7 Arrays

Overview

In this chapter you will learn about arrays in the context of a personnel database system for a commercial company. An array lets you store a large number of values of the same type. Arrays make it easy to implement the Vic class for Chapters Two and Three (since a sequence stores many CDs), the Network classes for Chapter Five (since a Position object stores many Nodes), and the Queue class for Chapter Six (since a queue stores many Objects). You will also see an elementary use of disk files for input.

- Sections 7.1-7.2 design the Worker class of objects and develop two of the many small programs that might be in the Personnel Database software suite.
- Sections 7.3-7.4 introduce arrays and apply them to another program using Workers and to the implementation of the Worker class.
- Sections 7.5-7.6 discuss the partially-filled array concept and develop many instance methods for an object that contains a list of Workers. The list varies in size.
- Sections 7.7-7.11 develop an elementary sorting method for Workers and describe implementations of the Vic, Network, and Queue classes used in earlier chapters. The Sun standard library class ArrayList is featured in Section 7.11.

You only need to study Sections 7.1-7.6 to be able to understand the rest of the book. The other sections are to give you additional practice working with arrays. Reminder: The answers to the unstarred exercises are at the end of the chapter.

7.1 Analysis And Design Of The Worker Class

A company offers your consulting firm a contract to develop software to handle all their personnel matters. You head a software engineering team at your firm that will design and build the software. The others on your team are all junior programmers, so you will have to create the basic design and delegate the details to the others. Your talks with the company executives tell you that the software must, among other things:

- Maintain a data file of workers, accepting additions and deletions.
- Go through the data file and print various reports, some with the workers in order of names, some in order of paycheck size, some in order of birth year, etc.
- Input hours worked for each worker and print pay checks, with taxes withheld.

When you look over your notes from your first meeting with the company's management, you realize that this software can be mostly a number of different small to medium main programs working with a common set of objects, including:

- A virtual worker, representing a single employee.
- A virtual input device, representing the keyboard or a data file.
- A virtual output device, representing the screen or printer.
- A virtual file cabinet, storing data for many workers.

Description of the Worker class

You decide to design the Worker class first. You need to be able to perform at least the following operations with Worker objects. As you develop the main logic of various programs, you expect to see the need for additional operations.

- Create a Worker object from a string of characters received as input from a disk file or from the keyboard.
- Update the hours worked each day, so a Worker can compute its pay for the week.
- Access one Worker's data: name, week's pay, birth year, and others.
- Compare two Workers to see which comes first alphabetically by name.
- Obtain a printable form of a Worker for use in reports.

This leads you to develop documentation for the Worker class as the sketch shown in Listing 7.1, with more to be added later by your subordinates. The methods in this documentation are **stubbed**. That is, the body of each has the minimum necessary to make it compile -- nothing if a void method, a simple return otherwise (null if it returns an object). The name `toString` is the conventional Java name for the method that returns the String representation of an object. If the constructor is given null or the empty String, it signals this by having `getName` return null.

Listing 7.1 Documentation for the Worker class, first draft

```
public class Worker implements Comparable //stubbed documentation
{
   /** Create a Worker from an input String, a single line with
    *  first name, last name, year, and rate in that order.
    *  If the String value is bad, the name is made null. */
   public Worker (String input)                              { }

   /** Return the first name plus last name of the Worker.
    *  But return null if it does not represent a real Worker. */
   public String getName()                       { return null; }

   /** Return the Worker's birth year. */
   public int getBirthYear()                        { return 0; }

   /** Return the Worker's gross pay for the week. */
   public double seeWeeksPay()                      { return 0; }

   /** Record the hours worked in the most recent day. */
   public void addHoursWorked (double hoursWorked)           { }


   /** Return 0 if equal, negative if the executor is before
    *  ob (executor has an earlier last name or else the same
    *  last name and an earlier first name), positive otherwise.
    *  Precondition:  ((Worker) ob).getName() is non-null. */
   public int compareTo (Object ob)                 { return 0; }

   /** Tell whether one Worker has the same name as another. */
   public boolean equals (Worker ob)            { return false; }

   /** Return a String value containing most or all of the
    *  Worker's data, suitable for printing in a report. */
   public String toString()                      { return null; }
}
```

The condition `x.compareTo(y) < 0` for objects tests whether `x` comes before `y` in a reasonable ordering, and `x.compareTo(y) >= 0` tests whether `x` equals or comes after `y`. They are analogous to `x < y` and `x >= y` for numeric variables. The presence of this method in the Worker class allows it to `implement Comparable` (discussed in Section 6.3). Both `compareTo` and `equals` have the same name and meaning as for String objects: `x.compareTo(y) < 0` for Strings of lowercase characters tests whether `x` is alphabetically before `y`.

Reading from a BufferedReader

Information about hundreds or thousands of workers is quite naturally stored in a file on a hard disk, so this Personnel Database software needs a class that provides methods for reading from a disk file. We will call this the Buffin class (Buffin is a subclass of the Sun standard library BufferedReader class). The Buffin class is developed in Section 10.3, but you do not have to study that material now. All you need for this chapter is how to use the following two Buffin methods that encapsulate reading data from a text file:

```
   Buffin file = new Buffin ("A:\\someFile.dat");
```

The above opens a channel to the data file named `someFile.dat` on the floppy disk and creates a special BufferedReader object. The `\\` sequence is what you must write inside quotes to indicate a single backslash character.

```
   String value = file.readLine();
```

The above requests the next available line from that text file. If you have already read all the information from the file, or if there is an error reading the next line, the `readLine` method returns the null value as a signal.

If the runtime system cannot open the file for some reason, or if you call `new Buffin` with the empty String as the parameter, all input will be obtained from the keyboard. This feature of the Buffin class lets you write a multipurpose method with a Buffin parameter that gets data from either a file or the keyboard.

If, after reading some data from a file, you decide you want to start over from the beginning of the file, just execute `file = new Buffin ("A:\\someFile.dat")` again.

Finding the last of three workers

Listing 7.2 (see next page) is a short application program that demonstrates the use of Worker objects. The main method creates three Worker objects with input from a file named `workers.txt`. Then it prints out the Worker with the alphabetically last name. The task of finding the last of three values is sufficiently complex that it deserves a separate method. We put this method in a separate utilities class named CompOp, for operations on Comparable objects. The `ordered` method of Listing 6.3 would be another candidate for this CompOp class. The class diagram is in Figure 7.1.

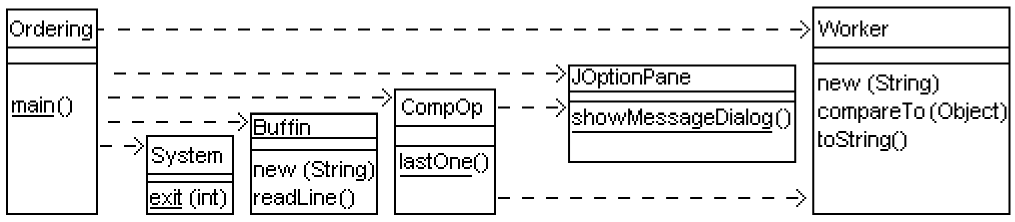


Figure 7.1 UML class diagram for the Ordering program and lastOne

Listing 7.2 Application program using Workers (compile in two files)

```
import javax.swing.JOptionPane;

public class Ordering
{
   /** Ask the user for three Workers.  Print the one that is
    *  last in alphabetical order of names. */

   public static void main (String[ ] args)
   {
      javax.swing.JOptionPane.showMessageDialog (null,
                  "reading 3 workers, printing the last.");
      Buffin infile = new Buffin ("workers.txt");
      Worker first  = new Worker (infile.readLine());
      Worker second = new Worker (infile.readLine());
      Worker third  = new Worker (infile.readLine());
      if (third.getName() != null)  // if insufficient input
      {  javax.swing.JOptionPane.showMessageDialog (null,
                 "The alphabetically last is "
                 + CompOp.lastOne (first, second, third));
      }
      System.exit (0);
   }  //======================
}  //#########################################################


public class CompOp
{
   /** Return the String representation of the last/largest of
    *  three Comparable values. */

   public static String lastOne (Comparable first,
                        Comparable second, Comparable third)
   {  Comparable last =  first.compareTo (second) >= 0
                         ?  first   :  second;
      if (last.compareTo (third) < 0)
         last = third;
      return last.toString();
   }  //======================
}
```

The `lastOne` method has three parameters of Comparable type. When called from Ordering's main method, the method queries the `compareTo` method in the Worker class twice to decide which of the three is alphabetically last, since the Comparable objects are instances of the Worker class. Then it prints the result.

Polymorphism in the Ordering class

You may wonder how `lastOne` can compile correctly when it calls `last.toString()` for one of the Comparable objects but `toString` is a Worker method. However, the Object class has a `toString` method to produce a textual representation of an object. The compiler accepts the phrase `last.toString()` because every Comparable object is in a subclass of Object. So it knows that the runtime system can use Object's `toString` method definition unless the object referred to by `last` has its own `toString` method to override the one from the Object class.

The expression `last.toString()` in the `lastOne` method may call the method in the Worker class or the method in the String class or in some other class, depending on the kind of Comparable objects passed as parameters. For instance, if some class contains the phrase `CompOp.lastOne ("chris", "sam", "pat")`, then `last.toString()` is the String equivalent of "sam", which is just "sam" itself. This is an example of polymorphism -- a method call that could invoke any of several different methods at different times during the execution of some program.

Key point The compiler does not look at the object referenced by the variable; it only looks at the variable. The object does not exist yet (since it is created at runtime).

Suppose you run a program that calls `CompOp.lastOne` for three String values. When the runtime system evaluates `last.toString()` in that program, it sees that `last` refers to a String object. So it calls the `toString` method in the String class. That produces the characters in the string.

Key point The runtime system does not look at the variable; it only looks at the object referenced by the variable. The variable does not exist anymore (since it is in the source file `CompOp.java`, not necessarily in the compiled file `CompOp.class`).

When the runtime system decides which method to call, it does not look at the type of the variable, it looks at the type of the object. The Comparable variable `last` in the source file is a **polymorph** -- it will sometimes refer to a Worker object and sometimes to a String object (or some other kind of object). "Polymorph" comes from the Greek, meaning a thing that takes on several different forms (a sort of shape-changer).

Caution If a class contains a method with the same signature as a method in its superclass, then the two methods must have the same return type. Also, you cannot have one be an instance method and the other be a class method.

Exercise 7.1 Write an independent method `public static void averageDailyPay (Worker karl)`: It prints the given Worker's name and average daily pay (assuming 5 days in a work week). Throw an Exception if `karl` is null.
Exercise 7.2 If some method has a Worker variable named `chris` and an int variable named `time`, and it mentions `chris.addHoursWorked (time)`, how would you refer to the value of `time` within the body of the `addHoursWorked` method? How would you refer to the value of `chris` within the body of `addHoursWorked`?
Exercise 7.3 Revise the `lastOne` method in Listing 7.2 to return the String form of the youngest of three Workers.
Exercise 7.4 How would you revise Listing 7.2 to have `lastOne` print the value of `last` instead of returning it? Have one statement in the main method that calls that revised `lastOne` to have it printed. Do not change the overall effect of the program.
Exercise 7.5* Write an application program to read three Worker values from the keyboard and then print the average age of those three Workers.
Exercise 7.6* Revise the body of the `lastOne` method in Listing 7.2 to return a String value that is (a) the name of that one of the three Workers who has the lowest weekly pay, followed by (b) the amount of that Worker's pay.

7.2 Analysis And Design Example: Finding The Alphabetically First

Problem A data file named `workers.txt` lists thousands of workers. Write a method to find and return the alphabetically first name of all the workers.

Analysis (clarifying what to do) You verify that the data file follows a standard format for worker data, so you can retrieve worker information from the data file as a single line to be processed by a Worker constructor. Then the `getName` method applied to that worker information produces the worker's name in the form needed to decide on alphabetical ordering, except it produces null when there was no Worker value to read.

No, wait! The `getName` method produces names with the first name first, so a comparison will find the person with the earliest first name. That is probably not what the client wants. You check back with the client and find that workers are to be ordered by last name, with the first name only to break ties. So you need to use the `compareTo` method in the Worker class instead of the `getName` method.

A program should be **robust**, which means that it should handle unexpected or unusual situations in a reasonable manner without crashing (at least terminate gracefully). For this program, the file may be empty, in which case the first Worker object will have no name. In that case, you could just have the program print an appropriate message.

Logic design (deciding how to do it) An initial plan for solving this problem could be as shown in the accompanying design block. It uses a sentinel-controlled loop pattern, i.e., the loop terminates when an "artificial" input value is received that indicates there are no more actual input values available. In this case, a Worker with a null name is the sentinel.

STRUCTURED NATURAL LANGUAGE DESIGN for the findEarliestWorker method

```
1.  Create the virtual data file connected to workers.txt.
2.  Attempt to read in the first worker value.
3.  If that first worker value does not even exist, then...
        Produce an appropriate message.
4.  Otherwise...
        4a.  Make answerSoFar equal to that Worker object.
        4b.  For each remaining worker you read from the data file, do...
                 If its name is before answerSoFar's, then...
                     Replace answerSoFar by that Worker object.
        4c.  Produce the name of the ending value of answerSoFar.
```

Object design and implementation To make the logic reusable, we put it in an instance method in a class named PersonnelData. Creating a PersonnelData object does step 1 of the design block, since each such object opens the input file named "workers.dat" for reading when it is constructed. Each step of this design translates to one or two Java statements, using classes you have already defined, so no further analysis is needed. The logic as expressed in Java is shown in Listing 7.3 (see next page).

The lower part of Listing 7.3 is an application program for testing purposes: Tell the user at the keyboard what program is about to run (line 13), create a PersonnelData object (line 14), ask it to `findEarliestWorker`, print it (line 15), then terminate the program.

Reminder: `Worker data = new Worker (itsFile.readLine());` is the same as the following two statements, but it avoids having an extra variable used in only one line:

```
String input = itsFile.readLine();
Worker data = new Worker (input);
```

Listing 7.3 PersonnelData class of objects, part 1

```
import javax.swing.JOptionPane;

public class PersonnelData
{
   private Buffin itsFile = new Buffin ("workers.txt");

   /** Read a file and return the name of the Worker that is
    *  alphabetically first. */

   public String findEarliestWorker()
   {  Worker answerSoFar = new Worker (itsFile.readLine());  //1
      if (answerSoFar.getName() == null)                     //2
         return "no workers!";                               //3
      else                                                   //4
      {  Worker data = new Worker (itsFile.readLine());      //5
         while (data.getName() != null)                      //6
         {  if (data.compareTo (answerSoFar) < 0)            //7
               answerSoFar = data;                           //8
            data = new Worker (itsFile.readLine());          //9
         }                                                   //10
         return answerSoFar.getName() + " is first of all."; //11
      }                                                      //12
   }  //======================
}  //#######################################################


class EarliestWorkerTester
{
   public static void main (String[ ] args)
   {  JOptionPane.showMessageDialog (null, "finding the first");
      String out = new PersonnelData().findEarliestWorker(); //14
      JOptionPane.showMessageDialog (null, out);             //15
      System.exit (0);                                       //16
   }  //======================
}
```

Similarly, line 14 is the same as the following two statements:

```
   PersonnelData base = new PersonnelData();
   String out = base.FindEarliestWorker();
```

This book generally avoids creating a variable to hold a value that is only used once, unless the alternative is a statement that is more than one line long. This speeds execution without making the coding any less clear. Note that lines 14 and 15 could have been combined into a single two-line statement, but that seemed to be too complex.

Review of Structured Natural Language Design features The SNL principle is that the entire design of a method should be expressed in ordinary natural language (English or whatever you are most comfortable in), with three exceptions:

1. Storage locations should be named where it clarifies the logic (e.g., `answerSoFar`).
2. Use `If whatever then... <stuff> Otherwise... <stuff>` where a choice is made (sometimes you do not need the `Otherwise` part). Indent for the `<stuff>` in each case.
3. Use `For each whatever do... <stuff>` for repetition (or perhaps use `repeat`). Indent for the `<stuff>`.

Test plans

You will need to test your program. Most people tend not to think seriously about the tests until they have finished the implementation. That is too late for maximum efficiency in the development of software.

A **test set** is a set of input values used for one run of the program, together with the expected output values (what you believe should be produced by the program for that input). You test your program by giving it the input values and then making sure that the output it produces is what you expected.

A test set for the EarliestWorkerTester program could have the following eight Worker names. At the right of each name in this list is the value that `answerSoFar` should have after the Worker is read. The two lines of output from the program are boldfaced:

```
finding the first
George Meaney        answerSoFar is George Meaney
Utah Phillips        answerSoFar is George Meaney
Joe Hill             answerSoFar is Joe Hill
Mother Jones         answerSoFar is Joe Hill
Walter Reuther       answerSoFar is Joe Hill
Marcus Garvey        answerSoFar is Marcus Garvey
Bill Heywood         answerSoFar is Marcus Garvey
Samuel Gompers       answerSoFar is Marcus Garvey
Marcus Garvey is first of all.
```

A **test plan** is a large enough number of test sets that you can be confident that almost all of the bugs are out of the program. Begin to develop your Test plan when you are finishing the Analysis stage of the development. Do not even start the Design stage until you have your Test plan at least partially planned. Three reasons why it is important to consider the Test plan before the Design stage are as follows:

1. As you make up a test set, you are reviewing the results of your Analysis stage. You are likely to find many of the errors that are in your analysis. And it is far more efficient to remove the errors before you design rather than after.
2. You get a deeper understanding of the specifications, which means that you will find it easier to design the software.
3. If you have much difficulty making up the test sets, that shows that you do not understand the specifications well enough, so you could not possibly make up a good design anyway.

Exercise 7.7 Modify Listing 7.3 to print the alphabetically last name.
Exercise 7.8 Add to Listing 7.3 coding to also print the total amount paid to all Workers this week.
Exercise 7.9 Add to Listing 7.3 coding to tell whether every worker was born in the same century.
Exercise 7.10* Modify Listing 7.3 to print the earliest birth year. Do not store the worker with the earliest birth year, only the year itself.
Exercise 7.11* Modify Listing 7.3 to print both the name and the birth year of the person with the alphabetically first name.
Exercise 7.12* Draw the UML class diagram for `FindEarliestWorker`.
Exercise 7.13** Add to Listing 7.3 coding to tell whether the workers are in increasing order of names as determined by `compareTo`. A sequence of less than two values is considered to be in increasing order in any case.

7.3 An Array Of Counters, An Array Of Strings

Problem A data file named `workers.txt` lists thousands of workers, all with birth years in the 1960s. Write a method that prints the number that were born in each one of the 10 years 1960 through 1969.

Analysis (clarifying what to do) You verify that the data file follows the standard format for worker data, so you can retrieve worker information from the data file as a single line to be processed by a Worker constructor. You should print ten lines after reading through the entire file. The first line should say "1960 had XX workers", the second should say "1961 had XX workers", etc., where XX is filled in by the non-negative count of workers.

What if some birth years in the data file are wrong? You decide to treat the assertion that all birth years are in the 1960s as a prediction to be verified rather than as a precondition to be trusted. That means that your program will be more robust because it will not crash if the prediction is wrong. This compensates for being less efficient by spending time checking that the assertion is true.

To handle the exceptional workers, you can print those whose birth years are not in the 1960s as they arise, or print the total number of such exceptions after finishing the file, or simply say whether such exceptions exist. The robustness-check is left as an exercise.

Logic design (deciding how to do it) You need to have a variable that counts the number born in 1960, a variable that counts the number born in 1961, a variable that counts the number born in 1962, etc. Read the data file one worker at a time using `readLine`. For each worker you read, add 1 to the counter for his or her year. An initial plan for solving this problem is in the accompanying design block. It uses the standard sentinel-controlled loop pattern you saw for Listing 7.3.

STRUCTURED NATURAL LANGUAGE DESIGN for the countBirthYears method

```
1.  Create the virtual data file connected to workers.txt.
2.  Create ten counters, one for each year, all initially zero.
3.  For each worker that you read in from the file, do...
        3a.  Add 1 to the counter corresponding to her or his birth year.
4.  Print the list of 10 counters with explanatory comments.
```

Using an array of counters

You could have 10 int variables named `count0`, `count1`, `count2`, etc. Then you would use the last digit of the birth year to decide which counter to increment. So you would increment `count4` if the birth year is 1964, `count7` if the birth year is 1967. But this makes the logic extremely clumsy.

You can instead use an **array** of 10 int variables, one for each year. When you execute

```
int[ ] count;
count = new int[10];
```

you create 10 counter variables at once (the `[ ]` characters are called **brackets**). Inside the program, you may refer to the one for 1960 as `count[0]`, the one for 1961 as `count[1]`, the one for 1962 as `count[2]`, etc. The `int[]count` part declares `count` to be an object reference. The `count = new int[10]` part creates an object having 10 int variables, each initialized to zero, and has `count` refer to that object. The array is shown in Figure 7.2 (the initialized values and also after counting seven years).

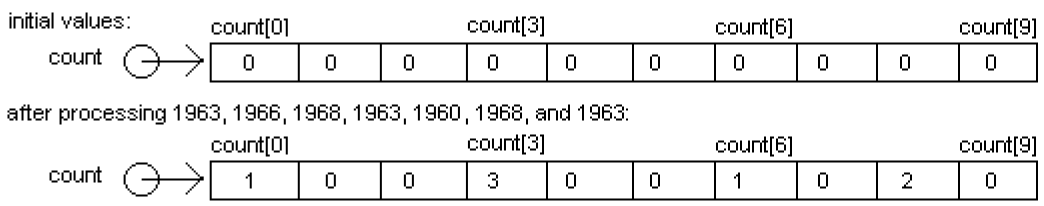


Figure 7.2 An array of 10 counters

Development of the sub-logic

Step 3a of the main logic, finding the right counter to add 1 to, is the step that is most complex. It can be designed as shown in the accompanying design block.

STRUCTURED NATURAL LANGUAGE DESIGN for step 3a
1. Get the year that the current worker was born.
2. Subtract 1960 from it to get a number ranging from 0 to 9.
3. Add 1 to the counter indexed by that number.

The implementation of this logic is in Listing 7.4. It uses constants for 1960 and 10 to make the logic clearer to the reader and to make the logic easier to modify in the future. This method is to be added to the PersonnelData class shown in Listing 7.3. A suitable tester application would be like that listing's EarliestWorkerTester's `main` except line 14 would become `String out = new PersonnelData().countBirthYears();`.

Listing 7.4 A PersonnelData method using an array of counters

```
public class PersonnelData   // continued
{
   private Buffin itsFile = new Buffin ("workers.txt");
   private static final int TIME_SPAN = 10;
   private static final int YEAR = 1960;


   /** Read a file and return the string representation of the
    *  number of Workers born in each of 1960 to 1969. */

   public String countBirthYears()
   {  int[ ] count = new int [TIME_SPAN];  // all zeros      //1

      //== READ THE WORKER DATA AND UPDATE THE COUNTS
      Worker data = new Worker (itsFile.readLine());         //2
      while (data.getName() != null)                         //3
      {  int lastDigit = data.getBirthYear() - YEAR;         //4
         count[lastDigit]++;  // error check left as exercise//5
         data = new Worker (itsFile.readLine());             //6
      }                                                      //7

      //== CONSTRUCT THE STRING OF ANSWERS
      String s = "";                                         //8
      for (int k = 0;  k < TIME_SPAN;  k++)                  //9
         s += (YEAR + k) + " : " + count[k] + " workers\n";  //10
      return s;                                              //11
   }  //======================
}
```

Programming Style Listing 7.4 declares `lastDigit` inside the loop where it is assigned instead of outside it. It is preferable to not declare a local variable any more broadly than it has to be. Some would say this wastes time redeclaring the variable each time through the loop. This assertion is faulty for two reasons: (a) a commercial-strength runtime system allocates all space for local variables when the method is called, not during execution of the method, so no time is actually wasted; (b) if that logic were cogent, you would be compelled to also "save time" when you code the Worker constructor by making all its local variables into class variables, since that constructor is called once each time through that same loop. But no one does that.

When you create an array, you must specify its length. For instance, `count = new int[10]` gives that new array named `count` a length of 10. The length cannot change later in the program. You get a **NegativeArraySizeException** if the specified length is negative. And you get an **ArrayIndexOutOfBoundsException** if you refer to `count[k]` when `k` is negative or greater than 9. ArrayIndexOutOfBoundsException is a subclass of **IndexOutOfBoundsException**. Another of its subclasses is **StringIndexOutOfBoundsException**, thrown e.g. by `s.charAt(-2)`.

You can retrieve the length of an array using length, a final instance variable for arrays. For instance, `count.length` has the value 10. Note that it has no parentheses, in contrast to `s.length()` for a String. The ten variables in the `count` array are called the array's components, e.g., `count[4]` is the **component** whose **index** is 4 and `count[6]` is the component whose index is 6. Java arrays are **zero-based**: The first component has index 0. Exercise 7.15 illustrates how to handle cases where it would be natural to have a non-zero starting index ('A' in that case).

Note: If you want to test-run the program in Listing 7.4 using input from the keyboard, you only have to make sure that the `workers.txt` file does not exist. Then the Buffin will automatically switch to keyboard input. When you want to stop entering data from the keyboard, you may use either CTRL/D or CTRL/Z (depending on the operating system you are using). That causes `readLine()` to return a null value.

Using an array of Strings

If you want to print the year in words, you will find it easiest to use an array of words for numbers. Here is one way to make the array:

```
   String [ ] unit = new String [10];
   unit[0] = "";
   unit[1] = "-one";
   unit[2] = "-two";
   unit[3] = "-three";
   unit[4] = "-four";
   unit[5] = "-five";
   unit[6] = "-six";
   unit[7] = "-seven";
   unit[8] = "-eight";
   unit[9] = "-nine";
```

Then the next-to-last statement of Listing 7.4 could be replaced by the following:

```
   s += "Nineteen sixty" + unit[k] + " : "
                         + count[k] + " workers.";
```

The output would therefore begin something like the following:

```
Nineteen sixty had 5 workers.
Nineteen sixty-one had 2 workers.
Nineteen sixty-two had 8 workers.
```

Initializer lists

An **initializer list** is a simpler way to define the values in the `unit` array. It can only be used at the time the array is declared. The following statement uses an initializer list to do the same for `unit` as what was just shown. The array is created with a length of 10 because 10 values are listed:

```
String [ ] unit = {"", "-one", "-two", "-three", "-four",
              "-five", "-six", "-seven", "-eight", "-nine"};
```

When you create a new array object, each component is initialized to zero (or null if an array of objects, or false if an array of booleans). Some people prefer to explicitly initialize the values in an array when it makes a difference what the values are. If you wanted to do that, you could just replace the declaration of `count` in Listing 7.4 by the following use of an initializer list:

```
int [ ] count = {0,0,0, 0,0,0, 0,0,0, 0};
```

Language elements

You may declare an array reference using:	Type [ ] VariableName = new Type [IntExpression]
or:	Type [ ] VariableName = { ExpressionList }
A component of an array is:	ArrayReference [IntExpression]
And the capacity of the array is:	ArrayReference . length

When an array is created, every component of that array is automatically initialized to zero or null or false (whichever is appropriate). By contrast, local variables are not automatically initialized.

Exercise 7.14 Improve Listing 7.4 to also guard against a worker with a birth year not in the 1960s by simply ignoring all such workers.
Exercise 7.15 (harder) Modify Listing 7.4 to count the number of workers whose name starts with `'A'`, the number for `'B'`, and so forth down to `'Z'`. Ignore workers whose name does not begin with a capital letter. Hint: `int index = something.charAt(0) - 'A'` could replace the assignment to `lastDigit`, since it is a number 0 through 25 if the string starts with a capital letter.
Exercise 7.16 (harder) Modify Listing 7.4 to count the number of workers born in every year 1920 through 1984, ignoring birth years outside that range. How many additional lines would you have in your program if you did not use an array, using `count20`, `count21`, ..., `count84` instead?
Exercise 7.17* Modify the preceding exercise to print the year all in words. Hint: Use the `unit` array plus a similar array for the tens digit.
Exercise 7.18* Improve Listing 7.4 to print one more number, namely, the number of workers whose birth years are not in the 1960s. Avoid a crash for such workers.
Exercise 7.19* Draw the UML class diagram for the CountBirthYears class.

7.4 Implementing The Worker Class With Arrays

Now that you know what kinds of messages you will be sending to Worker objects, you can decide what a Worker object needs to know in order to respond to the messages correctly. It needs to store its first name, last name, hourly pay rate, hours worked in each of the past five workdays (one workweek), birth year, address, etc. You decide that the first draft of the Worker class will have just the first five of those values to establish the basic idea. The junior programmers on your team can add the rest later.

When a worker is hired, the employer will know the name, the birth year, and the hourly wage rate at which the worker starts. The hours paid in a given week change from week to week and are not necessarily known at the time you need to construct a Worker object. It makes sense to just initialize the hours worked to a default value of eight hours in each workday.

Based on this, most of the methods for the Worker class are fairly obvious. Listing 7.5 (see next page) contains the coding for four of them. The last two methods in Listing 7.5, whose logic is discussed next, use the constants WEEK and NUM_DAYS.

The addHoursWorked and seeWeeksPay methods

A Worker is to keep track of only the five most recent workdays. When the `addHoursWorked` method adds to a worker's data the fact that he/she worked e.g. 9 hours today, that means you should move the data for the previous four days back one slot in the array to make room for it. You thereby lose the data for five days ago. That is, copy `itsHoursPaid[3]` into `itsHoursPaid[4]`, then copy `itsHoursPaid[2]` into `itsHoursPaid[3]`, etc., finally copying the parameter value `hoursWorked` into `itsHoursPaid[0]`.

The `seeWeeksPay` method needs to start by adding up the current five days' hours and multiplying that by `itsHourlyRate`. However, if the worker has worked more than forty hours in the past week, the client pays time-and-a-half for the overtime hours. For instance, if the worker has worked 50 hours in the past week, the worker is paid for 55 hours (5 bonus hours for the 10 hours of overtime). This is equivalent to paying the worker for 1.5 times the total number of hours worked, then subtracting off 20 hours because the first 40 hours should not be multiplied by 1.5.

Pictorial representation of a Worker object

Figure 7.3 shows how a typical Worker object can be represented. The Worker variable is an object reference, so its value is shown as an arrow pointing to a rectangle. That rectangle has no name on it, because objects themselves are not named variables. That rectangle contains five values, two of which are Strings. Since Strings and arrays are objects, their values are also shown as arrows pointing to rectangles.

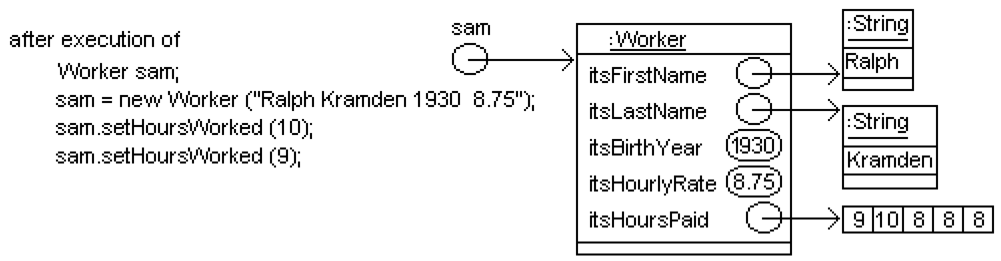
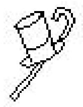


Figure 7.3 UML object diagram for a Worker object

Listing 7.5 The Worker class of objects, part 1

```
public class Worker  implements Comparable
{
   public static final double WEEK = 40.0; // regular work week
   public static final int NUM_DAYS = 5;   // days in 1 work week
   ////////////////////////////////////
   private String itsFirstName = null;
   private String itsLastName = null;
   private int itsBirthYear = 0;
   private double itsHourlyRate = 0;
   private double [] itsHoursPaid = {8, 8, 8, 8, 8}; // 1 per day


   /** Return the first name plus last name of the Worker.
    *  But return null if it does not represent a real Worker. */

   public String getName()
   {  return (itsLastName == null)  ?  null
                       :  itsFirstName + " " + itsLastName;
   }  //======================


   /** Return a String value containing most or all of the
    *  Worker's data, suitable for printing in a report. */

   public String toString()
   {  return itsLastName + ", " + itsFirstName + "; born "
             + itsBirthYear + ".  rate = " + itsHourlyRate;
   }  //======================


   /** Record the hours worked in the most recent day. */

   public void addHoursWorked (double hoursWorked)
   {  for (int k = NUM_DAYS - 1;  k > 0;  k--)
         itsHoursPaid[k] = itsHoursPaid[k - 1];
      itsHoursPaid[0] = hoursWorked;
   }  //======================


   /** Return the Worker's gross pay for the week. */

   public double seeWeeksPay()
   {  double sum = 0.0;
      for (int k = 0;  k < NUM_DAYS;  k++)
         sum += itsHoursPaid[k];
      return sum <= WEEK  ?  itsHourlyRate * sum
             :  itsHourlyRate * (sum * 1.5 - WEEK / 2);
   }  //======================
}
```



Programming Style It is good style to follow this Java naming convention: A method named getXXX obtains an attribute of an object without modifying that or any other object. The value is normally stored in an instance variable (as for `getName` and `getBirthYear`). Also, setXXX is conventionally a void method that modifies one or more attributes of the executor object to be equal to the parameter of setXXX, when the value of the parameter is reasonable.

The remaining Worker methods

The `equals` method for Workers tests whether they have the same first name and the same last name; work hours and other information does not count. An equals method should never throw an Exception. So the coding in the upper part of Listing 7.6 verifies that the parameter is not null and that the executor's last name is not null (which indicates that it does not represent an actual worker). Then it makes sure that the two names have the same characters in the same order.

Listing 7.6 The other methods in the Worker class of objects

```
//public class Worker, continued

   public boolean equals (Worker par)
   {  return par != null && this.itsLastName != null
                && this.itsLastName.equals (par.itsLastName)
                && this.itsFirstName.equals (par.itsFirstName);
   }  //======================


   public int getBirthYear()
   {  return itsBirthYear;
   }  //======================


   /** Compare two workers based on their last names,
    *  except based on their first names if the same last name.
    *  Precondition:  ob.getName() returns a non-null value. */

   public int compareTo (Object ob)
   {  Worker that = (Worker) ob;
      int comp = this.itsLastName.compareTo (that.itsLastName);
      return  comp != 0  ?  comp
              : this.itsFirstName.compareTo (that.itsFirstName);
   }  //======================


   /** Create a Worker from an input String, a single line with
    *  first name, last name, year, and rate in that order.
    *  If the String value is bad, the name is made null. */

   public Worker (String par)
   {  StringInfo si = new StringInfo (par);
      si.trimFront (0);
      if (si.toString().length() > 0)
      {  itsFirstName = si.firstWord();

         si.trimFront (itsFirstName.length());
         itsLastName = si.firstWord();

         si.trimFront (itsLastName.length());
         itsBirthYear = (int) si.parseDouble (-1);

         si.trimFront (si.firstWord().length());
         itsHourlyRate = si.parseDouble (-1);
      }
      if (itsBirthYear < 0 || itsHourlyRate < 0)
         itsLastName = null;  // in case of bad input
   }  //======================
```

Your client wants workers listed in order of last name. Workers with the same last name are to be listed in order of first name. So a reasonable logic for the Worker `compareTo` method is to apply String's `compareTo` method to `itsLastName` for the two workers and use that result. However, if the result of that String comparison is zero, meaning that they have the same last name, apply String's `compareTo` method to `itsFirstName` for the two workers and use that result.

The coding for the `compareTo` method is in the middle part of Listing 7.6. A `compareTo` method should always throw an Exception if `ob` is not Comparable with the executor. But `equals` should never throw an Exception. Moreover, when `x.equals(y)` is true, `x.compareTo(y)` should be zero. This is in accordance with the "general contract" for these two methods (i.e., standard practice for any object class's `compareTo and equals` methods).

<u>Reminder</u> Private instance variables are private to the <u>class</u>, not to the <u>object</u>. So the `compareTo` method can refer to `that`'s instance variables `itsFirstName` and `itsLastName` as well as `this`'s instance variables, because the `compareTo` method is inside the Worker class.

The constructor with a String parameter first makes sure the String value is acceptable. If it is null or contains nothing but whitespace, the constructor leaves the names with the null value as a signal to the caller of the method that there is no more valid input.

If input comes from the keyboard, the user indicates that all the Worker values have been entered by just pressing the ENTER key. This produces the empty String as a sentinel value (i.e., a value that stands at the end of the sequence of values to signal termination of the sequence). If input comes from a disk file, `readLine` returns null when it reaches the end of the file. Either way, this constructor leaves the names null.

If the string input has non-whitespace characters, it is split into four "words" (portions of non-whitespace characters separated from the other "words" by whitespace) using the StringInfo methods `trimFront` (remove leading blanks) and `firstWord` (the part down to the first blank) defined in Listing 6.4. The constructor uses the four "words" to fill in all instance variables except the work week (which defaults to 40 hours).

<u>Caution</u> No main method or other logic should call `compareTo` with null for the executor or for the parameter. If you have a dot after a variable with the value null, you are asking an object that does not exist to carry out an action or look at its instance variables. This does not make sense, and the runtime system tells you so -- it throws a NullPointerException.

Exercise 7.20 Explain why it does not work to have the heading of the for-statement in the `addHoursWorked` method be `for (int k = 1; k < NUM_DAYS; k++)`.
Exercise 7.21 Revise the `seeWeeksPay` method to pay time-and-a-half for any hours in excess of 8 in a single day, summed over all five days.
Exercise 7.22* Revise the Worker class for six-day workweeks, retaining the hours worked in the most recent six days.

7.5 Analysis And Design Example: Finding The Average In A Partially-filled Array

One of the programs that the client needs for the Personnel Database software is to list all workers who make more than fifty percent above the average. Discussion with the client clarifies the requirements: The workers are listed in the `workers.txt` file and the average is to be computed on the values that `seeWeeksPay()` returns.

You need to keep an array of all the Workers in RAM in order to solve this problem efficiently. The number of workers changes with hirings and firings; the client says it is currently almost 1600. So you decide the program will use an array large enough to store 5000 Worker objects. That should be more than enough for many years.

Design

The overall design is to first read all the workers into an array big enough to hold them, filling components 0, 1, 2, 3, etc., leaving the unneeded components at the end of the array. Then add up the week's pay for all workers to calculate the average pay (be careful not to divide by zero when the data file is empty). Finally, print out a description of each worker whose pay is greater than 1.5 times the average. If you name the array `item`, and you track the number of workers read in a variable named `size`, then the useable values are in the range `item[0]` through `item[size-1]` inclusive.

Implementation of the design

The key sublogic here is storing Worker objects into the array. The overall structure of the loop to read many Workers is similar to the loop in Listing 7.4, namely:

```
Worker data = new Worker (itsFile.readLine());
while (data.getName() != null)
{  // do whatever is appropriate
   data = new Worker (itsFile.readLine());
}
```

The appropriate thing to do here is assign data to the next available component of the array. When the loop begins, `size` is initially zero. We proceed as follows:
On the first iteration, we assign `item[0]=data`, so we increment `size` to 1.
On the second iteration, we assign `item[1]=data`, so we change `size` to be 2.
On the third iteration, we assign `item[2]=data`, which means we make `size` be 3.
In general, the appropriate thing to do on each iteration is as follows:

```
item[size] = data;
size++;
```

Although 5000 components should be far more than enough for a few years, we cannot be sure that the situation will not change drastically in a few months. We should refuse to accept any more Worker values if we do not have room for them. So the while-condition should be `(data.getName() != null && size < item.length)`.

Now that we have put values in the array, the logic is much the same as for the "full arrays" in Listing 7.5 (array length of NUM_DAYS) and Listing 7.4 (array length of TIME_SPAN) except that we use `size` in place of NUM_DAYS or TIME_SPAN. To add up the week's pay for all workers, we go through the components indexed from 0 to `size` (i.e., not including `size`), adding each one's `seeWeeksPay()` to a running total. To print out the highly-paid Workers, we go through the components from 0 to `size`, printing each one for which `seeWeeksPay()` is too large. The coding is in Listing 7.7.

Listing 7.7 A PersonnelData method using an array of Workers

```
public class PersonnelData   // continued
{
   private Buffin itsFile = new Buffin ("workers.txt");
   public static final int MAX_WORKERS = 5000;

   /** Read a file of up to 5000 Workers and display those who
    *  make more than fifty percent above the average pay. */

   public void printHighlyPaid()
   {
      //== INITIALIZE VARIABLES
      Worker[ ] item = new Worker[MAX_WORKERS];           //1
      int size = 0;  // number of workers in the array    //2

      //== READ THE WORKER DATA AND ADD EACH TO THE ARRAY
      Worker data = new Worker (itsFile.readLine());      //3
      while (data.getName() != null && size < item.length)   //4
      {  item[size] = data;                                //5
         size++;                                          //6
         data = new Worker (itsFile.readLine());          //7
      }                                                   //8

      //== CALCULATE THE AVERAGE WEEK'S PAY
      double totalPay = 0;                                //9
      for (int k = 0;  k < size;  k++)                    //10
         totalPay += item[k].seeWeeksPay();               //11
      double average = (size == 0) ? 0.0 : totalPay / size;  //12


      //== PRINT THOSE MAKING MORE THAN 50% OVER THE AVERAGE
      double highlyPaid = 1.5 * average;                  //13
      String s = "";                                      //14
      for (int k = 0;  k < size;  k++)                    //15
      {  if (item[k].seeWeeksPay() > highlyPaid)          //16
            s += item[k].toString() + "\n";               //17
      }                                                   //18
      JOptionPane.showMessageDialog (null, s);            //19
   }  //======================
}
```

A partially-filled array

The array in Listing 7.7 is a **partially-filled array** because the number of useable values changes each time you read the file. One conventionally keeps the useable data in components `0` through `size-1` of the array, as shown in Figure 7.4. Of course, the name of the variable that keeps count of the useable values does not have to be `size`.

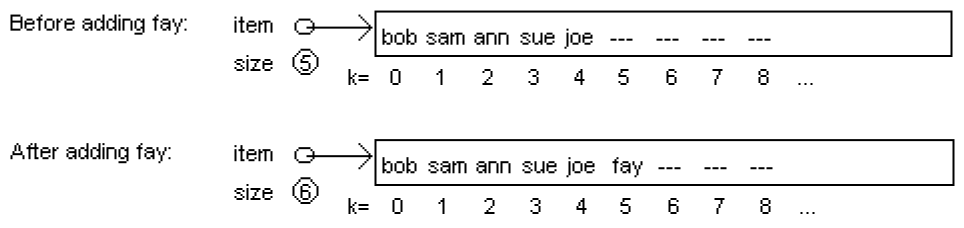


Figure 7.4 A partially-filled array and its size variable

Programming Style The only reason for the `highlyPaid` local variable in line 13 of Listing 7.7 is to avoid evaluating `1.5 * average` many times over inside the loop, each time getting the same result. In general, if your logic calls for evaluating the same expression three or more times, each time getting the same result, it is more efficient to evaluate it once and store it in a local variable for later use. In this particular situation, the logic shown "factors out" the expression, moving the calculation outside the loop.

When you complete a program, you should think it through one more time to make sure nothing could go wrong. You should realize for Listing 7.7 that the company may grow past 5000 workers. In that case, your program silently omits the last few. You need to add a statement somewhere, perhaps the following after the end of the while-loop:

```
if (data.getName() != null)
   JOptionPane.showMessageDialog (null, "Warning: This "
           + "program is skipping some workers.");
```

You will understand loops involving arrays much better if you learn how to describe the loops in ordinary English. When you have a loop with a heading such as `for(k = 0; k < size; k++)` process an array named `item`, you could refer to `item[k]` as "the current item". So the two for-loops in Listing 7.7 could be read as follows:

```
For each item in the array do...
   Add the current item's seeWeeksPay() to totalPay.

For each item in the array do...
   If the current item's seeWeeksPay() is larger than highlyPaid then...
      Print the String form of the current item.
```

Generalizing the Worker class

A university may someday come to your company for software that performs many of the same tasks as the Personnel Database software, but for Students rather than Workers. A doctor may ask for software that performs many of the same tasks, but for Patients rather than Workers. A lawyer may... but you get the idea.

The smart thing to do is to have a superclass from which all of Worker, Student, Patient, Client, etc. can inherit. You can then write much of the software to work with objects of this superclass rather than Workers. Then it will work with Worker objects, since Worker objects inherit all the public methods of the superclass. But it will also work with Patient or Student or Client objects.

You could name the superclass Person. A Person object should have a first name and last name on which the objects are ordered, and probably the year of birth, but not an hourly pay rate or other data specific to employees.

It would take very little extra work to use Person objects in place of Worker objects wherever possible for the Personnel Database software. You will not of course spend time on Students or Patients or Clients until the need arises. But when another customer asks for similar software, you should be able to reuse over half of your coding. This is called **modular programming** -- you create software for one purpose as a number of modules (methods and classes). Then when you want to use the software for a different purpose, you just pull out a few of the inappropriate modules and slot in others.

Exercise 7.23 Modify Listing 7.7 to print every Worker whose name comes alphabetically before that of the last Worker in the input file.
Exercise 7.24* Modify Listing 7.7 to print all Workers who are older than the average Worker.

7.6 Implementing The WorkerList Class With Arrays

A partially-filled array of Workers is a list of Workers that can be used and modified in many useful ways. The spirit of reusability calls upon us to put various tasks that a list of Workers can perform in separate methods in a suitable class called perhaps WorkerList. This allows us to easily use those methods in several different programs. When we finish, the `printHighlyPaid` method in Listing 7.7 could be written in just four statements as follows:

```
public void printHighlyPaid()
{  WorkerList job = new WorkerList (MAX_WORKERS);
   job.addFromFile (itsFile);
   double cutoff = 1.5 * job.getAveragePay();
   JOptionPane.showMessageDialog (null,
                    job.thosePaidOver (cutoff));
}  //======================
```

Implementing the WorkerList class

Listing 7.8 (see next page) shows a start on a WorkerList class. The constructor creates an array of the size specified by the parameter, except that it never makes an array of less than five components. This guards against the error of supplying a negative parameter, which would throw a NegativeArraySizeException. The array is named `itsItem` and the number of useable values in the array is `itsSize`. The logic for the other three methods in Listing 7.8 is virtually the same as in Listing 7.7 (compare them and see).

Figure 7.5 shows a representation of a WorkerList variable as containing an arrow pointing to the WorkerList object. That object's `itsItem` variable is itself an array object reference. So it is also shown as containing an arrow pointing to the object itself.

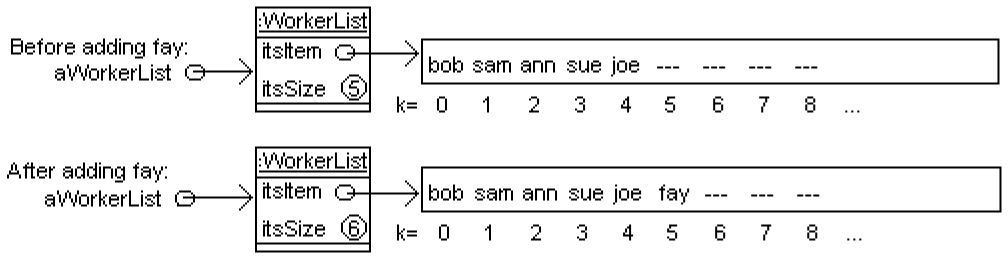


Figure 7.5 A WorkerList object before and after inserting at the end

The figure shows how the WorkerList object looks before and after adding a new Worker to the end of the list. For simplicity, the Worker values are not shown as arrows to yet more objects.

A "void method" (i.e., having no return value) can contain the statement **`return;`** to stop execution of the method early. Obviously, this return statement does not say what value is returned, since there is none. For instance, lines 3 and 4 of the `addFromFile` method in Listing 7.8 can be replaced by the following four equivalent lines:

```
while (data.getName() != null)
{  if (itsSize == itsItem.length)
      return;
   itsItem[itsSize] = data;
```

Listing 7.8 The WorkerList class of objects; some methods added later

```
public class WorkerList
{
   private Worker[ ] itsItem;
   private int itsSize = 0;


   /** Create an empty list capable of holding max Workers. */

   public WorkerList (int max)
   {  itsItem = (max > 5) ? new Worker[max] : new Worker[5]; //1
   }  //======================

   /** Add all Worker values in the non-null file to this list,
    *  except no more than there is room for in the list. */

   public void addFromFile (Buffin file)
   {  Worker data = new Worker (file.readLine());            //2
      while (data.getName() != null && itsSize < itsItem.length)
      {  itsItem[itsSize] = data;                            //4
         itsSize++;                                          //5
         data = new Worker (file.readLine());                //6
      }                                                      //7
   }  //======================

   /** Return the average pay for all workers in the list.
    *  Return zero if the list is empty. */

   public double getAveragePay()
   {  double totalPay = 0;                                   //8
      for (int k = 0;  k < itsSize;  k++)                    //9
         totalPay += itsItem[k].seeWeeksPay();               //10
      return itsSize == 0  ?  0.0  :  totalPay / itsSize;    //11
   }  //======================

   /** Return names of workers making more than the cutoff. */

   public String thosePaidOver (double cutoff)
   {  String s = "";                                         //12
      for (int k = 0;  k < itsSize;  k++)                    //13
      {  if (itsItem[k].seeWeeksPay() > cutoff)              //14
            s += itsItem[k].toString() + "\n";               //15
      }                                                      //16
      return s;                                              //17
   }  //======================
}
```

Postconditions and internal invariants

The **postcondition** for an action method is a statement of what has changed as a result of calling the method, assuming the precondition for that method has been met. For `printThosePaidOver`, the postcondition is that a list of people with a week's pay higher than the cutoff has appeared on the screen. For `addFromFile`, the postcondition is that each Worker value in the file, up to but not including the first one that would cause the WorkerList to have more than its capacity in Worker values, is stored at the end of the WorkerList in the sequence read. Neither of these two methods has a precondition other than the automatic precondition that the executor is non-null.

An **internal invariant** for a class of objects is a condition on the internal (private) structure of the instances of that class that is (a) true at the time each method is called and (b) true at the time that each method returns. An internal invariant thus describes a precondition of each method as well as a postcondition of each method. When you design a class of objects that stores a number of values, you often have an implicit internal invariant that you rely on in developing your coding. You should write it down so as to make it explicit, similar to the following:

<u>Internal invariant for WorkerLists</u>
1. `itsItem` is an array of Worker values and `0 <= itsSize <= itsItem.length`.
2. The values in components `itsItem[0]` through `itsItem[itsSize-1]` are the values on the conceptual list of Workers (the list that the user of the class thinks of when he/she uses it) and in the same order, with `itsItem[0]` being the first one. None of those Worker values are null.
3. The values stored in `itsItem[itsSize]` and higher do not affect anything.

An internal invariant is called "invariant" because it always remains true no matter what methods are called or in what order. It is called "internal" because the internal logic (the coding of the methods) keeps the precondition for each method true rather than relying on the (external) caller of the method to make the precondition true before the method is called. This particular internal invariant establishes the connection between the <u>abstract concept</u> of a number of values listed in a particular order and the <u>concrete</u> <u>realization</u> of that concept in coding.

The contains method

It is quite useful to have a method that searches the list to see if there is a match for a given value. As the executor goes through the list of Workers one by one, if it ever finds a match, it returns `true`. If the search comes to the end of the list without ever finding a match, the executor returns `false`.

You have seen this Some-A-are-B pattern several times in previous chapters, used to find out whether a certain condition is true for at least one of the values in a list or sequence. A reasonable coding is in the upper part of Listing 7.9. Note that the internal invariant justifies the use of the phrase `k = 0; k < itsSize`.

Listing 7.9 Two more WorkerList methods

```
/**  Tell whether par is a value in this WorkerList. */

public boolean contains (Worker par)    // in WorkerList
{  for (int k = 0;  k < itsSize;  k++)
   {  if (itsItem[k].equals (par))
         return true;
   }
   return false;
}  //======================


/**  Return a new object, a duplicate of this WorkerList. */

public WorkerList copy()                // in WorkerList
{  WorkerList valueToReturn;
   valueToReturn = new WorkerList (this.itsItem.length);
   valueToReturn.itsSize = this.itsSize;
   for (int k = 0;  k < this.itsSize;  k++)
      valueToReturn.itsItem[k] = this.itsItem[k];
   return valueToReturn;
}  //=======================
```

A WorkerList object as a return value

It is perfectly legal to have a method that returns a WorkerList object. For instance, you may at times want to make a second copy of a given WorkerList object. You first make a new WorkerList object with an array of the same size as the executor. Then you fill in the array components with the same values. This logic is in the lower part of Listing 7.9.

It should be clear that you cannot make a true copy of a WorkerList `sam` by declaring `WorkerList sue = sam`. That just makes `sam` and `sue` refer to one and the same WorkerList object. You have two WorkerList variables but only one WorkerList object. It is hopefully just as obvious that you cannot make a copy by executing `sue.itsItem = sam.itsItem`. You would have two WorkerList objects but only one array of Workers. Then any change you make in `sue's` workers perforce changes `sam`'s workers.

An array can be a method parameter. For instance, the filled-array-parameter analog of the `thosePaidOver` method in the earlier Listing 7.8 would be the following class method. Every other example and exercise in this section has a corresponding analog.

```
public static String thosePaidOver     // independent
           (Worker[] item, double cutoff)
{  String s = "";
   for (int k = 0;  k < item.length;  k++)
   {  if (item[k].seeWeeksPay() > cutoff)
         s += item[k].toString() + "\n";
   }
   return s;
}  //======================
```

In general, if `T` is any variable type, then `T[ ]` is also a variable type. A variable of such a type is called an **array variable**. If you declare `x` as a variable of type `T[ ]`, you may define `x = new T[whatever]` using a non-negative int expression inside the brackets. Thereafter, if `k` is an int expression in the range from 0 through `whatever-1`, then `x[k]` is a variable of type `T`.

Exercise 7.25 Write a method `public boolean everyonePaidLessThan (double cutoff)` in WorkerList: The executor tells whether all of its Workers have a week's pay that is less than a given double value (the parameter).
Exercise 7.26 (harder) Write a method `public String alphabeticallyFirst()` in WorkerList: The executor returns the name of the alphabetically first of its Workers. It returns null if there are no Workers. In case of a tie, return any of the tied ones.
Exercise 7.27 (harder) Write a method `public boolean inAscendingOrder()` in WorkerList: The executor tells whether all the names of its Workers are in increasing alphabetical order (each less than or equal to the next), as determined by `compareTo`.
Exercise 7.28 (harder) Write a method `public int countSame (WorkerList given)` in WorkerList: The executor tells how many Workers it has in common with the parameter (equal Workers in the same position). Throw an Exception if `given` is null.
Exercise 7.29* Write a method `public Worker oldestWorker()` in WorkerList: The executor returns the Worker object who is oldest (return null if no workers).
Exercise 7.30* Write a method `public boolean equals (WorkerList par)` in WorkerList: The executor tells whether it has the same Workers as the WorkerList parameter (using the `equals` method from the Worker class) in the same order.
Exercise 7.31* Write a method `public WorkerList highOnes()` in WorkerList: The executor returns a new WorkerList object containing only its Worker objects whose pay is above average.
Exercise 7.32* Write a method `public void addAll (WorkerList given)` in WorkerList: The executor adds all the Worker values from a WorkerList parameter to its own list, stopping only if its array becomes full. Throw an Exception if `given` is null.

Part B Enrichment And Reinforcement

7.7 A First Look At Sorting: The Insertion Sort

Consider this problem: You need a program that lists in order of birth year the Workers that are stored in a file. Specifically, you want the Workers to appear on the screen in **ascending order** of birth year (each less than or equal to the next), with ties printed in the same order that they were in the file. A reasonable design for this problem is in the accompanying design block.

Structured Natural Language Design for printOrderedByYear

```
1.   Create a WorkerList object.
2.   Connect to a file "workers.txt" containing strings that each represent one worker.
3.   Repeat until you reach the end of the file or the WorkerList is full...
         3a.  Read one Worker value.
         3b.  Insert it in the WorkerList before all Workers that have a later birth year.
4.   Print all the Workers in the order they occur in the WorkerList.
```

Once you have this main logic design, you need to see what additional capabilities a WorkerList object must have. A method to find its current size, a method to insert a new Worker in order of birth year, and a method to return a string equivalent of the WorkerList will all be useful. Listing 7.10 contains the fairly obvious coding for the given design.

Listing 7.10 A PersonnelData method using a WorkerList object

```
public class PersonnelData   // continued
{
   private Buffin itsFile = new Buffin ("workers.txt");
   public static final int MAX_WORKERS = 5000;


   /** Read a file of up to 5000 Workers and display them
    *  in ascending (increasing) order of birth years. */

   public void printOrderedByYear()
   {  WorkerList job = new WorkerList (MAX_WORKERS);
      Worker data = new Worker (itsFile.readLine());
      while (data.getName() != null  && job.size() < MAX_WORKERS)
      {  job.insertByYear (data);
         data = new Worker (itsFile.readLine());
      }
      System.out.println (job.toString());
   }  //======================
}  //########################################################


class OrderedPersonnelDataTester
{
   public static void main (String[ ] args)
   {  System.out.println ("Workers in order of birth year.");
      new PersonnelData().printOrderedByYear();
      System.exit (0);
   }  //======================
}
```

Coding the new WorkerList methods

The `size` method simply returns `itsSize`. The `toString` method can combine the `toString()` values of all the Workers together in the order they occur in the WorkerList, with an end-of-line marker between them. This can be useful for when you want to write the values in the WorkerList to a file, to be read in later by another program. These two methods are in the upper part of Listing 7.11.

Listing 7.11 Four more WorkerList methods

```
   /** Tell how many Workers are in this WorkerList. */

   public int size()                         // in WorkerList
   {  return itsSize;
   }  //======================


   /** Return a String value representing the entire WorkerList.
    *  It has one Worker's toString() value on each line. */

   public String toString()                  // in WorkerList
   {  String valueToReturn = "";
      for (int k = 0;  k < itsSize;  k++)
         valueToReturn += itsItem[k].toString() + "\n";
      return valueToReturn;
   }  //=======================


   /** Put the non-null data just before all Workers at the end
    *  of the WorkerList that have a larger birth year.
    *  Precondition:  The WorkerList has room for the data. */

   public void insertByYear (Worker data)   // in WorkerList
   {  int year = data.getBirthYear();
      int k = itsSize;
      for ( ; k > 0 && itsItem[k - 1].getBirthYear() > year; k--)
         itsItem[k] = itsItem[k - 1];
      itsItem[k] = data;
      itsSize++;
   }  //=======================


   /** Put all the Workers in ascending order of birth year.  */

   public void sort()                        // in WorkerList
   {  if (itsSize > 1)
      {  int save = itsSize;
         itsSize = 1;
         while (itsSize < save)
            insertByYear (itsItem[itsSize]);
      }
   }  //======================
```

The `insertByYear` method is more complicated. The obvious place to put a new piece of data is at index `itsSize`; assign `k = itsSize` to indicate that position is available for the new data. However, if the Worker value just below index `k` has a larger birth year than the new data to be added, that Worker value should be moved up to index `k`; then subtract 1 from `k` to indicate that the index one lower is now an available position for putting the new data.

Repeat this comparing and subtracting until you see a Worker that does not have a larger birth year or until you make index 0 available. Either way, put the new data value at that now-available index. This logic is in the middle part of Listing 7.11.

Listing 7.11 has one additional method made possible by the existence of the `insertByYear` method. If you have some non-empty WorkerList that is not in increasing order of birth year, and you want to make it so, just call the `sort` method for it. That `sort` method goes through each Worker value in the list, starting from the second one, and inserts it in increasing birth order among all the Worker values that come before it in the WorkerList. This logic is called the **insertion sort logic**. Chapter Thirteen discussed several other kinds of logic that sort a number of values in order.

The Sun standard library has a class of objects named **Collection**. It is much the same as WorkerList, except that it is on a higher level of abstraction. Therefore, it is more generally reusable than the WorkerList class. You will learn to develop the logic of the Collection methods in Chapter Fifteen. The exercises in this section introduce some of the Collection methods (`remove`, `add`, `containsAll`, `retainAll`) in the more concrete context of WorkerList objects. A precondition for all of the exercises here is that all object parameters are non-null.

Exercise 7.33 (harder) Write a method `public void insert (Worker given, int n)` in WorkerList: The executor inserts the given Worker at the given index `n`, making room by moving all the values at index `n` and above one component higher, so they remain in the original order. Take no action if `n` is out of range or the array is full.
Exercise 7.34 (harder) Write a method `public void remove (Worker given)` in WorkerList: The executor deletes from its list a single Worker object (if any) equal to a given Worker parameter. It moves the last Worker in the list in place of the one removed.
Exercise 7.35 (harder) Write a WorkerList method `public boolean add (Worker par)`: The executor adds `par` at the end of the list, if it is not equal to one already in the list and there is room. It returns true if it added the Worker and false if it did not.
Exercise 7.36* Write a method `public boolean containsAll (WorkerList given)` in WorkerList: The executor tells whether every Worker in the given parameter is also in the executor. Call on the `contains` method in Listing 7.9.
Exercise 7.37* Write a method `public void deleteYear (int given)` in WorkerList: The executor deletes every one of its Workers with the specified birth year. The order of those Workers that remain is not specified.
Exercise 7.38* Write a method `public void retainAll (WorkerList given)` in WorkerList: The executor deletes every one of its Workers that is not in the given parameter. The order of those Workers that remain is not specified.
Exercise 7.39** Write a method `public void reverse()` in WorkerList: The executor swaps its Worker values around so they are in the opposite order.
Exercise 7.40** Write a method `public boolean equivalent (WorkerList par)` in WorkerList: The executor tells whether it has exactly the same Workers as the WorkerList parameter (using the `equals` method from the Worker class) in some order, i.e., one list is a rearrangement of the values in the other. Hint: Create a copy of the executor, then write and repeatedly call a private method that tells whether a specified Worker value is in the copy and, if so, replaces it in the copy by null.
Exercise 7.41** Write a method that goes through an entire WorkerList, swapping any two adjacent Workers where the first has a larger birth year than the second. Repeat this process as many times as there are Workers in the list. Explain why this is guaranteed to put them in increasing order of birth year. Then explain why this **bubble sorting** logic works much more slowly than the insertion sorting logic in Listing 7.11.
Exercise 7.42** Revise your answer to the preceding exercise to stop repeating the swapping process as soon as one pass through the data does not cause any swaps. Discuss whether this improved logic is still much slower than the one in Listing 7.11.

7.8 A First Look At Two-Dimensional Arrays: Implementing The Network Classes

The new language features in this section are not used elsewhere in this book except in Chapter Twelve. They appear at this point only so that you can get a start on the concept of multi-dimensional arrays, which you can then carry further later.

Checkers as an example

A checkerboard for a computer game could be modeled by an 8-by-8 rectangular arrangement of values, some of which represent Checker pieces and some of which represent empty squares. You may declare an array with two indexes to represent the checkerboard, so that `board[0][0]` is the square in the lower-left corner, `board[0][1]` is the next square to its right, etc., up to `board[7][7]` for the upper-right corner. Figure 7.6 specifies the names all of the 64 variables to make this clear.

board[7][0]	board[7][1]	board[7][2]	board[7][3]	board[7][4]	board[7][5]	board[7][6]	board[7][7]
board[6][0]	board[6][1]	board[6][2]	board[6][3]	board[6][4]	board[6][5]	board[6][6]	board[6][7]
board[5][0]	board[5][1]	board[5][2]	board[5][3]	board[5][4]	board[5][5]	board[5][6]	board[5][7]
board[4][0]	board[4][1]	board[4][2]	board[4][3]	board[4][4]	board[4][5]	board[4][6]	board[4][7]
board[3][0]	board[3][1]	board[3][2]	board[3][3]	board[3][4]	board[3][5]	board[3][6]	board[3][7]
board[2][0]	board[2][1]	board[2][2]	board[2][3]	board[2][4]	board[2][5]	board[2][6]	board[2][7]
board[1][0]	board[1][1]	board[1][2]	board[1][3]	board[1][4]	board[1][5]	board[1][6]	board[1][7]
board[0][0]	board[0][1]	board[0][2]	board[0][3]	board[0][4]	board[0][5]	board[0][6]	board[0][7]

Figure 7.6 Checkerboard as an 8-by-8 array named board

If you use int values with named constants RED = 1, BLACK = 2, and EMPTY = 0, then you can declare the `board` variable as follows:

```
int[] [] board;  // variable is declared, array not created
```

Note that it has two pairs of brackets instead of just one, because you want to put two indexes on the `board` variable, not just one. You can initialize it as an array with 64 components in an 8-by-8 arrangements as follows:

```
board = new int [8] [8];  // creates array of 64 components
```

Just as with one-dimensional arrays of numbers, all components are initially zero. You can put four RED pieces on the board, in the bottom row at columns number 1, 3, 5, and 7, as follows:

```
for (int k = 1;  k < 8;  k += 2)
   board[0][k] = RED;
```

You can move a piece from square [3][4] to the northeast as follows:

```
board[4][5] = board[3][4];
board[3][4] = EMPTY;
```

At the end of the game, you can clear off all 64 squares on the board as follows:

```
for (int row = 0;  row < 8;  row++)
{  for (int col = 0;  col < 8;  col++)
      board[row][col] = EMPTY;
}
```

Two-dimensional arrays of objects

You can declare a variable that can refer to a rectangular array of components, each containing a Piece object, and then create the array with all entries null, as follows:

```
   Piece[] [] item = new Piece [NUM_ROWS] [NUM_COLS];
```

You can then refer to `item[x][y]` in the program as long as `x` has a value of 0 to NUM_ROWS and `y` has a value of 0 to NUM_COLS (including 0 but not including the upper limit). You can assign a value to such a component as follows:

```
   item[x][y] = somePieceValue;
```

And you can refer to the value at row number `x` and column number `y` in an expression, for instance:

```
   Piece p = item[x][y];
   if (item[x][y] == null)...
```

The array `item` is considered a one-dimensional array of variables of type `Piece[]`, so `item.length` is the number of rows, which is NUM_ROWS. One of those rows is `item[k]`, and the length of that row `item[k].length` is NUM_COLS.

Implementation of the Network classes

If you did not read at least some of Sections 5.5-5.7, skip the rest of this section; it discusses an implementation of the Network classes. For this implementation, the Network class stores the connection information in a two-dimensional array `itsNode`. The row of components whose first index is `k` is the set of Nodes that the kth Node connects to; excess components contain null. A reference to each Node in the Network is stored in the row of components whose first index is 0, followed by a component containing null.

When a Node is created, it is supplied three parameters: its name, whether it is blue, and the row of its connecting Nodes as a parameter. This row is itself a one-dimensional array. When that Node is then asked for a Position object to go through the list of its connections, it creates one with that row of connecting Nodes.

Position objects have two instance variables: one to remember the one-dimensional array that lists the Nodes it is to produce one at a time, and one to remember its current position in the sequence. This should be sufficient information that you could develop the Position and Node classes yourself; it is a major programming project. The Network class in Listing 7.12 (see next page) generates a random number of Nodes with random connections; you could replace that constructor by one that reads connection information from a disk file to handle real-world data.

For the randomizing constructor in Listing 7.12, if the number of Nodes in the Network is e.g. 12, the Network constructor creates a 13-by-13 array for storing the connection information. It creates 12 Nodes to go in the first row, named Node#1, Node#2, etc. Finally, it goes through each potential connection of Node `k` to Node `conn` and makes the connection 3/11 of the time, so each Node connects to 3 others on average.

Language elements
You may declare an array reference using the following, :
    Type [ ] [ ] VariableName = new Type [IntExpression] [IntExpression]
You may have more than two pairs of brackets, as long as you have the same number of brackets on the left of such a declaration as you have on the right.

Listing 7.12 The Network class of objects

```
public class Network
{
   /** Internal invariant:  The sequence of available Nodes is
    *  itsNode[0][k] for 0 <= k < itsNumNodes. The sequence of
    *  Nodes that itsNode[0][k] connects to is itsNode[k+1][j]
    *  for 0 <= j < n for some n <= itsNumNodes.  The null value
    *  comes after the last value of each such sequence. */

   private Node [][] itsNode;
   private int itsNumNodes;
   private java.util.Random randy = new java.util.Random();

   public Position nodes()
   {  return new Position (itsNode[0]);
   }  //=======================

   /** Create a random Network with 6 to 15 Nodes, about half
    *  blue, and with 3 connections per Node on average.  No Node
    *  connects to itself.  This is the only method you need to
    *  replace if you want to obtain Network data from a file. */

   public Network()
   {  itsNumNodes = 6 + randy.nextInt (10);
      itsNode = new Node [itsNumNodes + 1][itsNumNodes + 1];
      for (int k = 1;  k <= itsNumNodes;  k++)
         itsNode[0][k - 1] = new Node ("Node#" + k,
                     randy.nextInt(2) == 1, itsNode[k]);
      decideWhichConnectionsNodesHave();
   }  //=======================

   private void decideWhichConnectionsNodesHave()
   {  for (int k = 1;  k <= itsNumNodes;  k++)
      {  int j = 0;
         for (int conn = 1;  conn <= itsNumNodes;  conn++)
         {  if (randy.nextInt (itsNumNodes - 1) < 3 && conn != k)
            {  itsNode[k][j] = itsNode[0][conn - 1];
               j++;
            }
         }
      }
   }  //=======================
}
```

Exercise 7.43 Write statements to have the checker piece at `board[x][y]` jump the piece to its northwest, but only if the square beyond that piece exists and is empty. Do not forget to remove the piece you jump.
Exercise 7.44 (harder) Write statements to fill in the top three rows of the `board` with twelve BLACK checker pieces, four per row, alternating as in the initial position of a checkerboard. One piece goes in `board[7][0]`.
Exercise 7.45 (harder) Write an independent method for the checkerboard: `public static int numEmptySquares (int[][] board)` returns the number of empty squares. Throw an Exception if `board` is null.
Exercise 7.46* Write an independent method `public static int numNulls (Object[][] par)`: Return the number of null values in the given rectangular array, not counting components with either index being 0. Throw an Exception if `par` is null.
Exercise 7.47* How would you revise the Network constructor to have each Node k connect instead to the next three Nodes k+1, k+2, and k+3 (whenever they exist)?

7.9 Implementing A Vic Simulator With Arrays

You have now seen enough of Java that you can understand how the Vic class can be implemented using arrays. It will be quite similar to the simulator in Sections 5.4 and 5.5 using Strings (it is very helpful though not necessary to compare the development here with that development). The key difference is that the instance variable `itsSequence` is an array of Strings, each component representing one CD, declared as follows:

```
   private String[] itsSequence;
```

That allows you to store the full name of a CD if you wish, rather than faking it with a single letter. The other instance variables remain `itsPos` (a position in the sequence, numbering from 1 up for convenience) and `itsID` (a positive int by which the Vic object is identified). So `getPosition` has the one statement specified in Section 5.5:

```
   public String getPosition()            // in Vic
   {  return itsPos + "," + itsID;
   }  //======================
```

The `backUp` method is the same logic as earlier, except that an attempt to `backUp` when it is illegal calls a `fail` method to explain the problem and then terminate the program:

```
   public void backUp()                   // in Vic
   {  if (itsPos == 1)
         fail ("backUp from slot ");
      itsPos--;
      trace ("backUp to slot ");
   }  //======================

   private void fail (String message)     // in Vic
   {  System.out.println ("Vic# " + itsID + " crashed on "
                           + message + itsPos);
      System.exit (0);
   }  //======================
```

Working with an array of Strings

The `seesSlot` method checks that `itsSequence` has a slot at the current position, which requires that `itsPos` not be greater than the maximum allowed. Since `itsSequence` is an array rather than a String, we must compare with the final `length` variable of the array rather than calling the `length()` method of a String:

```
   public boolean seesSlot()              // in Vic
   {  return itsPos < itsSequence.length;
   }  //======================
```

The `seesCD` method should now be understandable. We have the class variable `NONE = "0"` with which we can compare the value in component `itsPos`:

```
   public boolean seesCD()                // in Vic
   {  if ( ! seesSlot())
         fail ("can't see a CD where there is no slot, at ");
      return ! itsSequence [itsPos].equals (NONE);
   }  //======================
```

The private `trace` method is a little more complex, because we have to build the String value for the sequence of CDs so we can print the current status. We can start with the first CD `itsSequence[1]` and add each additional CD string to that, with blanks separating them. The method can be coded as follows:

```
private void trace (String action)      // in Vic
{  String seq = itsSequence [1];
   for (int k = 2;  k < itsSequence.length;  k++)
      seq += " " + itsSequence [k];
   System.out.println ("Vic# " + itsID + ": " + action
              + itsPos + "; sequence= " + seq);
}  //======================
```

Implementing the stack

We have one stack in the entire Vic class, so `theStack` should be a class variable. The stack is also an array of String values, but unlike a sequence, its size is not fixed. That is, `theStack` is a partially-filled array but `itsSequence` is completely filled. So we need to track the current number of items `theStackSize` in `theStack`. We also need to make `theStack` large enough for any foreseeable demands. The following class variables should suffice, having the stack be empty to start with:

```
private static String[] theStack = new String[1000];
private static int theStackSize = 0;
```

For the `takeCD` method, you first have to make sure that a CD is at `itsPos`. If so, you copy it into `theStack` after all the other CDs currently in `theStack`, thereby increasing the number of items in the stack. Finally, you put `NONE` where the CD was in `itsSequence`:

```
public void takeCD()                    // in Vic
{  if (seesCD())
   {  theStack [theStackSize] = itsSequence [itsPos];
      theStackSize++;
      itsSequence [itsPos] = NONE;
   }
   trace ("takeCD at slot ");
}  //======================
```

The `say` and `reset` methods have exactly the same coding as in Section 5.4, using the same `theTableau` class variable to hold the array of String values that describes the various sequences and `theNumVics` class variable to tell how many Vic objects have been created so far. That leaves only the Vic constructor as a (hard) exercise.

Exercise 7.48 Write Vic's `stackHasCD` method in the context described in this section.
Exercise 7.49 Write Vic's `putCD` method in the context described in this section.
Exercise 7.50* Write Vic's `moveOn` method in the context described in this section.
Exercise 7.51** Write the Vic constructor in the context described in this section, using CD names such as "a1", "b1", "c1" for CDs in the first sequence, "a2", "b2", "c2" for CDs in the second sequence, etc.

7.10 Command-Line Arguments

The `(String[ ] args)` part of the heading of the main method allows it to receive information from the `java` command that started the program. Each word on the command line after `java ProgramX` is assigned to an array component, in the order it appears on the line: `args[0]`, `args[1]`, etc.

For instance, suppose that the statement `Debug.setTrace(true)` is to produce tracing printouts in your program. You could start your main method with this line:

```
   Debug.setTrace (args.length > 0 && args[0].equals ("trace"));
```

Now to run the program to produce tracing printouts, you enter `java ProgramX trace`. That will make `args` an array of length 1 with `args[0]` having the value `"trace"`. `trace` is the **command-line argument**. When you do not want tracing printouts, just enter the usual `java ProgramX`. That will make `args` an array of length zero. Note that the crash-guard before the && operator verifies the existence of the component of index zero before referring to it.

Using the command line to add numbers

If the command line has four words after the basic command, then `args.length` is 4, `args[0]` is the first word, `args[1]` is the second word, `args[2]` is the third word, and `args[3]` is the fourth word. Words are divided at blanks. Even numerals count as words. For instance, if you have the following Add class, you can enter something like `java Add 4.2 -2.16 13.7 5.1` and have it respond with the correct total of 20.839999999999996 (due to rounding off in base 2):

```
   public class Add
   {  public static void main (String[ ] args)
      {  double total = Double.parseDouble (args[0]);
         for (int k = 1;  k < args.length;  k++)
            total += Double.parseDouble (args[k]);
         System.out.println ("The total is " + total);
      }  //======================
   }
```

Note that this program is not robust; it throws an Exception if there are no command-line arguments or if any of them is not a numeral. The latter problem can be avoided by using StringInfo objects; the former problem is an exercise.

You could instead call this `main` method from a different class. For instance, executing the following two statements in another class prints "The total is 13.5":

```
   String[] values = {"4.25", "7.5", "1.75"};
   Add.main (values);
```

Exercise 7.52* Revise the main method for the Add class to give the answer zero when the user does not put anything after `java Add` (currently the program crashes).
Exercise 7.53* Revise the Ordering class in the earlier Listing 7.2 to let the user supply the file name in the command line. Only use `workers.txt` when the file name is not supplied.

*7.11 Implementing Queue As A Subclass Of ArrayList (*Enrichment)*

The Sun standard library offers a quite useful class of objects named ArrayList (it supercedes the Vector class that was in Java Version 1.0). Each instance of ArrayList represents a sequence of Object values, called the **elements** of the ArrayList, and is in many respects similar to a partially-filled array. The basic methods available for an **ArrayList** (from the `java.util` package) are the following:

- `new ArrayList()` creates an ArrayList object capable of holding however many objects you want to have. Initially it contain no objects at all (its size is zero).
- `someArrayList.size()` tells how many elements are stored in this ArrayList. It is analogous to `itsSize` for a partially-filled array `Object[] itsItem`.
- `someArrayList.get(indexInt)` returns the Object value stored at `indexInt`. This is analogous to using `itsItem[indexInt]` in an expression. ArrayLists are zero-based, i.e., the first value in the array is at index 0.
- `someArrayList.set(indexInt, someObject)` replaces the element at `indexInt` by `someObject`. So this is analogous to the assignment `itsItem[indexInt] = someObject`.

An example of coding using an ArrayList object named `al` is the following. It replaces each element that equals a given `target` value by null:

```
for (int k = 0;  k < al.size();  k++)
{  if (al.get (k).equals (target))
      al.set (k, null);
}
```

Changing the length of an ArrayList

If the preceding were coding involving an array, you would replace the three phrases involving `al` by `al.length`, `al[k].equals(target)`, and `al[k] = null`, respectively. So this ArrayList class would not be much use if it were not for the other methods it offers. Those methods have an ArrayList object grow and shrink in length, which of course is beyond the power of an ordinary array. For instance, `al.add(k,ob)` inserts `ob` into the list at index `k`, and `al.remove(3)` shrinks the list of elements by deleting the one at index 3 (i.e., the fourth element in the list). Listing 7.13 (see next page) describes the most useful ArrayList methods.

As an example, the following for-loop would remove every other data value in an ArrayList named `origin` and insert it in ascending order in an ArrayList named `other`, by calling on the `insertInOrder` method below it. It assumes that each data value is Comparable with the others, so the class cast `(Comparable)` can be used. You should compare this logic with the logic in the lower part of Listing 7.11:

```
for (int index = 0;  index < origin.size();  index++)
   insertInOrder (other, (Comparable) origin.remove (index));

public static void insertInOrder (ArrayList other,
                                  Comparable data)
{  int k = other.size();
   while (k > 0 && data.compareTo (other.get (k - 1)  < 0)
      k--;
   other.add (k, data);
}
```

Listing 7.13 Key methods in the ArrayList class, stubbed form

```
public class ArrayList    // stubbed documentation
{
   /** Create an empty list of elements. */
   public ArrayList()                                  { }

   /** Return the number of elements in this list. */
   public int size()                          { return 0;  }

   /** Return the element at this index.  Throw
    *  IndexOutOfBoundsException unless 0 <= index < size(). */
   public Object get (int index)            { return null; }

   /** Replace the element at this index by ob.  Throw
    *  IndexOutOfBoundsException unless 0 <= index < size(). */
   public void set (int index, Object ob)              { }

   /** Put ob as the last element in this list.  Return true. */
   public boolean add (Object ob)           { return true; }

   /** Delete the element at this index and return it.
    *  Shrink the list by 1 element. Throw
    *  IndexOutOfBoundsException unless 0 <= index < size(). */
   public Object remove (int index)         { return null; }

   /** Insert ob at this index.  Expand the list by 1.  Throw
    *  IndexOutOfBoundsException unless 0 <= index <= size(). */
   public void add (int index, Object ob)              { }
}
```

Implementing a Queue with an ArrayList

The RepairShop software in Chapter Six uses a Queue class that includes three operations:

- `aQueue.enqueue (ob)` adds the object value `ob` to `aQueue`, where `ob` can be any kind of object.
- `aQueue.dequeue()` returns the next available object on a first-in-first-out basis.
- `aQueue.isEmpty()` tells whether aQueue is empty.

The values stored in the Queue are of Object type. Since Object is a superclass of every class in Java, you can put any kind of object you like in a Queue. A Queue is a kind of object that you will see many uses for in later Computer Science courses. Its `dequeue` method removes the element that has been in the Queue for the longest period of time. For this reason, a Queue is known as a First-In-First-Out data structure (FIFO).

The standard specification for a Queue includes a `peekFront` method to allow people to see what they would get if they called the `dequeue` method, yet without modifying the Queue object. This method is not used by the RepairShop software, but it is standard to have it in the Queue class so that other applications that use Queues will have all they need.

Listing 7.14 has a straightforward implementation of the Queue class as a subclass of ArrayList. Note that `dequeue` and `peekFront` throw an IndexOutOfBoundsException when the Queue is empty, since that is what the corresponding ArrayList methods do.

Listing 7.14 The Queue class of objects

```
public class Queue extends java.util.ArrayList
{
   public Queue()
   {  super();  // just to remind you of the default constructor
   }  //======================

   /** Tell whether the queue has no more elements. */

   public boolean isEmpty()
   {  return size() == 0;
   }  //======================

   /** Remove the value that has been in the queue the longest
    *  time.  Throw an Exception if the queue is empty. */

   public Object dequeue()
   {  return remove (0);
   }  //======================

   /** Return what dequeue would give, without modifying
    *  the queue.  Throw an Exception if the queue is empty. */

   public Object peekFront()
   {  return get (0);
   }  //======================

   /** Add the given value to the queue. */

   public void enqueue (Object ob)
   {  add (ob);
   }  //======================
}
```

Exercise 7.54 Rewrite the methods in Listing 7.14 so that Queue does not extend the ArrayList class. Instead, it has an ArrayList instance variable. This is the use of composition rather than inheritance.
Exercise 7.55 Write an action method named `replaceNullsBy` for a subclass of ArrayList: The executor replaces each null value in it by a given Object parameter.
Exercise 7.56 (harder) Write a query method named `indexOf` for a subclass of ArrayList: The executor returns the index number of the earliest occurrence of an Object parameter, using the `equals` method to find it. It returns `-1` if the parameter is not found.

*7.12 More On System, String, And StringBuffer (*Sun Library)*

`System.out` is a PrintStream object that is already open and prepared to receive output data. It is normally the terminal window. Use `System.out.print (someString)` to print a String value without starting a new line immediately after it. For instance, if `System.out.print("a")` executes at one time and `System.out.println("b")` executes a while later, the output would be a single line containing `"ab"`. Thus the following statements print all the lowercase letters on one line of output:

```
for (int k = 'a';  k < 'z';  k++)
   System.out.print ((char) k);
System.out.println ('z');
```

The `print` and `println` methods are heavily overloaded to allow a boolean, char, int, double or other numeric value for the parameter, as well as any Object (for which the object's `toString` method is used to decide what appears). So if `sam` is a Worker object, then `System.out.println(sam)` actually prints `sam.toString()`.

Redirecting output to a disk file

You may redirect `System.out`'s output to a disk file. For instance,

```
java Whatever > results.txt
```

sends `System.out`'s output to the file named `results.txt` instead of to the terminal window. You can still send information to the terminal window using `System.err.println(whatever)` and `System.err.print(whatever)`; `System.err` is the standard error channel for reporting problems.

The System.arraycopy method

The statement `System.arraycopy (source, k, target, n, 200);` copies 200 values from the array of Objects named `source` to the array of Objects named `target`. `source[k]` is copied into `target[n]`, then `source[k + 1]` is copied into `target[n + 1]`, etc., until the required number of values have been copied. It throws an IndexOutOfBoundsException if data would be accessed outside of an array's bounds, and it throws an **ArrayStoreException** if it is illegal to make the assignments.

Additional String methods

The String class has many more methods than those discussed in Chapter Six (`equals`, `length`, `substring`, `compareTo`, and `charAt`). If `s` and `t` are two String values, then the following methods can be useful:

- `s.concat(t)` is the same as `s + t`.
- `s.equalsIgnoreCase(t)` is the same as `equals` except that a lowercase letter is considered equal to the corresponding capital letter. Thus `"abc".equalsIgnoreCase("AbC")` is `true`.
- `s.compareToIgnoreCase(t)` is the same as `compareTo` except that a lowercase letter is considered equal to the corresponding capital letter. Thus `"BOB".compareToIgnoreCase("abe")` is a positive int.
- `s.indexOf(t)` returns the index where the first copy of `t` begins in `s`; it returns -1 if no substring of `s` is equal to `t`. Thus `"aababa".indexOf("ba")` is 2.
- `s.indexOf(t, n)` is the same as the above except that it only finds those substrings at index `n` or higher. Thus `"aababa".indexOf("ba",3)` is 4.

- `s.endsWith(t)` tells whether `s.equals(someString + t)`. Thus `"apple".endsWith("ple")` is `true`.
- `s.startsWith(t)` tells whether `s.equals(t + someString)`. Thus `"cathy".startsWith("cat")` is `true`.

If `s` is a String value, then:

- `s.trim()` is the result of removing all whitespace from either end.
- `s.toLowerCase()` is the result of replacing each capital letter by the corresponding lowercase letter. Thus `"A1bC".toLowerCase()` is `"a1bc"`.
- `s.toUpperCase()` is the result of replacing each lowercase letter by the corresponding capital letter.
- `s.indexOf(someChar)` returns the first index at which the given character occurs in the string. If it is not in `s`, the method returns -1.
- `s.indexOf(someChar, someInt)` is the same as the above except that it returns -1 if no character at or after the given index equals the given char value.
- `s.replace(someChar, anotherChar)` returns the result of replacing every occurrence of the first parameter by the second parameter. Thus `"apple".replace('p', 'x')` returns "axxle".
- `s.toCharArray()` produces the corresponding array of characters with the same number of components as `s` has characters. If you store it in `char[] chArray` and make some modifications in it, then...
- `new String(chArray)` converts it back to the corresponding String object.

The four versions of `indexOf` have four corresponding versions of `lastIndexOf`, which returns the last index at which the char or substring occurs (or -1 if none).

StringBuffer objects (from java.lang)

You can create a StringBuffer object from a given String value when you want to make substantial changes in the characters of the string. A String object is immutable, so substantial changes require continually creating new String objects. With a StringBuffer object, you make the changes you want and then put the information back into a String object. The essential StringBuffer methods are the following:

- `new StringBuffer(someString)` makes a modifiable copy of the given String value.
- `someStringBuffer.length()` returns the number of characters in it (analogous to `itsSize`).
- `someStringBuffer.charAt(someInt)` returns the character at that index (zero-based as usual).
- `someStringBuffer.setCharAt(someInt, someChar)` puts the given character at the given index in place of whatever char value is currently there.
- `someStringBuffer.toString()` returns a copy of the information as a String object.

For instance, coding to reverse the order of the characters in a StringBuffer named `buf` could be as follows:

```
int len = buf.length();
for (int k = 0;  k < len / 2;  k++)
{  char saved = buf.charAt (k);
   buf.setCharAt (k, buf.charAt (len - 1 - k));
   buf.setCharAt (len - 1 - k, saved);
}
```

Additional methods that change the lengths of StringBuffers are as follows:

- `someStringBuffer.delete(startInt, endInt)` deletes characters at index `startInt` through `endInt-1` and also returns the result. It requires that `0 <= startInt <= endInt <= someStringBuffer.length()`.
- `someStringBuffer.insert(startInt, someString)` puts `someString` starting at position `startInt` (moving other characters up), and also returns the result, for `0 <= startInt <= someStringBuffer.length()`.
- `someStringBuffer.append(someString)` puts `someString` at the end of the executor. It also returns the result. This is overloaded to accept any numeric value.
- `someStringBuffer.replace(startInt, endInt, someString)` has the same effect as `someStringBuffer.delete(startInt, endInt).insert (startInt, someString)`.

String buffers are more efficient when you use string concatenation heavily. For instance, the WorkerList `toString` method in Listing 7.11 would be better as follows:

```
public String toString()                 // in WorkerList
{  StringBuffer buf = new StringBuffer ("");
   for (int k = 0;  k < itsSize;  k++)
      buf.append (itsItem[k].toString()).append ("\n");
   return buf.toString();
}  //======================
```

7.13 Review Of Chapter Seven

Listing 7.4, Listing 7.5, and Listing 7.8 illustrate almost all Java language features introduced in this chapter.

About the Java language:

- `Type[ ] t` declares `t` to be the name of an **array variable**, an object variable that can refer to an **array** of values of the specified Type.
- The only operations you can perform on an array variable are to put **brackets** `[ ]` after it or `.length` after it. The only operation you can perform on a non-array object variable is to put a dot after it, followed by a variable or method belonging to its class. If you do any of these things when the object or array variable is null, it throws a NullPointerException.
- `t = new Type[38]` creates the array itself, so its 38 **components** can be filled in with values. Any non-negative int value can be used instead of 38 for the size. The array components are numbered from 0 up through 37 if `t`'s length is 38. Any use of `t[0]` or `t[1]` or in general `t[k]` refers to a Type variable. Any non-negative int value less than the length of the array can be used in place of `k` for the **index**.
- The declaration `new someType[n]` where `n` is negative throws a **NegativeArraySizeException**. The use of `someArray[k]` when `k` is negative or not less than the length of the array throws an **ArrayIndexOutOfBoundsException**.
- `t.length` is the number of components the array `t` has available.
- `Type[ ] t = {...}` with a number of values listed inside those braces creates the array and initializes it to have the values listed. This is an **initializer list**.
- The **`return;`** statement (with no return value) exits a void method immediately.
- `Type[ ][ ] t` declares `t` to be the name of a **two-dimensional array**. Then `t = new Type[n][m]` declares t to have `n * m` components doubly-indexed, where the first index can be `0..(n-1)` and the second can be `0..(m-1)`. `t.length is n and t[k].length is m`.

Other vocabulary to remember:

- **Stubbed documentation** for a class is a list of the method headings with comments describing their functions and not much else. It can be compilable if you add bodies such as `{return 0;}`.
- A program should be **robust**, which means that it should handle unexpected or unusual situations in a reasonable manner without crashing.
- A **test set** is a set of input values used for one run of a program, together with the expected outputs. A **Test plan** is a large enough number of test sets that you can be confident that almost all of the bugs are out of the program when all tests go right.
- A **partially-filled array** has useable values only in part of the array, conventionally at `0...size-1` (though many programmers use a name different from `size`).
- For **modular programming**, you create software for one purpose as a number of moderately independent modules (methods and classes). Then when you want to use the software for a different purpose, you just pull out a few of the inappropriate modules and slot in others.
- The **postcondition** for an action method is a statement of what has changed as a result of calling the method, assuming the precondition for that method has been met. An **internal invariant** for a class of objects is a description of the state of all objects which the coding in the class maintains as true when each method is called and true when each method is exited. In other words, it is both precondition and postcondition. It describes the connection between the abstract concept for which the class of objects is a concrete realization in software.
- A **command-line argument** is a String value on the command line after `java ClassName`. These values are passed in to the `String[] args` parameter of the main method. `args.length` tells how many values are passed in. `args[0]` is the first String value, `args[1]` is the second String value, etc.

Answers to Selected Exercises

7.1
```
public static void averageDailyPay (Worker karl)
{	JOptionPane.showMessageDialog (null, "The worker's name is " + karl.getName()
			+ ", and the average daily pay is " + (karl.seeWeeksPay() / 5));
}
```
7.2 The word "hoursWorked" within the method refers to the value of time.
The word "this" within the method refers to the value of chris.

7.3
```
public static String youngest (Worker first, Worker second, Worker third)
{	Worker last =   first.getBirthYear() >= second.getBirthYear()  ?  first : second;
	if (last.getBirthYear() < third.getBirthYear())
		last = third;
	return last.toString();
}
```
7.4 Replace the last line of lastOne by:
JOptionPane.showMessageDialog (null, "The alphabetically last is " + last.toString());
Replace the first part of the heading of lastOne by:
public static void findLastOne
Replace the body of the if-statement in the main method by:
    CompOp.printLastOne (first, second, third);

7.7 Replace the less-than operator < by the greater-than operator > .
Also change "first" to "last" everywhere, to have it make sense.

7.8 Insert after "else {":   double total = answerSoFar.seeWeeksPay();
Insert before the innermost if-statement: total += data.seeWeeksPay();
Insert before the semicolon at the end of the second return statement:
    + "\nThe total amount paid is " + total;

7.9 Insert after "else {":   int century = answerSoFar.getBirthYear() / 100;
        String result = "the same century.";
Insert before the innermost if-statement:
if (data.getBirthYear() / 100 != century)
    result = "different centuries.";
Insert before the semicolon at the end of the second return statement:
    + "\nThe workers were born in " + result

7.14 Replace the statement count[lastDigit]++; by:

```
if (lastDigit >= 0 && lastDigit < TIME_SPAN)
        count[lastDigit]++;
```

7.15 Rename TIME_SPAN as NUM_LETTERS throughout, with a value of 26 instead of 10.
Replace lines 4 and 5 in the body of the while-statement by:

```
int index = data.getName().charAt (0) - 'A';
if (index >= 0 && index < NUM_LETTERS)
        count[index]++;
```

Replace the for-statement by:

```
for (int k = 0;  k < NUM_LETTERS;  k++)
        s += (char) ('A' + k) + " had " + count[k] + " workers";
```

7.16 Change the initial YEAR from 1960 to 1920 and the initial TIME_SPAN from 10 to 65.
Replace the statement count[lastDigit]++; by:

```
if (lastDigit >= 0 && lastDigit < TIME_SPAN)
        count[lastDigit]++;
```

If you were to do this program without arrays, then:
(a) The line that creates the array and initializes it to zero (line 1) would be replaced by 65 initializations to zero.
(b) The count[lastDigit]++ line (line 5) would have to be replaced by 130 lines such as:

```
        else if (lastDigit == 24)
                count24++;
```

(c) And the 3 lines that create the string would be replaced by 65 lines, so the net added is 255 lines.

7.20 Say the number in component 0 is 11. Then reversing the direction of the for-loop would copy the 11 into component 1, then copy the 11 into component 2, then copy the 11 into component 3, then again into component 4. You would lose three needed values.

7.21 Replace the for statement and the return statement by the following:

```
for (int k = 0;  k < NUM_DAYS;  k++)
        sum += (itsHoursPaid[k] <= 8)  ?  itsHoursPaid[k]  :  (itsHoursPaid[k] * 1.5 - 4);
return sum * itsHourlyRate;
```

7.23 Replace the statements beginning with "double totalPay" (lines 9-18) by the following:

```
if (size > 0)  // crash-guard
{       Worker lastWorker = item[size - 1];
        for (int k = 0;  k < size - 1;  k++)
        {       if (item[k].compareTo (lastWorker) < 0)
                        JOptionPane.showMessageDialog (null, item[k].toString());
        }
}
```

7.25

```
public boolean everyonePaidLessThan (double cutoff)
{       for (int k = 0;  k < itsSize;  k++)
        {       if (itsItem[k].seeWeeksPay() >= cutoff)
                        return false;
        }
        return true;  // including when there are no workers
}
```

7.26

```
public String alphabeticallyFirst()
{       if (itsSize == 0)
                return null;
        Worker early = itsItem[0];
        for (int k = 1;  k < itsSize;  k++)
        {       if (itsItem[k].compareTo (early) < 0)
                        early = itsItem[k];
        }
        return early.getName();
}
```

7.27

```
public boolean inAscendingOrder()
{       for (int k = 0;  k < itsSize - 1;  k++)  // note itsSize - 1
        {       if (itsItem[k].compareTo (itsItem[k + 1]) > 0)
                        return false;
        }
        return true;  // including when there are no workers
}
```

7.28

```
public int countSame (WorkerList given)
{       int upperLimit = this.itsSize > given.itsSize  ?  given.itsSize  :  this.itsSize;
        int count = 0;
        for (int k = 0;  k < upperLimit;  k++)
        {       if (this.itsItem[k].equals (given.itsItem[k]))    // we have found a match
                        count++;
        }
        return count;
}
```

```
7.33    public void insert (Worker given, int n)
        {       if (n < 0  ||  n > itsSize  ||  itsSize >= itsItem.length)
                        return;
                for (int k = itsSize;  k > n;  k--)
                        itsItem[k] = itsItem[k - 1];  // similar to addHoursWorked
                itsItem[n] = given;
                itsSize++;
        }
7.34    public void remove (Worker given)
        {       int k = 0;
                while (k < itsSize && ! itsItem[k].equals (given))
                        k++;
                if (k < itsSize)    // we have found a match
                {       itsItem[k] = itsItem[itsSize - 1];    // replace it by the top one
                        itsSize--;
                }
        }
7.35    public boolean add (Worker par)
        {       if (itsSize == itsItem.length)
                        return false;
                for (int k = 0;  k < itsSize;  k++)
                        if (itsItem[k].equals (par))
                                return false;
                itsItem[itsSize] = par;
                itsSize++;
                return true;
        }
7.43    if (x >= 2 && y <= 5 && board[x - 2][y + 2] == EMPTY)
        {       board[x - 2][y + 2] = board[x][y];
                board[x][y] = EMPTY;
                board[x - 1][y + 1] = EMPTY;
        }
7.44    for (int y = 0; y < 8; y += 2)
        {       board[5][y] = BLACK;
                board[6][y + 1] = BLACK;
                board[7][y] = BLACK;
        }
7.45    public static int numEmptySquares (int[][] board)
        {       int count = 0;
                for (int x = 0;  x < 8;  x++)
                        for (int y = 0;  y < 8;  y++)
                        {       if (board[x][y] == EMPTY)
                                        count++;
                        }
                return count;
        }
7.48    public static boolean stackHasCD()
        {       return theStackSize > 0;
        }
7.49    public void putCD()
        {       if (! seesCD() && theStackSize > 0)  // the second operand could be stackHasCD()
                {       itsSequence [itsPos] = theStack [theStackSize - 1];
                        theStackSize--;
                }
                trace ("putCD at slot ");
        }
7.54    public class Queue
        {       private ArrayList itsList = new ArrayList();
                //  all methods are the same as in Listing 7.13 except put "itsList." in front of each call
                //  of a method from the ArrayList class, e.g., return itsList.remove (0).
        }
7.55    public void replaceNullsBy (Object ob)
        {       for (int k = 0;  k < size();  k++)
                {       if (get (k) == null)
                                set (k, ob);
                }
        }
7.56    public int indexOf (Object ob)
        {       for (int k = 0;  k < size();  k++)
                {       if (get (k).equals (ob))
                                return k;
                }
                return -1;
        }
```