

Chapter 8

Data Structure: Arrays

8.1 Why We Need Arrays

8.2 Collecting Input Data in Arrays

8.3 Translation Tables

8.4 Internal Structure of One-Dimensional Arrays

8.5 Arrays of Objects

8.6 Case Study: Databases

8.6.1 Behaviors

8.6.2 Architecture

8.6.3 Specifications

8.6.4 Implementation

8.6.5 Forms of Records and Keys

8.7 Case Study: Playing Pieces for Card Games

8.8 Two-Dimensional Arrays

8.9 Internal Structure of Two-Dimensional Arrays

8.10 Case Study: Slide-Puzzle Game

8.11 Testing Programs with Arrays

8.12 Summary

8.13 Programming Projects

8.14 Beyond the Basics

Computer programs often manage many objects of the same type, e.g., a bank's accounting program must manage hundreds of customer accounts. It is inconvenient and usually impossible to declare distinctly named variables for each of the customer accounts; instead, one constructs a new form of object—a data structure—to collectively hold and name the customer accounts.

The most popular form of data structure is the array, and this chapter introduces standard uses of arrays. After studying this chapter, the reader should be able to

- *use arrays to model real-life collections like a library’s catalog, a company’s database of customer records, or the playing pieces of a game.*
- *understand when to use one-dimensional arrays to model sequences and when to use two-dimensional arrays to model grids.*

8.1 Why We Need Arrays

When a program manipulates many variables that contain “similar” forms of data, organizational problems quickly arise. Here is an example: In an ice-skating competition, each skater’s performance is judged by six judges, who assign fractional scores. The six scores must be collected and manipulated in various ways, e.g., printed from highest to lowest, averaged, multiplied by weighting factors, and so on.

Say that the six scores are saved in these variables,

```
double score0; double score1; double score2;
double score3; double score4; double score5;
```

and say that you must write a method that locates and prints the highest score of the six. How will you do this? Alas, the method you write almost certainly will use a sequence of conditional statements, like this:

```
double high_score = score0;
if ( score1 > high_score ) { high_score = score1; }
if ( score2 > high_score ) { high_score = score2; }
if ( score3 > high_score ) { high_score = score3; }
if ( score4 > high_score ) { high_score = score4; }
if ( score5 > high_score ) { high_score = score5; }
System.out.println(high_score);
```

This unpleasant approach becomes even more unpleasant if there are more even scores to compare or if a more difficult task, such as ordering the scores from highest to lowest, must be performed. Some other approach is required.

Can we employ a loop to examine each of `score0` through `score6`? But the six variables have distinct *names*, and a loop has no way of changing from name to name. Ideally, we wish to use “subscripts” or “indexes,” so that we may refer to `score0` as `score0`, `score1` as `score1`, and so on. The notation would be exploited by this for-loop,

```
double high_score = score0;
for ( int i = 1; i <= 5; i = i + 1 )
    { if ( scorei > high_score )
      { high_score = scorei; }
    }
System.out.println(high_score);
```

which would examine all six variables.

Java uses *array variables*, for indexing like this. An array variable names a collection of individual variables, each of which possesses the same data type. For example, we can declare and initialize an array variable that holds six doubles by stating the following:

```
double[] score = new double[6];
```

The name of this variable is `score`, and its declared data type is `double[]` (read this as “double array”). The data type indicates that `score` is the name of a collection of `doubles`, and the doubles named by the array variable are `score[0]`, `score[1]`, ..., `score[5]`.

The initialization statement’s right-hand side, `new double[6]`, constructs a new form of object that holds six *elements*, each of which is a double variable.

It is traditional to draw an array like this:

	0	1	2	3	4	5
score	0.0	0.0	0.0	0.0	0.0	0.0

The diagram shows that a newly constructed array that holds numbers starts with zeros in all the elements.

When we wish to refer to one of the elements in the array named `score`, we use an *index* (also known as *subscript*) to identify the element. The elements are indexed as `score[0]`, `score[1]`, and so on, up to `score[5]`. For example, we can print the number held in element 3 of `score` by writing,

```
System.out.println(score[3]);
```

Java requires that an array’s indexes *must be integers starting with zero*.

It is helpful to think of variable `score` as the name of a “hotel” that has six “rooms,” where the rooms are labelled 0 through 5. By stating `score[3]`, we specify the precise address of one of the hotel’s rooms, where we locate the room’s “occupant.”

Of course, each of the elements of an array is itself a variable that can be assigned. For example, say that the leading judge gives the score, 5.9, to a skater. We insert the number into the array with this assignment:

```
score[0] = 5.9;
```

If the skater’s next score is 4.4, then we might write,

```
score[1] = 4.4;
```

Here is a picture that shows what we have accomplished:

	0	1	2	3	4	5
score	5.9	4.4	0.0	0.0	0.0	0.0

We can insert values into all the array's elements in this fashion.

But most importantly, the *index contained within the square brackets may be a variable or even an integer-valued arithmetic expression*. For example,

```
int i = 3;
System.out.println(score[i]);
System.out.println(score[i + 1]);
```

locates and prints the doubles held in elements 3 and 4 of `score`. By using variables and expressions as indexes, we can write intelligent loops, such the one that locates and prints a skater's highest score:

```
double high_score = score[0];
for ( int i = 1; i <= 5; i = i + 1 )
    // invariant: high_score holds the highest score in the range,
    // score[0] ..upto.. score[i-1]
    { if ( score[i] > high_score )
        { high_score = score[i]; }
    }
System.out.println(high_score);
```

By changing the value of `i` at each iteration, the loop's body examines all the array's elements, solving the problem we faced at the beginning of this section.

As noted above, we can use integer-valued arithmetic expressions as indexes. For example, perhaps variable `i` remembers an array index; if we wish to exchange the number in `score[i]` with that of its predecessor element, we can write

```
int temp = score[i];
score[i - 1] = score[i];
score[i] = temp;
```

The phrase, `score[i - 1]`, refers to the element that immediately precedes element `score[i]`. For example, for the array pictured earlier and when `i` holds 2, the result of executing the above assignments produces this array:

	0	1	2	3	4	5
score	5.9	0.0	4.4	0.0	0.0	0.0

If variable `i` held 0 (or a negative number), then `score[i - 1]` would be a nonsensical reference, and the execution would halt with a run-time exception, called an `ArrayIndexOutOfBoundsException`.

The above example showed how an array might hold a set of numbers. But arrays can hold characters, booleans, strings, and indeed, *any form of object whatsoever*. For example, the words of a sentence might be stored into an array like this:

```
String[] word = new String[3];
word[0] = "Hello";
word[1] = "to";
word[2] = "you";
```

Array `word` is declared so that it keeps strings in its elements.

Second, if we have written a class, say, `class BankAccount`, then we can declare an array to hold objects constructed from the class:

```
BankAccount[] r = new BankAccount[10];
r[3] = new BankAccount();
BankAccount x = new BankAccount();
r[0] = x;
```

The previous sequence of statements constructs two `BankAccount` objects and assigns them to array `r`.

Because they can hold numbers, booleans, characters, and objects, arrays are heavily used in computer programming to model sets or collections. The collection of skating scores seen at the beginning of this section is one simple example, but there are many others:

- a bank's customer accounts
- a library's books
- playing pieces or players for an interactive game
- a table of logarithms or solutions to an algebraic equation
- indeed, any "data bank," where multiple objects are held for reference

The sections that follow show how to use arrays to model these and other examples.

Exercises

1. Say that we declare this array:

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

What do each of these loops print? (Hint: It will be helpful to draw a picture of array `r`, like the ones seen in this section, and update the picture while you trace the execution of each loop.)

- (a)

```
for ( int i = 0; i < 4; i = i + 1 )
    { System.out.println(r[i]); }
```
- (b)

```
int i = 1;
r[i] = 10;
r[i + 2] = r[i] + 2;
for ( int i = 0; i < 4; i = i + 1 )
    { System.out.println(r[i]); }
```
- (c)

```
for ( int i = 3; i >= 0; i = i - 1 )
    { r[i] = i * 2; }
for ( int i = 0; i < 4; i = i + 1 )
    { System.out.println(r[i]); }
```
- (d)

```
r[0] = 10;
for ( int i = 1; i != 4; i = i + 1 )
    { r[i] = r[i - 1] * 2; }
for ( int i = 0; i < 4; i = i + 1 )
    { System.out.println(r[i]); }
```

2. Declare an array, `powers_of_two`, that holds 10 integers; write a for-loop that places into `powers_of_two[i]` the value of 2^i , for all values of i in the range, 0 to 9.
3. Declare an array, `letter`, that holds 26 characters. Write a for-loop that initializes the array with the characters, 'a' through 'z'. Next, write a loop that reads the contents of `letter` and prints it, that is, the letters in the alphabet, all on one line, in reverse order.
4. Declare an array, `reciprocals`, that holds 10 doubles; write a for-loop that places into `reciprocals[i]` the value of $1.0 / i$, for all values of i in the range, 1 to 9. (What value remains in `reciprocals[0]`?)

8.2 Collecting Input Data in Arrays

Arrays are particularly useful for collecting input data that arrive in random order. A good example is vote counting: Perhaps you must write a program that tallies the votes of a four-candidate election. (For simplicity, we will say that the candidates' "names" are Candidate 0, Candidate 1, Candidate 2, and Candidate 3.) Votes arrive one at a time, where a vote for Candidate i is denoted by the number, i . For example, two votes for Candidate 3 followed by one vote for Candidate 0 would appear:

```
3
3
0
```

and so on.

Vote counting will go smoothly with an array that holds the tallies for the four candidates: We construct an array whose elements are integers,

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

where `votes[0]` holds Candidate 0's votes, and so on. When a vote arrives, it must be added to the appropriate element:

```
int v = ...read the next vote from the input...
votes[v] = votes[v] + 1;
```

The algorithm for vote counting follows the “input processing pattern” from Chapter 7:

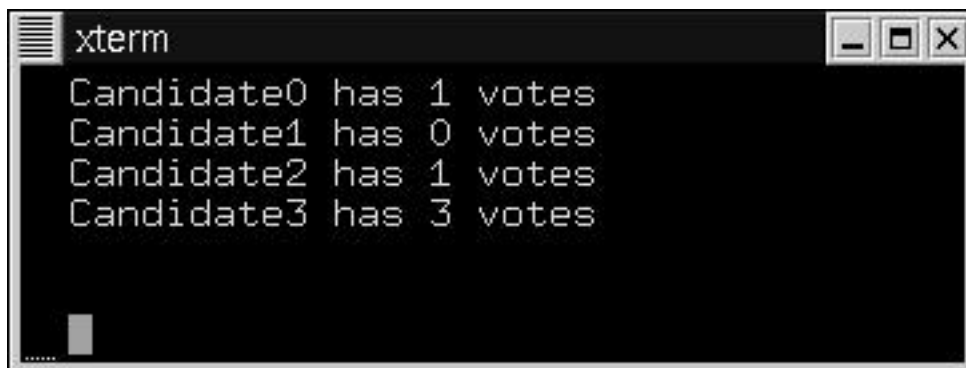
```
boolean processing = true;
while ( processing )
    { int v = ...read the next vote from the input...
      if ( v is a legal vote, that is, in the range, 0..3 )
          { votes[v] = votes[v] + 1; }
      else { processing = false; }
    }
```

Once all the votes are tallied, they must be printed. There is a standard way of printing the contents of an array like `votes`:

```
for ( int i = 0; i != votes.length; i = i + 1 )
    // invariant: values of votes[0]..votes[i-1] have been printed
    { System.out.println( ... votes[i] ... ); }
```

A for-loop works with a loop-counter variable, `i`, to print all the array's elements. Notice the phrase, `votes.length`, which is new and appears in the loop's termination test: The phrase is a built-in Java convenience that denotes the length of array `votes` (here, 4). *It is always better to use `votes.length`*, rather than 4, in the coding, because the former need not be changed if array `votes` is changed in a later version of the program.

Figure 1 presents the resulting vote counting program. For example, if the election's votes arrived in the order, 3, 3, 0, 2, 3, followed by a terminating number, e.g., -1, the program prints



```
xterm
Candidate0 has 1 votes
Candidate1 has 0 votes
Candidate2 has 1 votes
Candidate3 has 3 votes
```

Figure 8.1: vote counting

```

import javax.swing.*;
/** VoteCount tallies the votes for election candidates.
 * input: a sequence of votes, terminated by a -1
 * output: the listing of the candidates and their tallied votes */
public class VoteCount
{ public static void main(String[] args)
  { int num_candidates = 4;           // how many candidates
    int[] votes = new int[num_candidates]; // holds the votes;
                                     // recall that each element is initialized to 0
    // collect the votes:
    boolean processing = true;
    while ( processing )
      // invariant: all votes read have been tallied in array votes
      { int v = new Integer(JOptionPane.showInputDialog
                           ("Vote for (0,1,2,3):")).intValue();
        if ( v >= 0 && v < votes.length ) // is it a legal vote?
          { votes[v] = votes[v] + 1; }
        else { processing = false; } // quit if there is an illegal vote
      }
    // print the totals:
    for ( int i = 0; i != votes.length; i = i + 1 )
      // totals for votes[0]..votes[i-1] have been printed
      { System.out.println("Candidate" + i + " has " + votes[i] + " votes"); }
  }
}

```

Perhaps the key statement in the program is the conditional statement,

```

if ( v >= 0 && v < votes.length )
  { votes[v] = votes[v] + 1; }
else { ... }

```

The conditional adds the vote to the array *only if the vote falls within the range 0..3*. If an improperly valued v , say, 7, was used with `votes[v] = votes[v] + 1`, then execution would halt with this exception,

```

java.lang.ArrayIndexOutOfBoundsException: 7
    at Test.main(VoteCount.java:...)

```

because there is no element, `votes[7]`. It is always best to include conditional statements that help a program defend itself against possible invalid array indexes.

Exercises

1. Modify `class VoteCount` in Figure 1 so that it prints the total number of votes cast and the winning candidate's "name."
2. Modify `class VoteCount` so that the application first asks for the number of candidates in the election. After the number is typed, then votes are cast as usual and the results are printed.
3. Modify `class VoteCount` so that the application first requests the names of the candidates. After the names are typed, the votes are cast as usual and the results are printed with each candidate's name and votes. (Hint: Use an array of type `String[]` to hold the names.)
4. Write an application that reads a series of integers in the range 1 to 20. The input is terminated by an integer that does not fall in this range. For its output, the application prints the integer(s) that appeared most often in the input, the integer(s) that appeared least often, and the average of all the inputs.

8.3 Translation Tables

Arrays are useful for representing tables of information that must be frequently consulted. Here is an example: A simple form of coding is a *substitution code*, which systematically substitutes individual letters by integer codes. For example, if we replace every blank space by 0, every 'a' by 1, every 'b' by 2, and so on, we are using a substitution code. A sentence like,

a bed is read

is encoded into this sequence of integers:

1 0 2 5 4 0 9 19 0 18 5 1 4

This simple substitution code is all too easy for outsiders to decipher.

Here is a slightly more challenging substitution code: Starting with a "seed" integer, `k`, we encode a blank space by `k`. Call this value `code(' ')`. Next, we encode the alphabet in this pattern, where each character's code is the twice as large as its predecessor's, plus one:

```
code(' ') = k;
code('a') = (code(' ') * 2) + 1
code('b') = (code('a') * 2) + 1
...
code('z') = (code('y') * 2) + 1
```

For example, with a starting seed of 7, `a bed is read` encodes to

```
15 7 31 255 127 7 4095 4194303 7 2097151 255 15 127
```

The encoding program will work most efficiently if it first calculates the integer codes for all the letters and saves them in a translation table—an array. Then, the letters in the input words are quickly encoded by consulting the table for the codes.

We build the table as follows. First we declare the array:

```
int[] code = new int[27]; // this is the translation table:
                        // code[0] holds the code for ' ',
                        // code[1] holds the code for 'a',
                        // code[2] holds the code for 'b', and so on
```

Next, we systematically compute the codes to store in array `code`: the value of `code[i]` is defined in terms of its predecessor, `code[i - 1]`, when `i` is positive:

```
code[i] = (code[i - 1] * 2) + 1;
```

The arithmetic expression, `i - 1`, can be used, because Java allows integer-valued expressions as indexes.

We now write this loop to compute the codes:

```
int seed = ... ;
code[0] = seed;
for ( int i = 1; i != code.length; i = i + 1 )
    { code[i] = (code[i - 1] * 2) + 1; }
```

We are now ready to read a string and translate its characters one by one into integer codes: Java treats characters like they are integers, which makes it easy to check if a character, `c`, is a lower-case letter (`c >= 'a' && c <= 'z'` does this) and to convert the character into the correct index for array `code` (`(c - 'a') + 1` does this):

```
String input_line = JOptionPane.showInputDialog("type sentence to encode: ");
for ( int j = 0; j != input_line.length(); j = j + 1 )
    { char c = input_line.charAt(j);
      if ( c == ' ' )
          { System.out.println(code[0]); }
      else if ( c >= 'a' && c <= 'z' )
          { int index = (c - 'a') + 1;
            System.out.println(code[index]);
          }
      else { System.out.println("error: bad input character"); }
    }
```

(Recall that `S.length()` returns the length of string `S` and that `S.charAt(j)` extracts the `j`th character from `S`.)

When we build a translation table, we compute each possible translation exactly once, and we save and reuse the answers when we read the inputs. For this reason, translation tables are most useful when it is expensive to compute translations and there are many inputs to translate.

Exercises

1. Write the complete application which takes a seed integer and a line of words as input and produces a series of integer codes as output.
2. Write the corresponding decoder program, which takes the seed integer as its first input and then reads a sequence of integers, which are decoded into characters and printed.
3. Write statements that construct a translation table, `powers_of_two`, and assign to the table the powers of two, namely,

```
powers_of_two[i] = 2^i
```

for all values of `i` in the range, 0 to 9.

4. Translation tables have a special connection to recursively defined equations, like the ones we saw in Chapter 7. For example, this recursive definition of summation:

```
summation(0) = 0
summation(n) = n + summation(n-1), if n > 0
```

“defines” the following translation table:

```
int[] summation = new int[...];
summation[0] = 0;
for ( int n = 1; n != summation.length; n = n + 1 )
    { summation[n] = n + summation[n-1]; }
```

Write applications that build tables (arrays of 20 elements) for the following recursive definitions; make the applications print the contents of each table, in reverse order.

- (a) The factorial function:

```
0! = 1
n! = n * (n-1)!, when n is positive
```

- (b) The Fibonacci function:

```
Fib(0) = 1
Fib(1) = 1
Fib(n) = Fib(n-1) + Fib(n-2), when n >= 2
```

This example is especially interesting, because it is far more efficient to compute the entire translation table for a range of Fibonacci values and consult the table just once, than it is to compute a single Fibonacci value with a recursively defined method. Why is this so?

8.4 Internal Structure of One-Dimensional Arrays

Now that we have some experience with arrays, we should learn about their internal structure. The story starts innocently enough: A Java array variable is declared like any Java variable, e.g.,

```
int[] r;
```

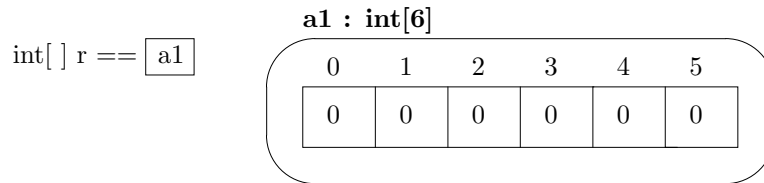
This declares variable `r` with data type `int[]` (“int array”). But variable `r` is meant to hold the *address* of an *array object*; it is not itself the array. We use assignment to place an address in `r`’s cell:

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

As noted earlier, the phrase, `new int[6]`, constructs an array object of six integer elements. We can do the two previous two steps in a single initialization statement:

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

As a result of the initialization to `r`, computer storage looks like this:

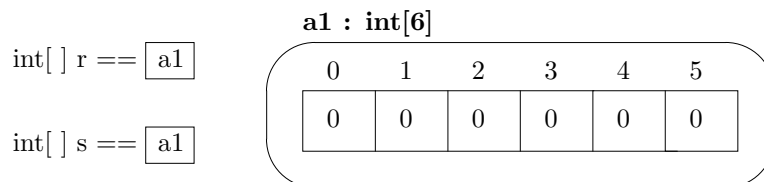


The address, `a1`, of the array object, `new int[6]`, is saved in variable `r`. Notice that the run-time data type of the object at address `a1` is `int[6]`—the data type includes the array’s length and its elements’ data type.

Many programmers fail to understand the difference between an array variable and an array object and try to duplicate an array like this:

```
int[] s = r;
```

This statement duplicates the *address* of the array object and not the object itself! When we examine computer storage, we see that variable `s` holds the same address held by `r`—the two variables share the same array object:



This means assignments to `s`’s elements alter `r`’s elements as well: For example, `s[0] = 3` will make `r[0] == 3` as well.

Once constructed, an array object's length cannot change. The length of an array, like `r`, can be obtained by the phrase, `r.length`. Note that `r.length` lacks a parentheses set, `()`. For whatever reason, the Java designers made `length` a "public field" of an array object, rather than a "public method."

The array named `r` is called *one dimensional* because one index (subscript) is required to identify a specific element in the array. Some programming languages permit construction of a two-dimensional array (also known as a *matrix* or *grid*), where two indexes are required to identify an element. Two dimensional arrays are studied later in this Chapter.

We can construct arrays of integers, doubles, booleans, strings, and indeed, of any legal data type. Here are some examples:

- `double[] score = new double[6]` constructs an array of six elements, each of which holds doubles. Each element is initialized to 0.0.
- `boolean[] safety_check = new boolean[i + 2]` constructs an array whose length is the value of integer variable `i` plus 2. The example shows that an integer-valued arithmetic expression states the length of the array object. (The value must be nonnegative.) Each element of a boolean array is initialized to `false`
- `BankAccount[] account = new BankAccount[100]` constructs an array that can hold 100 distinct `BankAccount` objects. (See Figure 11, Chapter 6, for `class BankAccount`.) *Each element is initialized to `null`, which means, "no value."*

The last statement of the last example is crucial to understanding Java arrays: When you construct an array whose elements hold objects, the array's elements are initialized to `null` values—*there is only a "container" but no objects in it*. Therefore, you must explicitly construct new objects and assign them to the elements. For the last example, we might write a loop that constructs and inserts objects into array `account`:

```
BankAccount[] account = new BankAccount[100];
for ( int i = 0; i != account.length; i = i + 1 )
    { account[i] = new BankAccount( ... ); } // see Figure 11, Chapter 6
```

An array can be partially filled with objects, just like an hotel can have some occupied rooms and some vacancies. We can test if an element is occupied by comparing the element's value to `null`. For example, here is a loop that prints the balances of all accounts in array `account`, skipping over those elements that are empty:

```
for ( int i = 0; i != account.length; i = i + 1 )
    { if ( account[i] != null )
        { System.out.println( "Balance of account " + i
                               + " is " + account[i].balanceOf() );
        }
    }
```

As noted in the previous section, integer-valued arithmetic expressions are used to index an array's elements. For example, these statements make short work of building a lookup table of powers of two:

```
int[] r = new int[6];
r[0] = 1;
for ( int i = 1; i < r.length; i = i + 1 )
    { r[i] = r[i - 1] * 2; }
```

Now, `r[j]` holds the value of 2^j :

0	1	2	3	4	5
1	2	4	8	16	32

A numeric or boolean array can be constructed and initialized with a set-like notation, which looks like this:

```
int[] r = {1, 2, 4, 8, 16, 32};
```

Because they are objects, array objects can be parameters to methods and can be results from methods. Here is an example: Method `reverse` accepts (the address of) an array object as its argument and returns as its result a newly constructed array object whose elements are arranged in reverse order of those in its argument:

```
public double[] reverse(double[] r)
{ double[] answer = new double[r.length];
  for ( int i = 0; i != r.length; i = i+1 )
    { answer[(r.length - 1) - i] = r[i]; }
  return answer;
}
```

When this method is invoked, for example,

```
double[] numbers d = {2.3, -4.6, 8, 3.14};
double[] e = reverse(d);
```

it is the *address* of the four-element array named by `d` that is bound to the formal parameter, `r`. Inside the method, a second array is constructed, and the numbers held in the array named `d` are copied into the new array object. When the method finishes, it returns the address of the second array—variables `d` and `e` hold the addresses of distinct array objects.

The `main` method that comes with every Java application uses an array parameter, `args`:

```
public static void main(String[] args)
{ ... args[0] ... args[1] ... }
```

When an application is started, whatever program arguments supplied in the application's start-up command are packaged into an array object, and the address of the array is bound to `args`. This explains why the leading program argument is referred to as `args[0]`, the next argument is `args[1]`, and so on. The number of program arguments supplied is of course `args.length`.

Exercises

Create the following arrays and assign to their elements as directed.

1. An array, `r`, of 15 integers, such that `r[0] = 0`, and the value of all the other `r[i]`s, `i` in `1..14`, is the summation of `i`. (Hint: Use the algorithm underlying Figure 1, Chapter 7, to calculate summations.)
2. An array, `d`, of 30 doubles, such that value of each `d[i]` is the square root of `i`. (Hint: Use `Math.sqrt`.) For `i` ranging from 0 to 29, print `i` and its square root.
3. An array, `b`, of 4 booleans, such that the value of `b[0]` is true, the value of `b[1]` is the negation of `b[0]`, and the value of `b[2]` is the conjunction of the previous two elements. (The value of `b[3]` does not matter.) Print the values of `b[3]` through `b[1]`.
4. Write this method:

```
/** maxElement returns the largest integer in its array parameter.
 * @param r - an array of 1 or more integers
 * @return the largest integer in the array */
public int maxElement(int[] r)
```

5. Write this method:

```
/** add adds the elements of two arrays, element-wise
 * @param r1 - an array
 * @param r2 - an array. Note: the two arrays' lengths must be the same
 * @return a newly constructed array, s, such that s[i] = r1[i] + r2[i],
 * for all i; return null, if r1 and r2 have different lengths. */
public double[] add (double[] r1, double[] r2)
```

6. Given this declaration,

```
BankAccount[] bank = new BankAccount[100];
```

- (a) Write a for-loop that creates one hundred distinct bank accounts, each with starting balance of 0, and assigns the accounts to the elements of `bank`.
- (b) Write a statement that adds 50 to the account at element 12; write statements that transfer all the money in the account at element 12 to the account at element 45.
- (c) Write a for-loop that prints the index numbers and balances for all accounts that have nonzero balances.
- (d) Write an assignment statement that makes the account at element 12 vanish.
- (e) Explain what happens when this assignment is executed: `bank[15] = bank[10];`

8.5 Arrays of Objects

The previous section pointed out that an array can hold objects. Arrays of objects can collect together bank accounts or library books or tax records into a single “data base.” Here is a small example.

Recall once more, `class BankAccount`, from Figure 11, Chapter 6. When we construct a `new BankAccount(N)`, we construct an object with an initial balance of `N` that can receive deposits and withdrawals. For example,

```
BankAccount x = new BankAccount(70);
... x.withdraw(50) ...
```

constructs an account named `x` with an initial balance of 70 and performs a withdrawal of 50 upon it.

A bank has hundreds, if not thousands, of customers, so a program that maintains the bank’s accounts must construct many `BankAccount` objects. An array is a good structure for saving the objects. Here is one simple modelling: Say that a bank is able to maintain at most 100 accounts. Each account is given an identification number in the range of 0 to 99. The program that does accounting for the bank will use an array like this:

```
BankAccount[] bank = new BankAccount[100];
```

This constructs an array that has 100 elements, such that *each element holds the initial value, null*. (There are no accounts yet, just a structure to hold the accounts.)

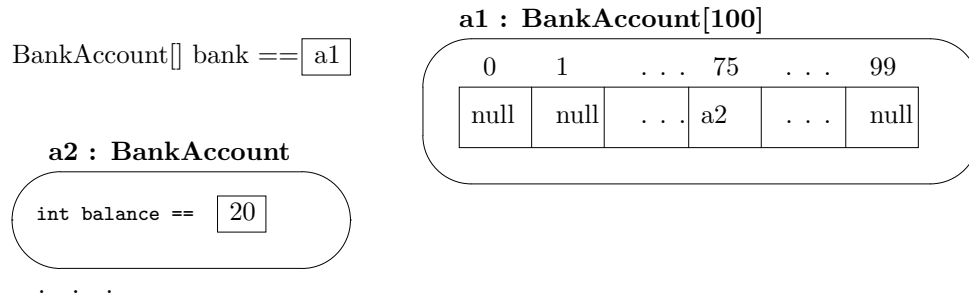
Now, say that a customer opens an account at the bank with an initial deposit of 200, and the customer wants her account to have the identification number, 75. The programming statements that enact this request might read,

```
BankAccount new_account = new BankAccount(200);
bank[75] = new_account;
```

Or, more directly stated,

```
bank[75] = new BankAccount(200);
```

Here is a diagram of the array and the newly constructed object whose address is saved within the array:



If the customer wishes to withdraw 60 from the account, this invocation,

```
bank[75].withdraw(60)
```

will perform the action. *Note that `bank[75]` names the bank account object whose `withdraw` method is invoked.* This example emphasizes that the array, `bank`, holds objects that are indexed by integers.

Next, say that the customer closes her account, so that it is no longer needed by the bank. The bank “erases” the account by stating,

```
bank[75] = null;
```

Since `null` means “no value,” this means another customer might open an account and choose 75 for its identification number.

As just noted, an array can be partially filled with objects; we can test if an array element is occupied by comparing the element’s value to `null`. For example, here is a loop that prints the balances of all accounts in array `bank`, skipping over those elements that are empty:

```
for ( int i = 0; i != bank.length; i = i + 1 )
  { if ( bank[i] != null )
    { System.out.println( "Balance of account " + i
                        + " is " + bank[i].getBalance() );
    }
  }
```

With these basic ideas in mind, we can write a simple bank-accounting application that lets customers open bank accounts, make deposits and withdrawals, check the balances, and close the accounts when they are no longer needed—we use an array to hold the accounts, and we use the array’s indexes as the identification numbers for the accounts.

Of course, it is overly simplistic to use simple integers as identification numbers for bank accounts—we should use a more general notion of a *key* to identify an account. The key might be a word, or a sequence of numerals and letters, or even an object itself. The case study in the next section develops a more realistic approach to maintaining databases of objects where objects are identified by keys.

Exercises

1. Here is a class, `Counter`, that can remember a count:

```
public class Counter
{ private int c;

    public Counter(int v) { c = v; }
    public void increment() { c = c + 1; }
    public int getCount() { return c; }
}
```

Say that we change the declaration of `votes` in the vote-counting application in Figure 1 to be

```
Counter[] votes = new Counter[num_candidates];
```

Revise the vote-counting application to operate with this array.

2. Let's write a class that “models” the bank described in this section:

```
/** Bank models a collection of bank accounts */
public class Bank
{ BankAccount[] bank; // the array that holds the account
  int max_account; // the maximum account number

  /** Constructor Bank initialize the bank
   * @param how_many - the maximum number of bank accounts */
  public Bank(int how_many)
  { max_account = how_many;
    bank = new BankAccount[how_many];
  }

  /** addNewAccount adds a new account to the bank
   * @param id_number - the account's identification number; must be in
   * the range, 0..maximum_account_number - 1
   * @param account - the new bank account object
   * @return true, if the account is succesfully added;
```

```

    * return false, if the id_number is illegal */
public boolean addNewAccount(int id_number, BankAccount account)
{ boolean result = false;
  if ( id_number >= 0 && id_number < max_account
      && bank[id_number] == null ) // is id_number legal?
    { bank[id_number] = account;
      result = true;
    }
  return result;
}

/** getAccount finds a bank account
 * @param id_number - the identification number of the desired account
 * @return the bank account whose identification is id_number;
 * if there is no account with the id_number, return null */
public BankAccount getAccount(int id_number)
{ ... }

/** deleteAccount removes a bank account
 * @param id_number - the identification number of the account to be removed
 * @return true, if the deletion was successful;
 * return false, if no account has the id_number */
public boolean deleteAccount(int id_number)
{ ... }
}

```

We might use the class as follows:

```

Bank b = new Bank(500);
b.addNewAccount(155, new BankAccount(100));
...
BankAccount a = b.getAccount(155);
... a.deposit(200) ...

```

Write the two missing methods for class `Bank`.

3. Use class `Bank` in the previous exercise to write an application that lets a user construct new bank accounts, do deposits and withdrawals, and print balances.

8.6 Case Study: Databases

A large collection of information, such as a company's sales records or its customer accounts or its payroll information, is called a *database*. An important programming challenge is determining the proper structure for a database.

In simplest terms, a database is a “container” into which objects are inserted, located, and removed; the objects that are stored in a database are called *records*. An important feature about a record is that it is uniquely identified by its *key*, which is held within the record itself. Here are some examples of records:

- A bank’s database holds records of accounts. Each account record is uniquely identified by a multi-letter-and-digit account number. A typical record would contain information such as
 1. its key, the multi-digit integer, which identifies the account
 2. the name (or some other identification) of the account’s owner
 3. the amount of money held in the account
- A library’s database holds records of books. Each record has for its key the book’s catalog number. For example, the U.S. Library of Congress catalog number is a pair: an alphabetic string and a fractional number, such as **QA 76.8**. The records held in a library’s database would have these attributes:
 1. the key, which is the book’s catalog number
 2. the book’s title, author, publisher, and publication date
 3. whether or not the book is borrowed, and if so, by which patron
- The U.S. Internal Revenue Service database hold records of taxpayers. Each record is identified by a nine-digit social-security number. The record holds the number as its key and also holds the taxpayer’s name, address, and copies of the person’s tax reports for the past five years.

Although the example records just listed differ markedly in their contents, they share the common feature of possessing a key. This crucial feature helps us understand the function of a database:

A database is a container that locates records by using the records’ keys as indices.

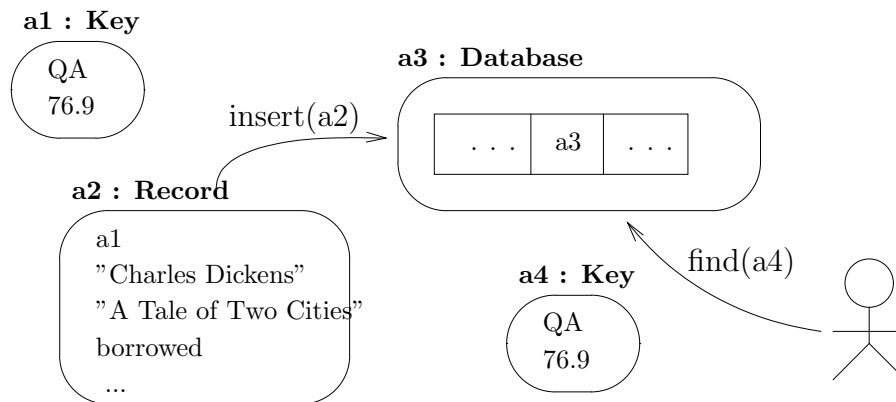
Compare this concept to that of an array: An array is a container that locates objects by using integer indices numbered 0, 1, 2, ..., and so on. A database is like a “smart array” that uses a record’s key to save and locate the record.

How can we model and build a general-purpose database in Java? Here are some crucial concepts:

1. keys are objects
2. records are objects, and a record holds as one of its attributes (the address of) its key object

3. a database is a kind of “array” of record objects; it must have methods for inserting a record, finding a record, and deleting a record
4. when the database’s user wishes to insert a record into the database, she calls the databases’ `insert` method, supplying the record as the argument; when she wishes to find a record, she calls the `find` method, supplying a key object as an argument; when she wishes to delete a record, she calls the `delete` method, supplying a key object as an argument

For example, if we build a database to hold library books, the key objects will be Library of Congress catalog numbers, and each record object will hold (the address of) a key object and information about a book. Such records are inserted, one by one, into the database. When a user wishes to find a book in the database, she must supply a key object to the database’s `find` method and she will receive in return (the address of) the desired book object; an informal picture of this situation looks like this:



The picture suggests that the database will operate the same, *regardless of whether books, bank accounts, and so on, are saved*. As long as the records—whatever they are—hold keys, the database can do its insertions, lookups, and deletions, *by manipulating the records’ keys and not the records themselves*. This is strongly reminiscent of arrays, which can hold a variety of objects without manipulating the objects themselves.

So, how does a database manipulate a key? Regardless of whether keys are numbers or strings or pairs of items, *keys are manipulated by comparing them for equality*. Consider a lookup operation: The database receives a key object, and the database searches its collection of records, asking each record to tell its key, so that each key can be compared for equality to the desired key. When an equality is found true, the corresponding record is returned. *This algorithm operates the same whether integers, strings, or whatever else is used for keys*.

In summary,

1. The **Database** holds a collection of **Record** objects, where each **Record** holds a **Key**

object. The remaining structure of the `Records` is unimportant and unknown to the database.

2. The `Database` will possess `insert`, `find`, and `delete` methods.
3. `Records`, regardless of their internal structure, will possess a `getKey` method that returns the `Record`'s `Key` object when asked.
4. `Key` objects, regardless of their internal structure, will have an `equals` method that compares two `Keys` for equality and returns true or false as the answer.

We are now ready to design and build a database subassembly in Java. We will build a subassembly—not an entire program—such that the subassembly can be inserted as the model into a complete application. We follow the usual stages for design and construction:

1. *State the subassembly's desired behaviors.*
2. *Select an architecture for the subassembly.*
3. *For each of the architecture's components, specify classes with appropriate attributes and methods.*
4. *Write and test the individual classes.*
5. *Integrate the classes into a complete subassembly.*

8.6.1 Behaviors

Regardless of whether a database holds bank accounts, tax records, or payroll information, its behaviors are the same: a database must be able to insert, locate, and delete records based on the records' keys. We plan to write a `class Database` so that an application can construct a database object by stating,

```
Database db = new Database(...);
```

Then, the application might insert a record—call it `r0`—into `db` with a method invocation like this:

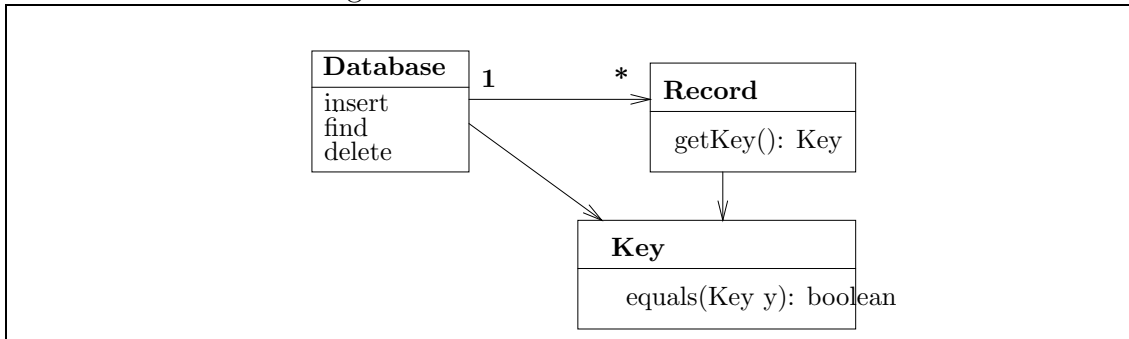
```
db.insert(r0);
```

As stated earlier, each record possesses its own key. Say that record `r0` holds object `k0` as its key. To retrieve record `r0` from the database, we use a command like this:

```
Record r = db.find(k0);
```

This places the address of record `r0` into variable `r` for later use. We can delete the record from the database by stating:

Figure 8.2: architecture for a database



```
db.delete(k0);
```

Notice that variable `r` still holds the address of the record, but the record no longer lives in the database.

The above behaviors imply nothing about the techniques that the database uses to store and retrieve records; *these activities are internal to class Database and are best left unknown to the database's users.*

8.6.2 Architecture

The previous examples suggest there are at least three components to the database's design: the `Database` itself, the `Records` that are inserted into it, and the `Keys` that are kept within records and are used to do insertions, lookups, and deletions. The class diagram in Figure 2 lists these components and their dependencies. There is a new notation in the Figure's class diagram: The annotation, `1 --> *`, on the arrow emphasizes that one `Database` collaborates with (or collects) multiple `Records`, suggesting that an array will be useful in the coding of `class Database`. As noted earlier, whatever a `Record` or `Key` might be, the methods `getKey` and `equals` are required. (The format of the `equals` method will be explained momentarily.)

8.6.3 Specifications

To keep its design as general as possible, we will not commit `class Database` to saving any particular form of `Record`—the only requirement that a database will make of a record is that a record can be asked for its key. Similarly, the only requirement a database will make of a key is that the key can be compared to another key for an equality check.

Since `class Database` must hold multiple records, its primary attribute will be an array of records, and the database will have at least the three methods listed in Figure 2.

Figure 8.3: specifications for database building

Database	a container for data items, called Records
Attribute	
<code>private Record[] base</code>	Holds the records inserted into the database.
Methods	
<code>insert(Record r): boolean</code>	Attempts to insert the record, r , into the database. Returns true if the record is successfully added, false otherwise.
<code>find(Key k): Record</code>	Attempts to locate the record whose key has value k . If successful, the address of the record is returned, otherwise, null is returned.
<code>delete(Key k): boolean</code>	Deletes the record whose key has value k . If successful, true is returned; if no record has key k , false is returned.
Record	a data item that can be stored in a database
Methods	
<code>getKey(): Key</code>	Returns the key that uniquely identifies the record.
Key	an identification, or “key,” value
Methods	
<code>equals(Key m): boolean</code>	Compares itself to another key, m , for equality. If this key and m are same key value, then true is returned; if m is a different key value, then false is returned.

The specification for **Record** is kept as minimal as possible: whatever a record object might be, it has a function, `getKey`, that returns the key that uniquely identifies the record. Similarly, it is unimportant whether a key is a number or a string or whatever else; therefore, we require only that a key possesses a method, `equals`, that checks the equality of itself to another key.

Table 3 presents the specifications that summarize our assumptions about databases, records, and keys. Because we have provided partial (incomplete) specifications for **Record** and **Key**, many different classes might implement the two specifications. For example, we might write `class Book` to implement a **Record** so that we can build a database of books, or we might write `class BankAccount` to implement a database of bank accounts. Different classes of keys might also be written, if only because books use different keys than do bank accounts.

Key's specification deserves a close look: the specification is written as if keys are objects (and not mere `ints`). For this reason, given two **Key** objects, **K1** and **K2**, we

must write `K1.equals(K2)` to ask if the two keys have the same value. (This is similar to writing `S1.equals(s2)` when comparing two strings, `S1` and `S2`, for equality.) We exploit this generality in the next section.

8.6.4 Implementation

The specifications for `Record` and `Key` make it possible to write a complete coding for `class Database` without knowing any details about the codings for the records and keys. Let's consider the implementation of `class Database`.

The database's primary attribute is an array that will hold the inserted records. `class Database` must contain this field declaration:

```
private Record[] base;
```

The constructor method for the class will initialize the field to an array:

```
base = new Record[HOW_MANY_RECORDS];
```

where all the array's elements have value `null`, because the array is empty. Records will be inserted into the database one by one. To do an `insert(Record r)`, follow this algorithm:

1. Search array `base` to see if `r` is present. (More precisely, search `base` to see if a record with the same key as `r`'s key is already present.)
2. If `r` is not in `base`, then search for the first element in `base` that is empty (that is, holds value `null`).
3. Insert `r` into the empty element.

Each of the algorithm's three steps requires more refinement: To fill in details in the first step, say that we write a helper method, `findLocation`, which searches the array for a record whose key equals `k`. The helper method might be specified like this:

```
/** findLocation is a helper method that searches base for a record
 * whose key is k. If found, the array index of the record within
 * base is returned, else -1 is returned. */
private int findLocation(Key k)
```

Then, Step 1 of the algorithm is merely,

```
if ( findLocation(r.getKey()) == -1 )
```

because `r.keyOf()` extracts the key held within record `r`, and a result of `-1` from `findLocation` means that no record with the same key is already present.

Step 2 of the algorithm is clearly a searching loop, and we use the techniques from Chapter 7 to write this loop, which searches for the first empty element in `base` where a new record can be inserted:

```

boolean found_empty_place = false;
int i = 0;
while ( !found_empty_place && i != base.length )
    // so far, all of base[0]..base[i-1] are occupied
    { if ( base[i] == null ) // is this element empty?
        { found_empty_place = true; }
      else { i = i + 1; }
    }

```

When this loop completes, `i` holds the index of the first empty element in `base`, meaning that Step 3 is just `base[i] = r`, *unless* array `base` is completely filled with records and there is no available space. What should we do in the latter situation?

Because Java arrays are objects, it is possible to construct a new array object that is larger than the current array and copy all the elements from the current array to the new array. Here is a standard technique for doing so:

```

// This constructs a new array twice as large as base:
Record[] temp = new Record[base.length * 2];
// Copy elements in array named by base into temp:
for ( int j = 0; j != base.length; j = j + 1 )
    { temp[j] = base[j]; }
// Change base to hold address of temp:
base = temp;

```

The last assignment, `base = temp`, *copies the address of the larger array into array variable base, meaning that base once again holds the address of an array of records.*

BeginFootnote: If you have studied the Java libraries, perhaps you discovered class `Vector`, which behaves like an array but automatically expands to a greater length when full. The technique that a Java `Vector` uses to expand is exactly the one presented above. *EndFootnote.*

Figure 4 displays the completed version of `insert`.

Next, we consider how to delete an element from the database: The algorithm for method, `delete(Key k)`, would go,

1. Search array `base` to see if a record with the key, `k`, is present.
2. If such a record is located, say, at element `index`, then delete it by assigning, `base[index] = null`.

We use the helper method, `findLocation`, to code Step 1. We have this coding:

```

int index = findLocation(k);
if ( index != -1 )
    { base[index] = null; }

```

See Figure 4 for the completed method.

We can write the `lookup` method so that it merely asks `findLocation` to find the desired record in the array. Again, see Figure 4.

To finish, we must write the `findLocation` method, which finds the record in array `base` whose key is `k`. The algorithm is a standard searching loop, but there is a small complication, because array `base` might have `null` values appearing in arbitrary places, due to deletions of previously inserted records:

```
private int locationOf(Key k)
{ int result = -1; // recall that -1 means ‘‘not found’’
  boolean found = false;
  int i = 0;
  while ( !found && i != base.length )
    { if ( base[i] != null // is this element occupied?
          && base[i].keyOf().equals(k) ) // is it the desired record?
      { found = true;
        result = i;
      }
      else { i = i + 1; }
    }
  return result; // return array index of the record found
}
```

Note the conditional statement in the loop’s body:

```
if ( base[i] != null // is this array element occupied?
     && base[i].keyOf().equals(k) ) // is it the desired record?
  { ... } // we found the record at array element, i
else { i = i + 1; } // the record is not yet found; try i + 1 next
```

The test expression first asks if there is a record stored in element, `base[i]`, and if the answer is true, then the element’s key (namely, `base[i].keyOf()`) is compared for equality to the desired key, `k`.

The completed `Database` class appears in Figure 4. In addition to attribute `base`, we define the variable, `NOT_FOUND`, as a memorable name for the `-1` answer used to denote when a search for a record failed.

The coding presents several lessons:

- Although `class Database` appears to store records based on their keys, a more primitive structure, an array, is used inside the class to hold the records. The helper method, `findLocation`, does the hard work of using records’ keys as if there were “indices.”
- Aside from the `getKey` and `equals` methods, nothing is known about the records and keys saved in the database. This makes `class Database` usable in a variety of applications, we see momentarily.

Figure 8.4: class Database

```

/** Database implements a database of records */
public class Database
{ private Record[] base; // the collection of records
  private int NOT_FOUND = -1; // int used to denote when a record not found

  /** Constructor Database initializes the database
   * @param initial_size - the size of the database */
  public Database(int initial_size)
  { if ( initial_size > 0 )
    { base = new Record[initial_size]; }
    else { base = new Record[1]; }
  }

  /** findLocation is a helper method that searches base for a record
   * whose key is k. If found, the index of the record is returned,
   * else NOT_FOUND is returned. */
  private int findLocation(Key k)
  { int result = NOT_FOUND;
    boolean found = false;
    int i = 0;
    while ( !found && i != base.length )
      { if ( base[i] != null && base[i].keyOf().equals(k) )
        { found = true;
          result = i;
        }
        else { i = i + 1; }
      }
    return result;
  }

  /** find locates a record in the database based on a key
   * @param key - the key of the desired record
   * @return (the address of) the desired record;
   * return null if record not found. */
  public Record find(Key k)
  { Record answer = null;
    int index = findLocation(k);
    if ( index != NOT_FOUND )
      { answer = base[index]; }
    return answer;
  }
  ...

```

Figure 8.4: class Database (concl.)

```

/** insert inserts a new record into the database.
 * @param r - the record
 * @return true, if record added; return false if record not added because
 * another record with the same key already exists in the database */
public boolean insert(Record r)
{ boolean success = false;
  if ( findLocation(r.keyOf()) == NOT_FOUND ) // r not already in base?
    { // find an empty element in base for insertion of r:
      boolean found_empty_place = false;
      int i = 0;
      while ( !found_empty_place && i != base.length )
        // so far, all of base[0]..base[i-1] are occupied
        { if ( base[i] == null ) // is this element empty?
          { found_empty_place = true; }
          else { i = i + 1; }
        }
      if ( found_empty_place )
        { base[i] = r; }
      else { // array is full! So, create a new one to hold more records:
        Record[] temp = new Record[base.length * 2];
        for ( int j = 0; j != base.length; j = j + 1 )
          { temp[j] = base[j]; } // copy base into temp
        temp[base.length] = r; // insert r in first free element
        base = temp; // change base to hold address of temp
      }
      success = true;
    }
  return success;
}

/** delete removes a record in the database based on a key
 * @param key - the record's key (identification)
 * @return true, if record is found and deleted; return false otherwise */
public boolean delete(Key k)
{ boolean result = false;
  int index = findLocation(k);
  if ( index != NOT_FOUND )
    { base[index] = null;
      result = true;
    }
  return result;
}
}

```

- Because the array of records can be filled, we use a standard technique within the `insert` method to build a new, larger array when needed.

8.6.5 Forms of Records and Keys

When we use `class Database` to hold records, we must write a `class Record` and a `class Key`. The contents of these classes depends of course on the application that requires the database, but we know from Table 3 that `class Record` must include a `getKey` method and `class Key` must include an `equals` methods. Figure 5 shows one such implementation: a record that models a simple bank account and a key that is merely a single integer value.

The `Record` in Figure 5 has additional methods that let us do deposits and check balances of a bank account, but the all-important `getKey` method is present, meaning that the record can be used with `class Database` of Figure 4.

In order to conform to the requirements demanded by `class Database`, the integer key must be embedded within a `class Key`. This means the integer is saved as a private field within `class Key` and that the `equals` method must be written so that it asks another key for its integer attribute, by means of an extra method, `getInt`.

Here is how we might use the classes in Figure 5 in combination with Figure 4. Perhaps we are modelling a bank, and we require this database:

```
Database bank = new Database(1000);
```

When a customer opens a new account, we might ask the customer to select an integer key for the account and make an initial deposit:

```
int i = ...some integer selected by the customer...;
int start_balance = ...some initial deposit by the customer...;
Key k1 = new Key(i);
boolean success = bank.insert( new Record(start_balance, k1) );
System.out.println("account inserted = " + success);
```

The fourth statement both constructs the new account and inserts it into the database.

Later, if the account must be fetched so that its balance can be checked, we can find it and print its balance like this:

```
Record r = bank.find(k1); // recall that k1 is the account's key
if ( r != null ) // did we successfully fetch the account?
    { System.out.println(r.getBalance()); }
```

To show that the database can be used in a completely different application, we find in Figure 6 a new coding of record and key, this time for library books. Now, `class Record` holds attributes for a book's title, author, publication date, and catalog number; the catalog number serves as the book's key.

Figure 8.5: BankAccount Record and AccountKey

```

/** Record models a bank account with an identification key */
public class Record
{ private int balance; // the account's balance
  private Key id;      // the identification key

  /** Constructor Record initializes the account
   * @param initial_amount - the starting account balance, a nonnegative.
   * @param id - the account's identification key */
  public Record(int initial_amount, Key id)
  { balance = initial_amount;
    key = id;
  }

  /** deposit adds money to the account.
   * @param amount - the amount of money to be added, a nonnegative int */
  public void deposit(int amount)
  { balance = balance + amount; }

  /** getBalance reports the current account balance
   * @return the balance */
  public int getBalance() { return balance; }

  /** getKey returns the account's key
   * @return the key */
  public int getKey() { return key; }
}

/** Key models an integer key */
public class Key
{ private int k; // the integer key

  /** Constructor Key constructs the Key
   * @param i - the integer that uniquely defines the key */
  public Key(int i) { k = i; }

  /** equals compares this Key to another for equality
   * @param c - the other key
   * @return true, if this key equals k's; return false, otherwise */
  public boolean equals(Key c)
  { return ( k == c.getInt() ); }

  /** getInt returns the integer value held within this key */
  public int getInt() { return k; }
}

```

Figure 8.6: Book Record and CatalogNumber Key

```
/** Record models a Library Book */
public class Record
{ // the names of the fields describe their contents:
  private Key catalog_number;
  private String title;
  private String author;
  private int publication_date;

  /** Constructor Record constructs the book.
   * @param num - the book's catalog number
   * @param a - the book's author
   * @param t - the book's title */
  public Record(Key num, String a, String t, int date)
  { catalog_number = num;
    title = t;
    author = a;
    publication_date = date;
    is_borrowed_by_someone = false;
  }

  /** getKey returns the key that identifies the record
   * @return the key */
  public Key getKey() { return catalog_number; }

  /** getTitle returns the book's title
   * @return the title */
  public String getTitle() { return title; }

  /** getAuthor returns the book's author
   * @return the author */
  public String getAuthor() { return author; }

  /** getDate returns the book's publication date
   * @return the date */
  public int getDate() { return publication_date; }
}
```

Figure 8.6: CatalogNumber Key (concl.)

```
/** Key models a Library-of-Congress-style id number,
 * consisting of a letter code concatenated to a decimal number */
public class Key
{ private String letter_code; // the letter code, e.g., "QA"
  private double number_code; // the number code, e.g., 76.884

  /** Constructor Key constructs a catalog number
   * @param letters - the letter code, e.g., "QA"
   * @param num - the decimal number code, e.g., 76.884 */
  public Key(String letters, double num)
  { letter_code = letters;
    number_code = num;
  }

  /** equals returns whether the catalog number held within this object
   * is identical to the catalog number held within c
   * @param c - the other catalog number
   * @return true, if this catalog number equals c; return false, otherwise */
  public boolean equals(Key c)
  { String s = c.getLetterCode();
    double d = c.getNumberCode();
    return ( s.equals(letter_code) && d == number_code );
  }

  /** getLetterCode returns the letter code part of this catalog number
   * @return the letter code, e.g., "QA" */
  public String getLetterCode() { return letter_code; }

  /** getNumberCode returns the number code part of this catalog number
   * @return the number code, e.g., "76.884" */
  public double getNumberCode() { return number_code; }
}
```

The structure of the catalog number is more complex: Its `class Key` holds a string and a double, because we are using the U.S. Library of Congress coding for catalog numbers, which requires a string and a fractional number. The class's `equals` method compares the strings and fractional numbers of two keys.

Here is a short code fragment that constructs a database for a library and inserts a book into it:

```
Database library = new Database(50000);

Record book = new Book( new Key("QA", 76.8), "Charles Dickens",
                        "Great Expectations", 1860 );
library.insert(book);

// We might locate the book this way:
Key lookup_key = new Key("QA", 76.8);
book = library.find(lookup_key);

// We can delete the book, if necessary:
boolean deleted = library.delete(lookup_key);
```

As noted by the statement, `Key lookup_key = new Key("QA", 76.8)`, we can manufacture keys as needed to perform lookups and deletions.

It is a bit unfortunate that the bank account record in Figure 5 was named `class Record` and that the book record in Figure 6 was also named `class Record`; more descriptive names, like `class BankAccount` and `class Book` would be far more appropriate *and would let us include both classes in the same application if necessary*. (Perhaps a database must store both bank accounts and books together, or perhaps one single application must construct one database for books and another for bank accounts.)

Of course, we were forced to use the name, `class Record`, for both records because of the coding for `class Database` demanded it. The Java language lets us repair this naming problem with a new construction, called a Java `interface`. We will return to the database example in the next chapter and show how to use a Java `interface` with `class Database` to resolve this difficulty.

Exercise

Write an application that uses `class Database` and classes `Record` and `Key` in Figure 5 to help users construct new bank accounts and do deposits on them.

8.7 Case Study: Playing Pieces for Card Games

Computerized games must model a game's playing pieces, the playing board, and even the game's players. A classic example of "playing pieces" are playing cards. Even

if you have no interest in building or playing card games, modelling playing cards is useful, because it shows how to model a set of pieces that must behave similarly.

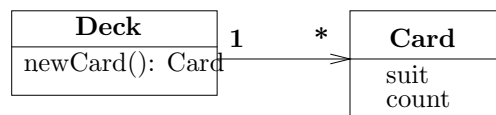
Behavior of Cards and Decks

Perhaps we do not think of playing cards as having behaviors, but they are playing pieces, and a playing piece, whether it be a chess pawn or a queen-of-hearts card, has abilities or attributes that make it distinct from other playing pieces. A playing card is a good example: It has a *suit* (diamonds, hearts, clubs, or spades) and *count* (ace through ten, jack, queen, or king), e.g., hearts and queen. A card's suit and count are attributes and not behaviors in themselves, but a card game assigns behaviors based on the attributes, e.g., in the card game, Blackjack, a card that has the count of queen has the ability to score 10 points for the player who holds it.

In almost every game, playing pieces must be placed onto or into a container or playing board—cards are collected into a container called a *deck*. Unlike the database developed in the previous section, a card deck begins completely filled with cards. The deck's primary behavior is to surrender a card from itself whenever asked, until the deck is finally empty.

Card games often use other forms of containers. For example, each player might have its own personal container for cards, called a *hand (of cards)*. A hand is initialized so that it is empty and cards can be added to it, one by one.

For this small case study, we will design a subassembly consisting only of ordinary playing cards and a deck to hold them. The architecture for the two classes is simple:



Specification

The specification for a card deck is presented in Table 7. Because it is a container for cards, `class Deck` requires an attribute that is an array. The class's methods include one that returns a card and one that replies whether or not there are more cards to return.

`CardDeck` collaborates with `class Card`, which we consider next. As noted earlier, a playing card has two crucial attributes: its suit and its count, as seen in Table 8. Of course, `class Card` will be coded to have accessor methods that return a card's suit and count.

Implementation

There is a technical issue that we should resolve before we write the two classes: When people use playing cards, they prefer to use the names of suits and counts,

Figure 8.7: specification for class CardDeck

<code>class CardDeck</code>	models a deck of cards
Attribute	
<code>private Card[] deck</code>	container for the cards left in the deck
Methods	
<code>newCard(): Card</code>	return a card from the deck; if the deck is empty, return <code>null</code>
<code>moreCards(): boolean</code>	return <code>true</code> if more cards remain in the deck; return <code>false</code> , otherwise

Figure 8.8: specification of class Card

<code>class Card</code>	models a playing card
Attributes	
<code>private String suit</code>	the card's suit, e.g., spades, hearts, diamonds, clubs
<code>private int count</code>	the card's count, e.g., ace, 2, 3, ..., king

like “hearts,” “clubs,” “ace,” and “queen.” These are values, just like integers, and we would like to use such values when we construct the playing cards, e.g., “`new Card(queen, hearts)`.”

We define values like “queen” and “hearts” in Java by declaring a `public static final` variable for each such value, and place the public static final variables within `class Card`. We see this in Figure 9, which presents `class Card`.

The `public static final` variables declared in `class Card` are public names that can be used by the other components of an application; the names are used as if they are new values. For example, other components can refer to the values, `Card.ACE`, `Card.DIAMOND`, and so on. Now, we can say:

```
Card c = new Card(Card.HEARTS, Card.QUEEN)
```

to construct a queen-of-hearts object. And, we can ask,

```
if ( c.getCount() == Card.QUEEN ) { ... }
```

Figure 8.9: playing card

```
/** Card models a playing card */
public class Card
{ // definitions that one can use to describe the value of a card:
  public static final String SPADES = "spades";
  public static final String HEARTS = "hearts";
  public static final String DIAMONDS = "diamonds";
  public static final String CLUBS = "clubs";

  public static final int ACE = 1;
  public static final int JACK = 11;
  public static final int QUEEN = 12;
  public static final int KING = 13;

  public static final int SIZE_OF_ONE_SUIT = 13; // how many cards in one suit

  // These are the card's attributes:
  private String suit;
  private int count;

  /** Constructor Card sets the suit and count.
   * @param s - the suit
   * @param c - the count */
  public Card(String s, int c)
  { suit = s;
    count = c;
  }

  /** getSuit returns the card's suit. */
  public String getSuit()
  { return suit; }

  /** getCount returns the card's count. */
  public int getCount()
  { return count; }
}
```

But remember that the `public static final` variables are merely names for integers and strings. For example, since `Card.QUEEN` is the name of an integer, we can state,

```
if ( c.getCount() >= Card.QUEEN ) { ... }
```

because integers can be compared by greater-than.

As always, the keyword, `public`, means that the variable can be referenced by other classes; the keyword, `final`, states that the variable name can not be changed by an assignment; the value is forever constant. Finally, the keyword, `static`, ensures that the variable is *not* copied as a field into any `Card` objects that are constructed from `class Card`.

It is traditional to declare `public static final` variables with names that are all upper-case letters.

We have used `public static final` variables already when in previous chapters we used predefined Java-library values like `Color.red` (to paint on a graphics window) and `Calendar.DAY_OF_MONTH` (to get the date from a Gregorian calendar object).

The remainder of `class Card` is simple—it contains a constructor method, which initializes a card object’s suit and count, and two accessor methods, which return the suit and count.

Next, we consider `class CardDeck`. Its attributes are

```
private int card_count; // how many cards remain in the deck
private Card[] deck = new Card[4 * Card.SIZE_OF_ONE_SUIT];
    // invariant: elements deck[0]..deck[card_count - 1] hold cards
```

Array `deck` is constructed to hold the four suits’s worth of cards, where the quantity of cards in a suit is the static variable defined in `class Card`.

The class’s constructor method must fill array `deck` with a complete collection of cards. As Figure 8 shows, a helper method, `createSuit` knows how to generate one complete suit of cards and insert it into the array; therefore, the constructor can be written as

```
createSuit(Card.SPADES);
createSuit(Card.HEARTS);
createSuit(Card.CLUBS);
createSuit(Card.DIAMONDS);
```

The helper method is written so that it inserts the cards into the array in ascending order. This is *not* the ideal state for the deck for playing a typical card game—the deck’s cards are expected to be randomly mixed, or “shuffled.”

Rather than write a method that randomly mixes the elements in the array, we can write the `newCard` method so that when it is asked to remove a card from the array it randomly calculates an array index from which the card is extracted. The algorithm might go like this:

1. Randomly calculate an integer in the range of 0 to `card_count - 1`; call it `index`.
2. Remove the card at element `deck[index]`
3. Fill the empty element at `index` by shifting leftwards one element the cards in the range, `deck[index + 1]` up to `deck[card_count - 1]`.
4. Decrease `card_count` by one.

Step 1 can be done with the built-in Java method, `Math.random()`, which computes a nonnegative pseudo-random fraction less than 1.0:

```
int index = (int)(Math.random() * card_count);
```

The computation, `Math.random() * card_count`, generates a nonnegative fractional number that must be less than `card_count` (why?), and the cast, `(int)`, truncates the fractional part, leaving an integer in the range, 0 to `card_count - 1`.

Step 3 is performed with a simple loop:

```
for ( int i = index + 1; i != card_count; i = i + 1 )
    // so far, cards from index+1 to i-1 have been shifted left
    // in the array by one position
    { deck[i - 1] = deck[i]; }
```

The completed version of the method is in Figure 10.

Exercises

1. Write a test application that creates a new card, the queen of hearts, and then asks the card what its suit and count are. The program prints the answers it receives in the command window. Does your program print `Queen` or `12`? What solution do you propose so that the former is printed?
2. Write an application that creates a new deck of cards and asks the deck to deal 53 cards. (This is one more than the deck holds!) As the deck returns cards one by one, print in the command window the count and suit of each card.
3. Write an application that lets a user request cards one by one, until the user says, “stop.”
4. Card decks are used with card games. A typical card game has a *dealer* and several *players*. A dealer owns a card deck and gives cards from the deck to the players. The following specifications summarize the behavior of dealer and player:

Figure 8.10: class CardDeck

```

/** CardDeck models a deck of cards. */
public class CardDeck
{
    private int card_count; // how many cards remain in the deck
    private Card[] deck = new Card[4 * Card.SIZE_OF_ONE_SUIT];
        // invariant: elements deck[0]..deck[card_count - 1] hold cards

    /** Constructor CardDeck creates a new card deck with all its cards */
    public CardDeck()
    { createSuit(Card.SPADES);
      createSuit(Card.HEARTS);
      createSuit(Card.CLUBS);
      createSuit(Card.DIAMONDS);
    }

    /** newCard gets a new card from the deck.
     * @return a card not used before, or return null, if no cards are left */
    public Card newCard()
    { Card next_card = null;
      if ( card_count == 0 )
          { System.out.println("CardDeck error: no more cards"); }
      else { int index = (int)(Math.random() * card_count); // randomly choose
            next_card = deck[index];
            // once card is extracted from deck, shift other cards to fill gap:
            for ( int i = index+1; i != card_count; i = i + 1 )
                // so far, cards from index+1 to i-1 have been shifted left
                // in the array by one position
                { deck[i - 1] = deck[i]; }
            card_count = card_count - 1;
          }
      return next_card;
    }

    /** moreCards states whether the deck has more cards to give.
     * @return whether the deck is nonempty */
    public boolean moreCards()
    { return (card_count > 0); }

    /** createSuit creates a suit of cards for a new card deck. */
    private void createSuit(String which_suit)
    { for ( int i = 1; i <= Card.SIZE_OF_ONE_SUIT; i = i + 1 )
        { deck[card_count] = new Card(which_suit, i);
          card_count = card_count + 1;
        }
    }
}

```

<code>class Dealer</code>	models a dealer of cards
Responsibilities (methods)	
<code>dealTo(Player p)</code>	gives cards, one by one, to player <code>p</code> , until <code>p</code> no longer wants a card

<code>class Player</code>	models a player of a card game
Responsibilities (methods)	
<code>wantsACard(): boolean</code>	replies as to whether another card is desired
<code>receiveCard(Card c)</code>	accepts card <code>c</code> and adds it to its hand
<code>showCards(): Card[]</code>	returns an array of the cards that it holds

Write classes for these two specifications. (Write `class Player` so that a player wants cards until it has exactly five cards.) Next, write a controller that creates a dealer object and a player object. The controller tells the dealer to deal to the player. Then, the controller asks the player to reveal its cards.

5. Revise the controller in the previous Exercise so that there is an array of 3 players; the dealer deals cards to each player in turn, and then all players show their cards.

8.8 Two-Dimensional Arrays

Often, we display a collection of names and/or numbers in a table- or grid-like layout; here is an example. Say that 4 candidates participate in a “national election,” where the candidates compete for votes in 3 different regions of the country. (In the United States, for example, there are 50 such regions—the 50 states of the union.) Therefore, separate vote tallies must be kept for each of the 3 regions, and the votes are recorded

in a matrix:

		Candidate			
		0	1	2	3
election	Region				
	0	0	0	0	0
	1	0	0	0	0
2	0	0	0	0	

The Candidates' names are listed along the top of the matrix (for simplicity, we call them Candidate 0, ..., Candidate 3), labelling the *columns*, and the regions are listed along the left (they are Regions 0, 1, and 2), labelling the matrix's *rows*—We say that the matrix has three *rows* and four *columns*.

Thus, a vote in Region 1 for Candidate 3 would be recorded in the middle row within its rightmost element:

		Candidate			
		0	1	2	3
election	Region				
	0	0	0	0	0
	1	0	0	0	1
2	0	0	0	0	

Other votes are recorded this manner. When voting is completed, we can see which candidate received the most votes overall by adding each of the four columns.

In programming, we use a *two-dimensional array* to model a matrix like the one just seen. The above matrix is coded as a two-dimensional array in Java by stating

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

The data type of variable `election` is `int[][]` (“int array array”), indicating that individual integers in the collection named `election` are uniquely identified by means of *two* indexes. For example,

```
election[1][3]
```

identifies the integer element that holds the votes in Region 1 for Candidate 3.

The right-hand side of the initialization, `new int[3][4]`, constructs the array object that holds the collection of twelve integers, organized into 3 rows of 4 elements, each—three rows and four columns. It is helpful to visualize the collection as a matrix, like the ones just displayed. As usual for Java, the array's elements are initialized with zeros.

Let's do some small exercises with array `election`: To add one more vote from Region 1 for Candidate 3, we write

```
election[1][3] = election[1][3] + 1;
```

The phrase, `election[1][3]`, indicates the specific element in the matrix.

To give every candidate in Region 2 exactly 100 votes, we would say

```
for ( int i = 0; i != 4; i = i + 1 )
    { election[2][i] = 100; }
```

Here, we use `election[2][i]` to indicate a cell within Row 2 of the matrix—the value of `i` determines the specific cell in the row.

Similarly, to give Candidate 3 an additional 200 votes in every region, we would say

```
for ( int i = 0; i != 3; i = i + 1 )
    { election[i][3] = election[i][3] + 200; }
```

To print each candidate's grand total of votes, we write a nested for-loop that totals each column of the matrix:

```
for ( int j = 0; j != 4; j = j + 1 )
    { int votes = 0;
      for ( int i = 0; i != 3; i = i + 1 )
          { votes = votes + election[i][j]; }
      System.out.println("Candidate " + j + " has " votes + " votes");
    }
```

The previous for loop displays the standard pattern for examining each and every element of a matrix. Yet another example is printing the total votes cast in each region of the election, which requires that we total each row of the matrix:

```
for ( int i = 0; i != 3; i = i + 1 )
    { int total = 0;
      for ( int j = 0; j != 4; j = j + 1 )
          { total = total + election[i][j]; }
      System.out.println(total + " votes were cast in Region " + i);
    }
```

In the above example, the order of the loops is reversed, because rows are traversed, rather than columns.

Exercises

1. Create a two-dimensional array of integers, `m`, with 12 rows and 14 columns and initialize it so that each `m[i][j]` holds `i*j`.
2. Create a two-dimensional array of Strings, `m`, with 7 rows and 5 columns and initialize it so that each `m[i][j]` holds the string "Element " + `i` + " " + `j`.

3. Given this array, `int[][] r = new int[4][4]`, and given this nested for-loop that prints `r`'s contents,

```
for ( int i = 0; i != 4; i = i + 1 )
    { for ( int j = 0; j != 4; j = j + 1 )
        { System.out.print( r[i][j] + " " ); }
      System.out.println();
    }
```

write for-loops that initialize `r` so that the following patterns are printed:

(a) 1 0 0 0
 1 2 0 0
 1 2 3 0
 1 2 3 4

(b) 1 2 3 4
 0 3 4 5
 0 0 5 6
 0 0 0 7

(c) 1 0 1 0
 0 1 0 1
 1 0 1 0
 0 1 0 1

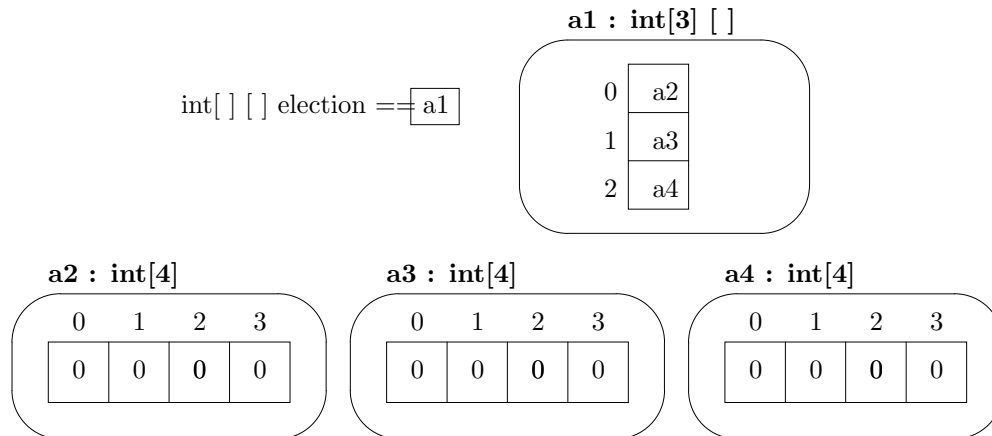
4. Modify the application in Figure 1 to use the `election` array. (Of course, a vote must be cast with a region number and a candidate number.)

8.9 Internal Structure of Two-Dimensional Arrays

We can imagine that a two-dimensional array is a matrix and draw it as such, but the array's true arrangement in computer storage is more complex: A two-dimensional array is in fact an *array of arrays*. To see this, reconsider

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

Here is its layout in computer storage:



In Java, a two-dimensional array is built in terms of multiple one-dimensional array objects. The diagram shows that the object at address `a1` is a one-dimensional array that holds the addresses of the matrix's rows.

For example, when we write, `election[1][3]`, this is computed to `a1[1][3]`, that is, the array object at `a1` is the one that will be indexed. Since element 1 within object `a1` is `a3`, the indexing is computed to `a3[3]`, which identifies element 3 in the array object at address `a3`.

Fortunately, this detailed address lookup is performed automatically and is hidden from the programmer—it is indeed safe to pretend that `election` is the name of a matrix of integer elements. But sometimes the knowledge of the matrix's underlying structure is exposed in the code that traverses the array, as in these for-loops, which print the total votes for each region:

```
for ( int i = 0; i != 3; i = i + 1 )
  { int[] region = election[i]; // region holds the address of a row
    int total = 0;
    for ( int j = 0; j != 4; j = j + 1 )
      { total = total + region[j]; }
    System.out.println(total + " votes were cast in Region " + i);
  }
```

This exposes that each row of matrix `election` is itself a (one-dimensional) array. Alas, we cannot treat a matrix's columns in a similar way.

Another consequence of the storage layout is that the “length” of a matrix is the number of rows it possesses. For the above example,

```
election.length
```

computes to 3. To learn the number of columns in a matrix, we ask for the length of one of the matrix's rows. For example,

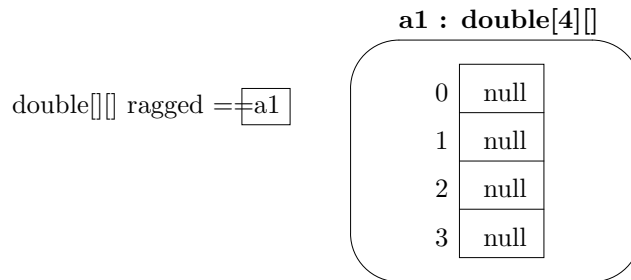
`election[0].length`

computes to 4.

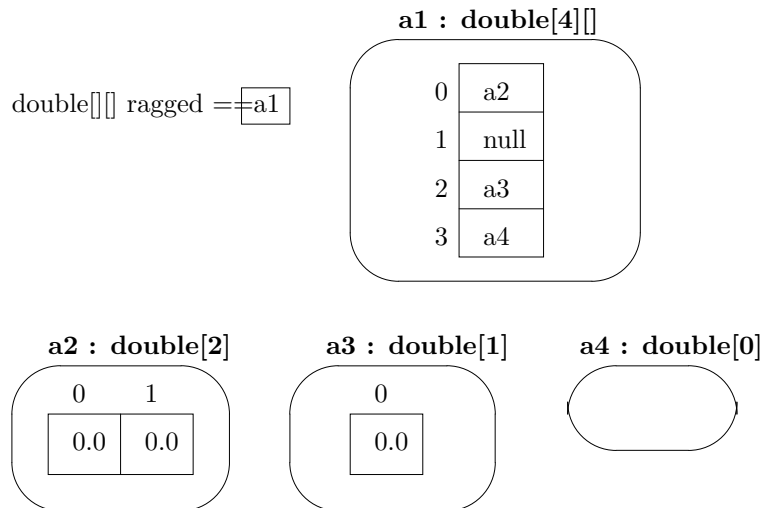
Finally, some care must be taken when asking for the number of columns of a two-dimensional array. Consider this odd example:

```
double[][] ragged = new double[4][];
double[0] = new double[2];
double[2] = new double[1];
double[3] = new double[0];
```

The first statement constructs an array variable, `ragged`, that will have four rows and an *undetermined* number of columns; it looks like this in storage:



The following three statements construct one-dimensional array objects and assign them to `ragged`'s elements, giving us



This is an example of a “ragged array”—a two-dimensional array whose rows have different lengths. For example, an election where different candidates are eligible in different regions of a country might be modelled by a ragged array.

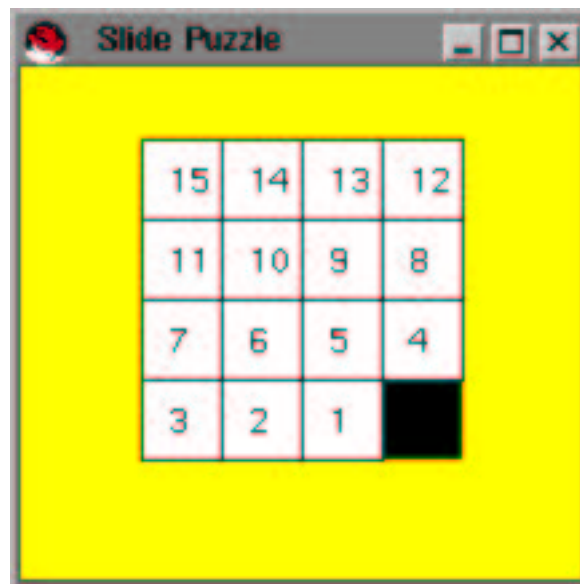
Notice that `ragged[0].length` equals 2, whereas `ragged[2].length` equals 1, whereas `ragged[1].length` generates an exception (run-time error), because there is no array of any length at the element. The array at element `ragged[3]` truly has length 0, meaning that there are no elements at all that can be indexed in that row.

8.10 Case Study: Slide-Puzzle Game

Two-dimensional arrays prove helpful for modelling game boards in computer games. As an example, we design and build a slide puzzle, which is a game board that holds numbered pieces that are moved by the user, one piece at a time.

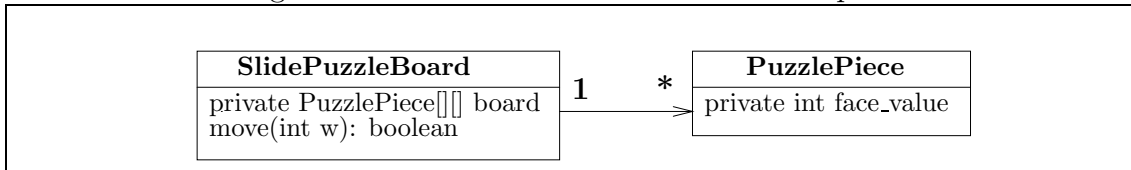
Behavior

The behavior of the puzzle game goes like this: the puzzle starts in this configuration:

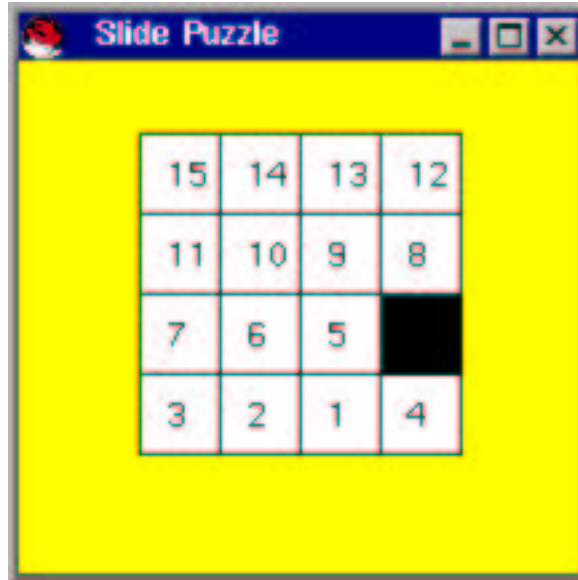


The user instructs the puzzle to move a piece by typing a number (in this configuration, only 1 or 4 would be allowed), and the game responds by moving the requested

Figure 8.11: architecture of model for slide puzzle



piece:



The user may request similar moves for as long as she wishes.

Architecture, Specification, and Implementation of the Model

The application that implements the puzzle will need a model subassembly that models the puzzle board and the pieces that move about the board; the model's architecture, presented in Figure 11, is simple and includes an initial specification of the two classes that must be written.

The `SlidePuzzleBoard`'s sole responsibility is to move a puzzle piece when asked, e.g., `move(4)` asks the board to move the piece whose face is labelled by 4.

A `PuzzlePiece` has only the attribute of the number on its face; it has no responsibilities. For this reason, its coding is simple and appears in Figure 12.

Next, we consider writing `class SlidePuzzleBoard`. The class's primary attribute is the array that holds the (addresses of the) puzzle pieces:

```
private PuzzlePiece[] [] board; // the array that holds the pieces
```

```
// one position on the board must be an empty space:
private int empty_row;
```

Figure 8.12: class PuzzlePiece

```

/** PuzzlePiece defines a slide-puzzle playing piece */
public class PuzzlePiece
{ private int face_value; // the value written on the piece's face

    /** Constructor PuzzlePiece creates a piece
     * @param value - the value that appears on the face of the piece */
    public PuzzlePiece(int value)
    { face_value = value; }

    /** valueOf returns the face value of the piece */
    public int valueOf()
    { return face_value; }
}

```

```

private int empty_col;
    // representation invariant: board[empty_row][empty_col] == null

```

Exactly one element within array `board` must be empty (hold `null`), and it is convenient to declare two fields, `empty_row` and `empty_col`, to remember the coordinates of the empty space.

Method `move(w)` must move the piece labelled by integer `w` into the empty space. For the move to succeed, the piece labelled by `w` must be adjacent to the empty space. This means the algorithm for `move(int w)` must perform the appropriate checks:

1. If the playing piece labelled by `w` is immediately above the empty space (marked by `empty_row` and `empty_col`), or if it is immediately below the empty space, or immediately to the left or right of the empty space,
2. Then, move the piece labelled by `w` to the empty space, and reset the values of `empty_row` and `empty_col` to be the position formerly occupied by `w`'s piece.

To write Step 1, we can make good use of this helper function, which looks at position, `row`, `col`, to see if the piece labelled `w` is there:

```

/** found returns whether the piece at position row, col is labeled w */
private boolean found(int w, int row, int col)

```

Then, Step 1 can be coded to ask the helper function to check the four positions surrounding the empty space whether piece `w` is there. See Figure 13 for the completed coding of method `move`.

The board's constructor method creates the `board` array and fills it with newly created puzzle pieces. Finally, we add a method that returns the contents of the

puzzle board. (This will be useful for painting the board on the display) The `contents` method returns as its result the value of the `board`. It is best that `contents` *not* return the address of its array `board`. (If it did, the client that received the address could alter the contents of the array!) Instead, a new two-dimensional array is created, and the addresses of the playing pieces are copied into the new array.

The Application's Architecture

The model subassembly neatly fits into a standard model-view-controller architecture, which we see in Figure 14.

Implementing the Controller

Figure 14's class diagram specifies the methods for the controller and view components. Consider the controller first; its `play` method should let the user repeatedly enter moves to the slide puzzle. The algorithm is merely a loop, which

1. tells the `PuzzleWriter` component to paint the current state of the puzzle;
2. reads the next move from the user;
3. tells the `SlidePuzzleBoard` to attempt the move

The controller and its `move` method is presented in Figure 15.

Implementing the View

The output-view class, `PuzzleWriter`, has the responsibility of painting the contents of the puzzle board in a graphics window. Because the controller, `class PuzzleController`, sends its repaint requests via method, `displayPuzzle`, the `displayPuzzle` method merely asks the view to repaint itself. The hard work of painting is done by `paintComponent`, which must display the puzzle pieces as a grid. A nested for-loop invokes the helper method, `paintPiece`, which paints each of the pieces, one by one. Figure 16 shows the coding.

Exercises

1. Test `class SlidePuzzleBoard` by creating an object, `board`, from it and immediately asking its `contents` method for the the board. Display the board with these loops:

```
PuzzlePiece[] [] r = board.contents();
for ( int i = 0; i != r.length; i = i+1 )
    { for ( int j = 0; j != r[i].length; j = j+1 )
        { if ( r[i][j] == null )
```

Figure 8.13: class SlidePuzzleBoard

```

/** SlidePuzzleBoard models a slide puzzle. */
public class SlidePuzzleBoard
{ private int size; // the board's size
  private PuzzlePiece[][] board; // the array that holds the pieces

  // one position on the board must be an empty space:
  private int empty_row;
  private int empty_col;
  // representation invariant: board[empty_row][empty_col] == null

  /** Constructor SlidePuzzleBoard constructs the initial puzzle, which has
   * the pieces arranged in descending numerical order.
   * @param s - the size of the puzzle, a positive integer (e.g., s==4 means
   * the puzzle is 4 x 4 and will have pieces numbered 15, 14, ..., 1) */
  public SlidePuzzleBoard(int s)
  { size = s;
    board = new PuzzlePiece[size][size];
    // create the individual pieces and place on the board in reverse order:
    for ( int num = 1; num != size * size; num = num + 1 )
      { PuzzlePiece p = new PuzzlePiece(num);
        int row = num / size;
        int col = num % size;
        // set p in a 'reversed position' on the board:
        board[size - 1 - row][size - 1 - col] = p;
      }
    // remember the location on the board where initially there is no piece:
    empty_row = size - 1;
    empty_col = size - 1;
  }

  /** contents returns the current state of the puzzle
   * @return a matrix that contains the addresses of the pieces */
  public PuzzlePiece[][] contents()
  { PuzzlePiece[][] answer = new PuzzlePiece[size][size];
    for ( int i = 0; i != size; i = i + 1 )
      { for ( int j = 0; j != size; j = j + 1 )
          { answer[i][j] = board[i][j]; }
      }
    return answer;
  }
  ...
}

```

Figure 8.13: class SlidePuzzleBoard (concl.)

```

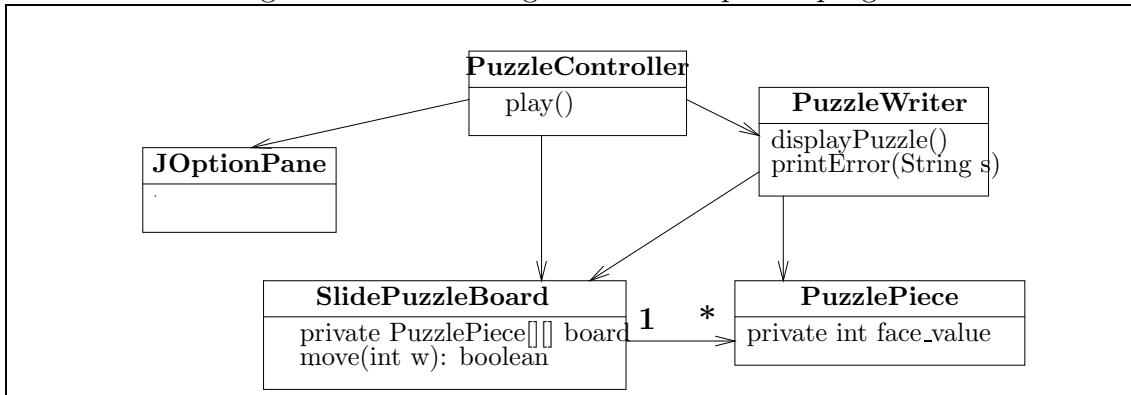
/** move moves a piece into the blank space, provided it is a legal move.
 * @param w - the face value of the piece that one wishes to move
 * @return true, if the piece labelled w was moved into the empty space;
 * return false if the piece cannot be moved into the empty space */
public boolean move(int w)
{ int NOT_FOUND = -1;
  int row = NOT_FOUND; // row and col will remember where piece w lives
  int col = NOT_FOUND;
  // try to find w adjacent to the empty space on the board:
  if ( found(w, empty_row - 1, empty_col) )
    { row = empty_row - 1;
      col = empty_col;
    }
  else if ( found(w, empty_row + 1, empty_col) )
    { row = empty_row + 1;
      col = empty_col;
    }
  else if ( found(w, empty_row, empty_col - 1) )
    { row = empty_row;
      col = empty_col - 1;
    }
  else if ( found(w, empty_row, empty_col + 1) )
    { row = empty_row;
      col = empty_col + 1;
    }

  if ( row != NOT_FOUND )
    { // move the piece into the empty space:
      board[empty_row][empty_col] = board[row][col];
      // mark the new empty space on board:
      empty_row = row;
      empty_col = col;
      board[empty_row][empty_col] = null;
    }
  return row != NOT_FOUND;
}

/** found returns whether the piece at position row, col is labeled v */
private boolean found(int v, int row, int col)
{ boolean answer = false;
  if ( row >= 0 && row < size && col >= 0 && col < size )
    { answer = ( board[row][col].valueOf() == v ); }
  return answer;
}
}

```

Figure 8.14: class diagram for slide-puzzle program



```

        { System.out.print("X "); }
        else { System.out.print( r[i][j].valueOf() + " "); }
    }
    System.out.println();
}

```

Next, use the object's `move` method to ask the board to move several numbers. Display the board resulting from each move.

2. Use the for-loops in the previous Exercise in an alternate implementation of class `PuzzleWriter` that displays the puzzle in the console window.
3. Test class `PuzzleWriter` by creating these objects,

```

SlidePuzzleBoard board = new SlidePuzzleBoard(3);
PuzzleWriter writer = new PuzzleWriter(board, 3);
writer.displayPuzzle();

```

where you use this “dummy” class:

```

public class SlidePuzzleBoard
{ private int size;

    public SlidePuzzleBoard(int s) { size = s; }

    public PuzzlePiece[][] contents()
    { PuzzlePiece[][] answer = new PuzzlePiece[size][size];
      int k = 0;
      for ( int i = 0; i != size; i= i+1 )
        { for ( int j = 0; j != size; j = j+1 )
          { answer[i][j] = new PuzzlePiece(k);

```

Figure 8.15: controller for slide puzzle program

```

import javax.swing.*;
/** PuzzleController controls the moves of a slide puzzle */
public class PuzzleController
{ private SlidePuzzleBoard board; // model
  private PuzzleWriter writer;    // output view

  /** Constructor PuzzleController initializes the controller
   * @param b - the model, the puzzle board
   * @param w - the output view */
  public PuzzleController(SlidePuzzleBoard b, PuzzleWriter w)
  { board = b;
    writer = w;
  }

  /** play lets the user play the puzzle */
  public void play()
  { while ( true )
    { writer.displayPuzzle();
      int i = new Integer
        (JOptionPane.showInputDialog("Your move:")).intValue();
      boolean good_outcome = board.move(i);
      if ( !good_outcome )
        { writer.printError("Bad move--puzzle remains the same."); }
    }
  }

  /** SlidePuzzle implements a 4 x 4 slide puzzle. Input to the program
   * is a sequence of integers between 1..15. The program never terminates. */
  public class SlidePuzzle
  { public static void main(String[] args)
    { int size = 4; // a 4 x 4 slide puzzle
      SlidePuzzleBoard board = new SlidePuzzleBoard(size);
      PuzzleWriter writer = new PuzzleWriter(board, size);
      PuzzleController controller = new PuzzleController(board, writer);
      controller.play();
    }
  }
}

```

Figure 8.16: output-view class for puzzle game

```

import java.awt.*; import javax.swing.*;
/** PuzzleWriter displays the contents of a slide puzzle */
public class PuzzleWriter extends JPanel
{ private SlidePuzzleBoard board; // the board that is displayed
  private int size; // the board's size
  private int piece_size = 30; // the size of one playing piece, in pixels
  private int panel_width; // the panel's width and height
  private int panel_height;

  /** Constructor PuzzleWriter builds the graphics window.
   * @param b - the slide puzzle that is displayed
   * @param s - the size of the slide puzzle, e.g., 4 means 4 x 4 */
  public PuzzleWriter(SlidePuzzleBoard b, int s)
  { board = b;
    size = s;
    panel_width = piece_size * size + 100;
    panel_height = piece_size * size + 100;
    JFrame my_frame = new JFrame();
    my_frame.getContentPane().add(this);
    my_frame.setTitle("Slide Puzzle");
    my_frame.setSize(panel_width, panel_height);
    my_frame.setVisible(true);
  }

  /** paintPiece draws piece p at position i,j in the window */
  private void paintPiece(Graphics g, PuzzlePiece p, int i, int j)
  { int initial_offset = piece_size;
    int x_pos = initial_offset + (piece_size * j);
    int y_pos = initial_offset + (piece_size * i);
    if ( p != null )
      { g.setColor(Color.white);
        g.fillRect(x_pos, y_pos, piece_size, piece_size);
        g.setColor(Color.black);
        g.drawRect(x_pos, y_pos, piece_size, piece_size);
        g.drawString(p.valueOf() + "", x_pos + 10, y_pos + 20);
      }
    else { g.setColor(Color.black);
          g.fillRect(x_pos, y_pos, piece_size, piece_size);
        }
  }
}
...

```

Figure 8.16: output-view class for puzzle game (concl.)

```

/** paintComponent displays the puzzle in the frame. */
public void paintComponent(Graphics g)
{ g.setColor(Color.yellow);
  g.fillRect(0, 0, panel_width, panel_height);
  PuzzlePiece[][] r = board.contents();
  for ( int i = 0; i != size; i= i+1 )
    { for ( int j = 0; j != size; j = j+1 )
      { paintPiece(g, r[i][j], i, j); }
    }
}

/** displayPuzzle displays the current state of the slide puzzle. */
public void displayPuzzle()
{ this.repaint(); }

/** printError displays an error message.
 * @param s - the error message */
public void printError(String s)
{ JOptionPane.showMessageDialog(null, "PuzzleWriter error: " + s ); }
}

```

```

        k = k + 1;
    }
}
return answer;
}
}

```

8.11 Testing Programs with Arrays

Programs with arrays prove notoriously difficult to test thoroughly, because we often use an arithmetic expression as an array index, and the expression might compute to an unacceptable integer. Here is an example: We have this method, which attempts to exchange two adjacent array elements:

```

/** exchange swaps the values in elements r[i] and r[i-1] */
public void exchange(int[] r, int i)
{ int temp = r[i];
  r[i] = r[i - 1];
  r[i - 1] = temp;
}

```

We wish to verify that the method behaves properly for all possible arguments. We have success for simple test cases, like this one,

```
int[] test0 = new int[10];
test0[3] = 3;
exchange(test0, 4);
```

But what other tests should we attempt? To answer this, we should list all the indexings of array `r` that appear in the method—they are `r[i]` and `r[i - 1]`—and we should predict the range of values that the index expressions, `i` and `i - 1`, might have. Remember that *the values must fall in the range, 0 to `r.length - 1`*. Now, do they?

A bit of thought lets us invent this test,

```
int[] test1 = new int[10];
test0[0] = 3;
exchange(test0, 0);
```

which generates an exception, because `i - 1` has a value that is invalid for array `r`. Another test case,

```
int[] test1 = new int[10];
test0[9] = 3;
exchange(test0, 10);
```

shows that the attempted indexing, `r[i]`, leads to an error.

The tests make clear that parameter `i` must be greater than zero and less than the length of the array. We modify the method to read,

```
public void exchange(int[] r, int i)
{ if ( i > 0 && i < r.length )
    { int temp = r[i];
      r[i] = r[i - 1];
      r[i - 1] = temp;
    }
  else { ... announce there is a problem ... }
}
```

There is, alas, one more test that exposes an error that goes beyond index-value calculation:

```
int[] test2 = null;
exchange(test2, 1);
```

This invocation of `exchange` leads to an exception when the array argument is referenced. If we are uncertain that we can validate that all the method's invocations are with proper arrays, then we must add one more test:

```

public void exchange(int[] r, int i)
{ if ( r != null && i > 0 && i < r.length )
    { int temp = r[i];
      r[i] = r[i - 1];
      r[i - 1] = temp;
    }
  else { ... announce there is a problem ... }
}

```

Testing the values of array indexes becomes harder still when loops are used to examine an array's elements, because it is crucial that the loop starts and terminates appropriately. Consider again the first loop we saw in this chapter, which attempts to select the largest number in an array:

```

double high_score = score[0];
for ( int i = 1; i <= 5; i = i + 1 )
    { if ( score[i] > high_score )
      { high_score = score[i]; }
    }
System.out.println(high_score);

```

We note that the only array indexing is `score[i]`, and we readily see that the range of values denoted by index `i` is 0 to 5. *But this range is sensible only if we are certain that the length of array `score` is at least 6—indeed, it should equal 6.* It is better to use `score.length` in the loop's termination test:

```

double high_score = score[0];
for ( int i = 1; i < score.length; i = i + 1 )
    { if ( score[i] > high_score )
      { high_score = score[i]; }
    }
System.out.println(high_score);

```

This ensures that the loop correctly examines all the elements of the array to select the high score.

The general strategy for testing a component that uses arrays goes as follows:

1. *Validate that every array-typed variable, `r`, is indeed assigned an array object as its value;*
2. *For every indexing expression, `r[e]`, calculate the range of values to which `e` might evaluate, testing these values and especially 0 and `r.length`.*

8.12 Summary

Here are the main points to remember from this chapter:

New Constructions

- *one-dimensional array* (from Figure 1):

```
int num_candidates = 4;
int[] votes = new int[num_candidates];
...
votes[v] = votes[v] + 1;
```

- *array initialization statement*:

```
int[] r = {1, 2, 4, 8, 16, 32};
```

- *array-length attribute*:

```
int[] r = new int[6];
r[0] = 1;
for ( int i = 1; i < r.length; i = i + 1 )
    { r[i] = r[i - 1] * 2; }
```

- *two-dimensional array* (from Figure 13):

```
private PuzzlePiece[][] board;
...
board = new PuzzlePiece[size][size];
..
board[size - 1 - row][size - 1 - col] = p;
```

- *public static final variable* (from Figure 9):

```
public static final int QUEEN = 12;
```

New Terminology

- *array*: an object that holds a collection of values, called *elements*, of the same data type (e.g., a collection of integers or a collection of `JPanel` objects). The elements are named or *indexed* by nonnegative integers. (See the above examples.)
- *one-dimensional array*: an array whose collection is a “sequence” of values, that is, each element is named by a single integer index, e.g., `votes[2]` names the third integer value in the sequence of values held by `votes`.

- *database*: a large collection of data values that must be maintained by means of insertions, retrievals, and deletions of the data values.
- *key*: the identity code used to retrieve a data value saved in a database.
- *final variable*: a variable whose value cannot be changed after it is initialized.
- *two-dimensional arrays*: “an array of arrays” that is typically drawn as a matrix or grid. Each element is named by a pair of integer indexes, e.g., `election[i][j]`. The first index is the *row* index, and the second index is the *column* index, where the terms refer to the depiction of the array as a matrix.

Points to Remember

- In Java, an array is an object that must be constructed with the `new` keyword, e.g., `int[] r = new int[6]`, which creates an array that can hold 6 integers, indexed from 0 to 5.
- Individual array elements are indexed by expressions and are used like ordinary variables, e.g., `r[i + 1] = 2 * r[i]`.
- The elements of an array can be (the addresses of) objects as well, e.g., `Card[] deck = new Card[52]` is an array that can hold 52 `Card` objects. *When the array is constructed, it holds no objects—all elements have value null.* The objects held in the array must be explicitly constructed and assigned to the array’s elements, e.g., `deck[0] = new Card(Card.HEARTS, Card.QUEEN)`.
- Arrays can be constructed with multiple dimensions—a one-dimensional array is a sequence of elements; a two-dimensional array is a matrix—an array of arrays. The rows of a two-dimensional array can have different lengths.

8.13 Programming Projects

1. Extend the simplistic vote-counting application in Figure 1 in the following ways:
 - (a) Each candidate has a name, an address, and an age. The application reads this information first, saves it in objects, and uses the information to count votes, which are now submitted by typing the candidates’ names.
 - (b) The election becomes a national election in 3 regions. Make the application display the total vote counts for each region as well as the total vote counts for each candidate.

2. The classic algorithm for calculating the prime numbers in the range $2..n$ is due to Eratosthenes:

```
initialize the set of primes, P, to be all integers in 2..n;
for ( i = 2; 2*i <= n; i = i+1 )
    { remove from P all multiples of i };
print the contents of P;
```

Implement this algorithm by modelling P as an array of booleans: `boolean[] P = new boolean[n + 1]`. (Hint: `P[i] == false` means that i is definitely not a prime.)

3. With the help of arrays, we can improve the output views for bar graphs, point graphs, and pie charts from the Programming Projects in Chapter 5. Reprogram each of the following.

- (a) Here is the new specification for class `BarGraphWriter`:

<code>class BarGraphWriter</code>	helps a user draw bar graphs
Constructor	
<code>BarGraphWriter(int x_pos, int y_pos, int x_length, int y_height, String top_label)</code>	draw the x- and y-axes of the graph. The pair, <code>x_pos</code> , <code>y_pos</code> , state the coordinates on the window where the two axes begin. The x-axis extends from <code>x_pos</code> , <code>y_pos</code> to the right for <code>x_length</code> pixels; the y-axis extends upwards from <code>x_pos</code> , <code>y_pos</code> for <code>y_height</code> pixels. The label placed at the top of the y-axis is <code>top_label</code> . (The label placed at the bottom of the y-axis is always 0.)
Method	
<code>setBar(String label, int height, Color c)</code>	draws a new bar in the graph, to the right of the bars already drawn, where the label underneath the bar is <code>label</code> , the height of the bar, in pixels, is <code>height</code> , and the bar's color is <code>c</code> .

- (b) Here is the new specification for class `PointGraphWriter`:

<code>class PointGraphWriter</code>	helps a user draw a graph of plotted points
Constructor	
<code>PointGraphWriter(int x_pos, int y_pos, int axis_length, String x_label, String y_label)</code>	draw the the vertical and horizontal axes of the graph, such that the intersection point of the axes lies at position <code>x_pos</code> , <code>y_pos</code> . Each axis has the length, <code>axis_length</code> pixels. The beginning labels of both the x- and y-axes are 0; the label at the top of the y-axis is <code>y_label</code> , and the label at the end of the x-axis is <code>x_label</code> .
Method	
<code>setPoint(int x_increment, int y_height)</code>	plot another point of the graph, so that its x-position is <code>x_increment</code> pixels to the right of the last point plotted, and its y-position is <code>y_height</code> on the y-axis.

(c) Here is the revised specification for `class PieChartWriter`:

<code>class PieChartWriter</code>	helps a user draw a pie chart
Method	
<code>setSlice(String label, int amount, Color c)</code>	add a new “slice” to the chart, such that <code>amount</code> indicates the amount of the slice, and <code>c</code> is the slice’s color. The <code>label</code> is printed to the right of the pie, and it is printed in the color, <code>c</code> .

4. Another form of coding words is by means of a *transposition code*, which encodes a message by transposing its letters. The simplest form of transposition code works as follows:

- (a) A string, `s`, is read; say that `s` has length `n`.
- (b) The smallest square matrix that can hold `n` characters is created.
- (c) The characters in `s` are copied one by one into the columns of the matrix.
- (d) The output is the rows of the matrix.

For example, the input string, `abcdefghijklmn`, would be stored into a 4-by-4 matrix as follows:

```
a e i m
b f j n
```

```

c g k x
d h l x

```

The output would be the words, *aeim*, *bfjn*, *cgkx*, *dhlx*.

Write an application that implements this algorithm. Next, write an application that decodes words produced by the algorithm.

5. To continue the development of the preceding Project, here is a slightly more sophisticated transposition code, based on a numeric key:
 - (a) An integer “key” of *m* distinct digits is read, and a string, *s* of length *n* is read.
 - (b) The smallest matrix with *m* columns that can hold *n* characters is created.
 - (c) The characters in *s* are copied one by one into the columns of the matrix.
 - (d) The digits in the key are sorted, and the columns of the array are accordingly rearranged.
 - (e) The output is the rows of the matrix.

For example, for input key 421 and *abcdefghijklmn*, a 5-by-3 matrix would be built:

```

4 2 1
-----
a f k
b g l
c h m
d i n
e j x

```

Since 421 “sorted” is 124, the columns are rearranged to appear:

```

1 2 4
-----
k f a
l g b
m h c
n i d
x j e

```

and the output are the words, *kfa*, *lgb*, *mhc*, *nid*, and *xje*.

Write programs to encode and decode messages with this algorithm.

6. For each of the following entities, design a class that models the entity. Most of the entity's attributes are listed; feel free to add more. Design appropriate methods.
 - (a) a library book: the book's name, its author, its catalog (id) number, and the id number of the person (if anyone) who has borrowed the book, the due date for the book to be returned, and the number of times the book has been borrowed.
 - (b) a patron's library information: the patron's name, address, id number, and the catalog numbers of all books currently loaned to the person (maximum of 6).
 - (c) an appointment record: the appointment's date, time of day, and topic
 - (d) an inventory record: the name of a sales item, its id number, its wholesale price, its retail price, and the quantity in stock.
 - (e) a purchase record: the id number of the purchaser, the (id numbers of the) items and quantities ordered of each, and the means of payment/
 - (f) a purchaser (customer) record: the id number, name, and address of a customer, the id numbers of the customer's outstanding orders, and the customer's purchase history for the past 12 months
 - (g) an email message: the address of its sender, the address of its receiver, the message's subject, and its body (text)
7. Use the classes you defined in the previous exercise plus `class Database` from Figure 4 to build the following applications:
 - (a) A library application, which maintains a database for the library's books and a database for the library's borrowers. Both databases must be used when books are borrowed and returned.
 - (b) A business accounting application, which uses databases of inventory records, purchases, and purchasers. The databases are used when customers purchase items.
 - (c) An email postal service, which allows multiple users to login, send, and receive email messages to/from one another.
8. Here are the rules for a simple card game: A player tries to obtain two cards that total the highest possible score, where a card's "score" is its count. (For simplicity, ace is worth 1, 2 is worth 2, ..., king is worth 13.) The dealer gives each player 2 cards. A player can surrender at most one card and accept a third card as a replacement. Then, all players must reveal their hands.

Write an application that lets a computerized dealer and two computerized players play this game. (Make the computerized player smart enough that it surrenders a card that has count 6 or less.)

Next, modify the application so that one human player plays against one computerized player.

Finally, modify the application so that two human players play against each other; only the dealer is computerized.

9. Here are the rules for playing the card game, “21”: A player tries to collect cards whose total score is 21 or as close as possible. The player with the highest score not exceeding 21 wins. (If the player’s score is higher than 21, the player is “busted” and loses.) A card’s has a point value based on its count (e.g., a four of clubs is valued 4), but face cards are valued 10, and we will say that an ace is valued 11.

Initially, each player receives two cards from the dealer. Then, each player can request additional cards, one at a time, from the dealer. After all the players have received desired additional cards, the players reveal their hands.

Write an application that lets a computerized dealer, a human player, and a computerized player play 21. The computerized player should be smart enough to request additional cards as long as the player’s total score is 16 or less.

10. Revise the “21” card game as follows:
 - (a) The dealer is also a player; therefore, the dealer must deal herself a hand.
 - (b) An ace has a value of either 1 or 11, based on the discretion of the player who holds the card.
 - (c) In some casinos, a dealer deals from two decks of cards. Alter the application to deal alternately from two decks.
 - (d) If a player’s first two cards have identical counts (e.g., two eights or two queens), the player can “split” the cards into two distinct hands and continue with two hands.
 - (e) The game lets the players play multiple rounds. In particular, this means that the deck of cards must be “reshuffled” with the cards not in the players’ hands when the deck is emptied of cards in the middle of a round.
11. Write an application that plays tic-tac-toe (noughts and crosses) with the user.
12. Here is a standard memory game, known as “Concentration” or “Husker Du”: A matrix is filled with pairs of letters, and the letters are covered. Next, two players take turns trying to discover the letter pairs: A player uncovers two letters; if the letters match, the letters remain uncovered, the player scores one

point, and she is allowed to uncover two more letters. If the letters fail to match, they are recovered and the next player tries. The players take turns until all the letter pairs are uncovered. The player with the most points wins.

Build a computerized version of this game.

13. Write an application that lets two human players play checkers.
14. Write an animation that lets the computer play the game of “Life.” The game goes as follows: the user specifies the size of game board, a matrix, and also gives the starting positions of where some pebbles (“cells”) live. Every second, the computer updates the board, creating and removing cells, according to the following rules:
 - an empty board position that is surrounded by exactly three cells gets a cell placed on it. (The new cell “comes to life.”)
 - a board position occupied by a cell retains the cell if the position was surrounded by exactly 2 other cells. (Otherwise the cell disappears—“dies”—due to “loneliness” or “overcrowding.”)

Here is an example of two seconds of the animation on a 5-by-5 board, where an X denotes a cell and . denotes an empty space:

```
.X.XX      ...XX      ...XX
X...X      ....X      ....X
XXX.. => X.X.. => ...X.
..X.X      X.X..      ..XX.
X...X      ...X.      .....
```

15. Write an appointments manager program. The program stores and retrieves appointments that are listed by date and hour of day for one full week. (For simplicity, assume that at most one appointment can be scheduled per hour.) Include input commands for inserting appointments, listing the appointments for a given day, deleting appointments, and printing appointments.
16. Write an application that reserves seats in an airplane based on input requests in the following format:
 - number of seats desired (should be seated in the same row, next to one another, if possible)
 - first class or economy
 - aisle or window seat (if two or more seats requested, one seat should meet this preference)

17. Write a program that plays the card game, “War”: There are two players; one is human, the other is computerized. Here are the rules: each player gets a hand of 10 cards. The players play 10 rounds; each round goes as follows:
 - (a) A player places one of the cards from her hand face up on the table.
 - (b) The other player does the same.
 - (c) The player whose card has the larger value of the two cards takes all the cards on the table and places them in her “winnings pile.” (Note: the definition of “value” is given below. A “winnings pile” is a stack of cards that is not used any more in the game. Each player keeps her own winnings pile.) The winning player must start the next round.
 - (d) If the two cards on the table have the same value (and this is called a “War”) , then all the cards on the table remain there for the next round. The player who started this round must start the next round.

After all ten rounds are played, the winner is the player with more cards in her winnings pile.

Here is the definition of “value”: regardless of suit, 2 has the lowest value, then 3, then 4, etc., then 9, then 10, then jack, then queen, then king, then ace.

There is a minimal amount of strategy that a player uses to win at War. Your strategy for the computerized player must be at least this smart: (1) If the computerized player plays the first card of a round, then any card remaining in the computerized player’s hand can be played. (2) If the computerized player plays the second card of a round, then the computerized player plays a card in its hand whose value is greater than or equal to the value of the card that the human just played. If the computerized player has no such card, then it plays any card in its hand.

18. Choose another card game of your choosing, e.g., “Hearts” or “Crazy Eights” and model it as a computer game. Or, chose a game that uses dice and implement it as a computer game.
19. Write an application that performs bin packing: The input consists of a sequence of “packages” whose sizes are coded as nonnegative integers along with a sequence of “bins” whose capacities are are coded also by an integer. (For simplicity, we assume that all bins have the same capacity.) The program assigns each package to a bin such that no bin’s capacity is exceeded. The objective is to use the minimum number of bins to hold all the packages. Attempt these implementations:
 - (a) Smallest packages first: The packages are sorted by size and smallest packages are used first.

- (b) Largest packages first: The packages are sorted by size and largest packages are used first.
- (c) Random filling: The packages are used in the order they appear in the input.

(Hint: read the Supplement section on sorting.)

After you have implemented all three programs, perform case studies to determine when one strategy performs better than another. This problem is famous because there is no efficient algorithm for best filling the bins.

8.14 Beyond the Basics

8.14.1 *Sorting*

8.14.2 *Searching*

8.14.3 *Time-Complexity Measures*

8.14.4 *Divide-and-Conquer Algorithms*

8.14.5 *Formal Description of Arrays*

These optional sections expand upon the concepts presented in this chapter. In particular, we emphasize using arrays to sort collections of numbers and efficiently search for numbers in a sorted collection.

8.14.1 **Sorting**

When an array is used as a database, where elements are fetched and updated frequently, there is a distinct advantage to ordering the elements by their keys—it becomes far easier to locate an element. The process of ordering an array’s elements is called *sorting*.

Algorithms for sorting have a rich history, and we cannot do justice here. Instead, we focus upon the development of two traditional sorting methods, *selection sort* and *insertion sort*. To simplify the algorithms that follow, we work with arrays of integers, where we sort the elements so that they are ordered in value from smallest integer to largest. (Of course, we can use the same techniques to sort elements by their key values.)

The idea behind selection sort is simple: Locate the least integer in the array, and move it to the front. Then, find the next least integer, and move it second to the front. Repeat this process until all integers have been selected in order of size. The algorithm that sorts array `r` in this manner goes

```

for ( i = 0; i != r.length; i = i+1 )
  { Find the least element in r within the range
    r[i] to r[r.length-1]; say that it is at r[j].
    Exchange r[i] with r[j].
  }

```

Here is the algorithm in action. Say that we have this array, *r*:

	0	1	2	3	4
<i>r</i>	11	8	-2	7	10

When selection sorting starts, its loop finds the least element in the range *r*[0] .. *r*[4] at index 2 and exchanges the elements at indexes 0 and 2:

	0	1	2	3	4
<i>r</i>	-2	8	11	7	10

The second loop iteration locates the least element in the range *r*[1] .. *r*[4] at index 3, and the elements at indexes 1 and 3 are exchanged:

	0	1	2	3	4
<i>r</i>	-2	7	11	8	10

The algorithm next considers the elements in range *r*[2] .. *r*[4] and so on until it reaches this end result:

	0	1	2	3	4
<i>r</i>	-2	7	8	10	11

Figure 15 shows the method.

There is more than one way to sort an array; a second classic approach, called insertion sort, rearranges elements the way most people sort a hand of playing cards: Start with the first card (element), then take the second card (element) and insert it either before or after the first card, so that the two cards are in order; then take the third card and insert it in its proper position so that the three cards are ordered, and so on. Eventually, all the cards are inserted where they belong in the ordering.

The algorithm based on this idea is simply stated as:

```

for ( i=1; i < r.length; i = i+1 )
  { Insert r[i] in its proper place within the already sorted prefix,
    r[0]..r[i-1].
  }

```

Figure 8.17: selection sort

```

/** selectionSort sorts the elements of its array parameter
 * @param r - the array to be sorted */
public void selectionSort(int[] r)
{ for ( int i = 0; i != r.length; i = i+1 )
    // invariant: subarray r[0]..r[i-1] is sorted
    { int j = findLeast(r, i, r.length-1); // get index of least element
      int temp = r[i];
      r[i] = r[j];
      r[j] = temp;
    }
}

/** findLeast finds the index of the least element in r[start]..r[end]
 * @param r - the array to be searched
 * @param start - the starting element for the search
 * @param end - the ending element for the search
 * @return the index of the smallest element in r[start]..r[end] */
private int findLeast(int[] r, int start, int end)
{ int least = start;
  for ( int i = start+1; i <= end; i = i+1 )
    // invariant: least is index of least element in range r[start]..r[i-1]
    { if ( r[i] < r[least] ) { least = i; } }
  return least; }

```

If we apply the algorithm to the example array, r , seen above,

	0	1	2	3	4
r	11	8	-2	7	10

we see that the algorithm first inserts the 8 where it belongs with respect to the 11:

	0	1	2	3	4
r	8	11	-2	7	10

This makes the prefix, $r[0]..r[1]$, correctly sorted. Next, the -2 must be inserted in its proper place with respect to the sorted prefix:

	0	1	2	3	4
r	-2	8	11	7	10

To make room for -2 at its proper position, $r[0]$, the two elements, 8 and 11, must be shifted one position to the right. Now, $r[0]..r[2]$ is correctly sorted. The last two elements are inserted similarly.

Figure 8.18: insertion sort

```

/** insertionSort sorts the elements of its array parameter
 * @param r - the array to be sorted */
public static void insertionSort(int[] r)
{ for ( int i = 1; i < r.length; i = i+1 )
    // invariant: prefix r[0]..r[i-1] is sorted
    { int v = r[i]; // v is the next element to insert into the prefix
      int j = i;
      while ( j != 0 && r[j-1] > v )
          // invariants:
          // (i) the original prefix, r[0]..r[i-1],
          //      is now arranged as r[0]..r[j-1], r[j+1]..r[i];
          // (ii) all of r[j+1]..r[i] are greater than v
          { r[j] = r[j-1];
            j = j-1;
          }
      r[j] = v;
    }
}

```

Figure 16 show the insertion sorting method. The method's most delicate step is searching the sorted prefix to find a space for v —the while-loop searches from right to left, shifting values one by one, until it encounters a value that is not larger than v . At all iterations, position $r[j]$ is reserved for v ; when the iterations stop, v is inserted at $r[j]$.

Exercises

1. Try selection sort and insertion sort on these arrays: $\{4, 3, 2, 2\}$; $\{1, 2, 3, 4\}$; $\{1\}$; $\{ \}$ (the array of length 0).
2. Explain which of the two sorting methods might finish faster when the array to be sorted is already or nearly sorted; when the array's elements are badly out of order.
3. Explain why the for-loop in method `selectionSort` iterates one more time than it truly needs.
4. Why is the test expression, $j \neq 0$, required in the while-loop in method `insertionSort`?
5. Another sorting technique is *bubble sort*: over and over, compare pairs of adjacent elements and exchange them if the one on the right is less than the one

on the left. In this way, the smaller elements move like “bubbles” to the left (“top”) of the array. The algorithm goes:

```

boolean did_exchanges = true;
while ( did_exchanges )
    { did_exchanges = false;
      for ( int i = 1; i < r.length; i = i+1 )
          { If r[i] < r[i-1],
            then exchange them and assign did_exchanges = true.
          }
    }

```

Program this sorting method.

8.14.2 Searching

Once an array is sorted, it becomes simpler to locate an element within it—rather than examining items one by one, from left to right, we can start searching in the middle, at approximately where the item might appear in the sorted collection. (This is what we do when we search for a word in a dictionary.) A standard searching algorithm, called *binary search*, exploits this idea.

Given a sorted array of integers, *r*, we wish to determine where a value, *item*, lives in *r*. We start searching in the middle of *r*; if *item* is not exactly the middle element, we compare what we found to it: If *item* is less than the middle element, then we next search the lower half of the array; if *item* is greater than the element, we search the upper half of the array. We repeat this strategy until *item* is found or the range of search narrows to nothing, which means that *item* is not present.

The algorithm goes

```

Set searching = true.
Set the lower bound of the search to be 0 and the upper bound of the
search to be the last index of array, r.
while ( searching && lower bound <= upper bound )
    { index = (lower bound + upper bound) / 2;
      if ( item == r[index] ) { found the item---set searching = false; }
      else if ( item < r[index] ) { reset upper bound = index-1; }
      else { reset lower bound = index+1; }
    }

```

Figure 17 shows the method, which is a standard example of the searching pattern of iteration.

If we searched for the item 10 in the sorted array *r* seen in the examples in the previous section, the first iteration of the loop in `binarySearch` gives us this

Figure 8.19: binary search

```

/** binarySearch searches for an item in a sorted array
 * @param r - the array to be searched
 * @param item - the desired item in array r
 * @return the index where item resides in r; if item is not
 * found, then return -1 */
public int binarySearch(int[] r, int item)
{ int lower = 0;
  int upper = r.length - 1;
  int index = -1;
  boolean searching = true;
  while ( searching && lower <= upper )
    // (1) searching == true implies item is in range r[lower]..r[upper],
    //     if it exists in r at all.
    // (2) searching == false implies that r[index] == item.
    { index = (lower + upper) / 2;
      if ( r[index] == item )
        { searching = false; }
      else if ( r[index] < item )
        { lower = index + 1; }
      else { upper = index - 1; }
    }
  if ( searching )
    { index = -1; } // implies lower > upper, hence item not in r
  return index;
}

```

configuration:

	0	1	2	3	4	int lower == 0
r	-2	7	8	10	11	int upper == 4
			^			int index == 2

The search starts exactly in the middle, and the loop examines `r[2]` to see if it is 10. It is not, and since 10 is larger than 8, the value found at `r[2]`, the search is revised as follows:

	0	1	2	3	4	int lower == 3
r	-2	7	8	10	11	int upper == 4
				^		int index == 3

Searching the upper half of the array, which is just two elements, moves the search to `r[3]`, which locates the desired item.

Notice that a linear search, that is,

```
int index = 0;
```

```

boolean searching = true;
while ( searching && index != r.length )
    { if ( r[index] == item )
        { searching = false; }
      else { index = index + 1; }
    }

```

would examine four elements of the array to locate element 10. The binary search examined just two. Binary search's speedup for larger arrays is enormous and is discussed in the next section.

Binary search is a well-known programming challenge because it is easy to formulate incorrect versions. (Although the loop in Figure 17 is small, its invariant suggests that a lot of thought is embedded within it.) Also, small adjustments lead to fascinating variations. Here is a clever reformulation, due to N. Wirth:

```

public int binarySearch(int[] r, int item)
{ int lower = 0;
  int upper = r.length-1;
  int index = -1;
  while ( lower <= upper )
    // (1) lower != upper+2 implies that item is in range
    //     r[lower]..r[upper], if it exists in r at all
    // (2) lower == upper+2 implies that r[index] == item
    { index = (lower + upper) / 2;
      if ( item <= r[index] )
        { upper = index - 1; };
      if ( item >= r[index] )
        { lower = index + 1; };
    }
  if ( lower != upper+2 )
    { index = -1; }
  return index;
}

```

This algorithm merges variable `searching` in Figure 17 with the lower and upper bounds of the search so that the loop's test becomes simpler. This alters the loop invariant so that the discovery of `item` is indicated by `lower == upper+2`.

Both searching algorithms must terminate, because the expression, `upper-lower` decreases in value at each iteration, ensuring that the loop test will eventually go false.

Exercises

1. Use the binary search method in Figure 17 on the sorted array, {1, 2, 2, 4, 6}: Ask the method to search for 6; for 2; for 3. Write execution traces for these searches.

2. Here is a binary search method due to R. Howell:

```
public int search(int[] r, int item)
{ int answer = -1;
  if ( r.length > 0 )
    { int lower = 0;
      int upper = r.length;
      while ( upper - lower > 1 )
        // item is in r[lower]..r[upper-1], if it is in r
        { int index = (lower + upper) / 2;
          if ( r[index] > item )
            { upper = index; }
          else { lower = index; }
        }
      if ( r[lower]== item ) { answer = lower; }
    }
  return answer;
}
```

Explain why the invariant and the termination of the loop ensure that the method returns a correct answer. Explain why the loop must terminate. (This is not trivial because the loop makes one extra iteration before it quits.)

8.14.3 Time-Complexity Measures

The previous section stated that binary search computes its answer far faster than does linear search. We can state how much faster by doing a form of counting analysis on the respective algorithms. The analysis will introduce us to a standard method for computing the *time complexity* of an algorithm. We then apply the method to analyze the time complexity of selection sort and insertion sort.

To analyze a searching algorithm, one counts the number of elements the algorithm must examine to find an item (or to report failure). Consider linear search: If array *r* has, say, *N* elements, we know in the very worst case that a linear search must examine all *N* elements to find the desired item or report failure. Of course, over many randomly generated test cases, the number of elements examined will average to about *N*/2, but in any case, the number of examinations is directly proportional to the the array' length, and we say that the algorithm has performance of *order N* (also known as *linear*) time complexity.

For example, a linear search of an array of 256 elements will require at most 256 examinations and 128 examinations on the average.

Because it halves its range of search at each element examination, binary search does significantly better than linear time complexity: For example, a worst case binary search of a 256-element array makes one examination in the middle of the 256

elements, then one examination in the middle of the remaining 128 elements, then one examination in the middle of the remaining 64 elements, and so on—a maximum of only 9 examinations are required!

We can state this behavior more precisely with a recursive definition. Let $E(N)$ stand for the number of examinations binary search makes (in worst case) to find an item in an array of N elements.

Here is the exact number of examinations binary search does:

$$\begin{aligned} E(N) &= 1 + E(N/2), \text{ for } N > 1 \\ E(1) &= 1 \end{aligned}$$

The first equation states that a search of an array with multiple elements requires an examination of the array's middle element, and assuming the desired item is not found in the middle, a subsequent search of an array of half the length. An array of length 1 requires just one examination to terminate the search.

To simplify our analysis of the above equations, say the array's length is a power of 2, that is, $N = 2^M$, for some positive M . (For example, for $N = 256$, M is 8. Of course, not all arrays have a length that is exactly a power of 2, but we can always pretend that an array is “padded” with extra elements to make its length a power of 2.)

Here are the equations again:

$$\begin{aligned} E(2^M) &= 1 + E(2^{M-1}), \text{ for } M > 0 \\ E(2^0) &= 1 \end{aligned}$$

After several calculations with this definition (and a proof by induction—see the Exercises), we can convince ourselves that

$$E(2^M) = M + 1$$

a remarkably small answer!

We say that the binary search algorithm has *order $\log N$* (or *logarithmic*) time complexity. (Recall that $\log N$, or more precisely, $\log_2 N$, is N 's *base-2 logarithm*, that is, the exponent, M , such that 2^M equals N . For example, $\log 256$ is 8, and $\log 100$ falls between 6 and 7.) Because we started our analysis with the assumption that $N = 2^M$, we conclude that

$$E(N) = (\log N) + 1$$

which shows that binary search has logarithmic time complexity.

It takes only a little experimentation to see, for large values of N , that $\log N$ is significantly less than N itself. This is reflected in the speed of execution of binary search, which behaves significantly better than linear search for large-sized arrays.

Analysis of Sorting Algorithms

Of course, binary search assumes that the array it searches is sorted, so we should calculate as well the time complexity of the sorting algorithms we studied. The two factors in the performance of a sorting algorithm are (i) the number of comparisons of array elements, and (ii) the number of exchanges of array elements. If either of these measures is high, this slows the algorithm.

Consider selection sort first (Figure 15); it locates and exchanges the smallest element, then the next smallest element, and so on. For an array of length N , it uses $N-1$ comparisons to find the smallest element, $N-2$ comparisons to find the next smallest element, and so on. The total number of comparisons is therefore

$$(N-1) + (N-2) + \dots \text{downto} \dots + 2 + 1$$

From number theory (and an induction proof), we can discover that this sequence totals

$$\frac{N * (N - 1)}{2}$$

that is, $(1/2)N^2 - (1/2)N$. When N has a substantial positive value, only the N^2 factor matters, so we say that the algorithm *has order N^2 (quadratic) time complexity*.

Algorithms with quadratic time complexity perform significantly slower than logarithmic and linear algorithms, and this slowness can be annoying when N is very large (e.g., for N equals 100, N^2 is 10,000).

It is easy to see that selection sort does exactly $N-1$ exchanges of elements—a linear time complexity—so the exchanges are not the costly part of the algorithm.

Next, we consider insertion sort (Figure 16); recall that it shifts elements, one by one, from right to left into their proper places. In worst case, insertion sort encounters an array whose elements are in reverse order. In this case, the algorithm's first iteration makes one comparison and one exchange; the second iteration makes two comparisons and two exchanges; and so on. The total number of comparisons and exchanges are the same, namely,

$$1 + 2 + \dots + (N-2) + N-1$$

This is the same sequence we encountered in our analysis of selection sort, so we conclude that insertion sort also has quadratic time complexity.

Although selection sort's time complexity is stable across all possible permutations of arrays to be sorted, insertion sort executes much faster when it is given an almost completely sorted array to sort. This is because insertion sort shifts elements only when they are out of order. For example, if insertion sort is given an array of length $N+1$ where only one element is out of order, it will take only order N (linear) time to shift the element to its proper position. For this reason, insertion sort is preferred for sorting almost-sorted arrays.

In contrast, insertion sort does badly at exchanging elements when sorting an arbitrary array—it makes order N^2 exchanges, whereas selection sort limits its exchanges to at worst order N . Therefore, selection sort is preferred if there is substantial difficulty in moving elements of the array. (But this is not normally the case for Java arrays, because the elements of a Java array are either primitive values, like numbers, or addresses of objects. These values are easy to exchange.)

Exercises

1. To get intuition about time complexities, calculate the values of N , $5*N$, $\log N$, N^2 , and $(1/2)(N^2) - (1/2)N$ for each of the following values of N : 4; 64; 128; 512; 1024; 16384.

Then, reexamine the time complexities of the searching and sorting algorithms and describe how the algorithms would behave on arrays of size N , for the above values of N . (To give some perspective to the analysis, pretend that your computer is very slow and takes 0.1 seconds to perform a comparison or exchange operation.)

2. Modify `class Database` in Figure 3 so that its `insert` method sorts the `base` array after a new record is added. (Warning—watch for `null` values in the array!) Because the contents of `base` are already sorted when a new element is inserted, does this simplify the sorting process? What form of sorting is better for this application—selection sort or insertion sort?

Next, modify `locationOf` so that it uses binary search.

3. Perform time-complexity analyses of the following methods:
 - (a) For Figure 1, Chapter 7, measure the time complexity of `summation(N)`, depending on the value of N . Count the number of assignments the method makes.
 - (b) For Figure 3, Chapter 7, measure the time complexity of `findChar(c, s)`, depending on the lengths of string `s`. Count the number of `charAt` operations the method makes.
 - (c) For Figure 13, measure the time complexity of `paint`, depending on the `size` of array that must be painted. Count the number of invocations of `paintPiece`.
4. Our time-complexity analyses are a bit simplistic: a precise time-complexity analysis would count every operation that a computer's processor makes, that is, every arithmetic operation, every comparison operation, every variable reference, every assignment, every method invocation, every method return, etc. Perform such a detailed analysis for the algorithms in the previous Exercise; for

linear search; for binary search. Are your answers significantly different than before?

5. Use mathematical induction to prove that $E(2^M) = M + 1$, for all nonnegative values of M . This requires that you prove these two claims:

- *basis step*: $E(2^0) = 0 + 1$
- *induction step*: Assume that $E(2^i) = i + 1$ holds true. Use this to prove $E(2^{i+1}) = (i + 1) + 1$.

6. Use mathematical induction to prove that $(N-1) + (N-2) + \dots \text{downto} \dots + 2 + 1$ equals $(1/2)(N^2) - (1/2)N$, for all values of N that are 2 or larger. This requires that you prove these two claims:

- *basis step*: $(2-1) + (2-2) + \dots \text{downto} \dots + 2 + 1$ equals $(1/2)(2^2) - (1/2)2$. (Hint: read the sequence, $1 + \dots \text{downto} \dots + 1$ as being just the one-element sequence, 1.)
- *induction step*: Assume that $(i-1) + (i-2) + \dots \text{downto} \dots + 2 + 1$ equals $(1/2)(i^2) - (1/2)i$. Use this to prove $((i+1)-1) + ((i+1)-2) + \dots \text{downto} \dots + 2 + 1$ equals $(1/2)((i + 1)^2) - (1/2)(i + 1)$.

8.14.4 Divide-and-Conquer Algorithms

In the previous section, we saw that the binary search algorithm has a significantly better time complexity than the linear search algorithm. The time measurement for binary search was expressed by a recursive definition, which suggests that a recursion might be a factor in binary search's performance. This is indeed the case—binary search is an example of a style of recursion known as *divide and conquer*, which we study in this section.

First, Figure 18 shows binary search written in recursive style. To search an entire array, `a`, for a value, `v`, the method is invoked as `binarySearch(a, v, 0, a.length-1)`. The method clearly shows that, at each recursive invocation, the segment searched is divided in half. Eventually, the desired item is found or the segment is divided into nothing.

The method in the Figure is an example of a *divide-and-conquer* algorithm, so called because the algorithm divides its argument, the array, into smaller segments at each invocation. The divide-and-conquer pattern uses recursion correctly, because each recursive invocation operates on parameters (the array segments) that grow smaller until they reach a stopping value (size 0).

Figure 8.20: binary search by recursion

```

/** binarySearch searches for an item within a segment of a sorted array
 * @param r - the array to be searched
 * @param item - the desired item
 * @param lower - the lower bound of the segment
 * @param upper - the upper bound of the segment
 * @return the index where item resides in r[lower]..r[upper];
 * return -1, if item is not present in the segment of r */
public int binarySearch(int[] r, int item, int lower, int upper)
{ int answer = -1;
  if ( lower <= upper )
    { int index = (lower + upper) / 2;
      if ( r[index] == item )
        { answer = index; }
      else if ( r[index] < item )
        { answer = binarySearch(r, item, index + 1, upper); }
      else { answer = binarySearch(r, item, lower, index - 1); }
    }
  return answer;
}

```

Merge sort

Sorting can be accomplished with a divide-and-conquer algorithm, which proceeds as follows: To sort a complete array, *r*,

1. Divide the array into two smaller segments, call them *s1* and *s2*.
2. Sort *s1*.
3. Sort *s2*.
4. *Merge* the two sorted segments to form the completely sorted array.

The merge step goes as follows: Say that you have a deck of cards you wish to sort. You divide the deck in half and somehow sort each half into its own pile. You merge the two piles by playing this “game”: Turn over the top card from each pile. (The top cards represent the lowest-valued cards of the two piles.) Take the lower-valued of the two cards, form a new pile with it, and turn over the next card from the pile from which you took the lower-valued card. Repeat the game until all the cards are moved into the third pile, which will be the entire deck, sorted.

Figure 19 shows the method based on this algorithm, called *merge sort*. Like the recursive version of binary search, `mergeSort` is first invoked as `mergeSort(a, 0,`

Figure 8.21: merge sort

```

/** mergeSort builds a sorted array segment
 * @param r - the array
 * @param lower - the lower bound of the segment to be sorted
 * @param upper - the upper bound of the segment to be sorted
 * @return a sorted array whose elements are those in r[lower]..r[upper] */
public int[] mergeSort(int[] r, int lower, int upper)
{ int[] answer;
  if ( lower > upper ) // is it an empty segment?
    { answer = new int[0]; }
  else if ( lower == upper ) // is it a segment of just one element?
    { answer = new int[1];
      answer[0] = r[lower];
    }
  else // it is a segment of length 2 or more, so divide and conquer:
    { int middle = (lower + upper) / 2;
      int[] s1 = mergeSort(r, lower, middle);
      int[] s2 = mergeSort(r, middle+1, upper);
      answer = merge(s1, s2);
    }
  return answer;
}

/** merge builds a sorted array by merging its two sorted arguments
 * @param r1 - the first sorted array
 * @param r2 - the second sorted array
 * @return a sorted array whose elements are exactly those of r1 and r2 */
private int[] merge(int[] r1, int[] r2)
{ int length = r1.length + r2.length;
  int[] answer = new int[length];
  int index1 = 0;
  int index2 = 0;
  for ( int i = 0; i != length; i = i+1 )
    // invariant: answer[0]..answer[i-1] is sorted and holds the elements of
    // r1[0]..r1[index1-1] and r2[0]..r2[index2-1]
    { if ( index1 == r1.length
          || ( index2 != r2.length && r2[index2] < r1[index1] ) )
        { answer[i] = r2[index2];
          index2 = index2 + 1;
        }
      else { answer[i] = r1[index1];
            index1 = index1 + 1;
          }
    }
  return answer;
}

```

`a.length-1`) to indicate that all the elements in array `a` should be sorted. The method returns a new array that contains `a`'s elements reordered.

Method `mergeSort` first verifies that the segment of the array it must sort has at least two elements; if it does, the segment is divided in two, the subsegments are sorted, and `merge` combines the two sorted subarrays into the answer.

The time complexity of merge sort is significantly better than the other sorting algorithms seen so far; we consider the number of comparisons the algorithm makes. (The analysis of element exchanges goes the same.)

First, we note that `merge(r1, r2)` makes as many comparisons as there are elements in the shorter of its two array parameters, but it will be convenient to overestimate and state that no more than `r1.length + r2.length` comparisons are ever made.

Next, we define the comparisons made by `mergeSort` on an array of length `N` as the quantity, $C(N)$:

$$\begin{aligned} C(N) &= C(N / 2) + C(N / 2) + N, \quad \text{if } N > 1 \\ C(1) &= 0 \end{aligned}$$

The first equation states that the total comparisons to sort an array of length 2 or more is the sum of the comparisons needed to sort the left segment, the comparisons needed to sort the right segment, and the comparisons needed to merge the two sorted segments. Of course, an array of length 1 requires no comparisons.

Our analysis of these equations goes simpler if we pretend the array's length is a power of 2, that is $N = 2^M$, for some nonnegative `M`:

$$\begin{aligned} C(2^M) &= C(2^{M-1}) + C(2^{M-1}) + 2^M \\ C(2^0) &= 0 \end{aligned}$$

These equations look like the ones discovered in the analysis of binary search. Indeed, if we divide both sides of the first equation by 2^M , we see the pattern in the binary search equation:

$$\frac{C(2^M)}{2^M} = \frac{C(2^{M-1})}{2^{M-1}} + 1$$

As with the binary search equation, we can conclude that

$$\frac{C(2^M)}{2^M} = M$$

When we multiply both sides of the above solution by 2^M , we see that

$$C(2^M) = 2^M * M$$

and since $N = 2^M$, we have that

$$C(N) = N * \log N$$

We say that merge sort has *order* $N \log N$ time complexity. Such algorithms perform almost as well as linear-time algorithms, so our discovery is significant.

Alas, `mergeSort` suffers from a significant flaw: When it sorts an array, it creates additional arrays for merging—this will prove expensive when sorting large arrays. The method in Figure 19 freely created many extra arrays, but if we are careful, we can write a version of `mergeSort` that creates no more than one extra array the same size as the original, unsorted array. For arrays that model large databases, even this might be unacceptable, unfortunately.

Quicksort

A brilliant solution to the extra-array problem was presented by C.A.R. Hoare in the guise of the “quicksort” algorithm. Like merge sort, quicksort uses the divide-and-conquer technique, but it cleverly rebuilds the sorted array segments within the original array: It replaces the merge step, which occurred after the recursive invocations, with a *partitioning* step, which occurs before the recursive invocations.

The idea behind partitioning can be understood this way: Say that you have a deck of unsorted playing cards. You partition the cards by (i) choosing a card at random from the deck and (ii) creating two piles from the remaining cards by placing those cards whose values are less than the chosen card in one pile and placing those cards whose values are greater than the chosen card in the other.

It is a small step from partitioning to sorting: If you sort the cards in each pile, then the entire deck is sorted by just concatenating the piles. This is a classic divide-and-conquer strategy and forms the algorithm for quicksort. Given an array, `r`, whose elements are numbered `r[lower]` to `r[upper]`:

1. Rearrange (partition) `r` into two nonempty subarrays so that there is an index, `m`, such that all the elements in `r[lower]..r[m]` are less than or equal to all elements in `r[m+1]..r[upper]`.
2. Sort the partition `r[lower]..r[m]`.
3. Sort the partition `r[m+1]..r[upper]`.

The end result must be the array entirely sorted.

Figure 20 gives the `quickSort` method, which is invoked as `quickSort(r, 0, r.length-1)`, for array `r`. The hard work is done by `partition(r, lower, upper)`, which partitions the elements in the range `r[lower]..r[upper]` into two groups. The method uses the element at `r[lower]` as the “pivot” value for partitioning as it scans the elements from left to right, moving those values less than the pivot to the left side of the subarray. Once all the elements are scanned, the ones less than the pivot form the first partition, and the ones greater-or-equal to the pivot form the second partition.

Figure 8.22: quicksort

```

/** quickSort sorts an array within the indicated bounds
 * @param r - the array to be sorted
 * @param lower - the lower bound of the elements to be sorted
 * @param upper - the upper bound of the elements to be sorted */
public void quickSort(int[] r, int lower, int upper)
{
    if ( lower < upper )
    {
        int middle = partition(r, lower, upper);
        quickSort(r, lower, middle);
        quickSort(r, middle+1, upper);
    }
}

/** partition rearranges an array's elements into two nonempty partitions
 * @param r - an array of length 2 or more
 * @param lower - the lower bound of the elements to be partitioned
 * @param upper - the upper bound of the elements to be partitioned
 * @return the index, m, such that all elements in the nonempty partition,
 * r[lower]..r[m], are <= all elements in the nonempty partition,
 * r[m+1]..r[upper] */
private int partition(int[] r, int lower, int upper)
{
    int v = r[lower]; // the 'pivot' value used to make the partitions
    int m = lower - 1; // marks the right end of the first partition
    int i = lower + 1; // marks the right end of the second partition
    while ( i <= upper )
    {
        // invariant: (i) all of r[lower]..r[m] are < v
        //              (ii) all of r[m+1]..r[i-1] are >= v,
        //              and the partition is nonempty
        {
            if ( r[i] < v )
            {
                // insert r[i] at the end of the first partition
                // by exchanging it with r[m+1]:
                m = m + 1;
                int temp = r[i];
                r[i] = r[m];
                r[m] = temp;
            }
            i = i + 1;
        }
    }
    if ( m == lower - 1 ) // after all the work, is the first partition empty?
    {
        m = m + 1; // then place r[lower], which is v, into it
    }
    return m;
}

```

It is essential that both of the partitions created by `partition` are nonempty. For this reason, a conditional statement after the while-loop asks whether the partition of elements less than the pivot is empty. If it is, this means the pivot is the smallest value in the subarray, no exchanges were made, and the pivot remains at `r[lower]`. In this case, the pivot value itself becomes the first partition.

We can see partitioning at work in an example. Say that we invoke `quickSort(r, 0, 6)`, which immediately invokes `partition(r, 0, 6)` for the array `r` shown below. The variables in `partition` are initialized as follows:

	0	1	2	3	4	5	6	int v == 5
r	5	8	4	1	7	3	9	int i == 0
m	i							int m == -1

We position `m` and `i` under the array to indicate the variables' values. The pivot value is `r[0]`—5. Values less than the pivot will be moved to the left; the other values will move to the right.

Within `partition`'s while-loop, `i` moves right, searching for a value less than 5; it finds one at element 2. This causes `r[2]` to be moved to the end of the first partition—it is exchanged with `r[m+1]`, and both `m` and `i` are incremented. Here is the resulting situation:

	0	1	2	3	4	5	6
r	4	8	5	1	7	3	9
m			i				

A check of the loop invariant verifies that the elements in the range `r[0]` to `r[m]` are less than the pivot, and the values in the range `r[m+1]` to `r[i-1]` are greater-or-equal to the pivot.

Immediately, `i` has located another value to be moved to the first partition. An exchange is undertaken between `r[1]` and `r[3]`, producing the following:

	0	1	2	3	4	5	6
r	4	1	5	8	7	3	9
m				i			

The process continues; one more exchange is made. When the method finishes, here is the partitioned array:

	0	1	2	3	4	5	6
r	4	1	3	8	7	5	9
			m				

Since `m` is 2, the partitions are `r[0]..r[2]` and `r[3]..r[6]`.

Once a partitioning step is complete, `quickSort` recursively sorts the two partitions. This causes each subarray, `r[0]..r[2]` and `r[3]..r[6]`, to be partitioned and

recursively sorted. (That is, the invocation, `quicksort(r, 0, 2)` invokes `partition(r, 0, 2)`, and `quicksort(r, 3, 6)` invokes `partition(r, 3, 6)`, and so on.) Eventually, partitions of size 1 are reached, stopping the recursive invocations.

For `quicksort` to perform at its best, the `partition` method must generate partitions that are equally sized. In such a case, each recursive invocation of `quicksort` operates on an array segment half the size of the previous one, and the time complexity is the same as mergesort—order $N \log N$. But there is no guarantee that `partition` will always break an array into two equally sized partitions—if the pivot value, `v`, is the largest (or smallest) value in an array segment of size N , then `partition` creates one partition of size 1 and one of size $N-1$. For example, if array `r` was already sorted

	0	1	2	3	4	5	6
r	1	3	4	5	7	8	9

and we invoked `partition(r, 0, 6)`, then `partition` would choose the pivot to be 1 and would create the partitions `r[0]` and `r[1]..r[6]`. The subsequent recursive invocation to `quicksort(r, 1, 6)` causes another such partitioning: `r[1]` and `r[2]..r[6]`. This behavior repeats for all the recursive calls.

In a case as the above, `quicksort` degenerates into a variation of insertion sort and operates with order N^2 time complexity. Obviously, if `quicksort` is applied often to sorted or almost-sorted arrays, then `partition` should choose a pivot value from the middle of the array rather than from the end (see the Exercises below). Studies of randomly generated arrays shows that quicksort behaves, on the average, with order $N \log N$ time complexity.

Exercises

1. To gain understanding, apply iterative `binarySearch` in Figure 17 and recursive `binarySearch` in Figure 18 to locate the value, 9, in the array, `int[] r = {-2, 5, 8, 9, 11, 14}`. Write execution traces.
2. Write an execution trace of `mergeSort` applied to the array `int[] r = {5, 8, -2, 11, 9}`.
3. Rewrite `mergeSort` in Figure 19 so that it does *not* create multiple new arrays. Instead, use this variant:

```

/** mergeSort sorts a segment of an array, r
 * @param r - the array whose elements must be sorted
 * @param scratch - an extra array that is the same length as r
 * @param lower - the lower bound of the segment to be sorted
 * @param upper - the upper bound of the segment to be sorted */
public void mergeSort(int[] r, int[] scratch, int lower, int upper)

```

```

{ ...
  mergeSort(r, scratch, lower, middle);
  mergeSort(r, scratch, middle+1, upper);
  ...
}

```

The method is initially invoked as follows: `mergeSort(a, new int[a.length], 0, a.length-1)`.

4. Finish the execution traces for the example in this section that uses `quickSort`.
5. Write a `partition` algorithm for use by `quickSort` that chooses a pivot value in the middle of the subarray to be partitioned.
6. Because `quickSort`'s `partition` method is sensitive to the pivot value it chooses for partitioning, a standard improvement is to revise `partition` so that, when it partitions a subarray of size 3 or larger, `partition` chooses 3 array elements from the subarray and picks the *median* (the “middle value”) as the pivot. Revise `partition` in this way.

8.14.5 Formal Description of Arrays

Arrays cause us to augment the syntax of data types, object construction, and variables to our Java subset. First, for every data type, `T`, `T[]` is the data type of “T-arrays.” The precise syntax of data types now reads

```

TYPE ::= PRIMITIVE_TYPE | REFERENCE_TYPE
PRIMITIVE_TYPE ::= boolean | ... | int | ...
REFERENCE_TYPE ::= IDENTIFIER | TYPE[]

```

The syntax allows one- and multi-dimensional array types, e.g., `int[]` as well as `GregorianCalendar[][]`.

Array Constructors

Array variables are declared like ordinary variables; within the initialization statement, we can construct an array object explicitly, e.g., `int[] r = new int[4]` or by means of a set-like initialization expression, e.g., `int[] r = {1, 2, 4, 8}`. Here is a syntax definition that includes the two formats:

```

DECLARATION ::= TYPE IDENTIFIER [[ = INITIAL_EXPRESSION ]]? ;

INITIAL_EXPRESSION ::= EXPRESSION
                    | { [[ INITIAL_EXPRESSION_LIST ]]? }
INITIAL_EXPRESSION_LIST ::= INITIAL_EXPRESSION [[ , INITIAL_EXPRESSION ]]*

```

(Recall that `[[E]]`? means that a phrase, `E`, is optional and `[[E]]`* means that phrase `E` can be repeated zero or more times.)

The syntax, `{ [[INITIAL_EXPRESSION_LIST]]?` } defines the set notation for array object construction. The syntax makes clear that multi-dimensional arrays can be constructed from nested set expressions:

```
double[[ ] d = { {0.1, 0.2}, {}, {2.3, 2.4, 2.6}};
```

This constructs an array with three rows of varying lengths and assigns it to `d`.

An array object can be constructed by a set expression only within an initialization statement. The compiler verifies that the dimensions of the set expression and the data types of the individual elements in the set expression are compatible with the data type listed with the variable declared. Only elements of primitive type can be listed.

An array constructed with the `new` keyword is defined by means of an `OBJECT_CONSTRUCTION` of the form, `new ARRAY_ELEMENT_TYPE DIMENSIONS`:

```
EXPRESSION ::= ... | STATEMENT_EXPRESSION
STATEMENT_EXPRESSION ::= OBJECT_CONSTRUCTION | ...

OBJECT_CONSTRUCTION ::= ... | new ARRAY_ELEMENT_TYPE DIMENSIONS

ARRAY_ELEMENT_TYPE ::= PRIMITIVE_TYPE | IDENTIFIER
DIMENSIONS ::= [ EXPRESSION ] [[ [ EXPRESSION ] ]]* [[ [ ] ]]*
```

That is, `ARRAY_ELEMENT_TYPE`, the data type of the array's individual elements, is listed first, followed by the all the array's dimensions. *The quantity of at least the first dimension must be given*; the quantities of the dimensions that follow can be omitted. For example, `new int[4] []` constructs a two-dimensional array object with 4 rows and an unspecified number of columns per row, and `new int[4] [3]` constructs a two-dimensional array object with 4 rows and 3 columns. The phrase, `new int []`, is unacceptable.

An array construction, `new ARRAY_ELEMENT_TYPE DIMENSIONS`, is type checked to validate that all expressions embedded in the `DIMENSIONS` have data types that are subtypes of `int`. The compiler calculates the data type of the phrase as `ARRAY_ELEMENT_TYPE` followed by the number of dimensions in `DIMENSIONS`.

The execution semantics of an array construction goes as follows: For simplicity, consider just a one-dimensional object, `new ARRAY_ELEMENT_TYPE[EXPRESSION]`:

1. `EXPRESSION` is computed to an integer value, `v`. (If `v` is negative, an exception results.)
2. An object is constructed with `v` distinct elements. The data type, `ARRAY_ELEMENT_TYPE[v]`, is saved within the object. Say that the object has storage address, `a`.

3. If `ARRAY_ELEMENT_TYPE` is a numeric type, the elements in the object are initialized to 0. If it is `boolean`, the elements are initialized to `false`. Otherwise, the elements are initialized to `null`.
4. The object's address, `a`, is returned as the result.

When an array variable is initialized with an array object, as in `int[] r = new int[3]`, data-type checking and execution semantics proceed the same as with any other variable initialization: The data type of the right-hand-side expression must be a subtype of the left-hand-side type, and the address of the constructed object is assigned to the left-hand-side variable's cell.

References and Assignments

Elements of arrays are referenced with bracket notation, e.g., `r[i + 1] = r[0]`. Here is the syntax for expressions and assignments extended to arrays:

```

ASSIGNMENT ::= VARIABLE = EXPRESSION
VARIABLE ::= IDENTIFIER | ... | RECEIVER [ EXPRESSION ]
EXPRESSION ::= ... | VARIABLE

RECEIVER ::= IDENTIFIER | ... | RECEIVER [ EXPRESSION ]

```

Recall that `VARIABLE` phrases must compute to addresses of storage cells (to which are assigned values); `RECEIVERS` must compute to addresses of objects that can receive messages; and `EXPRESSIONS` must compute to values that can be stored in cells.

The syntax allows an array to use multiple indexes, e.g., `d[3][2] = 4.5`. More importantly, since an array is an object, it is a “receiver” of messages that ask for indexings, e.g., `r[0]` sends a “message” to the object named `r`, asking it to index itself at element 0 and return the value in that cell.

For an indexing expression, `RECEIVER[EXPRESSION]`, the compiler verifies that the data type of `EXPRESSION` is a subtype of `int`, and it verifies that the data type of `RECEIVER` is an array type. When the indexing expression appears as a `VARIABLE` on the left-hand side of an assignment, the compiler verifies, as usual, that the data type of the right-hand side expression is a subtype of the left-hand side variable's type.

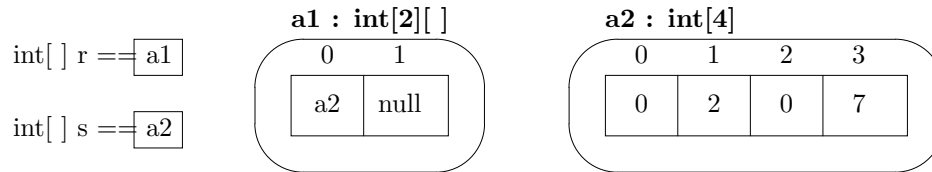
When used as a `VARIABLE` on the left-hand side of an assignment, the semantics of the phrase, `RECEIVER[EXPRESSION]`, computes an *address*:

1. `RECEIVER` is computed to its result, which will be an address, `a`, of an array object.
2. `EXPRESSION` is computed to its result, which must be an integer, `v`.
3. If `v` is nonnegative and is less than the length of the array at address `a`, then the address, `a[v]`, is returned as the result; otherwise, an exception is thrown.

For example, say that `r` holds the address, `a1`, of an array object. Then, the assignment, `r[1 + 2] = 4`, causes `r[1 + 2]` to compute to the address, `a1[3]`, and inserts 4 into the cell at that address.

When the phrase, `RECEIVER[EXPRESSION]`, is used as a `RECEIVER` or as an `EXPRESSION`, then of course the addressed cell is dereferenced and *the value in the cell is returned as the result*. For example, `System.out.println(r[3])` prints the value found in the cell addressed by `r[3]`.

Here is a more complex example. For the arrays,



the assignment, `r[0][2] = r[0][s[1] + 1]`, would execute these steps:

1. The variable part, `r[0][2]`, computes to an address:
 - (a) The leftmost `r` computes to `a1`.
 - (b) `r[0]` computes to the address, `a1[0]`, but this phrase is used as a receiver (of the message, `[2]`), so `a1[0]` is dereferenced, producing `a2`.
 - (c) The address, `a2[2]`, is formed as the address of the left-hand side variable. This is the target of the assignment.
2. The right-hand side, `r[0][s[1] + 1]`, computes to an integer:
 - (a) Since `r` has value `a1`, and `r[0]` is the receiver of the message, `[s[1] + 1]`, the address, `a1[0]` is dereferenced to the value `a2`.
 - (b) The expression, `s[1] + 1`, computes to 3, because `s` has value `a2`, `s[1]` appears as an expression, hence the address `a2[1]` is dereferenced to 2 and 1 is added to it.
 - (c) Because `r[0]` computed to `a2` and `s[1] + 1` computed to 3, the address `a2[3]` is formed. Since this appears as an expression, it is dereferenced to produce 7.
3. 7 is assigned to address `a2[2]`.

The above description of array assignment omits an important subtlety that is specific to the Java language: When this assignment is executed,

```
RECEIVER [ EXPRESSION1 ] = EXPRESSSION2
```

The complete listing of execution steps goes as follows:

1. RECEIVER is computed to its value, which will be an address, *a*, of an array object. The run-time type information is extracted from *a*; say that it is `element_type[size]`.
2. EXPRESSION1 is computed to an integer, *v*. If *v* is nonnegative and less than `size`, then the address, `a[v]`, is formed as the target of the assignment.
3. EXPRESSION2 is computed to its result, *w*.
4. *This is the surprising, additional step:* If *w* is not a primitive value, then it is an address of an object—the run-time type, *t*, is fetched from the object at address *w* and is compared to `element_type` to verify that *t* is a subtype of `element_type`.
5. If the types are compatible, *w* is assigned to the cell at `a[v]`; otherwise, an exception is thrown.

In most programming languages, Step 4 is not required, because the type checking already performed by the compiler suffices. But the additional type checking at execution is forced upon Java because of Java's subtyping laws for object (reference) types.

To see this, here is an example. Perhaps we write this method:

```
public void assignPanel(JPanel[] r, JPanel f)
{ r[0] = f; }
```

The Java compiler examines the method and judges it acceptable. Next, we write this class:

```
public class MyPanel extends JPanel
{ public MyPanel() { }

    public void paintComponent(Graphics g) { }

    public void newMethod() { }
}
```

This class is also acceptable to the Java compiler. But now, we play a trick:

```
MyPanel[] panels = new MyPanels[2];
JPanel x = new JPanel();
assignPanel(panels, x);
panels[0].newMethod();
```

Because `MyPanel` is a subtype of `JPanel`, `panels` is an acceptable actual parameter to `assignPanel`, which apparently assigns a `JPanel` object into an array that is meant to hold only `MyPanel` objects. If the assignment is allowed to proceed, then disaster

strikes at `panels[0].newMethod()`, which sends a message to an object that has no `newMethod`.

This is the reason why every assignment to an array element must be type checked at execution even though it was type checked previously by the Java compiler.