CHAPTER

6

SINGLE-DIMENSIONAL ARRAYS

Objectives

- To describe why arrays are necessary in programming (§6.1).
- To declare array reference variables and create arrays (§§6.2.1–6.2.2).
- To obtain array size using `arrayRefVar.length` and know default values in an array (§6.2.3).
- To access array elements using indexed variables (§6.2.4).
- To declare, create, and initialize an array using an array initializer (§6.2.5).
- To program common array operations (displaying arrays, summing all elements, finding the minimum and maximum elements, random shuffling, and shifting elements) (§6.2.6).
- To simplify programming using the for-each loops (§6.2.7).
- To apply arrays in application development (`LottoNumbers`, `DeckOfCards`) (§§6.3–6.4).
- To copy contents from one array to another (§6.5).
- To develop and invoke methods with array arguments and return values (§§6.6–6.8).
- To define a method with a variable-length argument list (§6.9).
- To search elements using the linear (§6.10.1) or binary (§6.10.2) search algorithm.
- To sort an array using the selection sort approach (§6.11.1).
- To sort an array using the insertion sort approach (§6.11.2).
- To use the methods in the `java.util.Arrays` class (§6.12).

6.1 Introduction

Key Point

A single array variable can reference a large collection of data.

problem

why array?

Often you will have to store a large number of values during the execution of a program. Suppose, for instance, that you need to read 100 numbers, compute their average, and find out how many numbers are above the average. Your program first reads the numbers and computes their average, then compares each number with the average to determine whether it is above the average. In order to accomplish this task, the numbers must all be stored in variables. You have to declare 100 variables and repeatedly write almost identical code 100 times. Writing a program this way would be impractical. So, how do you solve this problem?

what is array?

An efficient, organized approach is needed. Java and most other high-level languages provide a data structure, the *array*, which stores a fixed-size sequential collection of elements of the same type. In the present case, you can store all 100 numbers into an array and access them through a single array variable. The solution may look like this:

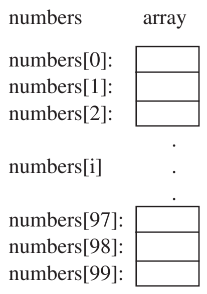
create array

store number in array

get average

above average?

```
 1  public class AnalyzeNumbers {
 2    public static void main(String[] args) {
 3      final int NUMBER_OF_ELEMENTS = 100;
 4      double[] numbers = new double[NUMBER_OF_ELEMENTS];
 5      double sum = 0;
 6
 7      java.util.Scanner input = new java.util.Scanner(System.in);
 8      for (int i = 0; i < NUMBER_OF_ELEMENTS; i++) {
 9        System.out.print("Enter a new number: ");
10        numbers[i] = input.nextDouble();
11        sum += numbers[i];
12      }
13
14      double average = sum / NUMBER_OF_ELEMENTS;
15
16      int count = 0; // The number of elements above average
17      for (int i = 0; i < NUMBER_OF_ELEMENTS; i++)
18        if (numbers[i] > average)
19          count++;
20
21      System.out.println("Average is " + average);
22      System.out.println("Number of elements above the average "
23        + count);
24    }
25  }
```



The program creates an array of **100** elements in line 4, stores numbers into the array in line 10, adds each number to **sum** in line 11, and obtains the average in line 14. It then compares each number in the array with the average to count the number of values above the average (lines 16–19).

This chapter introduces single-dimensional arrays. The next chapter will introduce two-dimensional and multidimensional arrays.

6.2 Array Basics

Key Point

Once an array is created, its size is fixed. An array reference variable is used to access the elements in an array using an index.

index

An array is used to store a collection of data, but often we find it more useful to think of an array as a collection of variables of the same type. Instead of declaring individual variables, such as **number0**, **number1**, . . . , and **number99**, you declare one array variable such as **numbers** and use **numbers[0]**, **numbers[1]**, . . . , and **numbers[99]** to represent individual variables.

This section introduces how to declare array variables, create arrays, and process arrays using indexed variables.

6.2.1 Declaring Array Variables

To use an array in a program, you must declare a variable to reference the array and specify the array's *element type*. Here is the syntax for declaring an array variable:

element type

```
elementType[] arrayRefVar;
```

The **elementType** can be any data type, and all elements in the array will have the same data type. For example, the following code declares a variable **myList** that references an array of double elements.

```
double[] myList;
```

Note

You can also use **elementType arrayRefVar[]** to declare an array variable. This style comes from the C language and was adopted in Java to accommodate C programmers. The style **elementType[] arrayRefVar** is preferred.

preferred syntax

6.2.2 Creating Arrays

Unlike declarations for primitive data type variables, the declaration of an array variable does not allocate any space in memory for the array. It creates only a storage location for the reference to an array. If a variable does not contain a reference to an array, the value of the variable is **null**. You cannot assign elements to an array unless it has already been created. After an array variable is declared, you can create an array by using the **new** operator with the following syntax:

null

```
arrayRefVar = new elementType[arraySize];
```

new operator

This statement does two things: (1) it creates an array using **new elementType[arraySize]**; (2) it assigns the reference of the newly created array to the variable **arrayRefVar**.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement as:

```
elementType[] arrayRefVar = new elementType[arraySize];
```

or

```
elementType arrayRefVar[] = new elementType[arraySize];
```

Here is an example of such a statement:

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

This statement declares an array variable, **myList**, creates an array of ten elements of **double** type, and assigns its reference to **myList**. To assign values to the elements, use the syntax:

```
arrayRefVar[index] = value;
```

For example, the following code initializes the array.

```
myList[0] = 5.6;
myList[1] = 4.5;
myList[2] = 3.3;
myList[3] = 13.2;
```

```
myList[4] = 4.0;
myList[5] = 34.33;
myList[6] = 34.0;
myList[7] = 45.45;
myList[8] = 99.993;
myList[9] = 11123;
```

This array is illustrated in Figure 6.1.

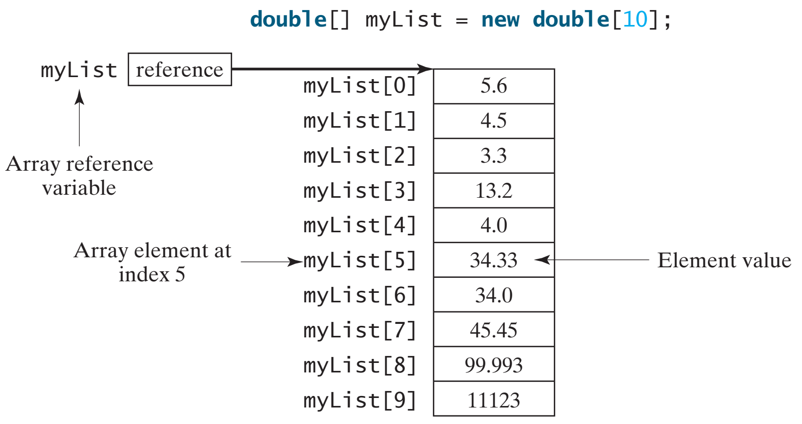


FIGURE 6.1 The array `myList` has ten elements of `double` type and `int` indices from `0` to `9`.

array vs. array variable

Note

An array variable that appears to hold an array actually contains a reference to that array. Strictly speaking, an array variable and an array are different, but most of the time the distinction can be ignored. Thus it is all right to say, for simplicity, that `myList` is an array, instead of stating, at greater length, that `myList` is a variable that contains a reference to an array of ten double elements.

6.2.3 Array Size and Default Values

array length

When space for an array is allocated, the array size must be given, specifying the number of elements that can be stored in it. The size of an array cannot be changed after the array is created. Size can be obtained using `arrayRefVar.length`. For example, `myList.length` is `10`.

default values

When an array is created, its elements are assigned the default value of `0` for the numeric primitive data types, `\u0000` for `char` types, and `false` for `boolean` types.

6.2.4 Array Indexed Variables

0 based

The array elements are accessed through the index. Array indices are `0` based; that is, they range from `0` to `arrayRefVar.length-1`. In the example in Figure 6.1, `myList` holds ten `double` values, and the indices are from `0` to `9`.

indexed variable

Each element in the array is represented using the following syntax, known as an *indexed variable*:

```
arrayRefVar[index];
```

For example, `myList[9]` represents the last element in the array `myList`.

Caution

Some programming languages use parentheses to reference an array element, as in `myList(9)`, but Java uses brackets, as in `myList[9]`.

After an array is created, an indexed variable can be used in the same way as a regular variable. For example, the following code adds the values in **myList[0]** and **myList[1]** to **myList[2]**.

```
myList[2] = myList[0] + myList[1];
```

The following loop assigns **0** to **myList[0]**, **1** to **myList[1]**, . . . , and **9** to **myList[9]**:

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

6.2.5 Array Initializers

Java has a shorthand notation, known as the *array initializer*, which combines the declaration, creation, and initialization of an array in one statement using the following syntax:

array initializer

```
elementType[] arrayRefVar = {value0, value1, ..., valuek};
```

For example, the statement

```
double[] myList = {1.9, 2.9, 3.4, 3.5};
```

declares, creates, and initializes the array **myList** with four elements, which is equivalent to the following statements:

```
double[] myList = new double[4];
myList[0] = 1.9;
myList[1] = 2.9;
myList[2] = 3.4;
myList[3] = 3.5;
```

Caution

The **new** operator is not used in the array-initializer syntax. Using an array initializer, you have to declare, create, and initialize the array all in one statement. Splitting it would cause a syntax error. Thus, the next statement is wrong:

```
double[] myList;
myList = {1.9, 2.9, 3.4, 3.5};
```

6.2.6 Processing Arrays

When processing array elements, you will often use a **for** loop—for two reasons:

- All of the elements in an array are of the same type. They are evenly processed in the same fashion repeatedly using a loop.
- Since the size of the array is known, it is natural to use a **for** loop.

Assume the array is created as follows:

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

The following are some examples of processing arrays.

1. *Initializing arrays with input values:* The following loop initializes the array **myList** with user input values.

```
java.util.Scanner input = new java.util.Scanner(System.in);
System.out.print("Enter " + myList.length + " values: ");
for (int i = 0; i < myList.length; i++)
  myList[i] = input.nextDouble();
```

2. *Initializing arrays with random values:* The following loop initializes the array `myList` with random values between `0.0` and `100.0`, but less than `100.0`.

```
for (int i = 0; i < myList.length; i++) {
  myList[i] = Math.random() * 100;
}
```

3. *Displaying arrays:* To print an array, you have to print each element in the array using a loop like the following:

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

print character array

Tip

For an array of the `char[]` type, it can be printed using one print statement. For example, the following code displays `Dallas`:

```
char[] city = {'D', 'a', 'l', 'l', 'a', 's'};
System.out.println(city);
```

4. *Summing all elements:* Use a variable named `total` to store the sum. Initially `total` is `0`. Add each element in the array to `total` using a loop like this:

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

5. *Finding the largest element:* Use a variable named `max` to store the largest element. Initially `max` is `myList[0]`. To find the largest element in the array `myList`, compare each element with `max`, and update `max` if the element is greater than `max`.

```
double max = myList[0];
for (int i = 1; i < myList.length; i++) {
  if (myList[i] > max) max = myList[i];
}
```

6. *Finding the smallest index of the largest element:* Often you need to locate the largest element in an array. If an array has more than one largest element, find the smallest index of such an element. Suppose the array `myList` is {`1`, `5`, `3`, `4`, `5`, `5`}. The largest element is `5` and the smallest index for `5` is `1`. Use a variable named `max` to store the largest element and a variable named `indexOfMax` to denote the index of the largest element. Initially `max` is `myList[0]`, and `indexOfMax` is `0`. Compare each element in `myList` with `max`, and update `max` and `indexOfMax` if the element is greater than `max`.

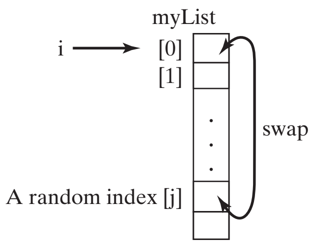
```
double max = myList[0];
int indexOfMax = 0;
for (int i = 1; i < myList.length; i++) {
  if (myList[i] > max) {
    max = myList[i];
    indexOfMax = i;
  }
}
```

VideoNote
Random shuffling

7. *Random shuffling:* In many applications, you need to randomly reorder the elements in an array. This is called *shuffling*. To accomplish this, for each element

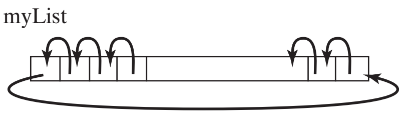
myList[i], randomly generate an index j and swap myList[i] with myList[j], as follows:

```
for (int i = 0; i < myList.length; i++) {
  // Generate an index j randomly
  int j = (int) (Math.random()
    * mylist.length);

  // Swap myList[i] with myList[j]
  double temp = myList[i];
  myList[i] = myList[j]
  myList[j] = temp;
}
```



8. *Shifting elements:* Sometimes you need to shift the elements left or right. Here is an example of shifting the elements one position to the left and filling the last element with the first element:

```
double temp = myList[0]; // Retain the first element

// Shift elements left
for (int i = 1; i < myList.length; i++) {
  myList[i - 1] = myList[i];
}

// Move the first element to fill in the last position
myList[myList.length - 1] = temp;
```



9. *Simplifying coding:* Arrays can be used to greatly simplify coding for certain tasks. For example, suppose you wish to obtain the English name of a given month by its number. If the month names are stored in an array, the month name for a given month can be accessed simply via the index. The following code prompts the user to enter a month number and displays its month name:

```
String[] months = {"January", "February", ..., "December"};
System.out.print("Enter a month number (1 to 12): ");
int monthNumber = input.nextInt();
System.out.println("The month is " + months[monthNumber - 1]);
```

If you didn't use the months array, you would have to determine the month name using a lengthy multi-way if-else statement as follows:

```
if (monthNumber == 1)
  System.out.println("The month is January");
else if (monthNumber == 2)
  System.out.println("The month is February");
...
else
  System.out.println("The month is December");
```

6.2.7 for-each Loops

Java supports a convenient for loop, known as a *for-each loop* or *enhanced for loop*, which enables you to traverse the array sequentially without using an index variable. For example, the following code displays all the elements in the array myList:

```
for (double u: myList) {
  System.out.println(u);
}
```

You can read the code as "for each element u in myList, do the following." Note that the variable, u, must be declared as the same type as the elements in myList.

In general, the syntax for a for-each loop is

```
for (elementType element: arrayRefVar) {
  // Process the element
}
```

You still have to use an index variable if you wish to traverse the array in a different order or change the elements in the array.

ArrayIndexOutOfBounds-Exception

Caution

Accessing an array out of bounds is a common programming error that throws a runtime ArrayIndexOutOfBoundsException. To avoid it, make sure that you do not use an index beyond arrayRefVar.length – 1.

off-by-one error

Programmers often mistakenly reference the first element in an array with index 1, but it should be 0. This is called the *off-by-one error*. Another common error in a loop is using <= where < should be used. For example, the following loop is wrong.

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

The <= should be replaced by <.

Check Point
MyProgrammingLab™

6.1 How do you declare an array reference variable and how do you create an array?

6.2 When is the memory allocated for an array?

6.3 What is the printout of the following code?

```
int x = 30;
int[] numbers = new int[x];
x = 60;
System.out.println("x is " + x);
System.out.println("The size of numbers is " + numbers.length);
```

6.4 Indicate true or false for the following statements:

- Every element in an array has the same type.
- The array size is fixed after an array reference variable is declared.
- The array size is fixed after it is created.
- The elements in an array must be a primitive data type.

6.5 Which of the following statements are valid?

```
int i = new int(30);
double d[] = new double[30];
char[] r = new char(1..30);
int i[] = (3, 4, 3, 2);
float f[] = {2.3, 4.5, 6.6};
char[] c = new char();
```

6.6 How do you access elements in an array? What is an array indexed variable?

6.7 What is the array index type? What is the lowest index? What is the representation of the third element in an array named a?

6.8 Write statements to do the following:

a. Create an array to hold 10 double values.

b. Assign the value 5.5 to the last element in the array.

c. Display the sum of the first two elements.

d. Write a loop that computes the sum of all elements in the array.

e. Write a loop that finds the minimum element in the array.

f. Randomly generate an index and display the element of this index in the array.

g. Use an array initializer to create another array with the initial values `3.5`, `5.5`, `4.52`, and `5.6`.

6.9 What happens when your program attempts to access an array element with an invalid index?

6.10 Identify and fix the errors in the following code:

```
public class Test {
  public static void main(String[] args) {
    double[100] r;

    for (int i = 0; i < r.length(); i++);
      r(i) = Math.random * 100;
  }
}
```

6.11 What is the output of the following code?

```
public class Test {
  public static void main(String[] args) {
    int list[] = {1, 2, 3, 4, 5, 6};
    for (int i = 1; i < list.length; i++)
      list[i] = list[i - 1];

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

6.3 Case Study: Lotto Numbers

The problem is to write a program that checks if all the input numbers cover **1** *to* **99**.

Key Point

VideoNote

Lotto numbers

Each ticket for the Pick-10 lotto has `10` unique numbers ranging from `1` to `99`. Suppose you buy a lot of tickets and like to have them cover all numbers from `1` to `99`. Write a program that reads the ticket numbers from a file and checks whether all numbers are covered. Assume the last number in the file is `0`. Suppose the file contains the numbers

```
80 3 87 62 30 90 10 21 46 27
12 40 83 9 39 88 95 59 20 37
80 40 87 67 31 90 11 24 56 77
11 48 51 42 8 74 1 41 36 53
52 82 16 72 19 70 44 56 29 33
54 64 99 14 23 22 94 79 55 2
60 86 34 4 31 63 84 89 7 78
43 93 97 45 25 38 28 26 85 49
47 65 57 67 73 69 32 71 24 66
92 98 96 77 6 75 17 61 58 13
35 81 18 15 5 68 91 50 76
0
```

Your program should display

```
The tickets cover all numbers
```

Suppose the file contains the numbers

```
11 48 51 42 8 74 1 41 36 53
52 82 16 72 19 70 44 56 29 33
0
```

Your program should display

```
The tickets don't cover all numbers
```

How do you mark a number as covered? You can create an array with **99** **boolean** elements. Each element in the array can be used to mark whether a number is covered. Let the array be **isCovered**. Initially, each element is **false**, as shown in Figure 6.2a. Whenever a number is read, its corresponding element is set to **true**. Suppose the numbers entered are **1**, **2**, **3**, **99**, **0**. When number **1** is read, **isCovered[0]** is set to **true** (see Figure 6.2b). When number **2** is read, **isCovered[2 - 1]** is set to **true** (see Figure 6.2c). When number **3** is read, **isCovered[3 - 1]** is set to **true** (see Figure 6.2d). When number **99** is read, **isCovered[98]** is set to **true** (see Figure 6.2e).

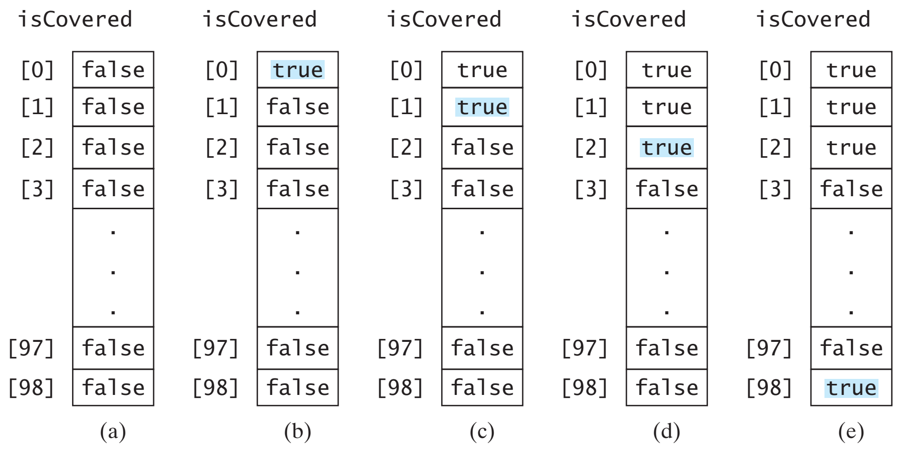

isCovered
[0] [1] [2] [3] [97] [98]
(a) false false false false false false
(b) true false false false false false
(c) true true false false false false
(d) true true true false false false
(e) true true true false false true

FIGURE 6.2 If number **i** appears in a Lotto ticket, **isCovered[i-1]** is set to **true**.

The algorithm for the program can be described as follows:

```
for each number k read from the file,
  mark number k as covered by setting isCovered[k – 1] true;
if every isCovered[i] is true
  The tickets cover all numbers
else
  The tickets don't cover all numbers
```

The complete program is given in Listing 6.1.

LISTING 6.1 LottoNumbers.java

```java
import java.util.Scanner;

public class LottoNumbers {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    boolean[] isCovered = new boolean[99]; // Default is false

    // Read each number and mark its corresponding element covered
    int number = input.nextInt();
    while (number != 0) {
      isCovered[number - 1] = true;
      number = input.nextInt();
    }

```

create and initialize array

read number

mark number covered

read number

```
15      // Check whether all covered
16      boolean allCovered = true; // Assume all covered initially
17      for (int i = 0; i < isCovered.length; i++)
18        if (!isCovered[i]) {
19          allCovered = false; // Find one number not covered
20          break;
21        }
22
23      // Display result
24      if (allCovered)
25        System.out.println("The tickets cover all numbers");
26      else
27        System.out.println("The tickets don't cover all numbers");
28    }
29  }
```

check allCovered?

Suppose you have created a text file named LottoNumbers.txt that contains the input data `2 5 6 5 4 3 23 43 2 0`. You can run the program using the following command:

```
java LottoNumbers < LottoNumbers.txt
```

The program can be traced as follows:

Line#	Representative elements in array isCovered							number	allCovered
	[1]	[2]	[3]	[4]	[5]	[22]	[42]		
6	false	false	false	false	false	false	false		
9								2	
11	true								
12								5	
11				true					
12								6	
11					true				
12								5	
11				true					
12								4	
11			true						
12								3	
11		true							
12								23	
11						true			
12								43	
11							true		
12								2	
11	true								
12								0	
16									true
18(i=0)									false

The program creates an array of **99 boolean** elements and initializes each element to **false** (line 6). It reads the first number from the file (line 9). The program then repeats the following operations in a loop:

- If the number is not zero, set its corresponding value in array **isCovered** to **true** (line 11);
- Read the next number (line 12).

When the input is **0**, the input ends. The program checks whether all numbers are covered in lines 16–21 and displays the result in lines 24–27.

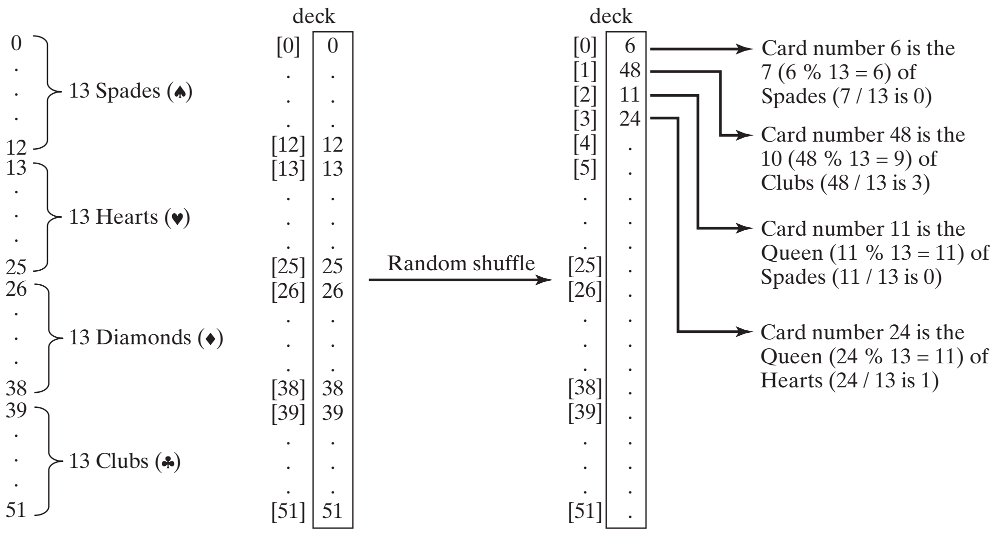
6.4 Case Study: Deck of Cards

Key Point *The problem is to create a program that will randomly select four cards from a deck of cards.*

Say you want to write a program that will pick four cards at random from a deck of **52** cards. All the cards can be represented using an array named **deck**, filled with initial values **0** to **51**, as follows:

```
int[] deck = new int[52];

// Initialize cards
for (int i = 0; i < deck.length; i++)
  deck[i] = i;
```

Card numbers **0** to **12**, **13** to **25**, **26** to **38**, and **39** to **51** represent 13 Spades, 13 Hearts, 13 Diamonds, and 13 Clubs, respectively, as shown in Figure 6.3. **cardNumber / 13** determines the suit of the card and **cardNumber % 13** determines the rank of the card, as shown in Figure 6.4. After shuffling the array **deck**, pick the first four cards from **deck**. The program displays the cards from these four card numbers.



13 Spades (♠)
13 Hearts (♥)
13 Diamonds (♦)
13 Clubs (♣)
deck
Random shuffle
deck
Card number 6 is the 7 (6 % 13 = 6) of Spades (7 / 13 is 0)
Card number 48 is the 10 (48 % 13 = 9) of Clubs (48 / 13 is 3)
Card number 11 is the Queen (11 % 13 = 11) of Spades (11 / 13 is 0)
Card number 24 is the Queen (24 % 13 = 11) of Hearts (24 / 13 is 1)

FIGURE 6.3 52 cards are stored in an array named **deck**.

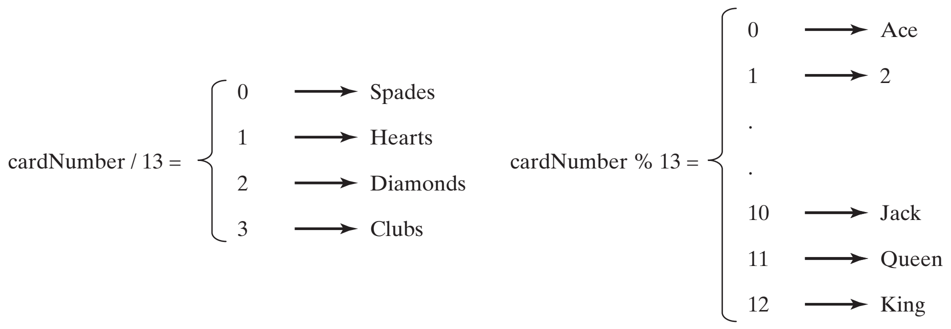


FIGURE 6.4 How `cardNumber` identifies a card's suit and rank number.

Listing 6.2 gives the solution to the problem.

LISTING 6.2 DeckOfCards.java

```java
public class DeckOfCards {
  public static void main(String[] args) {
    int[] deck = new int[52];
    String[] suits = {"Spades", "Hearts", "Diamonds", "Clubs"};
    String[] ranks = {"Ace", "2", "3", "4", "5", "6", "7", "8", "9",
      "10", "Jack", "Queen", "King"};

    // Initialize the cards
    for (int i = 0; i < deck.length; i++)
      deck[i] = i;

    // Shuffle the cards
    for (int i = 0; i < deck.length; i++) {
      // Generate an index randomly
      int index = (int)(Math.random() * deck.length);
      int temp = deck[i];
      deck[i] = deck[index];
      deck[index] = temp;
    }

    // Display the first four cards
    for (int i = 0; i < 4; i++) {
      String suit = suits[deck[i] / 13];
      String rank = ranks[deck[i] % 13];
      System.out.println("Card number " + deck[i] + ": "
        + rank + " of " + suit);
    }
  }
}
```

create array deck
array of strings
array of strings
initialize deck
shuffle deck
suit of a card
rank of a card

```
Card number 6: 7 of Spades
Card number 48: 10 of Clubs
Card number 11: Queen of Spades
Card number 24: Queen of Hearts
```

The program defines an array **suits** for four suits (line 4) and an array **ranks** for 13 cards in a suit (lines 5–6). Each element in these arrays is a string.

The program initializes **deck** with values **0** to **51** in lines 9–10. The **deck** value **0** represents the card Ace of Spades, **1** represents the card 2 of Spades, **13** represents the card Ace of Hearts, and **14** represents the card 2 of Hearts.

Lines 13–19 randomly shuffle the deck. After a deck is shuffled, **deck[i]** contains an arbitrary value. **deck[i] / 13** is **0**, **1**, **2**, or **3**, which determines the suit (line 23). **deck[i] % 13** is a value between **0** and **12**, which determines the rank (line 24). If the **suits** array is not defined, you would have to determine the suit using a lengthy multi-way **if-else** statement as follows:

```
if (deck[i] / 13 == 0)
  System.out.print("suit is Spades");
else if (deck[i] / 13 == 1)
  System.out.print("suit is Hearts");
else if(deck[i] / 13 == 2)
  System.out.print("suit is Diamonds");
else
  System.out.print("suit is Clubs");
```

With **suits = {"Spades", "Hearts", "Diamonds", "Clubs"}** created in an array, **suits[deck / 13]** gives the suit for the **deck**. Using arrays greatly simplifies the solution for this program.

6.5 Copying Arrays

Key Point *To copy the contents of one array into another, you have to copy the array's individual elements into the other array.*

Often, in a program, you need to duplicate an array or a part of an array. In such cases you could attempt to use the assignment statement (=), as follows:

```
list2 = list1;
```

copy reference

garbage collection

However, this statement does not copy the contents of the array referenced by **list1** to **list2**, but instead merely copies the reference value from **list1** to **list2**. After this statement, **list1** and **list2** reference the same array, as shown in Figure 6.5. The array previously referenced by **list2** is no longer referenced; it becomes garbage, which will automatically collected by the Java Virtual Machine (this process is called *garbage collection*).

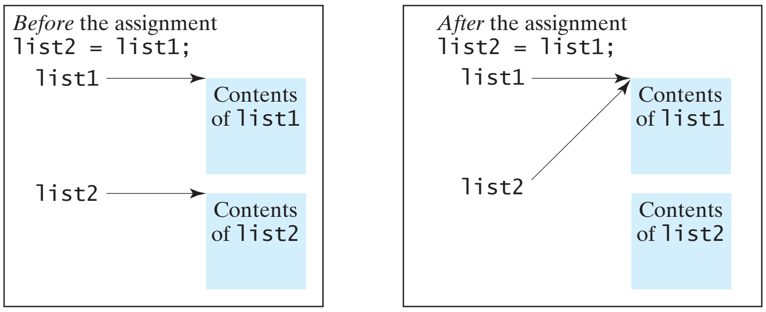


FIGURE 6.5 Before the assignment statement, **list1** and **list2** point to separate memory locations. After the assignment, the reference of the **list1** array is passed to **list2**.

In Java, you can use assignment statements to copy primitive data type variables, but not arrays. Assigning one array variable to another array variable actually copies one reference to another and makes both variables point to the same memory location.

There are three ways to copy arrays:

- Use a loop to copy individual elements one by one.
- Use the static **arraycopy** method in the **System** class.
- Use the **clone** method to copy arrays; this will be introduced in Chapter 15, Abstract Classes and Interfaces.

You can write a loop to copy every element from the source array to the corresponding element in the target array. The following code, for instance, copies **sourceArray** to **targetArray** using a **for** loop.

```
int[] sourceArray = {2, 3, 1, 5, 10};
int[] targetArray = new int[sourceArray.length];
for (int i = 0; i < sourceArray.length; i++) {
  targetArray[i] = sourceArray[i];
}
```

Another approach is to use the **arraycopy** method in the **java.lang.System** class to copy arrays instead of using a loop. The syntax for **arraycopy** is:

arraycopy method

```
arraycopy(sourceArray, src_pos, targetArray, tar_pos, length);
```

The parameters **src_pos** and **tar_pos** indicate the starting positions in **sourceArray** and **targetArray**, respectively. The number of elements copied from **sourceArray** to **targetArray** is indicated by **length**. For example, you can rewrite the loop using the following statement:

```
System.arraycopy(sourceArray, 0, targetArray, 0, sourceArray.length);
```

The **arraycopy** method does not allocate memory space for the target array. The target array must have already been created with its memory space allocated. After the copying takes place, **targetArray** and **sourceArray** have the same content but independent memory locations.

Note

The **arraycopy** method violates the Java naming convention. By convention, this method should be named **arrayCopy** (i.e., with an uppercase C).

Check Point
MyProgrammingLab™

6.12 Use the **arraycopy()** method to copy the following array to a target array **t**:

```
int[] source = {3, 4, 5};
```

6.13 Once an array is created, its size cannot be changed. Does the following code resize the array?

```
int[] myList;
myList = new int[10];
// Sometime later you want to assign a new array to myList
myList = new int[20];
```

6.6 Passing Arrays to Methods

Key Point

When passing an array to a method, the reference of the array is passed to the method.

Just as you can pass primitive type values to methods, you can also pass arrays to methods. For example, the following method displays the elements in an **int** array:

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

You can invoke it by passing an array. For example, the following statement invokes the **printArray** method to display **3**, **1**, **2**, **6**, **4**, and **2**.

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

Note

The preceding statement creates an array using the following syntax:

```
new elementType[]{value0, value1, ..., valuek};
```

anonymous array

There is no explicit reference variable for the array. Such array is called an *anonymous array*.

pass-by-value

Java uses *pass-by-value* to pass arguments to a method. There are important differences between passing the values of variables of primitive data types and passing arrays.

- For an argument of a primitive type, the argument's value is passed.
- For an argument of an array type, the value of the argument is a reference to an array; this reference value is passed to the method. Semantically, it can be best described as *pass-by-sharing*, that is, the array in the method is the same as the array being passed. Thus, if you change the array in the method, you will see the change outside the method.

pass-by-sharing

Take the following code, for example:

```
public class Test {
  public static void main(String[] args) {
    int x = 1; // x represents an int value
    int[] y = new int[10]; // y represents an array of int values

    m(x, y); // Invoke m with arguments x and y

    System.out.println("x is " + x);
    System.out.println("y[0] is " + y[0]);
  }

  public static void m(int number, int[] numbers) {
    number = 1001; // Assign a new value to number
    numbers[0] = 5555; // Assign a new value to numbers[0]
  }
}
```

```
x is 1
y[0] is 5555
```

You may wonder why after **m** is invoked, **x** remains **1**, but **y[0]** become **5555**. This is because **y** and **numbers**, although they are independent variables, reference the same array, as illustrated in Figure 6.6. When **m(x, y)** is invoked, the values of **x** and **y** are passed to **number** and **numbers**. Since **y** contains the reference value to the array, **numbers** now contains the same reference value to the same array.

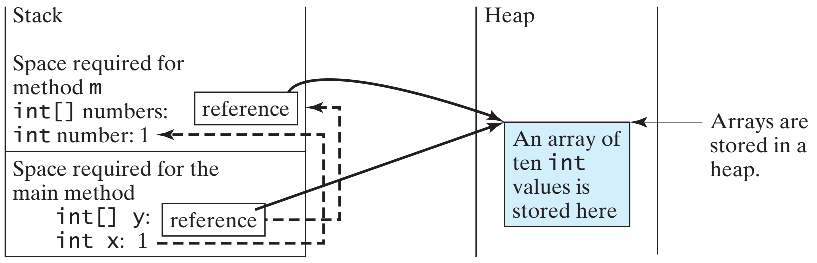

Stack
Space required for method m
int[] numbers:
reference
int number: 1
Space required for the main method
int[] y:
reference
int x: 1
Heap
An array of ten int values is stored here
Arrays are stored in a heap.

FIGURE 6.6 The primitive type value in `x` is passed to `number`, and the reference value in `y` is passed to `numbers`.

Note

Arrays are objects in Java (objects are introduced in Chapter 8). The JVM stores the objects in an area of memory called the *heap*, which is used for dynamic memory allocation. heap

Listing 6.3 gives another program that shows the difference between passing a primitive data type value and an array reference variable to a method.

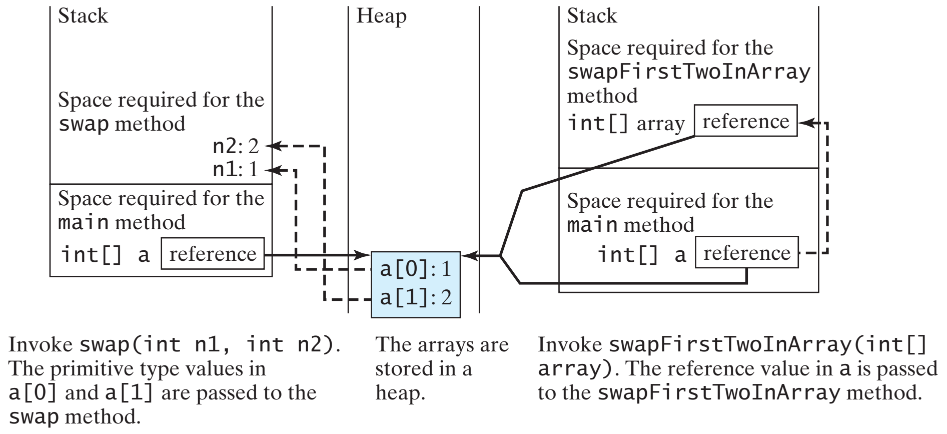
The program contains two methods for swapping elements in an array. The first method, named `swap`, fails to swap two `int` arguments. The second method, named `swapFirstTwoInArray`, successfully swaps the first two elements in the array argument.

LISTING 6.3 TestPassArray.java

```
public class TestPassArray {
  /** Main method */
  public static void main(String[] args) {
    int[] a = {1, 2};

    // Swap elements using the swap method
    System.out.println("Before invoking swap");
    System.out.println("array is {" + a[0] + ", " + a[1] + "}");
    swap(a[0], a[1]);                                         false swap
    System.out.println("After invoking swap");
    System.out.println("array is {" + a[0] + ", " + a[1] + "}");

    // Swap elements using the swapFirstTwoInArray method
    System.out.println("Before invoking swapFirstTwoInArray");
    System.out.println("array is {" + a[0] + ", " + a[1] + "}");
    swapFirstTwoInArray(a);                                   swap array elements
    System.out.println("After invoking swapFirstTwoInArray");
    System.out.println("array is {" + a[0] + ", " + a[1] + "}");
  }

  /** Swap two variables */
  public static void swap(int n1, int n2) {
    int temp = n1;
    n1 = n2;
    n2 = temp;
  }

  /** Swap the first two elements in the array */
  public static void swapFirstTwoInArray(int[] array) {
    int temp = array[0];
    array[0] = array[1];
    array[1] = temp;
  }
}
```

```
Before invoking swap
array is {1, 2}
After invoking swap
array is {1, 2}
Before invoking swapFirstTwoInArray
array is {1, 2}
After invoking swapFirstTwoInArray
array is {2, 1}
```

As shown in Figure 6.7, the two elements are not swapped using the **swap** method. However, they are swapped using the **swapFirstTwoInArray** method. Since the parameters in the **swap** method are primitive type, the values of **a[0]** and **a[1]** are passed to **n1** and **n2** inside the method when invoking **swap(a[0], a[1])**. The memory locations for **n1** and **n2** are independent of the ones for **a[0]** and **a[1]**. The contents of the array are not affected by this call.



Stack
Space required for the swap method
n2: 2
n1: 1
Space required for the main method
int[] a reference
Heap
a[0]: 1
a[1]: 2
Stack
Space required for the swapFirstTwoInArray method
int[] array reference
Space required for the main method
int[] a reference
Invoke swap(int n1, int n2). The primitive type values in a[0] and a[1] are passed to the swap method.
The arrays are stored in a heap.
Invoke swapFirstTwoInArray(int[] array). The reference value in a is passed to the swapFirstTwoInArray method.

FIGURE 6.7 When passing an array to a method, the reference of the array is passed to the method.

The parameter in the **swapFirstTwoInArray** method is an array. As shown in Figure 6.7, the reference of the array is passed to the method. Thus the variables **a** (outside the method) and **array** (inside the method) both refer to the same array in the same memory location. Therefore, swapping **array[0]** with **array[1]** inside the method **swapFirstTwoInArray** is the same as swapping **a[0]** with **a[1]** outside of the method.

6.7 Returning an Array from a Method

Key Point

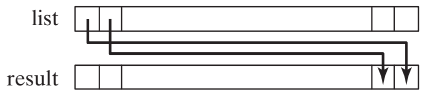
When a method returns an array, the reference of the array is returned.

You can pass arrays when invoking a method. A method may also return an array. For example, the following method returns an array that is the reversal of another array.

create array

return array

```
public static int[] reverse(int[] list) {
  int[] result = new int[list.length];

  for (int i = 0, j = result.length - 1;
       i < list.length; i++, j--) {
    result[j] = list[i];
  }

  return result;
}
```



Line 2 creates a new array `result`. Lines 4–7 copy elements from array `list` to array `result`. Line 9 returns the array. For example, the following statement returns a new array `list2` with elements `6`, `5`, `4`, `3`, `2`, `1`.

```
int[] list1 = {1, 2, 3, 4, 5, 6};
int[] list2 = reverse(list1);
```

6.8 Case Study: Counting the Occurrences of Each Letter

This section presents a program to count the occurrences of each letter in an array of characters.

The program given in Listing 6.4 does the following:

1. Generates `100` lowercase letters randomly and assigns them to an array of characters, as shown in Figure 6.8a. You can obtain a random letter by using the `getRandomLowerCaseLetter()` method in the `RandomCharacter` class in Listing 5.10.
2. Count the occurrences of each letter in the array. To do so, create an array, say `counts`, of `26` `int` values, each of which counts the occurrences of a letter, as shown in Figure 6.8b. That is, `counts[0]` counts the number of `a`'s, `counts[1]` counts the number of `b`'s, and so on.

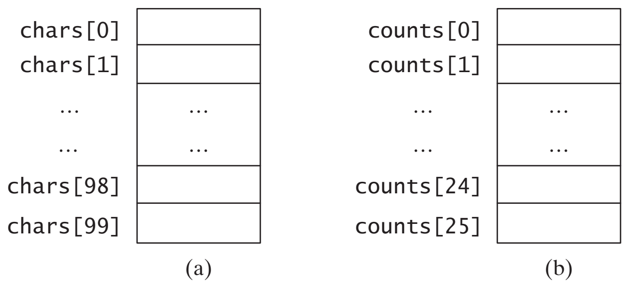


FIGURE 6.8 The `chars` array stores `100` characters, and the `counts` array stores `26` counts, each of which counts the occurrences of a letter.

LISTING 6.4 CountLettersInArray.java

```
public class CountLettersInArray {
  /** Main method */
  public static void main(String[] args) {
    // Declare and create an array
    char[] chars = createArray();

    // Display the array
    System.out.println("The lowercase letters are:");
    displayArray(chars);

    // Count the occurrences of each letter
    int[] counts = countLetters(chars);

    // Display counts
    System.out.println();
    System.out.println("The occurrences of each letter are:");
    displayCounts(counts);
  }

  /** Create an array of characters */
```

create array

pass array

return array

pass array

```java
  public static char[] createArray() {
    // Declare an array of characters and create it
    char[] chars = new char[100];

    // Create lowercase letters randomly and assign
    // them to the array
    for (int i = 0; i < chars.length; i++)
      chars[i] = RandomCharacter.getRandomLowerCaseLetter();

    // Return the array
    return chars;
  }

  /** Display the array of characters */
  public static void displayArray(char[] chars) {
    // Display the characters in the array 20 on each line
    for (int i = 0; i < chars.length; i++) {
      if ((i + 1) % 20 == 0)
        System.out.println(chars[i]);
      else
        System.out.print(chars[i] + " ");
    }
  }

  /** Count the occurrences of each letter */
  public static int[] countLetters(char[] chars) {
    // Declare and create an array of 26 int
    int[] counts = new int[26];

    // For each lowercase letter in the array, count it
    for (int i = 0; i < chars.length; i++)
```

increase count

```java
      counts[chars[i] - 'a']++;

    return counts;
  }

  /** Display counts */
  public static void displayCounts(int[] counts) {
    for (int i = 0; i < counts.length; i++) {
      if ((i + 1) % 10 == 0)
        System.out.println(counts[i] + " " + (char)(i + 'a'));
      else
        System.out.print(counts[i] + " " + (char)(i + 'a') + " ");
    }
  }
}
```

```
The lowercase letters are:
e y l s r i b k j v j h a b z n w b t v
s c c k r d w a m p w v u n q a m p l o
a z g d e g f i n d x m z o u l o z j v
h w i w n t g x w c d o t x h y v z y z
q e a m f w p g u q t r e n n w f c r f

The occurrences of each letter are:
5 a 3 b 4 c 4 d 4 e 4 f 4 g 3 h 3 i 3 j
2 k 3 l 4 m 6 n 4 o 3 p 3 q 4 r 2 s 4 t
3 u 5 v 8 w 3 x 3 y 6 z
```

The `createArray` method (lines 21–32) generates an array of `100` random lowercase letters. Line 5 invokes the method and assigns the array to `chars`. What would be wrong if you rewrote the code as follows?

```
char[] chars = new char[100];
chars = createArray();
```

You would be creating two arrays. The first line would create an array by using `new char[100]`. The second line would create an array by invoking `createArray()` and assign the reference of the array to `chars`. The array created in the first line would be garbage because it is no longer referenced, and as mentioned earlier Java automatically collects garbage behind the scenes. Your program would compile and run correctly, but it would create an array unnecessarily.

Invoking `getRandomLowerCaseLetter()` (line 28) returns a random lowercase letter. This method is defined in the `RandomCharacter` class in Listing 5.10.

The `countLetters` method (lines 46–55) returns an array of `26` `int` values, each of which stores the number of occurrences of a letter. The method processes each letter in the array and increases its count by one. A brute-force approach to count the occurrences of each letter might be as follows:

```
for (int i = 0; i < chars.length; i++)
  if (chars[i] == 'a')
    counts[0]++;
  else if (chars[i] == 'b')
    counts[1]++;
  ...
```

But a better solution is given in lines 51–52.

```
    for (int i = 0; i < chars.length; i++)
      counts[chars[i] - 'a']++;
```

If the letter (`chars[i]`) is `a`, the corresponding count is `counts['a' - 'a']` (i.e., `counts[0]`). If the letter is `b`, the corresponding count is `counts['b' - 'a']` (i.e., `counts[1]`), since the Unicode of `b` is one more than that of `a`. If the letter is `z`, the corresponding count is `counts['z' - 'a']` (i.e., `counts[25]`), since the Unicode of `z` is `25` more than that of `a`.

Figure 6.9 shows the call stack and heap *during* and *after* executing `createArray`. See Checkpoint Question 6.16 to show the call stack and heap for other methods in the program.

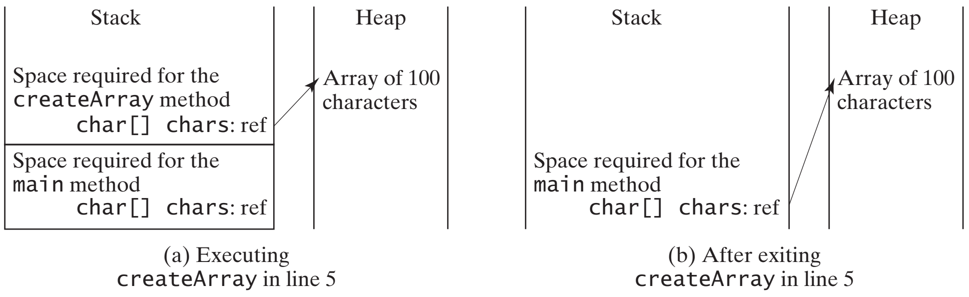

Stack
Heap
Space required for the createArray method
char[] chars: ref
Space required for the main method
char[] chars: ref
Array of 100 characters
(a) Executing createArray in line 5
Stack
Heap
Space required for the main method
char[] chars: ref
Array of 100 characters
(b) After exiting createArray in line 5

FIGURE 6.9 (a) An array of 100 characters is created when executing `createArray`. (b) This array is returned and assigned to the variable `chars` in the `main` method.

Check Point
MyProgrammingLab™

6.14 True or false? When an array is passed to a method, a new array is created and passed to the method.

6.15 Show the output of the following two programs:

```java
public class Test {
  public static void main(String[] args) {
    int number = 0;
    int[] numbers = new int[1];

    m(number, numbers);

    System.out.println("number is " + number
      + " and numbers[0] is " + numbers[0]);
  }

  public static void m(int x, int[] y) {
    x = 3;
    y[0] = 3;
  }
}
```

(a)

```java
public class Test {
  public static void main(String[] args) {
    int[] list = {1, 2, 3, 4, 5};
    reverse(list);
    for (int i = 0; i < list.length; i++)
      System.out.print(list[i] + " ");
  }

  public static void reverse(int[] list) {
    int[] newList = new int[list.length];

    for (int i = 0; i < list.length; i++)
      newList[i] = list[list.length - 1 - i];

    list = newList;
  }
}
```

(b)

6.16 Where are the arrays stored during execution? Show the contents of the stack and heap during and after executing `displayArray`, `countLetters`, `displayCounts` in Listing 6.4.

6.9 Variable-Length Argument Lists

Key Point

A variable number of arguments of the same type can be passed to a method and treated as an array.

You can pass a variable number of arguments of the same type to a method. The parameter in the method is declared as follows:

```
typeName... parameterName
```

In the method declaration, you specify the type followed by an ellipsis (`...`). Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter. Any regular parameters must precede it.

Java treats a variable-length parameter as an array. You can pass an array or a variable number of arguments to a variable-length parameter. When invoking a method with a variable number of arguments, Java creates an array and passes the arguments to it. Listing 6.5 contains a method that prints the maximum value in a list of an unspecified number of values.

LISTING 6.5 VarArgsDemo.java

pass variable-length arg list

pass an array arg

a variable-length arg parameter

```java
public class VarArgsDemo {
  public static void main(String[] args) {
    printMax(34, 3, 3, 2, 56.5);
    printMax(new double[]{1, 2, 3});
  }

  public static void printMax(double... numbers) {
    if (numbers.length == 0) {
      System.out.println("No argument passed");
10          return;
11        }
12
13        double result = numbers[0];
14
15        for (int i = 1; i < numbers.length; i++)
16          if (numbers[i] > result)
17            result = numbers[i];
18
19        System.out.println("The max value is " + result);
20      }
21    }
```

Line 3 invokes the `printMax` method with a variable-length argument list passed to the array `numbers`. If no arguments are passed, the length of the array is `0` (line 8).

Line 4 invokes the `printMax` method with an array.

Check Point

6.17 What is wrong in the following method header?

```
public static void print(String... strings, double... numbers)
public static void print(double... numbers, String name)
public static double... print(double d1, double d2)
```

6.18 Can you invoke the `printMax` method in Listing 6.5 using the following statements?

```
printMax(1, 2, 2, 1, 4);
printMax(new double[]{1, 2, 3});
printMax(new int[]{1, 2, 3});
```

6.10 Searching Arrays

Key Point

If an array is sorted, binary search is more efficient than linear search for finding an element in the array.

Searching is the process of looking for a specific element in an array—for example, discovering whether a certain score is included in a list of scores. Searching is a common task in computer programming. Many algorithms and data structures are devoted to searching. This section discusses two commonly used approaches, *linear search* and *binary search*.

linear search
binary search

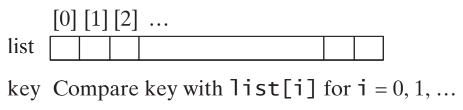
6.10.1 The Linear Search Approach

The linear search approach compares the key element `key` sequentially with each element in the array. It continues to do so until the key matches an element in the array or the array is exhausted without a match being found. If a match is made, the linear search returns the index of the element in the array that matches the key. If no match is found, the search returns `-1`. The `linearSearch` method in Listing 6.6 gives the solution.

linear search animation on Companion Website

LISTING 6.6 `LinearSearch.java`

```
public class LinearSearch {
  /** The method for finding a key in the list */
  public static int linearSearch(int[] list, int key) {
    for (int i = 0; i < list.length; i++) {
      if (key == list[i])
        return i;
    }
    return -1;
  }
}
```



To better understand this method, trace it with the following statements:

```
int[] list = {1, 4, 4, 2, 5, -3, 6, 2};
int i = linearSearch(list, 4);  // Returns 1
int j = linearSearch(list, -4); // Returns -1
int k = linearSearch(list, -3); // Returns 5
```

The linear search method compares the key with each element in the array. The elements can be in any order. On average, the algorithm will have to examine half of the elements in an array before finding the key, if it exists. Since the execution time of a linear search increases linearly as the number of array elements increases, linear search is inefficient for a large array.

6.10.2 The Binary Search Approach

Binary search is the other common search approach for a list of values. For binary search to work, the elements in the array must already be ordered. Assume that the array is in ascending order. The binary search first compares the key with the element in the middle of the array. Consider the following three cases:

- If the key is less than the middle element, you need to continue to search for the key only in the first half of the array.
- If the key is equal to the middle element, the search ends with a match.
- If the key is greater than the middle element, you need to continue to search for the key only in the second half of the array.

Clearly, the binary search method eliminates at least half of the array after each comparison. Sometimes you eliminate half of the elements, and sometimes you eliminate half plus one. Suppose that the array has *n* elements. For convenience, let **n** be a power of **2**. After the first comparison, **n/2** elements are left for further search; after the second comparison, **(n/2)/2** elements are left. After the **k**th comparison, $n/2^k$ elements are left for further search. When $k = \log_2 n$, only one element is left in the array, and you need only one more comparison. Therefore, in the worst case when using the binary search approach, you need $\log_2 n+1$ comparisons to find an element in the sorted array. In the worst case for a list of **1024** (2^{10}) elements, binary search requires only **11** comparisons, whereas a linear search requires **1023** comparisons in the worst case.

binary search animation on Companion Website

The portion of the array being searched shrinks by half after each comparison. Let **low** and **high** denote, respectively, the first index and last index of the array that is currently being searched. Initially, **low** is **0** and **high** is **list.length–1**. Let **mid** denote the index of the middle element, so **mid** is **(low + high)/2**. Figure 6.10 shows how to find key **11** in the list {**2**, **4**, **7**, **10**, **11**, **45**, **50**, **59**, **60**, **66**, **69**, **70**, **79**} using binary search.

You now know how the binary search works. The next task is to implement it in Java. Don't rush to give a complete implementation. Implement it incrementally, one step at a time. You may start with the first iteration of the search, as shown in Figure 6.11a. It compares the key with the middle element in the list whose **low** index is **0** and **high** index is **list.length - 1**. If **key < list[mid]**, set the **high** index to **mid - 1**; if **key == list[mid]**, a match is found and return **mid**; if **key > list[mid]**, set the **low** index to **mid + 1**.

Next consider implementing the method to perform the search repeatedly by adding a loop, as shown in Figure 6.11b. The search ends if the key is found, or if the key is not found when **low > high**.

why not –1?

When the key is not found, **low** is the insertion point where a key would be inserted to maintain the order of the list. It is more useful to return the insertion point than **-1**. The method must return a negative value to indicate that the key is not in the list. Can it simply return **-low**? No. If the key is less than **list[0]**, **low** would be **0**. **-0** is **0**. This would

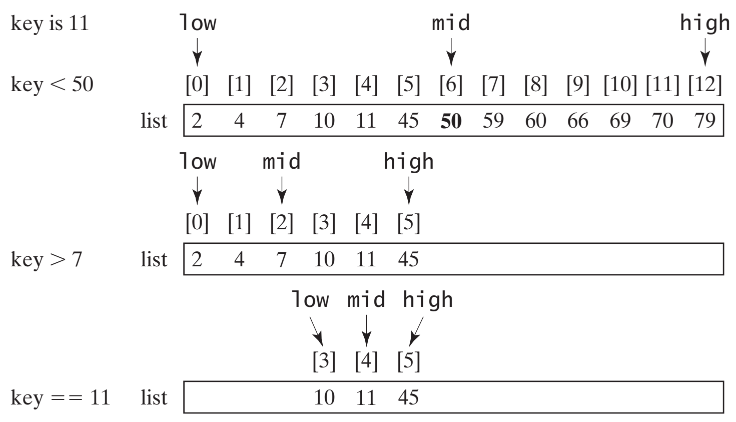


FIGURE 6.10 Binary search eliminates half of the list from further consideration after each comparison.

```java
public static int binarySearch(
    int[] list, int key) {
  int low = 0;
  int high = list.length - 1;


    int mid = (low + high) / 2;
    if (key < list[mid])
      high = mid - 1;
    else if (key == list[mid])
      return mid;
    else
      low = mid + 1;




}
```

(a) Version 1

```java
public static int binarySearch(
    int[] list, int key) {
  int low = 0;
  int high = list.length - 1;

  while (high >= low) {
    int mid = (low + high) / 2;
    if (key < list[mid])
      high = mid - 1;
    else if (key == list[mid])
      return mid;
    else
      low = mid + 1;
  }

  return -1; // Not found
}
```

(b) Version 2

FIGURE 6.11 Binary search is implemented incrementally.

indicate that the key matches `list[0]`. A good choice is to let the method return `-low - 1` if the key is not in the list. Returning `-low - 1` indicates not only that the key is not in the list, but also where the key would be inserted.

The complete program is given in Listing 6.7.

LISTING 6.7 BinarySearch.java

```java
public class BinarySearch {
  /** Use binary search to find the key in the list */
  public static int binarySearch(int[] list, int key) {
    int low = 0;
    int high = list.length - 1;

    while (high >= low) {
      int mid = (low + high) / 2;
      if (key < list[mid])
        high = mid - 1;
      else if (key == list[mid])
```

first half

second half

```
12          return mid;
13        else
14          low = mid + 1;
15      }
16
17      return -low - 1; // Now high < low, key not found
18    }
19  }
```

The binary search returns the index of the search key if it is contained in the list (line 12). Otherwise, it returns `-low - 1` (line 17).

What would happen if we replaced `(high >= low)` in line 7 with (`high > low`)? The search would miss a possible matching element. Consider a list with just one element. The search would miss the element.

Does the method still work if there are duplicate elements in the list? Yes, as long as the elements are sorted in increasing order. The method returns the index of one of the matching elements if the element is in the list.

To better understand this method, trace it with the following statements and identify `low` and `high` when the method returns.

```
int[] list = {2, 4, 7, 10, 11, 45, 50, 59, 60, 66, 69, 70, 79};
int i = BinarySearch.binarySearch(list, 2); // Returns 0
int j = BinarySearch.binarySearch(list, 11); // Returns 4
int k = BinarySearch.binarySearch(list, 12); // Returns -6
int l = BinarySearch.binarySearch(list, 1); // Returns -1
int m = BinarySearch.binarySearch(list, 3); // Returns -2
```

Here is the table that lists the `low` and `high` values when the method exits and the value returned from invoking the method.

Method	Low	High	Value Returned
`binarySearch(list, 2)`	0	1	0
`binarySearch(list, 11)`	3	5	4
`binarySearch(list, 12)`	5	4	-6
`binarySearch(list, 1)`	0	-1	-1
`binarySearch(list, 3)`	1	0	-2

binary search benefits

Note

Linear search is useful for finding an element in a small array or an unsorted array, but it is inefficient for large arrays. Binary search is more efficient, but it requires that the array be presorted.

6.11 Sorting Arrays

Key Point

There are many strategies for sorting elements in an array. Selection sort and insertion sort are two common approaches.

selection sort
insertion sort

Sorting, like searching, is a common task in computer programming. Many different algorithms have been developed for sorting. This section introduces two simple, intuitive sorting algorithms: *selection sort* and *insertion sort*.

6.11.1 Selection Sort

VideoNote
Selection sort

selection sort animation on Companion Website

Suppose that you want to sort a list in ascending order. Selection sort finds the smallest number in the list and swaps it with the first element. It then finds the smallest number remaining and swaps it with the second element, and so on, until only a single number remains. Figure 6.12 shows how to sort the list {**2**, **9**, **5**, **4**, **8**, **1**, **6**} using selection sort.

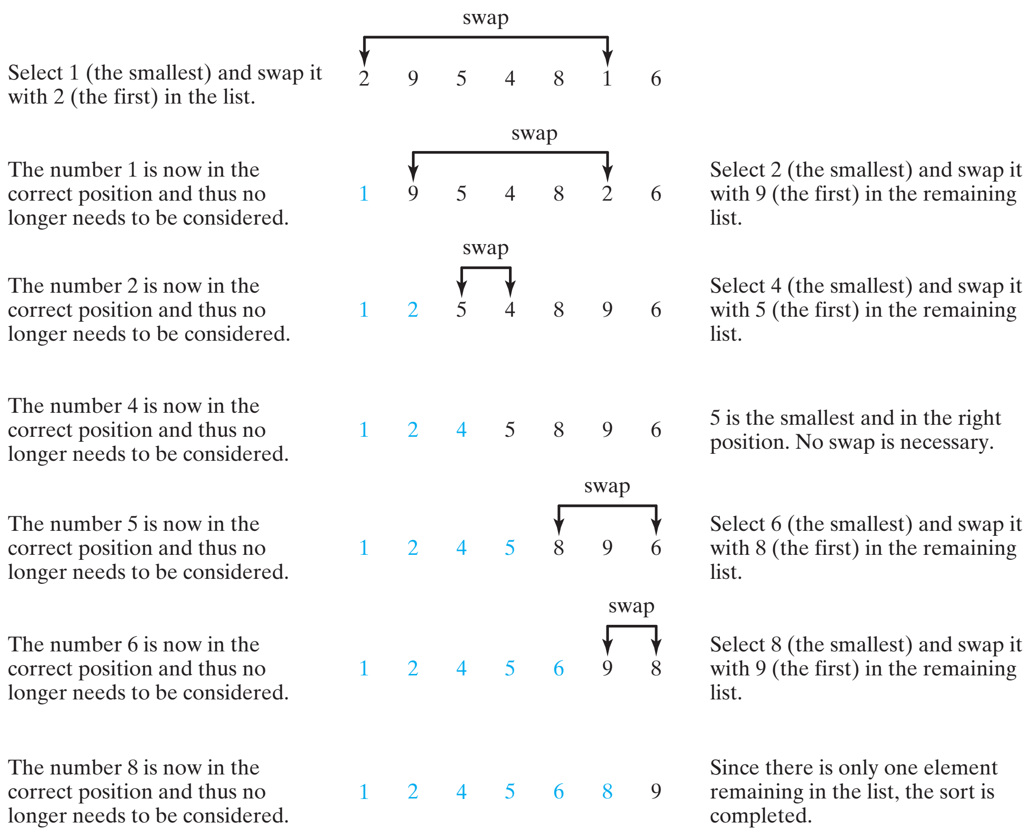

swap
Select 1 (the smallest) and swap it with 2 (the first) in the list.
2 9 5 4 8 1 6
swap
The number 1 is now in the correct position and thus no longer needs to be considered.
1 9 5 4 8 2 6
Select 2 (the smallest) and swap it with 9 (the first) in the remaining list.
swap
The number 2 is now in the correct position and thus no longer needs to be considered.
1 2 5 4 8 9 6
Select 4 (the smallest) and swap it with 5 (the first) in the remaining list.
The number 4 is now in the correct position and thus no longer needs to be considered.
1 2 4 5 8 9 6
5 is the smallest and in the right position. No swap is necessary.
swap
The number 5 is now in the correct position and thus no longer needs to be considered.
1 2 4 5 8 9 6
Select 6 (the smallest) and swap it with 8 (the first) in the remaining list.
swap
The number 6 is now in the correct position and thus no longer needs to be considered.
1 2 4 5 6 9 8
Select 8 (the smallest) and swap it with 9 (the first) in the remaining list.
The number 8 is now in the correct position and thus no longer needs to be considered.
1 2 4 5 6 8 9
Since there is only one element remaining in the list, the sort is completed.

FIGURE 6.12 Selection sort repeatedly selects the smallest number and swaps it with the first number in the list.

You know how the selection-sort approach works. The task now is to implement it in Java. Beginners find it difficult to develop a complete solution on the first attempt. Start by writing the code for the first iteration to find the smallest element in the list and swap it with the first element, and then observe what would be different for the second iteration, the third, and so on. The insight this gives will enable you to write a loop that generalizes all the iterations.

The solution can be described as follows:

```
for (int i = 0; i < list.length - 1; i++) {
  select the smallest element in list[i..list.length-1];
  swap the smallest with list[i], if necessary;
  // list[i] is in its correct position.
  // The next iteration apply on list[i+1..list.length-1]
}
```

Listing 6.8 implements the solution.

LISTING 6.8 SelectionSort.java

```
 1  public class SelectionSort {
 2    /** The method for sorting the numbers */
 3    public static void selectionSort(double[] list) {
 4      for (int i = 0; i < list.length - 1; i++) {
 5        // Find the minimum in the list[i..list.length-1]
 6        double currentMin = list[i];
 7        int currentMinIndex = i;
 8
 9        for (int j = i + 1; j < list.length; j++) {
10          if (currentMin > list[j]) {
11            currentMin = list[j];
12            currentMinIndex = j;
13          }
14        }
15
16        // Swap list[i] with list[currentMinIndex] if necessary
17        if (currentMinIndex != i) {
18          list[currentMinIndex] = list[i];
19          list[i] = currentMin;
20        }
21      }
22    }
23  }
```

select

swap

The `selectionSort(double[] list)` method sorts any array of `double` elements. The method is implemented with a nested `for` loop. The outer loop (with the loop control variable `i`) (line 4) is iterated in order to find the smallest element in the list, which ranges from `list[i]` to `list[list.length-1]`, and exchange it with `list[i]`.

The variable `i` is initially `0`. After each iteration of the outer loop, `list[i]` is in the right place. Eventually, all the elements are put in the right place; therefore, the whole list is sorted.

To understand this method better, trace it with the following statements:

```
double[] list = {1, 9, 4.5, 6.6, 5.7, -4.5};
SelectionSort.selectionSort(list);
```

6.11.2 Insertion Sort

insertion sort animation on Companion Website

Suppose that you want to sort a list in ascending order. The insertion-sort algorithm sorts a list of values by repeatedly inserting a new element into a sorted sublist until the whole list is sorted. Figure 6.13 shows how to sort the list {`2`, `9`, `5`, `4`, `8`, `1`, `6`} using insertion sort.

The algorithm can be described as follows:

```
for (int i = 1; i < list.length; i++) {
  insert list[i] into a sorted sublist list[0..i-1] so that
  list[0..i] is sorted.
}
```

To insert `list[i]` into `list[0..i-1]`, save `list[i]` into a temporary variable, say `currentElement`. Move `list[i-1]` to `list[i]` if `list[i-1] > currentElement`, move `list[i-2]` to `list[i-1]` if `list[i-2] > currentElement`, and so on, until `list[i-k] <= currentElement` or `k > i` (we pass the first element of the sorted list). Assign `currentElement` to `list[i-k+1]`. For example, to insert `4` into {`2`, `5`, `9`} in Step 4 in Figure 6.14, move `list[2]` (`9`) to `list[3]` since `9 > 4`, and move `list[1]` (`5`) to `list[2]` since `5 > 4`. Finally, move `currentElement` (`4`) to `list[1]`.

Step 1: Initially, the sorted sublist contains the first element in the list. Insert 9 into the sublist.

Step 2: The sorted sublist is {2, 9}. Insert 5 into the sublist.

Step 3: The sorted sublist is {2, 5, 9}. Insert 4 into the sublist.

Step 4: The sorted sublist is {2, 4, 5, 9}. Insert 8 into the sublist.

Step 5: The sorted sublist is {2, 4, 5, 8, 9}. Insert 1 into the sublist.

Step 6: The sorted sublist is {1, 2, 4, 5, 8, 9}. Insert 6 into the sublist.

Step 7: The entire list is now sorted.

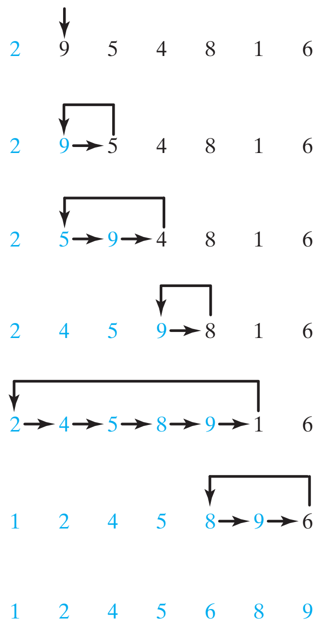
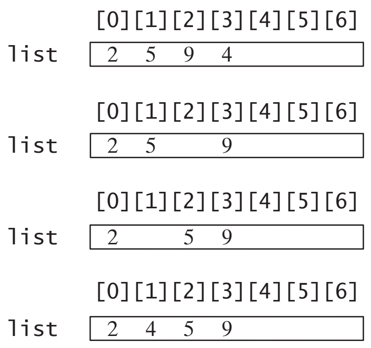


FIGURE 6.13 Insertion sort repeatedly inserts a new element into a sorted sublist.



Step 1: Save 4 to a temporary variable `currentElement`

Step 2: Move `list[2]` to `list[3]`

Step 3: Move `list[1]` to `list[2]`

Step 4: Assign `currentElement` to `list[1]`

FIGURE 6.14 A new element is inserted into a sorted sublist.

The algorithm can be expanded and implemented as in Listing 6.9.

LISTING 6.9 InsertionSort.java

```java
public class InsertionSort {
  /** The method for sorting the numbers */
  public static void insertionSort(double[] list) {
    for (int i = 1; i < list.length; i++) {
      /** Insert list[i] into a sorted sublist list[0..i-1] so that
           list[0..i] is sorted. */
      double currentElement = list[i];
      int k;
      for (k = i - 1; k >= 0 && list[k] > currentElement; k--) {
        list[k + 1] = list[k];
      }

      // Insert the current element into list[k + 1]
```

shift

insert

```
14          list[k + 1] = currentElement;
15        }
16      }
17    }
```

The `insertionSort(double[] list)` method sorts any array of `double` elements. The method is implemented with a nested `for` loop. The outer loop (with the loop control variable `i`) (line 4) is iterated in order to obtain a sorted sublist, which ranges from `list[0]` to `list[i]`. The inner loop (with the loop control variable `k`) inserts `list[i]` into the sublist from `list[0]` to `list[i-1]`.

To better understand this method, trace it with the following statements:

```
double[] list = {1, 9, 4.5, 6.6, 5.7, -4.5};
InsertionSort.insertionSort(list);
```

Check Point

MyProgrammingLab™

6.19 Use Figure 6.10 as an example to show how to apply the binary search approach to a search for key `10` and key `12` in list {`2`, `4`, `7`, `10`, `11`, `45`, `50`, `59`, `60`, `66`, `69`, `70`, `79`}.

6.20 If the binary search method returns `-4`, is the key in the list? Where should the key be inserted if you wish to insert the key into the list?

6.21 Use Figure 6.12 as an example to show how to apply the selection-sort approach to sort {`3.4`, `5`, `3`, `3.5`, `2.2`, `1.9`, `2`}.

6.22 Use Figure 6.13 as an example to show how to apply the insertion-sort approach to sort {`3.4`, `5`, `3`, `3.5`, `2.2`, `1.9`, `2`}.

6.23 How do you modify the `selectionSort` method in Listing 6.8 to sort numbers in decreasing order?

6.24 How do you modify the `insertionSort` method in Listing 6.9 to sort numbers in decreasing order?

6.12 The `Arrays` Class

Key Point

The `java.util.Arrays` class contains useful methods for common array operations such as sorting and searching.

The `java.util.Arrays` class contains various static methods for sorting and searching arrays, comparing arrays, filling array elements, and returning a string representation of the array. These methods are overloaded for all primitive types.

sort

You can use the `sort` method to sort a whole array or a partial array. For example, the following code sorts an array of numbers and an array of characters.

```
double[] numbers = {6.0, 4.4, 1.9, 2.9, 3.4, 3.5};
java.util.Arrays.sort(numbers); // Sort the whole array

char[] chars = {'a', 'A', '4', 'F', 'D', 'P'};
java.util.Arrays.sort(chars, 1, 3); // Sort part of the array
```

Invoking `sort(numbers)` sorts the whole array `numbers`. Invoking `sort(chars, 1, 3)` sorts a partial array from `chars[1]` to `chars[3-1]`.

binarySearch

You can use the `binarySearch` method to search for a key in an array. The array must be pre-sorted in increasing order. If the key is not in the array, the method returns `-(insertionindex + 1)`. For example, the following code searches the keys in an array of integers and an array of characters.

```
int[] list = {2, 4, 7, 10, 11, 45, 50, 59, 60, 66, 69, 70, 79};
System.out.println("(1) Index is " +
      java.util.Arrays.binarySearch(list, 11));
```

```
System.out.println("(2) Index is " +
       java.util.Arrays.binarySearch(list, 12));

char[] chars = {'a', 'c', 'g', 'x', 'y', 'z'};
System.out.println("(3) Index is " +
       java.util.Arrays.binarySearch(chars, 'a'));
System.out.println("(4) Index is " +
       java.util.Arrays.binarySearch(chars, 't'));
```

The output of the preceding code is

1. Index is 4
2. Index is –6
3. Index is 0
4. Index is –4

equals

You can use the **equals** method to check whether two arrays are equal. Two arrays are equal if they have the same contents. In the following code, **list1** and **list2** are equal, but **list2** and **list3** are not.

```
int[] list1 = {2, 4, 7, 10};
int[] list2 = {2, 4, 7, 10};
int[] list3 = {4, 2, 7, 10};
System.out.println(java.util.Arrays.equals(list1, list2)); // true
System.out.println(java.util.Arrays.equals(list2, list3)); // false
```

fill

You can use the **fill** method to fill in all or part of the array. For example, the following code fills **list1** with **5** and fills **8** into elements **list2[1]** and **list2[3-1]**.

```
int[] list1 = {2, 4, 7, 10};
int[] list2 = {2, 4, 7, 10};
java.util.Arrays.fill(list1, 5); // Fill 5 to the whole array
java.util.Arrays.fill(list2, 1, 3, 8); // Fill 8 to a partial array
```

toString

You can also use the **toString** method to return a string that represents all elements in the array. This is a quick and simple way to display all elements in the array. For example, the following code

```
int[] list = {2, 4, 7, 10};
System.out.println(Arrays.toString(list));
```

displays **[2, 4, 7, 10]**.

Check Point
MyProgrammingLab™

6.25 What types of array can be sorted using the **java.util.Arrays.sort** method? Does this **sort** method create a new array?

6.26 To apply **java.util.Arrays.binarySearch(array, key)**, should the array be sorted in increasing order, in decreasing order, or neither?

6.27 Show the output of the following code:

```
int[] list1 = {2, 4, 7, 10};
java.util.Arrays.fill(list1, 7);
System.out.println(java.util.Arrays.toString(list1));

int[] list2 = {2, 4, 7, 10};
System.out.println(java.util.Arrays.toString(list2));
System.out.print(java.util.Arrays.equals(list1, list2));
```

Key Terms

anonymous array	238
array	224
array initializer	227
binary search	245
garbage collection	236
index	224
indexed variable	226
insertion sort	248
linear search	245
off-by-one error	230
selection sort	248

Chapter Summary

1. A variable is declared as an *array* type using the syntax `elementType[] arrayRefVar` or `elementType arrayRefVar[]`. The style `elementType[] arrayRefVar` is preferred, although `elementType arrayRefVar[]` is legal.

2. Unlike declarations for primitive data type variables, the declaration of an array variable does not allocate any space in memory for the array. An array variable is not a primitive data type variable. An array variable contains a reference to an array.

3. You cannot assign elements to an array unless it has already been created. You can create an array by using the `new` operator with the following syntax: `new elementType[arraySize]`.

4. Each element in the array is represented using the syntax `arrayRefVar[index]`. An *index* must be an integer or an integer expression.

5. After an array is created, its size becomes permanent and can be obtained using `arrayRefVar.length`. Since the index of an array always begins with `0`, the last index is always `arrayRefVar.length - 1`. An out-of-bounds error will occur if you attempt to reference elements beyond the bounds of an array.

6. Programmers often mistakenly reference the first element in an array with index `1`, but it should be `0`. This is called the index *off-by-one error*.

7. When an array is created, its elements are assigned the default value of `0` for the numeric primitive data types, `\u0000` for char types, and `false` for `boolean` types.

8. Java has a shorthand notation, known as the *array initializer*, which combines declaring an array, creating an array, and initializing an array in one statement, using the syntax `elementType[] arrayRefVar = {value0, value1, ..., valuek}`.

9. When you pass an array argument to a method, you are actually passing the reference of the array; that is, the called method can modify the elements in the caller's original array.

10. If an array is sorted, *binary search* is more efficient than *linear search* for finding an element in the array.

11. *Selection sort* finds the smallest number in the list and swaps it with the first element. It then finds the smallest number remaining and swaps it with the first element in the remaining list, and so on, until only a single number remains.

12. The *insertion-sort algorithm* sorts a list of values by repeatedly inserting a new element into a sorted sublist until the whole list is sorted.

Test Questions

Do the test questions for this chapter online at www.cs.armstrong.edu/liang/intro9e/test.html.

Programming Exercises

MyProgrammingLab™

Sections 6.2–6.5

***6.1** (*Assign grades*) Write a program that reads student scores, gets the best score, and then assigns grades based on the following scheme:

Grade is A if score is >= best − 10

Grade is B if score is >= best − 20;

Grade is C if score is >= best − 30;

Grade is D if score is >= best − 40;

Grade is F otherwise.

The program prompts the user to enter the total number of students, then prompts the user to enter all of the scores, and concludes by displaying the grades. Here is a sample run:

```
Enter the number of students: 4 ↵Enter
Enter 4 scores: 40 55 70 58 ↵Enter
Student 0 score is 40 and grade is C
Student 1 score is 55 and grade is B
Student 2 score is 70 and grade is A
Student 3 score is 58 and grade is B
```

6.2 (*Reverse the numbers entered*) Write a program that reads ten integers and displays them in the reverse of the order in which they were read.

****6.3** (*Count occurrence of numbers*) Write a program that reads the integers between 1 and 100 and counts the occurrences of each. Assume the input ends with **0**. Here is a sample run of the program:

```
Enter the integers between 1 and 100: 2 5 6 5 4 3 23 43 2 0 ↵Enter
2 occurs 2 times
3 occurs 1 time
4 occurs 1 time
5 occurs 2 times
6 occurs 1 time
23 occurs 1 time
43 occurs 1 time
```

Note that if a number occurs more than one time, the plural word "times" is used in the output.

6.4 (*Analyze scores*) Write a program that reads an unspecified number of scores and determines how many scores are above or equal to the average and how many scores are below the average. Enter a negative number to signify the end of the input. Assume that the maximum number of scores is 100.

****6.5** (*Print distinct numbers*) Write a program that reads in ten numbers and displays distinct numbers (i.e., if a number appears multiple times, it is displayed only once). (*Hint*: Read a number and store it to an array if it is new. If the number is already in the array, ignore it.) After the input, the array contains the distinct numbers. Here is the sample run of the program:

```
Enter ten numbers: 1 2 3 2 1 6 3 4 5 2 ↵Enter
The distinct numbers are: 1 2 3 6 4 5
```

***6.6** (*Revise Listing 4.14, PrimeNumber.java*) Listing 4.14 determines whether a number `n` is prime by checking whether `2`, `3`, `4`, `5`, `6`, ..., `n/2` is a divisor. If a divisor is found, `n` is not prime. A more efficient approach is to check whether any of the prime numbers less than or equal to $\sqrt{n}$ can divide `n` evenly. If not, `n` is prime. Rewrite Listing 4.14 to display the first 50 prime numbers using this approach. You need to use an array to store the prime numbers and later use them to check whether they are possible divisors for `n`.

***6.7** (*Count single digits*) Write a program that generates 100 random integers between 0 and 9 and displays the count for each number. (*Hint*: Use `(int)(Math.random() * 10)` to generate a random integer between 0 and 9. Use an array of ten integers, say `counts`, to store the counts for the number of 0s, 1s, ..., 9s.)

Sections 6.6–6.8

6.8 (*Average an array*) Write two overloaded methods that return the average of an array with the following headers:

```
public static int average(int[] array)
public static double average(double[] array)
```

Write a test program that prompts the user to enter ten double values, invokes this method, and displays the average value.

6.9 (*Find the smallest element*) Write a method that finds the smallest element in an array of double values using the following header:

```
public static double min(double[] array)
```

Write a test program that prompts the user to enter ten numbers, invokes this method to return the minimum value, and displays the minimum value. Here is a sample run of the program:

```
Enter ten numbers: 1.9 2.5 3.7 2 1.5 6 3 4 5 2 ↵Enter
The minimum number is: 1.5
```

6.10 (*Find the index of the smallest element*) Write a method that returns the index of the smallest element in an array of integers. If the number of such elements is greater than 1, return the smallest index. Use the following header:

```
public static int indexOfSmallestElement(double[] array)
```

Write a test program that prompts the user to enter ten numbers, invokes this method to return the index of the smallest element, and displays the index.

***6.11** (*Statistics: compute deviation*) Programming Exercise 5.37 computes the standard deviation of numbers. This exercise uses a different but equivalent formula to compute the standard deviation of **n** numbers.

$$mean = \frac{\sum_{i=1}^{n} x_i}{n} = \frac{x_1 + x_2 + \ldots + x_n}{n} \qquad deviation = \sqrt{\frac{\sum_{i=1}^{n}(x_i - mean)^2}{n-1}}$$

To compute the standard deviation with this formula, you have to store the individual numbers using an array, so that they can be used after the mean is obtained.

Your program should contain the following methods:

```
/** Compute the deviation of double values */
public static double deviation(double[] x)

/** Compute the mean of an array of double values */
public static double mean(double[] x)
```

Write a test program that prompts the user to enter ten numbers and displays the mean and standard deviation, as shown in the following sample run:

```
Enter ten numbers: 1.9 2.5 3.7 2 1 6 3 4 5 2 Enter
The mean is 3.11
The standard deviation is 1.55738
```

***6.12** (*Reverse an array*) The **reverse** method in Section 6.7 reverses an array by copying it to a new array. Rewrite the method that reverses the array passed in the argument and returns this array. Write a test program that prompts the user to enter ten numbers, invokes the method to reverse the numbers, and displays the numbers.

Section 6.9

***6.13** (*Random number chooser*) Write a method that returns a random number between 1 and 54, excluding the numbers passed in the argument. The method header is specified as follows:

```
public static int getRandom(int... numbers)
```

6.14 (*Computing gcd*) Write a method that returns the gcd of an unspecified number of integers. The method header is specified as follows:

```
public static int gcd(int... numbers)
```

Write a test program that prompts the user to enter five numbers, invokes the method to find the gcd of these numbers, and displays the gcd.

Sections 6.10–6.12

6.15 (*Eliminate duplicates*) Write a method that returns a new array by eliminating the duplicate values in the array using the following method header:

```
public static int[] eliminateDuplicates(int[] list)
```

Write a test program that reads in ten integers, invokes the method, and displays the result. Here is the sample run of the program:

```
Enter ten numbers: 1 2 3 2 1 6 3 4 5 2 ↵Enter
The distinct numbers are: 1 2 3 6 4 5
```

6.16 (*Execution time*) Write a program that randomly generates an array of 100,000 integers and a key. Estimate the execution time of invoking the **linearSearch** method in Listing 6.6. Sort the array and estimate the execution time of invoking the **binarySearch** method in Listing 6.7. You can use the following code template to obtain the execution time:

```
long startTime = System.currentTimeMillis();
perform the task;
long endTime = System.currentTimeMillis();
long executionTime = endTime - startTime;
```

****6.17** (*Sort students*) Write a program that prompts the user to enter the number of students, the students' names, and their scores, and prints student names in decreasing order of their scores.

****6.18** (*Bubble sort*) Write a sort method that uses the bubble-sort algorithm. The bubble-sort algorithm makes several passes through the array. On each pass, successive neighboring pairs are compared. If a pair is not in order, its values are swapped; otherwise, the values remain unchanged. The technique is called a *bubble sort* or *sinking sort* because the smaller values gradually "bubble" their way to the top and the larger values "sink" to the bottom. Write a test program that reads in ten double numbers, invokes the method, and displays the sorted numbers.

****6.19** (*Sorted?*) Write the following method that returns true if the list is already sorted in increasing order.

```
public static boolean isSorted(int[] list)
```

Write a test program that prompts the user to enter a list and displays whether the list is sorted or not. Here is a sample run. Note that the first number in the input indicates the number of the elements in the list.

```
Enter list: 8 10 1 5 16 61 9 11 1 ↵Enter
The list is not sorted
```

```
Enter list: 10 1 1 3 4 4 5 7 9 11 21 ↵Enter
The list is already sorted
```

***6.20** (*Revise selection sort*) In Section 6.11.1, you used selection sort to sort an array. The selection-sort method repeatedly finds the smallest number in the current array and swaps it with the first. Rewrite this program by finding the largest number and swapping it with the last. Write a test program that reads in ten double numbers, invokes the method, and displays the sorted numbers.

*****6.21** (*Game: bean machine*) The bean machine, also known as a quincunx or the Galton box, is a device for statistics experiments named after English scientist Sir Francis Galton. It consists of an upright board with evenly spaced nails (or pegs) in a triangular form, as shown in Figure 6.15.

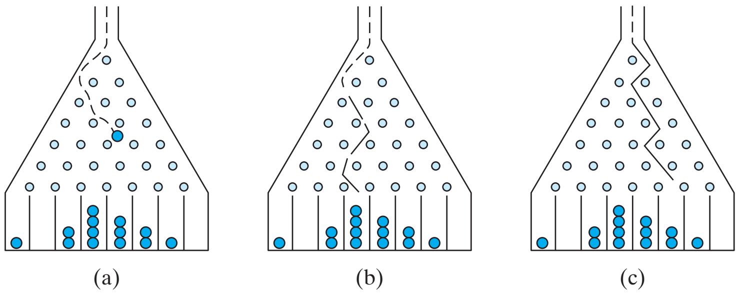


FIGURE 6.15 Each ball takes a random path and falls into a slot.

Balls are dropped from the opening of the board. Every time a ball hits a nail, it has a 50% chance of falling to the left or to the right. The piles of balls are accumulated in the slots at the bottom of the board.

Write a program that simulates the bean machine. Your program should prompt the user to enter the number of the balls and the number of the slots in the machine. Simulate the falling of each ball by printing its path. For example, the path for the ball in Figure 6.15b is LLRRLLR and the path for the ball in Figure 6.15c is RLRRLRR. Display the final buildup of the balls in the slots in a histogram. Here is a sample run of the program:

```
Enter the number of balls to drop: 5 ↵Enter
Enter the number of slots in the bean machine: 7 ↵Enter

LRLRLRR
RRLLLRR
LLRLLRR
RRLLLLL
LRLRRLR

    0
    0
  000
```

(*Hint*: Create an array named `slots`. Each element in `slots` stores the number of balls in a slot. Each ball falls into a slot via a path. The number of Rs in a path is the position of the slot where the ball falls. For example, for the path LRLRLRR, the ball falls into `slots[4]`, and for the path is RRLLLLL, the ball falls into `slots[2]`.)

*****6.22** (*Game: Eight Queens*) The classic Eight Queens puzzle is to place eight queens on a chessboard such that no two queens can attack each other (i.e., no two queens are on the same row, same column, or same diagonal). There are many possible solutions. Write a program that displays one such solution. A sample output is shown below:

```
|Q| | | | | | | |
| | | | |Q| | | |
| | | | | | | |Q|
| | | | | |Q| | |
| | |Q| | | | | |
| | | | | | |Q| |
| |Q| | | | | | |
| | | |Q| | | | |
```

**6.23 (*Game: locker puzzle*) A school has 100 lockers and 100 students. All lockers are closed on the first day of school. As the students enter, the first student, denoted S1, opens every locker. Then the second student, S2, begins with the second locker, denoted L2, and closes every other locker. Student S3 begins with the third locker and changes every third locker (closes it if it was open, and opens it if it was closed). Student S4 begins with locker L4 and changes every fourth locker. Student S5 starts with L5 and changes every fifth locker, and so on, until student S100 changes L100.

After all the students have passed through the building and changed the lockers, which lockers are open? Write a program to find your answer.

(*Hint*: Use an array of 100 Boolean elements, each of which indicates whether a locker is open (`true`) or closed (`false`). Initially, all lockers are closed.)

VideoNote
Coupon collector's problem

**6.24 (*Simulation: coupon collector's problem*) Coupon collector is a classic statistics problem with many practical applications. The problem is to pick objects from a set of objects repeatedly and find out how many picks are needed for all the objects to be picked at least once. A variation of the problem is to pick cards from a shuffled deck of 52 cards repeatedly and find out how many picks are needed before you see one of each suit. Assume a picked card is placed back in the deck before picking another. Write a program to simulate the number of picks needed to get four cards from each suit and display the four cards picked (it is possible a card may be picked twice). Here is a sample run of the program:

```
Queen of Spades
5 of Clubs
Queen of Hearts
4 of Diamonds
Number of picks: 12
```

6.25 (*Algebra: solve quadratic equations*) Write a method for solving a quadratic equation using the following header:

```
public static int solveQuadratic(double[] eqn, double[] roots)
```

The coefficients of a quadratic equation $ax^2 + bx + c = 0$ are passed to the array `eqn` and the noncomplex roots are stored in roots. The method returns the number of roots. See Programming Exercise 3.1 on how to solve a quadratic equation.

Write a program that prompts the user to enter values for *a*, *b*, and *c* and displays the number of roots and all noncomplex roots.

6.26 (*Strictly identical arrays*) The arrays `list1` and `list2` are *strictly identical* if their corresponding elements are equal. Write a method that returns `true` if `list1` and `list2` are strictly identical, using the following header:

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

Write a test program that prompts the user to enter two lists of integers and displays whether the two are strictly identical. Here are the sample runs. Note that the first number in the input indicates the number of the elements in the list.

```
Enter list1: 5 2 5 6 1 6 ↵Enter
Enter list2: 5 2 5 6 1 6 ↵Enter
Two lists are strictly identical
```

```
Enter list1: 5 2 5 6 6 1 ↵Enter
Enter list2: 5 2 5 6 1 6 ↵Enter
Two lists are not strictly identical
```

6.27 (*Identical arrays*) The arrays `list1` and `list2` are *identical* if they have the same contents. Write a method that returns `true` if `list1` and `list2` are identical, using the following header:

```java
public static boolean equals(int[] list1, int[] list2)
```

Write a test program that prompts the user to enter two lists of integers and displays whether the two are identical. Here are the sample runs. Note that the first number in the input indicates the number of the elements in the list.

```
Enter list1: 5 2 5 6 6 1 ↵Enter
Enter list2: 5 5 2 6 1 6 ↵Enter
Two lists are identical
```

```
Enter list1: 5 5 5 6 6 1 ↵Enter
Enter list2: 5 2 5 6 1 6 ↵Enter
Two lists are not identical
```

***6.28** (*Math: combinations*) Write a program that prompts the user to enter 10 integers and displays all combinations of picking two numbers from the 10.

***6.29** (*Game: pick four cards*) Write a program that picks four cards from a deck of 52 cards and computes their sum. An Ace, King, Queen, and Jack represent 1, 13, 12, and 11, respectively. Your program should display the number of picks that yields the sum of 24.

***6.30** (*Pattern recognition: consecutive four equal numbers*) Write the following method that tests whether the array has four consecutive numbers with the same value.

VideoNote
Consecutive four

```java
public static boolean isConsecutiveFour(int[] values)
```

Write a test program that prompts the user to enter a series of integers and displays true if the series contains four consecutive numbers with the same value. Otherwise, display false. Your program should first prompt the user to enter the input size—i.e., the number of values in the series.

****6.31** (*Merge two sorted lists*) Write the following method that merges two sorted lists into a new sorted list.

```java
public static int[] merge(int[] list1, int[] list2)
```

Implement the method in a way that takes `list1.length + list2.length` comparisons. Write a test program that prompts the user to enter two sorted lists and displays the merged list. Here is a sample run. Note that the first number in the input indicates the number of the elements in the list.

```
Enter list1: 5 1 5 16 61 111 ↵Enter
Enter list2: 4 2 4 5 6 ↵Enter
The merged list is 1 2 4 5 5 6 16 61 111
```

**6.32 (*Partition of a list*) Write the following method that partitions the list using the first element, called a *pivot*.

```
public static int partition(int[] list)
```

After the partition, the elements in the list are rearranged so that all the elements before the pivot are less than or equal to the pivot and the elements after the pivot are greater than the pivot. The method returns the index where the pivot is located in the new list. For example, suppose the list is {5, 2, 9, 3, 6, 8}. After the partition, the list becomes {3, 2, 5, 9, 6, 8}. Implement the method in a way that takes `list.length` comparisons. Write a test program that prompts the user to enter a list and displays the list after the partition. Here is a sample run. Note that the first number in the input indicates the number of the elements in the list.

```
Enter list: 8 10 1 5 16 61 9 11 1 ↵Enter
After the partition, the list is 9 1 5 1 10 61 11 16
```

*6.33 (*Culture: Chinese Zodiac*) Simplify Listing 3.10 using an array of strings to store the animal names.

***6.34 (*Game: multiple Eight Queens solutions*) Exercise 6.22 finds one solution for the Eight Queens problem. Write a program to count all possible solutions for the Eight Queens problem and display all solutions.

CHAPTER

7

Multidimensional Arrays

Objectives

- To give examples of representing data using two-dimensional arrays (§7.1).
- To declare variables for two-dimensional arrays, create arrays, and access array elements in a two-dimensional array using row and column indexes (§7.2).
- To program common operations for two-dimensional arrays (displaying arrays, summing all elements, finding the minimum and maximum elements, and random shuffling) (§7.3).
- To pass two-dimensional arrays to methods (§7.4).
- To write a program for grading multiple-choice questions using two-dimensional arrays (§7.5).
- To solve the closest-pair problem using two-dimensional arrays (§7.6).
- To check a Sudoku solution using two-dimensional arrays (§7.7).
- To use multidimensional arrays (§7.8).

7.1 Introduction

Key Point

Data in a table or a matrix can be represented using a two-dimensional array.

problem

The preceding chapter introduced how to use one-dimensional arrays to store linear collections of elements. You can use a two-dimensional array to store a matrix or a table. For example, the following table that lists the distances between cities can be stored using a two-dimensional array named **distances**.

Distance Table (in miles)

	Chicago	Boston	New York	Atlanta	Miami	Dallas	Houston
Chicago	0	983	787	714	1375	967	1087
Boston	983	0	214	1102	1763	1723	1842
New York	787	214	0	888	1549	1548	1627
Atlanta	714	1102	888	0	661	781	810
Miami	1375	1763	1549	661	0	1426	1187
Dallas	967	1723	1548	781	1426	0	239
Houston	1087	1842	1627	810	1187	239	0

```
double[][] distances = {
  {0, 983, 787, 714, 1375, 967, 1087},
  {983, 0, 214, 1102, 1763, 1723, 1842},
  {787, 214, 0, 888, 1549, 1548, 1627},
  {714, 1102, 888, 0, 661, 781, 810},
  {1375, 1763, 1549, 661, 0, 1426, 1187},
  {967, 1723, 1548, 781, 1426, 0, 239},
  {1087, 1842, 1627, 810, 1187, 239, 0},
};
```

7.2 Two-Dimensional Array Basics

Key Point

An element in a two-dimensional array is accessed through a row and column index.

How do you declare a variable for two-dimensional arrays? How do you create a two-dimensional array? How do you access elements in a two-dimensional array? This section addresses these issues.

7.2.1 Declaring Variables of Two-Dimensional Arrays and Creating Two-Dimensional Arrays

The syntax for declaring a two-dimensional array is:

```
elementType[][] arrayRefVar;
```

or

```
elementType arrayRefVar[][]; // Allowed, but not preferred
```

As an example, here is how you would declare a two-dimensional array variable **matrix** of **int** values:

```
int[][] matrix;
```

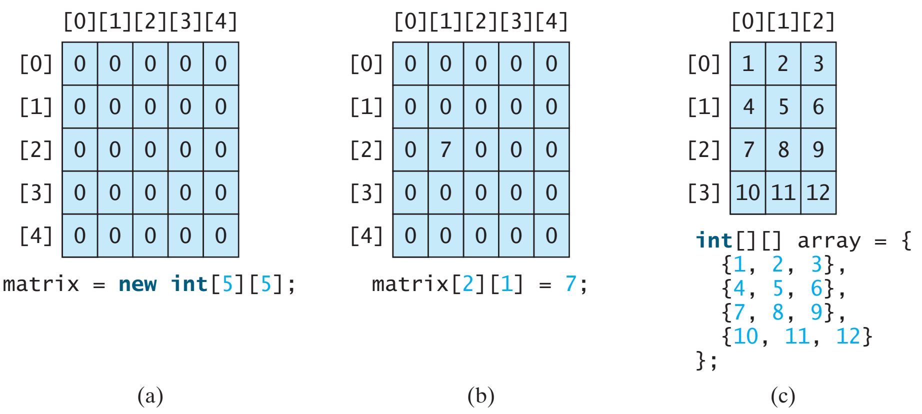
or

```
int matrix[][]; // This style is allowed, but not preferred
```

You can create a two-dimensional array of 5-by-5 `int` values and assign it to `matrix` using this syntax:

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

Two subscripts are used in a two-dimensional array, one for the row and the other for the column. As in a one-dimensional array, the index for each subscript is of the `int` type and starts from `0`, as shown in Figure 7.1a.



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

```
matrix[2][1] = 7;
```

```
int[][] array = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9},
  {10, 11, 12}
};
```

FIGURE 7.1 The index of each subscript of a two-dimensional array is an `int` value, starting from `0`.

To assign the value `7` to a specific element at row `2` and column `1`, as shown in Figure 7.1b, you can use the following syntax:

```
matrix[2][1] = 7;
```

Caution

It is a common mistake to use `matrix[2, 1]` to access the element at row `2` and column `1`. In Java, each subscript must be enclosed in a pair of square brackets.

You can also use an array initializer to declare, create, and initialize a two-dimensional array. For example, the following code in (a) creates an array with the specified initial values, as shown in Figure 7.1c. This is equivalent to the code in (b).

```
int[][] array = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9},
  {10, 11, 12}
};
```

(a)

Equivalent

```
int[][] array = new int[4][3];
array[0][0] = 1; array[0][1] = 2; array[0][2] = 3;
array[1][0] = 4; array[1][1] = 5; array[1][2] = 6;
array[2][0] = 7; array[2][1] = 8; array[2][2] = 9;
array[3][0] = 10; array[3][1] = 11; array[3][2] = 12;
```

(b)

7.2.2 Obtaining the Lengths of Two-Dimensional Arrays

A two-dimensional array is actually an array in which each element is a one-dimensional array. The length of an array `x` is the number of elements in the array, which can be obtained using `x.length`. `x[0]`, `x[1]`, . . . , and `x[x.length-1]` are arrays. Their lengths can be obtained using `x[0].length`, `x[1].length`, . . . , and `x[x.length-1].length`.

For example, suppose `x = new int[3][4]`, `x[0]`, `x[1]`, and `x[2]` are one-dimensional arrays and each contains four elements, as shown in Figure 7.2. `x.length` is `3`, and `x[0].length`, `x[1].length`, and `x[2].length` are `4`.

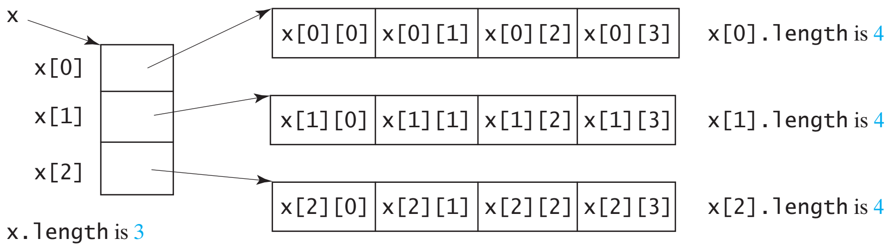

x
x[0]
x[1]
x[2]
x.length is 3
x[0][0] x[0][1] x[0][2] x[0][3]
x[1][0] x[1][1] x[1][2] x[1][3]
x[2][0] x[2][1] x[2][2] x[2][3]
x[0].length is 4
x[1].length is 4
x[2].length is 4

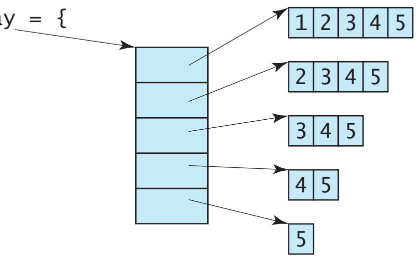
FIGURE 7.2 A two-dimensional array is a one-dimensional array in which each element is another one-dimensional array.

7.2.3 Ragged Arrays

ragged array

Each row in a two-dimensional array is itself an array. Thus, the rows can have different lengths. An array of this kind is known as a *ragged array*. Here is an example of creating a ragged array:

```
int[][] triangleArray = {
  {1, 2, 3, 4, 5},
  {2, 3, 4, 5},
  {3, 4, 5},
  {4, 5},
  {5}
};
```



As you can see, `triangleArray[0].length` is 5, `triangleArray[1].length` is 4, `triangleArray[2].length` is 3, `triangleArray[3].length` is 2, and `triangle-Array[4].length` is 1.

If you don't know the values in a ragged array in advance, but do know the sizes—say, the same as before—you can create a ragged array using the following syntax:

```
int[][] triangleArray = new int[5][];
triangleArray[0] = new int[5];
triangleArray[1] = new int[4];
triangleArray[2] = new int[3];
triangleArray[3] = new int[2];
triangleArray[4] = new int[1];
```

You can now assign values to the array. For example,

```
triangleArray[0][3] = 50;
triangleArray[4][0] = 45;
```

Note

The syntax `new int[5][]` for creating an array requires the first index to be specified. The syntax `new int[][]` would be wrong.

Check Point
MyProgrammingLab™

7.1 Declare an array reference variable for a two-dimensional array of `int` values, create a 4-by-5 `int` matrix, and assign it to the variable.

7.2 Can the rows in a two-dimensional array have different lengths?

7.3 What is the output of the following code?

```
int[][] array = new int[5][6];
int[] x = {1, 2};
array[0] = x;
System.out.println("array[0][1] is " + array[0][1]);
```

7.4 Which of the following statements are valid?

```
int[][] r = new int[2];
int[] x = new int[];
int[][] y = new int[3][];
int[][] z = {{1, 2}};
int[][] m = {{1, 2}, {2, 3}};
int[][] n = {{1, 2}, {2, 3}, };
```

7.3 Processing Two-Dimensional Arrays

Nested `for` *loops are often used to process a two-dimensional array.*

Key Point

Suppose an array `matrix` is created as follows:

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

The following are some examples of processing two-dimensional arrays.

1. *Initializing arrays with input values.* The following loop initializes the array with user input values:

```
java.util.Scanner input = new Scanner(System.in);
System.out.println("Enter " + matrix.length + " rows and " +
  matrix[0].length + " columns: ");
for (int row = 0; row < matrix.length; row++) {
  for (int column = 0; column < matrix[row].length; column++) {
    matrix[row][column] = input.nextInt();
  }
}
```

2. *Initializing arrays with random values.* The following loop initializes the array with random values between `0` and `99`:

```
for (int row = 0; row < matrix.length; row++) {
  for (int column = 0; column < matrix[row].length; column++) {
    matrix[row][column] = (int)(Math.random() * 100);
  }
}
```

3. *Printing arrays.* To print a two-dimensional array, you have to print each element in the array using a loop like the following:

```
for (int row = 0; row < matrix.length; row++) {
  for (int column = 0; column < matrix[row].length; column++) {
    System.out.print(matrix[row][column] + " ");
  }

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

4. *Summing all elements.* Use a variable named **total** to store the sum. Initially **total** is **0**. Add each element in the array to **total** using a loop like this:

```
int total = 0;
for (int row = 0; row < matrix.length; row++) {
  for (int column = 0; column < matrix[row].length; column++) {
    total += matrix[row][column];
  }
}
```

5. *Summing elements by column.* For each column, use a variable named **total** to store its sum. Add each element in the column to **total** using a loop like this:

```
for (int column = 0; column < matrix[0].length; column++) {
  int total = 0;
  for (int row = 0; row < matrix.length; row++)
    total += matrix[row][column];
  System.out.println("Sum for column " + column + " is "
    + total);
}
```

VideoNote

Find the row with the largest sum

6. *Which row has the largest sum?* Use variables **maxRow** and **indexOfMaxRow** to track the largest sum and index of the row. For each row, compute its sum and update **maxRow** and **indexOfMaxRow** if the new sum is greater.

```
int maxRow = 0;
int indexOfMaxRow = 0;

// Get sum of the first row in maxRow
for (int column = 0; column < matrix[0].length; column++) {
  maxRow += matrix[0][column];
}

for (int row = 1; row < matrix.length; row++) {
  int totalOfThisRow = 0;
  for (int column = 0; column < matrix[row].length; column++)
    totalOfThisRow += matrix[row][column];

  if (totalOfThisRow > maxRow) {
    maxRow = totalOfThisRow;
    indexOfMaxRow = row;
  }
}

System.out.println("Row " + indexOfMaxRow
  + " has the maximum sum of " + maxRow);
```

7. *Random shuffling.* Shuffling the elements in a one-dimensional array was introduced in Section 6.2.6. How do you shuffle all the elements in a two-dimensional array? To accomplish this, for each element **matrix[i][j]**, randomly generate indices **i1** and **j1** and swap **matrix[i][j]** with **matrix[i1][j1]**, as follows:

```
for (int i = 0; i < matrix.length; i++) {
  for (int j = 0; j < matrix[i].length; j++) {
    int i1 = (int)(Math.random() * matrix.length);
    int j1 = (int)(Math.random() * matrix[i].length);

    // Swap matrix[i][j] with matrix[i1][j1]
    int temp = matrix[i][j];
    matrix[i][j] = matrix[i1][j1];
    matrix[i1][j1] = temp;
  }
}
```

7.5 Show the printout of the following code:

Check Point
MyProgrammingLab™

```
int[][] array = {{1, 2}, {3, 4}, {5, 6}};
for (int i = array.length - 1; i >= 0; i--) {
  for (int j = array[i].length - 1; j >= 0; j--)
    System.out.print(array[i][j] + " ");
  System.out.println();
}
```

7.6 Show the printout of the following code:

```
int[][] array = {{1, 2}, {3, 4}, {5, 6}};
int sum = 0;
for (int i = 0; i < array.length; i++)
  sum += array[i][0];
System.out.println(sum);
```

7.4 Passing Two-Dimensional Arrays to Methods

When passing a two-dimensional array to a method, the reference of the array is passed to the method.

Key Point

You can pass a two-dimensional array to a method just as you pass a one-dimensional array. You can also return an array from a method. Listing 7.1 gives an example with two methods. The first method, `getArray()`, returns a two-dimensional array, and the second method, `sum(int[][] m)`, returns the sum of all the elements in a matrix.

LISTING 7.1 PassTwoDimensionalArray.java

```
import java.util.Scanner;

public class PassTwoDimensionalArray {
  public static void main(String[] args) {
    int[][] m = getArray(); // Get an array

    // Display sum of elements
    System.out.println("\nSum of all elements is " + sum(m));
  }

  public static int[][] getArray() {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Enter array values
    int[][] m = new int[3][4];
    System.out.println("Enter " + m.length + " rows and "
      + m[0].length + " columns: ");
    for (int i = 0; i < m.length; i++)
      for (int j = 0; j < m[i].length; j++)
        m[i][j] = input.nextInt();

    return m;
  }

  public static int sum(int[][] m) {
    int total = 0;
    for (int row = 0; row < m.length; row++) {
      for (int column = 0; column < m[row].length; column++) {
        total += m[row][column];
      }
```

get array

pass array

getArray method

return array

sum method

```
32        }
33
34        return total;
35      }
36    }
```

```
Enter 3 rows and 4 columns:
1 2 3 4 ↵Enter
5 6 7 8 ↵Enter
9 10 11 12 ↵Enter

Sum of all elements is 78
```

The method **getArray** prompts the user to enter values for the array (lines 11–24) and returns the array (line 23).

The method **sum** (lines 26–35) has a two-dimensional array argument. You can obtain the number of rows using **m.length** (line 28) and the number of columns in a specified row using **m[row].length** (line 29).

Check Point

MyProgrammingLab™

7.7 Show the printout of the following code:

```
public class Test {
  public static void main(String[] args) {
    int[][] array = {{1, 2, 3, 4}, {5, 6, 7, 8}};
    System.out.println(m1(array)[0]);
    System.out.println(m1(array)[1]);
  }

  public static int[] m1(int[][] m) {
    int[] result = new int[2];
    result[0] = m.length;
    result[1] = m[0].length;
    return result;
  }
}
```

7.5 Case Study: Grading a Multiple-Choice Test

Key Point

The problem is to write a program that will grade multiple-choice tests.

VideoNote
Grade multiple-choice test

Suppose you need to write a program that grades multiple-choice tests. Assume there are eight students and ten questions, and the answers are stored in a two-dimensional array. Each row records a student's answers to the questions, as shown in the following array.

Students' Answers to the Questions:

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

Student 0   A B A C C D E E A D
Student 1   D B A B C A E E A D
Student 2   E D D A C B E E A D
Student 3   C B A E D C E E A D
Student 4   A B D C C D E E A D
Student 5   B B E C C D E E A D
Student 6   B B A C C D E E A D
Student 7   E B E C C D E E A D
```

The key is stored in a one-dimensional array:

Key to the Questions:

	0	1	2	3	4	5	6	7	8	9
Key	D	B	D	C	C	D	A	E	A	D

Your program grades the test and displays the result. It compares each student's answers with the key, counts the number of correct answers, and displays it. Listing 7.2 gives the program.

LISTING 7.2 GradeExam.java

```
 1  public class GradeExam {
 2    /** Main method */
 3    public static void main(String[] args) {
 4      // Students' answers to the questions
 5      char[][] answers = {
 6        {'A', 'B', 'A', 'C', 'C', 'D', 'E', 'E', 'A', 'D'},
 7        {'D', 'B', 'A', 'B', 'C', 'A', 'E', 'E', 'A', 'D'},
 8        {'E', 'D', 'D', 'A', 'C', 'B', 'E', 'E', 'A', 'D'},
 9        {'C', 'B', 'A', 'E', 'D', 'C', 'E', 'E', 'A', 'D'},
10        {'A', 'B', 'D', 'C', 'C', 'D', 'E', 'E', 'A', 'D'},
11        {'B', 'B', 'E', 'C', 'C', 'D', 'E', 'E', 'A', 'D'},
12        {'B', 'B', 'A', 'C', 'C', 'D', 'E', 'E', 'A', 'D'},
13        {'E', 'B', 'E', 'C', 'C', 'D', 'E', 'E', 'A', 'D'}};
14
15      // Key to the questions
16      char[] keys = {'D', 'B', 'D', 'C', 'C', 'D', 'A', 'E', 'A', 'D'};
17
18      // Grade all answers
19      for (int i = 0; i < answers.length; i++) {
20        // Grade one student
21        int correctCount = 0;
22        for (int j = 0; j < answers[i].length; j++) {
23          if (answers[i][j] == keys[j])
24            correctCount++;
25        }
26
27        System.out.println("Student " + i + "'s correct count is " +
28          correctCount);
29      }
30    }
31  }
```

2-D array

1-D array

compare with key

```
Student 0's correct count is 7
Student 1's correct count is 6
Student 2's correct count is 5
Student 3's correct count is 4
Student 4's correct count is 8
Student 5's correct count is 7
Student 6's correct count is 7
Student 7's correct count is 7
```

The statement in lines 5–13 declares, creates, and initializes a two-dimensional array of characters and assigns the reference to **answers** of the **char[][]** type.

The statement in line 16 declares, creates, and initializes an array of **char** values and assigns the reference to **keys** of the **char[]** type.

Each row in the array **answers** stores a student's answer, which is graded by comparing it with the key in the array **keys**. The result is displayed immediately after a student's answer is graded.

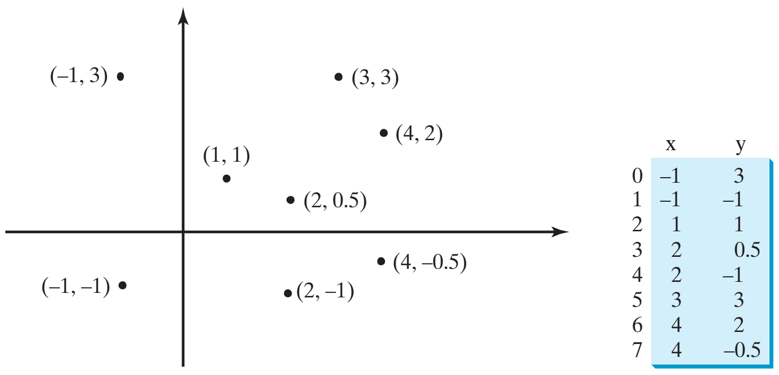
7.6 Case Study: Finding the Closest Pair

Key Point

This section presents a geometric problem for finding the closest pair of points.

closest-pair animation on the Companion Website

Given a set of points, the closest-pair problem is to find the two points that are nearest to each other. In Figure 7.3, for example, points `(1, 1)` and `(2, 0.5)` are closest to each other. There are several ways to solve this problem. An intuitive approach is to compute the distances between all pairs of points and find the one with the minimum distance, as implemented in Listing 7.3.



	x	y
0	–1	3
1	–1	–1
2	1	1
3	2	0.5
4	2	–1
5	3	3
6	4	2
7	4	–0.5

FIGURE 7.3 Points can be represented in a two-dimensional array.

LISTING 7.3 FindNearestPoints.java

```java
import java.util.Scanner;

public class FindNearestPoints {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter the number of points: ");
    int numberOfPoints = input.nextInt();

    // Create an array to store points
    double[][] points = new double[numberOfPoints][2];
    System.out.print("Enter " + numberOfPoints + " points: ");
    for (int i = 0; i < points.length; i++) {
      points[i][0] = input.nextDouble();
      points[i][1] = input.nextDouble();
    }

    // p1 and p2 are the indices in the points' array
    int p1 = 0, p2 = 1; // Initial two points
    double shortestDistance = distance(points[p1][0], points[p1][1],
      points[p2][0], points[p2][1]); // Initialize shortestDistance

    // Compute distance for every two points
    for (int i = 0; i < points.length; i++) {
```

number of points

2-D array

read points

track two points

track `shortestDistance`

for each point i

```
24         for (int j = i + 1; j < points.length; j++) {
25           double distance = distance(points[i][0], points[i][1],
26             points[j][0], points[j][1]); // Find distance
27 
28           if (shortestDistance > distance) {
29             p1 = i; // Update p1
30             p2 = j; // Update p2
31             shortestDistance = distance; // Update shortestDistance
32           }
33         }
34       }
35 
36       // Display result
37       System.out.println("The closest two points are " +
38         "(" + points[p1][0] + ", " + points[p1][1] + ") and (" +
39         points[p2][0] + ", " + points[p2][1] + ")");
40     }
41 
42     /** Compute the distance between two points (x1, y1) and (x2, y2)*/
43     public static double distance(
44         double x1, double y1, double x2, double y2) {
45       return Math.sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
46     }
47   }
```

for each point j

distance between i and j

distance between two points

update `shortestDistance`

```
Enter the number of points: 8 ↵Enter
Enter 8 points: -1 3  -1 -1  1 1  2 0.5  2 -1  3 3  4 2 4 -0.5 ↵Enter
The closest two points are (1, 1) and (2, 0.5)
```

The program prompts the user to enter the number of points (lines 6–7). The points are read from the console and stored in a two-dimensional array named **points** (lines 12–15). The program uses the variable **shortestDistance** (line 19) to store the distance between the two nearest points, and the indices of these two points in the **points** array are stored in **p1** and **p2** (line 18).

For each point at index **i**, the program computes the distance between **points[i]** and **points[j]** for all **j > i** (lines 23–34). Whenever a shorter distance is found, the variable **shortestDistance** and **p1** and **p2** are updated (lines 28–32).

The distance between two points **(x1, y1)** and **(x2, y2)** can be computed using the formula $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ (lines 43–46).

The program assumes that the plane has at least two points. You can easily modify the program to handle the case if the plane has zero or one point.

multiple closest pairs

Note that there might be more than one closest pair of points with the same minimum distance. The program finds one such pair. You may modify the program to find all closest pairs in Programming Exercise 7.8.

Tip

input file

It is cumbersome to enter all points from the keyboard. You may store the input in a file, say **FindNearestPoints.txt**, and compile and run the program using the following command:

```
java FindNearestPoints < FindNearestPoints.txt
```

7.7 Case Study: Sudoku

Key Point

The problem is to check whether a given Sudoku solution is correct.

VideoNote
Sudoku

This section presents an interesting problem of a sort that appears in the newspaper every day. It is a number-placement puzzle, commonly known as *Sudoku*. This is a very challenging problem. To make it accessible to the novice, this section presents a solution to a simplified version of the Sudoku problem, which is to verify whether a solution is correct. The complete solution for solving the Sudoku problem is presented in Supplement VI.A.

fixed cells

free cells

Sudoku is a 9 × 9 grid divided into smaller 3 × 3 boxes (also called *regions* or *blocks*), as shown in Figure 7.4a. Some cells, called *fixed cells*, are populated with numbers from **1** to **9**. The objective is to fill the empty cells, also called *free cells*, with the numbers **1** to **9** so that every row, every column, and every 3 × 3 box contains the numbers **1** to **9**, as shown in Figure 7.4b.

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

(a) Puzzle

Solution →

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

(b) Solution

FIGURE 7.4 The Sudoku puzzle in (a) is solved in (b).

representing a grid

For convenience, we use value **0** to indicate a free cell, as shown in Figure 7.5a. The grid can be naturally represented using a two-dimensional array, as shown in Figure 7.5b.

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

(a)

```
int[][] grid =
  {{5, 3, 0, 0, 7, 0, 0, 0, 0},
   {6, 0, 0, 1, 9, 5, 0, 0, 0},
   {0, 9, 8, 0, 0, 0, 0, 6, 0},
   {8, 0, 0, 0, 6, 0, 0, 0, 3},
   {4, 0, 0, 8, 0, 3, 0, 0, 1},
   {7, 0, 0, 0, 2, 0, 0, 0, 6},
   {0, 6, 0, 0, 0, 0, 2, 8, 0},
   {0, 0, 0, 4, 1, 9, 0, 0, 5},
   {0, 0, 0, 0, 8, 0, 0, 7, 9}
  };
```

(b)

FIGURE 7.5 A grid can be represented using a two-dimensional array.

To find a solution for the puzzle, we must replace each **0** in the grid with an appropriate number from **1** to **9**. For the solution to the puzzle in Figure 7.5, the grid should be as shown in Figure 7.6.

Once a solution to a Sudoku puzzle is found, how do you verify that it is correct? Here are two approaches:

- Check if every row has numbers from **1** to **9**, every column has numbers from **1** to **9**, and every small box has numbers from **1** to **9**.

- Check each cell. Each cell must be a number from **1** to **9** and the cell must be unique on every row, every column, and every small box.

```
A solution grid is
  {{5, 3, 4, 6, 7, 8, 9, 1, 2},
   {6, 7, 2, 1, 9, 5, 3, 4, 8},
   {1, 9, 8, 3, 4, 2, 5, 6, 7},
   {8, 5, 9, 7, 6, 1, 4, 2, 3},
   {4, 2, 6, 8, 5, 3, 7, 9, 1},
   {7, 1, 3, 9, 2, 4, 8, 5, 6},
   {9, 6, 1, 5, 3, 7, 2, 8, 4},
   {2, 8, 7, 4, 1, 9, 6, 3, 5},
   {3, 4, 5, 2, 8, 6, 1, 7, 9}
  };
```

FIGURE 7.6 A solution is stored in `grid`.

The program in Listing 7.4 prompts the user to enter a solution and reports whether it is valid. We use the second approach in the program to check whether the solution is correct.

LISTING 7.4 CheckSudokuSolution.java

```java
import java.util.Scanner;

public class CheckSudokuSolution {
  public static void main(String[] args) {
    // Read a Sudoku solution
    int[][] grid = readASolution();

    System.out.println(isValid(grid) ? "Valid solution" :
      "Invalid solution");
  }

  /** Read a Sudoku solution from the console */
  public static int[][] readASolution() {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    System.out.println("Enter a Sudoku puzzle solution:");
    int[][] grid = new int[9][9];
    for (int i = 0; i < 9; i++)
      for (int j = 0; j < 9; j++)
        grid[i][j] = input.nextInt();

    return grid;
  }

  /** Check whether a solution is valid */
  public static boolean isValid(int[][] grid) {
    for (int i = 0; i < 9; i++)
      for (int j = 0; j < 9; j++)
        if (grid[i][j] < 1 || grid[i][j] > 9
            || !isValid(i, j, grid))
          return false;
    return true; // The solution is valid
  }

  /** Check whether grid[i][j] is valid in the grid */
  public static boolean isValid(int i, int j, int[][] grid) {
    // Check whether grid[i][j] is valid in i's row
```

read input

solution valid?

read solution

check solution

check rows

check columns

check small boxes

```
39       for (int column = 0; column < 9; column++)
40         if (column != j && grid[i][column] == grid[i][j])
41           return false;
42
43       // Check whether grid[i][j] is valid in j's column
44       for (int row = 0; row < 9; row++)
45         if (row != i && grid[row][j] == grid[i][j])
46           return false;
47
48       // Check whether grid[i][j] is valid in the 3-by-3 box
49       for (int row = (i / 3) * 3; row < (i / 3) * 3 + 3; row++)
50         for (int col = (j / 3) * 3; col < (j / 3) * 3 + 3; col++)
51           if (row != i && col != j && grid[row][col] == grid[i][j])
52             return false;
53
54       return true; // The current value at grid[i][j] is valid
55     }
56   }
```

```
Enter a Sudoku puzzle solution:
9 6 3 1 7 4 2 5 8 ↵Enter
1 7 8 3 2 5 6 4 9 ↵Enter
2 5 4 6 8 9 7 3 1 ↵Enter
8 2 1 4 3 7 5 9 6 ↵Enter
4 9 6 8 5 2 3 1 7 ↵Enter
7 3 5 9 6 1 8 2 4 ↵Enter
5 8 9 7 1 3 4 6 2 ↵Enter
3 1 7 2 4 6 9 8 5 ↵Enter
6 4 2 5 9 8 1 7 3 ↵Enter
Valid solution
```

The program invokes the **readASolution()** method (line 6) to read a Sudoku solution and return a two-dimensional array representing a Sudoku grid.

isValid method

The **isValid(grid)** method checks whether the values in the grid are valid by verifying that each value is between **1** and **9** and that each value is valid in the grid (lines 27–34).

overloaded isValid method

The **isValid(i, j, grid)** method checks whether the value at **grid[i][j]** is valid. It checks whether **grid[i][j]** appears more than once in row **i** (lines 39–41), in column **j** (lines 44–46), and in the 3 × 3 box (lines 49–52).

How do you locate all the cells in the same box? For any **grid[i][j]**, the starting cell of the 3 × 3 box that contains it is **grid[(i / 3) * 3][(j / 3) * 3]**, as illustrated in Figure 7.7.

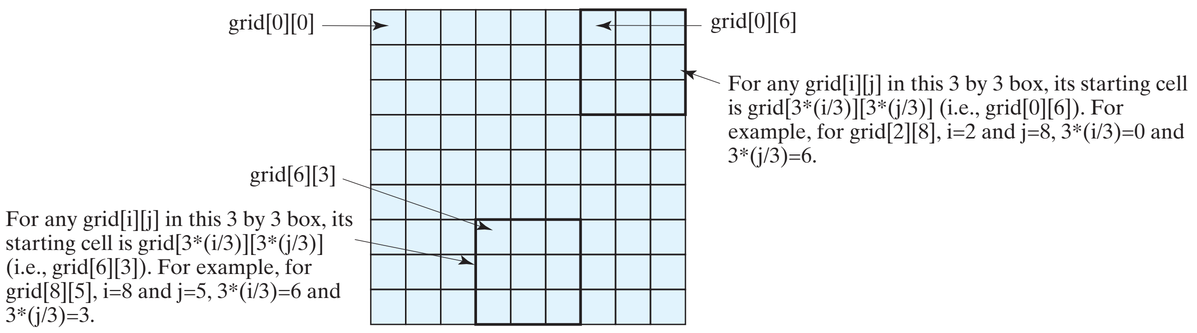

grid[0][0]
grid[0][6]
For any grid[i][j] in this 3 by 3 box, its starting cell is grid[3*(i/3)][3*(j/3)] (i.e., grid[0][6]). For example, for grid[2][8], i=2 and j=8, 3*(i/3)=0 and 3*(j/3)=6.
grid[6][3]
For any grid[i][j] in this 3 by 3 box, its starting cell is grid[3*(i/3)][3*(j/3)] (i.e., grid[6][3]). For example, for grid[8][5], i=8 and j=5, 3*(i/3)=6 and 3*(j/3)=3.

FIGURE 7.7 The location of the first cell in a 3 × 3 box determines the locations of other cells in the box.

With this observation, you can easily identify all the cells in the box. For instance, if **grid[r][c]** is the starting cell of a 3 × 3 box, the cells in the box can be traversed in a nested loop as follows:

```
// Get all cells in a 3-by-3 box starting at grid[r][c]
for (int row = r; row < r + 3; row++)
  for (int col = c; col < c + 3; col++)
    // grid[row][col] is in the box
```

input file

It is cumbersome to enter 81 numbers from the console. When you test the program, you may store the input in a file, say **CheckSudokuSolution.txt** (see www.cs.armstrong.edu/liang/data/CheckSudokuSolution.txt), and run the program using the following command:

```
java CheckSudokuSolution < CheckSudokuSolution.txt
```

7.8 Multidimensional Arrays

Key Point

A two-dimensional array consists of an array of one-dimensional arrays and a three-dimensional array consists of an array of two-dimensional arrays.

In the preceding section, you used a two-dimensional array to represent a matrix or a table. Occasionally, you will need to represent *n*-dimensional data structures. In Java, you can create *n*-dimensional arrays for any integer *n*.

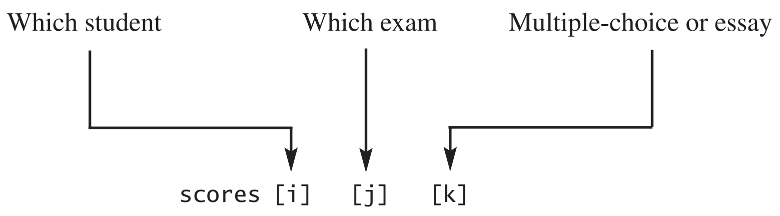
The way to declare two-dimensional array variables and create two-dimensional arrays can be generalized to declare *n*-dimensional array variables and create *n*-dimensional arrays for *n* >= 3. For example, you may use a three-dimensional array to store exam scores for a class of six students with five exams, and each exam has two parts (multiple-choice and essay). The following syntax declares a three-dimensional array variable **scores**, creates an array, and assigns its reference to **scores**.

```
double[][][] scores = new double[6][5][2];
```

You can also use the short-hand notation to create and initialize the array as follows:

```
double[][][] scores = {
  {{7.5, 20.5}, {9.0, 22.5}, {15, 33.5}, {13, 21.5}, {15, 2.5}},
  {{4.5, 21.5}, {9.0, 22.5}, {15, 34.5}, {12, 20.5}, {14, 9.5}},
  {{6.5, 30.5}, {9.4, 10.5}, {11, 33.5}, {11, 23.5}, {10, 2.5}},
  {{6.5, 23.5}, {9.4, 32.5}, {13, 34.5}, {11, 20.5}, {16, 7.5}},
  {{8.5, 26.5}, {9.4, 52.5}, {13, 36.5}, {13, 24.5}, {16, 2.5}},
  {{9.5, 20.5}, {9.4, 42.5}, {13, 31.5}, {12, 20.5}, {16, 6.5}}};
```

scores[0][1][0] refers to the multiple-choice score for the first student's second exam, which is **9.0**. **scores[0][1][1]** refers to the essay score for the first student's second exam, which is **22.5**. This is depicted in the following figure:

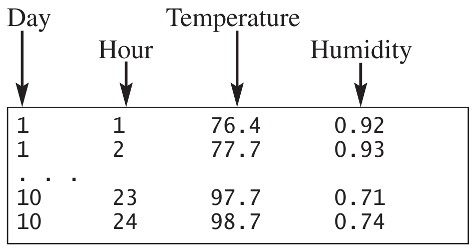


A multidimensional array is actually an array in which each element is another array. A three-dimensional array consists of an array of two-dimensional arrays. A two-dimensional array consists of an array of one-dimensional arrays. For example, suppose **x = new int[2][2][5]**, and **x[0]** and **x[1]** are two-dimensional arrays. **X[0][0]**, **x[0][1]**, **x[1][0]**, and **x[1][1]** are one-dimensional arrays and each contains five elements.

x.length is 2, x[0].length and x[1].length are 2, and X[0][0].length, x[0][1].length, x[1][0].length, and x[1][1].length are 5.

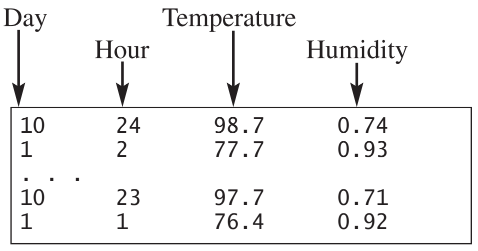
7.8.1 Case Study: Daily Temperature and Humidity

Suppose a meteorology station records the temperature and humidity every hour of every day and stores the data for the past ten days in a text file named **Weather.txt** (see www.cs.armstrong.edu/liang/data/Weather.txt). Each line of the file consists of four numbers that indicate the day, hour, temperature, and humidity. The contents of the file may look like the one in (a).



```
1       1       76.4     0.92
1       2       77.7     0.93
. . .
10      23      97.7     0.71
10      24      98.7     0.74
```

(a)

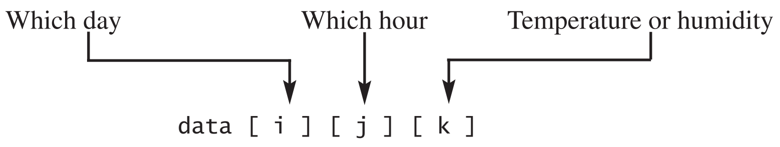


```
10      24      98.7     0.74
1       2       77.7     0.93
. . .
10      23      97.7     0.71
1       1       76.4     0.92
```

(b)

Note that the lines in the file are not necessarily in increasing order of day and hour. For example, the file may appear as shown in (b).

Your task is to write a program that calculates the average daily temperature and humidity for the 10 days. You can use the input redirection to read the file and store the data in a three-dimensional array named data. The first index of data ranges from 0 to 9 and represents 10 days, the second index ranges from 0 to 23 and represents 24 hours, and the third index ranges from 0 to 1 and represents temperature and humidity, as depicted in the following figure:



Note that the days are numbered from 1 to 10 and the hours from 1 to 24 in the file. Because the array index starts from 0, data[0][0][0] stores the temperature in day 1 at hour 1 and data[9][23][1] stores the humidity in day 10 at hour 24.

The program is given in Listing 7.5.

LISTING 7.5 Weather.java

three-dimensional array

```java
import java.util.Scanner;

public class Weather {
  public static void main(String[] args) {
    final int NUMBER_OF_DAYS = 10;
    final int NUMBER_OF_HOURS = 24;
    double[][][] data
      = new double[NUMBER_OF_DAYS][NUMBER_OF_HOURS][2];

    Scanner input = new Scanner(System.in);
    // Read input using input redirection from a file
    for (int k = 0; k < NUMBER_OF_DAYS * NUMBER_OF_HOURS; k++) {
      int day = input.nextInt();
      int hour = input.nextInt();
      double temperature = input.nextDouble();
      double humidity = input.nextDouble();
      data[day - 1][hour - 1][0] = temperature;
18        data[day - 1][hour - 1][1] = humidity;
19      }
20
21      // Find the average daily temperature and humidity
22      for (int i = 0; i < NUMBER_OF_DAYS; i++) {
23        double dailyTemperatureTotal = 0, dailyHumidityTotal = 0;
24        for (int j = 0; j < NUMBER_OF_HOURS; j++) {
25          dailyTemperatureTotal += data[i][j][0];
26          dailyHumidityTotal += data[i][j][1];
27        }
28
29        // Display result
30        System.out.println("Day " + i + "'s average temperature is "
31          + dailyTemperatureTotal / NUMBER_OF_HOURS);
32        System.out.println("Day " + i + "'s average humidity is "
33          + dailyHumidityTotal / NUMBER_OF_HOURS);
34      }
35    }
36  }
```

```
Day 0's average temperature is 77.7708
Day 0's average humidity is 0.929583
Day 1's average temperature is 77.3125
Day 1's average humidity is 0.929583
. . .
Day 9's average temperature is 79.3542
Day 9's average humidity is 0.9125
```

You can use the following command to run the program:

```
java Weather < Weather.txt
```

A three-dimensional array for storing temperature and humidity is created in line 8. The loop in lines 12–19 reads the input to the array. You can enter the input from the keyboard, but doing so will be awkward. For convenience, we store the data in a file and use input redirection to read the data from the file. The loop in lines 24–27 adds all temperatures for each hour in a day to **dailyTemperatureTotal** and all humidity for each hour to **dailyHumidityTotal**. The average daily temperature and humidity are displayed in lines 30–33.

7.8.2 Case Study: Guessing Birthdays

Listing 3.3, GuessBirthday.java, gives a program that guesses a birthday. The program can be simplified by storing the numbers in five sets in a three-dimensional array, and it prompts the user for the answers using a loop, as shown in Listing 7.6. The sample run of the program can be the same as shown in Listing 3.3.

LISTING 7.6 GuessBirthdayUsingArray.java

```
 1  import java.util.Scanner;
 2
 3  public class GuessBirthdayUsingArray {
 4    public static void main(String[] args) {
 5      int day = 0; // Day to be determined
 6      int answer;
 7
 8      int[][][] dates = {
 9        {{ 1,  3,  5,  7},
10         { 9, 11, 13, 15},
```

three-dimensional array

```
11          {17, 19, 21, 23},
12          {25, 27, 29, 31}},
13         {{ 2,  3,  6,  7},
14          {10, 11, 14, 15},
15          {18, 19, 22, 23},
16          {26, 27, 30, 31}},
17         {{ 4,  5,  6,  7},
18          {12, 13, 14, 15},
19          {20, 21, 22, 23},
20          {28, 29, 30, 31}},
21         {{ 8,  9, 10, 11},
22          {12, 13, 14, 15},
23          {24, 25, 26, 27},
24          {28, 29, 30, 31}},
25         {{16, 17, 18, 19},
26          {20, 21, 22, 23},
27          {24, 25, 26, 27},
28          {28, 29, 30, 31}}};
29
30       // Create a Scanner
31       Scanner input = new Scanner(System.in);
32
33       for (int i = 0; i < 5; i++) {
```

Set i

```
34         System.out.println("Is your birthday in Set" + (i + 1) + "?");
35         for (int j = 0; j < 4; j++) {
36           for (int k = 0; k < 4; k++)
37             System.out.printf("%4d", dates[i][j][k]);
38           System.out.println();
39         }
40
41         System.out.print("\nEnter 0 for No and 1 for Yes: ");
42         answer = input.nextInt();
43
44         if (answer == 1)
```

add to day

```
45           day += dates[i][0][0];
46       }
47
48       System.out.println("Your birthday is " + day);
49     }
50   }
```

A three-dimensional array **dates** is created in Lines 8–28. This array stores five sets of numbers. Each set is a 4-by-4 two-dimensional array.

The loop starting from line 33 displays the numbers in each set and prompts the user to answer whether the birthday is in the set (lines 41–42). If the day is in the set, the first number (**dates[i][0][0]**) in the set is added to variable **day** (line 45).

Check Point

MyProgrammingLab™

7.8 Declare an array variable for a three-dimensional array, create a 4 × 6 × 5 **int** array, and assign its reference to the variable.

7.9 Assume **int[][][] x = new char[12][5][2]**, how many elements are in the array? What are **x.length**, **x[2].length**, and **x[0][0].length**?

7.10 Show the printout of the following code:

```
int[][][] array = {{{1, 2}, {3, 4}}, {{5, 6},{7, 8}}};
System.out.println(array[0][0][0]);
System.out.println(array[1][1][1]);
```

Chapter Summary

1. A two-dimensional array can be used to store a table.
2. A variable for two-dimensional arrays can be declared using the syntax: `elementType[][] arrayVar`.
3. A two-dimensional array can be created using the syntax: `new elementType [ROW_SIZE][COLUMN_SIZE]`.
4. Each element in a two-dimensional array is represented using the syntax: `arrayVar[rowIndex][columnIndex]`.
5. You can create and initialize a two-dimensional array using an array initializer with the syntax: `elementType[][] arrayVar = {{row values}, . . . , {row values}}`.
6. You can use arrays of arrays to form multidimensional arrays. For example, a variable for three-dimensional arrays can be declared as `elementType[][][] arrayVar`, and a three-dimensional array can be created using `new elementType[size1][size2] [size3]`.

Test Questions

Do the test questions for this chapter online at www.cs.armstrong.edu/liang/intro9e/test.html.

Programming Exercises

MyProgrammingLab™

***7.1** (*Sum elements column by column*) Write a method that returns the sum of all the elements in a specified column in a matrix using the following header:

```
public static double sumColumn(double[][] m, int columnIndex)
```

Write a test program that reads a 3-by-4 matrix and displays the sum of each column. Here is a sample run:

```
Enter a 3-by-4 matrix row by row:
1.5 2 3 4 ↵Enter
5.5 6 7 8 ↵Enter
9.5 1 3 1 ↵Enter
Sum of the elements at column 0 is 16.5
Sum of the elements at column 1 is 9.0
Sum of the elements at column 2 is 13.0
Sum of the elements at column 3 is 13.0
```

***7.2** (*Sum the major diagonal in a matrix*) Write a method that sums all the numbers in the major diagonal in an $n \times n$ matrix of integers using the following header:

```
public static double sumMajorDiagonal(double[][] m)
```

Write a test program that reads a 4-by-4 matrix and displays the sum of all its elements on the major diagonal. Here is a sample run:

```
Enter a 4-by-4 matrix row by row:
1 2 3 4.0 ↵Enter
5 6.5 7 8 ↵Enter
9 10 11 12 ↵Enter
13 14 15 16 ↵Enter
Sum of the elements in the major diagonal is 34.5
```

***7.3** (*Sort students on grades*) Rewrite Listing 7.2, GradeExam.java, to display the students in increasing order of the number of correct answers.

****7.4** (*Compute the weekly hours for each employee*) Suppose the weekly hours for all employees are stored in a two-dimensional array. Each row records an employee's seven-day work hours with seven columns. For example, the following array stores the work hours for eight employees. Write a program that displays employees and their total hours in decreasing order of the total hours.

	Su	M	T	W	Th	F	Sa
Employee 0	2	4	3	4	5	8	8
Employee 1	7	3	4	3	3	4	4
Employee 2	3	3	4	3	3	2	2
Employee 3	9	3	4	7	3	4	1
Employee 4	3	5	4	3	6	3	8
Employee 5	3	4	4	6	3	4	4
Employee 6	3	7	4	8	3	8	4
Employee 7	6	3	5	9	2	7	9

7.5 (*Algebra: add two matrices*) Write a method to add two matrices. The header of the method is as follows:

```
public static double[][] addMatrix(double[][] a, double[][] b)
```

In order to be added, the two matrices must have the same dimensions and the same or compatible types of elements. Let **c** be the resulting matrix. Each element c_{ij} is $a_{ij} + b_{ij}$. For example, for two 3×3 matrices **a** and **b**, **c** is

VideoNote
Multiply two matrices

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} a_{11}+b_{11} & a_{12}+b_{12} & a_{13}+b_{13} \\ a_{21}+b_{21} & a_{22}+b_{22} & a_{23}+b_{23} \\ a_{31}+b_{31} & a_{32}+b_{32} & a_{33}+b_{33} \end{pmatrix}$$

Write a test program that prompts the user to enter two 3×3 matrices and displays their sum. Here is a sample run:

```
Enter matrix1: 1 2 3 4 5 6 7 8 9 ↵Enter
Enter matrix2: 0 2 4 1 4.5 2.2 1.1 4.3 5.2 ↵Enter
The matrices are added as follows
 1.0 2.0 3.0      0.0 2.0 4.0      1.0 4.0 7.0
 4.0 5.0 6.0  +   1.0 4.5 2.2  =   5.0 9.5 8.2
 7.0 8.0 9.0      1.1 4.3 5.2      8.1 12.3 14.2
```

****7.6** (*Algebra: multiply two matrices*) Write a method to multiply two matrices. The header of the method is:

```
public static double[][]
   multiplyMatrix(double[][] a, double[][] b)
```

To multiply matrix **a** by matrix **b**, the number of columns in **a** must be the same as the number of rows in **b**, and the two matrices must have elements of the same or compatible types. Let **c** be the result of the multiplication. Assume the column size of matrix a is **n**. Each element c_{ij} is $a_{i1} \times b_{1j} + a_{i2} \times b_{2j} + \ldots + a_{in} \times b_{nj}$. For example, for two 3 $\times$ 3 matrices **a** and **b**, **c** is

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix}$$

where $c_{ij} = a_{i1} \times b_{1j} + a_{i2} \times b_{2j} + a_{i3} \times b_{3j}$.

Write a test program that prompts the user to enter two 3 $\times$ 3 matrices and displays their product. Here is a sample run:

```
Enter matrix1: 1 2 3 4 5 6 7 8 9 ↵Enter
Enter matrix2: 0 2 4 1 4.5 2.2 1.1 4.3 5.2 ↵Enter
The multiplication of the matrices is
 1 2 3      0 2.0 4.0      5.3 23.9 24
 4 5 6   *  1 4.5 2.2  =   11.6 56.3 58.2
 7 8 9      1.1 4.3 5.2    17.9 88.7 92.4
```

***7.7** (*Points nearest to each other*) Listing 7.3 gives a program that finds two points in a two-dimensional space nearest to each other. Revise the program so that it finds two points in a three-dimensional space nearest to each other. Use a two-dimensional array to represent the points. Test the program using the following points:

```
double[][] points = {{-1, 0, 3}, {-1, -1, -1}, {4, 1, 1},
  {2, 0.5, 9}, {3.5, 2, -1}, {3, 1.5, 3}, {-1.5, 4, 2},
  {5.5, 4, -0.5}};
```

The formula for computing the distance between two points **(x1, y1, z1)** and **(x2, y2, z2)** is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$.

****7.8** (*All closest pairs of points*) Revise Listing 7.3, FindNearestPoints.java, to find all closest pairs of points with the same minimum distance.

*****7.9** (*Game: play a tic-tac-toe game*) In a game of tic-tac-toe, two players take turns marking an available cell in a 3 $\times$ 3 grid with their respective tokens (either X or O). When one player has placed three tokens in a horizontal, vertical, or diagonal row on the grid, the game is over and that player has won. A draw (no winner) occurs when all the cells on the grid have been filled with tokens and neither player has achieved a win. Create a program for playing tic-tac-toe.

The program prompts two players to enter an X token and O token alternately. Whenever a token is entered, the program redisplays the board on the console and determines the status of the game (win, draw, or continue). Here is a sample run:

```
-------------
|   |   |   |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
Enter a row (0, 1, or 2) for player X: 1 ↵Enter
Enter a column (0, 1, or 2) for player X: 1 ↵Enter

-------------
|   |   |   |
-------------
|   | X |   |
-------------
|   |   |   |
-------------
Enter a row (0, 1, or 2) for player O: 1 ↵Enter
Enter a column (0, 1, or 2) for player O: 2 ↵Enter

-------------
|   |   |   |
-------------
|   | X | O |
-------------
|   |   |   |
-------------
Enter a row (0, 1, or 2) for player X:

 . . .

-------------
| X |   |   |
-------------
| O | X | O |
-------------
|   |   | X |
-------------
X player won
```

***7.10** (*Largest row and column*) Write a program that randomly fills in 0s and 1s into a 4-by-4 matrix, prints the matrix, and finds the first row and column with the most 1s. Here is a sample run of the program:

```
0011
0011
1101
1010
The largest row index: 2
The largest column index: 2
```

****7.11** (*Game: nine heads and tails*) Nine coins are placed in a 3-by-3 matrix with some face up and some face down. You can represent the state of the coins using a 3-by-3 matrix with values **0** (heads) and **1** (tails). Here are some examples:

```
0 0 0    1 0 1    1 1 0    1 0 1    1 0 0
0 1 0    0 0 1    1 0 0    1 1 0    1 1 1
0 0 0    1 0 0    0 0 1    1 0 0    1 1 0
```

Each state can also be represented using a binary number. For example, the preceding matrices correspond to the numbers

```
000010000 101001100 110100001 101110100 100111110
```

There are a total of 512 possibilities, so you can use decimal numbers 0, 1, 2, 3, . . . , and 511 to represent all states of the matrix. Write a program that prompts the user to enter a number between 0 and 511 and displays the corresponding matrix with the characters H and T. Here is a sample run:

```
Enter a number between 0 and 511: 7 ↵Enter
H H H
H H H
T T T
```

The user entered 7, which corresponds to 000000111. Since 0 stands for H and 1 for T, the output is correct.

**7.12 (*Financial application: compute tax*) Rewrite Listing 3.6, ComputeTax.java, using arrays. For each filing status, there are six tax rates. Each rate is applied to a certain amount of taxable income. For example, from the taxable income of $400,000 for a single filer, $8,350 is taxed at 10%, (33,950 – 8,350) at 15%, (82,250 – 33,950) at 25%, (171,550 – 82,550) at 28%, (372,550 – 82,250) at 33%, and (400,000 – 372,950) at 36%. The six rates are the same for all filing statuses, which can be represented in the following array:

```
double[] rates = {0.10, 0.15, 0.25, 0.28, 0.33, 0.35};
```

The brackets for each rate for all the filing statuses can be represented in a two-dimensional array as follows:

```
int[][] brackets = {
  {8350, 33950, 82250, 171550, 372950},  // Single filer
  {16700, 67900, 137050, 20885, 372950}, // Married jointly
                                         // or qualifying widow(er)
  {8350, 33950, 68525, 104425, 186475},  // Married separately
  {11950, 45500, 117450, 190200, 372950} // Head of household
};
```

Suppose the taxable income is $400,000 for single filers. The tax can be computed as follows:

```
tax = brackets[0][0] * rates[0] +
  (brackets[0][1] – brackets[0][0]) * rates[1] +
  (brackets[0][2] – brackets[0][1]) * rates[2] +
  (brackets[0][3] – brackets[0][2]) * rates[3] +
  (brackets[0][4] – brackets[0][3]) * rates[4] +
  (400000 – brackets[0][4]) * rates[5]
```

*7.13 (*Locate the largest element*) Write the following method that returns the location of the largest element in a two-dimensional array.

```
public static int[] locateLargest(double[][] a)
```

The return value is a one-dimensional array that contains two elements. These two elements indicate the row and column indices of the largest element in the two-dimensional array. Write a test program that prompts the user to enter a two-dimensional array and displays the location of the largest element in the array. Here is a sample run:

```
Enter the number of rows and columns of the array: 3 4 ↵Enter
Enter the array:
23.5 35 2 10 ↵Enter
4.5 3 45 3.5 ↵Enter
35 44 5.5 9.6 ↵Enter
The location of the largest element is at (1, 2)
```

**7.14 (*Explore matrix*) Write a program that prompts the user to enter the length of a square matrix, randomly fills in `0`s and `1`s into the matrix, prints the matrix, and finds the rows, columns, and diagonals with all `0`s or `1`s. Here is a sample run of the program:

```
Enter the size for the matrix: 4 ↵Enter
0111
0000
0100
1111
All 0s on row 1
All 1s on row 3
No same numbers on a column
No same numbers on the major diagonal
No same numbers on the sub-diagonal
```

*7.15 (*Geometry: same line?*) Programming Exercise 5.39 gives a method for testing whether three points are on the same line.

Write the following method to test whether all the points in the array `points` are on the same line.

```
public static boolean sameLine(double[][] points)
```

Write a program that prompts the user to enter five points and displays whether they are on the same line. Here are sample runs:

```
Enter five points: 3.4 2 6.5 9.5 2.3 2.3 5.5 5 -5 4 ↵Enter
The five points are not on the same line
```

```
Enter five points: 1 1 2 2 3 3 4 4 5 5 ↵Enter
The five points are on the same line
```

*7.16 (*Sort two-dimensional array*) Write a method to sort a two-dimensional array using the following header:

```
public static void sort(int m[][])
```

The method performs a primary sort on rows and a secondary sort on columns. For example, the following array

```
{{4, 2},{1, 7},{4, 5},{1, 2},{1, 1},{4, 1}}
```

will be sorted to

```
{{1, 1},{1, 2},{1, 7},{4, 1},{4, 2},{4, 5}}.
```

*****7.17** (*Financial tsunami*) Banks lend money to each other. In tough economic times, if a bank goes bankrupt, it may not be able to pay back the loan. A bank's total assets are its current balance plus its loans to other banks. The diagram in Figure 7.8 shows five banks. The banks' current balances are 25, 125, 175, 75, and 181 million dollars, respectively. The directed edge from node 1 to node 2 indicates that bank 1 lends 40 million dollars to bank 2.

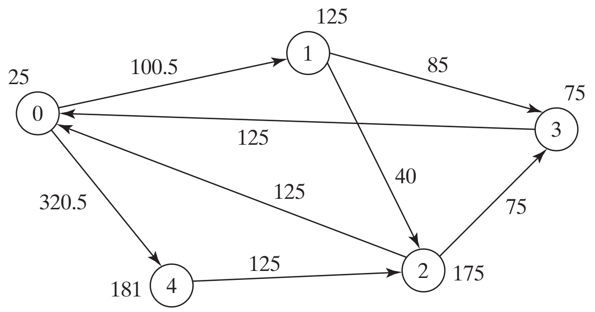


FIGURE 7.8 Banks lend money to each other.

If a bank's total assets are under a certain limit, the bank is unsafe. The money it borrowed cannot be returned to the lender, and the lender cannot count the loan in its total assets. Consequently, the lender may also be unsafe, if its total assets are under the limit. Write a program to find all the unsafe banks. Your program reads the input as follows. It first reads two integers `n` and `limit`, where `n` indicates the number of banks and `limit` is the minimum total assets for keeping a bank safe. It then reads `n` lines that describe the information for `n` banks with IDs from `0` to `n-1`.

The first number in the line is the bank's balance, the second number indicates the number of banks that borrowed money from the bank, and the rest are pairs of two numbers. Each pair describes a borrower. The first number in the pair is the borrower's ID and the second is the amount borrowed. For example, the input for the five banks in Figure 7.8 is as follows (note that the limit is 201):

```
5 201
25 2 1 100.5 4 320.5
125 2 2 40 3 85
175 2 0 125 3 75
75 1 0 125
181 1 2 125
```

The total assets of bank 3 are (75 + 125), which is under 201, so bank 3 is unsafe. After bank 3 becomes unsafe, the total assets of bank 1 fall below (125 + 40). Thus, bank 1 is also unsafe. The output of the program should be

```
Unsafe banks are 3 1
```

(*Hint*: Use a two-dimensional array `borrowers` to represent loans. `borrowers[i][j]` indicates the loan that bank `i` loans to bank `j`. Once bank `j` becomes unsafe, `borrowers[i][j]` should be set to `0`.)

***7.18** (*Shuffle rows*) Write a method that shuffles the rows in a two-dimensional `int` array using the following header:

```
public static void shuffle(int[][] m)
```

Write a test program that shuffles the following matrix:

```
int[][] m = {{1, 2}, {3, 4}, {5, 6}, {7, 8}, {9, 10}};
```

****7.19** (*Pattern recognition: four consecutive equal numbers*) Write the following method that tests whether a two-dimensional array has four consecutive numbers of the same value, either horizontally, vertically, or diagonally.

```
public static boolean isConsecutiveFour(int[][] values)
```

Write a test program that prompts the user to enter the number of rows and columns of a two-dimensional array and then the values in the array and displays true if the array contains four consecutive numbers with the same value. Otherwise, display false. Here are some examples of the true cases:

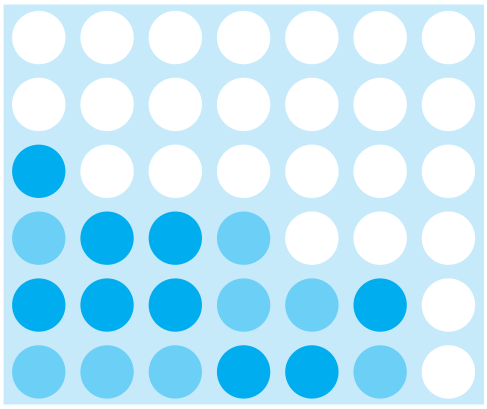
0	1	0	3	1	6	1
0	1	6	8	6	0	1
5	6	2	1	8	2	9
6	5	6	1	1	9	1
1	3	6	1	4	0	7
3	3	3	3	4	0	7

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

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

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

*****7.20** (*Game: connect four*) Connect four is a two-player board game in which the players alternately drop colored disks into a seven-column, six-row vertically suspended grid, as shown below.



The objective of the game is to connect four same-colored disks in a row, a column, or a diagonal before your opponent can do likewise. The program prompts two players to drop a red or yellow disk alternately. In the preceding figure, the red disk is shown in a dark color and the yellow in a light color. Whenever a disk is dropped, the program redisplays the board on the console and determines the status of the game (win, draw, or continue). Here is a sample run:

```
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
---------------
Drop a red disk at column (0–6): 0 ↵Enter

| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
|R| | | | | | |
---------------
Drop a yellow disk at column (0–6): 3 ↵Enter

| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
|R| | |Y| | | |

. . .
. . .
. . .

Drop a yellow disk at column (0–6): 6 ↵Enter
| | | | | | | |
| | | | | | | |
| | | |R| | | |
| | | |Y|R|Y| |
| | |R|Y|Y|Y|Y|
|R|Y|R|Y|R|R|R|
---------------
The yellow player won
```

***7.21** (*Central city*) Given a set of cities, the central point is the city that has the shortest total distance to all other cities. Write a program that prompts the user to enter the number of the cities and the locations of the cities (coordinates), and finds the central city.

```
Enter the number of cities: 5 ↵Enter
Enter the coordinates of the cities:
  2.5 5 5.1 3 1 9 5.4 54 5.5 2.1 ↵Enter
The central city is at (2.5, 5.0)
```

***7.22** (*Even number of 1s*) Write a program that generates a 6-by-6 two-dimensional matrix filled with 0s and 1s, displays the matrix, and checks if every row and every column have an even number of 1s.

VideoNote
Even number of 1s

***7.23** (*Game: find the flipped cell*) Suppose you are given a 6-by-6 matrix filled with 0s and 1s. All rows and all columns have an even number of 1s. Let the user flip one cell (i.e., flip from 1 to 0 or from 0 to 1) and write a program to find which cell was flipped. Your program should prompt the user to enter a 6-by-6 array with 0s and 1s and find the first row *r* and first column *c* where the even number of the 1s property is violated (i.e., the number of 1s is not even). The flipped cell is at (*r*, *c*).

***7.24** (*Check Sudoku solution*) Listing 7.4 checks whether a solution is valid by checking whether every number is valid in the board. Rewrite the program by checking whether every row, every column, and every small box has the numbers 1 to 9.

***7.25** (*Markov matrix*) An $n \times n$ matrix is called a *positive Markov matrix* if each element is positive and the sum of the elements in each column is 1. Write the following method to check whether a matrix is a Markov matrix.

```
public static boolean isMarkovMatrix(double[][] m)
```

Write a test program that prompts the user to enter a 3×3 matrix of double values and tests whether it is a Markov matrix. Here are sample runs:

```
Enter a 3-by-3 matrix row by row:
0.15 0.875 0.375
0.55 0.005 0.225
0.30 0.12 0.4
It is a Markov matrix
```

```
Enter a 3-by-3 matrix row by row:
0.95 -0.875 0.375
0.65 0.005 0.225
0.30 0.22 -0.4
It is not a Markov matrix
```

***7.26** (*Row sorting*) Implement the following method to sort the rows in a two-dimensional array. A new array is returned and the original array is intact.

```
public static double[][] sortRows(double[][] m)
```

Write a test program that prompts the user to enter a 3×3 matrix of double values and displays a new row-sorted matrix. Here is a sample run:

```
Enter a 3-by-3 matrix row by row:
0.15 0.875 0.375
0.55 0.005 0.225
0.30 0.12 0.4

The row-sorted array is
0.15 0.375 0.875
0.005 0.225 0.55
0.12 0.30 0.4
```

***7.27** (*Column sorting*) Implement the following method to sort the columns in a two-dimensional array. A new array is returned and the original array is intact.

```
public static double[][] sortColumns(double[][] m)
```

Write a test program that prompts the user to enter a 3 × 3 matrix of double values and displays a new column-sorted matrix. Here is a sample run:

```
Enter a 3-by-4 matrix row by row:
0.15 0.875 0.375 ↵Enter
0.55 0.005 0.225 ↵Enter
0.30 0.12 0.4 ↵Enter

The column-sorted array is
0.15 0.0050 0.225
0.3  0.12   0.375
0.55 0.875  0.4
```

7.28 (*Strictly identical arrays*) The two-dimensional arrays `m1` and `m2` are *strictly identical* if their corresponding elements are equal. Write a method that returns `true` if `m1` and `m2` are strictly identical, using the following header:

```
public static boolean equals(int[][] m1, int[][] m2)
```

Write a test program that prompts the user to enter two 3 × 3 arrays of integers and displays whether the two are strictly identical. Here are the sample runs:

```
Enter list1: 51 22 25 6 1 4 24 54 6 ↵Enter
Enter list2: 51 22 25 6 1 4 24 54 6 ↵Enter
The two arrays are strictly identical
```

```
Enter list1: 51 25 22 6 1 4 24 54 6 ↵Enter
Enter list2: 51 22 25 6 1 4 24 54 6 ↵Enter
The two arrays are not strictly identical
```

7.29 (*Identical arrays*) The two-dimensional arrays `m1` and `m2` are *identical* if they have the same contents. Write a method that returns `true` if `m1` and `m2` are identical, using the following header:

```
public static boolean equals(int[][] m1, int[][] m2)
```

Write a test program that prompts the user to enter two lists of integers and displays whether the two are identical. Here are the sample runs:

```
Enter list1: 51 25 22 6 1 4 24 54 6 ↵Enter
Enter list2: 51 22 25 6 1 4 24 54 6 ↵Enter
The two arrays are identical
```

```
Enter list1: 51 5 22 6 1 4 24 54 6 ↵Enter
Enter list2: 51 22 25 6 1 4 24 54 6 ↵Enter
The two arrays are not identical
```

***7.30** (*Algebra: solve linear equations*) Write a method that solves the following 2 × 2 system of linear equations:

$$a_{00}x + a_{01}y = b_0 \qquad x = \frac{b_0a_{11} - b_1a_{01}}{a_{00}a_{11} - a_{01}a_{10}} \qquad y = \frac{b_1a_{00} - b_0a_{10}}{a_{00}a_{11} - a_{01}a_{10}}$$
$$a_{10}x + a_{11}y = b_1$$

The method header is

```
public static double[] linearEquation(double[][] a, double[] b)
```

The method returns `null` if $a_{00}a_{11} - a_{01}a_{10}$ is `0`. Write a test program that prompts the user to enter a_{00}, a_{01}, a_{10}, a_{11}, b_0, and b_1, and displays the result. If $a_{00}a_{11} - a_{01}a_{10}$ is `0`, report that "The equation has no solution." A sample run is similar to Programming Exercise 3.3.

***7.31** (*Geometry: intersecting point*) Write a method that returns the intersecting point of two lines. The intersecting point of the two lines can be found by using the formula shown in Programming Exercise 3.25. Assume that (`x1`, `y1`) and (`x2`, `y2`) are the two points on line 1 and (`x3`, `y3`) and (`x4`, `y4`) are on line 2. The method header is

```
public static double[] getIntersectingPoint(double[][] points)
```

The points are stored in a 4-by-2 two-dimensional array `points` with (`points[0][0]`, `points[0][1]`) for (`x1`, `y1`). The method returns the intersecting point or `null` if the two lines are parallel. Write a program that prompts the user to enter four points and displays the intersecting point. See Programming Exercise 3.25 for a sample run.

***7.32** (*Geometry: area of a triangle*) Write a method that returns the area of a triangle using the following header:

```
public static double getTriangleArea(double[][] points)
```

The points are stored in a 3-by-2 two-dimensional array `points` with `points[0][0]` and `points[0][1]` for (`x1`, `y1`). The triangle area can be computed using the formula in Programming Exercise 2.15. The method returns `0` if the three points are on the same line. Write a program that prompts the user to enter two lines and displays the intersecting point. Here is a sample run of the program:

```
Enter x1, y1, x2, y2, x3, y3: 2.5 2 5 -1.0 4.0 2.0
The area of the triangle is 2.25
```

```
Enter x1, y1, x2, y2, x3, y3: 2 2 4.5 4.5 6 6
The three points are on the same line
```

***7.33** (*Geometry: polygon subareas*) A convex 4-vertex polygon is divided into four triangles, as shown in Figure 7.9.

Write a program that prompts the user to enter the coordinates of four vertices and displays the areas of the four triangles in increasing order. Here is a sample run:

```
Enter x1, y1, x2, y2, x3, y3, x4, y4:
  -2.5 2 4 4 3 -2 -2 -3.5
The areas are 6.17 7.96 8.08 10.42
```

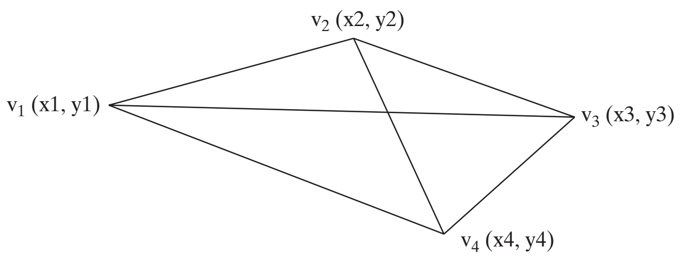


FIGURE 7.9 A 4-vertex polygon is defined by four vertices.

***7.34** (*Geometry: rightmost lowest point*) In computational geometry, often you need to find the rightmost lowest point in a set of points. Write the following method that returns the rightmost lowest point in a set of points.

```
public static double[]
       getRightmostLowestPoint(double[][] points)
```

Write a test program that prompts the user to enter the coordinates of six points and displays the rightmost lowest point. Here is a sample run:

```
Enter 6 points: 1.5 2.5 -3 4.5 5.6 -7 6.5 -7 8 1 10 2.5 ↵Enter
The rightmost lowest point is (6.5, -7.0)
```

****7.35** (*Largest block*) Given a square matrix with the elements 0 or 1, write a program to find a maximum square submatrix whose elements are all 1s. Your program should prompt the user to enter the number of rows in the matrix. The program then displays the location of the first element in the maximum square submatrix and the number of the rows in the submatrix. Here is a sample run:

```
Enter the number of rows in the matrix: 5 ↵Enter
Enter the matrix row by row:
1 0 1 0 1 ↵Enter
1 1 1 0 1 ↵Enter
1 0 1 1 1 ↵Enter
1 0 1 1 1 ↵Enter
1 0 1 1 1 ↵Enter

The maximum square submatrix is at (2, 2) with size 3
```

Your program should implement and use the following method to find the maximum square submatrix:

```
public static int[] findLargestBlock(int[][] m)
```

The return value is an array that consists of three values. The first two values are the row and column indices for the first element in the submatrix, and the third value is the number of the rows in the submatrix.

****7.36** (*Latin square*) A Latin square is an n-by-n array filled with **n** different Latin letters, each occurring exactly once in each row and once in each column. Write a program that prompts the user to enter the number n and the array of characters, as shown in the sample output, and checks if the input array is a Latin square. The characters are the first n characters starting from A.

```
Enter number n: 4 ↵Enter
Enter 4 rows of letters separated by spaces:
A B C D ↵Enter
B A D C ↵Enter
C D B A ↵Enter
D C A B ↵Enter
The input array is a Latin square
```

```
Enter number n: 3 ↵Enter
Enter 3 rows of letters separated by spaces:
A F D ↵Enter
Wrong input: the letters must be from A to C
```

CHAPTER

8

Objects and Classes

Objectives

- To describe objects and classes, and use classes to model objects (§8.2).
- To use UML graphical notation to describe classes and objects (§8.2).
- To demonstrate how to define classes and create objects (§8.3).
- To create objects using constructors (§8.4).
- To access objects via object reference variables (§8.5).
- To define a reference variable using a reference type (§8.5.1).
- To access an object's data and methods using the object member access operator (`.`) (§8.5.2).
- To define data fields of reference types and assign default values for an object's data fields (§8.5.3).
- To distinguish between object reference variables and primitive data type variables (§8.5.4).
- To use the Java library classes `Date`, `Random`, and `JFrame` (§8.6).
- To distinguish between instance and static variables and methods (§8.7).
- To define private data fields with appropriate `get` and `set` methods (§8.8).
- To encapsulate data fields to make classes easy to maintain (§8.9).
- To develop methods with object arguments and differentiate between primitive-type arguments and object-type arguments (§8.10).
- To store and process objects in arrays (§8.11).

8.1 Introduction

Key Point

Object-oriented programming enables you to develop large-scale software and GUIs effectively.

why OOP?

Having learned the material in the preceding chapters, you are able to solve many programming problems using selections, loops, methods, and arrays. However, these Java features are not sufficient for developing graphical user interfaces and large-scale software systems. Suppose you want to develop a graphical user interface (GUI, pronounced *goo-ee*) as shown in Figure 8.1. How would you program it? You will learn how in this chapter.

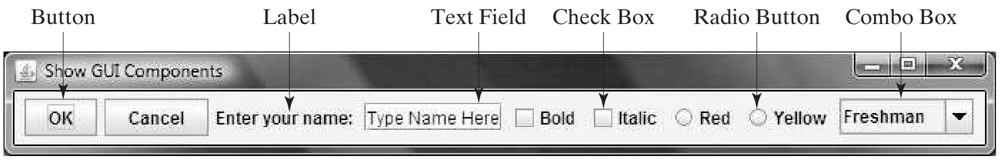


FIGURE 8.1 The GUI objects are created from classes.

This chapter introduces object-oriented programming, which you can use to develop GUI and large-scale software systems.

8.2 Defining Classes for Objects

Key Point

A class defines the properties and behaviors for objects.

VideoNote
Define classes and objects

object
state of an object
properties
attributes
data fields
behavior
actions

Object-oriented programming (OOP) involves programming using objects. An *object* represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects. An object has a unique identity, state, and behavior.

- The *state* of an object (also known as its *properties* or *attributes*) is represented by *data fields* with their current values. A circle object, for example, has a data field `radius`, which is the property that characterizes a circle. A rectangle object has the data fields `width` and `height`, which are the properties that characterize a rectangle.
- The *behavior* of an object (also known as its *actions*) is defined by methods. To invoke a method on an object is to ask the object to perform an action. For example, you may define methods named `getArea()` and `getPerimeter()` for circle objects. A circle object may invoke `getArea()` to return its area and `getPerimeter()` to return its perimeter. You may also define the `setRadius(radius)` method. A circle object can invoke this method to change its radius.

class
contract
instantiation
instance

Objects of the same type are defined using a common class. A *class* is a template, blueprint, or *contract* that defines what an object's data fields and methods will be. An object is an instance of a class. You can create many instances of a class. Creating an instance is referred to as *instantiation*. The terms *object* and *instance* are often interchangeable. The relationship between classes and objects is analogous to that between an apple-pie recipe and apple pies: You can make as many apple pies as you want from a single recipe. Figure 8.2 shows a class named `Circle` and its three objects.

data field
method
constructors

A Java class uses variables to define data fields and methods to define actions. Additionally, a class provides methods of a special type, known as *constructors*, which are invoked to create a new object. A constructor can perform any action, but constructors are designed to perform initializing actions, such as initializing the data fields of objects. Figure 8.3 shows an example of defining the class for circle objects.

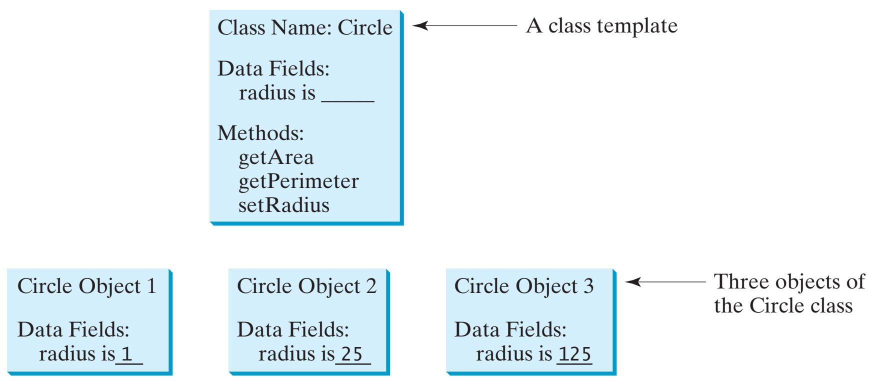
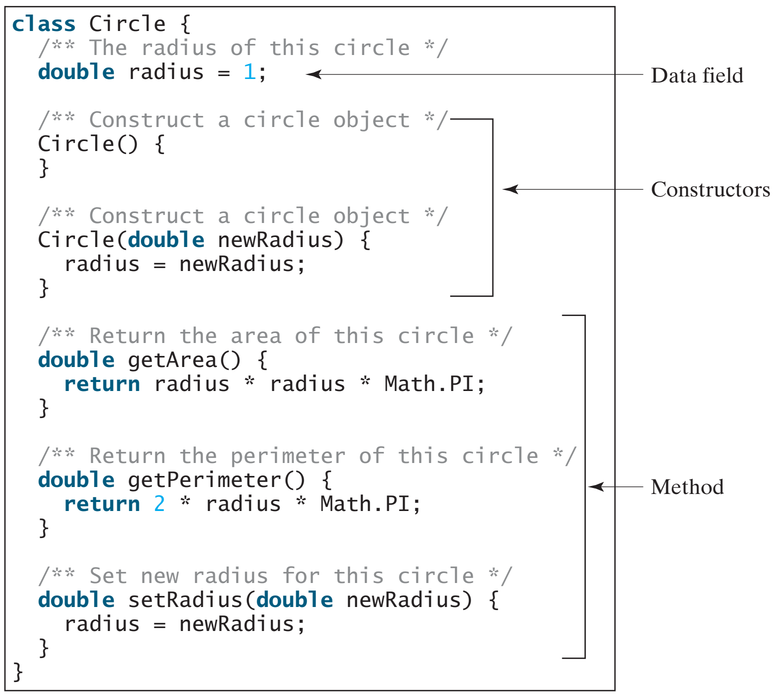


FIGURE 8.2 A class is a template for creating objects.



```
class Circle {
  /** The radius of this circle */
  double radius = 1;

  /** Construct a circle object */
  Circle() {
  }

  /** Construct a circle object */
  Circle(double newRadius) {
    radius = newRadius;
  }

  /** Return the area of this circle */
  double getArea() {
    return radius * radius * Math.PI;
  }

  /** Return the perimeter of this circle */
  double getPerimeter() {
    return 2 * radius * Math.PI;
  }

  /** Set new radius for this circle */
  double setRadius(double newRadius) {
    radius = newRadius;
  }
}
```

FIGURE 8.3 A class is a construct that defines objects of the same type.

The **Circle** class is different from all of the other classes you have seen thus far. It does not have a **main** method and therefore cannot be run; it is merely a definition for circle objects. The class that contains the **main** method will be referred to in this book, for convenience, as the *main class*.

main class

The illustration of class templates and objects in Figure 8.2 can be standardized using *Unified Modeling Language (UML)* notation. This notation, as shown in Figure 8.4, is called a *UML class diagram*, or simply a *class diagram*. In the class diagram, the data field is denoted as

Unified Modeling Language (UML)

class diagram

```
dataFieldName: dataFieldType
```

The constructor is denoted as

```
ClassName(parameterName: parameterType)
```

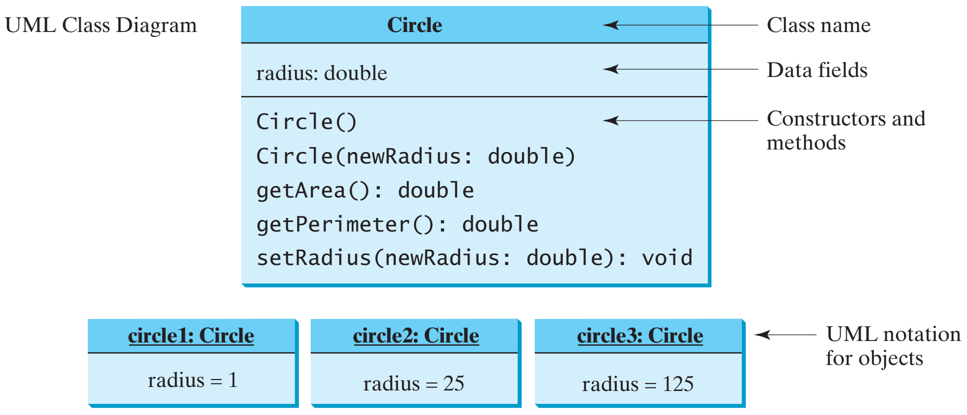


FIGURE 8.4 Classes and objects can be represented using UML notation.

The method is denoted as

```
methodName(parameterName: parameterType): returnType
```

8.3 Example: Defining Classes and Creating Objects

Key Point

Classes are definitions for objects and objects are created from classes.

This section gives two examples of defining classes and uses the classes to create objects. Listing 8.1 is a program that defines the `Circle` class and uses it to create objects. The program constructs three circle objects with radius `1`, `25`, and `125` and displays the radius and area of each of the three circles. It then changes the radius of the second object to `100` and displays its new radius and area.

avoid naming conflicts

Note

To avoid a naming conflict with several enhanced versions of the `Circle` class introduced later in the chapter, the `Circle` class in this example is named `SimpleCircle`. For simplicity, we will still refer to the class in the text as `Circle`.

LISTING 8.1 `TestSimpleCircle.java`

main class
main method
create object
create object
create object

```
public class TestSimpleCircle {
  /** Main method */
  public static void main(String[] args) {
    // Create a circle with radius 1
    SimpleCircle circle1 = new SimpleCircle();
    System.out.println("The area of the circle of radius "
      + circle1.radius + " is " + circle1.getArea());

    // Create a circle with radius 25
    SimpleCircle circle2 = new SimpleCircle(25);
    System.out.println("The area of the circle of radius "
      + circle2.radius + " is " + circle2.getArea());

    // Create a circle with radius 125
    SimpleCircle circle3 = new SimpleCircle(125);
    System.out.println("The area of the circle of radius "
      + circle3.radius + " is " + circle3.getArea());

    // Modify circle radius
    circle2.radius = 100; // or circle2.setRadius(100)
    System.out.println("The area of the circle of radius "
      + circle2.radius + " is " + circle2.getArea());
  }
}

// Define the circle class with two constructors
class SimpleCircle {
  double radius;

  /** Construct a circle with radius 1 */
  SimpleCircle() {
    radius = 1;
  }

  /** Construct a circle with a specified radius */
  SimpleCircle(double newRadius) {
    radius = newRadius;
  }

  /** Return the area of this circle */
  double getArea() {
    return radius * radius * Math.PI;
  }

  /** Return the perimeter of this circle */
  double getPerimeter() {
    return 2 * radius * Math.PI;
  }

  /** Set a new radius for this circle */
  void setRadius(double newRadius) {
    radius = newRadius;
  }
}
```

class `SimpleCircle`
data field
no-arg constructor
second constructor
`getArea`
`getPerimeter`
`setRadius`

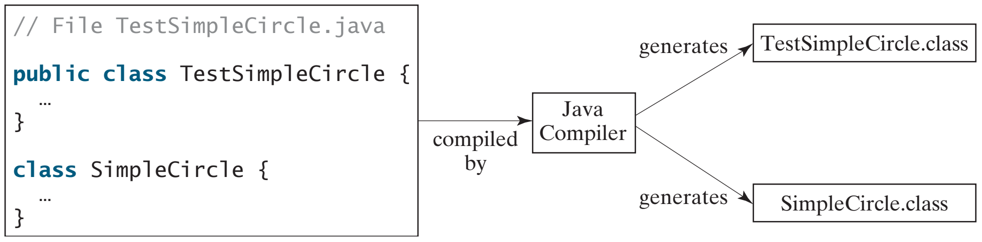
```
The area of the circle of radius 1.0 is 3.141592653589793
The area of the circle of radius 25.0 is 1963.4954084936207
The area of the circle of radius 125.0 is 49087.385212340516
The area of the circle of radius 100.0 is 31415.926535897932
```

The program contains two classes. The first of these, `TestSimpleCircle`, is the main class. Its sole purpose is to test the second class, `SimpleCircle`. Such a program that uses the class is often referred to as a *client* of the class. When you run the program, the Java runtime system invokes the `main` method in the main class.

client

You can put the two classes into one file, but only one class in the file can be a *public class*. Furthermore, the public class must have the same name as the file name. Therefore, the file name is **TestSimpleCircle.java**, since `TestSimpleCircle` is public. Each class in the source code is compiled into a **.class** file. When you compile **TestSimpleCircle.java**, two class files **TestSimpleCircle.class** and **SimpleCircle.class** are generated, as shown in Figure 8.5.

public class



```
// File TestSimpleCircle.java

public class TestSimpleCircle {
  …
}

class SimpleCircle {
  …
}
```

FIGURE 8.5 Each class in the source code file is compiled into a **.class** file.

The main class contains the **main** method (line 3) that creates three objects. As in creating an array, the **new** operator is used to create an object from the constructor. **new SimpleCircle()** creates an object with radius **1** (line 5), **new SimpleCircle(25)** creates an object with radius **25** (line 10), and **new SimpleCircle(125)** creates an object with radius **125** (line 15).

These three objects (referenced by **circle1**, **circle2**, and **circle3**) have different data but the same methods. Therefore, you can compute their respective areas by using the **getArea()** method. The data fields can be accessed via the reference of the object using **circle1.radius**, **circle2.radius**, and **circle3.radius**, respectively. The object can invoke its method via the reference of the object using **circle1.getArea()**, **circle2.getArea()**, and **circle3.getArea()**, respectively.

These three objects are independent. The radius of **circle2** is changed to **100** in line 20. The object's new radius and area are displayed in lines 21–22.

There are many ways to write Java programs. For instance, you can combine the two classes in the example into one, as shown in Listing 8.2.

LISTING 8.2 SimpleCircle.java

main method

data field

no-arg constructor

second constructor

method

```
 1 public class SimpleCircle {
 2   /** Main method */
 3   public static void main(String[] args) {
 4     // Create a circle with radius 1
 5     SimpleCircle circle1 = new SimpleCircle();
 6     System.out.println("The area of the circle of radius "
 7       + circle1.radius + " is " + circle1.getArea());
 8
 9     // Create a circle with radius 25
10     SimpleCircle circle2 = new SimpleCircle(25);
11     System.out.println("The area of the circle of radius "
12       + circle2.radius + " is " + circle2.getArea());
13
14     // Create a circle with radius 125
15     SimpleCircle circle3 = new SimpleCircle(125);
16     System.out.println("The area of the circle of radius "
17       + circle3.radius + " is " + circle3.getArea());
18
19     // Modify circle radius
20     circle2.radius = 100;
21     System.out.println("The area of the circle of radius "
22       + circle2.radius + " is " + circle2.getArea());
23   }
24
25   double radius;
26
27   /** Construct a circle with radius 1 */
28   SimpleCircle() {
29     radius = 1;
30   }
31
32   /** Construct a circle with a specified radius */
33   SimpleCircle(double newRadius) {
34     radius = newRadius;
35   }
36
37   /** Return the area of this circle */
38   double getArea() {
39     return radius * radius * Math.PI;
40   }
41
```

```
  /** Return the perimeter of this circle */
  double getPerimeter() {
    return 2 * radius * Math.PI;
  }

  /** Set a new radius for this circle */
  void setRadius(double newRadius) {
    radius = newRadius;
  }
}
```

Since the combined class has a **main** method, it can be executed by the Java interpreter. The **main** method is the same as in Listing 8.1. This demonstrates that you can test a class by simply adding a **main** method in the same class.

As another example, consider television sets. Each TV is an object with states (current channel, current volume level, power on or off) and behaviors (change channels, adjust volume, turn on/off). You can use a class to model TV sets. The UML diagram for the class is shown in Figure 8.6.

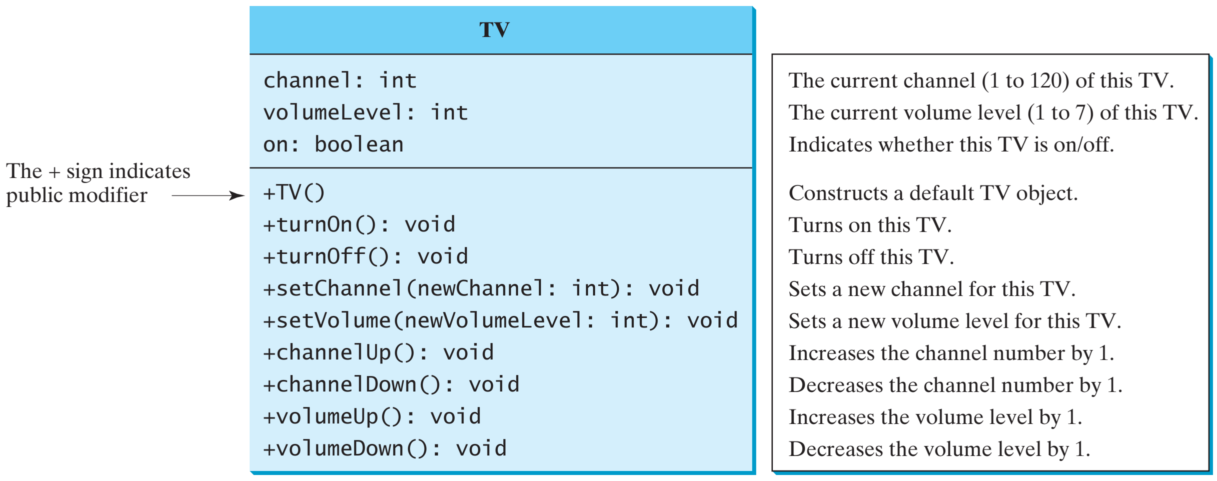

TV
channel: int
volumeLevel: int
on: boolean
+TV()
+turnOn(): void
+turnOff(): void
+setChannel(newChannel: int): void
+setVolume(newVolumeLevel: int): void
+channelUp(): void
+channelDown(): void
+volumeUp(): void
+volumeDown(): void
The + sign indicates public modifier
The current channel (1 to 120) of this TV.
The current volume level (1 to 7) of this TV.
Indicates whether this TV is on/off.
Constructs a default TV object.
Turns on this TV.
Turns off this TV.
Sets a new channel for this TV.
Sets a new volume level for this TV.
Increases the channel number by 1.
Decreases the channel number by 1.
Increases the volume level by 1.
Decreases the volume level by 1.

FIGURE 8.6 The TV class models TV sets.

Listing 8.3 gives a program that defines the **TV** class.

LISTING 8.3 TV.java

```
public class TV {
  int channel = 1; // Default channel is 1
  int volumeLevel = 1; // Default volume level is 1
  boolean on = false; // TV is off

  public TV() {
  }

  public void turnOn() {
    on = true;
  }

  public void turnOff() {
```

data fields

constructor

turn on TV

turn off TV

set a new channel
set a new volume
increase channel
decrease channel
increase volume
decrease volume

```
    on = false;
  }

  public void setChannel(int newChannel) {
    if (on && newChannel >= 1 && newChannel <= 120)
      channel = newChannel;
  }

  public void setVolume(int newVolumeLevel) {
    if (on && newVolumeLevel >= 1 && newVolumeLevel <= 7)
      volumeLevel = newVolumeLevel;
  }

  public void channelUp() {
    if (on && channel < 120)
      channel++;
  }

  public void channelDown() {
    if (on && channel > 1)
      channel--;
  }

  public void volumeUp() {
    if (on && volumeLevel < 7)
      volumeLevel++;
  }

  public void volumeDown() {
    if (on && volumeLevel > 1)
      volumeLevel--;
  }
}
```

The constructor and methods in the TV class are defined public so they can be accessed from other classes. Note that the channel and volume level are not changed if the TV is not on. Before either of these is changed, its current value is checked to ensure that it is within the correct range.

Listing 8.4 gives a program that uses the TV class to create two objects.

Listing 8.4 TestTV.java

main method
create a TV
turn on
set a new channel
set a new volume
create a TV
turn on
increase channel
increase volume
display state

```
public class TestTV {
  public static void main(String[] args) {
    TV tv1 = new TV();
    tv1.turnOn();
    tv1.setChannel(30);
    tv1.setVolume(3);

    TV tv2 = new TV();
    tv2.turnOn();
    tv2.channelUp();
    tv2.channelUp();
    tv2.volumeUp();

    System.out.println("tv1's channel is " + tv1.channel
      + " and volume level is " + tv1.volumeLevel);
    System.out.println("tv2's channel is " + tv2.channel
      + " and volume level is " + tv2.volumeLevel);
  }
}
```

```
tv1's channel is 30 and volume level is 3
tv2's channel is 3 and volume level is 2
```

The program creates two objects in lines 3 and 8 and invokes the methods on the objects to perform actions for setting channels and volume levels and for increasing channels and volumes. The program displays the state of the objects in lines 14–17. The methods are invoked using syntax such as `tv1.turnOn()` (line 4). The data fields are accessed using syntax such as `tv1.channel` (line 14).

These examples have given you a glimpse of classes and objects. You may have many questions regarding constructors, objects, reference variables, accessing data fields, and invoking object's methods. The sections that follow discuss these issues in detail.

Check Point
MyProgrammingLab™

8.1 Describe the relationship between an object and its defining class.

8.2 How do you define a class?

8.3 How do you declare an object's reference variable?

8.4 How do you create an object?

8.4 Constructing Objects Using Constructors

A constructor is invoked to create an object using the `new` *operator.*

Key Point

Constructors are a special kind of method. They have three peculiarities:

- A constructor must have the same name as the class itself. constructor's name
- Constructors do not have a return type—not even `void`. no return type
- Constructors are invoked using the `new` operator when an object is created. Constructors play the role of initializing objects. new operator

The constructor has exactly the same name as its defining class. Like regular methods, constructors can be overloaded (i.e., multiple constructors can have the same name but different signatures), making it easy to construct objects with different initial data values.

overloaded constructors

It is a common mistake to put the `void` keyword in front of a constructor. For example,

```
public void Circle() {
}
```

no void

In this case, `Circle()` is a method, not a constructor.

Constructors are used to construct objects. To construct an object from a class, invoke a constructor of the class using the `new` operator, as follows:

constructing objects

```
new ClassName(arguments);
```

For example, `new Circle()` creates an object of the `Circle` class using the first constructor defined in the `Circle` class, and `new Circle(25)` creates an object using the second constructor defined in the `Circle` class.

A class normally provides a constructor without arguments (e.g., `Circle()`). Such a constructor is referred to as a *no-arg* or *no-argument constructor*.

no-arg constructor

A class may be defined without constructors. In this case, a public no-arg constructor with an empty body is implicitly defined in the class. This constructor, called a *default constructor*, is provided automatically *only if no constructors are explicitly defined in the class*.

default constructor

Check Point
MyProgrammingLab™

8.5 What are the differences between constructors and methods?

8.6 When will a class have a default constructor?

8.5 Accessing Objects via Reference Variables

Key Point

An object's data and methods can be accessed through the dot (`.`) operator via the object's reference variable.

Newly created objects are allocated in the memory. They can be accessed via reference variables.

8.5.1 Reference Variables and Reference Types

reference variable

Objects are accessed via the object's *reference variables*, which contain references to the objects. Such variables are declared using the following syntax:

```
ClassName objectRefVar;
```

reference type

A class is essentially a programmer-defined type. A class is a *reference type*, which means that a variable of the class type can reference an instance of the class. The following statement declares the variable `myCircle` to be of the `Circle` type:

```
Circle myCircle;
```

The variable `myCircle` can reference a `Circle` object. The next statement creates an object and assigns its reference to `myCircle`:

```
myCircle = new Circle();
```

You can write a single statement that combines the declaration of an object reference variable, the creation of an object, and the assigning of an object reference to the variable with the following syntax:

```
ClassName objectRefVar = new ClassName();
```

Here is an example:

```
Circle myCircle = new Circle();
```

The variable `myCircle` holds a reference to a `Circle` object.

object vs. object reference variable

Note

An object reference variable that appears to hold an object actually contains a reference to that object. Strictly speaking, an object reference variable and an object are different, but most of the time the distinction can be ignored. Therefore, it is fine, for simplicity, to say that `myCircle` is a `Circle` object rather than use the longer-winded description that `myCircle` is a variable that contains a reference to a `Circle` object.

array object

Note

Arrays are treated as objects in Java. Arrays are created using the `new` operator. An array variable is actually a variable that contains a reference to an array.

8.5.2 Accessing an Object's Data and Methods

dot operator (.)

In OOP terminology, an object's member refers to its data fields and methods. After an object is created, its data can be accessed and its methods invoked using the *dot operator* (`.`), also known as the *object member access operator*:

- `objectRefVar.dataField` references a data field in the object.
- `objectRefVar.method(arguments)` invokes a method on the object.

For example, **myCircle.radius** references the radius in **myCircle**, and **myCircle.getArea()** invokes the **getArea** method on **myCircle**. Methods are invoked as operations on objects.

The data field **radius** is referred to as an *instance variable*, because it is dependent on a specific instance. For the same reason, the method **getArea** is referred to as an *instance method*, because you can invoke it only on a specific instance. The object on which an instance method is invoked is called a *calling object*.

instance variable

instance method

calling object

Caution

Recall that you use **Math.methodName(arguments)** (e.g., **Math.pow(3, 2.5)**) to invoke a method in the **Math** class. Can you invoke **getArea()** using **Circle.getArea()**? The answer is no. All the methods in the **Math** class are static methods, which are defined using the **static** keyword. However, **getArea()** is an instance method, and thus nonstatic. It must be invoked from an object using **objectRefVar.methodName(arguments)** (e.g., **myCircle.getArea()**). Further explanation is given in Section 8.7, Static Variables, Constants, and Methods.

invoking methods

Note

Usually you create an object and assign it to a variable, and then later you can use the variable to reference the object. Occasionally an object does not need to be referenced later. In this case, you can create an object without explicitly assigning it to a variable using the syntax:

```
new Circle();
```

or

```
System.out.println("Area is " + new Circle(5).getArea());
```

The former statement creates a **Circle** object. The latter creates a **Circle** object and invokes its **getArea** method to return its area. An object created in this way is known as an *anonymous object*.

anonymous object

8.5.3 Reference Data Fields and the null Value

The data fields can be of reference types. For example, the following **Student** class contains a data field **name** of the **String** type. **String** is a predefined Java class.

reference data fields

```
class Student {
  String name; // name has the default value null
  int age; // age has the default value 0
  boolean isScienceMajor; // isScienceMajor has default value false
  char gender; // gender has default value '\u0000'
}
```

If a data field of a reference type does not reference any object, the data field holds a special Java value, **null**. **null** is a literal just like **true** and **false**. While **true** and **false** are Boolean literals, **null** is a literal for a reference type.

null value

The default value of a data field is **null** for a reference type, **0** for a numeric type, **false** for a **boolean** type, and **\u0000** for a **char** type. However, Java assigns no default value to a local variable inside a method. The following code displays the default values of the data fields **name**, **age**, **isScienceMajor**, and **gender** for a **Student** object:

default field values

```
class Test {
  public static void main(String[] args) {
    Student student = new Student();
    System.out.println("name? " + student.name);
```

```
      System.out.println("age? " + student.age);
      System.out.println("isScienceMajor? " + student.isScienceMajor);
      System.out.println("gender? " + student.gender);
    }
  }
```

The following code has a compile error, because the local variables `x` and `y` are not initialized:

```
class Test {
  public static void main(String[] args) {
    int x; // x has no default value
    String y; // y has no default value
    System.out.println("x is " + x);
    System.out.println("y is " + y);
  }
}
```

NullPointerException

Caution

`NullPointerException` is a common runtime error. It occurs when you invoke a method on a reference variable with a `null` value. Make sure you assign an object reference to the variable before invoking the method through the reference variable.

8.5.4 Differences between Variables of Primitive Types and Reference Types

Every variable represents a memory location that holds a value. When you declare a variable, you are telling the compiler what type of value the variable can hold. For a variable of a primitive type, the value is of the primitive type. For a variable of a reference type, the value is a reference to where an object is located. For example, as shown in Figure 8.7, the value of `int` variable `i` is `int` value `1`, and the value of `Circle` object `c` holds a reference to where the contents of the `Circle` object are stored in memory.

When you assign one variable to another, the other variable is set to the same value. For a variable of a primitive type, the real value of one variable is assigned to the other variable. For a variable of a reference type, the reference of one variable is assigned to the other variable. As shown in Figure 8.8, the assignment statement `i = j` copies the contents of `j` into `i` for

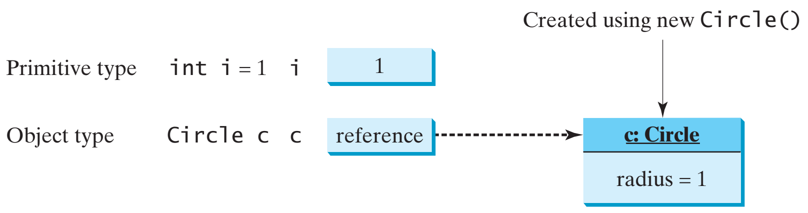


FIGURE 8.7 A variable of a primitive type holds a value of the primitive type, and a variable of a reference type holds a reference to where an object is stored in memory.

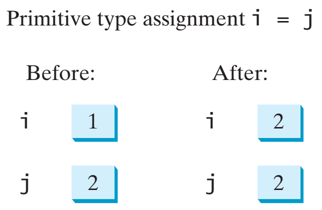


FIGURE 8.8 Primitive variable `j` is copied to variable `i`.

primitive variables. As shown in Figure 8.9, the assignment statement `c1 = c2` copies the reference of `c2` into `c1` for reference variables. After the assignment, variables `c1` and `c2` refer to the same object.

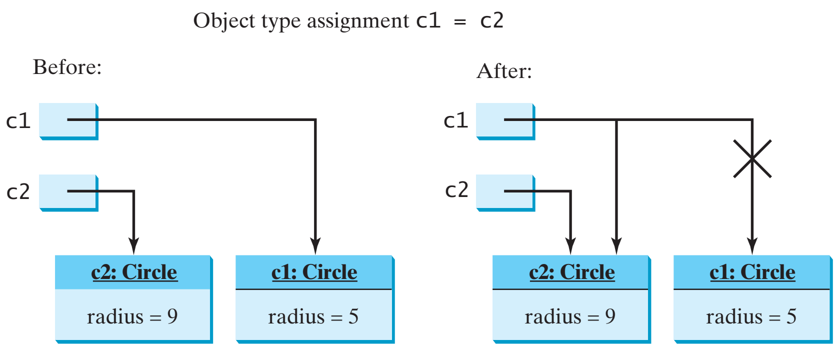


FIGURE 8.9 Reference variable `c2` is copied to variable `c1`.

Note

As illustrated in Figure 8.9, after the assignment statement `c1 = c2`, `c1` points to the same object referenced by `c2`. The object previously referenced by `c1` is no longer useful and therefore is now known as *garbage*. Garbage occupies memory space, so the Java runtime system detects garbage and automatically reclaims the space it occupies. This process is called *garbage collection*.

garbage

garbage collection

Tip

If you know that an object is no longer needed, you can explicitly assign `null` to a reference variable for the object. The JVM will automatically collect the space if the object is not referenced by any reference variable.

Check Point
MyProgrammingLab™

8.7 Which operator is used to access a data field or invoke a method from an object?

8.8 What is an anonymous object?

8.9 What is `NullPointerException`?

8.10 Is an array an object or a primitive type value? Can an array contain elements of an object type as well as a primitive type? Describe the default value for the elements of an array.

8.11 What is wrong with each of the following programs?

```java
public class ShowErrors {
  public static void main(String[] args) {
    ShowErrors t = new ShowErrors(5);
  }
}
```

(a)

```java
public class ShowErrors {
  public static void main(String[] args) {
    ShowErrors t = new ShowErrors();
    t.x();
  }
}
```

(b)

```java
public class ShowErrors {
  public void method1() {
    Circle c;
    System.out.println("What is radius "
      + c.getRadius());
    c = new Circle();
  }
}
```

(c)

```java
public class ShowErrors {
  public static void main(String[] args) {
    C c = new C(5.0);
    System.out.println(c.value);
  }
}

class C {
  int value = 2;
}
```

(d)

8.12 What is wrong in the following code?

```java
class Test {
  public static void main(String[] args) {
    A a = new A();
    a.print();
  }
}

class A {
  String s;

  A(String newS) {
    s = newS;
  }

  public void print() {
    System.out.print(s);
  }
}
```

8.13 What is the printout of the following code?

```java
public class A {
  private boolean x;

  public static void main(String[] args) {
    A a = new A();
    System.out.println(a.x);
  }
}
```

8.6 Using Classes from the Java Library

Key Point

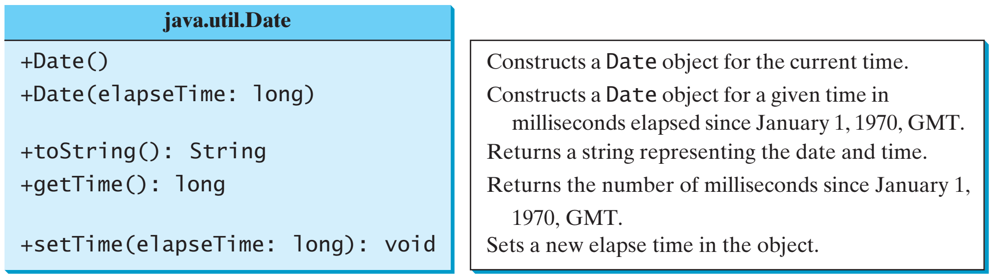
The Java API contains a rich set of classes for developing Java programs.

Listing 8.1 defined the `SimpleCircle` class and created objects from the class. You will frequently use the classes in the Java library to develop programs. This section gives some examples of the classes in the Java library.

8.6.1 The `Date` Class

In Listing 2.6, ShowCurrentTime.java, you learned how to obtain the current time using `System.currentTimeMillis()`. You used the division and remainder operators to extract the current second, minute, and hour. Java provides a system-independent encapsulation of date and time in the **java.util.Date** class, as shown in Figure 8.10.

java.util.Date class



Constructs a Date object for the current time.
Constructs a Date object for a given time in milliseconds elapsed since January 1, 1970, GMT.
Returns a string representing the date and time.
Returns the number of milliseconds since January 1, 1970, GMT.
Sets a new elapse time in the object.

FIGURE 8.10 A **Date** object represents a specific date and time.

You can use the no-arg constructor in the **Date** class to create an instance for the current date and time, the **getTime()** method to return the elapsed time since January 1, 1970, GMT, and the **toString()** method to return the date and time as a string. For example, the following code

```
java.util.Date date = new java.util.Date() ;
System.out.println("The elapsed time since Jan 1, 1970 is " +
  date.getTime()  + " milliseconds");
System.out.println(date.toString());
```

create object
get elapsed time
invoke toString

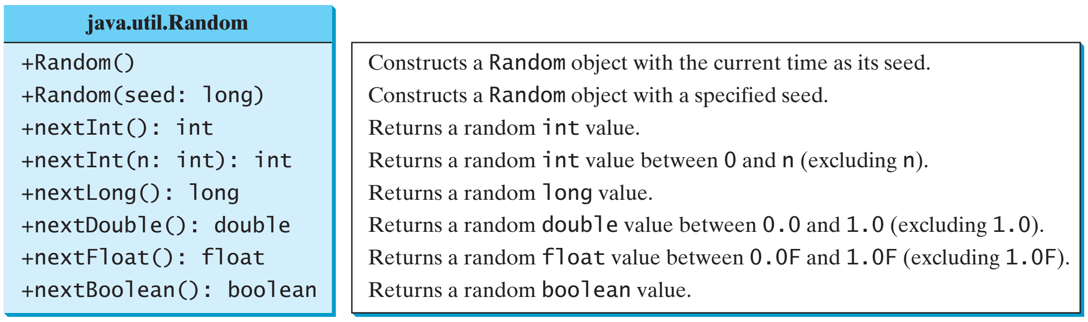
displays the output like this:

```
The elapsed time since Jan 1, 1970 is 1324903419651 milliseconds
Mon Dec 26 07:43:39 EST 2011
```

The **Date** class has another constructor, **Date(long elapseTime)**, which can be used to construct a **Date** object for a given time in milliseconds elapsed since January 1, 1970, GMT.

8.6.2 The Random Class

You have used **Math.random()** to obtain a random **double** value between **0.0** and **1.0** (excluding **1.0**). Another way to generate random numbers is to use the **java.util.Random** class, as shown in Figure 8.11, which can generate a random **int**, **long**, **double**, **float**, and **boolean** value.



Constructs a Random object with the current time as its seed.
Constructs a Random object with a specified seed.
Returns a random int value.
Returns a random int value between 0 and n (excluding n).
Returns a random long value.
Returns a random double value between 0.0 and 1.0 (excluding 1.0).
Returns a random float value between 0.0F and 1.0F (excluding 1.0F).
Returns a random boolean value.

FIGURE 8.11 A **Random** object can be used to generate random values.

When you create a **Random** object, you have to specify a seed or use the default seed. A seed is a number used to initialize a random number generator. The no-arg constructor creates a **Random** object using the current elapsed time as its seed. If two **Random** objects have the same seed, they will generate identical sequences of numbers. For example, the following code creates two **Random** objects with the same seed, **3**.

```
Random random1 = new Random(3);
System.out.print("From random1: ");
for (int i = 0; i < 10; i++)
  System.out.print(random1.nextInt(1000) + " ");

Random random2 = new Random(3);
System.out.print("\nFrom random2: ");
for (int i = 0; i < 10; i++)
  System.out.print(random2.nextInt(1000) + " ");
```

The code generates the same sequence of random **int** values:

```
From random1: 734 660 210 581 128 202 549 564 459 961
From random2: 734 660 210 581 128 202 549 564 459 961
```

same sequence

Note

The ability to generate the same sequence of random values is useful in software testing and many other applications. In software testing, often you need to reproduce the test cases from a fixed sequence of random numbers.

8.6.3 Displaying GUI Components

Pedagogical Note

Graphical user interface (GUI) components are good examples for teaching OOP. Simple GUI examples are introduced here for this purpose. The full introduction to GUI programming begins with Chapter 12, GUI Basics.

When you develop programs to create graphical user interfaces, you will use Java classes such as **JFrame**, **JButton**, **JRadioButton**, **JComboBox**, and **JList** to create frames, buttons, radio buttons, combo boxes, lists, and so on. Listing 8.5 is an example that creates two windows using the **JFrame** class. The output of the program is shown in Figure 8.12.

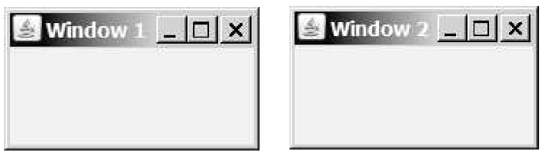


FIGURE 8.12 The program creates two windows using the **JFrame** class.

LISTING 8.5 TestFrame.java

create an object
invoke a method

```
import javax.swing.JFrame;

public class TestFrame {
  public static void main(String[] args) {
    JFrame frame1 = new JFrame();
    frame1.setTitle("Window 1");
    frame1.setSize(200, 150);
    frame1.setLocation(200, 100);
    frame1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame1.setVisible(true);
11
12       JFrame frame2 = new JFrame();
13       frame2.setTitle("Window 2");
14       frame2.setSize(200, 150);
15       frame2.setLocation(410, 100);
16       frame2.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17       frame2.setVisible(true);
18     }
19   }
```

create an object

invoke a method

This program creates two objects of the **JFrame** class (lines 5, 12) and then uses the methods **setTitle**, **setSize**, **setLocation**, **setDefaultCloseOperation**, and **setVisible** to set the properties of the objects. The **setTitle** method sets a title for the window (lines 6, 13). The **setSize** method sets the window's width and height (lines 7, 14). The **setLocation** method specifies the location of the window's upper-left corner (lines 8, 15). The **setDefaultCloseOperation** method terminates the program when the frame is closed (lines 9, 16). The **setVisible** method displays the window.

You can add graphical user interface components, such as buttons, labels, text fields, check boxes, and combo boxes to the window. The components are defined using classes. Listing 8.6 gives an example of creating a graphical user interface, as shown in Figure 8.1.

LISTING 8.6 GUIComponents.java

VideoNote
Use classes

```
import javax.swing.*;

public class GUIComponents {
  public static void main(String[] args) {
    // Create a button with text OK
    JButton jbtOK = new JButton("OK");

    // Create a button with text Cancel
    JButton jbtCancel = new JButton("Cancel");

    // Create a label with text "Enter your name: "
    JLabel jlblName = new JLabel("Enter your name: ");

    // Create a text field with text "Type Name Here"
    JTextField jtfName = new JTextField("Type Name Here");

    // Create a check box with text Bold
    JCheckBox jchkBold = new JCheckBox("Bold");

    // Create a check box with text Italic
    JCheckBox jchkItalic = new JCheckBox("Italic");

    // Create a radio button with text Red
    JRadioButton jrbRed = new JRadioButton("Red");

    // Create a radio button with text Yellow
    JRadioButton jrbYellow = new JRadioButton("Yellow");

    // Create a combo box with several choices
    JComboBox jcboColor = new JComboBox(new String[]{"Freshman",
      "Sophomore", "Junior", "Senior"});

    // Create a panel to group components
    JPanel panel = new JPanel();
    panel.add(jbtOK); // Add the OK button to the panel
    panel.add(jbtCancel); // Add the Cancel button to the panel
```

create a button

create a button

create a label

create a text field

create a check box

create a check box

create a radio button

create a radio button

create a combo box

create a panel

add to panel

```
37      panel.add(jlblName); // Add the label to the panel
38      panel.add(jtfName); // Add the text field to the panel
39      panel.add(jchkBold); // Add the check box to the panel
40      panel.add(jchkItalic); // Add the check box to the panel
41      panel.add(jrbRed); // Add the radio button to the panel
42      panel.add(jrbYellow); // Add the radio button to the panel
43      panel.add(jcboColor); // Add the combo box to the panel
44
45      JFrame frame = new JFrame(); // Create a frame
46      frame.add(panel); // Add the panel to the frame
47      frame.setTitle("Show GUI Components");
48      frame.setSize(450, 100);
49      frame.setLocation(200, 100);
50      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
51      frame.setVisible(true);
52    }
53  }
```

create a frame
add panel to frame
display frame

This program creates GUI objects using the classes `JButton`, `JLabel`, `JTextField`, `JCheckBox`, `JRadioButton`, and `JComboBox` (lines 6–31). Then, using the `JPanel` class (line 34), it then creates a panel object and adds the button, label, text field, check box, radio button, and combo box to it (lines 35–43). The program then creates a frame and adds the panel to the frame (line 45). The frame is displayed in line 51.

Check Point
MyProgrammingLab™

8.14 How do you create a `Date` for the current time? How do you display the current time?

8.15 How do you create a `JFrame`, set a title in a frame, and display a frame?

8.16 Which packages contain the classes `Date`, `JFrame`, `JOptionPane`, `System`, and `Math`?

8.7 Static Variables, Constants, and Methods

Key Point

A static variable is shared by all objects of the class. A static method cannot access instance members of the class.

VideoNote
Static vs. instance
instance variable

The data field `radius` in the circle class is known as an *instance variable*. An instance variable is tied to a specific instance of the class; it is not shared among objects of the same class. For example, suppose that you create the following objects:

```
Circle circle1 = new Circle();
Circle circle2 = new Circle(5);
```

The `radius` in `circle1` is independent of the `radius` in `circle2` and is stored in a different memory location. Changes made to `circle1`'s `radius` do not affect `circle2`'s `radius`, and vice versa.

static variable

If you want all the instances of a class to share data, use *static variables*, also known as *class variables*. Static variables store values for the variables in a common memory location. Because of this common location, if one object changes the value of a static variable, all objects of the same class are affected. Java supports static methods as well as static variables. *Static methods* can be called without creating an instance of the class.

static method

Let's modify the `Circle` class by adding a static variable `numberOfObjects` to count the number of circle objects created. When the first object of this class is created, `numberOfObjects` is `1`. When the second object is created, `numberOfObjects` becomes `2`. The UML of the new circle class is shown in Figure 8.13. The `Circle` class defines the instance variable `radius` and the static variable `numberOfObjects`, the instance methods `getRadius`, `setRadius`, and `getArea`, and the static method `getNumberOfObjects`. (Note that static variables and methods are underlined in the UML class diagram.)

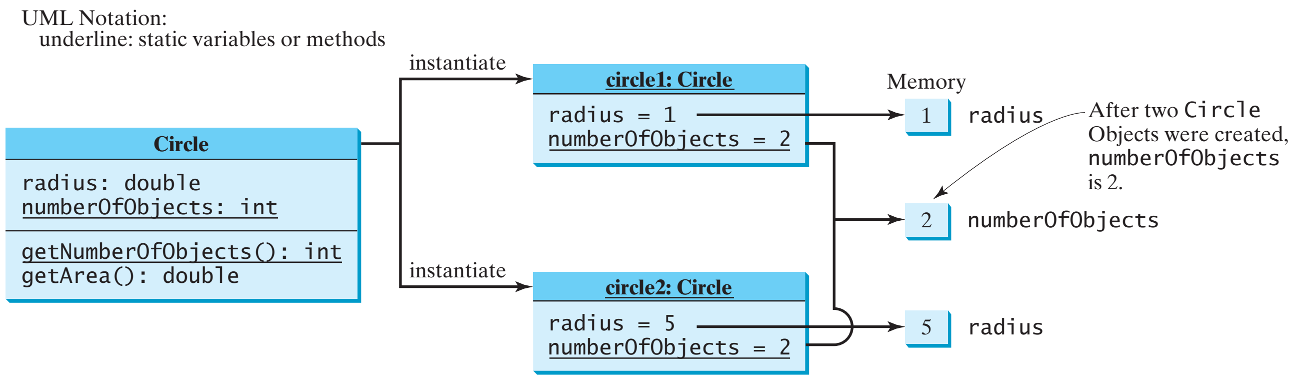


FIGURE 8.13 Instance variables belong to the instances and have memory storage independent of one another. Static variables are shared by all the instances of the same class.

To declare a static variable or define a static method, put the modifier `static` in the variable or method declaration. The static variable `numberOfObjects` and the static method `getNumberOfObjects()` can be declared as follows:

```
static int numberOfObjects;
```

declare static variable

```
static int getNumberObjects() {
  return numberOfObjects;
}
```

define static method

Constants in a class are shared by all objects of the class. Thus, constants should be declared as `final static`. For example, the constant `PI` in the `Math` class is defined as:

declare constant

```
final static double PI = 3.14159265358979323846;
```

The new circle class, named `CircleWithStaticMembers`, is defined in Listing 8.7:

LISTING 8.7 CircleWithStaticMembers.java

```
public class CircleWithStaticMembers {
  /** The radius of the circle */
  double radius;

  /** The number of objects created */
  static int numberOfObjects = 0;                       static variable

  /** Construct a circle with radius 1 */
  CircleWithStaticMembers() {
    radius = 1;
    numberOfObjects++;                                  increase by 1
  }

  /** Construct a circle with a specified radius */
  CircleWithStaticMembers(double newRadius) {
    radius = newRadius;
    numberOfObjects++;                                  increase by 1
  }

  /** Return numberOfObjects */
  static int getNumberOfObjects() {                     static method
    return numberOfObjects;
  }

  /** Return the area of this circle */
  double getArea() {
    return radius * radius * Math.PI;
  }
}
```

Method **getNumberOfObjects()** in **CircleWithStaticMembers** is a static method. Other examples of static methods are **showMessageDialog** and **showInputDialog** in the **JOptionPane** class and all the methods in the **Math** class. The **main** method is static, too.

Instance methods (e.g., **getArea()**) and instance data (e.g., **radius**) belong to instances and can be used only after the instances are created. They are accessed via a reference variable. Static methods (e.g., **getNumberOfObjects()**) and static data (e.g., **numberOfObjects**) can be accessed from a reference variable or from their class name.

The program in Listing 8.8 demonstrates how to use instance and static variables and methods and illustrates the effects of using them.

Listing 8.8 TestCircleWithStaticMembers.java

static variable

instance variable

static variable

instance variable

static variable

static variable

```
public class TestCircleWithStaticMembers {
  /** Main method */
  public static void main(String[] args) {
    System.out.println("Before creating objects");
    System.out.println("The number of Circle objects is " +
      CircleWithStaticMembers.numberOfObjects);

    // Create c1
    CircleWithStaticMembers c1 = new CircleWithStaticMembers();

    // Display c1 BEFORE c2 is created
    System.out.println("\nAfter creating c1");
    System.out.println("c1: radius (" + c1.radius +
      ") and number of Circle objects (" +
      c1.numberOfObjects + ")");

    // Create c2
    CircleWithStaticMembers c2 = new CircleWithStaticMembers(5);

    // Modify c1
    c1.radius = 9;

    // Display c1 and c2 AFTER c2 was created
    System.out.println("\nAfter creating c2 and modifying c1");
    System.out.println("c1: radius (" + c1.radius +
      ") and number of Circle objects (" +
      c1.numberOfObjects + ")");
    System.out.println("c2: radius (" + c2.radius +
      ") and number of Circle objects (" +
      c2.numberOfObjects + ")");
  }
}
```

```
Before creating objects
The number of Circle objects is 0
After creating c1
c1: radius (1.0) and number of Circle objects (1)
After creating c2 and modifying c1
c1: radius (9.0) and number of Circle objects (2)
c2: radius (5.0) and number of Circle objects (2)
```

When you compile TestCircleWithStaticMembers.java, the Java compiler automatically compiles CircleWithStaticMembers.java if it has not been compiled since the last change.

Static variables and methods can be accessed without creating objects. Line 6 displays the number of objects, which is 0, since no objects have been created.

The main method creates two circles, c1 and c2 (lines 9, 18). The instance variable radius in c1 is modified to become 9 (line 21). This change does not affect the instance variable radius in c2, since these two instance variables are independent. The static variable numberOfObjects becomes 1 after c1 is created (line 9), and it becomes 2 after c2 is created (line 18).

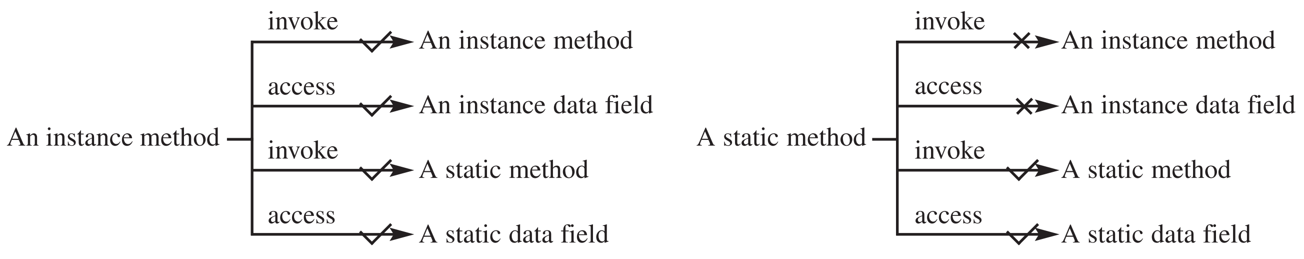
Note that PI is a constant defined in Math, and Math.PI references the constant. c1.numberOfObjects (line 27) and c2.numberOfObjects (line 30) are better replaced by CircleWithStaticMembers.numberOfObjects. This improves readability, because other programmers can easily recognize the static variable. You can also replace CircleWithStaticMembers.numberOfObjects with CircleWithStaticMembers.getNumberOfObjects().

Tip

Use ClassName.methodName(arguments) to invoke a static method and ClassName.staticVariable to access a static variable. This improves readability, because other programmers can easily recognize the static method and data in the class.

use class name

An instance method can invoke an instance or static method and access an instance or static data field. A static method can invoke a static method and access a static data field. However, a static method cannot invoke an instance method or access an instance data field, since static methods and static data fields don't belong to a particular object. The relationship between static and instance members is summarized in the following diagram:


An instance method
invoke ✓ An instance method
access ✓ An instance data field
invoke ✓ A static method
access ✓ A static data field
A static method
invoke ✗ An instance method
access ✗ An instance data field
invoke ✓ A static method
access ✓ A static data field

For example, the following code is wrong.

```
public class A {
  int i = 5;
  static int k = 2;

  public static void main(String[] args) {
    int j = i; // Wrong because i is an instance variable
    m1(); // Wrong because m1() is an instance method
  }

  public void m1() {
    // Correct since instance and static variables and methods
    // can be used in an instance method
    i = i + k + m2(i, k);
  }

  public static int m2(int i, int j) {
    return (int)(Math.pow(i, j));
  }
}
```

Note that if you replace the preceding code with the following new code, the program would be fine, because the instance data field `i` and method `m1` are now accessed from an object `a` (lines 7–8):

```
 1  public class A {
 2    int i = 5;
 3    static int k = 2;
 4
 5    public static void main(String[] args) {
 6      A a = new A();
 7      int j = a.i; // OK, a.i accesses the object's instance variable
 8      a.m1(); // OK. a.m1() invokes the object's instance method
 9    }
10
11    public void m1() {
12      i = i + k + m2(i, k);
13    }
14
15    public static int m2(int i, int j) {
16      return (int)(Math.pow(i, j));
17    }
18  }
```

instance or static?

Design Guide

How do you decide whether a variable or method should be an instance one or a static one? A variable or method that is dependent on a specific instance of the class should be an instance variable or method. A variable or method that is not dependent on a specific instance of the class should be a static variable or method. For example, every circle has its own radius, so the radius is dependent on a specific circle. Therefore, `radius` is an instance variable of the `Circle` class. Since the `getArea` method is dependent on a specific circle, it is an instance method. None of the methods in the `Math` class, such as `random`, `pow`, `sin`, and `cos`, is dependent on a specific instance. Therefore, these methods are static methods. The `main` method is static and can be invoked directly from a class.

common design error

Caution

It is a common design error to define an instance method that should have been defined as static. For example, the method `factorial(int n)` should be defined as static, as shown next, because it is independent of any specific instance.

```
public class Test {
  public int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
      result *= i;

    return result;
  }
}
```

(a) Wrong design

```
public class Test {
  public static int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
      result *= i;

    return result;
  }
}
```

(b) Correct design

Check Point
MyProgrammingLab™

8.17 Suppose that the class `F` is defined in (a). Let `f` be an instance of `F`. Which of the statements in (b) are correct?

```
public class F {
  int i;
  static String s;

  void imethod() {
  }

  static void smethod() {
  }
}
```

(a)

```
System.out.println(f.i);
System.out.println(f.s);
f.imethod();
f.smethod();
System.out.println(F.i);
System.out.println(F.s);
F.imethod();
F.smethod();
```

(b)

8.18 Add the `static` keyword in the place of `?` if appropriate.

```
public class Test {
  private int count;

  public ? void main(String[] args) {
    ...
  }

  public ? int getCount() {
    return count;
  }

  public ? int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
      result *= i;

    return result;
  }
}
```

8.19 Can you invoke an instance method or reference an instance variable from a static method? Can you invoke a static method or reference a static variable from an instance method? What is wrong in the following code?

```
public class C {
  public static void main(String[] args) {
    method1();
  }

  public void method1() {
    method2();
  }

  public static void method2() {
    System.out.println("What is radius " + c.getRadius());
  }

  Circle c = new Circle();
}
```

8.8 Visibility Modifiers

Visibility modifiers can be used to specify the visibility of a class and its members.

Key Point

You can use the `public` visibility modifier for classes, methods, and data fields to denote that they can be accessed from any other classes. If no visibility modifier is used, then by default the classes, methods, and data fields are accessible by any class in the same package. This is known as *package-private* or *package-access*.

package-private (or package-access)

using packages

Note

Packages can be used to organize classes. To do so, you need to add the following line as the first noncomment and nonblank statement in the program:

```
package packageName;
```

If a class is defined without the package statement, it is said to be placed in the *default package*.

Java recommends that you place classes into packages rather using a default package. For simplicity, however, this book uses default packages. For more information on packages, see Supplement III.G, Packages.

In addition to the `public` and default visibility modifiers, Java provides the `private` and `protected` modifiers for class members. This section introduces the `private` modifier. The `protected` modifier will be introduced in Section 11.13, The `protected` Data and Methods.

The `private` modifier makes methods and data fields accessible only from within its own class. Figure 8.14 illustrates how a public, default, and private data field or method in class `C1` can be accessed from a class `C2` in the same package and from a class `C3` in a different package.

```
package p1;

public class C1 {
  public int x;
  int y;
  private int z;

  public void m1() {
  }
  void m2() {
  }
  private void m3() {
  }
}
```

```
package p1;

public class C2 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    can access o.y;
    cannot access o.z;

    can invoke o.m1();
    can invoke o.m2();
    cannot invoke o.m3();
  }
}
```

```
package p2;

public class C3 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    cannot access o.y;
    cannot access o.z;

    can invoke o.m1();
    cannot invoke o.m2();
    cannot invoke o.m3();
  }
}
```

FIGURE 8.14 The private modifier restricts access to its defining class, the default modifier restricts access to a package, and the public modifier enables unrestricted access.

If a class is not defined as public, it can be accessed only within the same package. As shown in Figure 8.15, `C1` can be accessed from `C2` but not from `C3`.

```
package p1;

class C1 {
  ...
}
```

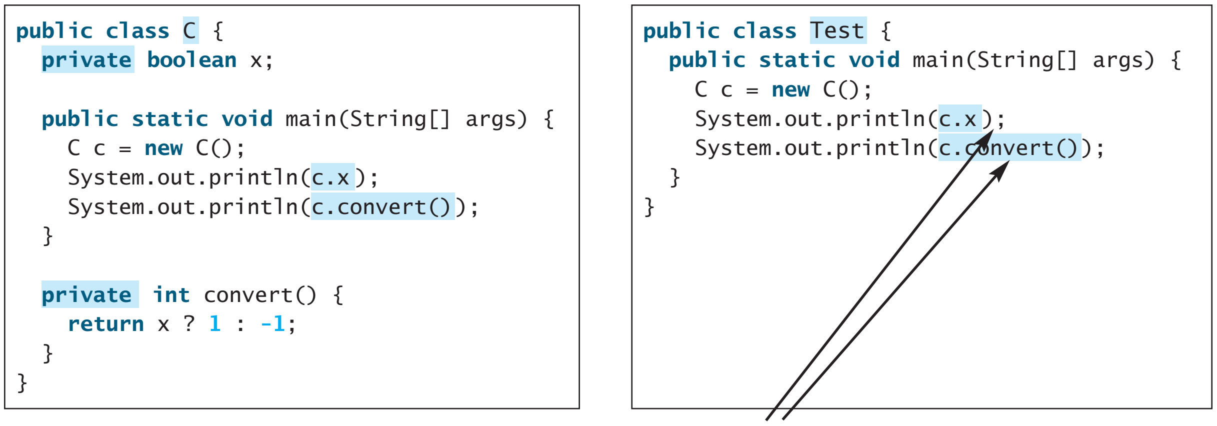
```
package p1;

public class C2 {
  can access C1
}
```

```
package p2;

public class C3 {
  cannot access C1;
  can access C2;
}
```

FIGURE 8.15 A nonpublic class has package-access.

inside access

A visibility modifier specifies how data fields and methods in a class can be accessed from outside the class. There is no restriction on accessing data fields and methods from inside the class. As shown in Figure 8.16b, an object `c` of class `C` cannot access its private members, because `c` is in the `Test` class. As shown in Figure 8.16a, an object `c` of class `C` can access its private members, because `c` is defined inside its own class.



```
public class C {
  private boolean x;

  public static void main(String[] args) {
    C c = new C();
    System.out.println(c.x);
    System.out.println(c.convert());
  }

  private int convert() {
    return x ? 1 : -1;
  }
}
```

(a) This is okay because object **c** is used inside the class **C**.

```
public class Test {
  public static void main(String[] args) {
    C c = new C();
    System.out.println(c.x);
    System.out.println(c.convert());
  }
}
```

(b) This is wrong because **x** and **convert** are private in class **C**.

FIGURE 8.16 An object can access its private members if it is defined in its own class.

Caution

The **private** modifier applies only to the members of a class. The **public** modifier can apply to a class or members of a class. Using the modifiers **public** and **private** on local variables would cause a compile error.

Note

private constructor

In most cases, the constructor should be public. However, if you want to prohibit the user from creating an instance of a class, use a *private constructor*. For example, there is no reason to create an instance from the **Math** class, because all of its data fields and methods are static. To prevent the user from creating objects from the **Math** class, the constructor in **java.lang.Math** is defined as follows:

```
private Math() {
}
```

8.9 Data Field Encapsulation

Key Point

Making data fields private protects data and makes the class easy to maintain.

VideoNote
Data field encapsulation

The data fields **radius** and **numberOfObjects** in the **CircleWithStaticMembers** class in Listing 8.7 can be modified directly (e.g., **c1.radius = 5** or **CircleWithStaticMembers.numberOfObjects = 10**). This is not a good practice—for two reasons:

- First, data may be tampered with. For example, **numberOfObjects** is to count the number of objects created, but it may be mistakenly set to an arbitrary value (e.g., **CircleWithStaticMembers.numberOfObjects = 10**).
- Second, the class becomes difficult to maintain and vulnerable to bugs. Suppose you want to modify the **CircleWithStaticMembers** class to ensure that the radius is nonnegative after other programs have already used the class. You have to change not only the **CircleWithStaticMembers** class but also the programs that use it, because the clients may have modified the radius directly (e.g., **c1.radius = -5**).

To prevent direct modifications of data fields, you should declare the data fields private, using the **private** modifier. This is known as *data field encapsulation*.

data field encapsulation

A private data field cannot be accessed by an object from outside the class that defines the private field. However, a client often needs to retrieve and modify a data field. To make a private data field accessible, provide a *get* method to return its value. To enable a private data field to be updated, provide a *set* method to set a new value.

getter (or accessor)
setter (or mutator)

Note

Colloquially, a **get** method is referred to as a *getter* (or *accessor*), and a **set** method is referred to as a *setter* (or *mutator*).

A **get** method has the following signature:

```
public returnType getPropertyName()
```

boolean accessor

If the **returnType** is **boolean**, the **get** method should be defined as follows by convention:

```
public boolean isPropertyName()
```

A **set** method has the following signature:

```
public void setPropertyName(dataType propertyValue)
```

Let's create a new circle class with a private data-field radius and its associated accessor and mutator methods. The class diagram is shown in Figure 8.17. The new circle class, named **CircleWithPrivateDataFields**, is defined in Listing 8.9:

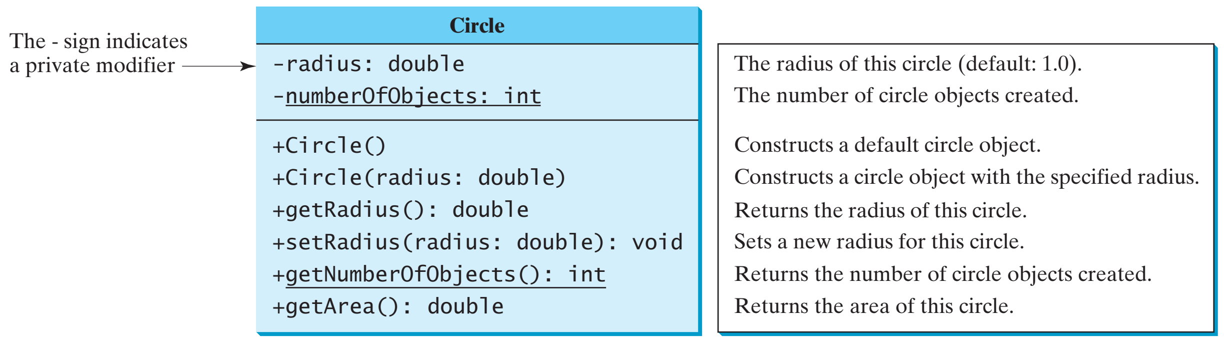

The - sign indicates a private modifier
Circle
-radius: double
-numberOfObjects: int
+Circle()
+Circle(radius: double)
+getRadius(): double
+setRadius(radius: double): void
+getNumberOfObjects(): int
+getArea(): double
The radius of this circle (default: 1.0).
The number of circle objects created.
Constructs a default circle object.
Constructs a circle object with the specified radius.
Returns the radius of this circle.
Sets a new radius for this circle.
Returns the number of circle objects created.
Returns the area of this circle.

FIGURE 8.17 The **Circle** class encapsulates circle properties and provides get/set and other methods.

LISTING 8.9 CircleWithPrivateDataFields.java

encapsulate radius

encapsulate numberOfObjects

```
public class CircleWithPrivateDataFields {
  /** The radius of the circle */
  private double radius = 1;

  /** The number of objects created */
  private static int numberOfObjects = 0;

  /** Construct a circle with radius 1 */
  public CircleWithPrivateDataFields() {
    numberOfObjects++;
  }

  /** Construct a circle with a specified radius */
  public CircleWithPrivateDataFields(double newRadius) {
    radius = newRadius;
16       numberOfObjects++;
17     }
18
19     /** Return radius */
20     public double getRadius() {
21       return radius;
22     }
23
24     /** Set a new radius */
25     public void setRadius(double newRadius) {
26       radius = (newRadius >= 0) ? newRadius : 0;
27     }
28
29     /** Return numberOfObjects */
30     public static int getNumberOfObjects() {
31       return numberOfObjects;
32     }
33
34     /** Return the area of this circle */
35     public double getArea() {
36       return radius * radius * Math.PI;
37     }
38   }
```

accessor method

mutator method

accessor method

The **getRadius()** method (lines 20–22) returns the radius, and the **setRadius(newRadius)** method (line 25–27) sets a new radius for the object. If the new radius is negative, **0** is set as the radius for the object. Since these methods are the only ways to read and modify the radius, you have total control over how the **radius** property is accessed. If you have to change the implementation of these methods, you don't need to change the client programs. This makes the class easy to maintain.

Listing 8.10 gives a client program that uses the **Circle** class to create a **Circle** object and modifies the radius using the **setRadius** method.

LISTING 8.10 TestCircleWithPrivateDataFields.java

```
public class TestCircleWithPrivateDataFields {
  /** Main method */
  public static void main(String[] args) {
    // Create a circle with radius 5.0
    CircleWithPrivateDataFields myCircle =
      new CircleWithPrivateDataFields(5.0);
    System.out.println("The area of the circle of radius "
      + myCircle.getRadius() + " is " + myCircle.getArea());

    // Increase myCircle's radius by 10%
    myCircle.setRadius(myCircle.getRadius() * 1.1);
    System.out.println("The area of the circle of radius "
      + myCircle.getRadius() + " is " + myCircle.getArea());

    System.out.println("The number of objects created is "
      + CircleWithPrivateDataFields.getNumberOfObjects());
  }
}
```

invoke public method

invoke public method

invoke public method

The data field **radius** is declared private. Private data can be accessed only within their defining class, so you cannot use **myCircle.radius** in the client program. A compile error would occur if you attempted to access private data from a client.

Since `numberOfObjects` is private, it cannot be modified. This prevents tampering. For example, the user cannot set `numberOfObjects` to `100`. The only way to make it `100` is to create `100` objects of the `Circle` class.

Suppose you combined `TestCircleWithPrivateDataFields` and `Circle` into one class by moving the `main` method in `TestCircleWithPrivateDataFields` into `Circle`. Could you use `myCircle.radius` in the `main` method? See Checkpoint Question 8.22 for the answer.

Design Guide

To prevent data from being tampered with and to make the class easy to maintain, declare data fields private.

Check Point

MyProgrammingLab™

8.20 What is an accessor method? What is a mutator method? What are the naming conventions for accessor methods and mutator methods?

8.21 What are the benefits of data field encapsulation?

8.22 In the following code, `radius` is private in the `Circle` class, and `myCircle` is an object of the `Circle` class. Does the highlighted code cause any problems? If so, explain why.

```java
public class Circle {
  private double radius = 1;

  /** Find the area of this circle */
  public double getArea() {
    return radius * radius * Math.PI;
  }

  public static void main(String[] args) {
    Circle myCircle = new Circle();
    System.out.println("Radius is " + myCircle.radius);
  }
}
```

8.10 Passing Objects to Methods

Key Point

Passing an object to a method is to pass the reference of the object.

You can pass objects to methods. Like passing an array, passing an object is actually passing the reference of the object. The following code passes the `myCircle` object as an argument to the `printCircle` method:

pass an object

```java
public class Test {
  public static void main(String[] args) {
    // CircleWithPrivateDataFields is defined in Listing 8.9
    CircleWithPrivateDataFields myCircle = new
      CircleWithPrivateDataFields(5.0);
    printCircle(myCircle);
  }

  public static void printCircle(CircleWithPrivateDataFields c) {
    System.out.println("The area of the circle of radius "
      + c.getRadius() + " is " + c.getArea());
  }
}
```

pass-by-value

Java uses exactly one mode of passing arguments: pass-by-value. In the preceding code, the value of `myCircle` is passed to the `printCircle` method. This value is a reference to a `Circle` object.

The program in Listing 8.11 demonstrates the difference between passing a primitive type value and passing a reference value.

LISTING 8.11 TestPassObject.java

pass object

object parameter

```java
 1 public class TestPassObject {
 2   /** Main method */
 3   public static void main(String[] args) {
 4     // Create a Circle object with radius 1
 5     CircleWithPrivateDataFields myCircle =
 6      new CircleWithPrivateDataFields(1);
 7
 8     // Print areas for radius 1, 2, 3, 4, and 5.
 9     int n = 5;
10     printAreas(myCircle, n);
11
12     // See myCircle.radius and times
13     System.out.println("\n" + "Radius is " + myCircle.getRadius());
14     System.out.println("n is " + n);
15   }
16
17   /** Print a table of areas for radius */
18   public static void printAreas(
19       CircleWithPrivateDataFields c, int times) {
20     System.out.println("Radius \t\tArea");
21     while (times >= 1) {
22       System.out.println(c.getRadius() + "\t\t" + c.getArea());
23       c.setRadius(c.getRadius() + 1);
24       times--;
25     }
26   }
27 }
```

```
Radius      Area
1.0         3.141592653589793
2.0         12.566370614359172
3.0         29.274333882308138
4.0         50.26548245743669
5.0         79.53981633974483
Radius is 6.0
n is 5
```

The **CircleWithPrivateDataFields** class is defined in Listing 8.9. The program passes a **CircleWithPrivateDataFields** object **myCircle** and an integer value from **n** to invoke **printAreas(myCircle, n)** (line 9), which prints a table of areas for radii **1**, **2**, **3**, **4**, **5**, as shown in the sample output.

Figure 8.18 shows the call stack for executing the methods in the program. Note that the objects are stored in a heap (see Section 6.6).

When passing an argument of a primitive data type, the value of the argument is passed. In this case, the value of **n** (**5**) is passed to **times**. Inside the **printAreas** method, the content of **times** is changed; this does not affect the content of **n**.

When passing an argument of a reference type, the reference of the object is passed. In this case, **c** contains a reference for the object that is also referenced via **myCircle**. Therefore, changing the properties of the object through **c** inside the **printAreas** method has the same effect as doing so outside the method through the variable **myCircle**. Pass-by-value on references can be best described semantically as *pass-by-sharing*; that is, the object referenced in the method is the same as the object being passed.

pass-by-sharing

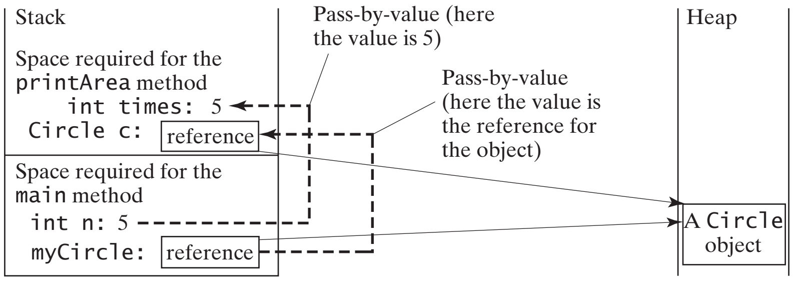

Stack
Space required for the printArea method
int times: 5
Circle c: reference
Space required for the main method
int n: 5
myCircle: reference
Pass-by-value (here the value is 5)
Pass-by-value (here the value is the reference for the object)
Heap
A Circle object

FIGURE 8.18 The value of `n` is passed to `times`, and the reference to `myCircle` is passed to `c` in the `printAreas` method.

Check Point
MyProgrammingLab™

8.23 Describe the difference between passing a parameter of a primitive type and passing a parameter of a reference type. Show the output of the following programs:

```
public class Test {
  public static void main(String[] args) {
    Count myCount = new Count();
    int times = 0;

    for (int i = 0; i < 100; i++)
      increment(myCount, times);

    System.out.println("count is " + myCount.count);
    System.out.println("times is " + times);
  }

  public static void increment(Count c, int times) {
    c.count++;
    times++;
  }
}
```

```
public class Count {
  public int count;

  public Count(int c) {
    count = c;
  }

  public Count() {
    count = 1;
  }
}
```

8.24 Show the output of the following program:

```
public class Test {
  public static void main(String[] args) {
    Circle circle1 = new Circle(1);
    Circle circle2 = new Circle(2);

    swap1(circle1, circle2);
    System.out.println("After swap1: circle1 = " +
      circle1.radius + " circle2 = " + circle2.radius);

    swap2(circle1, circle2);
    System.out.println("After swap2: circle1 = " +
      circle1.radius + " circle2 = " + circle2.radius);
  }

  public static void swap1(Circle x, Circle y) {
    Circle temp = x;
    x = y;
    y = temp;
  }

  public static void swap2(Circle x, Circle y) {
    double temp = x.radius;
    x.radius = y.radius;
    y.radius = temp;
  }
}

class Circle {
  double radius;

  Circle(double newRadius) {
    radius = newRadius;
  }
}
```

8.25 Show the printout of the following code:

```
public class Test {
  public static void main(String[] args) {
    int[] a = {1, 2};
    swap(a[0], a[1]);
    System.out.println("a[0] = " + a[0]
      + " a[1] = " + a[1]);
  }

  public static void swap(int n1, int n2) {
    int temp = n1;
    n1 = n2;
    n2 = temp;
  }
}
```

(a)

```
public class Test {
  public static void main(String[] args) {
    int[] a = {1, 2};
    swap(a);
    System.out.println("a[0] = " + a[0]
      + " a[1] = " + a[1]);
  }

  public static void swap(int[] a) {
    int temp = a[0];
    a[0] = a[1];
    a[1] = temp;
  }
}
```

(b)

```
public class Test {
  public static void main(String[] args) {
    T t = new T();
    swap(t);
    System.out.println("e1 = " + t.e1
      + " e2 = " + t.e2);
  }

  public static void swap(T t) {
    int temp = t.e1;
    t.e1 = t.e2;
    t.e2 = temp;
  }
}

class T {
  int e1 = 1;
  int e2 = 2;
}
```

(c)

```
public class Test {
  public static void main(String[] args) {
    T t1 = new T();
    T t2 = new T();
    System.out.println("t1's i = " +
      t1.i + " and j = " + t1.j);
    System.out.println("t2's i = " +
      t2.i + " and j = " + t2.j);
  }
}

class T {
  static int i = 0;
  int j = 0;

  T() {
    i++;
    j = 1;
  }
}
```

(d)

8.26 What is the output of the following programs?

```
import java.util.Date;

public class Test {
  public static void main(String[] args) {
    Date date = null;
    m1(date);
    System.out.println(date);
  }

  public static void m1(Date date) {
    date = new Date();
  }
}
```

(a)

```
import java.util.Date;

public class Test {
  public static void main(String[] args) {
    Date date = new Date(1234567);
    m1(date);
    System.out.println(date.getTime());
  }

  public static void m1(Date date) {
    date = new Date(7654321);
  }
}
```

(b)

```
import java.util.Date;

public class Test {
  public static void main(String[] args) {
    Date date = new Date(1234567);
    m1(date);
    System.out.println(date.getTime());
  }

  public static void m1(Date date) {
    date.setTime(7654321);
  }
}
```

(c)

```
import java.util.Date;

public class Test {
  public static void main(String[] args) {
    Date date = new Date(1234567);
    m1(date);
    System.out.println(date.getTime());
  }

  public static void m1(Date date) {
    date = null;
  }
}
```

(d)

8.11 Array of Objects

Key Point

An array can hold objects as well as primitive type values.

Chapter 6, Single-Dimensional Arrays, described how to create arrays of primitive type elements. You can also create arrays of objects. For example, the following statement declares and creates an array of ten `Circle` objects:

```
Circle[] circleArray = new Circle[10];
```

To initialize `circleArray`, you can use a `for` loop like this one:

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

An array of objects is actually an *array of reference variables*. So, invoking `circleArray[1].getArea()` involves two levels of referencing, as shown in Figure 8.19. `circleArray` references the entire array; `circleArray[1]` references a `Circle` object.

Note

When an array of objects is created using the `new` operator, each element in the array is a reference variable with a default value of `null`.

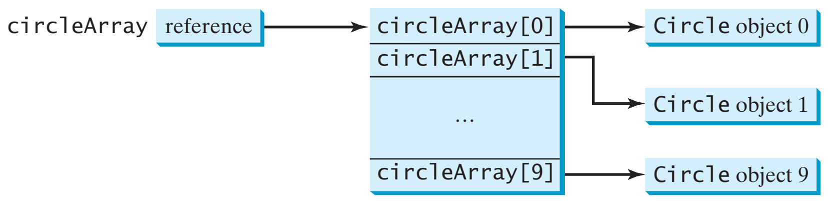


FIGURE 8.19 In an array of objects, an element of the array contains a reference to an object.

Listing 8.12 gives an example that demonstrates how to use an array of objects. The program summarizes the areas of an array of circles. The program creates **circleArray**, an array composed of five **Circle** objects; it then initializes circle radii with random values and displays the total area of the circles in the array.

LISTING 8.12 TotalArea.java

```java
public class TotalArea {
  /** Main method */
  public static void main(String[] args) {
    // Declare circleArray
    CircleWithPrivateDataFields[] circleArray;

    // Create circleArray
    circleArray = createCircleArray();

    // Print circleArray and total areas of the circles
    printCircleArray(circleArray);
  }

  /** Create an array of Circle objects */
  public static CircleWithPrivateDataFields[] createCircleArray() {
    CircleWithPrivateDataFields[] circleArray =
      new CircleWithPrivateDataFields[5];

    for (int i = 0; i < circleArray.length; i++) {
      circleArray[i] =
        new CircleWithPrivateDataFields(Math.random() * 100);
    }

    // Return Circle array
    return circleArray;
  }

  /** Print an array of circles and their total area */
  public static void printCircleArray(
      CircleWithPrivateDataFields[] circleArray) {
    System.out.printf("%-30s%-15s\n", "Radius", "Area");
    for (int i = 0; i < circleArray.length; i++) {
      System.out.printf("%-30f%-15f\n", circleArray[i].getRadius(),
        circleArray[i].getArea());
    }

    System.out.println("-----------------------------------------");

    // Compute and display the result
    System.out.printf("%-30s%-15f\n", "The total area of circles is",
      sum(circleArray) );
```

array of objects

return array of objects

pass array of objects

pass array of objects

```
42    }
43
44    /** Add circle areas */
45    public static double sum(CircleWithPrivateDataFields[] circleArray) {
46      // Initialize sum
47      double sum = 0;
48
49      // Add areas to sum
50      for (int i = 0; i < circleArray.length; i++)
51        sum += circleArray[i].getArea();
52
53      return sum;
54    }
55  }
```

```
Radius                        Area
70.577708                     15648.941866
44.152266                     6124.291736
24.867853                     1942.792644
 5.680718                     101.380949
36.734246                     4239.280350
----------------------------------------------
The total area of circles is  28056.687544
```

The program invokes `createCircleArray()` (line 8) to create an array of five circle objects. Several circle classes were introduced in this chapter. This example uses the `CircleWithPrivateDataFields` class introduced in Section 8.9, Data Field Encapsulation.

The circle radii are randomly generated using the `Math.random()` method (line 21). The `createCircleArray` method returns an array of `CircleWithPrivateDataFields` objects (line 25). The array is passed to the `printCircleArray` method, which displays the radius and area of each circle and the total area of the circles.

The sum of the circle areas is computed by invoking the `sum` method (line 41), which takes the array of `CircleWithPrivateDataFields` objects as the argument and returns a `double` value for the total area.

Check Point

MyProgrammingLab™

8.27 What is wrong in the following code?

```
public class Test {
  public static void main(String[] args) {
    java.util.Date[] dates = new java.util.Date[10];
    System.out.println(dates[0]);
    System.out.println(dates[0].toString());
  }
}
```

Key Terms

action 296	constructor 296
anonymous object 305	data field 296
attribute 296	data field encapsulation 319
behavior 296	default constructor 303
class 296	dot operator (.) 304
client 299	getter (or accessor) 320

instance 296
instance method 305
instance variable 305
instantiation 296
no-arg constructor 303
`null` value 305
object 296
object-oriented programming (OOP)
package-private (or package-access) 317
private constructor 319
property 296
public class 299
reference type 304
reference variable 304
setter (or mutator) 320
state 296
static method 312
static variable 312
Unified Modeling Language (UML) 297

Chapter Summary

1. A *class* is a template for *objects*. It defines the *properties* of objects and provides *constructors* for creating objects and methods for manipulating them.

2. A class is also a data type. You can use it to declare object *reference variables*. An object reference variable that appears to hold an object actually contains a reference to that object. Strictly speaking, an object reference variable and an object are different, but most of the time the distinction can be ignored.

3. An object is an *instance* of a class. You use the `new` operator to create an object, and the *dot operator* (`.`) to access members of that object through its reference variable.

4. An *instance variable* or *method* belongs to an instance of a class. Its use is associated with individual instances. A *static variable* is a variable shared by all instances of the same class. A *static method* is a method that can be invoked without using instances.

5. Every instance of a class can access the class's static variables and methods. For clarity, however, it is better to invoke static variables and methods using `ClassName.variable` and `ClassName.method`.

6. Modifiers specify how the class, method, and data are accessed. A `public` class, method, or data is accessible to all clients. A `private` method or data is accessible only inside the class.

7. You can provide a `get` method or a `set` method to enable clients to see or modify the data. Colloquially, a `get` method is referred to as a *getter* (or *accessor*), and a `set` method as a *setter* (or *mutator*).

8. A `get` method has the signature `public returnType getPropertyName()`. If the `returnType` is `boolean`, the `get` method should be defined as `public boolean isPropertyName()`. A `set` method has the signature `public void setPropertyName(dataType propertyValue)`.

9. All parameters are passed to methods using pass-by-value. For a parameter of a primitive type, the actual value is passed; for a parameter of a *reference type*, the reference for the object is passed.

10. A Java array is an object that can contain primitive type values or object type values. When an array of objects is created, its elements are assigned the default value of `null`.

Test Questions

Do the test questions for this chapter online at www.cs.armstrong.edu/liang/intro9e/test.html.

MyProgrammingLab™

Programming Exercises

three objectives

Pedagogical Note

The exercises in Chapters 8–11, 15 help you achieve three objectives:

- Design classes and draw UML class diagrams
- Implement classes from the UML
- Use classes to develop applications

Students can download solutions for the UML diagrams for the even-numbered exercises from the Companion Website, and instructors can download all solutions from the same site.

Sections 8.2–8.5

8.1 (*The* **`Rectangle`** *class*) Following the example of the `Circle` class in Section 8.2, design a class named `Rectangle` to represent a rectangle. The class contains:

- Two `double` data fields named `width` and `height` that specify the width and height of the rectangle. The default values are `1` for both `width` and `height`.
- A no-arg constructor that creates a default rectangle.
- A constructor that creates a rectangle with the specified `width` and `height`.
- A method named `getArea()` that returns the area of this rectangle.
- A method named `getPerimeter()` that returns the perimeter.

Draw the UML diagram for the class and then implement the class. Write a test program that creates two `Rectangle` objects—one with width `4` and height `40` and the other with width `3.5` and height `35.9`. Display the width, height, area, and perimeter of each rectangle in this order.

8.2 (*The* **`Stock`** *class*) Following the example of the `Circle` class in Section 8.2, design a class named `Stock` that contains:

- A string data field named `symbol` for the stock's symbol.
- A string data field named `name` for the stock's name.
- A `double` data field named `previousClosingPrice` that stores the stock price for the previous day.
- A `double` data field named `currentPrice` that stores the stock price for the current time.
- A constructor that creates a stock with the specified symbol and name.
- A method named `getChangePercent()` that returns the percentage changed from `previousClosingPrice` to `currentPrice`.

Draw the UML diagram for the class and then implement the class. Write a test program that creates a `Stock` object with the stock symbol `ORCL`, the name `Oracle Corporation`, and the previous closing price of `34.5`. Set a new current price to `34.35` and display the price-change percentage.

Section 8.6

***8.3** (*Use the* **`Date`** *class*) Write a program that creates a `Date` object, sets its elapsed time to `10000`, `100000`, `1000000`, `10000000`, `100000000`, `1000000000`, `10000000000`, and `100000000000`, and displays the date and time using the `toString()` method, respectively.

***8.4** (*Use the* **`Random`** *class*) Write a program that creates a `Random` object with seed `1000` and displays the first 50 random integers between `0` and `100` using the `nextInt(100)` method.

***8.5** (*Use the* **`GregorianCalendar`** *class*) Java API has the `GregorianCalendar` class in the `java.util` package, which you can use to obtain the year, month, and day of a date. The no-arg constructor constructs an instance for the current date, and the methods `get(GregorianCalendar.YEAR)`, `get(GregorianCalendar.MONTH)`, and `get(GregorianCalendar.DAY_OF_MONTH)` return the year, month, and day. Write a program to perform two tasks:

- Display the current year, month, and day.
- The `GregorianCalendar` class has the `setTimeInMillis(long)`, which can be used to set a specified elapsed time since January 1, 1970. Set the value to `1234567898765L` and display the year, month, and day.

Sections 8.7–8.9

****8.6** (*Display calendars*) Rewrite the `PrintCalendar` class in Listing 5.12 to display calendars in a message dialog box. Since the output is generated from several static methods in the class, you may define a static `String` variable `output` for storing the output and display it in a message dialog box.

8.7 (*The* **`Account`** *class*) Design a class named `Account` that contains:

- A private `int` data field named `id` for the account (default `0`).
- A private `double` data field named `balance` for the account (default `0`).
- A private `double` data field named `annualInterestRate` that stores the current interest rate (default `0`). Assume all accounts have the same interest rate.
- A private `Date` data field named `dateCreated` that stores the date when the account was created.
- A no-arg constructor that creates a default account.
- A constructor that creates an account with the specified id and initial balance.
- The accessor and mutator methods for `id`, `balance`, and `annualInterestRate`.
- The accessor method for `dateCreated`.
- A method named `getMonthlyInterestRate()` that returns the monthly interest rate.
- A method named `getMonthlyInterest()` that returns the monthly interest.
- A method named `withdraw` that withdraws a specified amount from the account.
- A method named `deposit` that deposits a specified amount to the account.

Draw the UML diagram for the class and then implement the class. (*Hint*: The method `getMonthlyInterest()` is to return monthly interest, not the interest rate. Monthly interest is `balance * monthlyInterestRate`. `monthlyInterestRate` is `annualInterestRate / 12`. Note that `annualInterestRate` is a percentage, e.g.,like 4.5%. You need to divide it by 100.)

Write a test program that creates an `Account` object with an account ID of 1122, a balance of $20,000, and an annual interest rate of 4.5%. Use the `withdraw` method to withdraw $2,500, use the `deposit` method to deposit $3,000, and print the balance, the monthly interest, and the date when this account was created.

8.8 (*The* **`Fan`** *class*) Design a class named `Fan` to represent a fan. The class contains:

VideoNote
The Fan class

- Three constants named `SLOW`, `MEDIUM`, and `FAST` with the values `1`, `2`, and `3` to denote the fan speed.

- A private **int** data field named **speed** that specifies the speed of the fan (the default is **SLOW**).
- A private **boolean** data field named **on** that specifies whether the fan is on (the default is **false**).
- A private **double** data field named **radius** that specifies the radius of the fan (the default is **5**).
- A string data field named **color** that specifies the color of the fan (the default is **blue**).
- The accessor and mutator methods for all four data fields.
- A no-arg constructor that creates a default fan.
- A method named **toString()** that returns a string description for the fan. If the fan is on, the method returns the fan speed, color, and radius in one combined string. If the fan is not on, the method returns the fan color and radius along with the string "fan is off" in one combined string.

Draw the UML diagram for the class and then implement the class. Write a test program that creates two **Fan** objects. Assign maximum speed, radius **10**, color **yellow**, and turn it on to the first object. Assign medium speed, radius **5**, color **blue**, and turn it off to the second object. Display the objects by invoking their **toString** method.

****8.9** (*Geometry: n-sided regular polygon*) In an *n*-sided regular polygon, all sides have the same length and all angles have the same degree (i.e., the polygon is both equilateral and equiangular). Design a class named **RegularPolygon** that contains:

- A private **int** data field named **n** that defines the number of sides in the polygon with default value **3**.
- A private **double** data field named **side** that stores the length of the side with default value **1**.
- A private **double** data field named **x** that defines the *x*-coordinate of the polygon's center with default value **0**.
- A private **double** data field named **y** that defines the *y*-coordinate of the polygon's center with default value **0**.
- A no-arg constructor that creates a regular polygon with default values.
- A constructor that creates a regular polygon with the specified number of sides and length of side, centered at (**0**, **0**).
- A constructor that creates a regular polygon with the specified number of sides, length of side, and *x*-and *y*-coordinates.
- The accessor and mutator methods for all data fields.
- The method **getPerimeter()** that returns the perimeter of the polygon.
- The method **getArea()** that returns the area of the polygon. The formula for computing the area of a regular polygon is $Area = \dfrac{n \times s^2}{4 \times \tan\left(\dfrac{\pi}{n}\right)}$.

Draw the UML diagram for the class and then implement the class. Write a test program that creates three **RegularPolygon** objects, created using the no-arg constructor, using **RegularPolygon(6, 4)**, and using **RegularPolygon(10, 4, 5.6, 7.8)**. For each object, display its perimeter and area.

***8.10** (*Algebra: quadratic equations*) Design a class named **QuadraticEquation** for a quadratic equation $ax^2 + bx + x = 0$. The class contains:

- Private data fields **a**, **b**, and **c** that represent three coefficients.
- A constructor for the arguments for **a**, **b**, and **c**.

- Three `get` methods for `a`, `b`, and `c`.
- A method named `getDiscriminant()` that returns the discriminant, which is $b^2 - 4ac$.
- The methods named `getRoot1()` and `getRoot2()` for returning two roots of the equation

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

These methods are useful only if the discriminant is nonnegative. Let these methods return `0` if the discriminant is negative.

Draw the UML diagram for the class and then implement the class. Write a test program that prompts the user to enter values for *a*, *b*, and *c* and displays the result based on the discriminant. If the discriminant is positive, display the two roots. If the discriminant is 0, display the one root. Otherwise, display "The equation has no roots." See Programming Exercise 3.1 for sample runs.

***8.11** (*Algebra: 2 × 2 linear equations*) Design a class named `LinearEquation` for a 2 × 2 system of linear equations:

$$\begin{aligned} ax + by &= e \\ cx + dy &= f \end{aligned} \qquad x = \frac{ed - bf}{ad - bc} \qquad y = \frac{af - ec}{ad - bc}$$

The class contains:

- Private data fields `a`, `b`, `c`, `d`, `e`, and `f`.
- A constructor with the arguments for `a`, `b`, `c`, `d`, `e`, and `f`.
- Six `get` methods for `a`, `b`, `c`, `d`, `e`, and `f`.
- A method named `isSolvable()` that returns true if $ad - bc$ is not 0.
- Methods `getX()` and `getY()` that return the solution for the equation.

Draw the UML diagram for the class and then implement the class. Write a test program that prompts the user to enter `a`, `b`, `c`, `d`, `e`, and `f` and displays the result. If $ad - bc$ is 0, report that "The equation has no solution." See Programming Exercise 3.3 for sample runs.

****8.12** (*Geometry: intersection*) Suppose two line segments intersect. The two endpoints for the first line segment are (`x1`, `y1`) and (`x2`, `y2`) and for the second line segment are (`x3`, `y3`) and (`x4`, `y4`). Write a program that prompts the user to enter these four endpoints and displays the intersecting point. (*Hint*: Use the `LinearEquation` class in Exercise 8.11.)

```
Enter the endpoints of the first line segment: 2.0 2.0 0 0 ↵Enter
Enter the endpoints of the second line segment: 0 2.0 2.0 0 ↵Enter
The intersecting point is: (1.0, 1.0)
```

****8.13** (*The* `Location` *class*) Design a class named `Location` for locating a maximal value and its location in a two-dimensional array. The class contains public data fields `row`, `column`, and `maxValue` that store the maximal value and its indices in a two-dimensional array with `row` and `column` as `int` types and `maxValue` as a `double` type.

Write the following method that returns the location of the largest element in a two-dimensional array:

```
public static Location locateLargest(double[][] a)
```

The return value is an instance of **Location**. Write a test program that prompts the user to enter a two-dimensional array and displays the location of the largest element in the array. Here is a sample run:

```
Enter the number of rows and columns in the array: 3 4 ↵Enter
Enter the array:
23.5 35 2 10 ↵Enter
4.5 3 45 3.5 ↵Enter
35 44 5.5 9.6 ↵Enter
The location of the largest element is 45 at (1, 2)
```

***8.14** (*Stopwatch*) Design a class named **StopWatch**. The class contains:

- Private data fields **startTime** and **endTime** with get methods.
- A no-arg constructor that initializes **startTime** with the current time.
- A method named **start()** that resets the **startTime** to the current time.
- A method named **stop()** that sets the **endTime** to the current time.
- A method named **getElapsedTime()** that returns the elapsed time for the stopwatch in milliseconds.

Draw the UML diagram for the class and then implement the class. Write a test program that measures the execution time of sorting 100,000 numbers using selection sort.

CHAPTER

9

STRINGS

Objectives

- To use the `String` class to process fixed strings (§9.2).
- To construct strings (§9.2.1).
- To understand that strings are immutable and to create an interned string (§9.2.2).
- To compare strings (§9.2.3).
- To get string length and characters, and combine strings (§9.2.4).
- To obtain substrings (§9.2.5).
- To convert, replace, and split strings (§9.2.6).
- To match, replace, and split strings by patterns (§9.2.7).
- To search for a character or substring in a string (§9.2.8).
- To convert between a string and an array (§9.2.9).
- To convert characters and numbers into a string (§9.2.10).
- To obtain a formatted string (§9.2.11).
- To check whether a string is a palindrome (§9.3).
- To convert hexadecimal numbers to decimal numbers (§9.4).
- To use the `Character` class to process a single character (§9.5).
- To use the `StringBuilder` and `StringBuffer` classes to process flexible strings (§9.6).
- To distinguish among the `String`, `StringBuilder`, and `StringBuffer` classes (§9.2–9.6).
- To learn how to pass arguments to the `main` method from the command line (§9.7).

9.1 Introduction

Key Point

The classes `String`, `StringBuilder`, and `StringBuffer` are used for processing strings.

A *string* is a sequence of characters. Strings are frequently used in programming. In many languages, strings are treated as an array of characters, but in Java a string is treated as an object. This chapter introduces the classes for processing strings.

9.2 The `String` Class

Key Point

A `String` object is immutable: Its content cannot be changed once the string is created.

The `String` class has 13 constructors and more than 40 methods for manipulating strings. Not only is it very useful in programming, but it is also a good example for learning classes and objects.

9.2.1 Constructing a String

You can create a string object from a string literal or from an array of characters. To create a string from a string literal, use the syntax:

```
String newString = new String(stringLiteral);
```

The argument `stringLiteral` is a sequence of characters enclosed inside double quotes. The following statement creates a `String` object `message` for the string literal `"Welcome to Java"`:

```
String message = new String("Welcome to Java");
```

string literal object

Java treats a string literal as a `String` object. Thus, the following statement is valid:

```
String message = "Welcome to Java";
```

You can also create a string from an array of characters. For example, the following statements create the string `"Good Day"`:

```
char[] charArray = {'G', 'o', 'o', 'd', ' ', 'D', 'a', 'y'};
String message = new String(charArray);
```

`String` variable, `String` object, string value

Note

A `String` variable holds a reference to a `String` object that stores a string value. Strictly speaking, the terms *`String` variable*, *`String` object*, and *string value* are different, but most of the time the distinctions between them can be ignored. For simplicity, the term *string* will often be used to refer to `String` variable, `String` object, and string value.

9.2.2 Immutable Strings and Interned Strings

immutable

A `String` object is immutable; its contents cannot be changed. Does the following code change the contents of the string?

```
String s = "Java";
s = "HTML";
```

The answer is no. The first statement creates a `String` object with the content `"Java"` and assigns its reference to `s`. The second statement creates a new `String` object with the content

"HTML" and assigns its reference to s. The first String object still exists after the assignment, but it can no longer be accessed, because variable s now points to the new object, as shown in Figure 9.1.

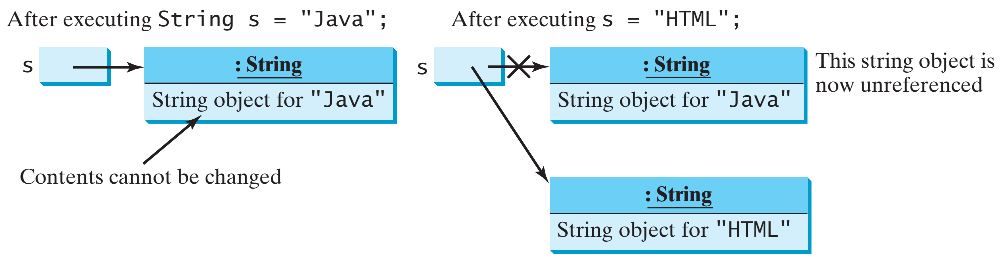
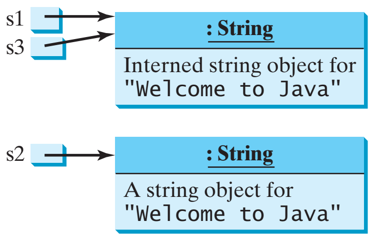

After executing String s = "Java";
s
: String
String object for "Java"
Contents cannot be changed
After executing s = "HTML";
s
: String
String object for "Java"
This string object is now unreferenced
: String
String object for "HTML"

FIGURE 9.1 Strings are immutable; once created, their contents cannot be changed.

Because strings are immutable and are ubiquitous in programming, the JVM uses a unique instance for string literals with the same character sequence in order to improve efficiency and save memory. Such an instance is called an *interned string*. For example, the following statements:

interned string

```
String s1 = "Welcome to Java";

String s2 = new String("Welcome to Java");

String s3 = "Welcome to Java";

System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
```


s1
s3
: String
Interned string object for "Welcome to Java"
s2
: String
A string object for "Welcome to Java"

display

```
s1 == s2 is false
s1 == s3 is true
```

In the preceding statements, s1 and s3 refer to the same interned string—"Welcome to Java"—so s1 == s3 is true. However, s1 == s2 is false, because s1 and s2 are two different string objects, even though they have the same contents.

9.2.3 String Comparisons

The String class provides the methods for comparing strings, as shown in Figure 9.2.

How do you compare the contents of two strings? You might attempt to use the == operator, as follows:

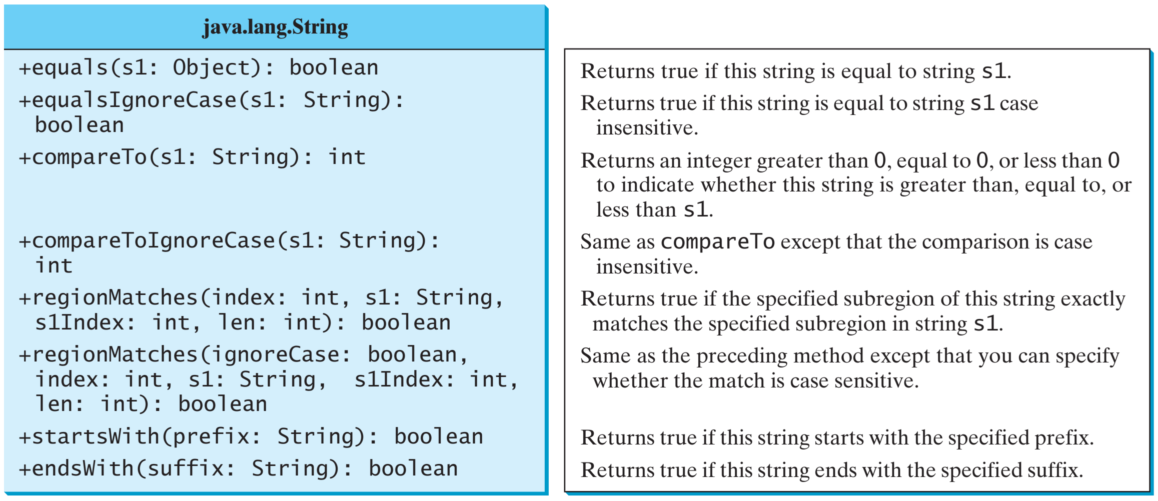
==

```
if (string1 == string2)
  System.out.println("string1 and string2 are the same object");
else
  System.out.println("string1 and string2 are different objects");
```

However, the == operator checks only whether string1 and string2 refer to the same object; it does not tell you whether they have the same contents. Therefore, you cannot use the == operator to find out whether two string variables have the same contents. Instead, you should use the equals method. The following code, for instance, can be used to compare two strings:

string1.equals(string2)

```
if (string1.equals(string2))
  System.out.println("string1 and string2 have the same contents");
else
  System.out.println("string1 and string2 are not equal");
```



java.lang.String	
`+equals(s1: Object): boolean`	Returns true if this string is equal to string `s1`.
`+equalsIgnoreCase(s1: String): boolean`	Returns true if this string is equal to string `s1` case insensitive.
`+compareTo(s1: String): int`	Returns an integer greater than `0`, equal to `0`, or less than `0` to indicate whether this string is greater than, equal to, or less than `s1`.
`+compareToIgnoreCase(s1: String): int`	Same as `compareTo` except that the comparison is case insensitive.
`+regionMatches(index: int, s1: String, s1Index: int, len: int): boolean`	Returns true if the specified subregion of this string exactly matches the specified subregion in string `s1`.
`+regionMatches(ignoreCase: boolean, index: int, s1: String, s1Index: int, len: int): boolean`	Same as the preceding method except that you can specify whether the match is case sensitive.
`+startsWith(prefix: String): boolean`	Returns true if this string starts with the specified prefix.
`+endsWith(suffix: String): boolean`	Returns true if this string ends with the specified suffix.

FIGURE 9.2 The **`String`** class contains the methods for comparing strings.

Note that parameter type for the **`equals`** method is **`Object`**. We will introduce the **`Object`** class in Chapter 11. For now, you can replace **`Object`** by **`String`** for using the **`equals`** method to compare two strings. For example, the following statements display **`true`** and then **`false`**.

```
String s1 = new String("Welcome to Java");
String s2 = "Welcome to Java";
String s3 = "Welcome to C++";
System.out.println(s1.equals(s2)); // true
System.out.println(s1.equals(s3)); // false
```

The **`compareTo`** method can also be used to compare two strings. For example, consider the following code:

`s1.compareTo(s2)`

```
s1.compareTo(s2)
```

The method returns the value **`0`** if **`s1`** is equal to **`s2`**, a value less than **`0`** if **`s1`** is lexicographically (i.e., in terms of Unicode ordering) less than **`s2`**, and a value greater than **`0`** if **`s1`** is lexicographically greater than **`s2`**.

The actual value returned from the **`compareTo`** method depends on the offset of the first two distinct characters in **`s1`** and **`s2`** from left to right. For example, suppose **`s1`** is **`abc`** and **`s2`** is **`abg`**, and **`s1.compareTo(s2)`** returns **`-4`**. The first two characters (**`a`** vs. **`a`**) from **`s1`** and **`s2`** are compared. Because they are equal, the second two characters (**`b`** vs. **`b`**) are compared. Because they are also equal, the third two characters (**`c`** vs. **`g`**) are compared. Since the character **`c`** is **`4`** less than **`g`**, the comparison returns **`-4`**.

Caution

Syntax errors will occur if you compare strings by using comparison operators **`>`**, **`>=`**, **`<`**, or **`<=`**. Instead, you have to use **`s1.compareTo(s2)`**.

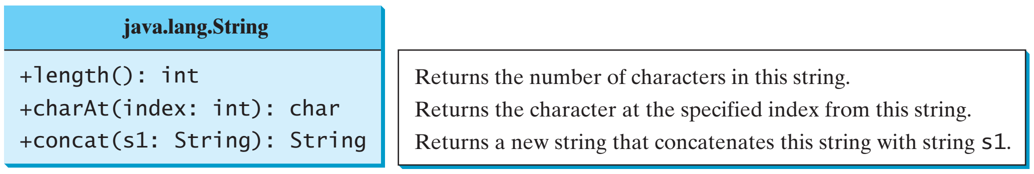
Note

The **`equals`** method returns **`true`** if two strings are equal and **`false`** if they are not. The **`compareTo`** method returns **`0`**, a positive integer, or a negative integer, depending on whether one string is equal to, greater than, or less than the other string.

The **`String`** class also provides the **`equalsIgnoreCase`**, **`compareToIgnoreCase`**, and **`regionMatches`** methods for comparing strings. The **`equalsIgnoreCase`** and compareToIgnoreCase methods ignore the case of the letters when comparing two strings. The regionMatches method compares portions of two strings for equality. You can also use str.startsWith(prefix) to check whether string str starts with a specified prefix, and str.endsWith(suffix) to check whether string str ends with a specified suffix.

9.2.4 Getting String Length and Characters, and Combining Strings

The String class provides the methods for obtaining a string's length, retrieving individual characters, and concatenating strings, as shown in Figure 9.3.



Returns the number of characters in this string.
Returns the character at the specified index from this string.
Returns a new string that concatenates this string with string s1.

FIGURE 9.3 The String class contains the methods for getting string length, individual characters, and combining strings.

length()

You can get the length of a string by invoking its length() method. For example, message.length() returns the length of the string message.

string length vs. array length

Caution

length is a method in the String class but is a property of an array object. Therefore, you have to use s.length() to get the number of characters in string s, and a.length to get the number of elements in array a.

charAt(index)

The s.charAt(index) method can be used to retrieve a specific character in a string s, where the index is between 0 and s.length()-1. For example, message.charAt(0) returns the character W, as shown in Figure 9.4.

string literal

Note

When you use a string, you often know its literal value. For convenience, Java allows you to use the string literal to refer directly to strings without creating new variables. Thus, "Welcome to Java".charAt(0) is correct and returns W.

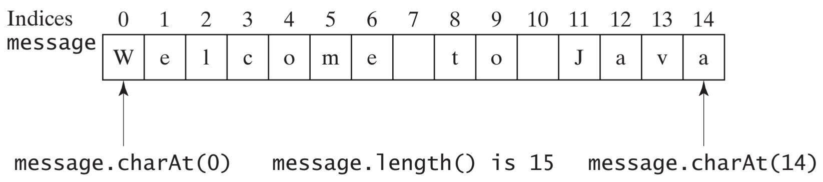


FIGURE 9.4 The characters in a String object are stored using an array internally.

encapsulating string

Note

The String class uses an array to store characters internally. The array is private and cannot be accessed outside of the String class. The String class provides many public methods, such as length() and charAt(index), to retrieve the string information. This is a good example of encapsulation: the data field of the class is hidden from the user through the private modifier, and thus the user cannot directly manipulate it. If the array were not private, the user would be able to change the string content by modifying the array. This would violate the tenet that the String class is immutable.

string index range

Caution

Attempting to access characters in a string `s` out of bounds is a common programming error. To avoid it, make sure that you do not use an index beyond `s.length() – 1`. For example, `s.charAt(s.length())` would cause a `StringIndexOutOfBoundsException`.

You can use the `concat` method to concatenate two strings. The statement shown below, for example, concatenates strings `s1` and `s2` into `s3`:

s1.concat(s2)

```
String s3 = s1.concat(s2);
```

Because string concatenation is heavily used in programming, Java provides a convenient way to accomplish it. You can use the plus (`+`) operator to concatenate two strings, so the previous statement is equivalent to

s1 + s2

```
String s3 = s1 + s2;
```

The following code combines the strings `message`, `" and "`, and `"HTML"` into one string:

```
String myString = message + " and " + "HTML";
```

Recall that the `+` operator can also concatenate a number with a string. In this case, the number is converted into a string and then concatenated. Note that at least one of the operands must be a string in order for concatenation to take place.

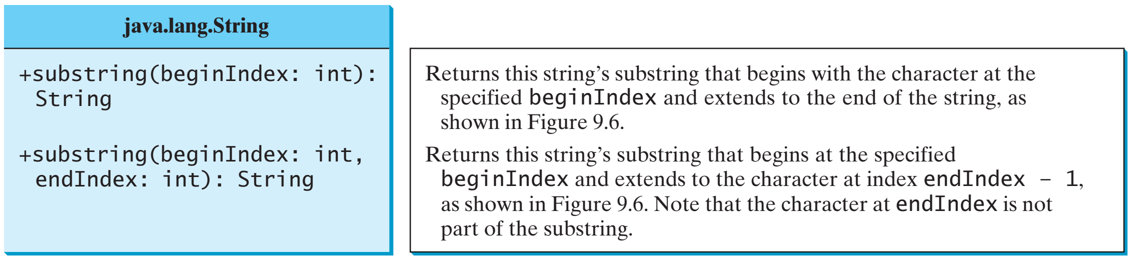
9.2.5 Obtaining Substrings

You can obtain a single character from a string using the `charAt` method, as shown in Figure 9.3. You can also obtain a substring from a string using the `substring` method in the `String` class, as shown in Figure 9.5.

For example,

```
String message = "Welcome to Java".substring(0, 11) + "HTML";
```

The string `message` now becomes `Welcome to HTML`.



java.lang.String	
`+substring(beginIndex: int): String`	Returns this string's substring that begins with the character at the specified `beginIndex` and extends to the end of the string, as shown in Figure 9.6.
`+substring(beginIndex: int, endIndex: int): String`	Returns this string's substring that begins at the specified `beginIndex` and extends to the character at index `endIndex – 1`, as shown in Figure 9.6. Note that the character at `endIndex` is not part of the substring.

FIGURE 9.5 The `String` class contains the methods for obtaining substrings.

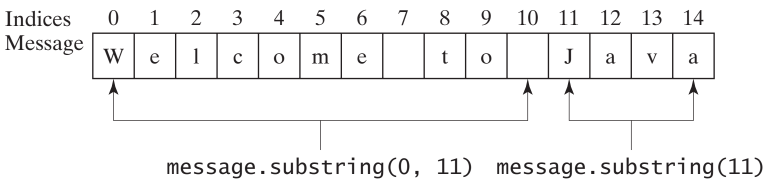


FIGURE 9.6 The `substring` method obtains a substring from a string.

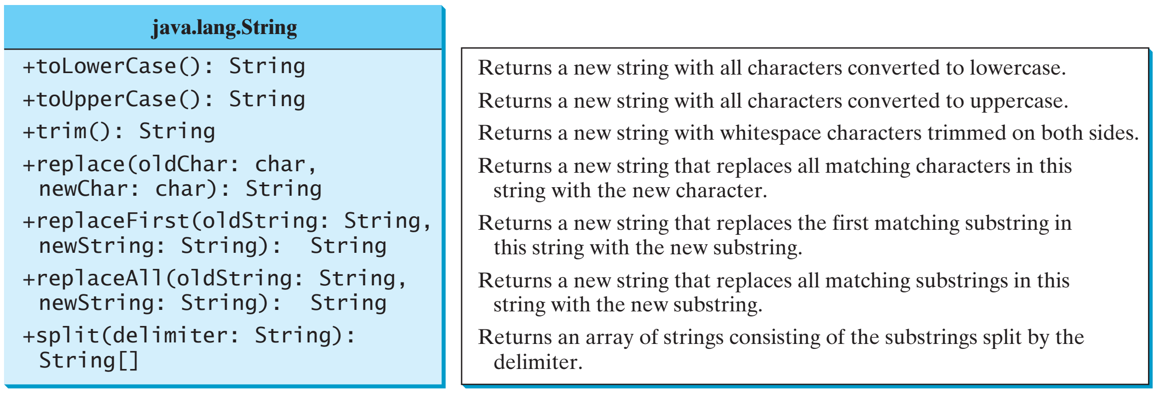
Note

If beginIndex is endIndex, substring(beginIndex, endIndex) returns an empty string with length 0. If beginIndex > endIndex, it would be a runtime error.

beginIndex <= endIndex

9.2.6 Converting, Replacing, and Splitting Strings

The String class provides the methods for converting, replacing, and splitting strings, as shown in Figure 9.7.



java.lang.String	
+toLowerCase(): String	Returns a new string with all characters converted to lowercase.
+toUpperCase(): String	Returns a new string with all characters converted to uppercase.
+trim(): String	Returns a new string with whitespace characters trimmed on both sides.
+replace(oldChar: char, newChar: char): String	Returns a new string that replaces all matching characters in this string with the new character.
+replaceFirst(oldString: String, newString: String): String	Returns a new string that replaces the first matching substring in this string with the new substring.
+replaceAll(oldString: String, newString: String): String	Returns a new string that replaces all matching substrings in this string with the new substring.
+split(delimiter: String): String[]	Returns an array of strings consisting of the substrings split by the delimiter.

FIGURE 9.7 The String class contains the methods for converting, replacing, and splitting strings.

Once a string is created, its contents cannot be changed. The methods toLowerCase, toUpperCase, trim, replace, replaceFirst, and replaceAll return a new string derived from the original string (without changing the original string!). The toLowerCase and toUpperCase methods return a new string by converting all the characters in the string to lowercase or uppercase. The trim method returns a new string by eliminating whitespace characters from both ends of the string. Several versions of the replace methods are provided to replace a character or a substring in the string with a new character or a new substring.

For example,

"Welcome". toLowerCase() returns a new string, welcome. toLowerCase()
"Welcome". toUpperCase() returns a new string, WELCOME. toUpperCase()
"\t Good Night \n". trim() returns a new string, Good Night. trim()
"Welcome". replace('e', 'A') returns a new string, WAlcomA. replace
"Welcome". replaceFirst("e", "AB") returns a new string, WABlcome. replaceFirst
"Welcome". replace("e", "AB") returns a new string, WABlcomAB. replace
"Welcome". replace("el", "AB") returns a new string, WABcome. replace

The split method can be used to extract tokens from a string with the specified delimiters. For example, the following code

split

```
String[] tokens = "Java#HTML#Perl".split("#");
for (int i = 0; i < tokens.length; i++)
  System.out.print(tokens[i] + " ");
```

displays

```
Java HTML Perl
```

9.2.7 Matching, Replacing and Splitting by Patterns

Often you will need to write code that validates user input, such as to check whether the input is a number, a string with all lowercase letters, or a Social Security number. How do you write this type of code? A simple and effective way to accomplish this task is to use the regular expression.

why regular expression?

regular expression

regex

A *regular expression* (abbreviated *regex*) is a string that describes a pattern for matching a set of strings. You can match, replace, or split a string by specifying a pattern. This is an extremely useful and powerful feature.

matches(regex)

Let us begin with the **matches** method in the **String** class. At first glance, the **matches** method is very similar to the **equals** method. For example, the following two statements both evaluate to **true**.

```
"Java".matches("Java");
"Java".equals("Java");
```

However, the **matches** method is more powerful. It can match not only a fixed string, but also a set of strings that follow a pattern. For example, the following statements all evaluate to **true**:

```
"Java is fun".matches("Java.*")
"Java is cool".matches("Java.*")
"Java is powerful".matches("Java.*")
```

Java.* in the preceding statements is a regular expression. It describes a string pattern that begins with Java followed by *any* zero or more characters. Here, the substring **.*** matches any zero or more characters.

The following statement evaluates to **true**.

```
"440-02-4534".matches("\\d{3}-\\d{2}-\\d{4}")
```

Here **\\d** represents a single digit, and **\\d{3}** represents three digits.

The **replaceAll**, **replaceFirst**, and **split** methods can be used with a regular expression. For example, the following statement returns a new string that replaces **$**, **+**, or **#** in **a+b$#c** with the string **NNN**.

replaceAll(regex)

```
String s = "a+b$#c".replaceAll("[$+#]", "NNN");
System.out.println(s);
```

Here the regular expression **[$+#]** specifies a pattern that matches **$**, **+**, or **#**. So, the output is **aNNNbNNNNNNc**.

The following statement splits the string into an array of strings delimited by punctuation marks.

split(regex)

```
String[] tokens = "Java,C?C#,C++".split("[.,:;?]");

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

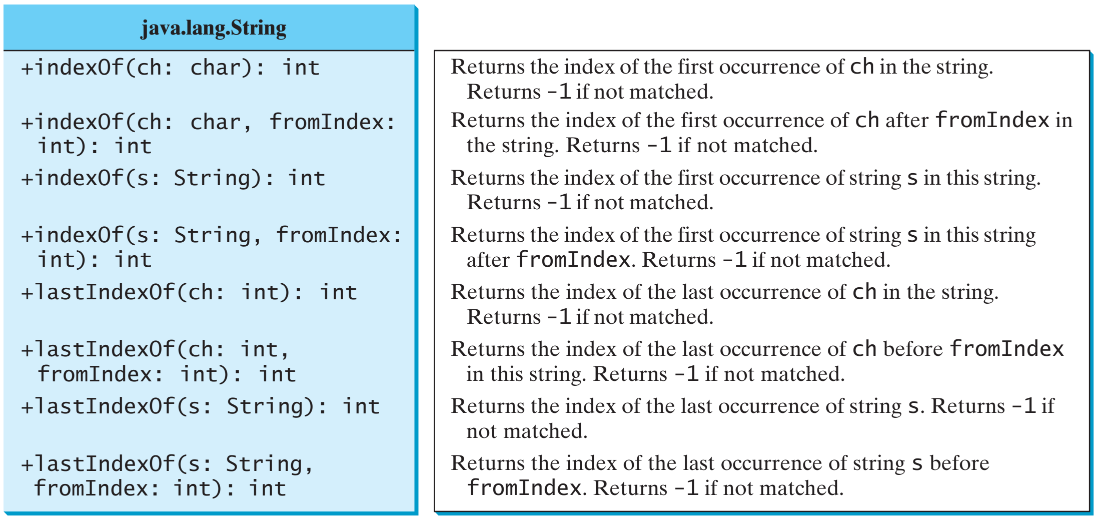
In this example, the regular expression **[.,:;?]** specifies a pattern that matches **.**, **,**, **:**, **;**, or **?**. Each of these characters is a delimiter for splitting the string. Thus, the string is split into **Java**, **C**, **C#**, and **C++**, which are stored in array **tokens**.

further studies

Regular expression patterns are complex for beginning students to understand. For this reason, simple patterns are introduced in this section. Please refer to Supplement III.H, Regular Expressions, to learn more about these patterns.

9.2.8 Finding a Character or a Substring in a String

The **String** class provides several overloaded **indexOf** and **lastIndexOf** methods to find a character or a substring in a string, as shown in Figure 9.8.



java.lang.String	
+indexOf(ch: char): int	Returns the index of the first occurrence of ch in the string. Returns -1 if not matched.
+indexOf(ch: char, fromIndex: int): int	Returns the index of the first occurrence of ch after fromIndex in the string. Returns -1 if not matched.
+indexOf(s: String): int	Returns the index of the first occurrence of string s in this string. Returns -1 if not matched.
+indexOf(s: String, fromIndex: int): int	Returns the index of the first occurrence of string s in this string after fromIndex. Returns -1 if not matched.
+lastIndexOf(ch: int): int	Returns the index of the last occurrence of ch in the string. Returns -1 if not matched.
+lastIndexOf(ch: int, fromIndex: int): int	Returns the index of the last occurrence of ch before fromIndex in this string. Returns -1 if not matched.
+lastIndexOf(s: String): int	Returns the index of the last occurrence of string s. Returns -1 if not matched.
+lastIndexOf(s: String, fromIndex: int): int	Returns the index of the last occurrence of string s before fromIndex. Returns -1 if not matched.

FIGURE 9.8 The `String` class contains the methods for matching substrings.

For example,

indexOf

`"Welcome to Java".indexOf('W')` returns `0`.
`"Welcome to Java".indexOf('o')` returns `4`.
`"Welcome to Java".indexOf('o', 5)` returns `9`.
`"Welcome to Java".indexOf("come")` returns `3`.
`"Welcome to Java".indexOf("Java", 5)` returns `11`.
`"Welcome to Java".indexOf("java", 5)` returns `-1`.

lastIndexOf

`"Welcome to Java".lastIndexOf('W')` returns `0`.
`"Welcome to Java".lastIndexOf('o')` returns `9`.
`"Welcome to Java".lastIndexOf('o', 5)` returns `4`.
`"Welcome to Java".lastIndexOf("come")` returns `3`.
`"Welcome to Java".lastIndexOf("Java", 5)` returns `-1`.
`"Welcome to Java".lastIndexOf("Java")` returns `11`.

9.2.9 Conversion between Strings and Arrays

toCharArray

Strings are not arrays, but a string can be converted into an array, and vice versa. To convert a string into an array of characters, use the `toCharArray` method. For example, the following statement converts the string `Java` to an array.

```
char[] chars = "Java".toCharArray();
```

Thus, `chars[0]` is `J`, `chars[1]` is `a`, `chars[2]` is `v`, and `chars[3]` is `a`.

You can also use the `getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)` method to copy a substring of the string from index `srcBegin` to index `srcEnd-1` into a character array `dst` starting from index `dstBegin`. For example, the following code copies a substring `"3720"` in `"CS3720"` from index `2` to index `6-1` into the character array `dst` starting from index `4`.

getChars

```
char[] dst = {'J', 'A', 'V', 'A', '1', '3', '0', '1'};
"CS3720".getChars(2, 6, dst, 4);
```

Thus, `dst` becomes `{'J', 'A', 'V', 'A', '3', '7', '2', '0'}`.

To convert an array of characters into a string, use the String(char[]) constructor or the valueOf(char[]) method. For example, the following statement constructs a string from an array using the String constructor.

```
String str = new String(new char[]{'J', 'a', 'v', 'a'});
```

valueOf

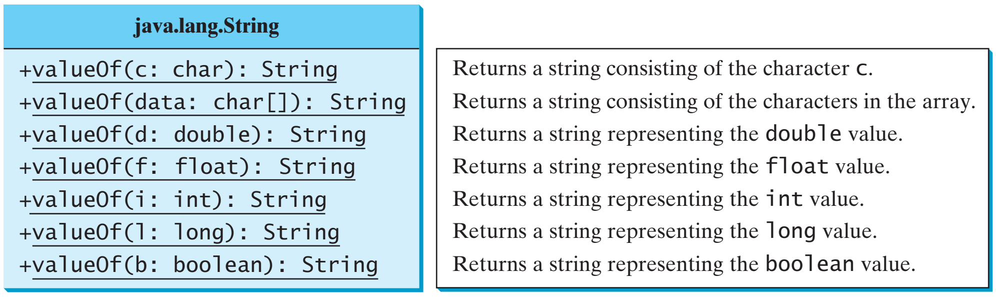
The next statement constructs a string from an array using the valueOf method.

```
String str = String.valueOf(new char[]{'J', 'a', 'v', 'a'});
```

9.2.10 Converting Characters and Numeric Values to Strings

overloaded valueOf

The static valueOf method can be used to convert an array of characters into a string. There are several overloaded versions of the valueOf method that can be used to convert a character and numeric values to strings with different parameter types, char, double, long, int, and float, as shown in Figure 9.9.



java.lang.String	
+valueOf(c: char): String	Returns a string consisting of the character c.
+valueOf(data: char[]): String	Returns a string consisting of the characters in the array.
+valueOf(d: double): String	Returns a string representing the double value.
+valueOf(f: float): String	Returns a string representing the float value.
+valueOf(i: int): String	Returns a string representing the int value.
+valueOf(l: long): String	Returns a string representing the long value.
+valueOf(b: boolean): String	Returns a string representing the boolean value.

Figure 9.9 The String class contains the static methods for creating strings from primitive type values.

For example, to convert a double value 5.44 to a string, use String.valueOf(5.44). The return value is a string consisting of the characters '5', '.', '4', and '4'.

Note

You can use Double.parseDouble(str) or Integer.parseInt(str) to convert a string to a double value or an int value. Double and Integer are two classes in the java.lang package.

9.2.11 Formatting Strings

The String class contains the static format method to create a formatted string. The syntax to invoke this method is:

```
String.format(format, item1, item2, ..., itemk)
```

This method is similar to the printf method except that the format method returns a formatted string, whereas the printf method displays a formatted string. For example,

```
String s = String.format("%7.2f%6d%-4s", 45.556, 14, "AB");
System.out.println(s);
```

displays

```
□□45.56□□□□14AB□□
```

Note that

```
System.out.printf(format, item1, item2, ..., itemk);
```

is equivalent to

```
System.out.printf(
  String.format(format, item1, item2, ..., itemk));
```

where the square box (□) denotes a blank space.

Check Point
MyProgrammingLab™

9.1 Suppose that `s1`, `s2`, `s3`, and `s4` are four strings, given as follows:

```
String s1 = "Welcome to Java";
String s2 = s1;
String s3 = new String("Welcome to Java");
String s4 = "Welcome to Java";
```

What are the results of the following expressions?

a. `s1 == s2`
b. `s2 == s3`
c. `s1.equals(s2)`
d. `s2.equals(s3)`
e. `s1.compareTo(s2)`
f. `s2.compareTo(s3)`
g. `s1 == s4`
h. `s1.charAt(0)`
i. `s1.indexOf('j')`
j. `s1.indexOf("to")`
k. `s1.lastIndexOf('a')`
l. `s1.lastIndexOf("o", 15)`
m. `s1.length()`
n. `s1.substring(5)`
o. `s1.substring(5, 11)`
p. `s1.startsWith("Wel")`
q. `s1.endsWith("Java")`
r. `s1.toLowerCase()`
s. `s1.toUpperCase()`
t. `"Welcome ".trim()`
u. `s1.replace('o', 'T')`
v. `s1.replaceAll("o", "T")`
w. `s1.replaceFirst("o", "T")`
x. `s1.toCharArray()`

9.2 To create the string `Welcome to Java`, you may use a statement like this:

```
String s = "Welcome to Java";
```

or:

```
String s = new String("Welcome to Java");
```

Which one is better? Why?

9.3 Suppose that `s1` and `s2` are two strings. Which of the following statements or expressions are incorrect?

```
String s = new String("new string");
String s3 = s1 + s2;
String s3 = s1 - s2;
s1 == s2;
s1 >= s2;
s1.compareTo(s2);
int i = s1.length();
char c = s1(0);
char c = s1.charAt(s1.length());
```

9.4 What is the printout of the following code?

```
String s1 = "Welcome to Java";
String s2 = s1.replace("o", "abc");
System.out.println(s1);
System.out.println(s2);
```

9.5 Let `s1` be `" Welcome "` and `s2` be `" welcome "`. Write the code for the following statements:

a. Check whether `s1` is equal to `s2` and assign the result to a Boolean variable `isEqual`.

b. Check whether `s1` is equal to `s2`, ignoring case, and assign the result to a Boolean variable `isEqual`.

c. Compare `s1` with `s2` and assign the result to an `int` variable `x`.

d. Compare `s1` with `s2`, ignoring case, and assign the result to an `int` variable `x`.

e. Check whether `s1` has the prefix `AAA` and assign the result to a Boolean variable `b`.

f. Check whether `s1` has the suffix `AAA` and assign the result to a Boolean variable `b`.

g. Assign the length of `s1` to an `int` variable `x`.

h. Assign the first character of `s1` to a `char` variable `x`.

i. Create a new string `s3` that combines `s1` with `s2`.

j. Create a substring of `s1` starting from index `1`.

k. Create a substring of `s1` from index `1` to index `4`.

l. Create a new string `s3` that converts `s1` to lowercase.

m. Create a new string `s3` that converts `s1` to uppercase.

n. Create a new string `s3` that trims blank spaces on both ends of `s1`.

o. Replace all occurrences of the character `e` with `E` in `s1` and assign the new string to `s3`.

p. Split `Welcome to Java and HTML` into an array `tokens` delimited by a space.

q. Assign the index of the first occurrence of the character `e` in `s1` to an `int` variable `x`.

r. Assign the index of the last occurrence of the string `abc` in `s1` to an `int` variable `x`.

9.6 Does any method in the `String` class change the contents of the string?

9.7 Suppose string `s` is created using `new String()`; what is `s.length()`?

9.8 How do you convert a `char`, an array of characters, or a number to a string?

9.9 Why does the following code cause a `NullPointerException`?

```java
public class Test {
  private String text;

  public Test(String s) {
    String text  = s;
  }

  public static void main(String[] args) {
    Test test = new Test("ABC");
    System.out.println(test.text.toLowerCase());
  }
}
```

9.10 What is wrong in the following program?

```java
public class Test {
  String text;

  public void Test(String s) {
    text = s;
  }

  public static void main(String[] args) {
    Test test = new Test("ABC");
    System.out.println(test);
  }
}
```

9.11 Show the output of the following code.

```
public class Test {
  public static void main(String[] args) {
    System.out.println("Hi, ABC, good".matches("ABC "));
    System.out.println("Hi, ABC, good".matches(".*ABC.*"));
    System.out.println("A,B;C".replaceAll(",;", "#"));
    System.out.println("A,B;C".replaceAll("[,;]", "#"));

    String[] tokens = "A,B;C".split("[,;]");
    for (int i = 0; i < tokens.length; i++)
      System.out.print(tokens[i] + " ");
  }
}
```

9.3 Case Study: Checking Palindromes

This section presents a program that checks whether a string is a palindrome.

Key Point

A string is a palindrome if it reads the same forward and backward. The words “mom,” “dad,” and “noon,” for instance, are all palindromes.

VideoNote
Check palindrome

The problem is to write a program that prompts the user to enter a string and reports whether the string is a palindrome. One solution is to check whether the first character in the string is the same as the last character. If so, check whether the second character is the same as the second-to-last character. This process continues until a mismatch is found or all the characters in the string are checked, except for the middle character if the string has an odd number of characters.

To implement this idea, use two variables, say `low` and `high`, to denote the position of the two characters at the beginning and the end in a string `s`, as shown in Listing 9.1 (lines 22, 25). Initially, `low` is `0` and `high` is `s.length() - 1`. If the two characters at these positions match, increment `low` by `1` and decrement `high` by `1` (lines 31–32). This process continues until (`low >= high`) or a mismatch is found.

LISTING 9.1 `CheckPalindrome.java`

```
 1  import java.util.Scanner;
 2
 3  public class CheckPalindrome {
 4    /** Main method */
 5    public static void main(String[] args) {
 6      // Create a Scanner
 7      Scanner input = new Scanner(System.in);
 8
 9      // Prompt the user to enter a string
10      System.out.print("Enter a string: ");
11      String s = input.nextLine();
12
13      if (isPalindrome(s))
```

input string

```
14        System.out.println(s + " is a palindrome");
15      else
16        System.out.println(s + " is not a palindrome");
17    }
18
19    /** Check if a string is a palindrome */
20    public static boolean isPalindrome(String s) {
21      // The index of the first character in the string
22      int low = 0;
23
24      // The index of the last character in the string
25      int high = s.length() - 1;
26
27      while (low < high) {
28        if (s.charAt(low) != s.charAt(high))
29          return false; // Not a palindrome
30
31        low++;
32        high--;
33      }
34
35      return true; // The string is a palindrome
36    }
37  }
```

low index

high index

update indices

```
Enter a string: noon
noon is a palindrome
```

```
Enter a string: moon
moon is not a palindrome
```

The **nextLine()** method in the **Scanner** class (line 11) reads a line into **s**, and then **isPalindrome(s)** checks whether **s** is a palindrome (line 13).

9.4 Case Study: Converting Hexadecimals to Decimals

Key Point

This section presents a program that converts a hexadecimal number into a decimal number.

Section 5.7 gives a program that converts a decimal to a hexadecimal. How do you convert a hex number into a decimal?

Given a hexadecimal number $h_n h_{n-1} h_{n-2} \ldots h_2 h_1 h_0$, the equivalent decimal value is

$$h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \ldots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

For example, the hex number **AB8C** is

$$10 \times 16^3 + 11 \times 16^2 + 8 \times 16^1 + 12 \times 16^0 = 43916$$

Our program will prompt the user to enter a hex number as a string and convert it into a decimal using the following method:

```
public static int hexToDecimal(String hex)
```

A brute-force approach is to convert each hex character into a decimal number, multiply it by 16^i for a hex digit at the **i**'s position, and then add all the items together to obtain the equivalent decimal value for the hex number.

Note that

$$h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \ldots + h_1 \times 16^1 + h_0 \times 16^0$$
$$= (\ldots((h_n \times 16 + h_{n-1}) \times 16 + h_{n-2}) \times 16 + \ldots + h_1) \times 16 + h_0$$

This observation, known as the Horner's algorithm, leads to the following efficient code for converting a hex string to a decimal number:

```
int decimalValue = 0;
for (int i = 0; i < hex.length(); i++) {
  char hexChar = hex.charAt(i);
  decimalValue = decimalValue * 16 + hexCharToDecimal(hexChar);
}
```

Here is a trace of the algorithm for hex number AB8C:

	i	hexChar	hexCharToDecimal(hexChar)	decimalValue
before the loop				0
after the 1st iteration	0	A	10	10
after the 2nd iteration	1	B	11	10 * 16 + 11
after the 3rd iteration	2	8	8	(10 * 16 + 11) * 16 + 8
after the 4th iteration	3	C	12	((10 * 16 + 11) * 16 + 8) * 16 + 12

Listing 9.2 gives the complete program.

LISTING 9.2 HexToDecimalConversion.java

```
import java.util.Scanner;

public class HexToDecimalConversion {
  /** Main method */
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter a string
    System.out.print("Enter a hex number: ");
    String hex = input.nextLine();                                   input string

    System.out.println("The decimal value for hex number "
      + hex + " is " + hexToDecimal(hex.toUpperCase()));              hex to decimal
  }

  public static int hexToDecimal(String hex) {
    int decimalValue = 0;
    for (int i = 0; i < hex.length(); i++) {
      char hexChar = hex.charAt(i);
      decimalValue = decimalValue * 16 + hexCharToDecimal(hexChar);
    }

    return decimalValue;
  }

```

hex char to decimal
to uppercase

```
27    public static int hexCharToDecimal(char ch) {
28      if (ch >= 'A' && ch <= 'F')
29        return 10 + ch - 'A';
30      else // ch is '0', '1', ..., or '9'
31        return ch - '0';
32    }
33  }
```

```
Enter a hex number: AB8C
The decimal value for hex number AB8C is 43916
```

```
Enter a hex number: af71
The decimal value for hex number af71 is 44913
```

The program reads a string from the console (line 11), and invokes the **hexToDecimal** method to convert a hex string to decimal number (line 14). The characters can be in either lowercase or uppercase. They are converted to uppercase before invoking the **hexToDecimal** method.

The **hexToDecimal** method is defined in lines 17–25 to return an integer. The length of the string is determined by invoking **hex.length()** in line 19.

The **hexCharToDecimal** method is defined in lines 27–32 to return a decimal value for a hex character. The character can be in either lowercase or uppercase. Recall that to subtract two characters is to subtract their Unicodes. For example, **'5' - '0'** is **5**.

9.5 The **Character** Class

Key Point

You can create an object for a character using the **Character** *class. A* **Character** *object contains a character value.*

Many methods in the Java API require an object argument. To enable the primitive data values to be treated as objects, Java provides a class for every primitive data type. These classes are **Character**, **Boolean**, **Byte**, **Short**, **Integer**, **Long**, **Float**, and **Double** for **char**, **boolean**, **byte**, **short**, **int**, **long**, **float**, and **double**, respectively. These classes are called *wrapper classes* because each wraps or encapsulates a primitive type value in an object. All these classes are in the **java.lang** package, and they contain useful methods for processing primitive values. This section introduces the **Character** class. The other wrapper classes will be introduced in Chapter 10, Thinking in Objects.

wrapper class

The **Character** class has a constructor and several methods for determining a character's category (uppercase, lowercase, digit, and so on) and for converting characters from uppercase to lowercase, and vice versa, as shown in Figure 9.10.

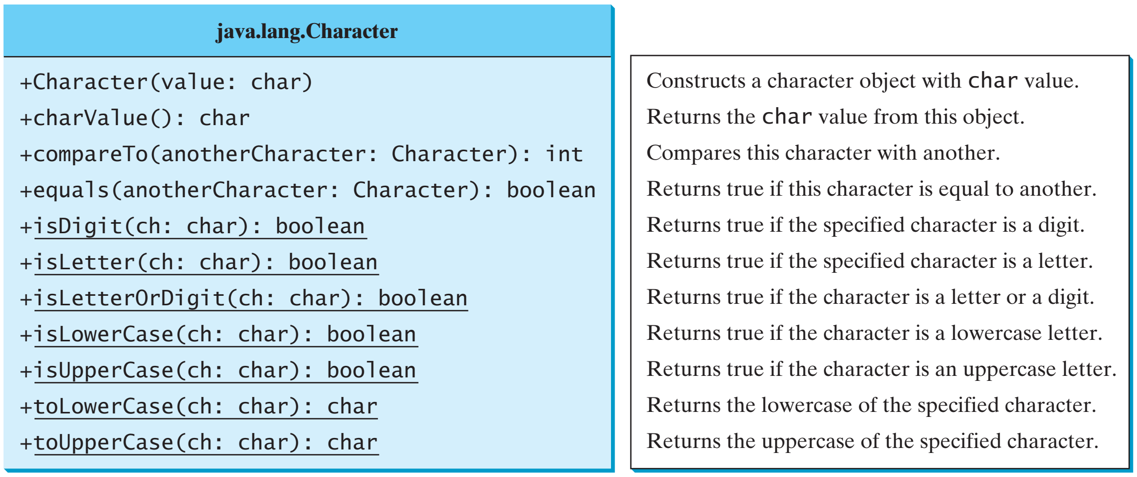
You can create a **Character** object from a **char** value. For example, the following statement creates a **Character** object for the character **a**.

```
Character character = new Character('a');
```

The **charValue** method returns the character value wrapped in the **Character** object. The **compareTo** method compares this character with another character and returns an integer that is the difference between the Unicode of this character and the Unicode of the other character. The **equals** method returns **true** if and only if the two characters are the same. For example, suppose **charObject** is **new Character('b')**:

charObject.compareTo(new Character('a')) returns **1**
charObject.compareTo(new Character('b')) returns **0**
charObject.compareTo(new Character('c')) returns **-1**

```
charObject.compareTo(new Character('d')) returns -2
charObject.equals(new Character('b')) returns true
charObject.equals(new Character('d')) returns false
```



java.lang.Character	
+Character(value: char)	Constructs a character object with `char` value.
+charValue(): char	Returns the `char` value from this object.
+compareTo(anotherCharacter: Character): int	Compares this character with another.
+equals(anotherCharacter: Character): boolean	Returns true if this character is equal to another.
+isDigit(ch: char): boolean	Returns true if the specified character is a digit.
+isLetter(ch: char): boolean	Returns true if the specified character is a letter.
+isLetterOrDigit(ch: char): boolean	Returns true if the character is a letter or a digit.
+isLowerCase(ch: char): boolean	Returns true if the character is a lowercase letter.
+isUpperCase(ch: char): boolean	Returns true if the character is an uppercase letter.
+toLowerCase(ch: char): char	Returns the lowercase of the specified character.
+toUpperCase(ch: char): char	Returns the uppercase of the specified character.

FIGURE 9.10 The `Character` class provides the methods for manipulating a character.

Most of the methods in the `Character` class are static methods. The `isDigit(char ch)` method returns `true` if the character is a digit, and the `isLetter(char ch)` method returns `true` if the character is a letter. The `isLetterOrDigit(char ch)` method returns `true` if the character is a letter or a digit. The `isLowerCase(char ch)` method returns `true` if the character is a lowercase letter, and the `isUpperCase(char ch)` method returns `true` if the character is an uppercase letter. The `toLowerCase(char ch)` method returns the lowercase letter for the character, and the `toUpperCase(char ch)` method returns the uppercase letter for the character.

Now let's write a program that prompts the user to enter a string and counts the number of occurrences of each letter in the string regardless of case.

Here are the steps to solve this problem:

1. Convert all the uppercase letters in the string to lowercase using the `toLowerCase` method in the `String` class.

2. Create an array, say `counts` of `26 int` values, each of which counts the occurrences of a letter. That is, `counts[0]` counts the number of `a`s, `counts[1]` counts the number of `b`s, and so on.

3. For each character in the string, check whether it is a (lowercase) letter. If so, increment the corresponding count in the array.

Listing 9.3 gives the complete program.

LISTING 9.3 CountEachLetter.java

```java
import java.util.Scanner;

public class CountEachLetter {
  /** Main method */
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

```

input string

count letters

count a letter

```
 9       // Prompt the user to enter a string
10       System.out.print("Enter a string: ");
11       String s = input.nextLine();
12
13       // Invoke the countLetters method to count each letter
14       int[] counts = countLetters(s.toLowerCase());
15
16       // Display results
17       for (int i = 0; i < counts.length; i++) {
18         if (counts[i] != 0)
19           System.out.println((char)('a' + i) + " appears  " +
20             counts[i] + ((counts[i] == 1) ? " time" : " times"));
21       }
22     }
23
24     /** Count each letter in the string */
25     public static int[] countLetters(String s ) {
26       int[] counts = new int[26];
27
28       for (int i = 0; i < s.length(); i++) {
29         if (Character.isLetter(s.charAt(i)))
30           counts[s.charAt(i) - 'a']++;
31       }
32
33       return counts;
34     }
35   }
```

```
Enter a string: abababx ↵Enter
a appears  3 times
b appears  3 times
x appears  1 time
```

The main method reads a line (line 11) and counts the number of occurrences of each letter in the string by invoking the **countLetters** method (line 14). Since the case of the letters is ignored, the program uses the **toLowerCase** method to convert the string into all lowercase and pass the new string to the **countLetters** method.

The **countLetters** method (lines 25–34) returns an array of **26** elements. Each element counts the number of occurrences of a letter in the string **s**. The method processes each character in the string. If the character is a letter, its corresponding count is increased by **1**. For example, if the character (**s.charAr(i)**) is **a**, the corresponding count is **counts['a' - 'a']** (i.e., **counts[0]**). If the character is **b**, the corresponding count is **counts['b' - 'a']** (i.e., **counts[1]**), since the Unicode of **b** is **1** more than that of **a**. If the character is **z**, the corresponding count is **counts['z' - 'a']** (i.e., **counts[25]**), since the Unicode of **z** is **25** more than that of **a**.

Check Point

MyProgrammingLab™

9.12 How do you determine whether a character is in lowercase or uppercase?

9.13 How do you determine whether a character is alphanumeric?

9.14 Show the output of the following code.

```
public class Test {
  public static void main(String[] args) {
    String s = "Hi, Good Morning";
    System.out.println(m(s));
  }
```

```
    public static int m(String s) {
      int count = 0;
      for (int i = 0; i < s.length(); i++)
        if (Character.isUpperCase(s.charAt(i)))
          count++;

      return count;
    }
  }
```

9.6 The StringBuilder and StringBuffer Classes

Key Point

The StringBuilder and StringBuffer classes are similar to the String class except that the String class is immutable.

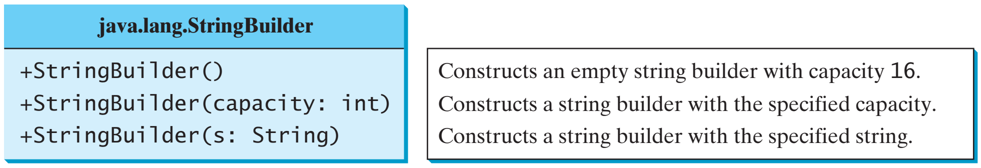
In general, the StringBuilder and StringBuffer classes can be used wherever a string is used. StringBuilder and StringBuffer are more flexible than String. You can add, insert, or append new contents into StringBuilder and StringBuffer objects, whereas the value of a String object is fixed once the string is created.

StringBuilder

The StringBuilder class is similar to StringBuffer except that the methods for modifying the buffer in StringBuffer are *synchronized*, which means that only one task is allowed to execute the methods. Use StringBuffer if the class might be accessed by multiple tasks concurrently. Concurrent programming will be introduced in Chapter 32. Using StringBuilder is more efficient if it is accessed by just a single task. The constructors and methods in StringBuffer and StringBuilder are almost the same. This section covers StringBuilder. You can replace StringBuilder in all occurrences in this section by StringBuffer. The program can compile and run without any other changes.

StringBuilder constructors

The StringBuilder class has three constructors and more than 30 methods for managing the builder and modifying strings in the builder. You can create an empty string builder or a string builder from a string using the constructors, as shown in Figure 9.11.



java.lang.StringBuilder	
+StringBuilder()	Constructs an empty string builder with capacity 16.
+StringBuilder(capacity: int)	Constructs a string builder with the specified capacity.
+StringBuilder(s: String)	Constructs a string builder with the specified string.

FIGURE 9.11 The StringBuilder class contains the constructors for creating instances of StringBuilder.

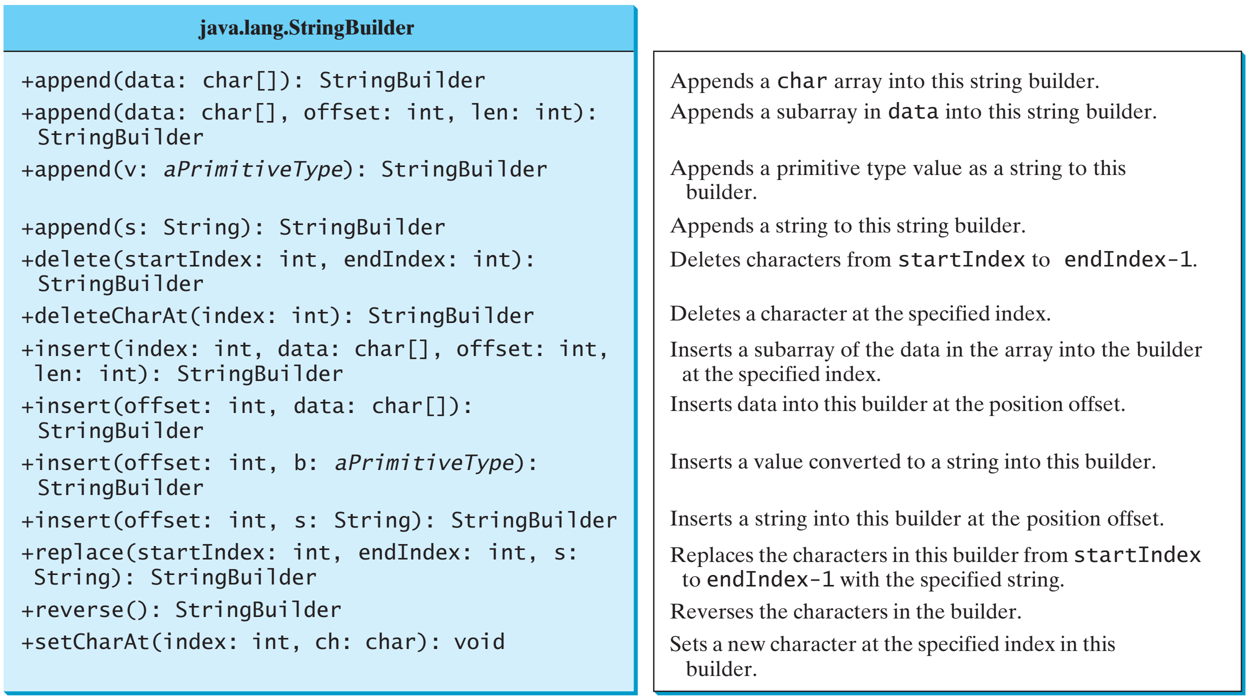
9.6.1 Modifying Strings in the StringBuilder

You can append new contents at the end of a string builder, insert new contents at a specified position in a string builder, and delete or replace characters in a string builder, using the methods listed in Figure 9.12.

The StringBuilder class provides several overloaded methods to append boolean, char, char[], double, float, int, long, and String into a string builder. For example, the following code appends strings and characters into stringBuilder to form a new string, Welcome to Java.

append

```
StringBuilder stringBuilder = new StringBuilder();
stringBuilder.append("Welcome");
stringBuilder.append(' ');
stringBuilder.append("to");
stringBuilder.append(' ');
stringBuilder.append("Java");
```



java.lang.StringBuilder	
+append(data: char[]): StringBuilder	Appends a char array into this string builder.
+append(data: char[], offset: int, len: int): StringBuilder	Appends a subarray in data into this string builder.
+append(v: *aPrimitiveType*): StringBuilder	Appends a primitive type value as a string to this builder.
+append(s: String): StringBuilder	Appends a string to this string builder.
+delete(startIndex: int, endIndex: int): StringBuilder	Deletes characters from startIndex to endIndex-1.
+deleteCharAt(index: int): StringBuilder	Deletes a character at the specified index.
+insert(index: int, data: char[], offset: int, len: int): StringBuilder	Inserts a subarray of the data in the array into the builder at the specified index.
+insert(offset: int, data: char[]): StringBuilder	Inserts data into this builder at the position offset.
+insert(offset: int, b: *aPrimitiveType*): StringBuilder	Inserts a value converted to a string into this builder.
+insert(offset: int, s: String): StringBuilder	Inserts a string into this builder at the position offset.
+replace(startIndex: int, endIndex: int, s: String): StringBuilder	Replaces the characters in this builder from startIndex to endIndex-1 with the specified string.
+reverse(): StringBuilder	Reverses the characters in the builder.
+setCharAt(index: int, ch: char): void	Sets a new character at the specified index in this builder.

FIGURE 9.12 The `StringBuilder` class contains the methods for modifying string builders.

The `StringBuilder` class also contains overloaded methods to insert `boolean`, `char`, `char array`, `double`, `float`, `int`, `long`, and `String` into a string builder. Consider the following code:

insert

```
stringBuilder.insert(11, "HTML and ");
```

Suppose `stringBuilder` contains `Welcome to Java` before the `insert` method is applied. This code inserts `"HTML and "` at position 11 in `stringBuilder` (just before the `J`). The new `stringBuilder` is `Welcome to HTML and Java`.

You can also delete characters from a string in the builder using the two `delete` methods, reverse the string using the `reverse` method, replace characters using the `replace` method, or set a new character in a string using the `setCharAt` method.

For example, suppose `stringBuilder` contains `Welcome to Java` before each of the following methods is applied:

delete `stringBuilder.delete(8, 11)` changes the builder to `Welcome Java`.
deleteCharAt `stringBuilder.deleteCharAt(8)` changes the builder to `Welcome o Java`.
reverse `stringBuilder.reverse()` changes the builder to `avaJ ot emocleW`.
replace `stringBuilder.replace(11, 15, "HTML")` changes the builder to `Welcome to HTML`.
setCharAt `stringBuilder.setCharAt(0, 'w')` sets the builder to `welcome to Java`.

All these modification methods except `setCharAt` do two things:

- Change the contents of the string builder
- Return the reference of the string builder

ignore return value

For example, the following statement

```
StringBuilder stringBuilder1 = stringBuilder.reverse();
```

reverses the string in the builder and assigns the builder's reference to **`stringBuilder1`**. Thus, **`stringBuilder`** and **`stringBuilder1`** both point to the same **`StringBuilder`** object. Recall that a value-returning method can be invoked as a statement, if you are not interested in the return value of the method. In this case, the return value is simply ignored. For example, in the following statement

```
stringBuilder.reverse();
```

the return value is ignored.

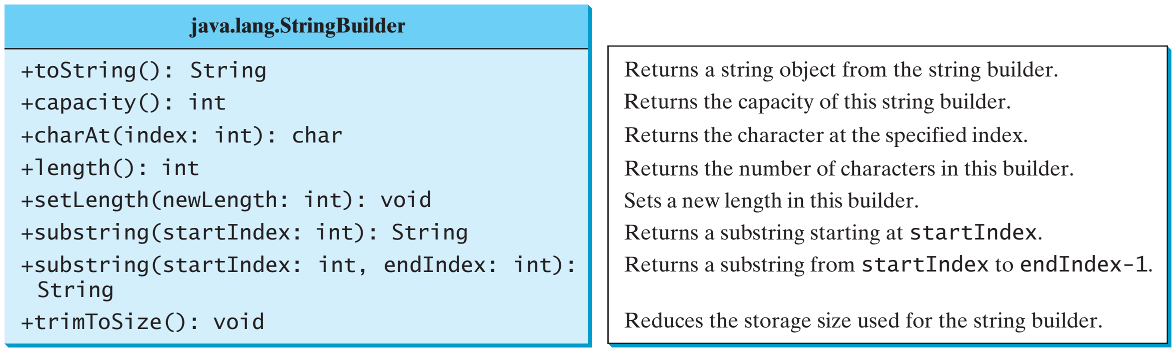
`String` or `StringBuilder`?

Tip

If a string does not require any change, use **`String`** rather than **`StringBuilder`**. Java can perform some optimizations for **`String`**, such as sharing interned strings.

9.6.2 The `toString`, `capacity`, `length`, `setLength`, and `charAt` Methods

The **`StringBuilder`** class provides the additional methods for manipulating a string builder and obtaining its properties, as shown in Figure 9.13.



java.lang.StringBuilder	
`+toString(): String`	Returns a string object from the string builder.
`+capacity(): int`	Returns the capacity of this string builder.
`+charAt(index: int): char`	Returns the character at the specified index.
`+length(): int`	Returns the number of characters in this builder.
`+setLength(newLength: int): void`	Sets a new length in this builder.
`+substring(startIndex: int): String`	Returns a substring starting at `startIndex`.
`+substring(startIndex: int, endIndex: int): String`	Returns a substring from `startIndex` to `endIndex-1`.
`+trimToSize(): void`	Reduces the storage size used for the string builder.

FIGURE 9.13 The **`StringBuilder`** class contains the methods for modifying string builders.

`capacity()`

The **`capacity()`** method returns the current capacity of the string builder. The capacity is the number of characters the string builder is able to store without having to increase its size.

`length()`
`setLength(int)`

The **`length()`** method returns the number of characters actually stored in the string builder. The **`setLength(newLength)`** method sets the length of the string builder. If the **`newLength`** argument is less than the current length of the string builder, the string builder is truncated to contain exactly the number of characters given by the **`newLength`** argument. If the **`newLength`** argument is greater than or equal to the current length, sufficient null characters (**`\u0000`**) are appended to the string builder so that **`length`** becomes the **`newLength`** argument. The **`newLength`** argument must be greater than or equal to **`0`**.

`charAt(int)`

The **`charAt(index)`** method returns the character at a specific **`index`** in the string builder. The index is **`0`** based. The first character of a string builder is at index **`0`**, the next at index **`1`**, and so on. The **`index`** argument must be greater than or equal to **`0`**, and less than the length of the string builder.

length and capacity

Note

The length of the string is always less than or equal to the capacity of the builder. The length is the actual size of the string stored in the builder, and the capacity is the current size of the builder. The builder's capacity is automatically increased if more characters are added to exceed its capacity. Internally, a string builder is an array of characters, so the builder's capacity is the size of the array. If the builder's capacity is exceeded, the array is replaced by a new array. The new array size is `2 * (the previous array size + 1)`.

Tip

initial capacity

You can use `new StringBuilder(initialCapacity)` to create a `StringBuilder` with a specified initial capacity. By carefully choosing the initial capacity, you can make your program more efficient. If the capacity is always larger than the actual length of the builder, the JVM will never need to reallocate memory for the builder. On the other hand, if the capacity is too large, you will waste memory space. You can use the `trimToSize()` method to reduce the capacity to the actual size.

`trimToSize()`

9.6.3 Case Study: Ignoring Nonalphanumeric Characters When Checking Palindromes

Listing 9.1, CheckPalindrome.java, considered all the characters in a string to check whether it was a palindrome. Write a new program that ignores nonalphanumeric characters in checking whether a string is a palindrome.

Here are the steps to solve the problem:

1. Filter the string by removing the nonalphanumeric characters. This can be done by creating an empty string builder, adding each alphanumeric character in the string to a string builder, and returning the string from the string builder. You can use the `isLetterOrDigit(ch)` method in the `Character` class to check whether character `ch` is a letter or a digit.

2. Obtain a new string that is the reversal of the filtered string. Compare the reversed string with the filtered string using the `equals` method.

The complete program is shown in Listing 9.4.

LISTING 9.4 PalindromeIgnoreNonAlphanumeric.java

check palindrome

```java
import java.util.Scanner;

public class PalindromeIgnoreNonAlphanumeric {
  /** Main method */
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter a string
    System.out.print("Enter a string: ");
    String s = input.nextLine();

    // Display result
    System.out.println("Ignoring nonalphanumeric characters, \nis "
      + s + " a palindrome? " + isPalindrome(s));
  }

  /** Return true if a string is a palindrome */
  public static boolean isPalindrome(String s) {
    // Create a new string by eliminating nonalphanumeric chars
    String s1 = filter(s);

    // Create a new string that is the reversal of s1
    String s2 = reverse(s1);

    // Check if the reversal is the same as the original string
27       return s2.equals(s1);
28     }
29
30     /** Create a new string by eliminating nonalphanumeric chars */
31     public static String filter(String s) {
32       // Create a string builder
33       StringBuilder stringBuilder = new StringBuilder();
34
35       // Examine each char in the string to skip alphanumeric char
36       for (int i = 0; i < s.length(); i++) {
37         if (Character.isLetterOrDigit(s.charAt(i))) {
38           stringBuilder.append(s.charAt(i));
39         }
40       }
41
42       // Return a new filtered string
43       return stringBuilder.toString();
44     }
45
46     /** Create a new string by reversing a specified string */
47     public static String reverse(String s) {
48       StringBuilder stringBuilder = new StringBuilder(s);
49       stringBuilder.reverse(); // Invoke reverse in StringBuilder
50       return stringBuilder.toString();
51     }
52   }
```

add letter or digit

```
Enter a string: ab<c>cb?a ↵Enter
Ignoring nonalphanumeric characters,
is ab<c>cb?a a palindrome? true
```

```
Enter a string: abcc><?cab ↵Enter
Ignoring nonalphanumeric characters,
is abcc><?cab a palindrome? false
```

The **filter(String s)** method (lines 31–44) examines each character in string **s** and copies it to a string builder if the character is a letter or a numeric character. The **filter** method returns the string in the builder. The **reverse(String s)** method (lines 47–51) creates a new string that reverses the specified string **s**. The **filter** and **reverse** methods both return a new string. The original string is not changed.

The program in Listing 9.1 checks whether a string is a palindrome by comparing pairs of characters from both ends of the string. Listing 9.4 uses the **reverse** method in the **StringBuilder** class to reverse the string, then compares whether the two strings are equal to determine whether the original string is a palindrome.

Check Point
MyProgrammingLab™

9.15 What is the difference between **StringBuilder** and **StringBuffer**?

9.16 How do you create a string builder from a string? How do you return a string from a string builder?

9.17 Write three statements to reverse a string **s** using the **reverse** method in the **StringBuilder** class.

9.18 Write three statements to delete a substring from a string **s** of **20** characters, starting at index **4** and ending with index **10**. Use the **delete** method in the **StringBuilder** class.

9.19 What is the internal storage for characters in a string and a string builder?

9.20 Suppose that s1 and s2 are given as follows:

```
StringBuilder s1 = new StringBuilder("Java");
StringBuilder s2 = new StringBuilder("HTML");
```

Show the value of s1 after each of the following statements. Assume that the statements are independent.

a. `s1.append(" is fun");`
b. `s1.append(s2);`
c. `s1.insert(2, "is fun");`
d. `s1.insert(1, s2);`
e. `s1.charAt(2);`
f. `s1.length();`
g. `s1.deleteCharAt(3);`
h. `s1.delete(1, 3);`
i. `s1.reverse();`
j. `s1.replace(1, 3, "Computer");`
k. `s1.substring(1, 3);`
l. `s1.substring(2);`

9.21 Show the output of the following program:

```
public class Test {
  public static void main(String[] args) {
    String s = "Java";
    StringBuilder builder = new StringBuilder(s);
    change(s, builder);

    System.out.println(s);
    System.out.println(builder);
  }

  private static void change(String s, StringBuilder builder) {
    s = s + " and HTML";
    builder.append(" and HTML");
  }
}
```

9.7 Command-Line Arguments

Key Point

The main method can receive string arguments from the command line.

Perhaps you have already noticed the unusual header for the main method, which has the parameter args of String[] type. It is clear that args is an array of strings. The main method is just like a regular method with a parameter. You can call a regular method by passing actual parameters. Can you pass arguments to main? Yes, of course you can. In the following examples, the main method in class TestMain is invoked by a method in A.

```
public class A {
  public static void main(String[] args) {
    String[] strings = {"New York",
      "Boston", "Atlanta"};
    TestMain.main(strings);
  }
}
```

```
public class TestMain {
  public static void main(String[] args) {
    for (int i = 0; i < args.length; i++)
      System.out.println(args[i]);
  }
}
```

A main method is just a regular method. Furthermore, you can pass arguments from the command line.

9.7.1 Passing Strings to the `main` Method

You can pass strings to a `main` method from the command line when you run the program. The following command line, for example, starts the program `TestMain` with three strings: `arg0`, `arg1`, and `arg2`:

```
java TestMain arg0 arg1 arg2
```

`arg0`, `arg1`, and `arg2` are strings, but they don't have to appear in double quotes on the command line. The strings are separated by a space. A string that contains a space must be enclosed in double quotes. Consider the following command line:

```
java TestMain "First num" alpha 53
```

It starts the program with three strings: `First num`, `alpha`, and `53`. Since `First num` is a string, it is enclosed in double quotes. Note that `53` is actually treated as a string. You can use `"53"` instead of `53` in the command line.

When the `main` method is invoked, the Java interpreter creates an array to hold the command-line arguments and pass the array reference to `args`. For example, if you invoke a program with `n` arguments, the Java interpreter creates an array like this one:

```
args = new String[n];
```

The Java interpreter then passes `args` to invoke the `main` method.

Note

If you run the program with no strings passed, the array is created with `new String[0]`. In this case, the array is empty with length `0`. `args` references to this empty array. Therefore, `args` is not `null`, but `args.length` is `0`.

9.7.2 Case Study: Calculator

VideoNote
Command-line argument

Suppose you are to develop a program that performs arithmetic operations on integers. The program receives an expression in one string argument. The expression consists of an integer followed by an operator and another integer. For example, to add two integers, use this command:

```
java Calculator "2 + 3"
```

The program will display the following output:

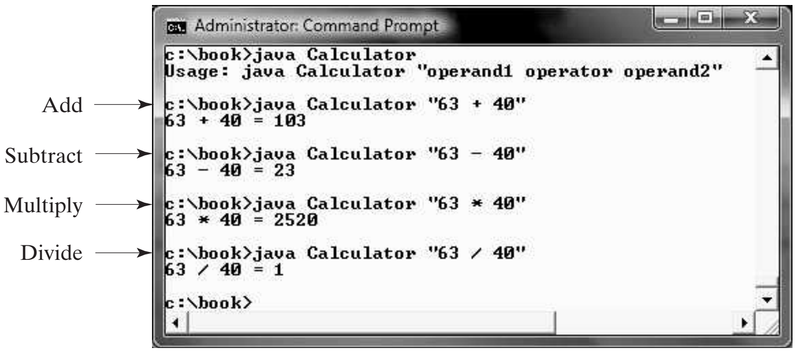
```
2 + 3 = 5
```

Figure 9.14 shows sample runs of the program.

The strings passed to the main program are stored in `args`, which is an array of strings. In this case, we pass the expression as one string. Therefore, the array contains only one element in `args[0]` and `args.length` is `1`.

Here are the steps in the program:

1. Use `args.length` to determine whether the expression has been provided as one argument in the command line. If not, terminate the program using `System.exit(1)`.
2. Split the expression in the string `args[0]` into three tokens in `tokens[0]`, `tokens[1]`, and `tokens[2]`.
3. Perform a binary arithmetic operation on the operands `tokens[0]` and `tokens[2]` using the operator in `tokens[1]`.



```
c:\book>java Calculator
Usage: java Calculator "operand1 operator operand2"

c:\book>java Calculator "63 + 40"
63 + 40 = 103

c:\book>java Calculator "63 - 40"
63 - 40 = 23

c:\book>java Calculator "63 * 40"
63 * 40 = 2520

c:\book>java Calculator "63 / 40"
63 / 40 = 1

c:\book>
```

FIGURE 9.14 The program takes an expression in one argument (`operand1 operator operand2`) from the command line and displays the expression and the result of the arithmetic operation.

The program is shown in Listing 9.5.

LISTING 9.5 Calculator.java

check argument

split string

check operator

```
public class Calculator {
  /** Main method */
  public static void main(String[] args) {
    // Check number of strings passed
    if (args.length != 1) {
      System.out.println(
        "Usage: java Calculator \"operand1 operator operand2\"");
      System.exit(1);
    }

    // The result of the operation
    int result = 0;

    // The result of the operation
    String[] tokens = args[0].split(" ");

    // Determine the operator
    switch (tokens[1].charAt(0)) {
      case '+': result = Integer.parseInt(tokens[0]) +
                         Integer.parseInt(tokens[2]);
                break;
      case '-': result = Integer.parseInt(tokens[0]) -
                         Integer.parseInt(tokens[2]);
                break;
      case '*': result = Integer.parseInt(tokens[0]) *
                         Integer.parseInt(tokens[2]);
                break;
      case '/': result = Integer.parseInt(tokens[0]) /
                         Integer.parseInt(tokens[2]);
    }

    // Display result
    System.out.println(tokens[0] + ' ' + tokens[1] + ' '
      + tokens[2] + " = " + result);
  }
}
```

The expression is passed as a string in one argument and it is split into three parts—`tokens[0]`, `tokens[1]`, and `tokens[2]`—using the `split` method (line 15) with a space as a delimiter.

`Integer.parseInt(tokens[0])` (line 19) converts a digital string into an integer. The string must consist of digits. If it doesn't, the program will terminate abnormally.

For this program to work, the expression must be entered in the form of "operand1 operator operand2". The operands and operator are separated by exactly one space. You can modify the program to accept the expressions in different forms (see Programming Exercise 9.28).

Check Point
MyProgrammingLab™

9.22 This book declares the `main` method as

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

Can it be replaced by one of the following lines?

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

9.23 Show the output of the following program when invoked using

1. **java Test I have a dream**
2. **java Test "1 2 3"**
3. **java Test**

```
public class Test {
  public static void main(String[] args) {
    System.out.println("Number of strings is " + args.length);
    for (int i = 0; i < args.length; i++)
      System.out.println(args[i]);
  }
}
```

Key Terms

interned string 337	wrapper class 350

Chapter Summary

1. Strings are objects encapsulated in the `String` class. A string can be constructed using one of the 13 constructors or simply using a string literal. Java treats a string literal as a `String` object.

2. A `String` object is immutable; its contents cannot be changed. To improve efficiency and save memory, the JVM stores two literal strings that have the same character sequence in a unique object. This unique object is called an *interned string object*.

3. You can get the length of a string by invoking its `length()` method, retrieve a character at the specified `index` in the string using the `charAt(index)` method, and use the `indexOf` and `lastIndexOf` methods to find a character or a substring in a string.

4. You can use the `concat` method to concatenate two strings, or the plus (`+`) operator to concatenate two or more strings.

5. You can use the `substring` method to obtain a substring from the string.

6. You can use the `equals` and `compareTo` methods to compare strings. The `equals` method returns `true` if two strings are equal, and `false` if they are not equal. The `compareTo` method returns `0`, a positive integer, or a negative integer, depending on whether one string is equal to, greater than, or less than the other string.

7. A *regular expression* (abbreviated *regex*) is a string that describes a pattern for matching a set of strings. You can match, replace, or split a string by specifying a pattern.

8. The `Character` class is a wrapper class for a single character. The `Character` class provides useful static methods to determine whether a character is a letter (`isLetter(char)`), a digit (`isDigit(char)`), uppercase (`isUpperCase(char)`), or lowercase (`isLowerCase(char)`).

9. The `StringBuilder` and `StringBuffer` classes can be used to replace the `String` class. The `String` object is immutable, but you can add, insert, or append new contents into `StringBuilder` and `StringBuffer` objects. Use `String` if the string contents do not require any change, and use `StringBuilder` or `StringBuffer` if they might change.

10. You can pass strings to the `main` method from the command line. Strings passed to the `main` program are stored in `args`, which is an array of strings. The first string is represented by `args[0]`, and `args.length` is the number of strings passed.

Test Questions

Do the test questions for this chapter online at www.cs.armstrong.edu/liang/intro9e/test.html.

MyProgrammingLab™

Programming Exercises

Sections 9.2–9.3

***9.1** (*Check SSN*) Write a program that prompts the user to enter a Social Security number in the format DDD-DD-DDDD, where D is a digit. The program displays `Valid SSN` for a correct Social Security number and `Invalid SSN` otherwise.

****9.2** (*Check substrings*) You can check whether a string is a substring of another string by using the `indexOf` method in the `String` class. Write your own method for this function. Write a program that prompts the user to enter two strings, and checks whether the first string is a substring of the second.

****9.3** (*Check password*) Some websites impose certain rules for passwords. Write a method that checks whether a string is a valid password. Suppose the password rules are as follows:

- A password must have at least eight characters.
- A password consists of only letters and digits.
- A password must contain at least two digits.

Write a program that prompts the user to enter a password and displays `Valid Password` if the rules are followed or `Invalid Password` otherwise.

9.4 (*Occurrences of a specified character*) Write a method that finds the number of occurrences of a specified character in a string using the following header:

```
public static int count(String str, char a)
```

For example, `count("Welcome", 'e')` returns `2`. Write a test program that prompts the user to enter a string followed by a character and displays the number of occurrences of the character in the string.

****9.5** (*Occurrences of each digit in a string*) Write a method that counts the occurrences of each digit in a string using the following header:

```
public static int[] count(String s)
```

The method counts how many times a digit appears in the string. The return value is an array of ten elements, each of which holds the count for a digit. For example, after executing `int[] counts = count("12203AB3")`, `counts[0]` is `1`, `counts[1]` is `1`, `counts[2]` is `2`, and `counts[3]` is `2`.

Write a test program that prompts the user to enter a string and displays the number of occurrences of each digit in the string.

***9.6** (*Count the letters in a string*) Write a method that counts the number of letters in a string using the following header:

```
public static int countLetters(String s)
```

Write a test program that prompts the user to enter a string and displays the number of letters in the string.

***9.7** (*Phone keypads*) The international standard letter/number mapping found on the telephone is:

1	2 ABC	3 DEF
4 GHI	5 JKL	6 MNO
7 PQRS	8 TUV	9 WXYZ
	0	

Write a method that returns a number, given an uppercase letter, as follows:

```
public static int getNumber(char uppercaseLetter)
```

Write a test program that prompts the user to enter a phone number as a string. The input number may contain letters. The program translates a letter (upper- or lowercase) to a digit and leaves all other characters intact. Here is a sample run of the program:

```
Enter a string: 1-800-Flowers ↵Enter
1-800-3569377
```

```
Enter a string: 1800flowers
18003569377
```

***9.8** (*Binary to decimal*) Write a method that parses a binary number as a string into a decimal integer. The method header is:

```
public static int binaryToDecimal(String binaryString)
```

For example, binary string 10001 is 17 ($1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2 + 1 = 17$). Therefore, `binaryToDecimal("10001")` returns `17`. Note that `Integer.parseInt("10001", 2)` parses a binary string to a decimal value. Do not use this method in this exercise.

Write a test program that prompts the user to enter a binary string and displays the corresponding decimal integer value.

Section 9.4

****9.9** (*Binary to hex*) Write a method that parses a binary number into a hex number. The method header is:

```
public static String binaryToHex(String binaryValue)
```

Write a test program that prompts the user to enter a binary number and displays the corresponding hexadecimal value.

VideoNote
Number conversion

****9.10** (*Decimal to binary*) Write a method that parses a decimal number into a binary number as a string. The method header is:

```
public static String decimalToBinary(int value)
```

Write a test program that prompts the user to enter a decimal integer value and displays the corresponding binary value.

****9.11** (*Sort characters in a string*) Write a method that returns a sorted string using the following header:

```
public static String sort(String s)
```

For example, `sort("acb")` returns `abc`.

Write a test program that prompts the user to enter a string and displays the sorted string.

****9.12** (*Anagrams*) Write a method that checks whether two words are anagrams. Two words are anagrams if they contain the same letters in any order. For example, `silent` and `listen` are anagrams. The header of the method is:

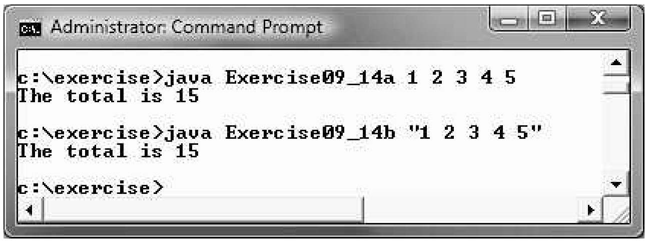
```
public static boolean isAnagram(String s1, String s2)
```

Write a test program that prompts the user to enter two strings and, if they are anagrams, displays `two strings are anagrams`, and displays `two strings are not anagrams` if they are not anagrams.

Section 9.5

***9.13** (*Pass a string to check palindromes*) Rewrite Listing 9.1 by passing the string as a command-line argument.

***9.14** (*Sum integers*) Write two programs. The first program passes an unspecified number of integers as separate strings to the `main` method and displays their total. The second program passes an unspecified number of integers delimited by one space in a string to the `main` method and displays their total. Name the two programs `Exercise9_14a` and `Exercise9_14b`, as shown in Figure 9.15.



```
c:\exercise>java Exercise09_14a 1 2 3 4 5
The total is 15

c:\exercise>java Exercise09_14b "1 2 3 4 5"
The total is 15

c:\exercise>
```

FIGURE 9.15 The program adds all the numbers passed from the command line.

***9.15** (*Find the number of uppercase letters in a string*) Write a program that passes a string to the `main` method and displays the number of uppercase letters in the string.

Comprehensive

****9.16** (*Implement the `String` class*) The `String` class is provided in the Java library. Provide your own implementation for the following methods (name the new class `MyString1`):

```
public MyString1(char[] chars);
public char charAt(int index);
public int length();
public MyString1 substring(int begin, int end);
public MyString1 toLowerCase();
public boolean equals(MyString1 s);
public static MyString1 valueOf(int i);
```

****9.17** (*Guess the capitals*) Write a program that repeatedly prompts the user to enter a capital for a state. Upon receiving the user input, the program reports whether the answer is correct. Assume that `50` states and their capitals are stored in a two-dimensional array, as shown in Figure 9.16. The program prompts the user to answer all states' capitals and displays the total correct count. The user's answer is not case-sensitive.

Alabama	Montgomery
Alaska	Juneau
Arizona	Phoenix
...	...
...	...

FIGURE 9.16 A two-dimensional array stores states and their capitals.

Here is a sample run:

```
What is the capital of Alabama? Montogomery ↵Enter
The correct answer should be Montgomery
What is the capital of Alaska? Juneau ↵Enter
Your answer is correct
What is the capital of Arizona? ...
...
The correct count is 35
```

****9.18** (*Implement the* **`String`** *class*) The `String` class is provided in the Java library. Provide your own implementation for the following methods (name the new class `MyString2`):

```
public MyString2(String s);
public int compare(String s);
public MyString2 substring(int begin);
public MyString2 toUpperCase();
public char[] toChars();
public static MyString2 valueOf(boolean b);
```

***9.19** (*Common prefix*) Write a method that returns the longest common prefix of two strings. For example, the longest common prefix of `distance` and `disinfection` is `dis`. The header of the method is:

```
public static String prefix(String s1, String s2)
```

If the two strings don't have a common prefix, the method returns an empty string.

Write a `main` method that prompts the user to enter two strings and displays their longest common prefix.

9.20 (*Implement the* **`Character`** *class*) The `Character` class is provided in the Java library. Provide your own implementation for this class. Name the new class `MyCharacter`.

****9.21** (*New string* **`split`** *method*) The `split` method in the `String` class returns an array of strings consisting of the substrings split by the delimiters. However, the delimiters are not returned. Implement the following new method that returns an array of strings consisting of the substrings split by the matching delimiters, including the matching delimiters.

```
public static String[] split(String s, String regex)
```

For example, `split("ab#12#453", "#")` returns `ab`, `#`, `12`, `#`, `453` in an array of `String`, and `split("a?b?gf#e", "[?#]")` returns `a`, `b`, `?`, `b`, `gf`, `#`, and `e` in an array of `String`.

****9.22** (*Implement the* **`StringBuilder`** *class*) The `StringBuilder` class is provided in the Java library. Provide your own implementation for the following methods (name the new class `MyStringBuilder1`):

```
public MyStringBuilder1(String s);
public MyStringBuilder1 append(MyStringBuilder1 s);
public MyStringBuilder1 append(int i);
public int length();
public char charAt(int index);
public MyStringBuilder1 toLowerCase();
public MyStringBuilder1 substring(int begin, int end);
public String toString();
```

****9.23** (*Financial: credit card number validation*) Rewrite Programming Exercise 5.31 using a string input for the credit card number. Redesign the program using the following methods:

```
/** Return true if the card number is valid */
public static boolean isValid(String cardNumber)

/** Get the result from Step 2 */
public static int sumOfDoubleEvenPlace(String cardNumber)
```

```
  /** Return this number if it is a single digit; otherwise,
   * return the sum of the two digits */
  public static int getDigit(int number)

  /** Return sum of odd-place digits in number */
  public static int sumOfOddPlace(String cardNumber)
```

****9.24** (*Implement the* **StringBuilder** *class*) The **StringBuilder** class is provided in the Java library. Provide your own implementation for the following methods (name the new class **MyStringBuilder2**):

```
public MyStringBuilder2();
public MyStringBuilder2(char[] chars);
public MyStringBuilder2(String s);
public MyStringBuilder2 insert(int offset, MyStringBuilder2 s);
public MyStringBuilder2 reverse();
public MyStringBuilder2 substring(int begin);
public MyStringBuilder2 toUpperCase();
```

*****9.25** (*Game: hangman*) Write a hangman game that randomly generates a word and prompts the user to guess one letter at a time, as shown in the sample run. Each letter in the word is displayed as an asterisk. When the user makes a correct guess, the actual letter is then displayed. When the user finishes a word, display the number of misses and ask the user whether to continue to play with another word. Declare an array to store words, as follows:

```
// Add any words you wish in this array
String[] words = {"write", "that", ...};
```

```
(Guess) Enter a letter in word ******* > p Enter
(Guess) Enter a letter in word p****** > r Enter
(Guess) Enter a letter in word pr**r** > p Enter
      p is already in the word
(Guess) Enter a letter in word pr**r** > o Enter
(Guess) Enter a letter in word pro*r** > g Enter
(Guess) Enter a letter in word progr** > n Enter
      n is not in the word
(Guess) Enter a letter in word progr** > m Enter
(Guess) Enter a letter in word progr*m > a Enter
The word is program. You missed 1 time
Do you want to guess another word? Enter y or n>
```

****9.26** (*Check ISBN-10*) Use string operations to simplify Programming Exercise 3.9. Enter the first 9 digits of an ISBN number as a string.

VideoNote
Check ISBN-10

***9.27** (*Bioinformatics: find genes*) Biologists use a sequence of the letters A, C, T, and G to model a *genome*. A *gene* is a substring of a genome that starts after a triplet ATG and ends before a triplet TAG, TAA, or TGA. Furthermore, the length of a gene string is a multiple of 3, and the gene does not contain any of the triplets ATG, TAG, TAA, or TGA. Write a program that prompts the user to enter a genome and displays all genes in the genome. If no gene is found in the input sequence, display "no gene is found". Here are the sample runs:

```
Enter a genome string: TTATGTTTTAAGGATGGGGCGTTAGTT Enter
TTT
GGGCGT
```

```
Enter a genome string: TGTGTGTATAT ↵Enter
no gene is found
```

***9.28** (*Calculator*) Revise Listing 9.5, Calculator.java, to accept an expression in which the operands and operator are separated by zero or more spaces. For example, `3+4` and `3 + 4` are acceptable expressions.

***9.29** (*Business: check ISBN-13*) **ISBN-13** is a new standard for identifying books. It uses the 13 digits $d_1d_2d_3d_4d_5d_6d_7d_8d_9d_{10}d_{11}d_{12}d_{13}$. The last digit, d_{13}, is a checksum, which is calculated from the other digits using the following formula:

$$10 - (d_1 + 3d_2 + d_3 + 3d_4 + d_5 + 3d_6 + d_7 + 3d_8 + d_9 + 3d_{10} + d_{11} + 3d_{12}) \% 10$$

If the checksum is `10`, replace it with `0`. Your program should read the input as a string. Here are sample runs:

```
Enter the first 12 digits of an ISBN-13 as a string:
  978013213080 ↵Enter
The ISBN-13 number is 9780132130806
```

```
Enter the first 12 digits of an ISBN-13 as a string:
  978013213079 ↵Enter
The ISBN-13 number is 9780132130790
```

***9.30** (*Capitalize first letter of each word*) Write the following method that returns a new string in which the first letter in each word is capitalized.

```
public static void title(String s)
```

Write a test program that prompts the user to enter a string and invokes this method, and displays the return value from this method. Here is a sample run:

```
Enter a string: london   england 2015 ↵Enter
The new string is: London   England 2015
```

Note that words may be separated by multiple blank spaces.

***9.31** (*Swap case*) Write the following method that returns a new string in which the uppercase letters are changed to lowercase and lowercase letters are changed to uppercase.

```
public static String swapCase(String s)
```

Write a test program that prompts the user to enter a string and invokes this method, and displays the return value from this method. Here is a sample run:

```
Enter a string: I'm here ↵Enter
The new string is: i'M HERE
```

CHAPTER

10

THINKING IN OBJECTS

Objectives

- To create immutable objects from immutable classes to protect the contents of objects (§10.2).
- To determine the scope of variables in the context of a class (§10.3).
- To use the keyword `this` to refer to the calling object itself (§10.4).
- To apply class abstraction to develop software (§10.5).
- To explore the differences between the procedural paradigm and object-oriented paradigm (§10.6).
- To develop classes for modeling composition relationships (§10.7).
- To design programs using the object-oriented paradigm (§§10.8–10.10).
- To design classes that follow the class-design guidelines (§10.11).
- To create objects for primitive values using the wrapper classes (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, and `Boolean`) (§10.12).
- To simplify programming using automatic conversion between primitive types and wrapper class types (§10.13).
- To use the `BigInteger` and `BigDecimal` classes for computing very large numbers with arbitrary precisions (§10.14).

10.1 Introduction

Key Point

The focus of this chapter is on class design and explores the differences between procedural programming and object-oriented programming.

The preceding two chapters introduced objects and classes. You learned how to define classes, create objects, and use objects from several classes in the Java API (e.g., **Date**, **Random**, **String**, **StringBuilder**, and **Scanner**). This book's approach is to teach problem solving and fundamental programming techniques before object-oriented programming. This chapter shows how procedural and object-oriented programming differ. You will see the benefits of object-oriented programming and learn to use it effectively.

We will use several examples to illustrate the advantages of the object-oriented approach. The examples involve designing new classes and using them in applications. We first introduce some language features supporting these examples.

10.2 Immutable Objects and Classes

Key Point

You can define immutable classes to create immutable objects. The contents of immutable objects cannot be changed.

VideoNote
Immutable objects and `this` keyword

immutable object

immutable class

Normally, you create an object and allow its contents to be changed later. However, occasionally it is desirable to create an object whose contents cannot be changed once the object has been created. We call such an object an *immutable object* and its class an *immutable class*. The **String** class, for example, is immutable. If you deleted the **set** method in the **CircleWithPrivateDataFields** class in Listing 8.9, the class would be immutable, because radius is private and cannot be changed without a **set** method.

If a class is immutable, then all its data fields must be private and it cannot contain public **set** methods for any data fields. A class with all private data fields and no mutators is not necessarily immutable. For example, the following **Student** class has all private data fields and no **set** methods, but it is not an immutable class.

Student class

```java
public class Student {
  private int id;
  private String name;
  private java.util.Date dateCreated;

  public Student(int ssn, String newName) {
    id = ssn;
    name = newName;
    dateCreated = new java.util.Date();
  }

  public int getId() {
    return id;
  }

  public String getName() {
    return name;
  }

  public java.util.Date getDateCreated() {
    return dateCreated;
  }
}
```

As shown in the following code, the data field **dateCreated** is returned using the **getDateCreated()** method. This is a reference to a **Date** object. Through this reference, the content for **dateCreated** can be changed.

```
public class Test {
  public static void main(String[] args) {
    Student student = new Student(111223333, "John");
    java.util.Date dateCreated = student.getDateCreated();
    dateCreated.setTime(200000); // Now dateCreated field is changed!
  }
}
```

For a class to be immutable, it must meet the following requirements:

- All data fields must be private.
- There can't be any mutator methods for data fields.
- No accessor methods can return a reference to a data field that is mutable.

Interested readers may refer to Supplement III.AB for an extended discussion on immutable objects.

Check Point
MyProgrammingLab™

10.1 If a class contains only private data fields and no **set** methods, is the class immutable?

10.2 If all the data fields in a class are private and primitive types, and the class doesn't contain any **set** methods, is the class immutable?

10.3 Is the following class immutable?

```
public class A {
  private int[] values;

  public int[] getValues() {
    return values;
  }
}
```

10.3 The Scope of Variables

The scope of instance and static variables is the entire class, regardless of where the variables are declared.

Key Point

Chapter 5, Methods, discussed local variables and their scope rules. Local variables are declared and used inside a method locally. This section discusses the scope rules of all the variables in the context of a class.

class's variables

Instance and static variables in a class are referred to as the *class's variables* or *data fields*. A variable defined inside a method is referred to as a *local variable*. The scope of a class's variables is the entire class, regardless of where the variables are declared. A class's variables and methods can appear in any order in the class, as shown in Figure 10.1a. The exception is when a data field is initialized based on a reference to another data field. In such cases, the

```
public class Circle {
  public double findArea() {
    return radius * radius * Math.PI;
  }

  private double radius = 1;
}
```

(a) The variable **radius** and method **findArea()** can be declared in any order.

```
public class F {
  private int i;
  private int j = i + 1;
}
```

(b) **i** has to be declared before **j** because **j**'s initial value is dependent on **i**.

FIGURE 10.1 Members of a class can be declared in any order, with one exception.

other data field must be declared first, as shown in Figure 10.1b. For consistency, this book declares data fields at the beginning of the class.

You can declare a class's variable only once, but you can declare the same variable name in a method many times in different nonnesting blocks.

hidden variables

If a local variable has the same name as a class's variable, the local variable takes precedence and the class's variable with the same name is *hidden*. For example, in the following program, **x** is defined both as an instance variable and as a local variable in the method.

```
public class F {
  private int x = 0; // Instance variable
  private int y = 0;

  public F() {
  }

  public void p() {
    int x = 1; // Local variable
    System.out.println("x = " + x);
    System.out.println("y = " + y);
  }
}
```

What is the printout for **f.p()**, where **f** is an instance of **F**? The printout for **f.p()** is **1** for **x** and **0** for **y**. Here is why:

- **x** is declared as a data field with the initial value of **0** in the class, but it is also declared in the method **p()** with an initial value of **1**. The latter **x** is referenced in the **System.out.println** statement.
- **y** is declared outside the method **p()**, but **y** is accessible inside the method.

Tip
To avoid confusion and mistakes, do not use the names of instance or static variables as local variable names, except for method parameters.

Check Point
MyProgrammingLab™

10.4 What is the output of the following program?

```
public class Test {
  private static int i = 0;
  private static int j = 0;

  public static void main(String[] args) {
    int i = 2;
    int k = 3;

    {
      int j = 3;
      System.out.println("i + j is " + i + j);
    }

    k = i + j;
    System.out.println("k is " + k);
    System.out.println("j is " + j);
  }
}
```

10.4 The this Reference

*The keyword **this** refers to the object itself. It can also be used inside a constructor to invoke another constructor of the same class.*

Key Point

this keyword

The **this** *keyword* is the name of a reference that an object can use to refer to itself. You can use the **this** keyword to refer to the object's instance members. For example, the following code in (a) uses **this** to reference the object's **radius** and invokes its **getArea()** method explicitly. The **this** reference is normally omitted, as shown in (b). However, the **this** reference is needed to reference hidden data fields or invoke an overloaded constructor.

```
public class Circle {
  private double radius;

  ...

  public double getArea() {
    return this.radius * this.radius
      * Math.PI;
  }

  public String toString() {
    return "radius: " + this.radius
      + "area: " + this.getArea();
  }
}
```

(a)

Equivalent

```
public class Circle {
  private double radius;

  ...

  public double getArea() {
    return radius * radius * Math.PI;
  }

  public String toString() {
    return "radius: " + radius
      + "area: " + getArea();
  }
}
```

(b)

10.4.1 Using this to Reference Hidden Data Fields

hidden data fields

The **this** keyword can be used to reference a class's *hidden data fields*. For example, a data-field name is often used as the parameter name in a **set** method for the data field. In this case, the data field is hidden in the **set** method. You need to reference the hidden data-field name in the method in order to set a new value to it. A hidden static variable can be accessed simply by using the **ClassName.staticVariable** reference. A hidden instance variable can be accessed by using the keyword **this**, as shown in Figure 10.2a.

```
public class F {
  private int i = 5;
  private static double k = 0;

  public void setI(int i) {
    this.i = i;
  }

  public static void setK(double k) {
    F.k = k;
  }

  // Other methods omitted
}
```

(a)

```
Suppose that f1 and f2 are two objects of F.

Invoking f1.setI(10) is to execute
   this.i = 10, where this refers f1

Invoking f2.setI(45) is to execute
   this.i = 45, where this refers f2

Invoking F.setK(33) is to execute
   F.k = 33. setK is a static method
```

(b)

FIGURE 10.2 The keyword **this** refers to the calling object that invokes the method.

The **this** keyword gives us a way to refer to the object that invokes an instance method. To invoke **f1.setI(10)**, **this.i = i** is executed, which assigns the value of parameter **i** to the data field **i** of this calling object **f1**. The keyword **this** refers to the object that invokes the instance method **setI**, as shown in Figure 10.2b. The line **F.k = k** means that the value in parameter **k** is assigned to the static data field **k** of the class, which is shared by all the objects of the class.

10.4.2 Using **this** to Invoke a Constructor

The **this** keyword can be used to invoke another constructor of the same class. For example, you can rewrite the **Circle** class as follows:

```java
public class Circle {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }

  public Circle() {
    this(1.0);
  }

  ...
}
```

The **this** keyword is used to reference the hidden data field radius of the object being constructed.

The **this** keyword is used to invoke another constructor.

The line **this(1.0)** in the second constructor invokes the first constructor with a **double** value argument.

Note

Java requires that the **this(arg-list)** statement appear first in the constructor before any other executable statements.

Tip

If a class has multiple constructors, it is better to implement them using **this(arg-list)** as much as possible. In general, a constructor with no or fewer arguments can invoke a constructor with more arguments using **this(arg-list)**. This syntax often simplifies coding and makes the class easier to read and to maintain.

10.5 Describe the role of the **this** keyword.

10.6 What is wrong in the following code?

```java
public class C {
  private int p;

  public C() {
    System.out.println("C's no-arg constructor invoked");
    this(0);
  }

  public C(int p) {
    p = p;
  }

  public void setP(int p) {
    p = p;
  }
}
```

10.7 What is wrong in the following code?

```
public class Test {
  private int id;

  public void m1() {
    this.id = 45;
  }

  public void m2() {
    Test.id = 45;
  }
}
```

10.5 Class Abstraction and Encapsulation

Key Point

Class abstraction *is the separation of class implementation from the use of a class. The details of implementation are encapsulated and hidden from the user. This is known as* class encapsulation*.*

In Chapter 5, you learned about method abstraction and used it in stepwise refinement. Java provides many levels of abstraction, and *class abstraction* separates class implementation from how the class is used. The creator of a class describes the functions of the class and lets the user know how the class can be used. The collection of methods and fields that are accessible from outside the class, together with the description of how these members are expected to behave, serves as the *class's contract*. As shown in Figure 10.3, the user of the class does not need to know how the class is implemented. The details of implementation are encapsulated and hidden from the user. This is called *class encapsulation*. For example, you can create a `Circle` object and find the area of the circle without knowing how the area is computed. For this reason, a class is also known as an *abstract data type* (ADT).

class abstraction

class's contract

class encapsulation

abstract data type

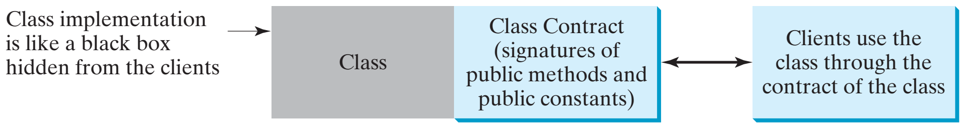

Class implementation is like a black box hidden from the clients
Class
Class Contract (signatures of public methods and public constants)
Clients use the class through the contract of the class

FIGURE 10.3 Class abstraction separates class implementation from the use of the class.

Class abstraction and encapsulation are two sides of the same coin. Many real-life examples illustrate the concept of class abstraction. Consider, for instance, building a computer system. Your personal computer has many components—a CPU, memory, disk, motherboard, fan, and so on. Each component can be viewed as an object that has properties and methods. To get the components to work together, you need know only how each component is used and how it interacts with the others. You don't need to know how the components work internally. The internal implementation is encapsulated and hidden from you. You can build a computer without knowing how a component is implemented.

The computer-system analogy precisely mirrors the object-oriented approach. Each component can be viewed as an object of the class for the component. For example, you might have a class that models all kinds of fans for use in a computer, with properties such as fan size and speed and methods such as start and stop. A specific fan is an instance of this class with specific property values.

As another example, consider getting a loan. A specific loan can be viewed as an object of a `Loan` class. The interest rate, loan amount, and loan period are its data properties, and

VideoNote
The Loan class

computing the monthly payment and total payment are its methods. When you buy a car, a loan object is created by instantiating the class with your loan interest rate, loan amount, and loan period. You can then use the methods to find the monthly payment and total payment of your loan. As a user of the **Loan** class, you don't need to know how these methods are implemented.

Listing 2.8, ComputeLoan.java, presented a program for computing loan payments. That program cannot be reused in other programs because the code for computing the payments is in the **main** method. One way to fix this problem is to define static methods for computing the monthly payment and total payment. However, this solution has limitations. Suppose you wish to associate a date with the loan. There is no good way to tie a date with a loan without using objects. The traditional procedural programming paradigm is action-driven, and data are separated from actions. The object-oriented programming paradigm focuses on objects, and actions are defined along with the data in objects. To tie a date with a loan, you can define a loan class with a date along with other of the loan's properties as data fields. A loan object now contains data and actions for manipulating and processing data, and the loan data and actions are integrated in one object. Figure 10.4 shows the UML class diagram for the **Loan** class.

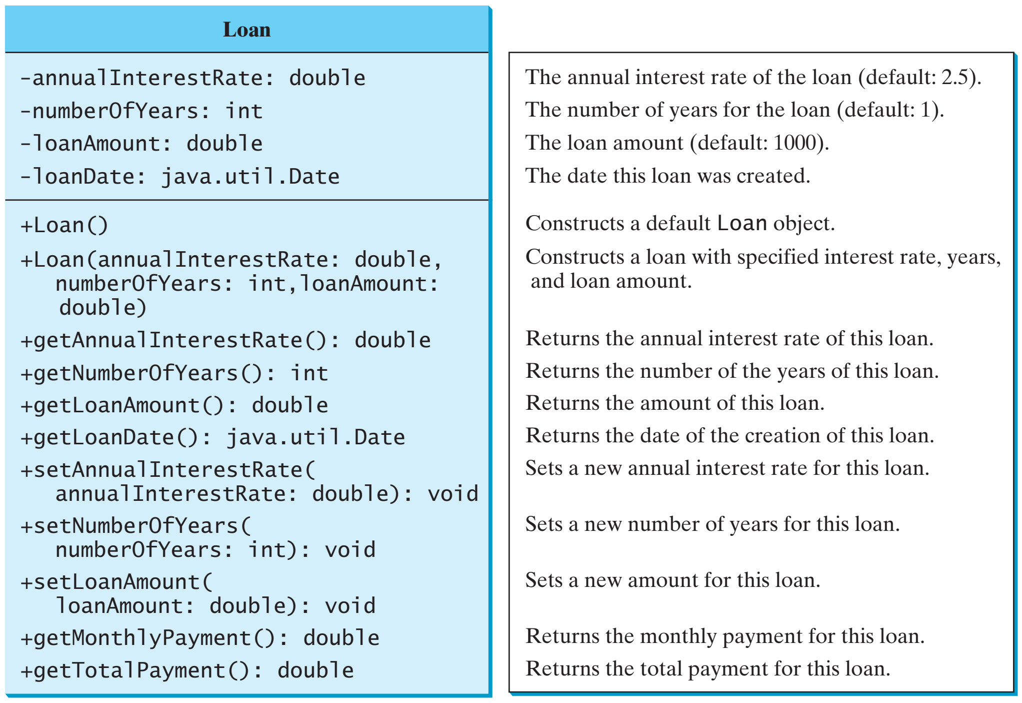

Loan
-annualInterestRate: double
-numberOfYears: int
-loanAmount: double
-loanDate: java.util.Date
+Loan()
+Loan(annualInterestRate: double, numberOfYears: int,loanAmount: double)
+getAnnualInterestRate(): double
+getNumberOfYears(): int
+getLoanAmount(): double
+getLoanDate(): java.util.Date
+setAnnualInterestRate(annualInterestRate: double): void
+setNumberOfYears(numberOfYears: int): void
+setLoanAmount(loanAmount: double): void
+getMonthlyPayment(): double
+getTotalPayment(): double
The annual interest rate of the loan (default: 2.5).
The number of years for the loan (default: 1).
The loan amount (default: 1000).
The date this loan was created.
Constructs a default Loan object.
Constructs a loan with specified interest rate, years, and loan amount.
Returns the annual interest rate of this loan.
Returns the number of the years of this loan.
Returns the amount of this loan.
Returns the date of the creation of this loan.
Sets a new annual interest rate for this loan.
Sets a new number of years for this loan.
Sets a new amount for this loan.
Returns the monthly payment for this loan.
Returns the total payment for this loan.

FIGURE 10.4 The **Loan** class models the properties and behaviors of loans.

The UML diagram in Figure 10.4 serves as the contract for the **Loan** class. Throughout this book, you will play the roles of both class user and class developer. Remember that a class user can use the class without knowing how the class is implemented.

Assume that the **Loan** class is available. The program in Listing 10.1 uses that class.

LISTING 10.1 TestLoanClass.java

```java
import java.util.Scanner;

public class TestLoanClass {
  /** Main method */
  public static void main(String[] args) {
```

```
6        // Create a Scanner
7        Scanner input = new Scanner(System.in);
8
9        // Enter annual interest rate
10       System.out.print(
11         "Enter annual interest rate, for example, 8.25: ");
12       double annualInterestRate = input.nextDouble();
13
14       // Enter number of years
15       System.out.print("Enter number of years as an integer: ");
16       int numberOfYears = input.nextInt();
17
18       // Enter loan amount
19       System.out.print("Enter loan amount, for example, 120000.95: ");
20       double loanAmount = input.nextDouble();
21
22       // Create a Loan object
23       Loan loan =
24         new Loan(annualInterestRate, numberOfYears, loanAmount);
25
26       // Display loan date, monthly payment, and total payment
27       System.out.printf("The loan was created on %s\n" +
28         "The monthly payment is %.2f\nThe total payment is %.2f\n",
29         loan.getLoanDate().toString(), loan.getMonthlyPayment(),
30         loan.getTotalPayment());
31     }
32   }
```

create Loan object

invoke instance method

invoke instance method

```
Enter annual interest rate, for example, 8.25: 2.5 ↵Enter
Enter number of years as an integer: 5 ↵Enter
Enter loan amount, for example, 120000.95: 1000 ↵Enter
The loan was created on Sat Jun 16 21:12:50 EDT 2012
The monthly payment is 17.75
The total payment is 1064.84
```

The **main** method reads the interest rate, the payment period (in years), and the loan amount; creates a **Loan** object; and then obtains the monthly payment (line 29) and the total payment (line 30) using the instance methods in the **Loan** class.

The **Loan** class can be implemented as in Listing 10.2.

LISTING 10.2 Loan.java

```
public class Loan {
  private double annualInterestRate;
  private int numberOfYears;
  private double loanAmount;
  private java.util.Date loanDate;

  /** Default constructor */
  public Loan() {
    this(2.5, 1, 1000);
  }

  /** Construct a loan with specified annual interest rate,
      number of years, and loan amount
    */
```

no-arg constructor

constructor

```
  public Loan(double annualInterestRate, int numberOfYears,
      double loanAmount) {
    this.annualInterestRate = annualInterestRate;
    this.numberOfYears = numberOfYears;
    this.loanAmount = loanAmount;
    loanDate = new java.util.Date();
  }

  /** Return annualInterestRate */
  public double getAnnualInterestRate() {
    return annualInterestRate;
  }

  /** Set a new annualInterestRate */
  public void setAnnualInterestRate(double annualInterestRate) {
    this.annualInterestRate = annualInterestRate;
  }

  /** Return numberOfYears */
  public int getNumberOfYears() {
    return numberOfYears;
  }

  /** Set a new numberOfYears */
  public void setNumberOfYears(int numberOfYears) {
    this.numberOfYears = numberOfYears;
  }

  /** Return loanAmount */
  public double getLoanAmount() {
    return loanAmount;
  }

  /** Set a new loanAmount */
  public void setLoanAmount(double loanAmount) {
    this.loanAmount = loanAmount;
  }

  /** Find monthly payment */
  public double getMonthlyPayment() {
    double monthlyInterestRate = annualInterestRate / 1200;
    double monthlyPayment = loanAmount * monthlyInterestRate / (1 -
      (1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12)));
    return monthlyPayment;
  }

  /** Find total payment */
  public double getTotalPayment() {
    double totalPayment = getMonthlyPayment() * numberOfYears * 12;
    return totalPayment;
  }

  /** Return loan date */
  public java.util.Date getLoanDate() {
    return loanDate;
  }
}
```

From a class developer's perspective, a class is designed for use by many different customers. In order to be useful in a wide range of applications, a class should provide a variety of ways for customization through constructors, properties, and methods.

The **Loan** class contains two constructors, four **get** methods, three **set** methods, and the methods for finding the monthly payment and the total payment. You can construct a **Loan** object by using the no-arg constructor or the constructor with three parameters: annual interest rate, number of years, and loan amount. When a loan object is created, its date is stored in the **loanDate** field. The **getLoanDate** method returns the date. The three **get** methods—**getAnnualInterest**, **getNumberOfYears**, and **getLoanAmount**—return the annual interest rate, payment years, and loan amount, respectively. All the data properties and methods in this class are tied to a specific instance of the **Loan** class. Therefore, they are instance variables and methods.

Important Pedagogical Tip

Use the UML diagram for the **Loan** class shown in Figure 10.4 to write a test program that uses the **Loan** class even though you don't know how the **Loan** class is implemented. This has three benefits:

- It demonstrates that developing a class and using a class are two separate tasks.
- It enables you to skip the complex implementation of certain classes without interrupting the sequence of this book.
- It is easier to learn how to implement a class if you are familiar with it by using the class.

For all the class examples from now on, create an object from the class and try using its methods before turning your attention to its implementation.

Check Point

MyProgrammingLab™

10.8 If you redefine the **Loan** class in Listing 10.2 without **set** methods, is the class immutable?

10.6 Object-Oriented Thinking

Key Point

The procedural paradigm focuses on designing methods. The object-oriented paradigm couples data and methods together into objects. Software design using the object-oriented paradigm focuses on objects and operations on objects.

Chapters 1–7 introduced fundamental programming techniques for problem solving using loops, methods, and arrays. Knowing these techniques lays a solid foundation for object-oriented programming. Classes provide more flexibility and modularity for building reusable software. This section improves the solution for a problem introduced in Chapter 3 using the object-oriented approach. From these improvements, you will gain insight into the differences between procedural and object-oriented programming and see the benefits of developing reusable code using objects and classes.

Listing 3.5, ComputeAndInterpretBMI.java, presented a program for computing body mass index. The code cannot be reused in other programs, because the code is in the **main** method. To make it reusable, define a static method to compute body mass index as follows:

```
public static double getBMI(double weight, double height)
```

This method is useful for computing body mass index for a specified weight and height. However, it has limitations. Suppose you need to associate the weight and height with a person's name and birth date. You could declare separate variables to store these values, but these values would not be tightly coupled. The ideal way to couple them is to create an object that contains them all. Since these values are tied to individual objects, they should be stored in instance data fields. You can define a class named **BMI** as shown in Figure 10.5.

VideoNote
The BMI class

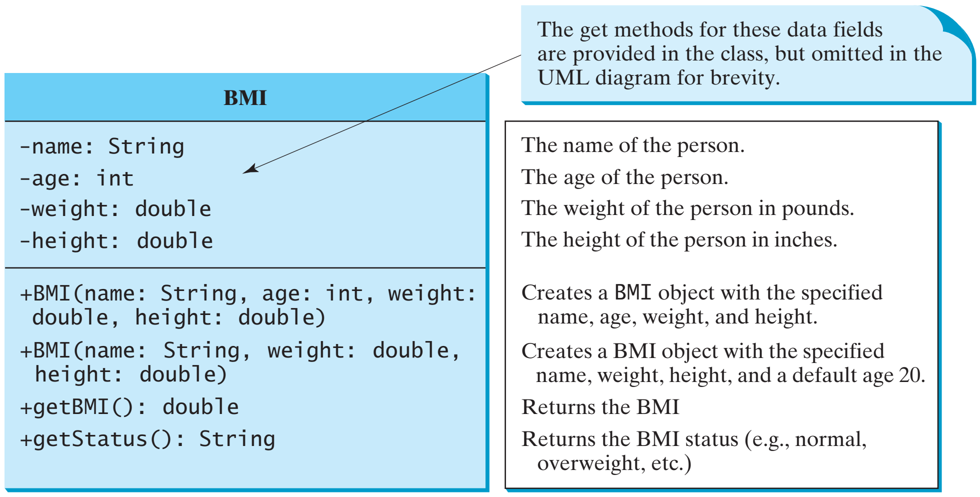

The get methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.
BMI
-name: String
-age: int
-weight: double
-height: double
+BMI(name: String, age: int, weight: double, height: double)
+BMI(name: String, weight: double, height: double)
+getBMI(): double
+getStatus(): String
The name of the person.
The age of the person.
The weight of the person in pounds.
The height of the person in inches.
Creates a BMI object with the specified name, age, weight, and height.
Creates a BMI object with the specified name, weight, height, and a default age 20.
Returns the BMI
Returns the BMI status (e.g., normal, overweight, etc.)

FIGURE 10.5 The `BMI` class encapsulates BMI information.

Assume that the `BMI` class is available. Listing 10.3 gives a test program that uses this class.

LISTING 10.3 UseBMIClass.java

create an object
invoke instance method

create an object
invoke instance method

```java
 1  public class UseBMIClass {
 2    public static void main(String[] args) {
 3      BMI bmi1 = new BMI("Kim Yang", 18, 145, 70);
 4      System.out.println("The BMI for " + bmi1.getName() + " is "
 5        + bmi1.getBMI() + " " + bmi1.getStatus());
 6
 7      BMI bmi2 = new BMI("Susan King", 215, 70);
 8      System.out.println("The BMI for " + bmi2.getName() + " is "
 9        + bmi2.getBMI() + " " + bmi2.getStatus());
10    }
11  }
```

```
The BMI for Kim Yang is 20.81 Normal
The BMI for Susan King is 30.85 Obese
```

Line 3 creates the object `bmi1` for `Kim Yang` and line 7 creates the object `bmi2` for `Susan King`. You can use the instance methods `getName()`, `getBMI()`, and `getStatus()` to return the BMI information in a `BMI` object.

The `BMI` class can be implemented as in Listing 10.4.

LISTING 10.4 BMI.java

```java
public class BMI {
  private String name;
  private int age;
  private double weight; // in pounds
  private double height; // in inches
  public static final double KILOGRAMS_PER_POUND = 0.45359237;
  public static final double METERS_PER_INCH = 0.0254;

  public BMI(String name, int age, double weight, double height) {
    this.name = name;
    this.age = age;
    this.weight = weight;
    this.height = height;
  }

  public BMI(String name, double weight, double height) {
    this(name, 20, weight, height);
  }

  public double getBMI() {
    double bmi = weight * KILOGRAMS_PER_POUND /
      ((height * METERS_PER_INCH) * (height * METERS_PER_INCH));
    return Math.round(bmi * 100) / 100.0;
  }

  public String getStatus() {
    double bmi = getBMI();
    if (bmi < 18.5)
      return "Underweight";
    else if (bmi < 25)
      return "Normal";
    else if (bmi < 30)
      return "Overweight";
    else
      return "Obese";
  }

  public String getName() {
    return name;
  }

  public int getAge() {
    return age;
  }

  public double getWeight() {
    return weight;
  }

  public double getHeight() {
    return height;
  }
}
```

constructor

constructor

getBMI

getStatus

The mathematical formula for computing the BMI using weight and height is given in Section 3.9. The instance method **getBMI()** returns the BMI. Since the weight and height are instance data fields in the object, the **getBMI()** method can use these properties to compute the BMI for the object.

The instance method **getStatus()** returns a string that interprets the BMI. The interpretation is also given in Section 3.9.

procedural vs. object-oriented paradigms

This example demonstrates the advantages of the object-oriented paradigm over the procedural paradigm. The procedural paradigm focuses on designing methods. The object-oriented paradigm couples data and methods together into objects. Software design using the object-oriented paradigm focuses on objects and operations on objects. The object-oriented approach combines the power of the procedural paradigm with an added dimension that integrates data with operations into objects.

In procedural programming, data and operations on the data are separate, and this methodology requires sending data to methods. Object-oriented programming places data and the operations that pertain to them in an object. This approach solves many of the problems inherent in procedural programming. The object-oriented programming approach organizes programs in a way that mirrors the real world, in which all objects are associated with both attributes and activities. Using objects improves software reusability and makes programs easier to develop and easier to maintain. Programming in Java involves thinking in terms of objects; a Java program can be viewed as a collection of cooperating objects.

Check Point

MyProgrammingLab™

10.9 Is the `BMI` class defined in Listing 10.4 immutable?

10.7 Object Composition

Key Point

An object can contain another object. The relationship between the two is called composition.

In Listing 10.2, you defined the `Loan` class to contain a `Date` data field. The relationship between `Loan` and `Date` is composition. In Listing 10.4, you defined the `BMI` class to contain a `String` data field. The relationship between `BMI` and `String` is composition.

aggregation

has-a relationship

Composition is actually a special case of the aggregation relationship. Aggregation models *has-a* relationships and represents an ownership relationship between two objects. The owner object is called an *aggregating object* and its class an *aggregating class*. The subject object is called an *aggregated object* and its class an *aggregated class*.

composition

An object may be owned by several other aggregating objects. If an object is exclusively owned by an aggregating object, the relationship between them is referred to as *composition*. For example, "a student has a name" is a composition relationship between the `Student` class and the `Name` class, whereas "a student has an address" is an aggregation relationship between the `Student` class and the `Address` class, because an address may be shared by several students. In UML notation, a filled diamond is attached to an aggregating class (e.g., `Student`) to denote the composition relationship with an aggregated class (e.g., `Name`), and an empty diamond is attached to an aggregating class (e.g., `Student`) to denote the aggregation relationship with an aggregated class (e.g., `Address`), as shown in Figure 10.6.

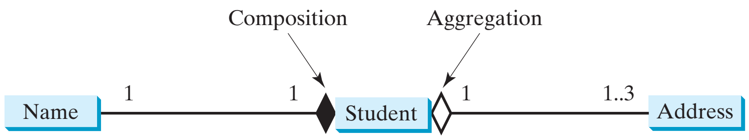


FIGURE 10.6 A student has a name and an address.

multiplicity

Each class involved in a relationship may specify a *multiplicity*. A multiplicity could be a number or an interval that specifies how many of the class's objects are involved in the relationship. The character `*` means an unlimited number of objects, and the interval `m..n` means that the number of objects should be between `m` and `n`, inclusive. In Figure 10.6, each student has only one address, and each address may be shared by up to `3` students. Each student has one name, and a name is unique for each student.

An aggregation relationship is usually represented as a data field in the aggregating class. For example, the relationship in Figure 10.6 can be represented as follows:

```
public class Name {
  ...
}
```

Aggregated class

```
public class Student {
  private Name name;
  private Address address;

  ...
}
```

Aggregating class

```
public class Address {
  ...
}
```

Aggregated class

Aggregation may exist between objects of the same class. For example, a person may have a supervisor. This is illustrated in Figure 10.7.

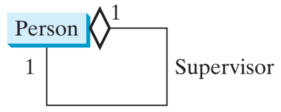
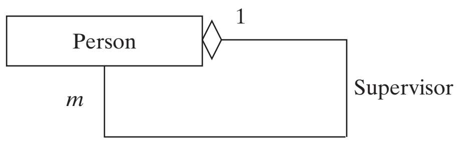


FIGURE 10.7 A person may have a supervisor.

In the relationship "a person has a supervisor," as shown in Figure 10.7, a supervisor can be represented as a data field in the **Person** class, as follows:

```
public class Person {
  // The type for the data is the class itself
  private Person supervisor;

  ...
}
```

If a person can have several supervisors, as shown in Figure 10.8a, you may use an array to store supervisors, as shown in Figure 10.8b.



(a)

```
public class Person {
  ...
  private Person[] supervisors;
}
```

(b)

FIGURE 10.8 A person can have several supervisors.

Note

Since aggregation and composition relationships are represented using classes in the same way, many texts don't differentiate them and call both compositions. We will do the same in this book for simplicity.

aggregation or composition

Check Point
MyProgrammingLab™

10.10 What is an aggregation relationship between two objects?

10.11 What is a composition relationship between two objects?

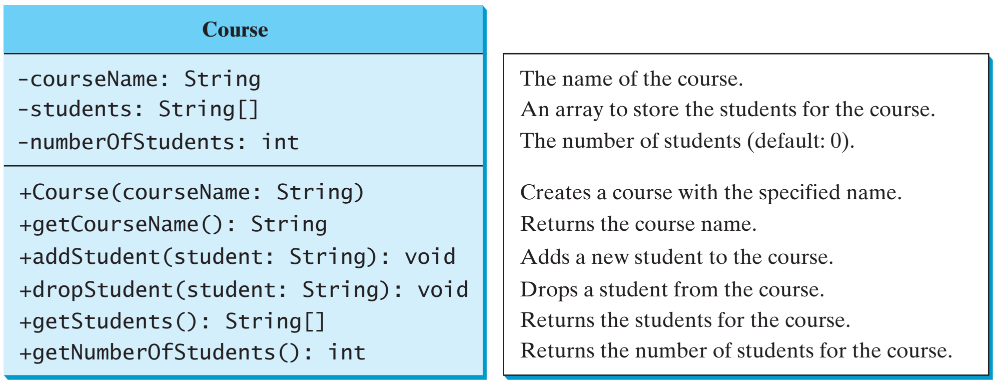
10.8 Case Study: Designing the **Course** Class

Key Point

This section designs a class for modeling courses.

This book's philosophy is *teaching by example and learning by doing*. The book provides a wide variety of examples to demonstrate object-oriented programming. This section and the next two offer additional examples on designing classes.

Suppose you need to process course information. Each course has a name and has students enrolled. You should be able to add/drop a student to/from the course. You can use a class to model the courses, as shown in Figure 10.9.



Course	
-courseName: String	The name of the course.
-students: String[]	An array to store the students for the course.
-numberOfStudents: int	The number of students (default: 0).
+Course(courseName: String)	Creates a course with the specified name.
+getCourseName(): String	Returns the course name.
+addStudent(student: String): void	Adds a new student to the course.
+dropStudent(student: String): void	Drops a student from the course.
+getStudents(): String[]	Returns the students for the course.
+getNumberOfStudents(): int	Returns the number of students for the course.

FIGURE 10.9 The **Course** class models the courses.

A **Course** object can be created using the constructor **Course(String name)** by passing a course name. You can add students to the course using the **addStudent(String student)** method, drop a student from the course using the **dropStudent(String student)** method, and return all the students in the course using the **getStudents()** method. Suppose the class is available; Listing 10.5 gives a test class that creates two courses and adds students to them.

LISTING 10.5 TestCourse.java

create a course

add a student

number of students

return students

```java
public class TestCourse {
  public static void main(String[] args) {
    Course course1 = new Course("Data Structures");
    Course course2 = new Course("Database Systems");

    course1.addStudent("Peter Jones");
    course1.addStudent("Kim Smith");
    course1.addStudent("Anne Kennedy");

    course2.addStudent("Peter Jones");
    course2.addStudent("Steve Smith");

    System.out.println("Number of students in course1: "
      + course1.getNumberOfStudents());
    String[] students = course1.getStudents();
    for (int i = 0; i < course1.getNumberOfStudents(); i++)
      System.out.print(students[i] + ", ");

    System.out.println();
    System.out.print("Number of students in course2: "
        + course2.getNumberOfStudents());
  }
}
```

```
Number of students in course1: 3
Peter Jones, Kim Smith, Anne Kennedy,
Number of students in course2: 2
```

The **Course** class is implemented in Listing 10.6. It uses an array to store the students in the course. For simplicity, assume that the maximum course enrollment is **100**. The array is created using **new String[100]** in line 3. The **addStudent** method (line 10) adds a student to the array. Whenever a new student is added to the course, **numberOfStudents** is increased (line 12). The **getStudents** method returns the array. The **dropStudent** method (line 27) is left as an exercise.

Listing 10.6 Course.java

```
 1  public class Course {
 2    private String courseName;
 3    private String[] students = new String[100];
 4    private int numberOfStudents;
 5
 6    public Course(String courseName) {
 7      this.courseName = courseName;
 8    }
 9
10    public void addStudent(String student) {
11      students[numberOfStudents] = student;
12      numberOfStudents++;
13    }
14
15    public String[] getStudents() {
16      return students;
17    }
18
19    public int getNumberOfStudents() {
20      return numberOfStudents;
21    }
22
23    public String getCourseName() {
24      return courseName;
25    }
26
27    public void dropStudent(String student) {
28      // Left as an exercise in Programming Exercise 10.9
29    }
30  }
```

create students

add a course

return students

number of students

The array size is fixed to be **100** (line 3), so you cannot have more than 100 students in the course. You can improve the class by automatically increasing the array size in Programming Exercise 10.9.

When you create a **Course** object, an array object is created. A **Course** object contains a reference to the array. For simplicity, you can say that the **Course** object contains the array.

The user can create a **Course** object and manipulate it through the public methods **addStudent**, **dropStudent**, **getNumberOfStudents**, and **getStudents**. However, the user doesn't need to know how these methods are implemented. The `Course` class encapsulates the internal implementation. This example uses an array to store students, but you could use a different data structure to store `students`. The program that uses `Course` does not need to change as long as the contract of the public methods remains unchanged.

10.9 Case Study: Designing a Class for Stacks

Key Point

This section designs a class for modeling stacks.

stack

Recall that a *stack* is a data structure that holds data in a last-in, first-out fashion, as shown in Figure 10.10.

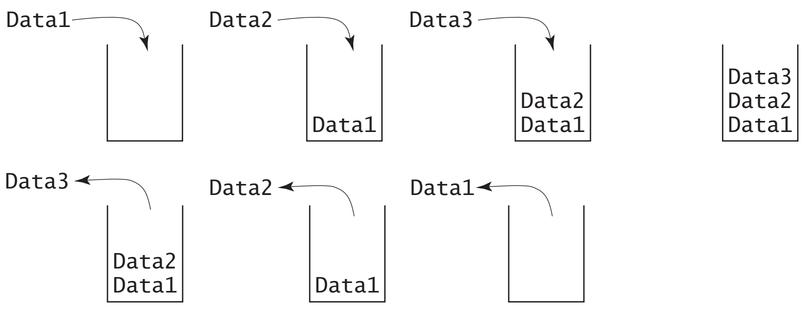
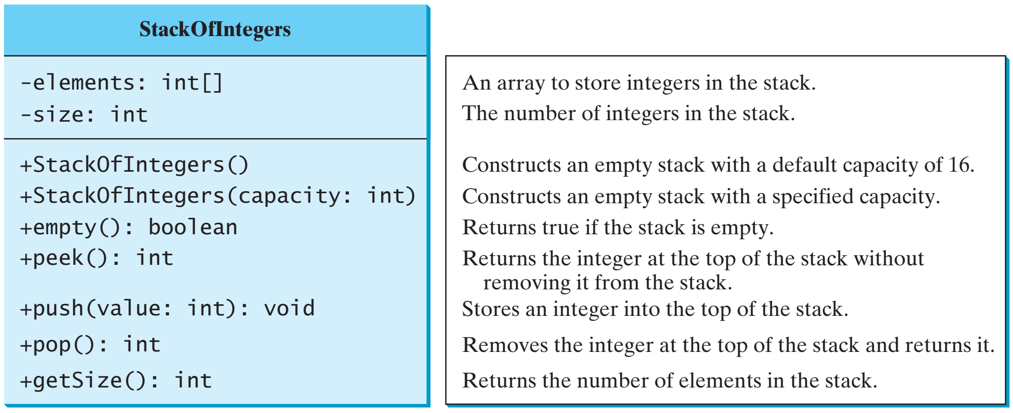


FIGURE 10.10 A stack holds data in a last-in, first-out fashion.

Stacks have many applications. For example, the compiler uses a stack to process method invocations. When a method is invoked, its parameters and local variables are pushed into a stack. When a method calls another method, the new method's parameters and local variables are pushed into the stack. When a method finishes its work and returns to its caller, its associated space is released from the stack.

VideoNote
The `StackOfIntegers` class

You can define a class to model stacks. For simplicity, assume the stack holds the `int` values. So name the stack class `StackOfIntegers`. The UML diagram for the class is shown in Figure 10.11.



StackOfIntegers	
`-elements: int[]`	An array to store integers in the stack.
`-size: int`	The number of integers in the stack.
`+StackOfIntegers()`	Constructs an empty stack with a default capacity of 16.
`+StackOfIntegers(capacity: int)`	Constructs an empty stack with a specified capacity.
`+empty(): boolean`	Returns true if the stack is empty.
`+peek(): int`	Returns the integer at the top of the stack without removing it from the stack.
`+push(value: int): void`	Stores an integer into the top of the stack.
`+pop(): int`	Removes the integer at the top of the stack and returns it.
`+getSize(): int`	Returns the number of elements in the stack.

FIGURE 10.11 The `StackOfIntegers` class encapsulates the stack storage and provides the operations for manipulating the stack.

Suppose that the class is available. The test program in Listing 10.7 uses the class to create a stack (line 3), store ten integers `0`, `1`, `2`, . . . , and `9` (line 6), and displays them in reverse order (line 9).

Listing 10.7 TestStackOfIntegers.java

```java
public class TestStackOfIntegers {
  public static void main(String[] args) {
    StackOfIntegers stack = new StackOfIntegers();

    for (int i = 0; i < 10; i++)
      stack.push(i);

    while (!stack.empty())
      System.out.print(stack.pop() + " ");
  }
}
```

create a stack

push to stack

pop from stack

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

How do you implement the **StackOfIntegers** class? The elements in the stack are stored in an array named **elements**. When you create a stack, the array is also created. The no-arg constructor creates an array with the default capacity of **16**. The variable **size** counts the number of elements in the stack, and **size – 1** is the index of the element at the top of the stack, as shown in Figure 10.12. For an empty stack, **size** is **0**.

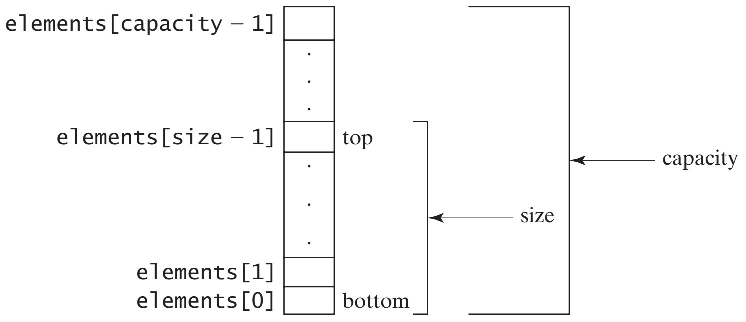


Figure 10.12 The **StackOfIntegers** class encapsulates the stack storage and provides the operations for manipulating the stack.

The **StackOfIntegers** class is implemented in Listing 10.8. The methods **empty()**, **peek()**, **pop()**, and **getSize()** are easy to implement. To implement **push(int value)**, assign **value** to **elements[size]** if **size < capacity** (line 24). If the stack is full (i.e., **size >= capacity**), create a new array of twice the current capacity (line 19), copy the contents of the current array to the new array (line 20), and assign the reference of the new array to the current array in the stack (line 21). Now you can add the new value to the array (line 24).

Listing 10.8 StackOfIntegers.java

```java
1 public class StackOfIntegers {
2   private int[] elements;
3   private int size;
4   public static final int DEFAULT_CAPACITY = 16;
5
6   /** Construct a stack with the default capacity 16 */
7   public StackOfIntegers() {
8     this (DEFAULT_CAPACITY);
```

max capacity 16

```
  }

  /** Construct a stack with the specified maximum capacity */
  public StackOfIntegers(int capacity) {
    elements = new int[capacity];
  }

  /** Push a new integer to the top of the stack */
  public void push(int value) {
    if (size >= elements.length) {
      int[] temp = new int[elements.length * 2];
      System.arraycopy(elements, 0, temp, 0, elements.length);
      elements = temp;
    }

    elements[size++] = value;
  }

  /** Return and remove the top element from the stack */
  public int pop() {
    return elements[--size];
  }

  /** Return the top element from the stack */
  public int peek() {
    return elements[size - 1];
  }

  /** Test whether the stack is empty */
  public boolean empty() {
    return size == 0;
  }

  /** Return the number of elements in the stack */
  public int getSize() {
    return size;
  }
}
```

double the capacity

add to stack

10.10 Case Study: Designing the **GuessDate** Class

Key Point

You can define utility classes that contain static methods and static data.

Listing 3.3, GuessBirthday.java, and Listing 7.6, GuessBirthdayUsingArray.java, presented two programs for guessing birthdays. Both programs use the same data developed with the procedural paradigm. The majority of the code in these two programs is to define the five sets of data. You cannot reuse the code in these two programs, because the code is in the `main` method. To make the code reusable, design a class to encapsulate the data, as defined in Figure 10.13.

Note that `getValue` is defined as a static method because it is not dependent on a specific object of the `GuessDate` class. The `GuessDate` class encapsulates `dates` as a private member. The user of this class doesn't need to know how `dates` is implemented or even that the `dates` field exists in the class. All that the user needs to know is how to use this method to access dates. Suppose this class is available. As shown in Section 3.4, there are five sets of dates. Invoking `getValue(setNo, row, column)` returns the date at the specified row and column in the given set. For example, `getValue(1, 0, 0)` returns `2`.

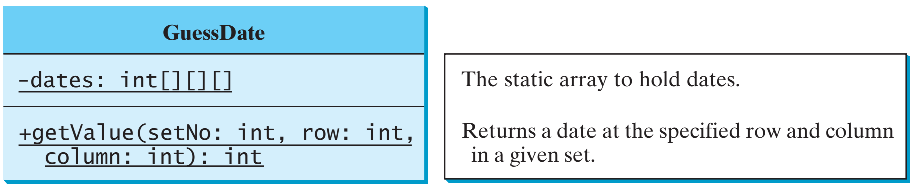

GuessDate
-dates: int[][][]
+getValue(setNo: int, row: int, column: int): int
The static array to hold dates.
Returns a date at the specified row and column in a given set.

FIGURE 10.13 The `GuessDate` class defines data for guessing birthdays.

Assume that the `GuessDate` class is available. Listing 10.9 is a test program that uses this class.

LISTING 10.9 UseGuessDateClass.java

```java
import java.util.Scanner;

public class UseGuessDateClass {
  public static void main(String[] args) {
    int date = 0; // Date to be determined
    int answer;

    // Create a Scanner
    Scanner input = new Scanner(System.in);

    for (int i = 0; i < 5; i++) {
      System.out.println("Is your birthday in Set" + (i + 1) + "?");
      for (int j = 0; j < 4; j++) {
        for (int k = 0; k < 4; k++)
          System.out.print(GuessDate.getValue(i, j, k) + "  ");
        System.out.println();
      }

      System.out.print("\nEnter 0 for No and 1 for Yes: ");
      answer = input.nextInt();

      if (answer == 1)
        date += GuessDate.getValue(i, 0, 0);
    }

    System.out.println("Your birthday is " + date);
  }
}
```

invoke static method

invoke static method

```
Is your birthday in Set1?
1   3   5   7
9   11  13  15
17  19  21  23
25  27  29  31
Enter 0 for No and 1 for Yes: 0

Is your birthday in Set2?
2   3   6   7
10  11  14  15
18  19  22  23
26  27  30  31
Enter 0 for No and 1 for Yes: 1
```

```
Is your birthday in Set3?
4   5   6   7
12  13  14  15
20  21  22  23
28  29  30  31
Enter 0 for No and 1 for Yes: 0

Is your birthday in Set4?
8   9   10  11
12  13  14  15
24  25  26  27
28  29  30  31
Enter 0 for No and 1 for Yes: 1

Is your birthday in Set5?
16  17  18  19
20  21  22  23
24  25  26  27
28  29  30  31
Enter 0 for No and 1 for Yes: 1

Your birthday is 26
```

Since `getValue` is a static method, you don't need to create an object in order to invoke it. `GuessDate.getValue(i, j, k)` (line 15) returns the date at row `j` and column `k` in Set `i`.

The `GuessDate` class can be implemented as in Listing 10.10.

LISTING 10.10 GuessDate.java

static field

private constructor

```java
public class GuessDate {
  private final static int[][][] dates = {
    {{ 1,  3,  5,  7},
     { 9, 11, 13, 15},
     {17, 19, 21, 23},
     {25, 27, 29, 31}},
    {{ 2,  3,  6,  7},
     {10, 11, 14, 15},
     {18, 19, 22, 23},
     {26, 27, 30, 31}},
    {{ 4,  5,  6,  7},
     {12, 13, 14, 15},
     {20, 21, 22, 23},
     {28, 29, 30, 31}},
    {{ 8,  9, 10, 11},
     {12, 13, 14, 15},
     {24, 25, 26, 27},
     {28, 29, 30, 31}},
    {{16, 17, 18, 19},
     {20, 21, 22, 23},
     {24, 25, 26, 27},
     {28, 29, 30, 31}}};

  /** Prevent the user from creating objects from GuessDate */
  private GuessDate() {
  }

  /** Return a date at the specified row and column in a given set */
29   public static int getValue(int setNo, int i, int j) {
30     return dates[setNo][i][j];
31   }
32 }
```

static method

This class uses a three-dimensional array to store dates (lines 2–22). You could use a different data structure (i.e., five two-dimensional arrays for representing five sets of numbers). The implementation of the `getValue` method would change, but the program that uses `GuessDate` wouldn't need to change as long as the contract of the public method `getValue` remains unchanged. This shows the benefit of data encapsulation.

benefit of data encapsulation

private constructor

The class defines a private no-arg constructor (line 25) to prevent the user from creating objects for this class. Since all methods are static in this class, there is no need to create objects from this class.

Check Point

MyProgrammingLab™

10.12 Why is the no-arg constructor in the `Math` class defined private?

10.11 Class Design Guidelines

Key Point

Class design guidelines are helpful for designing sound classes.

You have learned how to design classes from the preceding two examples and from many other examples in the preceding chapters. This section summarizes some of the guidelines.

10.11.1 Cohesion

coherent purpose

A class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose. You can use a class for students, for example, but you should not combine students and staff in the same class, because students and staff are different entities.

separating responsibilities

A single entity with many responsibilities can be broken into several classes to separate the responsibilities. The classes `String`, `StringBuilder`, and `StringBuffer` all deal with strings, for example, but have different responsibilities. The `String` class deals with immutable strings, the `StringBuilder` class is for creating mutable strings, and the `StringBuffer` class is similar to `StringBuilder` except that `StringBuffer` contains synchronized methods for updating strings.

10.11.2 Consistency

naming conventions

Follow standard Java programming style and naming conventions. Choose informative names for classes, data fields, and methods. A popular style is to place the data declaration before the constructor and place constructors before methods.

naming consistency

Make the names consistent. It is not a good practice to choose different names for similar operations. For example, the `length()` method returns the size of a `String`, a `StringBuilder`, and a `StringBuffer`. It would be inconsistent if different names were used for this method in these classes.

no-arg constructor

In general, you should consistently provide a public no-arg constructor for constructing a default instance. If a class does not support a no-arg constructor, document the reason. If no constructors are defined explicitly, a public default no-arg constructor with an empty body is assumed.

If you want to prevent users from creating an object for a class, you can declare a private constructor in the class, as is the case for the `Math` class and the `GuessDate` class.

10.11.3 Encapsulation

encapsulating data fields

A class should use the `private` modifier to hide its data from direct access by clients. This makes the class easy to maintain.

Provide a **get** method only if you want the field to be readable, and provide a **set** method only if you want the field to be updateable. For example, the **Course** class provides a **get** method for **courseName**, but no **set** method, because the user is not allowed to change the course name once it is created.

10.11.4 Clarity

Cohesion, consistency, and encapsulation are good guidelines for achieving design clarity. Additionally, a class should have a clear contract that is easy to explain and easy to understand.

easy to explain

Users can incorporate classes in many different combinations, orders, and environments. Therefore, you should design a class that imposes no restrictions on how or when the user can use it, design the properties in a way that lets the user set them in any order and with any combination of values, and design methods that function independently of their order of occurrence. For example, the **Loan** class contains the properties **loanAmount**, **numberOfYears**, and **annualInterestRate**. The values of these properties can be set in any order.

independent methods

intuitive meaning

Methods should be defined intuitively without causing confusion. For example, the **substring(int beginIndex, int endIndex)** method in the **String** class is somewhat confusing. The method returns a substring from **beginIndex** to **endIndex – 1**, rather than to **endIndex**. It would be more intuitive to return a substring from **beginIndex** to **endIndex**.

independent properties

You should not declare a data field that can be derived from other data fields. For example, the following **Person** class has two data fields: **birthDate** and **age**. Since **age** can be derived from **birthDate**, **age** should not be declared as a data field.

BAD CODE

```
public class Person {
  private java.util.Date birthDate;
  private int age;

  ...
}
```

10.11.5 Completeness

Classes are designed for use by many different customers. In order to be useful in a wide range of applications, a class should provide a variety of ways for customization through properties and methods. For example, the **String** class contains more than 40 methods that are useful for a variety of applications.

10.11.6 Instance vs. Static

A variable or method that is dependent on a specific instance of the class must be an instance variable or method. A variable that is shared by all the instances of a class should be declared static. For example, the variable **numberOfObjects** in **CircleWithPrivateDataFields** in Listing 8.9 is shared by all the objects of the **CircleWithPrivateDataFields** class and therefore is declared static. A method that is not dependent on a specific instance should be defined as a static method. For instance, the **getNumberOfObjects** method in **CircleWithPrivateDataFields** is not tied to any specific instance and therefore is defined as a static method.

Always reference static variables and methods from a class name (rather than a reference variable) to improve readability and avoid errors.

Do not pass a parameter from a constructor to initialize a static data field. It is better to use a **set** method to change the static data field. Thus, the following class in (a) is better replaced by (b).

```java
public class SomeThing {
  private int t1;
  private static int t2;

  public SomeThing(int t1, int t2) {
    ...
  }
}
```

(a)

```java
public class SomeThing {
  private int t1;
  private static int t2;

  public SomeThing(int t1) {
    ...
  }

  public static void setT2(int t2) {
    SomeThing.t2 = t2;
  }
}
```

(b)

Instance and static are integral parts of object-oriented programming. A data field or method is either instance or static. Do not mistakenly overlook static data fields or methods. It is a common design error to define an instance method that should have been static. For example, the **factorial(int n)** method for computing the factorial of **n** should be defined static, because it is independent of any specific instance.

common design error

A constructor is always instance, because it is used to create a specific instance. A static variable or method can be invoked from an instance method, but an instance variable or method cannot be invoked from a static method.

10.13 Describe class design guidelines.

Check Point
MyProgrammingLab™

10.12 Processing Primitive Data Type Values as Objects

Key Point

A primitive type value is not an object, but it can be wrapped in an object using a wrapper class in the Java API.

Owing to performance considerations, primitive data type values are not objects in Java. Because of the overhead of processing objects, the language's performance would be adversely affected if primitive data type values were treated as objects. However, many Java methods require the use of objects as arguments. Java offers a convenient way to incorporate, or wrap, a primitive data type into an object (e.g., wrapping **int** into the **Integer** class, and wrapping **double** into the **Double** class). Recall that a **char** value can be wrapped into a **Character** object in Section 9.5. By using a wrapper class, you can process primitive data type values as objects. Java provides **Boolean**, **Character**, **Double**, **Float**, **Byte**, **Short**, **Integer**, and **Long** wrapper classes in the **java.lang** package for primitive data types. The **Boolean** class wraps a Boolean value **true** or **false**. This section uses **Integer** and **Double** as examples to introduce the numeric wrapper classes.

why wrapper class?

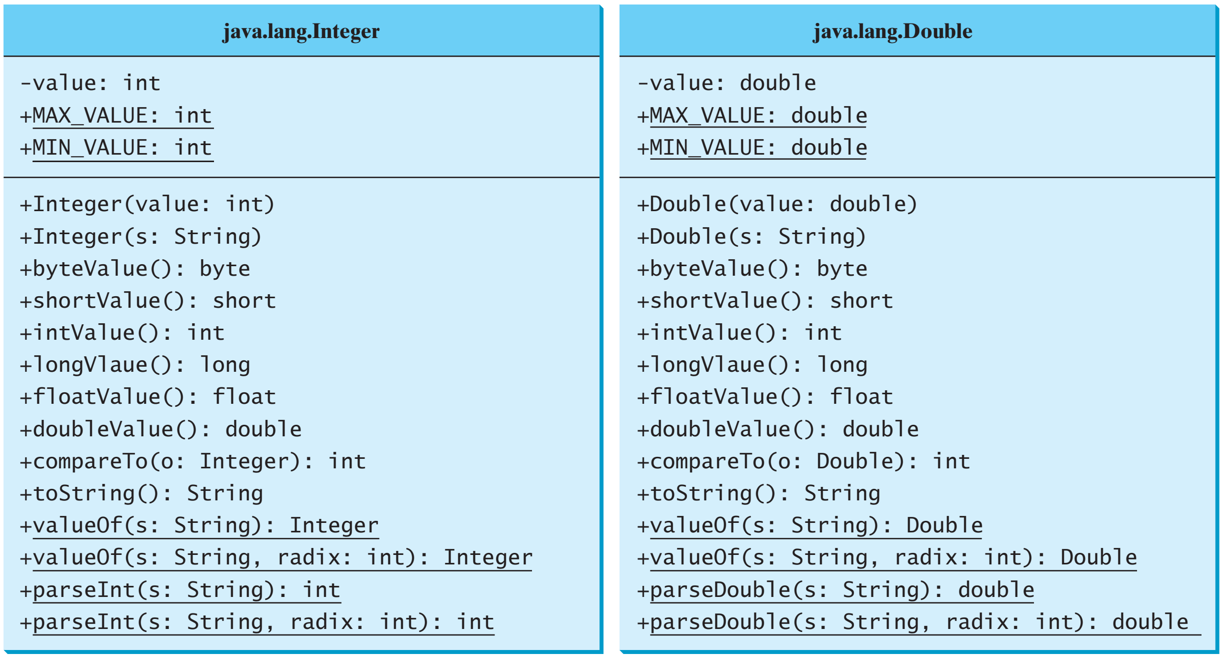
Note
Most wrapper class names for a primitive type are the same as the primitive data type name with the first letter capitalized. The exceptions are **Integer** and **Character**.

naming convention

Numeric wrapper classes are very similar to each other. Each contains the methods **doubleValue()**, **floatValue()**, **intValue()**, **longValue()**, **shortValue()**, and **byteValue()**. These methods "convert" objects into primitive type values. The key features of **Integer** and **Double** are shown in Figure 10.14.

You can construct a wrapper object either from a primitive data type value or from a string representing the numeric value—for example, **new Double(5.0)**, **new Double("5.0")**, **new Integer(5)**, and **new Integer("5")**.

constructors



```
java.lang.Integer

-value: int
+MAX_VALUE: int
+MIN_VALUE: int

+Integer(value: int)
+Integer(s: String)
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longVlaue(): long
+floatValue(): float
+doubleValue(): double
+compareTo(o: Integer): int
+toString(): String
+valueOf(s: String): Integer
+valueOf(s: String, radix: int): Integer
+parseInt(s: String): int
+parseInt(s: String, radix: int): int
```

```
java.lang.Double

-value: double
+MAX_VALUE: double
+MIN_VALUE: double

+Double(value: double)
+Double(s: String)
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longVlaue(): long
+floatValue(): float
+doubleValue(): double
+compareTo(o: Double): int
+toString(): String
+valueOf(s: String): Double
+valueOf(s: String, radix: int): Double
+parseDouble(s: String): double
+parseDouble(s: String, radix: int): double
```

FIGURE 10.14 The wrapper classes provide constructors, constants, and conversion methods for manipulating various data types.

no no-arg constructor
immutable

The wrapper classes do not have no-arg constructors. The instances of all wrapper classes are immutable; this means that, once the objects are created, their internal values cannot be changed.

constants

Each numeric wrapper class has the constants `MAX_VALUE` and `MIN_VALUE`. `MAX_VALUE` represents the maximum value of the corresponding primitive data type. For `Byte`, `Short`, `Integer`, and `Long`, `MIN_VALUE` represents the minimum `byte`, `short`, `int`, and `long` values. For `Float` and `Double`, `MIN_VALUE` represents the minimum *positive* `float` and `double` values. The following statements display the maximum integer (2,147,483,647), the minimum positive float (1.4E–45), and the maximum double floating-point number (1.79769313486231570e+308d).

```
System.out.println("The maximum integer is " + Integer.MAX_VALUE);
System.out.println("The minimum positive float is " +
  Float.MIN_VALUE);
System.out.println(
  "The maximum double-precision floating-point number is " +
  Double.MAX_VALUE);
```

conversion methods

Each numeric wrapper class contains the methods `doubleValue()`, `floatValue()`, `intValue()`, `longValue()`, and `shortValue()` for returning a `double`, `float`, `int`, `long`, or `short` value for the wrapper object. For example,

`new Double("12.4").intValue()` returns `12`;
`new Integer("12").doubleValue()` returns `12.0`;

compareTo method

Recall that the `String` class contains the `compareTo` method for comparing two strings. The numeric wrapper classes contain the `compareTo` method for comparing two numbers and returns `1`, `0`, or `-1`, if this number is greater than, equal to, or less than the other number. For example,

`new Double("12.4").compareTo(new Double("12.3"))` returns `1`;
`new Double("12.3").compareTo(new Double("12.3"))` returns `0`;
`new Double("12.3").compareTo(new Double("12.51"))` returns `-1`;

The numeric wrapper classes have a useful static method, `valueOf (String s)`. This method creates a new object initialized to the value represented by the specified string. For example,

static `valueOf` methods

```
Double doubleObject = Double.valueOf("12.4");
Integer integerObject = Integer.valueOf("12");
```

You have used the `parseInt` method in the `Integer` class to parse a numeric string into an `int` value and the `parseDouble` method in the `Double` class to parse a numeric string into a `double` value. Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value based on `10` (decimal) or any specified radix (e.g., `2` for binary, `8` for octal, and `16` for hexadecimal). The following examples show how to use these methods.

static parsing methods

```
// These two methods are in the Byte class
public static byte parseByte(String s)
public static byte parseByte(String s, int radix)

// These two methods are in the Short class
public static short parseShort(String s)
public static short parseShort(String s, int radix)

// These two methods are in the Integer class
public static int parseInt(String s)
public static int parseInt(String s, int radix)

// These two methods are in the Long class
public static long parseLong(String s)
public static long parseLong(String s, int radix)

// These two methods are in the Float class
public static float parseFloat(String s)
public static float parseFloat(String s, int radix)

// These two methods are in the Double class
public static double parseDouble(String s)
public static double parseDouble(String s, int radix)
```

For example,

`Integer.parseInt("11", 2)` returns `3`;
`Integer.parseInt("12", 8)` returns `10`;
`Integer.parseInt("13", 10)` returns `13`;
`Integer.parseInt("1A", 16)` returns `26`;

`Integer.parseInt("12", 2)` would raise a runtime exception because `12` is not a binary number.

Note that you can convert a decimal number into a hex number using the `format` method. For example,

converting decimal to hex

`String.format("%x", 26)` returns `1A`;

Check Point
MyProgrammingLab™

10.14 Describe primitive-type wrapper classes.

10.15 Can each of the following statements be compiled?

a. `Integer i = new Integer("23");`

b. `Integer i = new Integer(23);`

c. `Integer i = Integer.valueOf("23");`

d. `Integer i = Integer.parseInt("23", 8);`

e. `Double d = new Double();`

f. `Double d = Double.valueOf("23.45");`

g. `int i = (Integer.valueOf("23")).intValue();`

h. `double d = (Double.valueOf("23.4")).doubleValue();`

i. `int i = (Double.valueOf("23.4")).intValue();`

j. `String s = (Double.valueOf("23.4")).toString();`

10.16 How do you convert an integer into a string? How do you convert a numeric string into an integer? How do you convert a double number into a string? How do you convert a numeric string into a double value?

10.17 Show the output of the following code.

```
public class Test {
  public static void main(String[] args) {
    Integer x = new Integer(3);
    System.out.println(x.intValue());
    System.out.println(x.compareTo(new Integer(4)));
  }
}
```

10.18 What is the output of the following code?

```
public class Test {
  public static void main(String[] args) {
    System.out.println(Integer.parseInt("10"));
    System.out.println(Integer.parseInt("10", 10));
    System.out.println(Integer.parseInt("10", 16));
    System.out.println(Integer.parseInt("11"));
    System.out.println(Integer.parseInt("11", 10));
    System.out.println(Integer.parseInt("11", 16));
  }
}
```

10.13 Automatic Conversion between Primitive Types and Wrapper Class Types

Key Point

A primitive type value can be automatically converted to an object using a wrapper class, and vice versa, depending on the context.

boxing
unboxing

Converting a primitive value to a wrapper object is called *boxing*. The reverse conversion is called *unboxing*. Java allows primitive types and wrapper classes to be converted automatically. The compiler will automatically box a primitive value that appears in a context requiring an object, and will unbox an object that appears in a context requiring a primitive value. This is called *autoboxing* and *autounboxing*.

autoboxing
autounboxing

For instance, the following statement in (a) can be simplified as in (b) due to autoboxing.

```
Integer intObject = new Integer(2);
```

(a)

Equivalent

```
Integer intObject = 2;
```

(b)

autoboxing

Consider the following example:

```
1  Integer[] intArray = {1, 2, 3};
2  System.out.println(intArray[0] + intArray[1] + intArray[2]);
```

In line 1, the primitive values `1`, `2`, and `3` are automatically boxed into objects `new Integer(1)`, `new Integer(2)`, and `new Integer(3)`. In line 2, the objects `intArray[0]`, `intArray[1]`, and `intArray[2]` are automatically converted into `int` values that are added together.

Check Point
MyProgrammingLab™

10.19 What are autoboxing and autounboxing? Are the following statements correct?

a. `Integer x = 3 + new Integer(5);`

b. `Integer x = 3;`

c. `Double x = 3;`

d. `Double x = 3.0;`

e. `int x = new Integer(3);`

f. `int x = new Integer(3) + new Integer(4);`

10.20 Show the output of the following code?

```
public class Test {
  public static void main(String[] args) {
    Double x = new Double(3.5);
    System.out.println(x.intValue());
    System.out.println(x.compareTo(4.5));
  }
}
```

10.14 The **BigInteger** and **BigDecimal** Classes

The `BigInteger` and `BigDecimal` classes can be used to represent integers or decimal numbers of any size and precision.

Key Point

VideoNote
Process large numbers

immutable

If you need to compute with very large integers or high-precision floating-point values, you can use the `BigInteger` and `BigDecimal` classes in the `java.math` package. Both are *immutable*. The largest integer of the `long` type is `Long.MAX_VALUE` (i.e., `9223372036854775807`). An instance of `BigInteger` can represent an integer of any size. You can use `new BigInteger(String)` and `new BigDecimal(String)` to create an instance of `BigInteger` and `BigDecimal`, use the `add`, `subtract`, `multiple`, `divide`, and `remainder` methods to perform arithmetic operations, and use the `compareTo` method to compare two big numbers. For example, the following code creates two `BigInteger` objects and multiplies them.

```
BigInteger a = new BigInteger("9223372036854775807");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);
```

The output is `18446744073709551614`.

There is no limit to the precision of a `BigDecimal` object. The `divide` method may throw an `ArithmeticException` if the result cannot be terminated. However, you can use the overloaded `divide(BigDecimal d, int scale, int roundingMode)` method to specify a scale and a rounding mode to avoid this exception, where `scale` is the maximum number of digits after the decimal point. For example, the following code creates two `BigDecimal` objects and performs division with scale `20` and rounding mode `BigDecimal.ROUND_UP`.

```
BigDecimal a = new BigDecimal(1.0);
BigDecimal b = new BigDecimal(3);
BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);
System.out.println(c);
```

The output is `0.33333333333333333334`.

Note that the factorial of an integer can be very large. Listing 10.11 gives a method that can return the factorial of any integer.

LISTING 10.11 LargeFactorial.java

```
 1  import java.math.*;
 2
 3  public class LargeFactorial {
 4    public static void main(String[] args) {
 5      System.out.println("50! is \n" + factorial(50));
 6    }
 7
 8    public static BigInteger factorial(long n) {
 9      BigInteger result = BigInteger.ONE;
10      for (int i = 1; i <= n; i++)
11        result = result.multiply(new BigInteger(i + ""));
12
13      return result;
14    }
15  }
```

constant

multiply

```
50! is
30414093201713378043612608166064768844377641568960512000000000000
```

`BigInteger.ONE` (line 9) is a constant defined in the `BigInteger` class. `BigInteger.ONE` is the same as `new BigInteger("1")`.

A new result is obtained by invoking the `multiply` method (line 11).

Check Point
MyProgrammingLab™

10.21 What is the output of the following code?

```
public class Test {
  public static void main(String[] args) {
    java.math.BigInteger x = new java.math.BigInteger("3");
    java.math.BigInteger y = new java.math.BigInteger("7");
    x.add(y);
    System.out.println(x);
  }
}
```

Key Terms

abstract data type (ADT)	375	has-a relationship	382
aggregation	382	immutable class	370
boxing	396	immutable object	370
class abstraction	375	multiplicity	382
class encapsulation	375	stack	386
class's contract	375	`this` keyword	373
class's variable	371	unboxing	396
composition	382		

Chapter Summary

1. Once it is created, an *immutable object* cannot be modified. To prevent users from modifying an object, you can define *immutable classes*.

2. The scope of instance and static variables is the entire class, regardless of where the variables are declared. Instance and static variables can be declared anywhere in the class. For consistency, they are declared at the beginning of the class in this book.

3. The keyword `this` can be used to refer to the calling object. It can also be used inside a constructor to invoke another constructor of the same class.

4. The procedural paradigm focuses on designing methods. The object-oriented paradigm couples data and methods together into objects. Software design using the object-oriented paradigm focuses on objects and operations on objects. The object-oriented approach combines the power of the procedural paradigm with an added dimension that integrates data with operations into objects.

5. Many Java methods require the use of objects as arguments. Java offers a convenient way to incorporate, or wrap, a primitive data type into an object (e.g., wrapping `int` into the `Integer` class, and wrapping `double` into the `Double` class).

6. Java can automatically convert a primitive type value to its corresponding wrapper object in the context and vice versa.

7. The `BigInteger` class is useful for computing and processing integers of any size. The `BigDecimal` class can be used to compute and process floating-point numbers with any arbitrary precision.

Test Questions

Do the test questions for this chapter online at www.cs.armstrong.edu/liang/intro9e/test.html.

Programming Exercises

Sections 10.2–10.6

***10.1** (*The* `Time` *class*) Design a class named `Time`. The class contains:

- The data fields `hour`, `minute`, and `second` that represent a time.
- A no-arg constructor that creates a `Time` object for the current time. (The values of the data fields will represent the current time.)

- A constructor that constructs a `Time` object with a specified elapsed time since midnight, January 1, 1970, in milliseconds. (The values of the data fields will represent this time.)
- A constructor that constructs a `Time` object with the specified hour, minute, and second.
- Three `get` methods for the data fields `hour`, `minute`, and `second`, respectively.
- A method named `setTime(long elapseTime)` that sets a new time for the object using the elapsed time. For example, if the elapsed time is `555550000` milliseconds, the hour is `10`, the minute is `19`, and the second is `10`.

Draw the UML diagram for the class and then implement the class. Write a test program that creates two `Time` objects (using `new Time()` and `new Time(555550000)`) and displays their hour, minute, and second in the format hour:minute:second.

(*Hint*: The first two constructors will extract the hour, minute, and second from the elapsed time. For the no-arg constructor, the current time can be obtained using `System.currentTimeMillis()`, as shown in Listing 2.6, ShowCurrentTime.java.)

10.2 (*The* `BMI` *class*) Add the following new constructor in the `BMI` class:

```
/** Construct a BMI with the specified name, age, weight,
 * feet, and inches
 */
public BMI(String name, int age, double weight, double feet,
  double inches)
```

10.3 (*The* `MyInteger` *class*) Design a class named `MyInteger`. The class contains:

- An `int` data field named `value` that stores the `int` value represented by this object.
- A constructor that creates a `MyInteger` object for the specified `int` value.
- A `get` method that returns the `int` value.
- The methods `isEven()`, `isOdd()`, and `isPrime()` that return `true` if the value in this object is even, odd, or prime, respectively.
- The static methods `isEven(int)`, `isOdd(int)`, and `isPrime(int)` that return `true` if the specified value is even, odd, or prime, respectively.
- The static methods `isEven(MyInteger)`, `isOdd(MyInteger)`, and `isPrime(MyInteger)` that return `true` if the specified value is even, odd, or prime, respectively.
- The methods `equals(int)` and `equals(MyInteger)` that return `true` if the value in this object is equal to the specified value.
- A static method `parseInt(char[])` that converts an array of numeric characters to an `int` value.
- A static method `parseInt(String)` that converts a string into an `int` value.

Draw the UML diagram for the class and then implement the class. Write a client program that tests all methods in the class.

10.4 (*The* `MyPoint` *class*) Design a class named `MyPoint` to represent a point with `x`- and `y`-coordinates. The class contains:

VideoNote
The `MyPoint` class

- The data fields `x` and `y` that represent the coordinates with `get` methods.
- A no-arg constructor that creates a point (`0`, `0`).
- A constructor that constructs a point with specified coordinates.
- Two `get` methods for the data fields `x` and `y`, respectively.

- A method named `distance` that returns the distance from this point to another point of the `MyPoint` type.
- A method named `distance` that returns the distance from this point to another point with specified `x`- and `y`-coordinates.

Draw the UML diagram for the class and then implement the class. Write a test program that creates the two points (`0`, `0`) and (`10`, `30.5`) and displays the distance between them.

Sections 10.7–10.11

***10.5** (*Displaying the prime factors*) Write a program that prompts the user to enter a positive integer and displays all its smallest factors in decreasing order. For example, if the integer is `120`, the smallest factors are displayed as `5`, `3`, `2`, `2`, `2`. Use the `StackOfIntegers` class to store the factors (e.g., `2`, `2`, `2`, `3`, `5`) and retrieve and display them in reverse order.

***10.6** (*Displaying the prime numbers*) Write a program that displays all the prime numbers less than `120` in decreasing order. Use the `StackOfIntegers` class to store the prime numbers (e.g., `2`, `3`, `5`, . . .) and retrieve and display them in reverse order.

****10.7** (*Game: ATM machine*) Use the `Account` class created in Programming Exercise 8.7 to simulate an ATM machine. Create ten accounts in an array with id `0`, `1`, . . . , `9`, and initial balance $100. The system prompts the user to enter an id. If the id is entered incorrectly, ask the user to enter a correct id. Once an id is accepted, the main menu is displayed as shown in the sample run. You can enter a choice `1` for viewing the current balance, `2` for withdrawing money, `3` for depositing money, and `4` for exiting the main menu. Once you exit, the system will prompt for an id again. Thus, once the system starts, it will not stop.

```
Enter an id: 4 ↵Enter

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 1 ↵Enter
The balance is 100.0

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 2 ↵Enter
Enter an amount to withdraw: 3 ↵Enter

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 1 ↵Enter
The balance is 97.0
```

```
Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 3 ↵Enter
Enter an amount to deposit: 10 ↵Enter

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 1 ↵Enter
The balance is 107.0

Main menu
1: check balance
2: withdraw
3: deposit
4: exit
Enter a choice: 4 ↵Enter

Enter an id:
```

*****10.8** (*Financial: the* `Tax` *class*) Programming Exercise 7.12 writes a program for computing taxes using arrays. Design a class named `Tax` to contain the following instance data fields:

- `int filingStatus`: One of the four tax-filing statuses: `0`—single filer, `1`—married filing jointly or qualifying widow(er), `2`—married filing separately, and `3`—head of household. Use the public static constants `SINGLE_FILER` (`0`), `MARRIED_JOINTLY_OR_QUALIFYING_WIDOW(ER)` (`1`), `MARRIED_SEPARATELY` (`2`), `HEAD_OF_HOUSEHOLD` (`3`) to represent the statuses.
- `int[][] brackets`: Stores the tax brackets for each filing status.
- `double[] rates`: Stores the tax rates for each bracket.
- `double taxableIncome`: Stores the taxable income.

Provide the `get` and `set` methods for each data field and the `getTax()` method that returns the tax. Also provide a no-arg constructor and the constructor `Tax(filingStatus, brackets, rates, taxableIncome)`.

Draw the UML diagram for the class and then implement the class. Write a test program that uses the `Tax` class to print the 2001 and 2009 tax tables for taxable income from $50,000 to $60,000 with intervals of $1,000 for all four statuses. The tax rates for the year 2009 were given in Table 3.2. The tax rates for 2001 are shown in Table 10.1.

****10.9** (*The* `Course` *class*) Revise the `Course` class as follows:

- The array size is fixed in Listing 10.6. Improve it to automatically increase the array size by creating a new larger array and copying the contents of the current array to it.
- Implement the `dropStudent` method.
- Add a new method named `clear()` that removes all students from the course.

Write a test program that creates a course, adds three students, removes one, and displays the students in the course.

TABLE 10.1 2001 United States Federal Personal Tax Rates

Tax rate	*Single filers*	*Married filing jointly or qualifying widow(er)*	*Married filing separately*	*Head of household*
15%	Up to $27,050	Up to $45,200	Up to $22,600	Up to $36,250
27.5%	$27,051–$65,550	$45,201–$109,250	$22,601–$54,625	$36,251–$93,650
30.5%	$65,551–$136,750	$109,251–$166,500	$54,626–$83,250	$93,651–$151,650
35.5%	$136,751–$297,350	$166,501–$297,350	$83,251–$148,675	$151,651–$297,350
39.1%	$297,351 or more	$297,351 or more	$ 148,676 or more	$297,351 or more

***10.10** (*Game: The* **GuessDate** *class*) Modify the **GuessDate** class in Listing 10.10. Instead of representing dates in a three-dimensional array, use five two-dimensional arrays to represent the five sets of numbers. Thus, you need to declare:

```
private static int[][] set1 = {{1,  3,  5,  7}, ... };
private static int[][] set2 = {{2,  3,  6,  7}, ... };
private static int[][] set3 = {{4,  5,  6,  7}, ... };
private static int[][] set4 = {{8,  9, 10, 11}, ... };
private static int[][] set5 = {{16, 17, 18, 19}, ... };
```

***10.11** (*Geometry: The* **Circle2D** *class*) Define the **Circle2D** class that contains:

- Two **double** data fields named **x** and **y** that specify the center of the circle with **get** methods.
- A data field **radius** with a **get** method.
- A no-arg constructor that creates a default circle with (**0**, **0**) for (**x**, **y**) and **1** for **radius**.
- A constructor that creates a circle with the specified **x**, **y**, and **radius**.
- A method **getArea()** that returns the area of the circle.
- A method **getPerimeter()** that returns the perimeter of the circle.
- A method **contains(double x, double y)** that returns **true** if the specified point (**x**, **y**) is inside this circle (see Figure 10.15a).
- A method **contains(Circle2D circle)** that returns **true** if the specified circle is inside this circle (see Figure 10.15b).
- A method **overlaps(Circle2D circle)** that returns **true** if the specified circle overlaps with this circle (see Figure 10.15c).

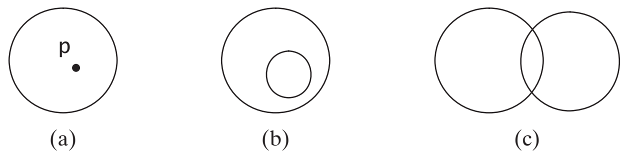


FIGURE 10.15 (a) A point is inside the circle. (b) A circle is inside another circle. (c) A circle overlaps another circle.

Draw the UML diagram for the class and then implement the class. Write a test program that creates a **Circle2D** object **c1** (**new Circle2D(2, 2, 5.5)**), displays its area and perimeter, and displays the result of **c1.contains(3, 3)**, **c1.contains(new Circle2D(4, 5, 10.5))**, and **c1.overlaps(new Circle2D(3, 5, 2.3))**.

*****10.12** (*Geometry: The* `Triangle2D` *class*) Define the `Triangle2D` class that contains:

- Three points named `p1`, `p2`, and `p3` of the type `MyPoint` with `get` and `set` methods. `MyPoint` is defined in Exercise 10.4.
- A no-arg constructor that creates a default triangle with the points (`0`, `0`), (`1`, `1`), and (`2`, `5`).
- A constructor that creates a triangle with the specified points.
- A method `getArea()` that returns the area of the triangle.
- A method `getPerimeter()` that returns the perimeter of the triangle.
- A method `contains(MyPoint p)` that returns `true` if the specified point `p` is inside this triangle (see Figure 10.16a).
- A method `contains(Triangle2D t)` that returns `true` if the specified triangle is inside this triangle (see Figure 10.16b).
- A method `overlaps(Triangle2D t)` that returns `true` if the specified triangle overlaps with this triangle (see Figure 10.16c).

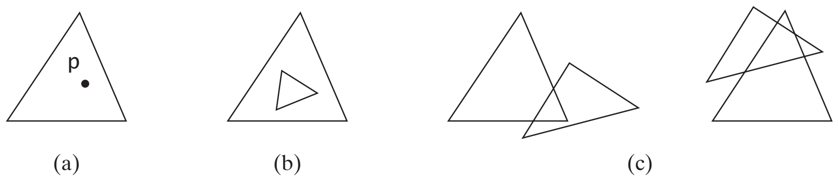


FIGURE 10.16 (a) A point is inside the triangle. (b) A triangle is inside another triangle. (c) A triangle overlaps another triangle.

Draw the UML diagram for the class and then implement the class. Write a test program that creates a `Triangle2D` objects `t1` using the constructor `new Triangle2D(new MyPoint(2.5, 2), new MyPoint(4.2, 3), new MyPoint(5, 3.5))`, displays its area and perimeter, and displays the result of `t1.contains(3, 3)`, `r1.contains(new Triangle2D(new MyPoint(2.9, 2), new MyPoint(4, 1), MyPoint(1, 3.4)))`, and `t1.overlaps(new Triangle2D(new MyPoint(2, 5.5), new MyPoint(4, -3), MyPoint(2, 6.5)))`.

(*Hint*: For the formula to compute the area of a triangle, see Programming Exercise 2.15. Use the `java.awt.geo.Line2D` class in the Java API to implement the `contains` and `overlaps` methods. The `Line2D` class contains the methods for checking whether two line segments intersect and whether a line contains a point, and so on. Please see the Java API for more information on `Line2D`. To detect whether a point is inside a triangle, draw three dashed lines, as shown in Figure 10.17. If the point is inside a triangle, each dashed line should intersect a side only once. If a dashed line intersects a side twice, then the point must be outside the triangle.)

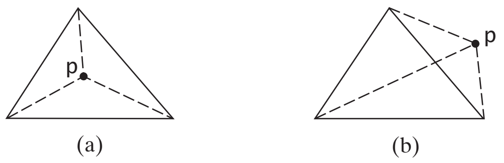


FIGURE 10.17 (a) A point is inside the triangle. (b) A point is outside the triangle.

***10.13** (*Geometry: the* `MyRectangle2D` *class*) Define the `MyRectangle2D` class that contains:

- Two `double` data fields named `x` and `y` that specify the center of the rectangle with `get` and `set` methods. (Assume that the rectangle sides are parallel to `x`- or `y`- axes.)
- The data fields `width` and `height` with `get` and `set` methods.
- A no-arg constructor that creates a default rectangle with (`0`, `0`) for (`x`, `y`) and `1` for both `width` and `height`.
- A constructor that creates a rectangle with the specified `x`, `y`, `width`, and `height`.
- A method `getArea()` that returns the area of the rectangle.
- A method `getPerimeter()` that returns the perimeter of the rectangle.
- A method `contains(double x, double y)` that returns `true` if the specified point (`x`, `y`) is inside this rectangle (see Figure 10.18a).
- A method `contains(MyRectangle2D r)` that returns `true` if the specified rectangle is inside this rectangle (see Figure 10.18b).
- A method `overlaps(MyRectangle2D r)` that returns `true` if the specified rectangle overlaps with this rectangle (see Figure 10.18c).

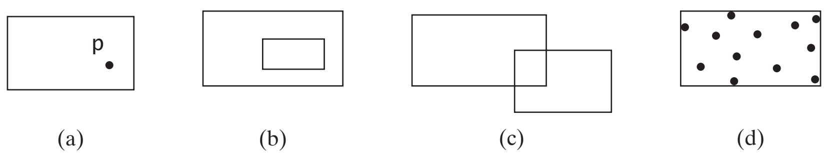


FIGURE 10.18 A point is inside the rectangle. (b) A rectangle is inside another rectangle. (c) A rectangle overlaps another rectangle. (d) Points are enclosed inside a rectangle.

Draw the UML diagram for the class and then implement the class. Write a test program that creates a `MyRectangle2D` object `r1` (`new MyRectangle2D(2, 2, 5.5, 4.9)`), displays its area and perimeter, and displays the result of `r1.contains(3, 3)`, `r1.contains(new MyRectangle2D(4, 5, 10.5, 3.2))`, and `r1.overlaps(new MyRectangle2D(3, 5, 2.3, 5.4))`.

***10.14** (*The* `MyDate` *class*) Design a class named `MyDate`. The class contains:

- The data fields `year`, `month`, and `day` that represent a date. `month` is 0-based, i.e., `0` is for January.
- A no-arg constructor that creates a `MyDate` object for the current date.
- A constructor that constructs a `MyDate` object with a specified elapsed time since midnight, January 1, 1970, in milliseconds.
- A constructor that constructs a `MyDate` object with the specified year, month, and day.
- Three `get` methods for the data fields `year`, `month`, and `day`, respectively.
- A method named `setDate(long elapsedTime)` that sets a new date for the object using the elapsed time.

Draw the UML diagram for the class and then implement the class. Write a test program that creates two `MyDate` objects (using `new MyDate()` and `new MyDate(34355555133101L)`) and displays their year, month, and day.

(*Hint*: The first two constructors will extract the year, month, and day from the elapsed time. For example, if the elapsed time is `561555550000` milliseconds, the year is

1987, the month is 9, and the day is 18. You may use the GregorianCalendar class discussed in Programming Exercise 8.5 to simplify coding.)

***10.15** (*Geometry: finding the bounding rectangle*) A bounding rectangle is the minimum rectangle that encloses a set of points in a two-dimensional plane, as shown in Figure 10.18d. Write a method that returns a bounding rectangle for a set of points in a two-dimensional plane, as follows:

```
public static MyRectangle2D getRectangle(double[][] points)
```

The Rectangle2D class is defined in Exercise 10.13. Write a test program that prompts the user to enter five points and displays the bounding rectangle's center, width, and height. Here is a sample run:

```
Enter five points: 1.0 2.5 3 4 5 6 7 8 9 10 ↵Enter
The bounding rectangle's center (5.0, 6.25), width 8.0, height 7.5
```

Sections 10.12–10.14

***10.16** (*Divisible by 2 or 3*) Find the first ten numbers with 50 decimal digits that are divisible by 2 or 3.

***10.17** (*Square numbers*) Find the first ten square numbers that are greater than Long.MAX_VALUE. A square number is a number in the form of n^2.

***10.18** (*Large prime numbers*) Write a program that finds five prime numbers larger than Long.MAX_VALUE.

***10.19** (*Mersenne prime*) A prime number is called a *Mersenne prime* if it can be written in the form $2^p - 1$ for some positive integer p. Write a program that finds all Mersenne primes with $p \leq 100$ and displays the output as shown below. (*Hint*: You have to use BigInteger to store the number, because it is too big to be stored in long. Your program may take several hours to run.)

```
      p        2^p - 1

      2           3
      3           7
      5          31
...
```

***10.20** (*Approximate e*) Programming Exercise 4.26 approximates e using the following series:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \ldots + \frac{1}{i!}$$

In order to get better precision, use BigDecimal with 25 digits of precision in the computation. Write a program that displays the e value for i = 100, 200, . . . , and 1000.

10.21 (*Divisible by 5 or 6*) Find the first ten numbers (greater than Long.MAX_VALUE) that are divisible by 5 or 6.