

Chapter 7

Arrays

COMPUTERS GET A LOT OF THEIR POWER from working with *data structures*. A data structure is an organized collection of related data. An object is a data structure, but this type of data structure—consisting of a fairly small number of named instance variables—is just the beginning. In many cases, programmers build complicated data structures by hand, by linking objects together. We’ll look at these custom-built data structures in Chapter 9. But there is one type of data structure that is so important and so basic that it is built into every programming language: the array.

An *array* is a data structure consisting of a numbered list of items, where all the items are of the same type. In Java, the items in an array are always numbered from zero up to some maximum value, which is set when the array is created. For example, an array might contain 100 integers, numbered from zero to 99. The items in an array can belong to one of Java’s primitive types. They can also be references to objects, so that you could, for example, make an array containing all the buttons in a GUI program.

This chapter discusses how arrays are created and used in Java. It also covers the standard class `java.util.ArrayList`. An object of type *ArrayList* is very similar to an array of *Objects*, but it can grow to hold any number of items.

7.1 Creating and Using Arrays

WHEN A NUMBER OF DATA ITEMS are chunked together into a unit, the result is a *data structure*. Data structures can be very complex, but in many applications, the appropriate data structure consists simply of a sequence of data items. Data structures of this simple variety can be either *arrays* or *records*.

The term “record” is not used in Java. A record is essentially the same as a Java object that has instance variables only, but no instance methods. Some other languages, which do not support objects in general, nevertheless do support records. The C programming language, for example, is not object-oriented, but it has records, which in C go by the name “struct.” The data items in a record—in Java, an object’s instance variables—are called the *fields* of the record. Each item is referred to using a *field name*. In Java, field names are just the names of the instance variables. The distinguishing characteristics of a record are that the data items in the record are referred to by name and that different fields in a record are allowed to be of different types. For example, if the class *Person* is defined as:

```
class Person {  
    String name;
```

```

    int id_number;
    Date birthday;
    int age;
}

```

then an object of class *Person* could be considered to be a record with four fields. The field names are *name*, *id_number*, *birthday*, and *age*. Note that the fields are of various types: *String*, *int*, and *Date*.

Because records are just a special type of object, I will not discuss them further.

7.1.1 Arrays

Like a record, an array is a sequence of items. However, where items in a record are referred to by **name**, the items in an array are numbered, and individual items are referred to by their **position number**. Furthermore, all the items in an array must be of the same type. The definition of an array is: a numbered sequence of items, which are all of the same type. The number of items in an array is called the **length** of the array. The position number of an item in an array is called the **index** of that item. The type of the individual items in an array is called the **base type** of the array.

The base type of an array can be any Java type, that is, one of the primitive types, or a class name, or an interface name. If the base type of an array is *int*, it is referred to as an “array of *ints*.” An array with base type *String* is referred to as an “array of *Strings*.” However, an array is not, properly speaking, a list of integers or strings or other **values**. It is better thought of as a list of **variables** of type *int*, or of type *String*, or of some other type. As always, there is some potential for confusion between the two uses of a variable: as a name for a memory location and as a name for the value stored in that memory location. Each position in an array acts as a variable. Each position can hold a value of a specified type (the base type of the array). The value can be changed at any time. Values are stored **in** an array. The array **is** the container, not the values.

The items in an array—really, the individual variables that make up the array—are more often referred to as the **elements** of the array. In Java, the elements in an array are always numbered starting from zero. That is, the index of the first element in the array is zero. If the length of the array is *N*, then the index of the last element in the array is *N*-1. Once an array has been created, its length cannot be changed.

Java arrays are **objects**. This has several consequences. Arrays are created using a form of the **new** operator. No variable can ever **hold** an array; a variable can only **refer** to an array. Any variable that can refer to an array can also hold the value **null**, meaning that it doesn’t at the moment refer to anything. Like any object, an array belongs to a class, which like all classes is a subclass of the class *Object*. The elements of the array are, essentially, instance variables in the array object, except that they are referred to by number rather than by name.

Nevertheless, even though arrays are objects, there are differences between arrays and other kinds of objects, and there are a number of special language features in Java for creating and using arrays.

7.1.2 Using Arrays

Suppose that *A* is a variable that refers to an array. Then the element at index *k* in *A* is referred to as *A*[*k*]. The first element is *A*[0], the second is *A*[1], and so forth. “*A*[*k*]” is really a variable, and it can be used just like any other variable. You can assign values to it, you can

use it in expressions, and you can pass it as a parameter to a subroutine. All of this will be discussed in more detail below. For now, just keep in mind the syntax

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

for referring to an element of an array.

Although every array, as an object, belongs to some class, array classes never have to be defined. Once a type exists, the corresponding array class exists automatically. If the name of the type is *BaseType*, then the name of the associated array class is *BaseType*[]. That is to say, an object belonging to the class *BaseType*[] is an array of items, where each item is a variable of type *BaseType*. The brackets, “[]”, are meant to recall the syntax for referring to the individual items in the array. “*BaseType*[]” is read as “array of *BaseType*” or “*BaseType* array.” It might be worth mentioning here that if *ClassA* is a subclass of *ClassB*, then the class *ClassA*[] is automatically a subclass of *ClassB*[].

The base type of an array can be any legal Java type. From the primitive type **int**, the array type *int*[] is derived. Each element in an array of type *int*[] is a variable of type **int**, which holds a value of type **int**. From a class named *Shape*, the array type *Shape*[] is derived. Each item in an array of type *Shape*[] is a variable of type *Shape*, which holds a value of type *Shape*. This value can be either **null** or a reference to an object belonging to the class *Shape*. (This includes objects belonging to subclasses of *Shape*.)

* * *

Let’s try to get a little more concrete about all this, using arrays of integers as our first example. Since *int*[] is a class, it can be used to declare variables. For example,

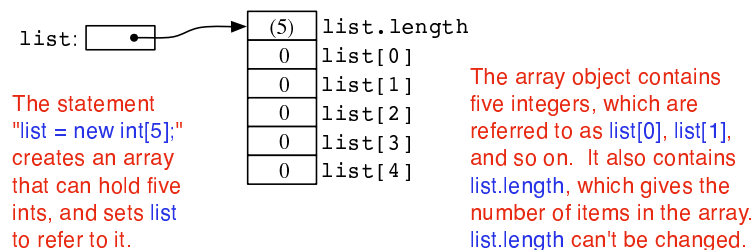
```
int[] list;
```

creates a variable named **list** of type *int*[]. This variable is **capable** of referring to an array of **ints**, but initially its value is **null** (if **list** is a member variable in a class) or undefined (if **list** is a local variable in a method). The **new** operator is used to create a new array object, which can then be assigned to **list**. The syntax for using **new** with arrays is different from the syntax you learned previously. As an example,

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

creates an array of five integers. More generally, the constructor “**new** *BaseType*[N]” is used to create an array belonging to the class *BaseType*[]. The value N in brackets specifies the length of the array, that is, the number of elements that it contains. Note that the array “knows” how long it is. The length of the array is an instance variable in the array object. In fact, the length of an array, **list**, can be referred to as **list.length**. (However, you are not allowed to change the value of **list.length**, so it’s really a “**final**” instance variable, that is, one whose value cannot be changed after it has been initialized.)

The situation produced by the statement “**list** = **new** **int**[5];” can be pictured like this:



Note that the newly created array of integers is automatically filled with zeros. In Java, a newly created array is **always** filled with a known, default value: zero for numbers, **false** for **boolean**, the character with Unicode number zero for **char**, and **null** for objects.

The elements in the array, `list`, are referred to as `list[0]`, `list[1]`, `list[2]`, `list[3]`, and `list[4]`. (Note again that the index for the last item is one less than `list.length`.) However, array references can be much more general than this. The brackets in an array reference can contain any expression whose value is an integer. For example if `indx` is a variable of type **int**, then `list[indx]` and `list[2*indx+7]` are syntactically correct references to elements of the array `list`. Thus, the following loop would print all the integers in the array, `list`, to standard output:

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

The first time through the loop, `i` is 0, and `list[i]` refers to `list[0]`. So, it is the value stored in the variable `list[0]` that is printed. The second time through the loop, `i` is 1, and the value stored in `list[1]` is printed. The loop ends after printing the value of `list[4]`, when `i` becomes equal to 5 and the continuation condition “`i < list.length`” is no longer true. This is a typical example of using a loop to process an array. I’ll discuss more examples of array processing throughout this chapter.

Every use of a variable in a program specifies a memory location. Think for a moment about what the computer does when it encounters a reference to an array element, `list[k]`, while it is executing a program. The computer must determine which memory location is being referred to. To the computer, `list[k]` means something like this: “Get the pointer that is stored in the variable, `list`. Follow this pointer to find an array object. Get the value of `k`. Go to the `k`-th position in the array, and that’s the memory location you want.” There are two things that can go wrong here. Suppose that the value of `list` is **null**. If that is the case, then `list` doesn’t even refer to an array. The attempt to refer to an element of an array that doesn’t exist is an error that will cause an exception of type *NullPointerException* to be thrown. The second possible error occurs if `list` does refer to an array, but the value of `k` is outside the legal range of indices for that array. This will happen if `k < 0` or if `k >= list.length`. This is called an “array index out of bounds” error. When an error of this type occurs, an exception of type *ArrayIndexOutOfBoundsException* is thrown. When you use arrays in a program, you should be mindful that both types of errors are possible. However, array index out of bounds errors are by far the most common error when working with arrays.

7.1.3 Array Initialization

For an array variable, just as for any variable, you can declare the variable and initialize it in a single step. For example,

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

If `list` is a local variable in a subroutine, then this is exactly equivalent to the two statements:

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

(If `list` is an instance variable, then of course you can’t simply replace “`int[] list = new int[5];`” with “`int[] list; list = new int[5];`” since the assignment statement “`list = new int[5];`” is only legal inside a subroutine.)

The new array is filled with the default value appropriate for the base type of the array—zero for **int** and **null** for class types, for example. However, Java also provides a way to initialize an array variable with a new array filled with a specified list of values. In a declaration statement that creates a new array, this is done with an *array initializer*. For example,

```
int[] list = { 1, 4, 9, 16, 25, 36, 49 };
```

creates a new array containing the seven values 1, 4, 9, 16, 25, 36, and 49, and sets **list** to refer to that new array. The value of **list[0]** will be 1, the value of **list[1]** will be 4, and so forth. The length of **list** is seven, since seven values are provided in the initializer.

An array initializer takes the form of a list of values, separated by commas and enclosed between braces. The length of the array does not have to be specified, because it is implicit in the list of values. The items in an array initializer don't have to be constants. They can be variables or arbitrary expressions, provided that their values are of the appropriate type. For example, the following declaration creates an array of eight *Colors*. Some of the colors are given by expressions of the form “**new Color(r,g,b)**” instead of by constants:

```
Color[] palette = {
    Color.BLACK,
    Color.RED,
    Color.PINK,
    new Color(0,180,0), // dark green
    Color.GREEN,
    Color.BLUE,
    new Color(180,180,255), // light blue
    Color.WHITE
};
```

A list initializer of this form can be used **only** in a declaration statement, to give an initial value to a newly declared array variable. It cannot be used in an assignment statement to assign a value to a variable that has been previously declared. However, there is another, similar notation for creating a new array that can be used in an assignment statement or passed as a parameter to a subroutine. The notation uses another form of the **new** operator to both create and initialize a new array object at the same time. (The rather odd syntax is similar to the syntax for anonymous classes, which were discussed in Subsection 5.7.3.) For example to assign a new value to an array variable, **list**, that was declared previously, you could use:

```
list = new int[] { 1, 8, 27, 64, 125, 216, 343 };
```

The general syntax for this form of the **new** operator is

```
new <base-type> [ ] { <list-of-values> }
```

This is actually an expression whose value is a reference to a newly created array object. This means that it can be used in any context where an object of type *<base-type>[]* is expected. For example, if **makeButtons** is a method that takes an array of *Strings* as a parameter, you could say:

```
makeButtons( new String[] { "Stop", "Go", "Next", "Previous" } );
```

Being able to create and use an array “in place” in this way can be very convenient, in the same way that anonymous nested classes are convenient.

By the way, it is perfectly legal to use the “**new BaseType[] { ... }**” syntax instead of the array initializer syntax in the declaration of an array variable. For example, instead of saying:

```
int[] primes = { 2, 3, 5, 7, 11, 13, 17, 19 };
```

you can say, equivalently,

```
int[] primes = new int[] { 2, 3, 5, 7, 11, 17, 19 };
```

In fact, rather than use a special notation that works only in the context of declaration statements, I prefer to use the second form.

* * *

One final note: For historical reasons, an array declaration such as

```
int[] list;
```

can also be written as

```
int list[];
```

which is a syntax used in the languages C and C++. However, this alternative syntax does not really make much sense in the context of Java, and it is probably best avoided. After all, the intent is to declare a variable of a certain type, and the name of that type is “int[]”. It makes sense to follow the “*<type-name> <variable-name>;*” syntax for such declarations.

7.2 Programming With Arrays

ARRAYS ARE THE MOST BASIC and the most important type of data structure, and techniques for processing arrays are among the most important programming techniques you can learn. Two fundamental array processing techniques—searching and sorting—will be covered in Section 7.4. This section introduces some of the basic ideas of array processing in general.

7.2.1 Arrays and for Loops

In many cases, processing an array means applying the same operation to each item in the array. This is commonly done with a **for** loop. A loop for processing all the elements of an array **A** has the form:

```
// do any necessary initialization
for (int i = 0; i < A.length; i++) {
    . . . // process A[i]
}
```

Suppose, for example, that **A** is an array of type `double[]`. Suppose that the goal is to add up all the numbers in the array. An informal algorithm for doing this would be:

```
Start with 0;
Add A[0];    (process the first item in A)
Add A[1];    (process the second item in A)
.
.
.
Add A[ A.length - 1 ];    (process the last item in A)
```

Putting the obvious repetition into a loop and giving a name to the sum, this becomes:

```
double sum; // The sum of the numbers in A.
sum = 0;    // Start with 0.
for (int i = 0; i < A.length; i++)
    sum += A[i]; // add A[i] to the sum, for
                // i = 0, 1, ..., A.length - 1
```

Note that the continuation condition, “`i < A.length`”, implies that the last value of `i` that is actually processed is `A.length-1`, which is the index of the final item in the array. It’s important to use “`<`” here, not “`<=`”, since “`<=`” would give an array index out of bounds error. There is no element at position `A.length` in `A`.

Eventually, you should just about be able to write loops similar to this one in your sleep. I will give a few more simple examples. Here is a loop that will count the number of items in the array `A` which are less than zero:

```
int count; // For counting the items.
count = 0; // Start with 0 items counted.
for (int i = 0; i < A.length; i++) {
    if (A[i] < 0.0) // if this item is less than zero...
        count++; // ...then count it
}
// At this point, the value of count is the number
// of items that have passed the test of being < 0
```

Replace the test “`A[i] < 0.0`”, if you want to count the number of items in an array that satisfy some other property. Here is a variation on the same theme. Suppose you want to count the number of times that an item in the array `A` is equal to the item that follows it. The item that follows `A[i]` in the array is `A[i+1]`, so the test in this case is “`if (A[i] == A[i+1])`”. But there is a catch: This test cannot be applied when `A[i]` is the last item in the array, since then there is no such item as `A[i+1]`. The result of trying to apply the test in this case would be an *ArrayIndexOutOfBoundsException*. This just means that we have to stop one item short of the final item:

```
int count = 0;
for (int i = 0; i < A.length - 1; i++) {
    if (A[i] == A[i+1])
        count++;
}
```

Another typical problem is to find the largest number in `A`. The strategy is to go through the array, keeping track of the largest number found so far. We’ll store the largest number found so far in a variable called `max`. As we look through the array, whenever we find a number larger than the current value of `max`, we change the value of `max` to that larger value. After the whole array has been processed, `max` is the largest item in the array overall. The only question is, what should the original value of `max` be? One possibility is to start with `max` equal to `A[0]`, and then to look through the rest of the array, starting from `A[1]`, for larger items:

```
double max = A[0];
for (int i = 1; i < A.length; i++) {
    if (A[i] > max)
        max = A[i];
}
// at this point, max is the largest item in A
```

(There is one subtle problem here. It's possible in Java for an array to have length zero. In that case, `A[0]` doesn't exist, and the reference to `A[0]` in the first line gives an array index out of bounds error. However, zero-length arrays are normally something that you want to avoid in real problems. Anyway, what would it mean to ask for the largest item in an array that contains no items at all?)

As a final example of basic array operations, consider the problem of copying an array. To make a copy of our sample array `A`, it is **not** sufficient to say

```
double[] B = A;
```

since this does not create a new array object. All it does is declare a new array variable and make it refer to the same object to which `A` refers. (So that, for example, a change to `A[i]` will automatically change `B[i]` as well.) To make a new array that is a copy of `A`, it is necessary to make a new array object and to copy each of the individual items from `A` into the new array:

```
double[] B = new double[A.length]; // Make a new array object,
                                   // the same size as A.
for (int i = 0; i < A.length; i++)
    B[i] = A[i]; // Copy each item from A to B.
```

Copying values from one array to another is such a common operation that Java has a predefined subroutine to do it. The subroutine, `System.arraycopy()`, is a static member subroutine in the standard `System` class. Its declaration has the form

```
public static void arraycopy(Object sourceArray, int sourceStartIndex,
                             Object destArray, int destStartIndex, int count)
```

where `sourceArray` and `destArray` can be arrays with any base type. Values are copied from `sourceArray` to `destArray`. The `count` tells how many elements to copy. Values are taken from `sourceArray` starting at position `sourceStartIndex` and are stored in `destArray` starting at position `destStartIndex`. For example, to make a copy of the array, `A`, using this subroutine, you would say:

```
double B = new double[A.length];
System.arraycopy( A, 0, B, 0, A.length );
```

7.2.2 Arrays and for-each Loops

Java 5.0 introduced a new form of the `for` loop, the “for-each loop” that was introduced in Subsection 3.4.4. The for-each loop is meant specifically for processing all the values in a data structure. When used to process an array, a for-each loop can be used to perform the same operation on each value that is stored in the array. If `anArray` is an array of type `BaseType[]`, then a for-each loop for `anArray` has the form:

```
for ( BaseType item : anArray ) {
    .
    . // process the item
    .
}
```

In this loop, `item` is the loop control variable. It is being declared as a variable of type *BaseType*, where *BaseType* is the base type of the array. (In a for-each loop, the loop control variable **must** be declared in the loop.) When this loop is executed, each value from the array is assigned to `item` in turn and the body of the loop is executed for each value. Thus, the above loop is exactly equivalent to:


```

for ( int index = 0; index < anArray.length; index++ ) {
    BaseType item;
    item = anArray[index]; // Get one of the values from the array
    .
    . // process the item
    .
}

```

For example, if `A` is an array of type `int[]`, then we could print all the values from `A` with the for-each loop:

```

for ( int item : A )
    System.out.println( item );

```

and we could add up all the positive integers in `A` with:

```

int sum = 0; // This will be the sum of all the positive numbers in A
for ( int item : A ) {
    if ( item > 0 )
        sum = sum + item;
}

```

The for-each loop is not always appropriate. For example, there is no simple way to use it to process the items in just a part of an array. However, it does make it a little easier to process all the values in an array, since it eliminates any need to use array indices.

It's important to note that a for-each loop processes the **values** in the array, not the **elements** (where an element means the actual memory location that is part of the array). For example, consider the following incorrect attempt to fill an array of integers with 17's:

```

int[] intList = new int[10];
for ( int item : intList ) { // INCORRECT! DOES NOT MODIFY THE ARRAY!
    item = 17;
}

```

The assignment statement `item = 17` assigns the value 17 to the loop control variable, `item`. However, this has nothing to do with the array. When the body of the loop is executed, the value from one of the elements of the array is copied into `item`. The statement `item = 17` replaces that copied value but has no effect on the array element from which it was copied; the value in the array is not changed.

7.2.3 Array Types in Subroutines

Any array type, such as `double[]`, is a full-fledged Java type, so it can be used in all the ways that any other Java type can be used. In particular, it can be used as the type of a formal parameter in a subroutine. It can even be the return type of a function. For example, it might be useful to have a function that makes a copy of an array of `double`:

```

/**
 * Create a new array of doubles that is a copy of a given array.
 * @param source the array that is to be copied; the value can be null
 * @return a copy of source; if source is null, then the return value is also null
 */
public static double[] copy( double[] source ) {
    if ( source == null )

```

```

        return null;
    double[] cpy; // A copy of the source array.
    cpy = new double[source.length];
    System.arraycopy( source, 0, cpy, 0, source.length );
    return cpy;
}

```

The `main()` routine of a program has a parameter of type `String[]`. You’ve seen this used since all the way back in Section 2.1, but I haven’t really been able to explain it until now. The parameter to the `main()` routine is an array of *Strings*. When the system calls the `main()` routine, the strings in this array are the **command-line arguments** from the command that was used to run the program. When using a command-line interface, the user types a command to tell the system to execute a program. The user can include extra input in this command, beyond the name of the program. This extra input becomes the command-line arguments. For example, if the name of the class that contains the `main()` routine is `myProg`, then the user can type “`java myProg`” to execute the program. In this case, there are no command-line arguments. But if the user types the command

```
java myProg one two three
```

then the command-line arguments are the strings “one”, “two”, and “three”. The system puts these strings into an array of *Strings* and passes that array as a parameter to the `main()` routine. Here, for example, is a short program that simply prints out any command line arguments entered by the user:

```

public class CLDemo {

    public static void main(String[] args) {
        System.out.println("You entered " + args.length
                           + " command-line arguments");

        if (args.length > 0) {
            System.out.println("They were:");
            for (int i = 0; i < args.length; i++)
                System.out.println("    " + args[i]);
        }
    } // end main()

} // end class CLDemo

```

Note that the parameter, `args`, is never `null` when `main()` is called by the system, but it might be an array of length zero.

In practice, command-line arguments are often the names of files to be processed by the program. I will give some examples of this in Chapter 11, when I discuss file processing.

7.2.4 Random Access

So far, all my examples of array processing have used **sequential access**. That is, the elements of the array were processed one after the other in the sequence in which they occur in the array. But one of the big advantages of arrays is that they allow **random access**. That is, every element of the array is equally accessible at any given time.

As an example, let’s look at a well-known problem called the birthday problem: Suppose that there are N people in a room. What’s the chance that there are two people in the room who have the same birthday? (That is, they were born on the same day in the same month, but not necessarily in the same year.) Most people severely underestimate the probability. We

will actually look at a different version of the question: Suppose you choose people at random and check their birthdays. How many people will you check before you find one who has the same birthday as someone you've already checked? Of course, the answer in a particular case depends on random factors, but we can simulate the experiment with a computer program and run the program several times to get an idea of how many people need to be checked on average.

To simulate the experiment, we need to keep track of each birthday that we find. There are 365 different possible birthdays. (We'll ignore leap years.) For each possible birthday, we need to keep track of whether or not we have already found a person who has that birthday. The answer to this question is a boolean value, true or false. To hold the data for all 365 possible birthdays, we can use an array of 365 boolean values:

```
boolean[] used;
used = new boolean[365];
```

The days of the year are numbered from 0 to 364. The value of `used[i]` is true if someone has been selected whose birthday is day number `i`. Initially, all the values in the array, `used`, are false. When we select someone whose birthday is day number `i`, we first check whether `used[i]` is true. If so, then this is the second person with that birthday. We are done. If `used[i]` is false, we set `used[i]` to be true to record the fact that we've encountered someone with that birthday, and we go on to the next person. Here is a subroutine that carries out the simulated experiment (of course, in the subroutine, there are no simulated people, only simulated birthdays):

```
/**
 * Simulate choosing people at random and checking the day of the year they
 * were born on. If the birthday is the same as one that was seen previously,
 * stop, and output the number of people who were checked.
 */
private static void birthdayProblem() {

    boolean[] used; // For recording the possible birthdays
                    // that have been seen so far. A value
                    // of true in used[i] means that a person
                    // whose birthday is the i-th day of the
                    // year has been found.

    int count;      // The number of people who have been checked.

    used = new boolean[365]; // Initially, all entries are false.

    count = 0;

    while (true) {
        // Select a birthday at random, from 0 to 364.
        // If the birthday has already been used, quit.
        // Otherwise, record the birthday as used.
        int birthday; // The selected birthday.
        birthday = (int)(Math.random()*365);
        count++;
        if ( used[birthday] ) // This day was found before; It's a duplicate.
            break;
        used[birthday] = true;
    }

    System.out.println("A duplicate birthday was found after "
```

```

        + count + " tries.");

    } // end birthdayProblem()

```

This subroutine makes essential use of the fact that every element in a newly created array of **boolean** is set to be **false**. If we wanted to reuse the same array in a second simulation, we would have to reset all the elements in it to be **false** with a **for** loop:

```

    for (int i = 0; i < 365; i++)
        used[i] = false;

```

The program that uses this subroutine is *BirthdayProblemDemo.java*. An applet version of the program can be found in the online version of this section.

7.2.5 Arrays of Objects

One of the examples in Subsection 6.4.2 was an applet that shows multiple copies of a message in random positions, colors, and fonts. When the user clicks on the applet, the positions, colors, and fonts are changed to new random values. Like several other examples from that chapter, the applet had a flaw: It didn't have any way of storing the data that would be necessary to redraw itself. Arrays provide us with one possible solution to this problem. We can write a new version of the RandomStrings applet that uses an array to store the position, font, and color of each string. When the content pane of the applet is painted, this information is used to draw the strings, so the applet will paint itself correctly whenever it has to be redrawn. When the user clicks on the applet, the array is filled with new random values and the applet is repainted using the new data. So, the only time that the picture will change is in response to a mouse click.

In this applet, the number of copies of the message is given by a named constant, `MESSAGE_COUNT`. One way to store the position, color, and font of `MESSAGE_COUNT` strings would be to use four arrays:

```

int[] x = new int[MESSAGE_COUNT];
int[] y = new int[MESSAGE_COUNT];
Color[] color = new Color[MESSAGE_COUNT];
Font[] font = new Font[MESSAGE_COUNT];

```

These arrays would be filled with random values. In the `paintComponent()` method, the *i*-th copy of the string would be drawn at the point `(x[i],y[i])`. Its color would be given by `color[i]`. And it would be drawn in the font `font[i]`. This would be accomplished by the `paintComponent()` method

```

public void paintComponent(Graphics g) {
    super.paintComponent(); // (Fill with background color.)
    for (int i = 0; i < MESSAGE_COUNT; i++) {
        g.setColor( color[i] );
        g.setFont( font[i] );
        g.drawString( message, x[i], y[i] );
    }
}

```

This approach is said to use *parallel arrays*. The data for a given copy of the message is spread out across several arrays. If you think of the arrays as laid out in parallel columns—array `x` in the first column, array `y` in the second, array `color` in the third, and array `font` in the fourth—then the data for the *i*-th string can be found along the *i*-th row. There

is nothing wrong with using parallel arrays in this simple example, but it does go against the object-oriented philosophy of keeping related data in one object. If we follow this rule, then we don't have to **imagine** the relationship among the data, because all the data for one copy of the message is physically in one place. So, when I wrote the applet, I made a simple class to represent all the data that is needed for one copy of the message:

```
/**
 * An object of this type holds the position, color, and font
 * of one copy of the string.
 */
private static class StringData {
    int x, y;        // The coordinates of the left end of baseline of string.
    Color color;     // The color in which the string is drawn.
    Font font;       // The font that is used to draw the string.
}
```

(This class is actually defined as a static nested class in the main applet class.) To store the data for multiple copies of the message, I use an array of type `StringData[]`. The array is declared as an instance variable, with the name `stringData`:

```
StringData[] stringData;
```

Of course, the value of `stringData` is `null` until an actual array is created and assigned to it. This is done in the `init()` method of the applet with the statement

```
stringData = new StringData[MESSAGE_COUNT];
```

The base type of this array is *StringData*, which is a class. We say that `stringData` is an **array of objects**. This means that the elements of the array are variables of type *StringData*. Like any object variable, each element of the array can either be `null` or can hold a reference to an object. (Note that the term “array of objects” is a little misleading, since the objects are not in the array; the array can only contain references to objects.) When the `stringData` array is first created, the value of each element in the array is `null`.

The data needed by the RandomStrings program will be stored in objects of type *StringData*, but no such objects exist yet. All we have so far is an array of variables that are capable of referring to such objects. I decided to create the *StringData* objects in the applet's `init` method. (It could be done in other places—just so long as we avoid trying to use an object that doesn't exist. This is important: Remember that a newly created array whose base type is an object type is always filled with `null` elements. There are **no** objects in the array until you put them there.) The objects are created with the `for` loop

```
for (int i = 0; i < MESSAGE_COUNT; i++)
    stringData[i] = new StringData();
```

For the RandomStrings applet, the idea is to store data for the *i*-th copy of the message in the variables `stringData[i].x`, `stringData[i].y`, `stringData[i].color`, and `stringData[i].font`. Make sure that you understand the notation here: `stringData[i]` refers to an object. That object contains instance variables. The notation `stringData[i].x` tells the computer: “Find your way to the object that is referred to by `stringData[i]`. Then go to the instance variable named `x` in that object.” Variable names can get even more complicated than this, so it is important to learn how to read them. Using the array, `stringData`, the `paintComponent()` method for the applet could be written

```

public void paintComponent(Graphics g) {
    super.paintComponent(g); // (Fill with background color.)
    for (int i = 0; i < MESSAGE_COUNT; i++) {
        g.setColor( stringData[i].color );
        g.setFont( stringData[i].font );
        g.drawString( message, stringData[i].x, stringData[i].y );
    }
}

```

However, since the for loop is processing every value in the array, an alternative would be to use a for-each loop:

```

public void paintComponent(Graphics g) {
    super.paintComponent(g);
    for (StringData data : stringData) {
        // Draw a copy of the message in the position, color,
        // and font stored in data.
        g.setColor( data.color );
        g.setFont( data.font );
        g.drawString( message, data.x, data.y );
    }
}

```

In the loop, the loop control variable, `data`, holds a copy of one of the values from the array. That value is a reference to an object of type *StringData*, which has instance variables named `color`, `font`, `x`, and `y`. Once again, the use of a for-each loop has eliminated the need to work with array indices.

There is still the matter of filling the array, `data`, with random values. If you are interested, you can look at the source code for the applet, *RandomStringsWithArray.java*.

* * *

The *RandomStrings* applet uses one other array of objects. The font for a given copy of the message is chosen at random from a set of five possible fonts. In the original version of the applet, there were five variables of type *Font* to represent the fonts. The variables were named `font1`, `font2`, `font3`, `font4`, and `font5`. To select one of these fonts at random, a `switch` statement could be used:

```

Font randomFont; // One of the 5 fonts, chosen at random.
int rand;        // A random integer in the range 0 to 4.

rand = (int)(Math.random() * 5);
switch (rand) {
    case 0:
        randomFont = font1;
        break;
    case 1:
        randomFont = font2;
        break;
    case 2:
        randomFont = font3;
        break;
    case 3:
        randomFont = font4;
        break;
    case 4:

```

```

        randomFont = font5;
        break;
    }

```

In the new version of the applet, the five fonts are stored in an array, which is named `fonts`. This array is declared as an instance variable of type `Font[]`

```
Font[] fonts;
```

The array is created in the `init()` method of the applet, and each element of the array is set to refer to a new *Font* object:

```

fonts = new Font[5]; // Create the array to hold the five fonts.

fonts[0] = new Font("Serif", Font.BOLD, 14);
fonts[1] = new Font("SansSerif", Font.BOLD + Font.ITALIC, 24);
fonts[2] = new Font("Monospaced", Font.PLAIN, 20);
fonts[3] = new Font("Dialog", Font.PLAIN, 30);
fonts[4] = new Font("Serif", Font.ITALIC, 36);

```

This makes it much easier to select one of the fonts at random. It can be done with the statements

```

Font randomFont; // One of the 5 fonts, chosen at random.
int fontIndex;   // A random number in the range 0 to 4.
fontIndex = (int)(Math.random() * 5);
randomFont = fonts[ fontIndex ];

```

The `switch` statement has been replaced by a single line of code. In fact, the preceding four lines could be replaced by the single line:

```
Font randomFont = fonts[ (int)(Math.random() * 5) ];
```

This is a very typical application of arrays. Note that this example uses the random access property of arrays: We can pick an array index at random and go directly to the array element at that index.

Here is another example of the same sort of thing. Months are often stored as numbers 1, 2, 3, ..., 12. Sometimes, however, these numbers have to be translated into the names January, February, ..., December. The translation can be done with an array. The array can be declared and initialized as

```

static String[] monthName = { "January", "February", "March",
                               "April",   "May",      "June",
                               "July",    "August",   "September",
                               "October", "November", "December" };

```

If `mnth` is a variable that holds one of the integers 1 through 12, then `monthName[mnth-1]` is the name of the corresponding month. We need the “-1” because months are numbered starting from 1, while array elements are numbered starting from 0. Simple array indexing does the translation for us!

7.2.6 Variable Arity Methods

Arrays are used in the implementation of one of the new features in Java 5.0. Before version 5.0, every method in Java had a fixed arity. (The *arity* of a subroutine is defined as the number of parameters in a call to the method.) In a fixed arity method, the number of parameters must be the same in every call to the method. Java 5.0 introduced *variable arity methods*. In a

variable arity method, different calls to the method can have different numbers of parameters. For example, the formatted output method `System.out.printf`, which was introduced in Subsection 2.4.4, is a variable arity method. The first parameter of `System.out.printf` must be a *String*, but it can have any number of additional parameters, of any types.

Calling a variable arity method is no different from calling any other sort of method, but writing one requires some new syntax. As an example, consider a method that can compute the average of any number of values of type **double**. The definition of such a method could begin with:

```
public static double average( double... numbers ) {
```

Here, the `...` after the type name, **double**, indicates that any number of values of type **double** can be provided when the subroutine is called, so that for example `average(1,2,3)`, `average(3.14,2.17)`, `average(0.375)`, and even `average()` are all legal calls to this method. Note that actual parameters of type **int** can be passed to `average`. The integers will, as usual, be automatically converted to real numbers.

When the method is called, the values of all the actual parameters that correspond to the variable arity parameter are placed into an array, and it is this array that is actually passed to the method. That is, in the body of a method, a variable arity parameter of type *T* actually looks like an ordinary parameter of type `T[]`. The length of the array tells you how many actual parameters were provided in the method call. In the `average` example, the body of the method would see an array named `numbers` of type `double[]`. The number of actual parameters in the method call would be `numbers.length`, and the values of the actual parameters would be `numbers[0]`, `numbers[1]`, and so on. A complete definition of the method would be:

```
public static double average( double... numbers ) {
    double sum;          // The sum of all the actual parameters.
    double average;      // The average of all the actual parameters.
    sum = 0;
    for (int i = 0; i < numbers.length; i++) {
        sum = sum + numbers[i]; // Add one of the actual parameters to the sum.
    }
    average = sum / numbers.length;
    return average;
}
```

Note that the “...” can be applied only to the **last** formal parameter in a method definition. Note also that it is possible to pass an actual array to the method, instead of a list of individual values. For example, if `salesData` is a variable of type `double[]`, then it would be legal to call `average(salesData)`, and this would compute the average of all the numbers in the array.

As another example, consider a method that can draw a polygon through any number of points. The points are given as values of type *Point*, where an object of type *Point* has two instance variables, `x` and `y`, of type **int**. In this case, the method has one ordinary parameter—the graphics context that will be used to draw the polygon—in addition to the variable arity parameter:

```
public static void drawPolygon(Graphics g, Point... points) {
    if (points.length > 1) { // (Need at least 2 points to draw anything.)
        for (int i = 0; i < points.length - 1; i++) {
            // Draw a line from i-th point to (i+1)-th point
            g.drawLine( points[i].x, points[i].y, points[i+1].x, points[i+1].y );
        }
    }
}
```



```

        // Now, draw a line back to the starting point.
        g.drawLine( points[points.length-1].x, points[points.length-1].y,
                    points[0].x, points[0].y );
    }
}

```

Because of automatic type conversion, a variable arity parameter of type “`Object...`” can take actual parameters of any type whatsoever. Even primitive type values are allowed, because of autoboxing. (A primitive type value belonging to a type such as `int` is converted to an object belonging to a “wrapper” class such as *Integer*. See Subsection 5.3.2.) For example, the method definition for `System.out.printf` could begin:

```
public void printf(String format, Object... values) {
```

This allows the `printf` method to output values of any type. Similarly, we could write a method that strings together the string representations of all its parameters into one long string:

```

    public static String concat( Object... values ) {
        String str = ""; // Start with an empty string.
        for ( Object obj : values ) { // A "for each" loop for processing the values.
            if (obj == null )
                str = str + "null"; // Represent null values by "null".
            else
                str = str + obj.toString();
        }
    }
}

```

7.3 Dynamic Arrays and ArrayLists

THE SIZE OF AN ARRAY is fixed when it is created. In many cases, however, the number of data items that are actually stored in the array varies with time. Consider the following examples: An array that stores the lines of text in a word-processing program. An array that holds the list of computers that are currently downloading a page from a Web site. An array that contains the shapes that have been added to the screen by the user of a drawing program. Clearly, we need some way to deal with cases where the number of data items in an array is not fixed.

7.3.1 Partially Full Arrays

Consider an application where the number of items that we want to store in an array changes as the program runs. Since the size of the array can’t actually be changed, a separate counter variable must be used to keep track of how many spaces in the array are in use. (Of course, every space in the array has to contain something; the question is, how many spaces contain useful or valid items?)

Consider, for example, a program that reads positive integers entered by the user and stores them for later processing. The program stops reading when the user inputs a number that is less than or equal to zero. The input numbers can be kept in an array, `numbers`, of type `int[]`. Let’s say that no more than 100 numbers will be input. Then the size of the array can be fixed at 100. But the program must keep track of how many numbers have actually been read and stored in the array. For this, it can use an integer variable, `numCount`. Each time a number is stored in the array, `numCount` must be incremented by one. As a rather silly example, let’s write a program that will read the numbers input by the user and then print them in reverse

order. (This is, at least, a processing task that requires that the numbers be saved in an array. Remember that many types of processing, such as finding the sum or average or maximum of the numbers, can be done without saving the individual numbers.)

```
public class ReverseInputNumbers {

    public static void main(String[] args) {

        int[] numbers; // An array for storing the input values.
        int numCount;   // The number of numbers saved in the array.
        int num;        // One of the numbers input by the user.

        numbers = new int[100]; // Space for 100 ints.
        numCount = 0;           // No numbers have been saved yet.

        TextIO.putln("Enter up to 100 positive integers; enter 0 to end.");

        while (true) { // Get the numbers and put them in the array.
            TextIO.put("? ");
            num = TextIO.getlnInt();
            if (num <= 0)
                break;
            numbers[numCount] = num;
            numCount++;
        }

        TextIO.putln("\nYour numbers in reverse order are:\n");

        for (int i = numCount - 1; i >= 0; i--) {
            TextIO.putln( numbers[i] );
        }

        } // end main();

    } // end class ReverseInputNumbers
```

It is especially important to note that the variable `numCount` plays a dual role. It is the number of items that have been entered into the array. But it is also the index of the next available spot in the array. For example, if 4 numbers have been stored in the array, they occupy locations number 0, 1, 2, and 3. The next available spot is location 4. When the time comes to print out the numbers in the array, the last occupied spot in the array is location `numCount - 1`, so the `for` loop prints out values starting from location `numCount - 1` and going down to 0.

Let's look at another, more realistic example. Suppose that you write a game program, and that players can join the game and leave the game as it progresses. As a good object-oriented programmer, you probably have a class named *Player* to represent the individual players in the game. A list of all players who are currently in the game could be stored in an array, `playerList`, of type `Player[]`. Since the number of players can change, you will also need a variable, `playerCt`, to record the number of players currently in the game. Assuming that there will never be more than 10 players in the game, you could declare the variables as:

```
Player[] playerList = new Player[10]; // Up to 10 players.
int      playerCt = 0; // At the start, there are no players.
```

After some players have joined the game, `playerCt` will be greater than 0, and the `player` objects representing the players will be stored in the array elements `playerList[0]`, `playerList[1]`, ..., `playerList[playerCt-1]`. Note that the array element

`playerList[playerCt]` is **not** in use. The procedure for adding a new player, `newPlayer`, to the game is simple:

```
playerList[playerCt] = newPlayer; // Put new player in next
                                //      available spot.
playerCt++; // And increment playerCt to count the new player.
```

Deleting a player from the game is a little harder, since you don't want to leave a "hole" in the array. Suppose you want to delete the player at index `k` in `playerList`. If you are not worried about keeping the players in any particular order, then one way to do this is to move the player from the last occupied position in the array into position `k` and then to decrement the value of `playerCt`:

```
playerList[k] = playerList[playerCt - 1];
playerCt--;
```

The player previously in position `k` is no longer in the array. The player previously in position `playerCt - 1` is now in the array twice. But it's only in the occupied or valid part of the array once, since `playerCt` has decreased by one. Remember that every element of the array has to hold some value, but only the values in positions 0 through `playerCt - 1` will be looked at or processed in any way. (By the way, you should think about what happens if the player that is being deleted is in the last position in the list. The code does still work in this case. What exactly happens?)

Suppose that when deleting the player in position `k`, you'd like to keep the remaining players in the same order. (Maybe because they take turns in the order in which they are stored in the array.) To do this, all the players in positions `k+1` and above must move down one position in the array. Player `k+1` replaces player `k`, who is out of the game. Player `k+2` fills the spot left open when player `k+1` is moved. And so on. The code for this is

```
for (int i = k+1; i < playerCt; i++) {
    playerList[i-1] = playerList[i];
}
playerCt--;
```

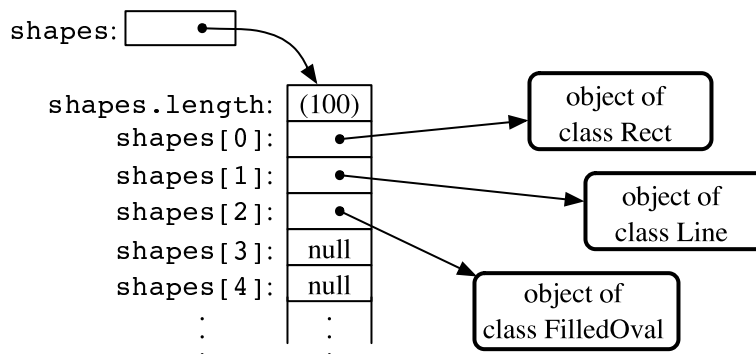
* * *

It's worth emphasizing that the *Player* example deals with an array whose base type is a class. An item in the array is either `null` or is a reference to an object belonging to the class, *Player*. The *Player* objects themselves are not really stored in the array, only references to them. Note that because of the rules for assignment in Java, the objects can actually belong to subclasses of *Player*. Thus there could be different classes of players such as computer players, regular human players, players who are wizards, ..., all represented by different subclasses of *Player*.

As another example, suppose that a class *Shape* represents the general idea of a shape drawn on a screen, and that it has subclasses to represent specific types of shapes such as lines, rectangles, rounded rectangles, ovals, filled-in ovals, and so forth. (*Shape* itself would be an abstract class, as discussed in Subsection 5.5.5.) Then an array of type `Shape[]` can hold references to objects belonging to the subclasses of *Shape*. For example, the situation created by the statements

```
Shape[] shapes = new Shape[100]; // Array to hold up to 100 shapes.
shapes[0] = new Rect();           // Put some objects in the array.
shapes[1] = new Line();
shapes[2] = new FilledOval();
int shapeCt = 3; // Keep track of number of objects in array.
```

could be illustrated as:



Such an array would be useful in a drawing program. The array could be used to hold a list of shapes to be displayed. If the `Shape` class includes a method, “`void redraw(Graphics g)`”, for drawing the shape in a graphics context `g`, then all the shapes in the array could be redrawn with a simple for loop:

```

for (int i = 0; i < shapeCt; i++)
    shapes[i].redraw(g);
  
```

The statement “`shapes[i].redraw(g);`” calls the `redraw()` method belonging to the particular shape at index `i` in the array. Each object knows how to redraw itself, so that repeated executions of the statement can produce a variety of different shapes on the screen. This is nice example both of polymorphism and of array processing.

7.3.2 Dynamic Arrays

In each of the above examples, an arbitrary limit was set on the number of items—100 `ints`, 10 `Players`, 100 `Shapes`. Since the size of an array is fixed, a given array can only hold a certain maximum number of items. In many cases, such an arbitrary limit is undesirable. Why should a program work for 100 data values, but not for 101? The obvious alternative of making an array that’s so big that it will work in any practical case is not usually a good solution to the problem. It means that in most cases, a lot of computer memory will be wasted on unused space in the array. That memory might be better used for something else. And what if someone is using a computer that could handle as many data values as the user actually wants to process, but doesn’t have enough memory to accommodate all the extra space that you’ve allocated for your huge array?

Clearly, it would be nice if we could increase the size of an array at will. This is not possible, but what **is** possible is almost as good. Remember that an array variable does not actually hold an array. It just holds a reference to an array object. We can’t make the array bigger, but we can make a new, bigger array object and change the value of the array variable so that it refers to the bigger array. Of course, we also have to copy the contents of the old array into the new array. The array variable then refers to an array object that contains all the data of the old array, with room for additional data. The old array will be garbage collected, since it is no longer in use.

Let’s look back at the game example, in which `playerList` is an array of type `Player[]` and `playerCt` is the number of spaces that have been used in the array. Suppose that we don’t want to put a pre-set limit on the number of players. If a new player joins the game and the

current array is full, we just make a new, bigger one. The same variable, `playerList`, will refer to the new array. Note that after this is done, `playerList[0]` will refer to a different memory location, but the value stored in `playerList[0]` will still be the same as it was before. Here is some code that will do this:

```
// Add a new player, even if the current array is full.

if (playerCt == playerList.length) {
    // Array is full. Make a new, bigger array,
    // copy the contents of the old array into it,
    // and set playerList to refer to the new array.
    int newSize = 2 * playerList.length; // Size of new array.
    Player[] temp = new Player[newSize]; // The new array.
    System.arraycopy(playerList, 0, temp, 0, playerList.length);
    playerList = temp; // Set playerList to refer to new array.
}

// At this point, we KNOW there is room in the array.

playerList[playerCt] = newPlayer; // Add the new player...
playerCt++;                       // ...and count it.
```

If we are going to be doing things like this regularly, it would be nice to define a reusable class to handle the details. An array-like object that changes size to accommodate the amount of data that it actually contains is called a *dynamic array*. A dynamic array supports the same operations as an array: putting a value at a given position and getting the value that is stored at a given position. But there is no upper limit on the positions that can be used (except those imposed by the size of the computer's memory). In a dynamic array class, the `put` and `get` operations must be implemented as instance methods. Here, for example, is a class that implements a dynamic array of **ints**:

```
/**
 * An object of type DynamicArrayOfInt acts like an array of int
 * of unlimited size. The notation A.get(i) must be used instead
 * of A[i], and A.set(i,v) must be used instead of A[i] = v.
 */
public class DynamicArrayOfInt {

    private int[] data; // An array to hold the data.

    /**
     * Constructor creates an array with an initial size of 1,
     * but the array size will be increased whenever a reference
     * is made to an array position that does not yet exist.
     */
    public DynamicArrayOfInt() {
        data = new int[1];
    }

    /**
     * Get the value from the specified position in the array.
     * Since all array elements are initialized to zero, when the
     * specified position lies outside the actual physical size
     * of the data array, a value of 0 is returned. Note that
     * a negative value of position will still produce an
     * ArrayIndexOutOfBoundsException.
     */
}
```

```

    */
    public int get(int position) {
        if (position >= data.length)
            return 0;
        else
            return data[position];
    }

    /**
     * Store the value in the specified position in the array.
     * The data array will increase in size to include this
     * position, if necessary.
     */
    public void put(int position, int value) {
        if (position >= data.length) {
            // The specified position is outside the actual size of
            // the data array. Double the size, or if that still does
            // not include the specified position, set the new size
            // to 2*position.
            int newSize = 2 * data.length;
            if (position >= newSize)
                newSize = 2 * position;
            int[] newData = new int[newSize];
            System.arraycopy(data, 0, newData, 0, data.length);
            data = newData;
            // The following line is for demonstration purposes only !!
            System.out.println("Size of dynamic array increased to " + newSize);
        }
        data[position] = value;
    }

} // end class DynamicArrayOfInt

```

The data in a *DynamicArrayOfInt* object is actually stored in a regular array, but that array is discarded and replaced by a bigger array whenever necessary. If `numbers` is a variable of type *DynamicArrayOfInt*, then the command `numbers.put(pos,val)` stores the value `val` at position number `pos` in the dynamic array. The function `numbers.get(pos)` returns the value stored at position number `pos`.

The first example in this section used an array to store positive integers input by the user. We can rewrite that example to use a *DynamicArrayOfInt*. A reference to `numbers[i]` is replaced by `numbers.get(i)`. The statement “`numbers[numCount] = num;`” is replaced by “`numbers.put(numCount,num);`”. Here’s the program:

```

public class ReverseWithDynamicArray {

    public static void main(String[] args) {

        DynamicArrayOfInt numbers; // To hold the input numbers.
        int numCount; // The number of numbers stored in the array.
        int num; // One of the numbers input by the user.

        numbers = new DynamicArrayOfInt();
        numCount = 0;

        TextIO.putln("Enter some positive integers; Enter 0 to end");
        while (true) { // Get numbers and put them in the dynamic array.

```

```

        TextIO.put("? ");
        num = TextIO.getlnInt();
        if (num <= 0)
            break;
        numbers.put(numCount, num); // Store num in the dynamic array.
        numCount++;
    }

    TextIO.putln("\nYour numbers in reverse order are:\n");

    for (int i = numCount - 1; i >= 0; i--) {
        TextIO.putln( numbers.get(i) ); // Print the i-th number.
    }

} // end main();

} // end class ReverseWithDynamicArray

```

7.3.3 ArrrayLists

The *DynamicArrayOfInt* class could be used in any situation where an array of **int** with no preset limit on the size is needed. However, if we want to store *Shapes* instead of **ints**, we would have to define a new class to do it. That class, probably named “*DynamicArrayOfShape*”, would look exactly the same as the *DynamicArrayOfInt* class except that everywhere the type “**int**” appears, it would be replaced by the type “*Shape*”. Similarly, we could define a *DynamicArrayOfDouble* class, a *DynamicArrayOfPlayer* class, and so on. But there is something a little silly about this, since all these classes are close to being identical. It would be nice to be able to write some kind of source code, once and for all, that could be used to generate any of these classes on demand, given the type of value that we want to store. This would be an example of *generic programming*. Some programming languages, including C++, have had support for generic programming for some time. With version 5.0, Java introduced true generic programming, but even before that it had something that was very similar: One can come close to generic programming in Java by working with data structures that contain elements of type *Object*. We will first consider the almost-generic programming that has been available in Java from the beginning, and then we will look at the change that was introduced in Java 5.0. A full discussion of generic programming will be given in Chapter 10.

In Java, every class is a subclass of the class named *Object*. This means that every object can be assigned to a variable of type *Object*. Any object can be put into an array of type *Object*[]. If we defined a *DynamicArrayOfObject* class, then we could store objects of any type. This is not true generic programming, and it doesn’t apply to the primitive types such as **int** and **double**. But it does come close. In fact, there is no need for us to define a *DynamicArrayOfObject* class. Java already has a standard class named *ArrayList* that serves much the same purpose. The *ArrayList* class is in the package `java.util`, so if you want to use it in a program, you should put the directive “`import java.util.ArrayList;`” at the beginning of your source code file.

The *ArrayList* class differs from my *DynamicArrayOfInt* class in that an *ArrayList* object always has a definite size, and it is illegal to refer to a position in the *ArrayList* that lies outside its size. In this, an *ArrayList* is more like a regular array. However, the size of an *ArrayList* can be increased at will. The *ArrayList* class defines many instance methods. I’ll describe some of the most useful. Suppose that `list` is a variable of type *ArrayList*. Then we have:

- `list.size()` — This function returns the current size of the *ArrayList*. The only valid positions in the list are numbers in the range 0 to `list.size()-1`. Note that the size can be zero. A call to the default constructor `new ArrayList()` creates an *ArrayList* of size zero.
- `list.add(obj)` — Adds an object onto the end of the list, increasing the size by 1. The parameter, `obj`, can refer to an object of any type, or it can be `null`.
- `list.get(N)` — This function returns the value stored at position `N` in the *ArrayList*. `N` must be an integer in the range 0 to `list.size()-1`. If `N` is outside this range, an error of type *IndexOutOfBoundsException* occurs. Calling this function is similar to referring to `A[N]` for an array, `A`, except that you can't use `list.get(N)` on the left side of an assignment statement.
- `list.set(N, obj)` — Assigns the object, `obj`, to position `N` in the *ArrayList*, replacing the item previously stored at position `N`. The integer `N` must be in the range from 0 to `list.size()-1`. A call to this function is equivalent to the command `A[N] = obj` for an array `A`.
- `list.remove(obj)` — If the specified object occurs somewhere in the *ArrayList*, it is removed from the list. Any items in the list that come after the removed item are moved down one position. The size of the *ArrayList* decreases by 1. If `obj` occurs more than once in the list, only the first copy is removed.
- `list.remove(N)` — For an integer, `N`, this removes the `N`-th item in the *ArrayList*. `N` must be in the range 0 to `list.size()-1`. Any items in the list that come after the removed item are moved down one position. The size of the *ArrayList* decreases by 1.
- `list.indexOf(obj)` — A function that searches for the object, `obj`, in the *ArrayList*. If the object is found in the list, then the position number where it is found is returned. If the object is not found, then `-1` is returned.

For example, suppose again that players in a game are represented by objects of type *Player*. The players currently in the game could be stored in an *ArrayList* named `players`. This variable would be declared as

```
ArrayList players;
```

and initialized to refer to a new, empty *ArrayList* object with

```
players = new ArrayList();
```

If `newPlayer` is a variable that refers to a *Player* object, the new player would be added to the *ArrayList* and to the game by saying

```
players.add(newPlayer);
```

and if player number `i` leaves the game, it is only necessary to say

```
players.remove(i);
```

Or, if `player` is a variable that refers to the *Player* that is to be removed, you could say

```
players.remove(player);
```

All this works very nicely. The only slight difficulty arises when you use the function `players.get(i)` to get the value stored at position `i` in the *ArrayList*. The return type of this function is *Object*. In this case the object that is returned by the function is actually of type *Player*. In order to do anything useful with the returned value, it's usually necessary to type-cast it to type *Player*:


```
Player plr = (Player)players.get(i);
```

For example, if the *Player* class includes an instance method `makeMove()` that is called to allow a player to make a move in the game, then the code for letting every player make a move is

```
for (int i = 0; i < players.size(); i++) {
    Player plr = (Player)players.get(i);
    plr.makeMove();
}
```

The two lines inside the `for` loop can be combined to a single line:

```
((Player)players.get(i)).makeMove();
```

This gets an item from the list, type-casts it, and then calls the `makeMove()` method on the resulting *Player*. The parentheses around “`(Player)players.get(i)`” are required because of Java’s precedence rules. The parentheses force the type-cast to be performed before the `makeMove()` method is called.

For-each loops work for *ArrayLists* just as they do for arrays. But note that since the items in an *ArrayList* are only known to be *Objects*, the type of the loop control variable must be *Object*. For example, the `for` loop used above to let each *Player* make a move could be written as the for-each loop

```
for ( Object plrObj : players ) {
    Player plr = (Player)plrObj;
    plr.makeMove();
}
```

In the body of the loop, the value of the loop control variable, `plrObj`, is one of the objects from the list, `players`. This object must be type-cast to type *Player* before it can be used.

* * *

In Subsection 5.5.5, I discussed a program, *ShapeDraw*, that uses *ArrayLists*. Here is another version of the same idea, simplified to make it easier to see how *ArrayList* is being used. The program supports the following operations: Click the large white drawing area to add a colored rectangle. (The color of the rectangle is given by a “rainbow palette” along the bottom of the applet; click the palette to select a new color.) Drag rectangles using the right mouse button. Hold down the Alt key and click on a rectangle to delete it. Shift-click a rectangle to move it out in front of all the other rectangles. You can try an applet version of the program in the on-line version of this section.

Source code for the main panel for this program can be found in *SimpleDrawRects.java*. You should be able to follow the source code in its entirety. (You can also take a look at the file *RainbowPalette.java*, which defines the color palette shown at the bottom of the applet, if you like.) Here, I just want to look at the parts of the program that use an *ArrayList*.

The applet uses a variable named `rects`, of type *ArrayList*, to hold information about the rectangles that have been added to the drawing area. The objects that are stored in the list belong to a static nested class, *ColoredRect*, that is defined as

```
/**
 * An object of type ColoredRect holds the data for one colored rectangle.
 */
private static class ColoredRect {
    int x,y;           // Upper left corner of the rectangle.
    int width,height;  // Size of the rectangle.
    Color color;       // Color of the rectangle.
}
```

If `g` is a variable of type `Graphics`, then the following code draws all the rectangles that are stored in the list `rects` (with a black outline around each rectangle):

```
for (int i = 0; i < rects.size(); i++) {
    ColoredRect rect = (ColoredRect)rects.get(i);
    g.setColor( rect.color );
    g.fillRect( rect.x, rect.y, rect.width, rect.height);
    g.setColor( Color.BLACK );
    g.drawRect( rect.x, rect.y, rect.width - 1, rect.height - 1);
}
```

The *i*-th rectangle in the list is obtained by calling `rects.get(i)`. Since this method returns a value of type *Object*, the return value must be typecast to its actual type, *ColoredRect*, to get access to the data that it contains.

To implement the mouse operations, it must be possible to find the rectangle, if any, that contains the point where the user clicked the mouse. To do this, I wrote the function

```
/**
 * Find the topmost rect that contains the point (x,y). Return null
 * if no rect contains that point. The rects in the ArrayList are
 * considered in reverse order so that if one lies on top of another,
 * the one on top is seen first and is returned.
 */
ColoredRect findRect(int x, int y) {
    for (int i = rects.size() - 1; i >= 0; i--) {
        ColoredRect rect = (ColoredRect)rects.get(i);
        if ( x >= rect.x && x < rect.x + rect.width
            && y >= rect.y && y < rect.y + rect.height )
            return rect; // (x,y) is inside this rect.
    }

    return null; // No rect containing (x,y) was found.
}
```

The code for removing a *ColoredRect*, `rect`, from the drawing area is simply `rects.remove(rect)` (followed by a `repaint()`). Bringing a given rectangle out in front of all the other rectangles is just a little harder. Since the rectangles are drawn in the order in which they occur in the *ArrayList*, the rectangle that is in the last position in the list is in front of all the other rectangles on the screen. So we need to move the selected rectangle to the last position in the list. This can most easily be done in a slightly tricky way using built-in *ArrayList* operations: The rectangle is simply removed from its current position in the list and then added back at the end of the list:

```
void bringToFront(ColoredRect rect) {
    if (rect != null) {
        rects.remove(rect); // Remove rect from the list.
        rects.add(rect);    // Add it back; it will be placed in the last position.
        repaint();
    }
}
```

This should be enough to give you the basic idea. You can look in the source code for more details.

7.3.4 Parameterized Types

The main difference between true generic programming and the *ArrayList* examples in the previous subsection is the use of the type *Object* as the basic type for objects that are stored in a list. This has at least two unfortunate consequences: First, it makes it necessary to use type-casting in almost every case when an element is retrieved from that list. Second, since any type of object can legally be added to the list, there is no way for the compiler to detect an attempt to add the wrong type of object to the list; the error will be detected only at run time when the object is retrieved from the list and the attempt to type-cast the object fails. Compare this to arrays. An array of type `BaseType[]` can **only** hold objects of type *BaseType*. An attempt to store an object of the wrong type in the array will be detected by the compiler, and there is no need to type-cast items that are retrieved from the array back to type *BaseType*.

To address this problem, Java 5.0 introduced *parameterized types*. *ArrayList* is an example: Instead of using the plain “`ArrayList`” type, it is possible to use `ArrayList<BaseType>`, where *BaseType* is any object type, that is, the name of a class or of an interface. (*BaseType cannot* be one of the primitive types.) `ArrayList<BaseType>` can be used to create lists that can hold only objects of type *BaseType*. For example,

```
ArrayList<ColoredRect> rects;
```

declares a variable named `rects` of type `ArrayList<ColoredRect>`, and

```
rects = new ArrayList<ColoredRect>();
```

sets `rects` to refer to a newly created list that can only hold objects belonging to the class *ColoredRect* (or to a subclass). The funny-looking name “`ArrayList<ColoredRect>`” is being used here in exactly the same way as an ordinary class name—don’t let the “`<ColoredRect>`” confuse you; it’s just part of the name of the type. When a statement such as `rects.add(x)`; occurs in the program, the compiler can check whether `x` is in fact of type *ColoredRect*. If not, the compiler will report a syntax error. When an object is retrieved from the list, the compiler knows that the object must be of type *ColoredRect*, so no type-cast is necessary. You can say simply:

```
ColoredRect rect = rects.get(i)
```

You can even refer directly to an instance variable in the object, such as `rects.get(i).color`. This makes using `ArrayList<ColoredRect>` very similar to using `ColoredRect[]` with the added advantage that the list can grow to any size. Note that if a for-each loop is used to process the items in `rects`, the type of the loop control variable can be *ColoredRect*, and no type-cast is necessary. For example, when using `ArrayList<ColoredRect>` as the type for the list `rects`, the code for drawing all the rectangles in the list could be rewritten as:

```
for ( ColoredRect rect : rects ) {
    g.setColor( rect.color );
    g.fillRect( rect.x, rect.y, rect.width, rect.height );
    g.setColor( Color.BLACK );
    g.drawRect( rect.x, rect.y, rect.width - 1, rect.height - 1 );
}
```

You can use `ArrayList<ColoredRect>` anywhere where you could use a normal type: to declare variables, as the type of a formal parameter in a subroutine, or as the return type of a subroutine. You can even create a subclass of `ArrayList<ColoredRect>`! (Nevertheless, technically speaking, `ArrayList<ColoredRect>` is not considered to be a separate class from *ArrayList*. An object of

type `ArrayList<ColoredRect>` actually belongs to the class `ArrayList`, but the compiler restricts the type of objects that can be added to the list.)

The only drawback to using parameterized types is that the base type cannot be a primitive type. For example, there is no such thing as “`ArrayList<int>`”. However, this is not such a big drawback as it might seem at first, because of the “wrapper types” and “autoboxing” that were introduced in Subsection 5.3.2. A wrapper type such as `Double` or `Integer` can be used as a base type for a parameterized type. An object of type `ArrayList<Double>` can hold objects of type `Double`. Since each object of type `Double` holds a value of type **double**, it’s almost like having a list of **doubles**. If `numlist` is declared to be of type `ArrayList<Double>` and if `x` is of type **double**, then the value of `x` can be added to the list by saying:

```
numlist.add( new Double(x) );
```

Furthermore, because of autoboxing, the compiler will automatically do **double-to-Double** and **Double-to-double** type conversions when necessary. This means that the compiler will treat “`numlist.add(x)`” as being equivalent to “`numlist.add(new Double(x))`”. So, behind the scenes, “`numlist.add(x)`” is actually adding an object to the list, but it looks a lot as if you are working with a list of **doubles**.

* * *

The sample program *SimplePaint2.java* demonstrates the use of parameterized types. In this program, the user can sketch curves in a drawing area by clicking and dragging with the mouse. The curves can be of any color, and the user can select the drawing color using a menu. The background color of the drawing area can also be selected using a menu. And there is a “Control” menu that contains several commands: An “Undo” command, which removes the most recently drawn curve from the screen, a “Clear” command that removes all the curves, and a “Use Symmetry” command that turns a symmetry feature on and off. Curves that are drawn by the user when the symmetry option is on are reflected horizontally and vertically to produce a symmetric pattern. You can try an applet version of the program in the on-line version of this section.

Unlike the original SimplePaint program in Subsection 6.4.4, this new version uses a data structure to store information about the picture that has been drawn by the user. This data is used in the `paintComponent()` method to redraw the picture whenever necessary. Thus, the picture doesn’t disappear when, for example, the picture is covered and then uncovered. The data structure is implemented using *ArrayLists*.

The main data for a curve consists of a list of the points on the curve. This data can be stored in an object of type `ArrayList<Point>`, where `java.awt.Point` is one of Java’s standard classes. (A *Point* object contains two public integer variables `x` and `y` that represent the coordinates of a point.) However, to redraw the curve, we also need to know its color, and we need to know whether the symmetry option should be applied to the curve. All the data that is needed to redraw the curve can be grouped into an object of type *CurveData* that is defined as

```
private static class CurveData {
    Color color; // The color of the curve.
    boolean symmetric; // Are horizontal and vertical reflections also drawn?
    ArrayList<Point> points; // The points on the curve.
}
```

However, a picture can contain many curves, not just one, so to store all the data necessary to redraw the entire picture, we need a **list** of objects of type *CurveData*. For this list, we can use a variable `curves` declared as

```
ArrayList<CurveData> curves = new ArrayList<CurveData>();
```

Here we have a list of objects, where each object contains a list of points as part of its data! Let's look at a few examples of processing this data structure. When the user clicks the mouse on the drawing surface, it's the start of a new curve, and a new *CurveData* object must be created and added to the list of curves. The instance variables in the new *CurveData* object must also be initialized. Here is the code from the `mousePressed()` routine that does this:

```
currentCurve = new CurveData();           // Create a new CurveData object.

currentCurve.color = currentColor;         // The color of the curve is taken from an
                                           // instance variable that represents the
                                           // currently selected drawing color.

currentCurve.symmetric = useSymmetry;      // The "symmetric" property of the curve
                                           // is also copied from the current value
                                           // of an instance variable, useSymmetry.

currentCurve.points = new ArrayList<Point>(); // Create a new point list object.

currentCurve.points.add( new Point(evt.getX(), evt.getY()) );
    // The point where the user pressed the mouse is the first point on
    // the curve. A new Point object is created to hold the coordinates
    // of that point and is added to the list of points for the curve.

curves.add(currentCurve); // Add the CurveData object to the list of curves.
```

As the user drags the mouse, new points are added to `currentCurve`, and `repaint()` is called. When the picture is redrawn, the new point will be part of the picture.

The `paintComponent()` method has to use the data in `curves` to draw all the curves. The basic structure is a for-each loop that processes the data for each individual curve in turn. This has the form:

```
for ( CurveData curve : curves ) {
    .
    . // Draw the curve represented by the object, curve, of type CurveData.
    .
}
```

In the body of this loop, `curve.points` is a variable of type `ArrayList<Point>` that holds the list of points on the curve. The *i*-th point on the curve can be obtained by calling the `get()` method of this list: `curve.points.get(i)`. This returns a value of type *Point* which contains instance variables named `x` and `y`. We can refer directly to the `x`-coordinate of the *i*-th point as:

```
curve.points.get(i).x
```

This might seem rather complicated, but it's a nice example of a complex name that specifies a path to a desired piece of data: Go to the object, `curve`. Inside `curve`, go to `points`. Inside `points`, get the *i*-th item. And from that item, get the instance variable named `x`. Here is the complete definition of the `paintComponent()` method:

```
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    for ( CurveData curve : curves ) {
        g.setColor(curve.color);
        for (int i = 1; i < curve.points.size(); i++) {
```

```

        // Draw a line segment from point number i-1 to point number i.
        int x1 = curve.points.get(i-1).x;
        int y1 = curve.points.get(i-1).y;
        int x2 = curve.points.get(i).x;
        int y2 = curve.points.get(i).y;
        g.drawLine(x1,y1,x2,y2);
        if (curve.symmetric) {
            // Also draw the horizontal and vertical reflections
            // of the line segment.
            int w = getWidth();
            int h = getHeight();
            g.drawLine(w-x1,y1,w-x2,y2);
            g.drawLine(x1,h-y1,x2,h-y2);
            g.drawLine(w-x1,h-y1,w-x2,h-y2);
        }
    }
} // end paintComponent()

```

I encourage you to read the full source code, *SimplePaint2.java*. In addition to serving as an example of using parameterized types, it also serves as another example of creating and using menus.

7.3.5 Vectors

The *ArrayList* class was introduced in Java version 1.2, as one of a group of classes designed for working with collections of objects. We'll look at these "collection classes" in Chapter 10. Early versions of Java did not include *ArrayList*, but they did have a very similar class named *java.util.Vector*. You can still see *Vectors* used in older code and in many of Java's standard classes, so it's worth knowing about them. Using a *Vector* is similar to using an *ArrayList*, except that different names are used for some commonly used instance methods, and some instance methods in one class don't correspond to any instance method in the other class.

Like an *ArrayList*, a *Vector* is similar to an array of *Objects* that can grow to be as large as necessary. The default constructor, `new Vector()`, creates a vector with no elements. Suppose that *vec* is a *Vector*. Then we have:

- `vec.size()` — a function that returns the number of elements currently in the vector.
- `vec.elementAt(N)` — returns the *N*-th element of the vector, for an integer *N*. *N* must be in the range 0 to `vec.size()-1`. This is the same as `get(N)` for an *ArrayList*.
- `vec.setElementAt(obj,N)` — sets the *N*-th element in the vector to be *obj*. *N* must be in the range 0 to `vec.size()-1`. This is the same as `set(N,obj)` for an *ArrayList*.
- `vec.addElement(obj)` — adds the *Object*, *obj*, to the end of the vector. This is the same as the `add()` method of an *ArrayList*.
- `vec.removeElement(obj)` — removes *obj* from the vector, if it occurs. Only the first occurrence is removed. This is the same as `remove(obj)` for an *ArrayList*.
- `vec.removeElementAt(N)` — removes the *N*-th element, for an integer *N*. *N* must be in the range 0 to `vec.size()-1`. This is the same as `remove(N)` for an *ArrayList*.
- `vec.setSize(N)` — sets the size of the vector to *N*. If there were more than *N* elements in *vec*, the extra elements are removed. If there were fewer than *N* elements, extra spaces are filled with `null`. The *ArrayList* class, unfortunately, does not have a `setSize()` method.

The *Vector* class includes many more methods, but these are probably the most commonly used. Note that in Java 5.0, *Vector* can be used as a parameterized type in exactly the same way as *ArrayList*. That is, if *BaseType* is any class or interface name, then *Vector<BaseType>* represents vectors that can hold only objects of type *BaseType*.

7.4 Searching and Sorting

TWO ARRAY PROCESSING TECHNIQUES that are particularly common are *searching* and *sorting*. Searching here refers to finding an item in the array that meets some specified criterion. Sorting refers to rearranging all the items in the array into increasing or decreasing order (where the meaning of increasing and decreasing can depend on the context).

Sorting and searching are often discussed, in a theoretical sort of way, using an array of numbers as an example. In practical situations, though, more interesting types of data are usually involved. For example, the array might be a mailing list, and each element of the array might be an object containing a name and address. Given the name of a person, you might want to look up that person's address. This is an example of searching, since you want to find the object in the array that contains the given name. It would also be useful to be able to sort the array according to various criteria. One example of sorting would be ordering the elements of the array so that the names are in alphabetical order. Another example would be to order the elements of the array according to zip code before printing a set of mailing labels. (This kind of sorting can get you a cheaper postage rate on a large mailing.)

This example can be generalized to a more abstract situation in which we have an array that contains objects, and we want to search or sort the array based on the value of one of the instance variables in that array. We can use some terminology here that originated in work with "databases," which are just large, organized collections of data. We refer to each of the objects in the array as a *record*. The instance variables in an object are then called *fields* of the record. In the mailing list example, each record would contain a name and address. The fields of the record might be the first name, last name, street address, state, city and zip code. For the purpose of searching or sorting, one of the fields is designated to be the *key* field. Searching then means finding a record in the array that has a specified value in its key field. Sorting means moving the records around in the array so that the key fields of the record are in increasing (or decreasing) order.

In this section, most of my examples follow the tradition of using arrays of numbers. But I'll also give a few examples using records and keys, to remind you of the more practical applications.

7.4.1 Searching

There is an obvious algorithm for searching for a particular item in an array: Look at each item in the array in turn, and check whether that item is the one you are looking for. If so, the search is finished. If you look at every item without finding the one you want, then you can be sure that the item is not in the array. It's easy to write a subroutine to implement this algorithm. Let's say the array that you want to search is an array of **ints**. Here is a method that will search the array for a specified integer. If the integer is found, the method returns the index of the location in the array where it is found. If the integer is not in the array, the method returns the value -1 as a signal that the integer could not be found:

```
/**
```

```

    * Searches the array A for the integer N.  If N is not in the array,
    * then -1 is returned.  If N is in the array, then return value is
    * the first integer i that satisfies A[i] == N.
    */
static int find(int[] A, int N) {
    for (int index = 0; index < A.length; index++) {
        if ( A[index] == N )
            return index;  // N has been found at this index!
    }

    // If we get this far, then N has not been found
    // anywhere in the array.  Return a value of -1.

    return -1;
}

```

This method of searching an array by looking at each item in turn is called *linear search*. If nothing is known about the order of the items in the array, then there is really no better alternative algorithm. But if the elements in the array are known to be in increasing or decreasing order, then a much faster search algorithm can be used. An array in which the elements are in order is said to be *sorted*. Of course, it takes some work to sort an array, but if the array is to be searched many times, then the work done in sorting it can really pay off.

Binary search is a method for searching for a given item in a **sorted** array. Although the implementation is not trivial, the basic idea is simple: If you are searching for an item in a sorted list, then it is possible to eliminate half of the items in the list by inspecting a single item. For example, suppose that you are looking for the number 42 in a sorted array of 1000 integers. Let's assume that the array is sorted into increasing order. Suppose you check item number 500 in the array, and find that the item is 93. Since 42 is less than 93, and since the elements in the array are in increasing order, we can conclude that if 42 occurs in the array at all, then it must occur somewhere before location 500. All the locations numbered 500 or above contain values that are greater than or equal to 93. These locations can be eliminated as possible locations of the number 42.

The next obvious step is to check location 250. If the number at that location is, say, -21, then you can eliminate locations before 250 and limit further search to locations between 251 and 499. The next test will limit the search to about 125 locations, and the one after that to about 62. After just 10 steps, there is only one location left. This is a whole lot better than looking through every element in the array. If there were a million items, it would still take only 20 steps for binary search to search the array! (Mathematically, the number of steps is approximately equal to the logarithm, in the base 2, of the number of items in the array.)

In order to make binary search into a Java subroutine that searches an array A for an item N, we just have to keep track of the range of locations that could possibly contain N. At each step, as we eliminate possibilities, we reduce the size of this range. The basic operation is to look at the item in the middle of the range. If this item is greater than N, then the second half of the range can be eliminated. If it is less than N, then the first half of the range can be eliminated. If the number in the middle just happens to be N exactly, then the search is finished. If the size of the range decreases to zero, then the number N does not occur in the array. Here is a subroutine that returns the location of N in a sorted array A. If N cannot be found in the array, then a value of -1 is returned instead:

```

/**

```



```

* Searches the array A for the integer N.
* Precondition: A must be sorted into increasing order.
* Postcondition: If N is in the array, then the return value, i,
*   satisfies A[i] == N. If N is not in the array, then the
*   return value is -1.
*/
static int binarySearch(int[] A, int N) {

    int lowestPossibleLoc = 0;
    int highestPossibleLoc = A.length - 1;

    while (highestPossibleLoc >= lowestPossibleLoc) {
        int middle = (lowestPossibleLoc + highestPossibleLoc) / 2;
        if (A[middle] == N) {
            // N has been found at this index!
            return middle;
        }
        else if (A[middle] > N) {
            // eliminate locations >= middle
            highestPossibleLoc = middle - 1;
        }
        else {
            // eliminate locations <= middle
            lowestPossibleLoc = middle + 1;
        }
    }

    // At this point, highestPossibleLoc < LowestPossibleLoc,
    // which means that N is known to be not in the array. Return
    // a -1 to indicate that N could not be found in the array.

    return -1;
}

```

7.4.2 Association Lists

One particularly common application of searching is with *association lists*. The standard example of an association list is a dictionary. A dictionary associates definitions with words. Given a word, you can use the dictionary to look up its definition. We can think of the dictionary as being a list of *pairs* of the form (w, d) , where w is a word and d is its definition. A general association list is a list of pairs (k, v) , where k is some “key” value, and v is a value associated to that key. In general, we want to assume that no two pairs in the list have the same key. There are two basic operations on association lists: Given a key, k , find the value v associated with k , if any. And given a key, k , and a value v , add the pair (k, v) to the association list (replacing the pair, if any, that had the same key value). The two operations are usually called *get* and *put*.

Association lists are very widely used in computer science. For example, a compiler has to keep track of the location in memory associated with each variable. It can do this with an association list in which each key is a variable name and the associated value is the address of that variable in memory. Another example would be a mailing list, if we think of it as associating an address to each name on the list. As a related example, consider a phone directory that

associates a phone number to each name. The items in the list could be objects belonging to the class:

```
class PhoneEntry {
    String name;
    String phoneNum;
}
```

The data for a phone directory consists of an array of type `PhoneEntry[]` and an integer variable to keep track of how many entries are actually stored in the directory. The technique of “dynamic arrays” (Subsection 7.3.2) can be used in order to avoid putting an arbitrary limit on the number of entries that the phone directory can hold. Using an *ArrayList* would be another possibility. A *PhoneDirectory* class should include instance methods that implement the “get” and “put” operations. Here is one possible simple definition of the class:

```
/**
 * A PhoneDirectory holds a list of names with a phone number for
 * each name. It is possible to find the number associated with
 * a given name, and to specify the phone number for a given name.
 */
public class PhoneDirectory {

    /**
     * An object of type PhoneEntry holds one name/number pair.
     */
    private static class PhoneEntry {
        String name;        // The name.
        String number;      // The associated phone number.
    }

    private PhoneEntry[] data; // Array that holds the name/number pairs.
    private int dataCount;     // The number of pairs stored in the array.

    /**
     * Constructor creates an initially empty directory.
     */
    public PhoneDirectory() {
        data = new PhoneEntry[1];
        dataCount = 0;
    }

    /**
     * Looks for a name/number pair with a given name. If found, the index
     * of the pair in the data array is returned. If no pair contains the
     * given name, then the return value is -1.
     */
    private int find( String name ) {
        for (int i = 0; i < dataCount; i++) {
            if (data[i].name.equals(name))
                return i; // The name has been found in position i.
        }
        return -1; // The name does not exist in the array.
    }

    /**
     * Finds the phone number, if any, for a given name.
     */
}
```

```

    * @return The phone number associated with the name; if the name does
    *         not occur in the phone directory, then the return value is null.
    */
    public String getNumber( String name ) {
        int position = find(name);
        if (position == -1)
            return null;    // There is no phone entry for the given name.
        else
            return data[position].number;
    }

    /**
     * Associates a given name with a given phone number.  If the name
     * already exists in the phone directory, then the new number replaces
     * the old one.  Otherwise, a new name/number pair is added.  The
     * name and number should both be non-null.  An IllegalArgumentException
     * is thrown if this is not the case.
     */
    public void putNumber( String name, String number ) {
        if (name == null || number == null)
            throw new IllegalArgumentException("name and number cannot be null");
        int i = find(name);
        if (i >= 0) {
            // The name already exists, in position i in the array.
            // Just replace the old number at that position with the new.
            data[i].number = number;
        }
        else {
            // Add a new name/number pair to the array.  If the array is
            // already full, first create a new, larger array.
            if (dataCount == data.length) {
                PhoneEntry[] newData = new PhoneEntry[ 2*data.length ];
                System.arraycopy(newData,0,data,0,dataCount);
                data = newData;
            }
            PhoneEntry newEntry = new PhoneEntry(); // Create a new pair.
            newEntry.name = name;
            newEntry.number = number;
            data[dataCount] = newEntry; // Add the new pair to the array.
            dataCount++;
        }
    }
}

} // end class PhoneDirectory

```

The class defines a private instance method, `find()`, that uses linear search to find the position of a given name in the array of name/number pairs. The `find()` method is used both in the `getNumber()` method and in the `putNumber()` method. Note in particular that `putNumber(name,number)` has to check whether the name is in the phone directory. If so, it just changes the number in the existing entry; if not, it has to create a new phone entry and add it to the array.

This class could use a lot of improvement. For one thing, it would be nice to use binary search instead of simple linear search in the `getNumber` method. However, we could only do that if the list of `PhoneEntries` were sorted into alphabetical order according to name. In fact,

it's really not all that hard to keep the list of entries in sorted order, as you'll see in the next subsection.

7.4.3 Insertion Sort

We've seen that there are good reasons for sorting arrays. There are many algorithms available for doing so. One of the easiest to understand is the *insertion sort* algorithm. This method is also applicable to the problem of **keeping** a list in sorted order as you add new items to the list. Let's consider that case first:

Suppose you have a sorted list and you want to add an item to that list. If you want to make sure that the modified list is still sorted, then the item must be inserted into the right location, with all the smaller items coming before it and all the bigger items after it. This will mean moving each of the bigger items up one space to make room for the new item.

```
/*
 * Precondition:  itemsInArray is the number of items that are
 *               stored in A.  These items must be in increasing order
 *               (A[0] <= A[1] <= ... <= A[itemsInArray-1]).
 *               The array size is at least one greater than itemsInArray.
 * Postcondition: The number of items has increased by one,
 *               newItem has been added to the array, and all the items
 *               in the array are still in increasing order.
 * Note: To complete the process of inserting an item in the
 *       array, the variable that counts the number of items
 *       in the array must be incremented, after calling this
 *       subroutine.
 */
static void insert(int[] A, int itemsInArray, int newItem) {
    int loc = itemsInArray - 1;  // Start at the end of the array.

    /* Move items bigger than newItem up one space;
       Stop when a smaller item is encountered or when the
       beginning of the array (loc == 0) is reached. */
    while (loc >= 0 && A[loc] > newItem) {
        A[loc + 1] = A[loc];  // Bump item from A[loc] up to loc+1.
        loc = loc - 1;        // Go on to next location.
    }

    A[loc + 1] = newItem;  // Put newItem in last vacated space.
}
```

Conceptually, this could be extended to a sorting method if we were to take all the items out of an unsorted array, and then insert them back into the array one-by-one, keeping the list in sorted order as we do so. Each insertion can be done using the `insert` routine given above. In the actual algorithm, we don't really take all the items from the array; we just remember what part of the array has been sorted:

```
static void insertionSort(int[] A) {
    // Sort the array A into increasing order.

    int itemsSorted; // Number of items that have been sorted so far.

    for (itemsSorted = 1; itemsSorted < A.length; itemsSorted++) {
```

```

// Assume that items A[0], A[1], ... A[itemsSorted-1]
// have already been sorted. Insert A[itemsSorted]
// into the sorted part of the list.

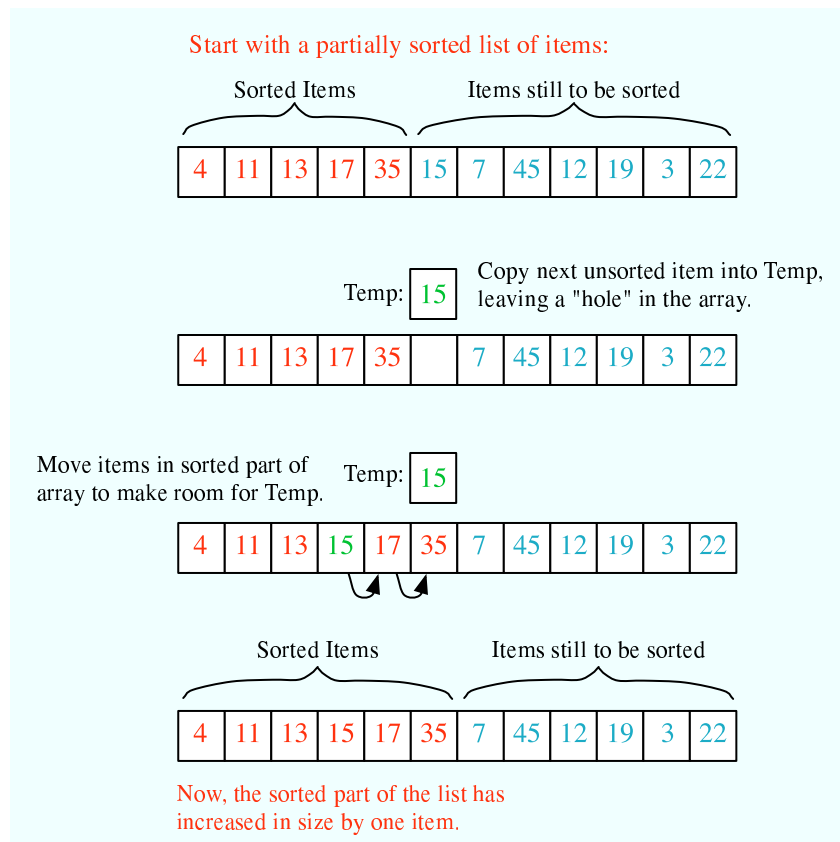
int temp = A[itemsSorted]; // The item to be inserted.
int loc = itemsSorted - 1; // Start at end of list.

while (loc >= 0 && A[loc] > temp) {
    A[loc + 1] = A[loc]; // Bump item from A[loc] up to loc+1.
    loc = loc - 1;      // Go on to next location.
}

A[loc + 1] = temp; // Put temp in last vacated space.
}
}

```

The following is an illustration of one stage in insertion sort. It shows what happens during one execution of the `for` loop in the above method, when `itemsSorted` is 5:



7.4.4 Selection Sort

Another typical sorting method uses the idea of finding the biggest item in the list and moving it to the end—which is where it belongs if the list is to be in increasing order. Once the biggest item is in its correct location, you can then apply the same idea to the remaining items. That is, find the next-biggest item, and move it into the next-to-last space, and so forth. This algorithm is called *selection sort*. It's easy to write:

```

static void selectionSort(int[] A) {
    // Sort A into increasing order, using selection sort

    for (int lastPlace = A.length-1; lastPlace > 0; lastPlace--) {
        // Find the largest item among A[0], A[1], ...,
        // A[lastPlace], and move it into position lastPlace
        // by swapping it with the number that is currently
        // in position lastPlace.

        int maxLoc = 0; // Location of largest item seen so far.

        for (int j = 1; j <= lastPlace; j++) {
            if (A[j] > A[maxLoc]) {
                // Since A[j] is bigger than the maximum we've seen
                // so far, j is the new location of the maximum value
                // we've seen so far.
                maxLoc = j;
            }
        }

        int temp = A[maxLoc]; // Swap largest item with A[lastPlace].
        A[maxLoc] = A[lastPlace];
        A[lastPlace] = temp;
    } // end of for loop
}

```

Insertion sort and selection sort are suitable for sorting fairly small arrays (up to a few hundred elements, say). There are more complicated sorting algorithms that are much faster than insertion sort and selection sort for large arrays. I'll discuss one such algorithm in Chapter 9.

* * *

A variation of selection sort is used in the *Hand* class that was introduced in Subsection 5.4.1. (By the way, you are finally in a position to fully understand the source code for both the *Hand* class and the *Deck* class from that section. See the source files *Deck.java* and *Hand.java*.)

In the *Hand* class, a hand of playing cards is represented by a *Vector*. This is older code, which used *Vector* instead of *ArrayList*, and I have chosen not to modify it so that you would see at least one example of using *Vectors*. See Subsection 7.3.5 for a discussion of *Vectors*.

The objects stored in the *Vector* are of type *Card*. A *Card* object contains instance methods *getSuit()* and *getValue()* that can be used to determine the suit and value of the card. In my sorting method, I actually create a new vector and move the cards one-by-one from the old vector to the new vector. The cards are selected from the old vector in increasing order. In the end, the new vector becomes the hand and the old vector is discarded. This is certainly not the most efficient procedure! But hands of cards are so small that the inefficiency is negligible. Here is the code for sorting cards by suit:

```

/**
 * Sorts the cards in the hand so that cards of the same suit are
 * grouped together, and within a suit the cards are sorted by value.
 * Note that aces are considered to have the lowest value, 1.
 */
public void sortBySuit() {
    Vector newHand = new Vector();
    while (hand.size() > 0) {

```

```

        int pos = 0; // Position of minimal card found so far.
        Card c = (Card)hand.elementAt(0); // The minimal card.
        for (int i = 1; i < hand.size(); i++) {
            Card c1 = (Card)hand.elementAt(i);
            if ( c1.getSuit() < c.getSuit() ||
                (c1.getSuit() == c.getSuit() && c1.getValue() < c.getValue()) ) {
                pos = i;
                c = c1;
            }
        }
        hand.removeElementAt(pos);
        newHand.addElement(c);
    }
    hand = newHand;
}

```

This example illustrates the fact that comparing items in a list is not usually as simple as using the operator “<”. In this case, we consider one card to be less than another if the suit of the first card is less than the suit of the second, and also if the suits are the same and the value of the second card is less than the value of the first. The second part of this test ensures that cards with the same suit will end up sorted by value.

Sorting a list of *Strings* raises a similar problem: the “<” operator is not defined for strings. However, the *String* class does define a `compareTo` method. If `str1` and `str2` are of type *String*, then

```
str1.compareTo(str2)
```

returns an **int** that is 0 when `str1` is equal to `str2`, is less than 0 when `str1` preceeds `str2`, and is greater than 0 when `str1` follows `str2`. The definition of “succeeds” and “follows” for strings uses what is called *lexicographic ordering*, which is based on the Unicode values of the characters in the strings. Lexicographic ordering is not the same as alphabetical ordering, even for strings that consist entirely of letters (because in lexicographic ordering, all the upper case letters come before all the lower case letters). However, for words consisting strictly of the 26 lower case letters in the English alphabet, lexicographic and alphabetic ordering are the same. Thus, if `str1` and `str2` are strings containing only letters from the English alphabet, then the test

```
str1.toLowerCase().compareTo(str2.toLowerCase()) < 0
```

is true if and only if `str1` comes before `str2` in alphabetical order.

7.4.5 Unsorting

I can’t resist ending this section on sorting with a related problem that is much less common, but is a bit more fun. That is the problem of putting the elements of an array into a random order. The typical case of this problem is shuffling a deck of cards. A good algorithm for shuffling is similar to selection sort, except that instead of moving the biggest item to the end of the list, an item is selected at random and moved to the end of the list. Here is a subroutine to shuffle an array of `ints`:

```

/**
 * Postcondition: The items in A have been rearranged into a random order.
 */
static void shuffle(int[] A) {

```

```

    for (int lastPlace = A.length-1; lastPlace > 0; lastPlace--) {
        // Choose a random location from among 0,1,...,lastPlace.
        int randLoc = (int)(Math.random()*(lastPlace+1));
        // Swap items in locations randLoc and lastPlace.
        int temp = A[randLoc];
        A[randLoc] = A[lastPlace];
        A[lastPlace] = temp;
    }
}

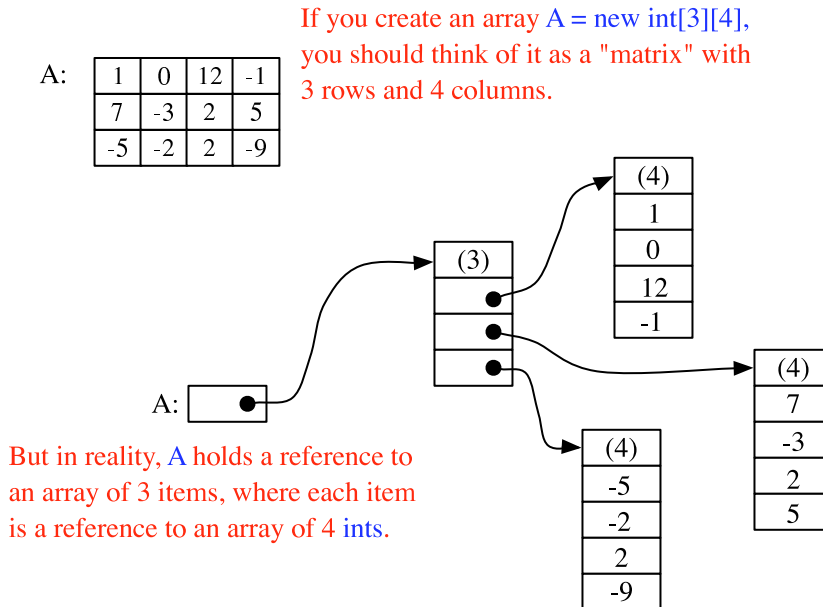
```

7.5 Multi-dimensional Arrays

ANY TYPE CAN BE USED as the base type of an array. You can have an array of **ints**, an array of *Strings*, an array of *Objects*, and so on. In particular, since an array type is a first-class Java type, you can have an array of arrays. For example, an array of **ints** has type `int[]`. This means that there is automatically another type, `int[][]`, which represents an “array of arrays of ints”. Such an array is said to be a *two-dimensional array*. Of course once you have the type `int[][]`, there is nothing to stop you from forming the type `int[][][]`, which represents a *three-dimensional array*—and so on. There is no limit on the number of dimensions that an array type can have. However, arrays of dimension three or higher are fairly uncommon, and I concentrate here mainly on two-dimensional arrays. The type `BaseType[][]` is usually read “two-dimensional array of *BaseType*” or “*BaseType* array array”.

7.5.1 Creating Two-dimensional Arrays

The declaration statement “`int[][] A;`” declares a variable named `A` of type `int[][]`. This variable can hold a reference to an object of type `int[][]`. The assignment statement “`A = new int[3][4];`” creates a new two-dimensional array object and sets `A` to point to the newly created object. As usual, the declaration and assignment could be combined in a single declaration statement “`int[][] A = new int[3][4];`”. The newly created object is an array of arrays-of-**ints**. The notation `int[3][4]` indicates that there are 3 arrays-of-**ints** in the array `A`, and that there are 4 **ints** in each array-of-**ints**. However, trying to think in such terms can get a bit confusing—as you might have already noticed. So it is customary to think of a two-dimensional array of items as a rectangular *grid* or *matrix* of items. The notation “`new int[3][4]`” can then be taken to describe a grid of **ints** with 3 rows and 4 columns. The following picture might help:



For the most part, you can ignore the reality and keep the picture of a grid in mind. Sometimes, though, you will need to remember that each row in the grid is really an array in itself. These arrays can be referred to as `A[0]`, `A[1]`, and `A[2]`. Each row is in fact a value of type `int[]`. It could, for example, be passed to a subroutine that asks for a parameter of type `int[]`.

The notation `A[1]` refers to one of the rows of the array `A`. Since `A[1]` is itself an array of **ints**, you can use another subscript to refer to one of the positions in that row. For example, `A[1][3]` refers to item number 3 in row number 1. Keep in mind, of course, that both rows and columns are numbered starting from zero. So, in the above example, `A[1][3]` is 5. More generally, `A[i][j]` refers to the grid position in row number `i` and column number `j`. The 12 items in `A` are named as follows:

<code>A[0][0]</code>	<code>A[0][1]</code>	<code>A[0][2]</code>	<code>A[0][3]</code>
<code>A[1][0]</code>	<code>A[1][1]</code>	<code>A[1][2]</code>	<code>A[1][3]</code>
<code>A[2][0]</code>	<code>A[2][1]</code>	<code>A[2][2]</code>	<code>A[2][3]</code>

`A[i][j]` is actually a variable of type **int**. You can assign integer values to it or use it in any other context where an integer variable is allowed.

It might be worth noting that `A.length` gives the number of rows of `A`. To get the number of columns in `A`, you have to ask how many **ints** there are in a row; this number would be given by `A[0].length`, or equivalently by `A[1].length` or `A[2].length`. (There is actually no rule that says that all the rows of an array must have the same length, and some advanced applications of arrays use varying-sized rows. But if you use the `new` operator to create an array in the manner described above, you'll always get an array with equal-sized rows.)

Three-dimensional arrays are treated similarly. For example, a three-dimensional array of **ints** could be created with the declaration statement `"int[][][] B = new int[7][5][11];"`. It's possible to visualize the value of `B` as a solid 7-by-5-by-11 block of cells. Each cell holds an **int** and represents one position in the three-dimensional array. Individual positions in the array can be referred to with variable names of the form `B[i][j][k]`. Higher-dimensional arrays

follow the same pattern, although for dimensions greater than three, there is no easy way to visualize the structure of the array.

It's possible to fill a multi-dimensional array with specified items at the time it is declared. Recall that when an ordinary one-dimensional array variable is declared, it can be assigned an "array initializer," which is just a list of values enclosed between braces, { and }. Array initializers can also be used when a multi-dimensional array is declared. An initializer for a two-dimensional array consists of a list of one-dimensional array initializers, one for each row in the two-dimensional array. For example, the array **A** shown in the picture above could be created with:

```
int[] [] A = { { 1, 0, 12, -1 },
               { 7, -3, 2, 5 },
               { -5, -2, 2, -9 }
             };
```

If no initializer is provided for an array, then when the array is created it is automatically filled with the appropriate value: zero for numbers, **false** for boolean, and **null** for objects.

7.5.2 Using Two-dimensional Arrays

Just as in the case of one-dimensional arrays, two-dimensional arrays are often processed using **for** statements. To process all the items in a two-dimensional array, you have to use one **for** statement nested inside another. If the array **A** is declared as

```
int[] [] A = new int[3][4];
```

then you could store a zero into each location in **A** with:

```
for (int row = 0; row < 3; row++) {
    for (int column = 0; column < 4; column++) {
        A[row][column] = 0;
    }
}
```

The first time the outer **for** loop executes (with **row** = 0), the inner **for** loop fills in the four values in the first row of **A**, namely **A**[0][0] = 0, **A**[0][1] = 0, **A**[0][2] = 0, and **A**[0][3] = 0. The next execution of the outer **for** loop fills in the second row of **A**. And the third and final execution of the outer loop fills in the final row of **A**.

Similarly, you could add up all the items in **A** with:

```
int sum = 0;
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 4; j++)
        sum = sum + A[i][j];
```

This could even be done with nested **for-each** loops. Keep in mind that the elements in **A** are objects of type **int**[], while the elements in each row of **A** are of type **int**:

```
int sum = 0;
for ( int[] row : A ) {           // For each row in A...
    for ( int item : row )        // For each item in that row...
        sum = sum + item;         // Add item to the sum.
}
```

To process a three-dimensional array, you would, of course, use triply nested `for` loops.

* * *

A two-dimensional array can be used whenever the data that is being represented can be arranged into rows and columns in a natural way. Often, the grid is built into the problem. For example, a chess board is a grid with 8 rows and 8 columns. If a class named *ChessPiece* is available to represent individual chess pieces, then the contents of a chess board could be represented by a two-dimensional array:

```
ChessPiece[] [] board = new ChessPiece[8][8];
```

Or consider the “mosaic” of colored rectangles used in an example in Subsection 4.6.2. The mosaic is implemented by a class named *MosaicCanvas.java*. The data about the color of each of the rectangles in the mosaic is stored in an instance variable named `grid` of type `Color[][]`. Each position in this grid is occupied by a value of type *Color*. There is one position in the grid for each colored rectangle in the mosaic. The actual two-dimensional array is created by the statement:

```
grid = new Color[ROWS][COLUMNS];
```

where `ROWS` is the number of rows of rectangles in the mosaic and `COLUMNS` is the number of columns. The value of the `Color` variable `grid[i][j]` is the color of the rectangle in row number `i` and column number `j`. When the color of that rectangle is changed to some color, `c`, the value stored in `grid[i][j]` is changed with a statement of the form “`grid[i][j] = c;`”. When the mosaic is redrawn, the values stored in the two-dimensional array are used to decide what color to make each rectangle. Here is a simplified version of the code from the *MosaicCanvas* class that draws all the colored rectangles in the grid. You can see how it uses the array:

```
int rowHeight = getHeight() / ROWS;
int colWidth = getWidth() / COLUMNS;
for (int row = 0; row < ROWS; row++) {
    for (int col = 0; col < COLUMNS; col++) {
        g.setColor( grid[row][col] ); // Get color from array.
        g.fillRect( col*colWidth, row*rowHeight,
                    colWidth, rowHeight );
    }
}
```

Sometimes two-dimensional arrays are used in problems in which the grid is not so visually obvious. Consider a company that owns 25 stores. Suppose that the company has data about the profit earned at each store for each month in the year 2006. If the stores are numbered from 0 to 24, and if the twelve months from January '06 through December '06 are numbered from 0 to 11, then the profit data could be stored in an array, `profit`, constructed as follows:

```
double[] [] profit = new double[25][12];
```

`profit[3][2]` would be the amount of profit earned at store number 3 in March, and more generally, `profit[storeNum][monthNum]` would be the amount of profit earned in store number `storeNum` in month number `monthNum`. In this example, the one-dimensional array `profit[storeNum]` has a very useful meaning: It is just the profit data for one particular store for the whole year.

Let's assume that the `profit` array has already been filled with data. This data can be processed in a lot of interesting ways. For example, the total profit for the company—for the whole year from all its stores—can be calculated by adding up all the entries in the array:

```

double totalProfit; // Company's total profit in 2006.

totalProfit = 0;
for (int store = 0; store < 25; store++) {
    for (int month = 0; month < 12; month++)
        totalProfit += profit[store][month];
}

```

Sometimes it is necessary to process a single row or a single column of an array, not the entire array. For example, to compute the total profit earned by the company in December, that is, in month number 11, you could use the loop:

```

double decemberProfit = 0.0;
for (storeNum = 0; storeNum < 25; storeNum++)
    decemberProfit += profit[storeNum][11];

```

Let's extend this idea to create a one-dimensional array that contains the total profit for each month of the year:

```

double[] monthlyProfit; // Holds profit for each month.
monthlyProfit = new double[12];

for (int month = 0; month < 12; month++) {
    // compute the total profit from all stores in this month.
    monthlyProfit[month] = 0.0;
    for (int store = 0; store < 25; store++) {
        // Add the profit from this store in this month
        // into the total profit figure for the month.
        monthlyProfit[month] += profit[store][month];
    }
}

```

As a final example of processing the profit array, suppose that we wanted to know which store generated the most profit over the course of the year. To do this, we have to add up the monthly profits for each store. In array terms, this means that we want to find the sum of each row in the array. As we do this, we need to keep track of which row produces the largest total.

```

double maxProfit; // Maximum profit earned by a store.
int bestStore;    // The number of the store with the
                  // maximum profit.

double total = 0.0; // Total profit for one store.

// First compute the profit from store number 0.
for (int month = 0; month < 12; month++)
    total += profit[0][month];

bestStore = 0; // Start by assuming that the best
maxProfit = total; // store is store number 0.

// Now, go through the other stores, and whenever we
// find one with a bigger profit than maxProfit, revise
// the assumptions about bestStore and maxProfit.
for (store = 1; store < 25; store++) {

    // Compute this store's profit for the year.

    total = 0.0;

```

```

    for (month = 0; month < 12; month++)
        total += profit[store][month];

    // Compare this store's profits with the highest
    // profit we have seen among the preceding stores.

    if (total > maxProfit) {
        maxProfit = total;    // Best profit seen so far!
        bestStore = store;    // It came from this store.
    }

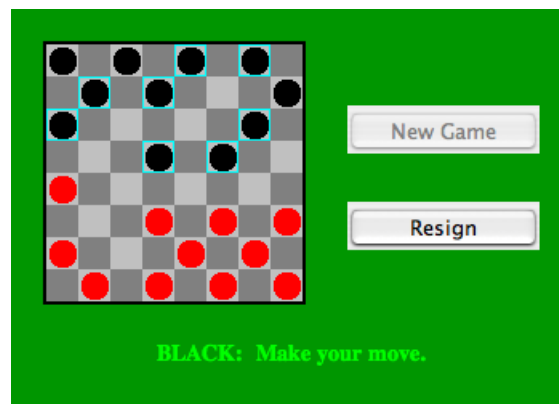
} // end for

// At this point, maxProfit is the best profit of any
// of the 25 stores, and bestStore is a store that
// generated that profit. (Note that there could also be
// other stores that generated exactly the same profit.)

```

7.5.3 Example: Checkers

For the rest of this section, we'll look at a more substantial example. We look at a program that lets two users play checkers against each other. A player moves by clicking on the piece to be moved and then on the empty square to which it is to be moved. The squares that the current player can legally click are hilited. The square containing a piece that has been selected to be moved is surrounded by a white border. Other pieces that can legally be moved are surrounded by a cyan-colored border. If a piece has been selected, each empty square that it can legally move to is hilited with a green border. The game enforces the rule that if the current player can jump one of the opponent's pieces, then the player must jump. When a player's piece becomes a king, by reaching the opposite end of the board, a big white "K" is drawn on the piece. You can try an applet version of the program in the on-line version of this section. Here is what it looks like:



I will only cover a part of the programming of this applet. I encourage you to read the complete source code, *Checkers.java*. At over 750 lines, this is a more substantial example than anything you've seen before in this course, but it's an excellent example of state-based, event-driven programming.

The data about the pieces on the board are stored in a two-dimensional array. Because of the complexity of the program, I wanted to divide it into several classes. In addition to the

main class, there are several nested classes. One of these classes is *CheckersData*, which handles the data for the board. It is mainly this class that I want to talk about.

The *CheckersData* class has an instance variable named `board` of type `int[][]`. The value of `board` is set to “`new int[8][8]`”, an 8-by-8 grid of integers. The values stored in the grid are defined as constants representing the possible contents of a square on a checkerboard:

```
static final int
    EMPTY = 0,           // Value representing an empty square.
    RED = 1,             // A regular red piece.
    RED_KING = 2,        // A red king.
    BLACK = 3,           // A regular black piece.
    BLACK_KING = 4;      // A black king.
```

The constants `RED` and `BLACK` are also used in my program (or, perhaps, misused) to represent the two players in the game. When a game is started, the values in the variable, `board`, are set to represent the initial state of the board. The grid of values looks like

	0	1	2	3	4	5	6	7
0	BLACK	EMPTY	BLACK	EMPTY	BLACK	EMPTY	BLACK	EMPTY
1	EMPTY	BLACK	EMPTY	BLACK	EMPTY	BLACK	EMPTY	BLACK
2	BLACK	EMPTY	BLACK	EMPTY	BLACK	EMPTY	BLACK	EMPTY
3	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY
4	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY
5	EMPTY	RED	EMPTY	RED	EMPTY	RED	EMPTY	RED
6	RED	EMPTY	RED	EMPTY	RED	EMPTY	RED	EMPTY
7	EMPTY	RED	EMPTY	RED	EMPTY	RED	EMPTY	RED

A black piece can only move “down” the grid. That is, the row number of the square it moves to must be greater than the row number of the square it comes from. A red piece can only move up the grid. Kings of either color, of course, can move in both directions.

One function of the *CheckersData* class is to take care of all the details of making moves on the board. An instance method named `makeMove()` is provided to do this. When a player moves a piece from one square to another, the values stored at two positions in the array are changed. But that’s not all. If the move is a jump, then the piece that was jumped is removed from the board. (The method checks whether the move is a jump by checking if the square to which the piece is moving is two rows away from the square where it starts.) Furthermore, a `RED` piece that moves to row 0 or a `BLACK` piece that moves to row 7 becomes a king. This is good programming: the rest of the program doesn’t have to worry about any of these details. It just calls this `makeMove()` method:

```
/**
 * Make the move from (fromRow,fromCol) to (toRow,toCol). It is
 * ASSUMED that this move is legal! If the move is a jump, the
 * jumped piece is removed from the board. If a piece moves
 * to the last row on the opponent’s side of the board, the
 * piece becomes a king.
 */
void makeMove(int fromRow, int fromCol, int toRow, int toCol) {
```

```

board[toRow][toCol] = board[fromRow][fromCol]; // Move the piece.
board[fromRow][fromCol] = EMPTY;

if (fromRow - toRow == 2 || fromRow - toRow == -2) {
    // The move is a jump. Remove the jumped piece from the board.
    int jumpRow = (fromRow + toRow) / 2; // Row of the jumped piece.
    int jumpCol = (fromCol + toCol) / 2; // Column of the jumped piece.
    board[jumpRow][jumpCol] = EMPTY;
}

if (toRow == 0 && board[toRow][toCol] == RED)
    board[toRow][toCol] = RED_KING; // Red piece becomes a king.
if (toRow == 7 && board[toRow][toCol] == BLACK)
    board[toRow][toCol] = BLACK_KING; // Black piece becomes a king.
} // end makeMove()

```

An even more important function of the *CheckersData* class is to find legal moves on the board. In my program, a move in a Checkers game is represented by an object belonging to the following class:

```

/**
 * A CheckersMove object represents a move in the game of
 * Checkers. It holds the row and column of the piece that is
 * to be moved and the row and column of the square to which
 * it is to be moved. (This class makes no guarantee that
 * the move is legal.)
 */
private static class CheckersMove {

    int fromRow, fromCol; // Position of piece to be moved.
    int toRow, toCol;     // Square it is to move to.

    CheckersMove(int r1, int c1, int r2, int c2) {
        // Constructor. Set the values of the instance variables.
        fromRow = r1;
        fromCol = c1;
        toRow = r2;
        toCol = c2;
    }

    boolean isJump() {
        // Test whether this move is a jump. It is assumed that
        // the move is legal. In a jump, the piece moves two
        // rows. (In a regular move, it only moves one row.)
        return (fromRow - toRow == 2 || fromRow - toRow == -2);
    }
} // end class CheckersMove.

```

The *CheckersData* class has an instance method which finds all the legal moves that are currently available for a specified player. This method is a function that returns an array of type *CheckersMove*[]. The array contains all the legal moves, represented as *CheckersMove* objects. The specification for this method reads

```

/**
 * Return an array containing all the legal CheckersMoves
 * for the specified player on the current board. If the player
 * has no legal moves, null is returned. The value of player
 * should be one of the constants RED or BLACK; if not, null
 * is returned. If the returned value is non-null, it consists
 * entirely of jump moves or entirely of regular moves, since
 * if the player can jump, only jumps are legal moves.
 */
CheckersMove[] getLegalMoves(int player)

```

A brief pseudocode algorithm for the method is

```

Start with an empty list of moves
Find any legal jumps and add them to the list
if there are no jumps:
    Find any other legal moves and add them to the list
if the list is empty:
    return null
else:
    return the list

```

Now, what is this “list”? We have to return the legal moves in an array. But since an array has a fixed size, we can’t create the array until we know how many moves there are, and we don’t know that until near the end of the method, after we’ve already made the list! A neat solution is to use an *ArrayList* instead of an array to hold the moves as we find them. In fact, I use an object defined by the parameterized type `ArrayList<CheckersMove>` so that the list is restricted to holding objects of type *CheckersMove*. As we add moves to the list, it will grow just as large as necessary. At the end of the method, we can create the array that we really want and copy the data into it:

```

Let "moves" be an empty ArrayList<CheckerMove>
Find any legal jumps and add them to moves
if moves.size() is 0:
    Find any other legal moves and add them to moves
if moves.size() is 0:
    return null
else:
    Let moveArray be an array of CheckersMoves of length moves.size()
    Copy the contents of moves into moveArray
    return moveArray

```

Now, how do we find the legal jumps or the legal moves? The information we need is in the `board` array, but it takes some work to extract it. We have to look through all the positions in the array and find the pieces that belong to the current player. For each piece, we have to check each square that it could conceivably move to, and check whether that would be a legal move. There are four squares to consider. For a jump, we want to look at squares that are two rows and two columns away from the piece. Thus, the line in the algorithm that says “Find any legal jumps and add them to moves” expands to:

```

For each row of the board:
    For each column of the board:
        if one of the player's pieces is at this location:
            if it is legal to jump to row + 2, column + 2
                add this move to moves

```



```

        if it is legal to jump to row - 2, column + 2
            add this move to moves
        if it is legal to jump to row + 2, column - 2
            add this move to moves
        if it is legal to jump to row - 2, column - 2
            add this move to moves

```

The line that says “Find any other legal moves and add them to moves” expands to something similar, except that we have to look at the four squares that are one column and one row away from the piece. Testing whether a player can legally move from one given square to another given square is itself non-trivial. The square the player is moving to must actually be on the board, and it must be empty. Furthermore, regular red and black pieces can only move in one direction. I wrote the following utility method to check whether a player can make a given non-jump move:

```

/**
 * This is called by the getLegalMoves() method to determine
 * whether the player can legally move from (r1,c1) to (r2,c2).
 * It is ASSUMED that (r1,c1) contains one of the player's
 * pieces and that (r2,c2) is a neighboring square.
 */
private boolean canMove(int player, int r1, int c1, int r2, int c2) {

    if (r2 < 0 || r2 >= 8 || c2 < 0 || c2 >= 8)
        return false; // (r2,c2) is off the board.

    if (board[r2][c2] != EMPTY)
        return false; // (r2,c2) already contains a piece.

    if (player == RED) {
        if (board[r1][c1] == RED && r2 > r1)
            return false; // Regular red piece can only move down.
        return true; // The move is legal.
    }
    else {
        if (board[r1][c1] == BLACK && r2 < r1)
            return false; // Regular black piece can only move up.
        return true; // The move is legal.
    }
}

} // end canMove()

```

This method is called by my `getLegalMoves()` method to check whether one of the possible moves that it has found is actually legal. I have a similar method that is called to check whether a jump is legal. In this case, I pass to the method the square containing the player's piece, the square that the player might move to, and the square between those two, which the player would be jumping over. The square that is being jumped must contain one of the opponent's pieces. This method has the specification:

```

/**
 * This is called by other methods to check whether
 * the player can legally jump from (r1,c1) to (r3,c3).
 * It is assumed that the player has a piece at (r1,c1), that
 * (r3,c3) is a position that is 2 rows and 2 columns distant
 * from (r1,c1) and that (r2,c2) is the square between (r1,c1)
 * and (r3,c3).

```

```

    */
    private boolean canJump(int player, int r1, int c1,
                           int r2, int c2, int r3, int c3) { . . .

```

Given all this, you should be in a position to understand the complete `getLegalMoves()` method. It's a nice way to finish off this chapter, since it combines several topics that we've looked at: one-dimensional arrays, *ArrayLists*, and two-dimensional arrays:

```

CheckersMove[] getLegalMoves(int player) {

    if (player != RED && player != BLACK)
        return null;

    int playerKing; // The constant for a King belonging to the player.
    if (player == RED)
        playerKing = RED_KING;
    else
        playerKing = BLACK_KING;

    ArrayList<CheckerMove> moves = new ArrayList<CheckerMove>();
        // Moves will be stored in this list.

    /* First, check for any possible jumps. Look at each square on
       the board. If that square contains one of the player's pieces,
       look at a possible jump in each of the four directions from that
       square. If there is a legal jump in that direction, put it in
       the moves ArrayList.
    */

    for (int row = 0; row < 8; row++) {
        for (int col = 0; col < 8; col++) {
            if (board[row][col] == player || board[row][col] == playerKing) {
                if (canJump(player, row, col, row+1, col+1, row+2, col+2))
                    moves.add(new CheckersMove(row, col, row+2, col+2));
                if (canJump(player, row, col, row-1, col+1, row-2, col+2))
                    moves.add(new CheckersMove(row, col, row-2, col+2));
                if (canJump(player, row, col, row+1, col-1, row+2, col-2))
                    moves.add(new CheckersMove(row, col, row+2, col-2));
                if (canJump(player, row, col, row-1, col-1, row-2, col-2))
                    moves.add(new CheckersMove(row, col, row-2, col-2));
            }
        }
    }

    /* If any jump moves were found, then the user must jump, so we
       don't add any regular moves. However, if no jumps were found,
       check for any legal regular moves. Look at each square on
       the board. If that square contains one of the player's pieces,
       look at a possible move in each of the four directions from
       that square. If there is a legal move in that direction,
       put it in the moves ArrayList.
    */

    if (moves.size() == 0) {
        for (int row = 0; row < 8; row++) {
            for (int col = 0; col < 8; col++) {
                if (board[row][col] == player || board[row][col] == playerKing) {

```

```

        if (canMove(player,row,col,row+1,col+1))
            moves.add(new CheckersMove(row,col,row+1,col+1));
        if (canMove(player,row,col,row-1,col+1))
            moves.add(new CheckersMove(row,col,row-1,col+1));
        if (canMove(player,row,col,row+1,col-1))
            moves.add(new CheckersMove(row,col,row+1,col-1));
        if (canMove(player,row,col,row-1,col-1))
            moves.add(new CheckersMove(row,col,row-1,col-1));
    }
}

/* If no legal moves have been found, return null. Otherwise, create
   an array just big enough to hold all the legal moves, copy the
   legal moves from the ArrayList into the array, and return the array.
*/

if (moves.size() == 0)
    return null;
else {
    CheckersMove[] moveArray = new CheckersMove[moves.size()];
    for (int i = 0; i < moves.size(); i++)
        moveArray[i] = moves.get(i);
    return moveArray;
}

} // end getLegalMoves

```

Exercises for Chapter 7

1. An example in Subsection 7.2.4 tried to answer the question, How many random people do you have to select before you find a duplicate birthday? The source code for that program can be found in the file *BirthdayProblemDemo.java*. Here are some related questions:

- How many random people do you have to select before you find **three** people who share the same birthday? (That is, all three people were born on the same day in the same month, but not necessarily in the same year.)
- Suppose you choose 365 people at random. How many different birthdays will they have? (The number could theoretically be anywhere from 1 to 365).
- How many different people do you have to check before you've found at least one person with a birthday on each of the 365 days of the year?

Write **three** programs to answer these questions. Each of your programs should simulate choosing people at random and checking their birthdays. (In each case, ignore the possibility of leap years.)

2. Write a program that will read a sequence of positive real numbers entered by the user and will print the same numbers in sorted order from smallest to largest. The user will input a zero to mark the end of the input. Assume that at most 100 positive numbers will be entered.
3. A ***polygon*** is a geometric figure made up of a sequence of connected line segments. The points where the line segments meet are called the ***vertices*** of the polygon. The ***Graphics*** class includes commands for drawing and filling polygons. For these commands, the coordinates of the vertices of the polygon are stored in arrays. If ***g*** is a variable of type ***Graphics*** then

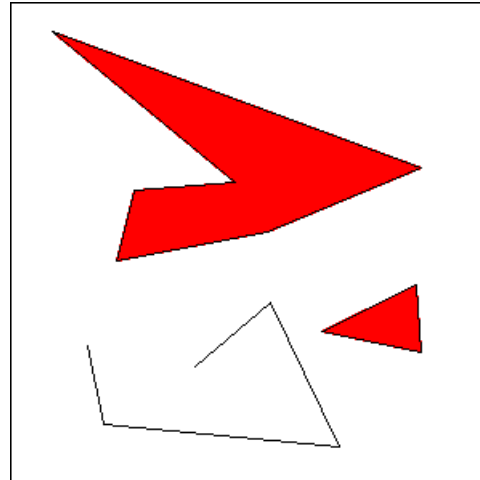
- ***g.drawPolygon(xCoords, yCoords, pointCt)*** will draw the outline of the polygon with vertices at the points ***(xCoords[0],yCoords[0])***, ***(xCoords[1],yCoords[1])***, ..., ***(xCoords[pointCt-1],yCoords[pointCt-1])***. The third parameter, ***pointCt***, is an **int** that specifies the number of vertices of the polygon. Its value should be 3 or greater. The first two parameters are arrays of type ***int[]***. Note that the polygon automatically includes a line from the last point, ***(xCoords[pointCt-1],yCoords[pointCt-1])***, back to the starting point ***(xCoords[0],yCoords[0])***.
- ***g.fillPolygon(xCoords, yCoords, pointCt)*** fills the interior of the polygon with the current drawing color. The parameters have the same meaning as in the ***drawPolygon()*** method. Note that it is OK for the sides of the polygon to cross each other, but the interior of a polygon with self-intersections might not be exactly what you expect.

Write a panel class that lets the user draw polygons, and use your panel as the content pane in an applet (or standalone application). As the user clicks a sequence of points, count them and store their x- and y-coordinates in two arrays. These points will be the vertices of the polygon. Also, draw a line between each consecutive pair of points to give the user some visual feedback. When the user clicks near the starting point, draw the

complete polygon. Draw it with a red interior and a black border. The user should then be able to start drawing a new polygon. When the user shift-clicks on the applet, clear it.

For this exercise, there is no need to store information about the contents of the applet. Do the drawing directly in the `mousePressed()` routine, and use the `getGraphics()` method to get a *Graphics* object that you can use to draw the line. (Remember, though, that this is considered to be bad style.) You will not need a `paintComponent()` method, since the default action of filling the panel with its background color is good enough.

Here is a picture of my solution after the user has drawn a few polygons:



4. For this problem, you will need to use an array of objects. The objects belong to the class *MovingBall*, which I have already written. You can find the source code for this class in the file *MovingBall.java*. A *MovingBall* represents a circle that has an associated color, radius, direction, and speed. It is restricted to moving in a rectangle in the (x,y) plane. It will “bounce back” when it hits one of the sides of this rectangle. A *MovingBall* does not actually move by itself. It’s just a collection of data. You have to call instance methods to tell it to update its position and to draw itself. The constructor for the *MovingBall* class takes the form

```
new MovingBall(xmin, xmax, ymin, ymax)
```

where the parameters are integers that specify the limits on the x and y coordinates of the ball. In this exercise, you will want balls to bounce off the sides of the applet, so you will create them with the constructor call

```
new MovingBall(0, getWidth(), 0, getHeight())
```

The constructor creates a ball that initially is colored red, has a radius of 5 pixels, is located at the center of its range, has a random speed between 4 and 12, and is headed in a random direction. There is one **problem** here: You can’t use this constructor until the width and height of the component are known. It would be OK to use it in the `init()` method of an applet, but not in the constructor of an applet or panel class. If you are using a panel class to display the ball, one slightly messy solution is to create the *MovingBall* objects in the panel’s `paintComponent()` method the first time that method is called. You can be sure that the size of the panel has been determined before `paintComponent()` is called. This is what I did in my own solution to this exercise.

If `ball` is a variable of type *MovingBall*, then the following methods are available:

- `ball.draw(g)` — draw the ball in a graphics context. The parameter, `g`, must be of type `Graphics`. (The drawing color in `g` will be changed to the color of the ball.)
- `ball.travel()` — change the (x,y)-coordinates of the ball by an amount equal to its speed. The ball has a certain direction of motion, and the ball is moved in that direction. Ordinarily, you will call this once for each frame of an animation, so the speed is given in terms of “pixels per frame”. Calling this routine does not move the ball on the screen. It just changes the values of some instance variables in the object. The next time the object’s `draw()` method is called, the ball will be drawn in the new position.
- `ball.headTowards(x,y)` — change the direction of motion of the ball so that it is headed towards the point (x,y). This does not affect the speed.

These are the methods that you will need for this exercise. There are also methods for setting various properties of the ball, such as `ball.setColor(color)` for changing the color and `ball.setRadius(radius)` for changing its size. See the source code for more information.

For this exercise, you should create an applet that shows an animation of balls bouncing around on a black background. Use a *Timer* to drive the animation. (See Subsection 6.5.1.) Use an array of type `MovingBall[]` to hold the data for the balls. In addition, your program should listen for mouse and mouse motion events. When the user presses the mouse or drags the mouse, call each of the ball’s `headTowards()` methods to make the balls head towards the mouse’s location. My solution uses 50 balls and a time delay of 50 milliseconds for the timer.

5. The sample program *RandomArtPanel.java* from Subsection 6.5.1 shows a different random “artwork” every four seconds. There are three types of “art”, one made from lines, one from circles, and one from filled squares. However, the program does not save the data for the picture that is shown on the screen. As a result, the picture cannot be redrawn when necessary. In fact, every time `paintComponent()` is called, a new picture is drawn.

Write a new version of *RandomArtPanel.java* that saves the data needed to redraw its pictures. The `paintComponent()` method should simply use the data to draw the picture. New data should be recomputed only every four seconds, in response to an event from the timer that drives the program.

To make this interesting, write a separate class for each of the three different types of art. Also write an abstract class to serve as the common base class for the three classes. Since all three types of art use a random gray background, the background color can be defined in their superclass. The superclass also contains a `draw()` method that draws the picture; this is an abstract method because its implementation depends on the particular type of art that is being drawn. The abstract class can be defined as:

```
private abstract class ArtData {
    Color backgroundColor; // The background color for the art.
    ArtData() { // Constructor sets background color to be a random gray.
        int x = (int)(256*Math.random());
        backgroundColor = new Color( x, x, x, );
    }
    abstract void draw(Graphics g); // Draws this artwork.
}
```

Each of the three subclasses of `ArtData` must define its own `draw()` method. It must also define instance variables to hold the data necessary to draw the picture. I suggest that you should create random data for the picture in the constructor of the class, so that constructing the object will automatically create the data for a random artwork. (One problem with this is that you can't create the data until you know the size of the panel, so you can't create an `artdata` object in the constructor of the panel. One solution is to create an `artdata` object at the beginning of the `paintComponent()` method, if the object has not already been created.) In all three subclasses, you will need to use several arrays to store the data.

The file *RandomArtPanel.java* only defines a panel class. A main program that uses this panel can be found in *RandomArt.java*, and an applet that uses it can be found in *RandomArtApplet.java*.

6. Write a program that will read a text file selected by the user, and will make an alphabetical list of all the different words in that file. All words should be converted to lower case, and duplicates should be eliminated from the list. The list should be written to an output file selected by the user. As discussed in Subsection 2.4.5, you can use *TextIO* to read and write files. Use a variable of type `ArrayList<String>` to store the words. (See Subsection 7.3.4.) It is not easy to separate a file into words as you are reading it. You can use the following method:

```
/**
 * Read the next word from TextIO, if there is one. First, skip past
 * any non-letters in the input. If an end-of-file is encountered before
 * a word is found, return null. Otherwise, read and return the word.
 * A word is defined as a sequence of letters. Also, a word can include
 * an apostrophe if the apostrophe is surrounded by letters on each side.
 * @return the next word from TextIO, or null if an end-of-file is
 * encountered
 */
private static String readNextWord() {
    char ch = TextIO.peek(); // Look at next character in input.
    while (ch != TextIO.EOF && ! Character.isLetter(ch)) {
        TextIO.getAnyChar(); // Read the character.
        ch = TextIO.peek(); // Look at the next character.
    }
    if (ch == TextIO.EOF) // Encountered end-of-file
        return null;
    // At this point, we know the next character is a letter, so read a word.
    String word = ""; // This will be the word that is read.
    while (true) {
        word += TextIO.getAnyChar(); // Append the letter onto word.
        ch = TextIO.peek(); // Look at next character.
        if ( ch == '\'' ) {
            // The next character is an apostrophe. Read it, and
            // if the following character is a letter, add both the
            // apostrophe and the letter onto the word and continue
            // reading the word. If the character after the apostrophe
            // is not a letter, the word is done, so break out of the loop.
            TextIO.getAnyChar(); // Read the apostrophe.
            ch = TextIO.peek(); // Look at char that follows apostrophe.
            if (Character.isLetter(ch)) {
```

```

        word += "\"" + TextIO.getAnyChar();
        ch = TextIO.peek(); // Look at next char.
    }
    else
        break;
}
if ( ! Character.isLetter(ch) ) {
    // If the next character is not a letter, the word is
    // finished, so break out of the loop.
    break;
}
// If we haven't broken out of the loop, next char is a letter.
}
return word; // Return the word that has been read.
}

```

Note that this method will return `null` when the file has been entirely read. You can use this as a signal to stop processing the input file.

7. The game of Go Moku (also known as Pente or Five Stones) is similar to Tic-Tac-Toe, except that it played on a much larger board and the object is to get five squares in a row rather than three. Players take turns placing pieces on a board. A piece can be placed in any empty square. The first player to get five pieces in a row—horizontally, vertically, or diagonally—wins. If all squares are filled before either player wins, then the game is a draw. Write a program that lets two players play Go Moku against each other.

Your program will be simpler than the *Checkers* program from Subsection 7.5.3. Play alternates strictly between the two players, and there is no need to hilight the legal moves. You will only need two classes, a short panel class to set up the interface and a *Board* class to draw the board and do all the work of the game. Nevertheless, you will probably want to look at the source code for the checkers program, *Checkers.java*, for ideas about the general outline of the program.

The hardest part of the program is checking whether the move that a player makes is a winning move. To do this, you have to look in each of the four possible directions from the square where the user has placed a piece. You have to count how many pieces that player has in a row in that direction. If the number is five or more in any direction, then that player wins. As a hint, here is part of the code from my applet. This code counts the number of pieces that the user has in a row in a specified direction. The direction is specified by two integers, `dirX` and `dirY`. The values of these variables are 0, 1, or -1, and at least one of them is non-zero. For example, to look in the horizontal direction, `dirX` is 1 and `dirY` is 0.

```

int ct = 1; // Number of pieces in a row belonging to the player.

int r, c;   // A row and column to be examined

r = row + dirX; // Look at square in specified direction.
c = col + dirY;
while ( r >= 0 && r < 13 && c >= 0 && c < 13
        && board[r][c] == player ) {
    // Square is on the board, and it
    // contains one of the players's pieces.
    ct++;
}

```



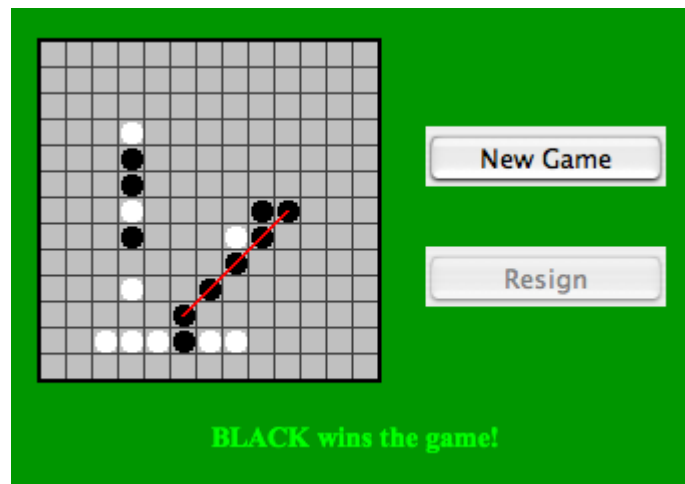
```

    r += dirX; // Go on to next square in this direction.
    c += dirY;
}

r = row - dirX; // Now, look in the opposite direction.
c = col - dirY;
while ( r >= 0 && r < 13 && c >= 0 && c < 13
        && board[r][c] == player ) {
    ct++;
    r -= dirX; // Go on to next square in this direction.
    c -= dirY;
}

```

Here is a picture of my program. It uses a 13-by-13 board. You can do the same or use a normal 8-by-8 checkerboard.



Quiz on Chapter 7

1. What does the computer do when it executes the following statement? Try to give as complete an answer as possible.

```
Color[] palette = new Color[12];
```

2. What is meant by the *basetype* of an array?
3. What does it mean to sort an array?
4. What is the main advantage of binary search over linear search? What is the main disadvantage?
5. What is meant by a *dynamic array*? What is the advantage of a dynamic array over a regular array?
6. Suppose that a variable `strlst` has been declared as

```
ArrayList<String> strlst = new ArrayList<String>();
```

Assume that the list is not empty and that all the items in the list are non-null. Write a code segment that will find and print the string in the list that comes first in lexicographic order. How would your answer change if `strlst` were declared to be of type *ArrayList* instead of *ArrayList<String>*?

7. What is the purpose of the following subroutine? What is the meaning of the value that it returns, in terms of the value of its parameter?

```
static String concat( String[] str ) {
    if (str == null)
        return "";
    String ans = "";
    for (int i = 0; i < str.length; i++) {
        ans = ans + str[i];
    }
    return ans;
}
```

8. Show the exact output produced by the following code segment.

```
char[][] pic = new char[6][6];
for (int i = 0; i < 6; i++)
    for (int j = 0; j < 6; j++) {
        if ( i == j || i == 0 || i == 5 )
            pic[i][j] = '*';
        else
            pic[i][j] = '.';
    }
for (int i = 0; i < 6; i++) {
    for (int j = 0; j < 6; j++)
        System.out.print(pic[i][j]);
    System.out.println();
}
```

9. Write a complete subroutine that finds the largest value in an array of `ints`. The subroutine should have one parameter, which is an array of type `int[]`. The largest number in the array should be returned as the value of the subroutine.
10. Suppose that temperature measurements were made on each day of 1999 in each of 100 cities. The measurements have been stored in an array

```
int[][] temps = new int[100][365];
```

where `temps[c][d]` holds the measurement for city number `c` on the d^{th} day of the year. Write a code segment that will print out the average temperature, over the course of the whole year, for each city. The average temperature for a city can be obtained by adding up all 365 measurements for that city and dividing the answer by 365.0.

11. Suppose that a class, *Employee*, is defined as follows:

```
class Employee {
    String lastName;
    String firstName;
    double hourlyWage;
    int yearsWithCompany;
}
```

Suppose that data about 100 employees is **already** stored in an array:

```
Employee[] employeeData = new Employee[100];
```

Write a code segment that will output the first name, last name, and hourly wage of each employee who has been with the company for 20 years or more.

12. Suppose that `A` has been declared and initialized with the statement

```
double[] A = new double[20];
```

and suppose that `A` has **already** been filled with 20 values. Write a program segment that will find the average of all the **non-zero** numbers in the array. (The average is the sum of the numbers, divided by the number of numbers. Note that you will have to count the number of non-zero entries in the array.) Declare any variables that you use.

