, 8}};
mystery2d(numbers);
```

36. Write a piece of code that constructs a jagged two-dimensional array of integers with five rows and an increasing number of columns in each row, such that the first row has one column, the second row has two, the third has three, and so on. The array elements should have increasing values in top-to-bottom, left-to-right order (also called row-major order). In other words, the array's contents should be the following:

```
1
2, 3
4, 5, 6
7, 8, 9, 10
11, 12, 13, 14, 15
```

Use nested `for` loops to build the array.

37. When examining a 2D array of pixels, how could you figure out the width and height of the image even if you don't have access to the `DrawingPanel` object?

38. Finish the following code for a method that converts an image into its red channel; that is, removing any green or blue from each pixel and keeping only the red component.

```
public static void toRedChannel(DrawingPanel panel) {
    Color[][] pixels = panel.getPixels();
    for (int row = 0; row < pixels.length; row++) {
        for (int col = 0; col < pixels[0].length; col++) {
            // your code goes here
        }
    }
    panel.setPixels(pixels);
}
```

39. What is the result of the following code? What will the image look like?

```
public static void pixelMystery(DrawingPanel panel) {
    Color[][] pixels = panel.getPixels();
    for (int row = 0; row < pixels.length; row++) {
        for (int col = 0; col < pixels[0].length; col++) {
            int n = Math.min(row + col, 255);
            pixels[row][col] = new Color(n, n, n);
        }
    }
    panel.setPixels(pixels);
}
```

Exercises

1. Write a method called `lastIndexOf` that accepts an array of integers and an integer value as its parameters and returns the last index at which the value occurs in the array. The method should return `-1` if the value is not found. For example, in the array [`74, 85, 102, 99, 101, 85, 56`], the last index of the value `85` is 5.
2. Write a method called `range` that returns the range of values in an array of integers. The range is defined as 1 more than the difference between the maximum and minimum values in the array. For example, if an array called `list` contains the values `[36, 12, 25, 19, 46, 31, 22]`, the call of `range(list)` should return `35` (46 − 12 + 1). You may assume that the array has at least one element.
3. Write a method called `countInRange` that accepts an array of integers, a minimum value, and a maximum value as parameters and returns the count of how many elements from the array fall between the minimum and maximum (inclusive). For example, in the array `[14, 1, 22, 17, 36, 7, -43, 5]`, for minimum value `4` and maximum value `17`, there are four elements whose values fall between `4` and `17`.
4. Write a method called `isSorted` that accepts an array of real numbers as a parameter and returns `true` if the list is in sorted (nondecreasing) order and `false` otherwise. For example, if arrays named `list1` and `list2` store `[16.1, 12.3, 22.2, 14.4]` and `[1.5, 4.3, 7.0, 19.5, 25.1, 46.2]` respectively, the calls `isSorted(list1)` and `isSorted(list2)` should return `false` and `true` respectively. Assume the array has at least one element. A one-element array is considered to be sorted.
5. Write a method called `mode` that returns the most frequently occurring element of an array of integers. Assume that the array has at least one element and that every element in the array has a value between `0` and `100` inclusive. Break ties by choosing the lower value. For example, if the array passed contains the values `[27, 15, 15, 11, 27]`,

your method should return 15. (*Hint*: You may wish to look at the Tally program from this chapter to get an idea how to solve this problem.) Can you write a version of this method that does not rely on the values being between 0 and 100?

6. Write a method called stdev that returns the standard deviation of an array of integers. Standard deviation is computed by taking the square root of the sum of the squares of the differences between each element and the mean, divided by one less than the number of elements. (It's just that simple!) More concisely and mathematically, the standard deviation of an array *a* is written as follows:

$$stdev(a) = \sqrt{\frac{\sum_{i=0}^{a.length-1} (a[i] - average(a)^2)}{a.length - 1}}$$

For example, if the array passed contains the values [1, -2, 4, -4, 9, -6, 16, -8, 25, -10], your method should return approximately 11.237.

7. Write a method called kthLargest that accepts an integer *k* and an array *a* as its parameters and returns the element such that *k* elements have greater or equal value. If *k* = 0, return the largest element; if *k* = 1, return the second-largest element, and so on. For example, if the array passed contains the values [74, 85, 102, 99, 101, 56, 84] and the integer *k* passed is 2, your method should return 99 because there are two values at least as large as 99 (101 and 102). Assume that 0 ≤ *k* < *a*.length. (*Hint:* Consider sorting the array or a copy of the array first.)

8. Write a method called median that accepts an array of integers as its parameter and returns the median of the numbers in the array. The median is the number that appears in the middle of the list if you arrange the elements in order. Assume that the array is of odd size (so that one sole element constitutes the median) and that the numbers in the array are between 0 and 99 inclusive. For example, the median of [5, 2, 4, 17, 55, 4, 3, 26, 18, 2, 17] is 5 and the median of [42, 37, 1, 97, 1, 2, 7, 42, 3, 25, 89, 15, 10, 29, 27] is 25. (*Hint*: You may wish to look at the Tally program from earlier in this chapter for ideas.)

9. Write a method called minGap that accepts an integer array as a parameter and returns the minimum difference or gap between adjacent values in the array, where the gap is defined as the later value minus the earlier value. For example, in the array [1, 3, 6, 7, 12], the first gap is 2 $(3 - 1)$, the second gap is 3 $(6 - 3)$, the third gap is 1 $(7 - 6)$, and the fourth gap is 5 $(12 - 7)$. So your method should return 1 if passed this array. The minimum gap could be a negative number if the list is not in sorted order. If you are passed an array with fewer than two elements, return 0.

10. Write a method called percentEven that accepts an array of integers as a parameter and returns the percentage of even numbers in the array as a real number. For example, if the array stores the elements [6, 2, 9, 11, 3], then your method should return 40.0. If the array contains no even elements or no elements at all, return 0.0.

11. Write a method called isUnique that accepts an array of integers as a parameter and returns a boolean value indicating whether or not the values in the array are unique (true for yes, false for no). The values in the list are considered unique if there is no pair of values that are equal. For example, if passed an array containing [3, 8, 12, 2, 9, 17, 43, -8, 46], your method should return true, but if passed [4, 7, 3, 9, 12, -47, 3, 74], your method should return false because the value 3 appears twice.

12. Write a method called priceIsRight that mimics the guessing rules from the game show *The Price Is Right*. The method accepts as parameters an array of integers representing the contestants' bids and an integer representing a correct price. The method returns the element in the bids array that is closest in value to the correct price without being larger than that price. For example, if an array called `bids` stores the values `[200, 300, 250, 1, 950, 40]`, the call of `priceIsRight(bids, 280)` should return `250`, since 250 is the bid closest to 280 without going over 280. If all bids are larger than the correct price, your method should return `-1`.

13. Write a method called `longestSortedSequence` that accepts an array of integers as a parameter and returns the length of the longest sorted (nondecreasing) sequence of integers in the array. For example, in the array `[3, 8, 10, 1, 9, 14, -3, 0, 14, 207, 56, 98, 12]`, the longest sorted sequence in the array has four values in it (the sequence −3, 0, 14, 207), so your method would return `4` if passed this array. Sorted means nondecreasing, so a sequence could contain duplicates. Your method should return `0` if passed an empty array.

14. Write a method called `contains` that accepts two arrays of integers *a1* and *a2* as parameters and that returns a `boolean` value indicating whether or not the sequence of elements in *a2* appears in *a1* (`true` for yes, `false` for no). The sequence must appear consecutively and in the same order. For example, consider the following arrays:

```
int[] list1 = {1, 6, 2, 1, 4, 1, 2, 1, 8};
int[] list2 = {1, 2, 1};
```

The call of `contains(list1, list2)` should return `true` because the sequence of values in `list2 [1, 2, 1]` is contained in `list1` starting at index 5. If `list2` had stored the values `[2, 1, 2]`, the call of `contains(list1, list2)` would return `false`. Any two lists with identical elements are considered to contain each other. Every array contains the empty array, and the empty array does not contain any arrays other than the empty array itself.

15. Write a method called `collapse` that accepts an array of integers as a parameter and returns a new array containing the result of replacing each pair of integers with the sum of that pair. For example, if an array called `list` stores the values `[7, 2, 8, 9, 4, 13, 7, 1, 9, 10]`, then the call of `collapse(list)` should return a new array containing `[9, 17, 17, 8, 19]`. The first pair from the original list is collapsed into 9 $(7 + 2)$, the second pair is collapsed into 17 $(8 + 9)$, and so on. If the list stores an odd number of elements, the final element is not collapsed. For example, if the list had been `[1, 2, 3, 4, 5]`, then the call would return `[3, 7, 5]`. Your method should not change the array that is passed as a parameter.

16. Write a method called `append` that accepts two integer arrays as parameters and returns a new array that contains the result of appending the second array's values at the end of the first array. For example, if arrays `list1` and `list2` store `[2, 4, 6]` and `[1, 2, 3, 4, 5]` respectively, the call of `append(list1, list2)` should return a new array containing `[2, 4, 6, 1, 2, 3, 4, 5]`. If the call instead had been `append(list2, list1)`, the method would return an array containing `[1, 2, 3, 4, 5, 2, 4, 6]`.

17. Write a method called `vowelCount` that accepts a `String` as a parameter and produces and returns an array of integers representing the counts of each vowel in the string. The array returned by your method should hold five elements: the first is the count of As, the second is the count of Es, the third Is, the fourth Os, and the fifth Us. Assume that the string contains no uppercase letters. For example, the call `vowelCount("i think, therefore i am")` should return the array `[1, 3, 3, 1, 0]`.

18. Write a method called `evenBeforeOdd` that accepts an array of integers and rearranges its elements so that all even values appear before all odds. For example, if the array is `[5, 4, 2, 11, 9, 10, 4, 7, 3]`, then after the method has been called, one acceptable ordering of the elements would be `[4, 2, 10, 4, 5, 11, 9, 7, 3]`. The exact order of the elements does not matter, so long as all even values appear before all odd values. The array might contain no even elements or no odd elements. Do not use any temporary arrays in your solution, and do not call `Arrays.sort`.

19. Write a method called `wordLengths` that accepts a `Scanner` for an input file as its parameter. Your method should open the given file, count the number of letters in each token in the file, and output a result diagram of how many words contain each number of letters. For example, consider a file containing the following text:

```
Before sorting:
13 23 480 -18 75
hello how are you feeling today

After sorting:
-18 13 23 75 480
are feeling hello how today you
```

Your method should produce the following output to the console. Use tabs so that the stars line up:

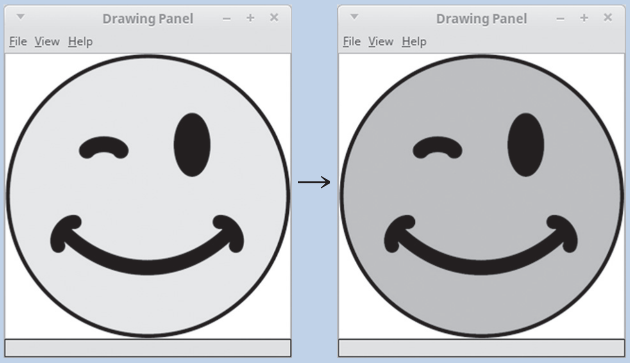
```
1: 0
2: 6 ******
3: 10 **********
4: 0
5: 5 *****
6: 1 *
7: 2 **
8: 2 **
```

Assume that no token in the file is more than 80 characters in length.

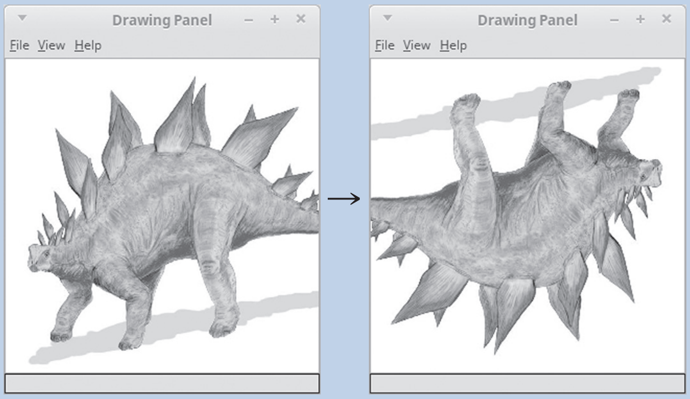
20. Write a method called `matrixAdd` that accepts a pair of two-dimensional arrays of integers as parameters, treats the arrays as two-dimensional matrixes, and returns their sum. The sum of two matrixes A and B is a matrix C, where for every row `i` and column `j`, $C_{ij} = A_{ij} + B_{ij}$. You may assume that the arrays passed as parameters have the same dimensions.

21. Write a method called `isMagicSquare` that accepts a two-dimensional array of integers as a parameter and returns `true` if it is a magic square. A square matrix is a *magic square* if all of its row, column, and diagonal sums are equal. For example, `[[2, 7, 6], [9, 5, 1], [4, 3, 8]]` is a square matrix because all eight of the sums are exactly 15.

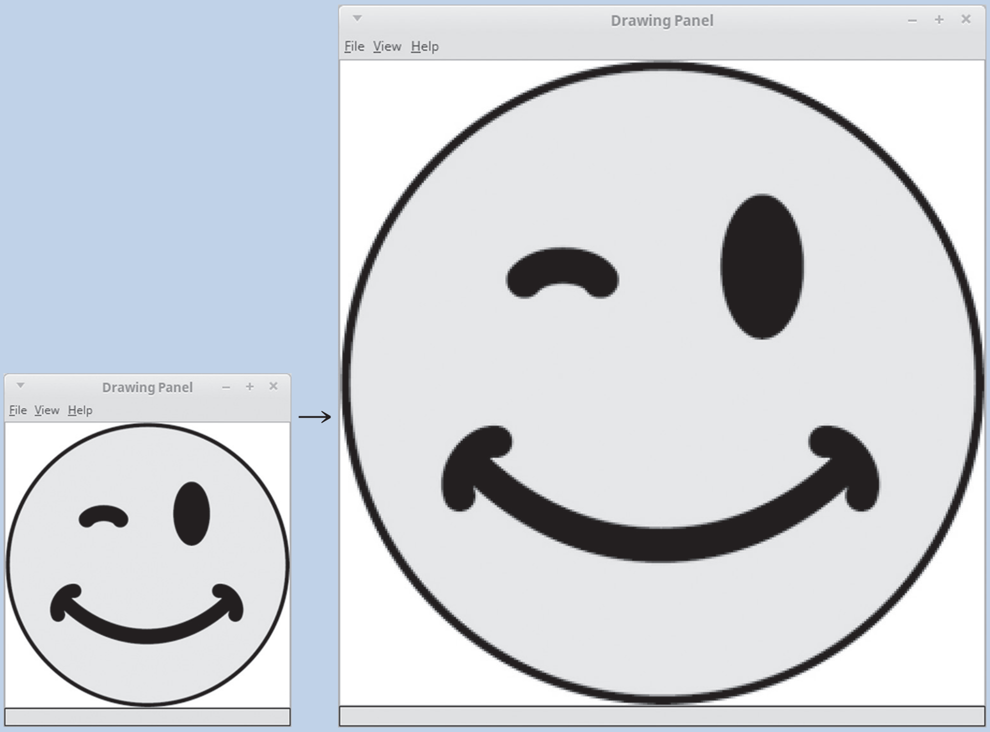
22. Write a method `grayscale` that converts a color image into a black-and-white image. This is done by averaging the red, green, and blue components of each pixel. For example, if a pixel has RGB values of (red $=$ 100, green $=$ 30 blue $=$ 80), the average of the three components is $(100 + 30 + 80)/3 = 70$, so that pixel becomes (red $=$ 70, green $=$ 70, blue $=$ 70).



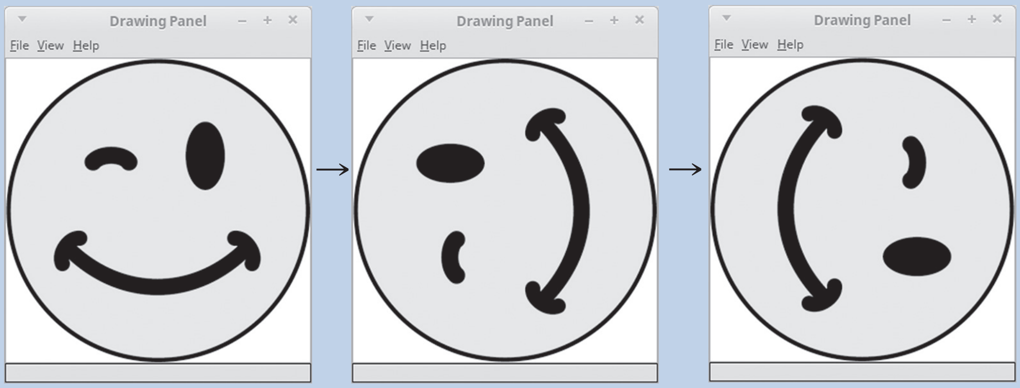
23. Write a method `transpose` that accepts a `DrawingPanel` as a parameter and inverts the image about both the *x* and *y* axes. You may assume that the image is square, that is, that its width and height are equal.



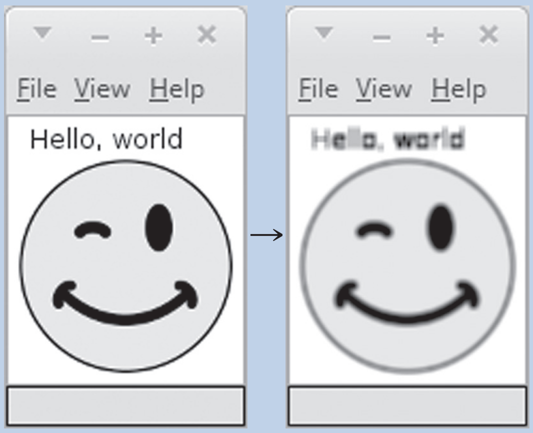
24. Write a method `zoomIn` that accepts a `DrawingPanel` as a parameter and converts it into an image twice as large in both dimensions. Each pixel from the original image becomes a cluster of 4 pixels (2 rows and 2 columns) in the new zoomed image.



25. Write methods `rotateLeft` and `rotateRight` that rotate the pixels of an image counter-clockwise or clockwise by 90 degrees respectively. You should not assume that the image is square in shape; its width and height might be different.



26. Write a method `blur` that makes an image look "blurry" using the following specific algorithm. Set each pixel to be the average of itself and the 8 pixels around it. That is, for the pixel at position (x, y), set its RGB value to be the average of the RGB values at positions ($x - 1$, $y - 1$) through ($x + 1$, $y + 1$). Be careful not to go out of bounds near the edge of the image; if a pixel lies along the edge of the image, average whatever neighbors it does have.



Programming Projects

1. Java's type `int` has a limit on how large an integer it can store. This limit can be circumvented by representing an integer as an array of digits. Write an interactive program that adds two integers of up to 50 digits each.

2. Write a game of Hangman using arrays. Allow the user to guess letters and represent which letters have been guessed in an array.

3. Write a program that plays a variation of the game of Mastermind with a user. For example, the program can use pseudorandom numbers to generate a four-digit number. The user should be allowed to make guesses until she gets the number correct. Clues should be given to the user indicating how many digits of the guess are correct and in the correct place and how many digits are correct but in the wrong place.

4. Write a program to score users' responses to the classic Myers–Briggs personality test. Assume that the test has 70 questions that determine a person's personality in four dimensions. Each question has two answer choices that we'll call the "A" and "B" answers. Questions are organized into 10 groups of seven questions, with the following repeating pattern in each group:

- The first question in each group (questions 1, 8, 15, 22, etc.) tells whether the person is introverted or extroverted.
- The next two questions (questions 2 and 3, 9 and 10, 16 and 17, 23 and 24, etc.) test whether the person is guided by his or her senses or intuition.
- The next two questions (questions 4 and 5, 11 and 12, 18 and 19, 25 and 26, etc.) test whether the person focuses on thinking or feeling.
- The final two questions in each group (questions 6 and 7, 13 and 14, 20 and 21, 27 and 28, etc.) test whether the person prefers to judge or be guided by perception.

In other words, if we consider introversion/extraversion (I/E) to be dimension 1, sensing/intuition (S/N) to be dimension 2, thinking/feeling (T/F) to be dimension 3, and judging/perception (J/P) to be dimension 4, the map of questions to their respective dimensions would look like this:

```
1223344122334412233441223344122334412233441223344122334412233441223344
BABAAAABAAAAAAAABAAAABBAAAAAAABAAAABABAABAAABABABAABAAAAAAABAAAAAAABAAAAAA
```

The following is a partial sample input file of names and responses:

```
Betty Boop
BABAAAABAAAAAAAABAAAABBAAAAAAABAAAABABAABAAABABABAABAAAAAAABAAAAAAABAAAAAA
Snoopy
AABBAABBBBBABABAAAAABABBAABBAAAABBBAAABAABAABABAAAABAABBBBAAABBAABABBB
```

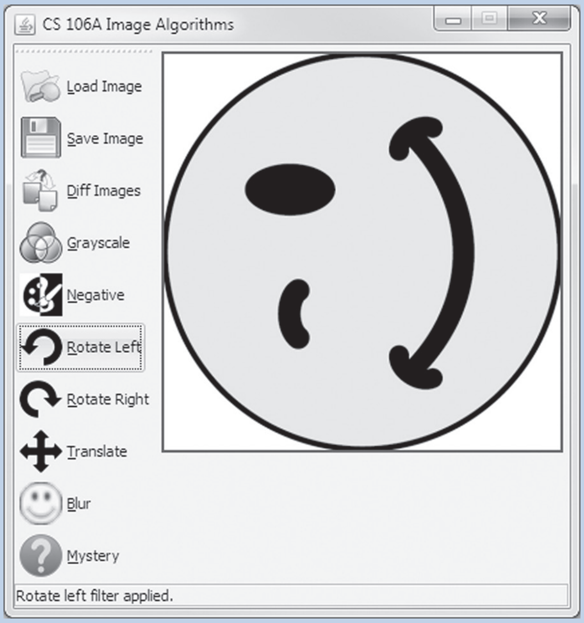
If less than 50% of a person's responses are B for a given personality dimension, the person's type for that dimension should be the first of its two choices. If the person has 50% or more B responses, the person's type for that dimension is the second choice. Your program should output each person's name, the number of A and B responses for each dimension, the percentage of Bs in each dimension, and the overall personality type. The following should be your program's output for the preceding input data:

```
Betty Boop:
1A-9B 17A-3B 18A-2B 18A-2B
[90%, 15%, 10%, 10%] = ISTJ
Snoopy:
7A-3B 11A-9B 14A-6B 6A-14B
[30%, 45%, 30%, 70%] = ESTP
```

5. Use a two-dimensional array to write a game of Tic-Tac-Toe that represents the board.

6. Write a program that reads a file of DNA data and searches for protein sequences. DNA data consists of long `Strings` of the letters A, C, G, and T, corresponding to chemical nucleotides called adenine, cytosine, guanine, and thymine. Proteins can be identified by looking for special triplet sequences of nucleotides that indicate the start and stop of a protein range. Store relevant data in arrays as you make your computation. See our textbook's web site for example DNA input files and more details about heuristics for identifying proteins.

7. Write a basic Photoshop or Instagram-inspired program with a menu of available image manipulation algorithms similar to those described in the exercises in this chapter. The user can load an image from a file and then select which manipulation to perform, such as grayscale, zoom, rotate, or blur.



Rotate left filter applied.