

5

Selection Statements

Objectives

After you have read and studied this chapter, you should be able to

- Implement selection control in a program using **if** statements.
- Implement selection control in a program using **switch** statements.
- Write boolean expressions using relational and boolean operators.
- Evaluate given boolean expressions correctly.
- Nest an **if** statement inside another **if** statement's **then** or **else** part correctly.
- Describe how objects are compared.
- Choose the appropriate selection control statement for a given task.
- Define and use enumerated constants.
- Draw geometric shapes on a window.

Introduction



D

ecisions, decisions, decisions. From the moment we are awake until the time we go to sleep, we are making decisions. Should I eat cereal or toast? What should I wear to school today? Should I eat at the cafeteria today? And so forth. We make many of these decisions by evaluating some criteria. If the number of students in line for registration seems long, then come back tomorrow for another try. If today is Monday, Wednesday, or Friday, then eat lunch at the cafeteria.

Computer programs are no different. Any practical computer program contains many statements that make decisions. Often a course of action is determined by evaluating some kind of a test (e.g., Is the remaining balance of a meal card below the minimum?). Statements in programs are executed in sequence, which is called *sequential execution* or *sequential control flow*. However, we can add decision-making statements to a program to alter this control flow. For example, we can add a statement that causes a portion of a program to be skipped if an input value is greater than 100. Or we can add a statement to disallow the purchase of food items if the balance of a meal card goes below a certain minimum. The statement that alters the control flow is called a *control statement*. In this chapter we describe some important control statements, called *selection statements*. In Chapter 6 we will describe other control statements, called *repetition statements*.

sequential
execution

control
statement

5.1 The if Statement

if statement

There are two versions of the *if statement*, called *if-then-else* and *if-then*. We begin with the first version. Suppose we wish to enter a student's test score and print out the message You did not pass if the score is less than 70 and You did pass if the score is 70 or higher. Here's how we express this logic in Java:

```
Scanner scanner = new Scanner(System.in);
System.out.print("Enter test score: ");
int testScore = scanner.nextInt();
if (testScore < 70)
    System.out.println("You did not pass");
else
    System.out.println("You did pass");
```

This statement is
executed if **testScore**
is less than 70.

This statement is
executed if **testScore**
is 70 or higher.

We use an if statement to specify which block of code to execute. A block of code may contain zero or more statements. Which block is executed depends on the

Hints,
& Tips,
Pitfalls

One very common error in writing programs is to mix up the assignment and equality operators. We frequently make the mistake of writing

```
if (x = 5) ...
```

when we actually wanted to say

```
if (x == 5) ...
```

If the boolean expression evaluates to true, then the statements in the <then block> are executed. Otherwise, the statements in the <else block> are executed. We will cover more complex boolean expressions in Section 5.2. Notice that we can reverse the relational operator and switch the then and else blocks to derive the equivalent code, for example,

```
if (testScore >= 70)
    System.out.println("You did pass");
else
    System.out.println("You did not pass");
```

Notice that the reverse of < is >=, not >.

selection
statement

The if statement is called a *selection* or *branching statement* because it selects (or branches to) one of the alternative blocks for execution. In our example, either

```
System.out.println("You did not pass");
```

or

```
System.out.println("You did pass");
```

is executed depending on the value of the boolean expression. We can illustrate a branching path of execution with the diagram shown in Figure 5.2.

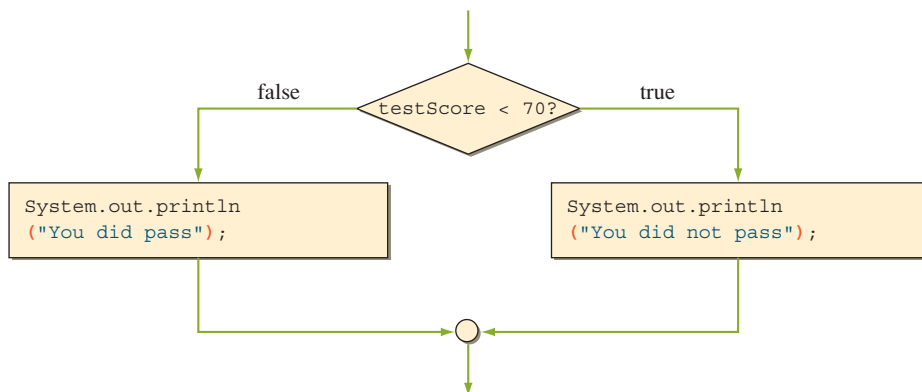


Figure 5.2 The diagram showing the control flow of the sample if-then-else statement.

In the preceding if statement, both blocks contain only one statement. The then or else block can contain more than one statement. The general format for both the <then block> and the <else block> is either a

```
<single statement>
```

or a

```
<compound statement>
```

where <single statement> is a Java statement and <compound statement> is a sequence of Java statements surrounded by braces, as shown below with $n \geq 0$ statements:

```
{
  <statement 1>
  <statement 2>
  ...
  <statement n>
}
```

If multiple statements are needed in the <then block> or the <else block>, they must be surrounded by braces { and }. For example, suppose we want to print out additional messages for each case. Let's say we also want to print Keep up the good work when the student passes and print Try harder next time when the student fails. Here's how:

```

if (testScore < 70)
{
  System.out.println("You did not pass");
  System.out.println("Try harder next time");
}
else
{
  System.out.println("You did pass");
  System.out.println("Keep up the good work");
}

```

The diagram illustrates an if-else statement. The condition is `testScore < 70`. The if block contains two lines of code: `System.out.println("You did not pass");` and `System.out.println("Try harder next time");`. The else block contains two lines of code: `System.out.println("You did pass");` and `System.out.println("Keep up the good work");`. Both blocks are enclosed in curly braces. A label "Compound Statements" on the left has two arrows pointing to the two blocks, indicating that each block contains a compound statement.

The braces are necessary to delineate the statements inside the block. Without the braces, the compiler will not be able to tell whether a statement is part of the block or part of the statement that follows the if statement.

Notice the absence of semicolons after the right braces. A semicolon is never necessary immediately after a right brace. A compound statement may contain zero

or more statements, so it is perfectly valid for a compound statement to include only one statement. Indeed, we can write the sample if statement as

```

if (testScore < 70)
{
    System.out.println("You did not pass");
}

else

{
    System.out.println("You did pass");
}

```

Although it is not required, many programmers prefer to use the syntax for the compound statement even if the then or else block includes only one statement. In this textbook, we use the syntax for the compound statement regardless of the number of statements inside the then and else blocks. Following this policy is beneficial for a number of reasons. One is the ease of adding temporary output statements inside the blocks. Frequently, we want to include a temporary output statement to verify that the boolean expression is written correctly. Suppose we add output statements such as these:

```

if (testScore < 70)
{
    System.out.println("inside then: " + testScore);
    System.out.println("You did not pass");
}
else
{
    System.out.println("inside else: " + testScore);
    System.out.println("You did pass");
}

```

If we always use the syntax for the compound statement, we just add and delete the temporary output statements. However, if we use the syntax of the single statement, then we have to remember to add the braces when we want to include a temporary output statement. Another reason for using the compound statement syntax exclusively is to avoid the dangling else problem. We discuss this problem in Section 5.2.

The placement of left and right braces does not matter to the compiler. The compiler will not complain if you write the earlier if statement as

```

if (testScore < 70)
{ System.out.println("You did not pass");
  System.out.println("Try harder next time");} else
{
  System.out.println("You did pass");
  System.out.println("Keep up the good work");}

```

However, to keep your code readable and easy to follow, you should format your if statements using one of the two most common styles:

Style 1

```
if ( <boolean expression> ) {
    ...
} else {
    ...
}
```

Style 2

```
if ( <boolean expression> )
{
    ...
}
else
{
    ...
}
```

In this book, we will use style 1, mainly because this style adheres to the code conventions for the Java programming language. If you prefer style 2, then go ahead and use it. Whichever style you choose, be consistent, because a consistent look and feel is very important to make your code readable.



The document that provides the details of code conventions for Java can be found at <http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html>

This document describes the Java language coding standards dictated in the Java Language Specification. It is important to follow the code conventions as closely as possible to increase the readability of the software.

There is a second variation of style 1 in which we place the reserved word else on a new line as

Style 3

```
if ( <boolean expression> ) {
    ...
}
else {
    ...
}
```

Many programmers prefer this variation of style 1 because the reserved word else aligns with the matching if. However, if we nitpick, style 3 goes against the logic behind the recommended style 1 format, which is to begin a new statement at one position with a reserved word. The reserved word else is a part of the if statement, not the beginning of a new statement. Thus style 1 places the reserved word else to the right of the matching if.

Again, the actual format is not that important. Consistent use of the same format is. So, whichever style you use, use it consistently. To promote consistency among all programmers, we recommend that everybody to adopt the code conventions. Even though the recommended format may look peculiar at first, with some repeated use, the format becomes natural in no time.

Let's summarize the key points to remember:

Things to Remember



Rules for writing the **then** and **else** blocks:

1. Left and right braces are necessary to surround the statements if the **then** or **else** block contains multiple statements.
2. Braces are not necessary if the **then** or **else** block contains only one statement.
3. A semicolon is not necessary after a right brace.

Now let's study a second version of the if statement called *if-then*. Suppose we want to print out the message You are an honor student if the test score is 95 or above and print out nothing otherwise. For this type of testing, we use the second version of the if statement, whose general format is

if-then syntax

```
if ( <boolean expression> )
    <then block>
```

The second version contains only the <then block>. Using this version and the compound statement syntax, we express the selection control as

```
if (testScore >= 95) {
    System.out.println("You are an honor student");
}
```

Figure 5.3 shows the diagram that illustrates the control flow for this if-then statement. We will refer collectively to both versions as the *if* statement.

Notice that the if-then statement is not necessary, because we can write any if-then statement using if-then-else by including no statement in the else block. For instance, the sample if-then statement can be written as

```
if (testScore >= 95) {
    System.out.println("You are an honor student");
} else { }
```

In this book, we use if-then statements whenever appropriate.

Let's conclude this section with a sample class that models a circle. We will name the class Ch5Circle, and its instances are capable of computing the

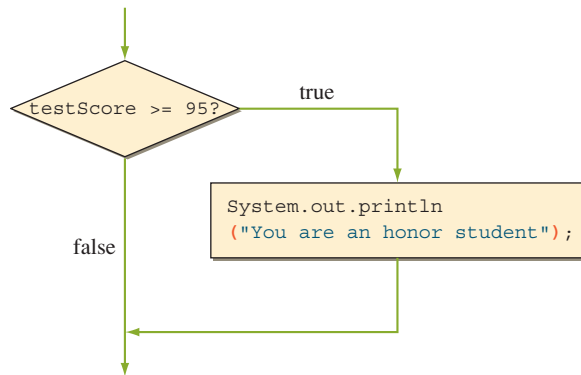


Figure 5.3 The diagram showing the control flow of the second version of the **if** statement.

circumference and area. We will include a test in this class so the methods such as `getArea` and `getCircumference` return the constant `INVALID_DIMENSION` when the dimension of the radius is invalid. Here's the `Ch5Circle` class (most comments are removed for the sake of brevity):

```

/*
  Chapter 5 The Circle class
  File: Ch5Circle.java
*/
class Ch5Circle {
    public static final int INVALID_DIMENSION = -1;
    private double radius;

    public Ch5Circle(double r) {
        setRadius(r);
    }

    public double getArea( ) {
        double result = INVALID_DIMENSION;

        if (isRadiusValid()) {
            result = Math.PI * radius * radius;
        }

        return result;
    }
}
  
```

As the number of methods gets larger, we will use this marker to quickly locate the program components. Shaded icon is used for a private element.

Data Members

`getArea`

```
public double getCircumference( ) {  
    double result = INVALID_DIMENSION;  
    if (isRadiusValid()) {  
        result = 2.0 * Math.PI * radius;  
    }  
    return result;  
}
```

getCircumference

```
public double getDiameter( ) {  
    double diameter = INVALID_DIMENSION;  
    if (isRadiusValid()) {  
        diameter = 2.0 * radius;  
    }  
    return diameter;  
}
```

getDiameter

```
public double getRadius( ) {  
    return radius;  
}
```

getRadius

```
public void setDiameter(double d) {  
    if (d > 0) {  
        setRadius(d/2.0);  
    } else {  
        setRadius(INVALID_DIMENSION);  
    }  
}
```

setDiameter

```
public void setRadius(double r) {  
    if (r > 0) {  
        radius = r;  
    } else {  
        radius = INVALID_DIMENSION;  
    }  
}
```

setRadius

```
private boolean isRadiusValid( ) {  
    return radius != INVALID_DIMENSION;  
}
```

isRadiusValid

}

Notice the if statement in the getArea method is written as

```
    if (isRadiusValid()) {
        ...
    }
```

The <boolean expression> in the if statement can be any expression that evaluates to true or false, including a call to a method whose return type is boolean, such as the isRadiusValid method. The use of such a boolean method often makes the code easier to read, and easier to modify if the boolean method is called from many methods (e.g., there are three methods calling the isRadiusValid method).

Here's a short main class to test the functionality of the Ch5Circle class:

```
/*
   Chapter 5 Sample Program: Computing Circle Dimensions
   File: Ch5Sample1.java
*/

import java.util.*;

class Ch5Sample1 {

    public static void main(String[] args) {

        double    radius, circumference, area;

        Ch5Circle circle;

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter radius: ");
        radius = scanner.nextDouble();

        circle = new Ch5Circle(radius);

        circumference = circle.getCircumference();

        area          = circle.getArea();

        System.out.println("Input radius: " + radius);
        System.out.println("Circumference: " + circumference);
        System.out.println("Area:          " + area);

    }
}
```

Notice that the program will display -1.0 when the input radius is invalid. We can improve the display by adding an if test in the main program as follows:

```
System.out.print("Circumference: ");
if (circumference == Ch5Circle.INVALID_DIMENSION) {
    System.out.println("Cannot compute. Input invalid");
} else {
    System.out.println(circumference);
}
```

Another possible improvement in the main program is to check the input value first. For instance,

```
radius = ... ;
if (radius > 0) {
    //do the computation as the sample main method
} else {
    //print out the error message
}
```

Even when a client programmer does not include appropriate tests in his program, we must define a reusable class in a robust manner so it will not crash or produce erroneous results. For the Ch5Circle class, we add a test so the data member radius is set to either a valid datum or a specially designated value (INVALID_DIMENSION) for any invalid data. By designing the class in this manner, we protect the class from a possible misuse (e.g., attempting to assign a negative radius) and producing meaningless results, such as -5.88. We always strive for a reliable and robust reusable class that will withstand the abuse and misuse of client programmers.



1. Identify the invalid if statements:

a. **if** (a < b) then
 x = y;
else
 x = z;

b. **if** (a < b)
 else x = y;

c. **if** (a < b)
 x = y;
else {
 x = z;
};

d. **if** (a < b) {
 x = y; } **else**
 x = z;

2. Are the following two if statements equivalent?

```
/*A*/ if ( x < y )
    System.out.println("Hello");
else
    System.out.println("Bye");

/*B*/ if ( x > y )
    System.out.println("Bye");
else
    System.out.println("Hello");
```

5.2 | Nested if Statements

nested if
statement

The then and else blocks of an if statement can contain any statement including another if statement. An if statement that contains another if statement in either its then or else block is called a *nested if* statement. Let's look at an example. In the earlier example, we printed out the messages You did pass or You did not pass depending on the test score. Let's modify the code to print out three possible messages. If the test score is lower than 70, then we print You did not pass, as before. If the test score is 70 or higher, then we will check the student's age. If the age is less than 10, we will print You did a great job. Otherwise, we will print You did pass, as before. Figure 5.4 is a diagram showing the logic of this nested test. The code is written as follows:

```

if (testScore >= 70) {
    if (studentAge < 10) {
        System.out.println("You did a great job");
    } else {
        System.out.println("You did pass");//test score >= 70
        //and age >= 10
    }
} else { //test score < 70
    System.out.println("You did not pass");
}

```

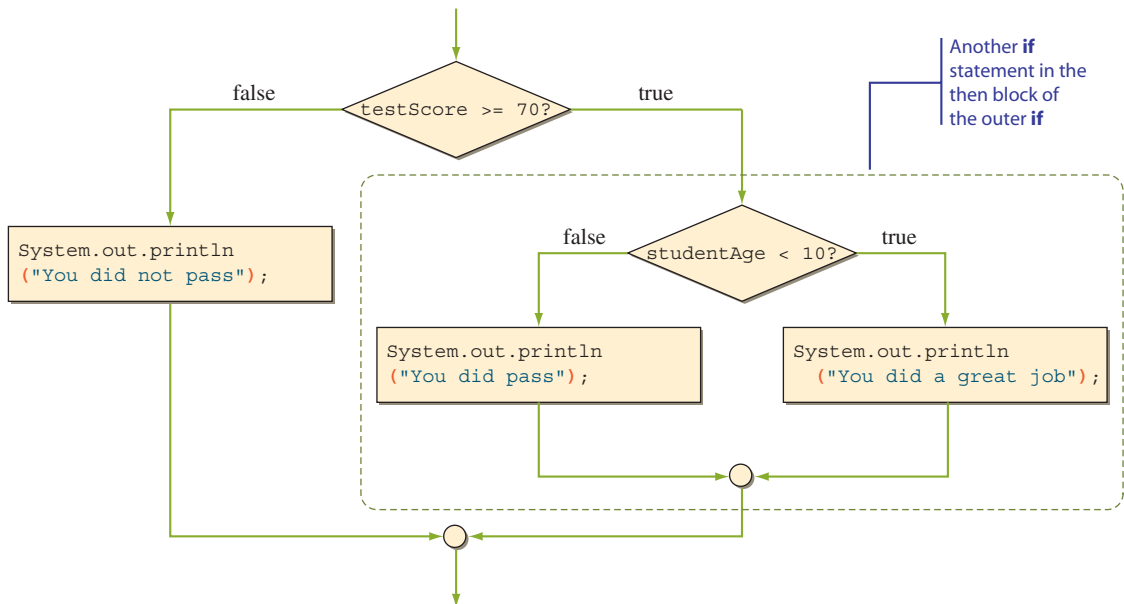


Figure 5.4 A diagram showing the control flow of the example nested if statement.

Since the then block of the outer if contains another if statement, the outer if is called a nested if statement. It is possible to write if tests in different ways to achieve the same result. For example, the preceding code can also be expressed as

```

if (testScore >= 70 && studentAge < 10) {
    System.out.println("You did a great job");
} else {
    //either testScore < 70 OR studentAge >= 10

    if (testScore >= 70) {
        System.out.println("You did pass");
    } else {
        System.out.println("You did not pass");
    }
}

```

Several other variations can also achieve the same result. As a general rule, we strive to select the one that is most readable (i.e., most easily understood) and most efficient. Often no one variation stands out, and the one you choose depends on your preferred style of programming.

Here's an example in which one variation is clearly a better choice. Suppose we input three integers and determine how many of them are negative. Here's the first variation. To show the structure more clearly, we purposely do not use the braces in the then and else blocks.

```

if (num1 < 0)
    if (num2 < 0)
        if (num3 < 0)
            negativeCount = 3; //all three are negative
        else
            negativeCount = 2; //num1 and num2 are negative
    else
        if (num3 < 0)
            negativeCount = 2; //num1 and num3 are negative
        else
            negativeCount = 1; //num1 is negative
else
    if (num2 < 0)
        if (num3 < 0)
            negativeCount = 2; //num2 and num3 are negative
        else
            negativeCount = 1; //num2 is negative
    else
        if (num3 < 0)
            negativeCount = 1; //num3 is negative
        else
            negativeCount = 0; //no negative numbers

```

In this and the following examples, we purposely do not use the braces so we can provide a better illustration of the topics we are presenting.

It certainly did the job. But elegantly? Here's the second variation:

```
negativeCount = 0;
if (num1 < 0)
    negativeCount = negativeCount + 1;
if (num2 < 0)
    negativeCount = negativeCount + 1;
if (num3 < 0)
    negativeCount = negativeCount + 1;
```

Which version should we use? The second variation is the only reasonable way to go. The first variation is not a viable option because it is very inefficient and very difficult to read. We apply the nested if structure if we have to test conditions in some required order. In this example these three tests are independent of one another, so they can be executed in any order. In other words, it doesn't matter whether we test num1 first or last.

The statement

```
negativeCount = negativeCount + 1;
```

increments the variable by 1. This type of statement that changes the value of a variable by adding a fixed number occurs frequently in programs. Instead of repeating the same variable name twice, we can write it succinctly as

```
negativeCount++;
```

Similarly, a statement such as

```
count = count - 1;
```

can be written as

```
count--;
```

increment and
decrement
operators

The double plus operator (`++`) is called the *increment operator*, and the double minus operator (`--`) is the *decrement operator* (which decrements the variable by 1). The increment and decrement operators have higher precedence than unary operators. See Table 5.3 on page 235. *Note:* There are prefix and postfix increment/decrement operators in which the operators come before and after the variable (or an expression), respectively. We only use the postfix operators in this book, and the precedence rules presented in the table apply to the postfix operators only.

Notice that we indent the then and else blocks to show the nested structure clearly. Indentation is used as a visual guide for the readers. It makes no difference to a Java compiler. For example, we make our intent clear by writing the statement as

```
if (x < y)
    if (z != w)
        a = b + 1;
    else
        a = c + 1;
else
    a = b * c;
```

Hints, & Tips, Pitfalls



It takes some practice before you can write well-formed **if** statements. Here are some rules to help you write the **if** statements.

- Rule 1:** Minimize the number of nestings.
- Rule 2:** Avoid complex boolean expressions. Make them as simple as possible. Don't include many ANDs and ORs.
- Rule 3:** Eliminate any unnecessary comparisons.
- Rule 4:** Don't be satisfied with the first correct statement. Always look for improvement.
- Rule 5:** Read your code again. Can you follow the statement easily? If not, try to improve it.

But to the Java compiler, it does not matter if we write the same code as

```
if (x < y) if (z != w) a = b + 1; else a = c + 1; else a = b * c;
```

Although indentation is not required to run the program, using proper indentation is an important aspect of good programming style. Since the goal is to make your code readable, not to follow any one style of indentation, you are free to choose your own style. We recommend style 1 shown on page 219.

The next example shows a style of indentation accepted as standard for a nested if statement in which nesting occurs only in the else block. Instead of determining whether a student passes or not, we will now display a letter grade based on the following formula:

Test Score	Grade
$90 \leq \text{score}$	A
$80 \leq \text{score} < 90$	B
$70 \leq \text{score} < 80$	C
$60 \leq \text{score} < 70$	D
$\text{score} < 60$	F

The statement can be written as

```
if (score >= 90)
    System.out.println("Your grade is A");
else
    if (score >= 80)
        System.out.println("Your grade is B");
    else
        if (score >= 70)
            System.out.println("Your grade is C");
```



```

else
    if (score >= 60)
        System.out.println("Your grade is D");
    else
        System.out.println("Your grade is F");

```

However, the standard way to indent the statement is as follows:

```

if (score >= 90)
    System.out.println("Your grade is A");

else if (score >= 80)
    System.out.println("Your grade is B");

else if (score >= 70)
    System.out.println("Your grade is C");

else if (score >= 60)
    System.out.println("Your grade is D");

else
    System.out.println("Your grade is F");

```

We mentioned that indentation is meant for human eyes only. For example, we can clearly see the intent of a programmer just by looking at the indentation when we read

```

if (x < y)
    if (x < z)
        System.out.println("Hello");
else
    System.out.println("Good bye");

```

Indentation style A

A Java compiler, however, will interpret the above as

```

if (x < y)
    if (x < z)
        System.out.println("Hello");
else
    System.out.println("Good bye");

```

Indentation style B

dangling else
problem

This example has a *dangling else problem*. The Java compiler matches an else with the previous unmatched if, so the compiler will interpret the statement by matching the else with the inner if (if (x < z)), whether you use indentation style A or B. If you want to express the logic of indentation style A, you have to express it as

```

if (x < y) {
    if (x < z)
        System.out.println("Hello");
} else
    System.out.println("Good bye");

```

This dangling else problem is another reason why we recommend that beginners use the syntax for <compound statement> in the then and else blocks. In other words, always use the braces in the then and else blocks.

Let's conclude this section by including tests inside the add and deduct methods of the Account class from Chapter 4. For both methods, we will update the balance only when the amount passed is positive. Furthermore, for the deduct method, we will update the balance only if it does not become a negative number after the deduction. This will require the use of a nested if statement. The following is the class declaration. The name is Ch5Account, and this class is based on AccountVer2 from Chapter 4. We only list the two methods here because other parts are the same as in AccountVer2.

```
class Ch5Account {
    ...
    //Adds the passed amount to the balance
    public void add(double amt) {
        //add if amt is positive; otherwise, do nothing
        if (amt > 0) {
            balance = balance + amt;
        }
    }
    //Deducts the passed amount from the balance
    public void deduct(double amt) {
        //deduct if amt is positive; do nothing otherwise
        if (amt > 0) {
            double newbalance = balance - amt;
            if (newbalance >= 0) { //if a new balance is positive, then
                balance = newbalance; //update the balance; otherwise,
            } //do nothing.
        }
    }
    ...
}
```

add

deduct



1. Rewrite the following nested if statements without using any nesting.

```
a. if ( a < c )
    if ( b < c )
        x = y;
    else
        x = z;
```

```

else
    x = z;
b. if ( a == b )
    x = y;
else
    if ( a > b )
        x = y;
    else
        x = z;
c. if ( a < b )
    if ( a >= b )
        x = z;
    else
        x = y;
else
    x = z;

```

2. Format the following if statements with indentation.

```

a. if ( a < b ) if ( c > d ) x = y;
   else x = z;
b. if ( a < b ) { if ( c > d ) x = y; }
   else x = z;
c. if ( a < b ) x = y; if ( a < c ) x = z;
   else if ( c < d ) z = y;

```

5.3 Boolean Expressions and Variables

boolean
operator

In addition to the arithmetic operators introduced in Chapter 3 and relational operators introduced in Section 5.2, boolean expressions can contain conditional and boolean operators. A *boolean operator*, also called a *logical operator*, takes boolean values as its operands and returns a boolean value. Three boolean operators are AND, OR, and NOT. In Java, the symbols `&&`, `||`, and `!` represent the AND, OR, and NOT operators, respectively. Table 5.1 explains how these operators work.

The AND operation results in true only if both P and Q are true. The OR operation results in true if either P or Q is true. The NOT operation is true if A is false and is false if P is true. Combining boolean operators with relational and arithmetic operators, we can come up with long boolean expressions such as

```

(x + 150) == y || x < y && !(y < z && z < x)
(x < y) && (a == b || a == c)
a != 0 && b != 0 && (a + b < 10)

```

In Section 5.1 we stated that we can reverse the relational operator and switch the then and else blocks to derive the equivalent code. For example,

```

if (age < 0) {
    System.out.println("Invalid age is entered");
} else {
    System.out.println("Valid age is entered");
}

```

Table 5.1 Boolean operators and their meanings

P	Q	P && Q	P Q	!P
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

is equivalent to

```
if ( !(age < 0) ) {
    System.out.println("Valid age is entered");
} else {
    System.out.println("Invalid age is entered");
}
```

which can be written more naturally as

```
if (age >= 0) {
    System.out.println("Valid age is entered");
} else {
    System.out.println("Invalid age is entered");
}
```

Reversing the relational operator means negating the boolean expression. In other words, `!(age < 0)` is equivalent to `(age >= 0)`. Now, consider the following if-then-else statement:

```
if (temperature >= 65 && distanceToDestination < 2) {
    System.out.println("Let's walk");
} else {
    System.out.println("Let's drive");
}
```

If the temperature is greater than or equal to 65 degrees and the distance to the destination is less than 2 mi., we walk. Otherwise (it's too cold or too far away), we drive. How do we reverse the if-then-else statement? We can rewrite the statement by negating the boolean expression and switching the then and else blocks as

```
if ( !(temperature >= 65 && distanceToDestination < 2) ) {
    System.out.println("Let's drive");
} else {
    System.out.println("Let's walk");
}
```

or more directly and naturally as

```
if (temperature < 65 || distanceToDestination >= 2) {
    System.out.println("Let's drive");
} else {
    System.out.println("Let's walk");
}
```

The expression

```
!(temperature >= 65 && distanceToDestination < 2)
```

is equivalent to

```
!(temperature >= 65) || !(distanceToDestination < 2)
```

which, in turn, is equivalent to

```
(temperature < 65 || distanceToDestination >= 2)
```

The logical equivalence is derived by applying the following DeMorgan's law:

Rule 1: $!(P \ \&\& \ Q) \iff !P \ || \ !Q$

Rule 2: $!(P \ || \ Q) \iff !P \ \&\& \ !Q$

Equivalence symbol

Table 5.2 shows their equivalence.

Now consider the following expression:

```
x / y > z || y == 0
```

arithmetic
exception

What will be the result if y is equal to 0? Easy, the result is true, many of you might say. Actually a runtime error called an *arithmetic exception* will result, because the expression

```
x / y
```

Table 5.2 The truth table illustrating DeMorgan's law

P	Q	!(P && Q)	!P !Q	!(P Q)	!P && !Q
false	false	true	true	true	true
false	true	true	true	false	false
true	false	true	true	false	false
true	true	false	false	false	false

divide-by-zero
error

causes a problem known as a *divide-by-zero error*. Remember that you cannot divide a number by zero.

However, if we reverse the order to

```
y == 0 || x / y > z
```

then no arithmetic exception will occur because the test $x / y > z$ will not be evaluated. For the OR operator `||`, if the left operand is evaluated to true, then the right operand will not be evaluated, because the whole expression is true, whether the value of the right operand is true or false. We call such an evaluation method a *short-circuit evaluation*. For the AND operator `&&`, the right operand need not be evaluated if the left operand is evaluated to false, because the result will then be false whether the value of the right operand is true or false.

short-circuit
evaluation

Just as the operator precedence rules are necessary to evaluate arithmetic expressions unambiguously, they are required for evaluating boolean expressions. Table 5.3 expands Table 3.3 by including all operators introduced so far.

In mathematics, we specify the range of values for a variable as

$$80 \leq x < 90$$

In Java, to test that the value for x is within the specified lower and upper bounds, we express it as

```
80 <= x && x < 90
```

You cannot specify it as

```
80 <= x < 90
```



Wrong

This is a syntax error because the relational operators (`<`, `<=`, etc.) are binary operators whose operands must be numerical values. Notice that the result of the subexpression

```
80 <= x
```

is a boolean value, which cannot be compared to the numerical value 90. Their data types are not compatible.

The result of a boolean expression is either true or false, which are the two values of data type `boolean`. As is the case with other data types, a value of a data type can be assigned to a variable of the same data type. In other words, we can declare a variable of data type `boolean` and assign a boolean value to it. Here are examples:

```
boolean pass, done;

pass = 70 < x;
done = true;
```

One possible use of boolean variables is to keep track of the program settings or user preferences. A variable (of any data type, not just boolean) used for this

Table 5.3

Operator precedence rules. Groups are listed in descending order of precedence. An operator with a higher precedence will be evaluated first. If two operators have the same precedence, then the associativity rule is applied

Group	Operator	Precedence	Associativity
Subexpression	()	10 (If parentheses are nested, then innermost subexpression is evaluated first.)	Left to right
Postfix increment and decrement operators	++ --	9	Right to left
Unary operators	- !	8	Right to left
Multiplicative operators	* / %	7	Left to right
Additive operators	+ -	6	Left to right
Relational operators	< <= > >=	5	Left to right
Equality operators	== !=	4	Left to right
Boolean AND	&&	3	Left to right
Boolean OR		2	Left to right
Assignment	=	1	Right to left

flag

purpose is called a *flag*. Suppose we want to allow the user to display either short or long messages. Many people, when using a new program, prefer to see long messages, such as Enter a person's age and press the Enter key to continue. But once they are familiar with the program, many users prefer to see short messages, such as Enter age. We can use a boolean flag to remember the user's preference. We can set the flag `longMessageFormat` at the beginning of the program to true or false depending on the user's choice. Once this boolean flag is set, we can refer to the flag at different points in the program as follows:

```
if (longMessageFormat) {
    //display the message in long format
```

```

    } else {
        //display the message in short format
    }

```

Notice the value of a boolean variable is `true` or `false`, so even though it is valid, we do not write a boolean expression as

```

if (isRaining == true) {
    System.out.println("Store is open");
} else {
    System.out.println("Store is closed");
}

```

but more succinctly as

```

if (isRaining) {
    System.out.println("Store is open");
} else {
    System.out.println("Store is closed");
}

```

Another point that we have to be careful about in using boolean variables is the choice of identifier. Instead of using a boolean variable such as `motionStatus`, it is more meaningful and descriptive to use the variable `isMoving`. For example, the statement

```

if (isMoving) {
    //the mobile robot is moving
} else {
    //the mobile robot is not moving
}

```

is much clearer than the statement

```

if (motionStatus) {
    //the mobile robot is moving
} else {
    //the mobile robot is not moving
}

```

When we define a boolean data member for a class, it is a Java convention to use the prefix `is` instead of `get` for the accessor.

We again conclude the section with a sample class. Let's improve the `Ch5Account` class by adding a boolean data member `active` to represent the state of an account. When an account is first open, it is set to an active state. Deposits and

withdrawals can be made only when the account is active. If the account is inactive, then the requested operation is ignored. Here's how the class is defined (the actual class name is Ch5AccountVer2):

```
class Ch5AccountVer2 {  
  
    // Data Members  
    private String ownerName;  
  
    private double balance;  
  
    private boolean active;  
  
    //Constructor  
    public Ch5AccountVer2(String name, double startingBalance ) {  
  
        ownerName = name;  
        balance = startingBalance;  
  
        setActive(true);  
    }  
  
    //Adds the passed amount to the balance  
    public void add(double amt) {  
        //add if amt is positive; do nothing otherwise  
        if (isActive() && amt > 0) {  
            balance = balance + amt;  
        }  
    }  
  
    //Closes the account; set 'active' to false  
    public void close() {  
  
        setActive(false);  
    }  
  
    //Deducts the passed amount from the balance  
    public void deduct(double amt) {  
  
        //deduct if amt is positive; do nothing otherwise  
        if (isActive() && amt > 0) {  
            double newbalance = balance - amt;  
  
            if (newbalance >= 0) { //don't let the balance become negative  
                balance = newbalance;  
            }  
        }  
    }  
}
```

Data Members

add

close

deduct

<code>//Returns the current balance of this account public double getCurrentBalance() { return balance; }</code>	getCurrentBalance
<code>//Returns the name of this account's owner public String getOwnerName() { return ownerName; }</code>	getOwnerName
<code>//Is the account active? public boolean isActive() { return active; }</code>	isActive
<code>//Assigns the name of this account's owner public void setOwnerName(String name) { ownerName = name; }</code>	setOwnerName
<code>//Sets 'active' to true or false private void setActive(boolean state) { active = state; }</code>	setActive

Quick CHECK

- Evaluate the following boolean expressions. Assume x, y, and z have some numerical values.
 - `4 < 5 || 6 == 6`
 - `2 < 4 && (false || 5 <= 4)`
 - `x <= y && !(z != z) || x > y`
 - `x < y || z < y && y <= z`
- Identify errors in the following boolean expressions and assignments. Assume x and y have some numerical values.
 - `boolean done;
done = x = y;`
 - `2 < 4 && (3 < 5) + 1 == 3`
 - `boolean quit;
quit = true;
quit == (34 == 20) && quit;`

Hints, & Tips, Pitfalls



We introduced the logical AND and OR operations using the symbols `&&` and `||`. In Java, there are single ampersand and single vertical bar operations. For example, if we write an `if` statement as

```
if ( 70 <= x & x < 90 )
```

it will compile and run. Unlike the double ampersand, the single ampersand will not do a short-circuit evaluation. It will evaluate both left and right operands. The single vertical bar works in an analogous manner. So, which one should we use? Use double ampersand for AND and double vertical bars for OR. We will most likely never encounter a situation where we cannot use the double ampersand or the double vertical bars.

5.4 | Comparing Objects

With primitive data types, we have only one way to compare them, but with objects (reference data type), we have two ways to compare them. We discuss the ways the objects can be compared in this section. First, let's review how we compare primitive data types. What would be the output of the following code?

```
int num1, num2;

num1 = 15;
num2 = 15;

if (num1 == num2) {
    System.out.println("They are equal");
} else {
    System.out.println("They are not equal");
}
```

Because the two variables hold the same value, the output is

```
They are equal
```

Now, let's see how the objects can be compared. We will use `String` objects for illustration. Since we use string data all the time in our programs, it is very important for us to understand perfectly how `String` objects can be compared.

Consider the following code that attempts to compare two `String` objects:

```
String str1, str2;

str1 = new String("Java");
str2 = new String("Java");

if (str1 == str2) {
    System.out.println("They are equal");
}
```

```

    } else {
        System.out.println("They are not equal");
    }
}

```

What would be an output? The answer is

```
They are not equal
```

The two objects are constructed with the same sequence of characters, but the result of comparison came back that they were not equal. Why?

When two variables are compared, we are comparing their contents. In the case of primitive data types, the content is the actual value. In case of reference data types, the content is the address where the object is stored. Since there are two distinct `String` objects, stored at different addresses, the contents of `str1` and `str2` are different, and therefore, the equality testing results in false. If we change the code to

```

String str1, str2;

str1 = new String("Java");
str2 = str1;

if (str1 == str2) {
    System.out.println("They are equal");
} else {
    System.out.println("They are not equal");
}

```

No new object is created here. The content (address) of `str1` is copied to `str2`, making them both point to the same object.

then the output would be

```
They are equal
```

because now we have one `String` object and both variables `str1` and `str2` point to this object. This means the contents of `str1` and `str2` are the same because they refer to the same address. Figure 5.5 shows the distinction.

What can we do if we need to check whether two distinct `String` objects have the same sequence of characters? Many standard classes include different types of comparison methods. The `String` class, for example, includes the `equals` and `equalsIgnoreCase` comparison methods. The `equals` method returns true if two `String` objects have the exact same sequence of characters. The `equalsIgnoreCase` method does the same as the `equals` method, but the comparison is done in a case-insensitive manner. Using the `equals` method, we can rewrite the first sample code as

```

String str1, str2;

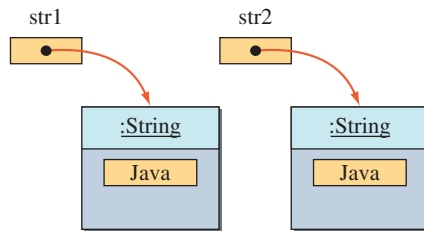
str1 = new String("Java");
str2 = new String("Java");

if (str1.equals(str2)) {
    System.out.println("They are equal");
} else {
    System.out.println("They are not equal");
}

```

Use the `equals` method.

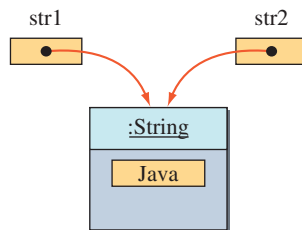
Case A: Two variables refer to two different objects.



```
String str1, str2;
str1 = new String("Java");
str2 = new String("Java");
```

`str1 == str2` → false

Case B: Two variables refer to the same object.



```
String str1, str2;
str1 = new String("Java");
str2 = str1;
```

`str1 == str2` → true

Figure 5.5 How the equality `==` testing works with the objects.

and get the result

```
They are equal
```

Just as the `String` and many standard classes provide the `equals` method, it is common to include such an `equals` method in programmer-defined classes also. Consider a `Fraction` class. We say two `Fraction` objects are equal if they have the same value for the numerator and the denominator. Here's how we can define the `equals` method for the `Fraction` class:

```
class Fraction {
    private int numerator;
    private int denominator;
    ...
    //constructor and other methods
    ...
    public int getNumerator( ) {
        return numerator;
    }
}
```

```

public int getDenominator( ) {
    return denominator;
}

public boolean equals(Fraction number) {
    return (numerator == number.getNumerator()
        && denominator == number.getDenominator());
}

```

Compare this object's values to the values of **number**

Notice that the body of the equals method is a concise version of

```

if (numerator == number.getNumerator( )
    && denominator == number.getDenominator()) {

    return true;

} else {

    return false;

}

```

Using the equals method, we can compare two Fraction objects in the following manner:

```

Fraction frac1, frac2;

//create frac1 and frac2 objects
...

if (frac1.equals(frac2)) {
    ...
}

```

or equivalently as

```

if (frac2.equals(frac1)) {
    ...
}

```

Note that the equals method as defined is incomplete. For example, if we compare fractions $4/8$ and $3/6$, using this equals method, we get false as the result because the method does not compare the fractions in their simplified form. The method should have reduced both $4/8$ and $3/6$ to $1/2$ and then compared. To implement a method that reduces a fraction to its simplest form, we need to use a repetition control statement. We will revisit this problem when we learn how to write repetition control statements in Chapter 6. Also, we will provide the complete definition of the Fraction class in Chapter 7.

We conclude this section by presenting an exception to the rule for comparing objects. This exception applies to the String class only. We already mentioned in

Chapter 2 that for the `String` class only, we do not have to use the `new` operator to create an instance. In other words, instead of writing

```
String str = new String("Java");
```

we can write

```
String str = "Java";
```

which is a more common form. These two statements are not identical in terms of memory allocation, which in turn affects how the string comparisons work. Figure 5.6 shows the difference in assigning a `String` object to a variable. If we do not use the `new` operator, then string data are treated as if they are a primitive data type. When we use the same literal `String` constants in a program, there will be exactly one `String` object.

This means we can use the equal symbol `==` to compare `String` objects when no `new` operators are used. However, regardless of how the `String` objects are created, it is always correct and safe to use the equals and other comparison methods to compare two strings.

Things to Remember



Always use the **equals** and other comparison methods to compare **String** objects. Do not use `==` even though it may work correctly in certain cases.

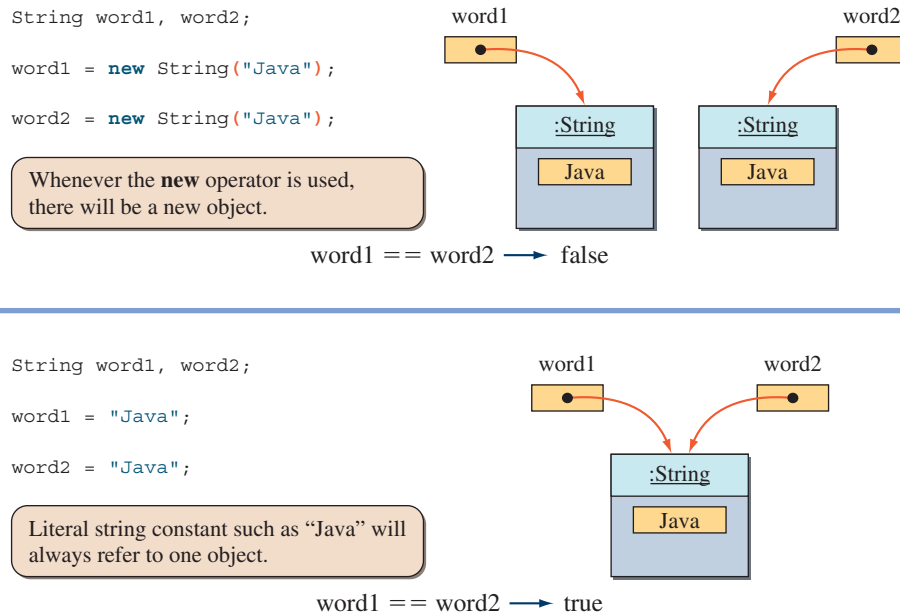


Figure 5.6 Difference between using and not using the `new` operator for `String`.



1. Determine the output of the following code.

```
String str1 = "Java";
String str2 = "Java";

boolean result1 = str1 == str2;
boolean result2 = str1.equals(str2);

System.out.println(result1);
System.out.println(result2);
```

2. Determine the output of the following code.

```
String str1 = new String("latte");
String str2 = new String("LATTE");

boolean result1 = str1 == str2;
boolean result2 = str1.equals(str2);

System.out.println(result1);
System.out.println(result2);
```

3. Show the state of memory after the following statements are executed.

```
String str1, str2, str3;
str1 = "Jasmine";
str2 = "Oolong";
str3 = str2;
str2 = str1;
```

5.5 The switch Statement

switch
statement

Another Java statement that implements a selection control flow is the *switch statement*. Suppose we want to direct the students to the designated location for them to register for classes. The location where they register is determined by their grade level. The user enters 1 for freshman, 2 for sophomore, 3 for junior, and 4 for senior. Using the switch statement, we can write the code as

```
int gradeLevel;

Scanner scanner = new Scanner(System.in);
System.out.print("Grade (Frosh-1,Soph-2,...): ");
gradeLevel = scanner.nextInt();

switch (gradeLevel) {

    case 1: System.out.println("Go to the Gymnasium");
            break;

    case 2: System.out.println("Go to the Science Auditorium");
            break;
```



```

    case 3: System.out.println("Go to Halligan Hall Rm 104");
            break;

    case 4: System.out.println("Go to Root Hall Rm 101");
            break;
}

```

The syntax for the `switch` statement is

switch
statement
syntax

```

switch ( <integer expression> ) {
    <case label 1> : <case body 1>
    ...
    <case label n> : <case body n>
}

```

Figure 5.7 illustrates the correspondence between the `switch` statement we wrote and the general format.

The `<case label i>` has the form

default
reserved word

```

case <integer constant> or default

```

and `<case body i>` is a sequence of zero or more statements. Notice that `<case body i>` is not surrounded by left and right braces. The `<constant>` can be either a named or literal constant.

The data type of `<arithmetic expression>` must be `char`, `byte`, `short`, or `int`. (*Note:* We will cover the data type `char` in Chap. 9.) The value of `<arithmetic expression>` is compared against the constant value `i` of `<case label i>`. If there is a

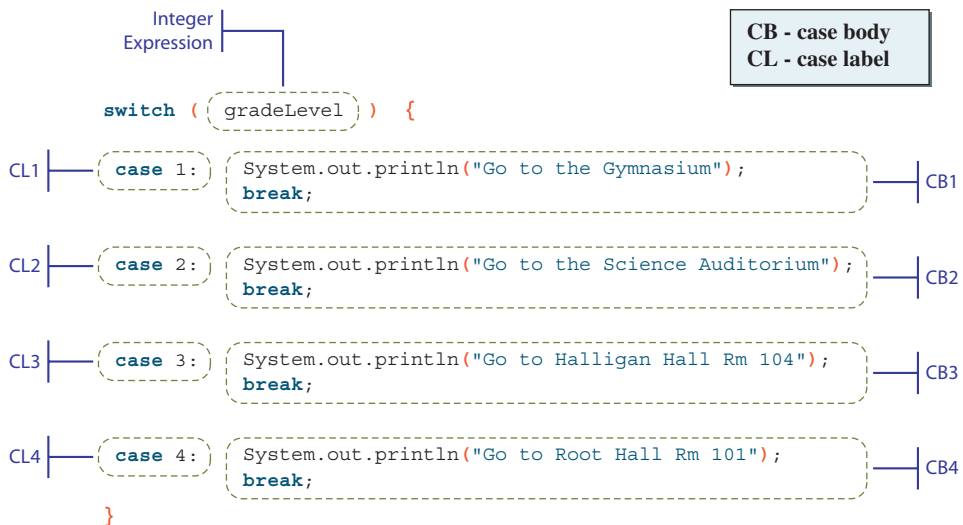


Figure 5.7 Mapping of the sample `switch` statement to the general format.

matching case, then its case body is executed. If there is no matching case, then the execution continues to the statement that follows the switch statement. No two cases are allowed to have the same value for <constant>, and the cases can be listed in any order.

break statement

Notice that each case in the sample switch statement is terminated with the **break statement**. The break statement causes execution to continue from the statement following this switch statement, skipping the remaining portion of the switch statement. The following example illustrates how the break statement works:

```
//Assume necessary declaration and object creation are done
selection = 1;

switch (selection) {
    case 0: System.out.println(0);
    case 1: System.out.println(1);
    case 2: System.out.println(2);
    case 3: System.out.println(3);
}
```

When this code is executed, the output is

```
1
2
3
```

because after the statement in case 1 is executed, statements in the remaining cases will be executed also. To execute statements in one and only one case, we need to include the break statement at the end of each case, as we have done in the first example. Figure 5.8 shows the effect of the break statement.

The break statement is not necessary in the last case, but for consistency we place it in every case. Also, by doing so we don't have to remember to include the break statement in the last case when we add more cases to the end of the switch statement.

Individual cases do not have to include a statement, so we can write something like this:

```
Scanner scanner = new Scanner(System.in);
System.out.print("Input: ");
int ranking = scanner.nextInt();
switch (ranking) {
    case 10:
    case 9:
    case 8: System.out.println("Master");
            break;
    case 7:
    case 6: System.out.println("Journeyman");
            break;
```

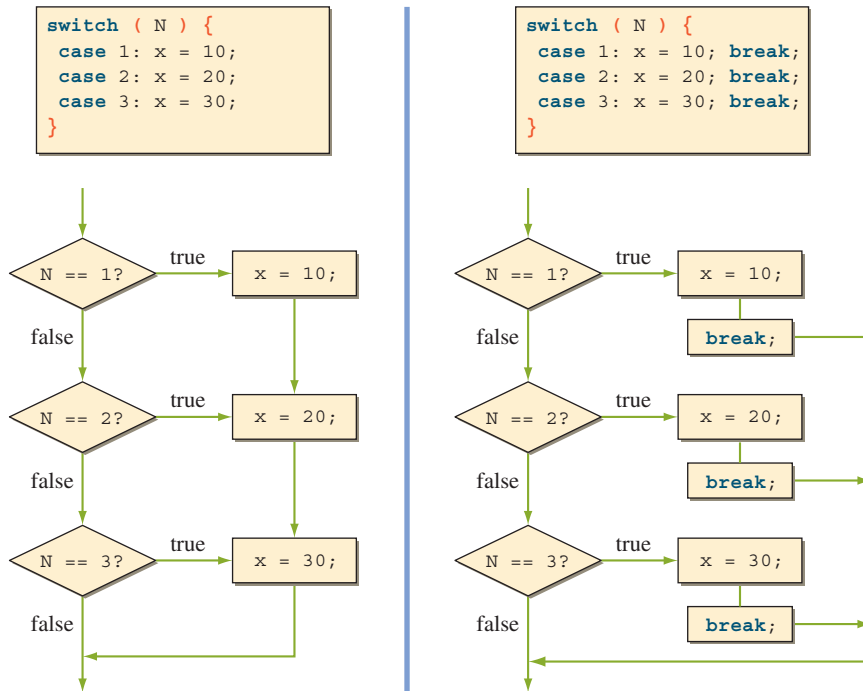


Figure 5.8 A diagram showing the control flow of the `switch` statement with and without the `break` statements.

```

    case 5:
    case 4: System.out.println("Apprentice");
           break;
}

```

The code will print Master if the value of ranking is 10, 9, or 8; Journeyman if the value of ranking is either 7 or 6; or Apprentice if the value of ranking is either 5 or 4.

We may include a default case that will always be executed if there is no matching case. For example, we can add a default case to print out an error message if any invalid value for ranking is entered.

```

switch (ranking) {

    case 10:
    case 9:
    case 8: System.out.println("Master");
           break;

    case 7:
    case 6: System.out.println("Journeyman");
           break;

    case 5:

```

```

        case 4: System.out.println("Apprentice");
                break;

        default: System.out.println("Error: Invalid Data");
                break;
    }

```

There can be at most one default case. Since the execution continues to the next statement if there is no matching case (and no default case is specified), it is safer to always include a default case. By placing some kind of output statement in the default case, we can detect an unexpected switch value. Such a style of programming is characterized as *defensive programming*. Although the default case does not have to be placed as the last case, we recommend you do so, in order to make the switch statement more readable.

defensive
programming



1. What's wrong with the following switch statement?

```

switch ( N ) {
    case 0:
    case 1: x = 11;
            break;

    default: System.out.println("Switch Error");
            break;

    case 2: x = 22;
            break;

    case 1: x = 33;
            break;
}

```

2. What's wrong with the following switch statement?

```

switch ( ranking ) {
    case >4.55: pay = pay * 0.20;
                break;

    case =4.55: pay = pay * 0.15;
                break;

    default:    pay = pay * 0.05;
                break;
}

```

5.6 | Drawing Graphics

We introduce four standard classes related to drawing geometric shapes on a window. These four standard classes will be used in Section 5.7 on the sample development. We describe their core features here. More details can be found in the online Java API documentation.

java.awt.Graphics

java.awt.
Graphics

We can draw geometric shapes on a frame window by calling appropriate methods of the Graphics object. For example, if g is a Graphics object, then we can write

```
g.drawRect(50, 50, 100, 30);
```

to display a rectangle 100 pixels wide and 30 pixels high at the specified position (50, 50). The position is determined as illustrated in Figure 5.9. The complete program is shown below. The top left corner, just below the window title bar, is position (0, 0), and the x value increases from left to right and the y value increases from top to bottom. Notice that the direction in which the y value increases is opposite to the normal two-dimensional graph.

content pane
of a frame

The area of a frame which we can draw is called the *content pane of a frame*. The content pane excludes the area of a frame that excludes the regions such as the border, scroll bars, the title bar, the menu bar, and others. To draw on the content pane of a frame window, first we must get the content pane's Graphic object. Then we call this Graphics method to draw geometric shapes. Here's a sample:

```

/*
   Chapter 5 Sample Program: Draw a rectangle on a frame
   window's content pane

   File: Ch5SampleGraphics.java
*/
import javax.swing.*; //for JFrame
import java.awt.*; //for Graphics and Container

class Ch5SampleGraphics {

    public static void main( String[] args ) {

        JFrame win;
        Container contentPane;
        Graphics g;

        win = new JFrame("My First Rectangle");
        win.setSize(300, 200);
        win.setLocation(100,100);
        win.setVisible(true);

        contentPane = win.getContentPane();
        g = contentPane.getGraphics();
        g.drawRect(50,50,100,30);

    }
}

```

win must be visible on the screen before you get its content pane.

Syntax
 A rectangle <width>
 wide and <height>
 high is displayed at
 position (<x>, <y>).

```
graphic.drawRect ( <x>, <y>, <width>, <height> );
```

Example: `graphic.drawRect (50, 50, 100, 30);`

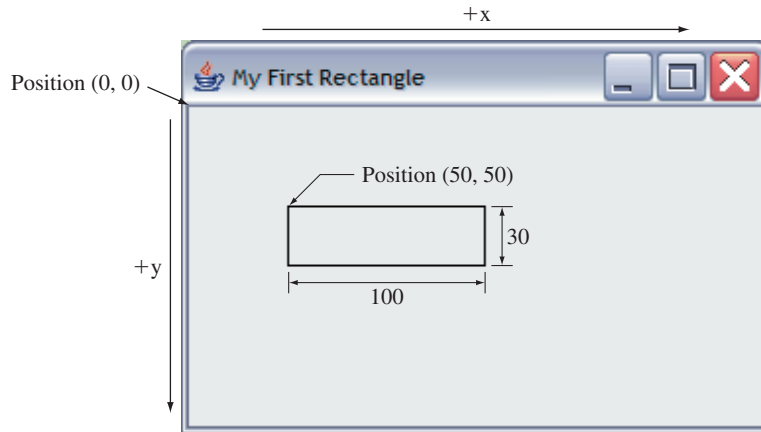


Figure 5.9 The diagram illustrates how the position of the rectangle is determined by the `drawRect` method.

Here are the key points to remember in drawing geometric shapes on the content pane of a frame window.

Things to Remember



To draw geometric shapes on the content pane of a frame window:

1. The content pane is declared as a **Container**, for example,
`Container contentPane;`
2. The frame window must be visible on the screen before we can get its content pane and the content pane's **Graphics** object.

Hints, & Tips, Pitfalls



Depending on the speed of your PC, you may have to include the following **try** statement

```
try {Thread.sleep(200);} catch (Exception e) {}
```

to put a delay before drawing the rectangle. Place this **try** statement before the last statement. The argument in the **sleep** method specifies the amount of delay in milliseconds (1000 ms = 1 s). If you still do not see a rectangle drawn in the window after including the delay statement, increase the amount of delay until you see the rectangle drawn. We will describe the **try** statement in Chapter 8.



If there is a window that covers the area in which the drawing takes place or the drawing window is minimized and restored to its normal size, the drawn shape (or portion of it, in the case of the overlapping windows) gets erased. The **DrawingBoard** class used in the sample development (Sec. 5.7) eliminates this problem. For information on the technique to avoid the disappearance of the drawn shape, please check our website at www.drcaffeine.com

Table 5.4 lists some of the available graphic drawing methods.

Table 5.4 A partial list of drawing methods defined for the `Graphics` class

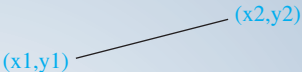
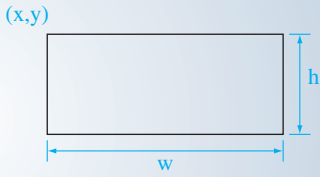
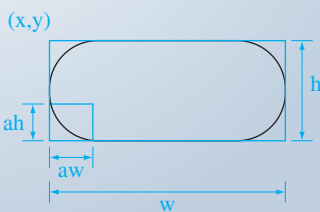
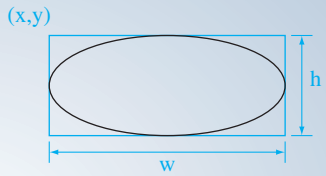

Method	Meaning
<code>drawLine(x1,y1,x2,y2)</code>	<p>Draws a line between $(x1, y1)$ and $(x2, y2)$.</p> 
<code>drawRect(x,y,w,h)</code>	<p>Draws a rectangle with width w and height h at (x, y).</p> 
<code>drawRoundRect(x,y,w,h,aw,ah)</code>	<p>Draws a rounded-corner rectangle with width w and height h at (x, y). Parameters aw and ah determine the angle for the rounded corners.</p> 

Table 5.4 A partial list of drawing methods defined for the `Graphics` class (Continued)

Method	Meaning
<code>drawOval(x, y, w, h)</code>	<p>Draws an oval with width <code>w</code> and height <code>h</code> at <code>(x, y)</code>.</p> 
<code>drawString("text", x, y)</code>	<p>Draws the string <code>text</code> at <code>(x, y)</code>.</p> 
<code>fillRect(x, y, w, h)</code>	Same as the <code>drawRect</code> method but fills the region with the currently set color.
<code>fillRoundRect(x, y, w, h, aw, ah)</code>	Same as the <code>drawRoundRect</code> method but fills the region with the currently set color.
<code>fillOval(x, y, w, h)</code>	Same as the <code>drawOval</code> method but fills the region with the currently set color.

Notice the distinction between the draw and fill methods. The draw method will draw the boundary only, while the fill method fills the designated area with the currently selected color. Figure 5.10 illustrates the difference.

java.awt.Color

java.awt.Color

To designate the color for drawing, we will use the `Color` class from the standard `java.awt` package. A `Color` object uses a coloring scheme called the *RGB scheme*, which specifies a color by combining three values, ranging from 0 to 255, for red, green, and blue. For example, the color black is expressed by setting red, green, and blue to 0, and the color white by setting all three values to 255. We create, for example, a `Color` object for the pink color by executing

```
Color pinkColor;
pinkColor = new Color(255,175,175);
```

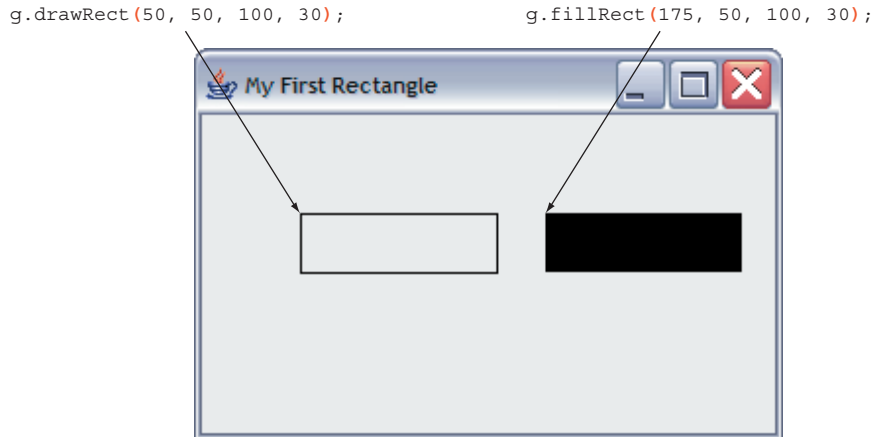



Figure 5.10 The diagram illustrates the distinction between the **draw** and **fill** methods. We assume the currently selected color is black (default).

Instead of dealing with the three numerical values, we can use the public class constants defined in the `Color` class. The class constants for common colors are

```
Color.BLACK           Color.MAGENTA
Color.BLUE            Color.ORANGE
Color.CYAN            Color.PINK
Color.DARK_GRAY       Color.RED
Color.GRAY             Color.WHITE
Color.GREEN           Color.YELLOW
Color.LIGHT_GRAY
```

The class constants in lowercase letters are also defined (such as `Color.black`, `Color.blue`, and so forth). In the older versions of Java, only the constants in lowercase letters were defined. But the Java convention is to name constants using only the uppercase letters, so the uppercase color constants are added to the class definition.

Each of the above is a `Color` object with its RGB values correctly set up. We will pass a `Color` object as an argument to the `setColor` method of the `Graphics` class to change the color. To draw a blue rectangle, for example, we write

```
//Assume g is set correctly
g.setColor(Color.BLUE);
g.drawRect(50, 50, 100, 30);
```

We can also change the background color of a content pane by using the `setBackground` method of `Container` as

```
contentPane.setBackground(Color.LIGHT_GRAY);
```

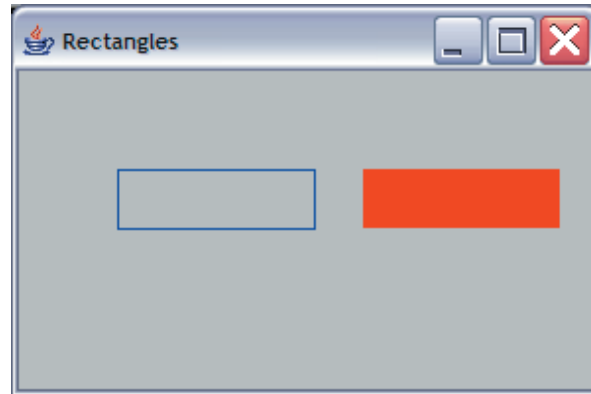


Figure 5.11 A frame with a white background content pane and two rectangles.

Running the following program will result in the frame shown in Figure 5.11.

```

/*
   Chapter 5 Sample Program: Draw one blue rectangle and
                           one filled red rectangle on light gray
                           background content pane

   File: Ch5SampleGraphics2.java
*/

import javax.swing.*;
import java.awt.*;

class Ch5SampleGraphics2 {

    public static void main( String[] args ) {

        JFrame    win;
        Container contentPane;
        Graphics  g;

        win = new JFrame("Rectangles");
        win.setSize(300, 200);
        win.setLocation(100,100);
        win.setVisible(true);

        contentPane = win.getContentPane();
        contentPane.setBackground(Color.LIGHT_GRAY);

        g = contentPane.getGraphics();
        g.setColor(Color.BLUE);
        g.drawRect(50,50,100,30);
    }
}

```

```

g.setColor(Color.RED);
g.fillRect(175,50,100,30);
    }
}

```

java.awt.Point

A `Point` object is used to represent a point in two-dimensional space. It contains x and y values, and we can access these values via its public data member `x` and `y`. Here's an example to assign a position (10, 20):

```

Point pt = new Point();
pt.x = 10;
pt.y = 20;

```

It is also possible to set the position at the creation time as follows:

```

Point pt = new Point(10, 20);

```

java.awt.Dimension

In manipulating shapes, such as moving them around a frame's content pane, the concept of the bounding rectangle becomes important. A *bounding rectangle* is a rectangle that completely surrounds the shape. Figure 5.12 shows some examples of bounding rectangles.

Just as the (x, y) values are stored as a single `Point` object, we can store the width and height of a bounding rectangle as a single `Dimension` object. The `Dimension` class has the two public data members `width` and `height` to maintain the width and height of a bounding rectangle. Here's an example to create a 40 pixels by 70 pixels high bounding rectangle:

```

Dimension dim = new Dimension();
dim.width = 40;
dim.height = 70;

```

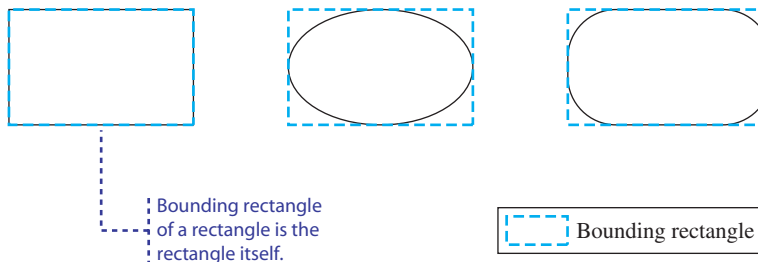


Figure 5.12 Bounding rectangles of various shapes.

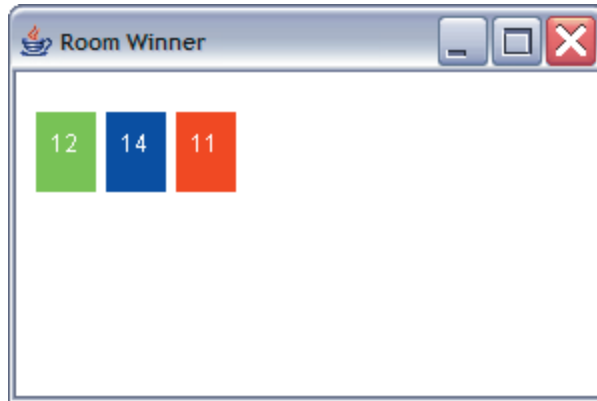


Figure 5.13 Sample output of `Ch5RoomWinner` program.

It is also possible to set the values at the creation time as follows:

```
Dimension dim = new Dimension(40, 70);
```

Let's apply the drawing techniques to an early sample program. In Chapter 4, we wrote the `RoomWinner` program that randomly selects and displays the dorm room lottery cards. The display was only in text, something like this:

```
Winning Card Combination:
1 - red; 2 - green; 3 - blue

           color   number
Card 1:    2       13
Card 2:    2       12
Card 3:    1       14
```

We will make a graphical version of the program. Figure 5.13 shows a sample output.

Here's the main class `Ch5RoomWinner`, which has a structure similar to the one for `Ch5SampleGraphics2`.

```
import java.awt.*;
import javax.swing.*;

class Ch5RoomWinner {
    public static void main( String[] args ) {
        JFrame win;
        Container contentPane;
        Graphics g;
```

```

GraphicLotteryCard one, two, three;

win = new JFrame("Room Winner");
win.setSize(300, 200);
win.setLocation(100,100);
win.setVisible(true);

contentPane = win.getContentPane();
contentPane.setBackground(Color.WHITE);

g = contentPane.getGraphics();

one = new GraphicLotteryCard( );
two = new GraphicLotteryCard( );
three = new GraphicLotteryCard( );

one.spin();
two.spin();
three.spin();

one.draw(g, 10, 20);
two.draw(g, 50, 20);
three.draw(g, 90, 20);
}
}

```

These objects will draw themselves on g at the specified positions.

We modify the `LotteryCard` class from Chapter 4 by adding code that will draw a card on a given `Graphics` context. The name of the new class is `GraphicLotteryCard`. Here's the class definition (we list only the portions that are new):

```

import java.awt.*;

class GraphicLotteryCard {

    // Data Members

    //width of this card for drawing
    public static final int WIDTH = 30;

    //height of this card for drawing
    public static final int HEIGHT = 40;

    //the other data members and methods are the same as before

    public void draw(Graphics g, int xOrigin, int yOrigin) {

        switch (color) {
            case 1: g.setColor(Color.RED);
                    break;

```

```

        case 2: g.setColor(Color.GREEN);
                break;

        case 3: g.setColor(Color.BLUE);
                break;
    }

    g.fillRect(xOrigin, yOrigin, WIDTH, HEIGHT);
    g.setColor(Color.WHITE); //draw text in white
    g.drawString( "" + number", xOrigin + WIDTH/4, yOrigin + HEIGHT/2);
}
}

```

This is a quick way to convert a numerical value to **String**

Notice that the statements in `Ch5RoomWinner`

```

one.draw(g, 10, 20);
two.draw(g, 50, 20);
three.draw(g, 90, 20);

```

are not as flexible as they can be. If the values for the constant `WIDTH` and `HEIGHT` in the `GraphicLotteryCard` class are changed, these three statements could result in drawing the card inadequately (such as overlapping cards). The two constants are declared `public` for a reason. Using the `WIDTH` constant, for example, we can rewrite the three statements as

```

int cardWidth = GraphicLotteryCard.WIDTH;
one.draw(g, 10, 20);
two.draw(g, 10 + cardWidth + 5, 20);
three.draw(g, 10 + 2*(cardWidth+ 5), 20);

```

The statements will draw cards with a 5-pixel interval between cards. This code will continue to work correctly even after the value of `WIDTH` is modified.

5.7 | Enumerated Constants

We learned in Section 3.3 how to define numerical constants and the benefits of using them in writing readable programs. In this section, we will introduce an additional type of constant called *enumerated constants* that were added to the Java language from Version 5.0. Let's start with an example. Suppose we want to define a `Student` class and define constants to distinguish four undergraduate grade

levels—freshman, sophomore, junior, and senior. Using the numerical constants, we can define the grade levels as such:

```
class Student {
    public static final int FRESHMAN = 0;
    public static final int SOPHOMORE = 1;
    public static final int JUNIOR = 2;
    public static final int SENIOR = 3;
    ...
}
```

With the new enumerated constants, this is how we can define the grade levels in the Student class:

```
class Student {
    public static enum GradeLevel
        {FRESHMAN, SOPHOMORE, JUNIOR, SENIOR}
    ...
}
```

The word `enum` is a new reserved word, and the basic syntax for defining enumerated constants is

```
enum <enumerated type> { <constant values> }
```

where `<enumerated type>` is an identifier and `<constant values>` is a list of identifiers separated by commas. Notice that for the most common usage of enumerated constants, we append the modifiers `public` and `static`; but they are not a required part of defining enumerated constants. Here are more examples:

```
enum Month {JANUARY, FEBRUARY, MARCH, APRIL,
            MAY, JUNE, JULY, AUGUST,
            SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER}
```

```
enum Gender {MALE, FEMALE}
```

```
enum SkillLevel {NOVICE, INTERMEDIATE, ADVANCED, EXPERT}
```

One restriction when declaring an enumerated type is that it cannot be a local declaration. In other words, we must declare it outside of any method, just as for the other data members of a class.

Unlike numerical constants, which are simply identifiers with fixed numerical values, enumerated constants do not have any assigned numerical values. They are said to belong to, or be members of, the associated *enumerated type*. For example, two enumerated constants `MALE` and `FEMALE` belong to the enumerated type `Gender`. (*Note:* We keep the discussion of the enumerated type to its simplest form here. It is beyond the scope of an introductory programming textbook to discuss Java's enumerated type in full detail.)

enumerated
type

Just as with any other data types, we can declare variables of an enumerated type and assign values to them. Here is an example (for the sake of brevity, we list the enum declaration and its usage together, but remember that the declaration in the actual use cannot be a local declaration):

```
enum Fruit {APPLE, ORANGE, BANANA}

Fruit f1, f2, f3;

f1 = Fruit.APPLE;
f2 = Fruit.BANANA;
f3 = f1;
```

NOTE: The constant value is prefixed by the name of the enumerated type.

type safety

Because variables `f1`, `f2`, and `f3` are declared to be of the type `Fruit`, we can only assign one of the associated enumerated constants to them. This restriction supports a desired feature called *type safety*. So what is the big deal? Consider the following numerical constants and assignment statements:

```
final int APPLE = 1;
final int ORANGE = 2;
final int BANANA = 3;

int fOne, fTwo, fThree;

fOne = APPLE;
fTwo = ORANGE;
fThree = fOne;
```

The code may look comparable to the one that uses enumerated constants, but what will happen if we write the following?

```
fOne = 45;
```

The assignment is logically wrong. It does not make any sense to assign meaningless value such as 45 to the variable `fOne` if it is supposed to represent one of the defined fruit. However, no compiler error will result because the data type of `fOne` is `int`. The statement may or may not cause the runtime error depending on how the variable `fOne` is used in the rest of the program. In either case, the program cannot be expected to produce a correct result because of the logical error.

By defining an enumerated type, a variable of that type can only accept the associated enumerated constants. Any violation will be detected by the compiler. This will eliminate the possibility of assigning a nonsensical value as seen in the case for the numerical constants. Type safety means that we can assign only meaningful values to a declared variable.

Another benefit of the enumerated type is the informative output values. Assuming the variables `fTwo` and `f2` retain the values assigned to them in the sample code, the statement

```
System.out.println("Favorite fruit is " + fTwo);
```


will produce a cryptic output:

```
Favorite fruit is 2
```

In contrast, the statement

```
System.out.println("Favorite fruit is " + f2);
```

will produce a more informative output:

```
Favorite fruit is BANANA
```

We will describe other advantages of using enumerated types later in the book.

As shown, when referring to an enumerated constant in the code, we must prefix it with its enumerated type name, for example,

```
Fruit f = Fruit.APPLE;
. . .
if (f == Fruit.ORANGE) {
    System.out.println("I like orange, too.");
}
. . .
```

A case label for a switch statement is the only exception to this rule. Instead of writing, for example,

```
Fruit fruit;
fruit = ... ;
switch (fruit) {
    case Fruit.APPLE: . . . ;
                        break;
    case Fruit.ORANGE: . . . ;
                        break;
    case Fruit.BANANA: . . . ;
                        break;
}
```

we can specify the case labels without the prefix as in

```
Fruit fruit;
fruit = ... ;
switch (fruit) {
    case APPLE: . . . ;
                break;
}
```

```

        case ORANGE: . . . ;
            break;

        case BANANA: . . . ;
            break;
    }

```

Hints, & Tips, Pitfalls



It is not necessary to prefix the enumerated constant with its enumerated type name when it is used as a **case** label in a **switch** statement.

The enumerated type supports a useful method named `valueOf`. The method accepts one `String` argument and returns the enumerated constant whose value matches the given argument. For example, the following statement assigns the enumerated constant `APPLE` to the variable `fruit`:

```
Fruit fruit = Fruit.valueOf("APPLE");
```

In which situations could the `valueOf` method be useful? One is the input routine. Consider the following:

```

Scanner scanner = new Scanner(System.in);
System.out.print("Enter your favorite fruit " +
                "(APPLE, ORANGE, BANANA): ");
String fruitName = scanner.next( );
Fruit favoriteFruit = Fruit.valueOf(fruitName);

```

Be aware, however, that if you pass a `String` value that does not match any of the defined constants, it will result in a runtime error. This means if the user enters `Orange`, for example, it will result in an error (the input has to be all capital letters to match). We will discuss how to handle such runtime errors without causing the program to terminate abruptly in Chapter 8.

To access the enumerated constants in a programmer-defined class from outside the class, we must reference them through the associated enumerated type (assuming, of course, the visibility modifier is `public`). Consider the following `Faculty` class:

```

class Faculty {
    public static enum Rank
        {LECTURER, ASSISTANT, ASSOCIATE, FULL}
    private Rank rank;
    . . .
}

```

```

public void setRank(Rank r) {
    rank = r;
}

public Rank getRank() {
    return rank;
}

. . .
}

```

Notice how the enumerate type `Rank` is used in the `setRank` and `getRank` methods. It is treated just as any other types are. To access the `Rank` constants from outside of the `Faculty` class, we write

```

Faculty.Rank.ASSISTANT

Faculty.Rank.FULL

```

and so forth. Here's an example that assigns the rank of `ASSISTANT` to a `Faculty` object:

```

Faculty prof = new Faculty(...);
prof.setRank(Faculty.Rank.ASSISTANT);

```

And here's an example to retrieve the rank of a `Faculty` object:

```

Faculty teacher;

//assume 'teacher' is properly created

Faculty.Rank rank;

rank = teacher.getRank();

```

Quick CHECK

1. Define an enumerated type `Day` that includes the constants `SUNDAY` through `SATURDAY`.
2. What is the method that returns an enumerated constant, given the matching `String` value?
3. Detect the error(s) in the following code:

```

enum Fruit {APPLE, ORANGE, BANANA}

Fruit f1, f2;
int f3;

f1 = 1;
f2 = ORANGE;
f3 = f1;
f1 = "BANANA";

```

5.8 Sample Development

Drawing Shapes

When a certain period of time passes without any activity on a computer, a screensaver becomes active and draws different types of geometric patterns or textual messages. In this section we will develop an application that simulates a screensaver. We will learn a development skill very commonly used in object-oriented programming. Whether we develop alone or as a project team member, we often find ourselves writing a class that needs to behave in a specific way so that it works correctly with other classes. The other classes may come from standard Java packages or could be developed by the other team members.

In this particular case, we use the **DrawingBoard** class (written by the author). This is a helper class that takes care of programming aspects we have not yet mastered, such as moving multiple geometric shapes in a smooth motion across the screen. It is not an issue of whether we can develop this class by ourselves, because no matter how good we become as programmers, we would rarely develop an application completely on our own.

We already used many predefined classes from the Java standard libraries, but the way we will use the predefined class here is different. When we developed programs before, the classes we wrote called the methods of predefined classes. Our main method creating a **GregorianCalendar** object and calling its methods is one example. Here, for us to use a predefined class, we must define another class that provides necessary services to this predefined class. Figure 5.14 differentiates the two types of predefined classes. The first type does not place any restriction other than calling the methods correctly, while the second type requires us to implement helper classes in a specific manner to support it.

In our case, the predefined class **DrawingBoard** will require another class named **DrawableShape** that will assume the responsibility of drawing individual geometric shapes. So, to use the **DrawingBoard** class in our program, we must implement the class named **DrawableShape**. And we must implement the **DrawableShape** class in a specific way. The use of the **DrawingBoard** class dictates that we define a set of fixed methods in the **DrawableShape** class. We can add more, but we must at the minimum provide the specified set of fixed methods because the **DrawingBoard** class will need to call these methods. The methods are “fixed” in the method signature—method name, the number of parameters and their types, and return type—but the method body can be defined in any way we like. This is how the flexibility is achieved. For example, the **DrawableShape** class we define must include a method named **draw** with the dictated signature. But it’s up to us to decide what we put in the method body. So we can choose, for example, to implement the method to draw a circle, rectangle, or any other geometric shape of our choosing.

As always, we will develop this program following incremental development steps. The incremental development steps we will take here are slightly different in character from those we have seen so far. In the previous incremental developments, we knew all the ingredients, so to speak. Here we have to include a step to explore the **DrawingBoard** class. We will find out shortly that to use the **DrawingBoard** class, we will have to deal with some Java standard classes we have not seen yet. Pedagogically, a textbook may try to

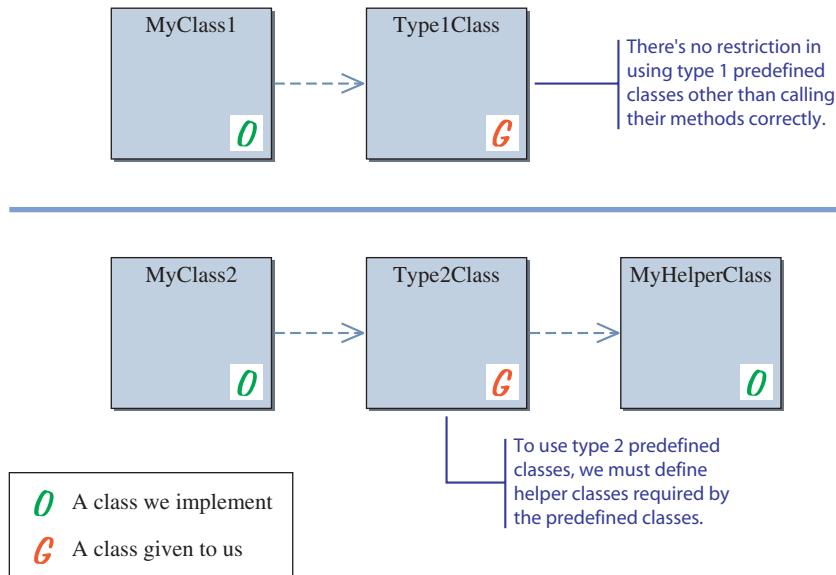


Figure 5.14 Two types of predefined classes. The first type does not require us to do anything more than use the predefined classes by calling their methods. The second type requires us to define helper classes for the predefined classes we want to use.

explain beforehand everything that is necessary to understand the sample programs. But no textbook can explain everything. When we develop programs, there will always be a time when we encounter some unknown classes. We need to learn how to deal with such a situation in our development steps.

Problem Statement

Write an application that simulates a screensaver by drawing various geometric shapes in different colors. The user has the option of choosing a type (ellipse or rectangle), color, and movement (stationary, smooth, or random).

Overall Plan

We will begin with our overall plan for the development. Let's begin with the outline of program logic. We first let the user select the shape, its movement, and its color, and we then start drawing. We express the program flow as having four tasks:

program
tasks

1. Get the shape the user wants to draw.
2. Get the color of the chosen shape.
3. Get the type of movement the user wants to use.
4. Start the drawing.

5.8 Sample Development—continued

Let's look at each task and determine an object that will be responsible for handling the task. For the first three tasks, we can use our old friend **Scanner**. We will get into the details of exactly how we ask the users to input those values in the later incremental steps. For the last task of actually drawing the selected shape, we need to define our own class. The task is too specific to the program, and there is no suitable object in the standard packages that does the job. As discussed earlier, we will use a given predefined class **DrawingBoard** and define the required helper class **DrawableShape**.

We will define a top-level control object that manages all these objects. We will call this class **Ch5DrawShape**. As explained in Section 4.10, we will make this control object the main class. Here's our working design document:

program
classes

Design Document: Ch5DrawShape	
Class	Purpose
Ch5DrawShape	The top-level control object that manages other objects in the program. This is the main class, as explained in Section 4.10.
DrawingBoard	The given predefined class that handles the movement of <code>DrawableShape</code> objects.
DrawableShape	The class for handling the drawing of individual shapes.
Scanner	The standard class for handling input routines.

Figure 5.15 is the program diagram for this program.

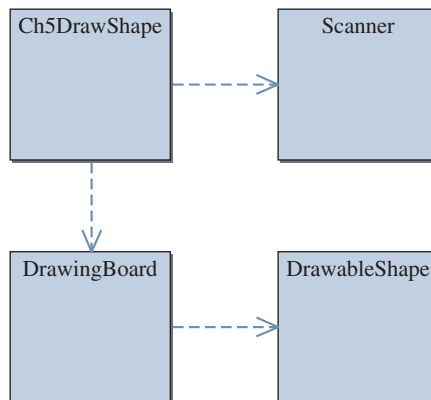


Figure 5.15 The program diagram for the **Ch5DrawShape** program.

development
steps

We will implement this program in the following six major steps:

1. Start with a program skeleton. Explore the **DrawingBoard** class.
2. Define an experimental **DrawableShape** class that draws a dummy shape.
3. Add code to allow the user to select a shape. Extend the **DrawableShape** and other classes as necessary.
4. Add code to allow the user to specify the color. Extend the **DrawableShape** and other classes as necessary.
5. Add code to allow the user to specify the motion type. Extend the **DrawableShape** and other classes as necessary.
6. Finalize the code by tying up loose ends.

Our first task is to find out about the given class. We could have designed the input routines first, but without knowing much about the given class, it would be difficult to design suitable input routines. When we use an unknown class, it is most appropriate to find out more about this given class before we plan any input or output routines. Just as the development steps are incremental, our exploration of the given class will be incremental. Instead of trying to find out everything about the class at once, we begin with the basic features and skeleton code. As we learn more about the given class incrementally, we extend our code correspondingly.

Step 1 Development: Program Skeleton

step 1
design

We begin the development with the skeleton main class. The main purpose in step 1 is to use the **DrawingBoard** class in the simplest manner to establish the launch pad for the development. To do so, we must first learn a bit about the given **DrawingBoard** class. Here's a brief description of the **DrawingBoard** class. In a real-world situation, we would be finding out about the given class by reading its accompanying documentation or some other external sources. The documentation may come in form of online javadoc documents or reference manuals.

DrawingBoard

An instance of this class will support the drawing of `DrawableShape` objects. Shapes can be drawn at fixed stationary positions, at random positions, or in a smooth motion at the specified speed. The actual drawing of the individual shapes is done inside the `DrawableShape` class. The client programmer decides which shape to draw.

```
public void addShape ( DrawableShape shape )
```

 Adds shape to this `DrawingBoard` object. You can add an unlimited number of `DrawableShape` objects.

(Continued)

5.8 Sample Development—continued

DrawingBoard (Continued)

```
public void setBackground( java.awt.Color color )
    Sets the background color of this DrawingBoard object to the designated
    color. The default background color is black.
```

```
public void setDelayTime( double delay )
    Sets the delay time between drawings to delay seconds. The smaller the delay
    time, the faster the shapes move. If the movement type is other than SMOOTH,
    then setting the delay time has no visual effect.
```

```
public void setMovement( Movement type )
    Sets the movement type to type. Class constants for three types of motion are
    Movement.STATIONARY—draw shapes at fixed positions,
    Movement.RANDOM—draw shapes at random positions, and
    Movement.SMOOTH—draw shapes in a smooth motion.
```

```
public void setVisible( boolean state )
    Makes this DrawingBoard object appear on or disappear from the screen
    if state is true or false, respectively. To simulate the screensaver,
    setting it visible will cause a maximized window to appear on the screen.
```

```
public void start( )
    Starts the drawing. If the window is not visible yet, it will be made visible before
    the drawing begins.
```

Among the defined methods, we see the **setVisible** method is the one to make it appear on the screen. All other methods pertain to adding **DrawableShape** objects and setting the properties of a **DrawingBoard** object. We will explain the standard **java.awt.Color** class when we use the **setBackground** method in the later step. In this step, we will keep the code very simple by only making it appear on the screen. We will deal with other methods in the later steps.

Our working design document for the **Ch5DrawShape** class is as follows:

Design Document: The Ch5DrawShape Class		
Method	Visibility	Purpose
<constructor>	Public	Creates a DrawingBoard object.
main	Public	This is main method of the class.
start	Public	Starts the program by opening a DrawingBoard object.

step 1 code

Since this is a skeleton code, it is very basic. Here's the code:

```

/*
    Chapter 5 Sample Development: Drawing Shapes (Step 1)

    The main class of the program.
*/
class Ch5DrawShape {
    private DrawingBoard canvas;

    public Ch5DrawShape( ) {
        canvas = new DrawingBoard( );
    }

    public void start( ) {
        canvas.setVisible(true);
    }

    public static void main(String[] args) {
        Ch5DrawShape screensaver = new Ch5DrawShape( );
        screensaver.start();
    }
}

```

step 1 test

The purpose of step 1 testing is to verify that a **DrawingBoard** object appears correctly on the screen. Since this is our first encounter with the **DrawingBoard** class, it is probable that we are not understanding its documentation fully and completely. We need to verify this in this step. When a maximized window with the black background appears on the screen, we know the main class was executed properly. After we verify the correct execution of the step 1 program, we will proceed to implement additional methods of **Ch5DrawShape** and gradually build up the required **DrawableShape** class.

Step 2 Development: Draw a Shape

step 2 design

In the second development step, we will implement a preliminary **DrawableShape** class and make some shapes appear on a **DrawingBoard** window. To draw shapes, we need to add them to a **DrawingBoard** window. And to do so, we need to define the

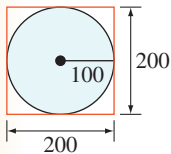
5.8 Sample Development—continued

DrawableShape class with the specified set of methods. Here are the required methods and a brief description of what to accomplish in them:

Required Methods of <code>DrawableShape</code>	
<code>public void draw(java.awt.Graphics)</code>	Draw a geometric shape on the <code>java.awt.Graphics</code> . The <code>DrawingBoard</code> window calls the <code>draw</code> method of <code>DrawableShape</code> objects added to it.
<code>public java.awt.Point getCenterPoint()</code>	Return the center point of this shape.
<code>public java.awt.Dimension getDimension()</code>	Return the bounding rectangle of this shape as a <code>Dimension</code> .
<code>public void setCenterPoint(java.awt.Point)</code>	Set the center point of this shape. The <code>DrawingBoard</code> window calls the <code>setCenterPoint</code> method of <code>DrawableShape</code> objects to update their positions in the <code>SMOOTH</code> movement type.

At this stage, the main task is for us to confirm our understanding of the requirements in implementing the **DrawableShape** class. Once we get this confirmation, we can get into the details of the full-blown **DrawableShape** class.

To keep the preliminary class simple, we draw three filled circles of a fixed size and color. The **DrawableShape** class includes a single data member **centerPoint** to keep track of the shape's center point. If we fix the radius of the circles to 100 pixels, that is, the bounding rectangle is 200 pixels by 200 pixels, and the color to blue, then the **draw** method can be written as follows:



```
public void draw(Graphics g) {
    g.setColor(Color.blue);
    g.fillOval(centerPoint.x-100, centerPoint.y-100, 200, 200);
}
```

Since the size is fixed, we simply return a new **Dimension** object for the **getDimension** method:

```
public Dimension getDimension() {
    return new Dimension(200, 200);
}
```

For the **setCenterPoint** and **getCenterPoint** methods, we assign the passed parameter to the data member **centerPoint** and return the current value of the data member **centerPoint**, respectively.

We are now ready to modify the **Ch5DrawShape** class to draw three filled circles. We will implement this by modifying the **start** method. First we need to create three **DrawableShape** objects and add them to the **DrawingBoard** object **canvas**:

```
DrawableShape shape1 = new DrawableShape ();
DrawableShape shape2 = new DrawableShape ();
DrawableShape shape3 = new DrawableShape ();

shape1.setCenterPoint (new Point (250,300));
shape2.setCenterPoint (new Point (500,300));
shape3.setCenterPoint (new Point (750,300));

canvas.addShape (shape1);
canvas.addShape (shape2);
canvas.addShape (shape3);
```

Then we set the motion type to **SMOOTH** movement, make the window appear on the screen, and start the drawing:

```
canvas.setMovement (DrawingBoard.Movement.SMOOTH);
canvas.setVisible (true);
canvas.start ();
```

step 2 code

Here's the code for the preliminary **DrawableShape** class:

```
import java.awt.*;

/*
   Step 2: Add a preliminary DrawableShape class
   A class whose instances know how to draw themselves.
 */
class DrawableShape {
    private Point centerPoint;

    public DrawableShape ( ) {
        centerPoint = null;
    }

    public void draw(Graphics g) {
        g.setColor (Color.blue);
        g.fillOval (centerPoint.x-100, centerPoint.y-100, 200, 200);
    }
}
```

Data Members

Constructors

draw

5.8 Sample Development—continued

```

public Point getCenterPoint( ) {
    return centerPoint;
}

public Dimension getDimension( ) {
    return new Dimension(200, 200);
}

public void setCenterPoint(Point point) {
    centerPoint = point;
}
}

```

getCenterPoint

getDimension

setCenterPoint

The **Ch5DrawShape** class now has the modified **start** method as designed (the rest of the class remains the same):

```

import java.awt.*;

/*
Chapter 5 Sample Development: Start drawing shapes (Step 2)
The main class of the program.
*/

class Ch5DrawShape {
    . . .

    public void start( ) {

        DrawableShape shape1 = new DrawableShape();
        DrawableShape shape2 = new DrawableShape();
        DrawableShape shape3 = new DrawableShape();

        shape1.setCenterPoint(new Point(250,300));
        shape2.setCenterPoint(new Point(500,300));
        shape3.setCenterPoint(new Point(750,300));

        canvas.addShape(shape1);
        canvas.addShape(shape2);
        canvas.addShape(shape3);
    }
}

```

start

```

        canvas.setMovement(DrawingBoard.Movement.SMOOTH);

        canvas.setVisible(true);
        canvas.start();
    }
    . . .
}

```

step 2 test

Now we run the program and verify the three bouncing circles moving around. To test other options of the **DrawingBoard** class, we will try the other methods with different parameters:

Method	Test Parameter
setMovement	Try both <code>DrawingBoard.STATIONARY</code> and <code>DrawingBoard.RANDOM</code> .
setDelayTime	Try values ranging from 0.1 to 3.0.
setBackground	Try several different <code>Color</code> constants such as <code>Color.white</code> , <code>Color.red</code> , and <code>Color.green</code> .

We insert these testing statements before the statement

```
canvas.setVisible(true);
```

in the **start** method.

Another testing option we should try is the drawing of different geometric shapes. We can replace the drawing statement inside the **draw** method from

```
g.fillOval(centerPoint.x-100, centerPoint.y-100,
           200, 200);
```

to

```
g.fillRect(centerPoint.x-100, centerPoint.y-100,
           200, 200);
```

or

```
g.fillRoundRect(centerPoint.x-100, centerPoint.y-100,
                200, 200, 50, 50);
```

to draw a filled rectangle or a filled rounded rectangle, respectively.

5.8 Sample Development—continued

step 3
design**Step 3 Development: Allow the User to Select a Shape**

Now that we know how to interact with the **DrawingBoard** class, we can proceed to develop the user interface portion of the program. There are three categories in which the user can select an option: shape, color, and motion. We will work on the shape selection here and on the color and motion selection in the next two steps. Once we are done with this step, the next two steps are fairly straightforward because the idea is essentially the same.

Let's allow the user to select one of three shapes—ellipse, rectangle, and rounded rectangle—the shapes we know how to draw at this point. We can add more fancy shapes later. In what ways should we allow the user to input the shape? There are two possible alternatives: The first would ask the user to enter the text and spell out the shape, and the second would ask the user to enter a number that corresponds to the shape (1 for ellipse, 2 for rectangle, 3 for rounded rectangle, e.g.). Which is the better alternative?

design
alternative 1

We anticipate at least two problems with the first input style. When we need to get a user's name, for example, there's no good alternative other than asking the user to enter his or her name. But when we want the user to select one of the few available choices, it is cumbersome and too much of a burden for the user. Moreover, it is prone to mistyping.

design
alternative 2

To allow the user to make a selection quickly and easily, we can let the user select one of the available choices by entering a corresponding number. We will list the choices with numbers 1, 2, and 3 and get the user's selection as follows:

```
System.out.print("Selection: Enter the Shape number\n" +
    " 1 - Ellipse \n" +
    " 2 - Rectangle \n" +
    " 3 - Rounded Rectangle \n");

int selection = scanner.nextInt();
```

For getting the dimension of the shape, we accept the width and height values from the user. The values cannot be negative, for sure, but we also want to restrict the values to a certain range. We do not want the shape to be too small or too large. Let's set the minimum to 100 pixels and the maximum to 500 pixels. If the user enters a value outside the acceptable range, we will set the value to 100. The input routine for the width can be written as follows:

```
System.out.print("Enter the width of the shape\n" +
    "between 100 and 500 inclusive: ");

int width = scanner.nextInt();

if (width < 100 || width > 500) {
    width = 100;
}
```

The input routine for the height will work in the same manner.

For getting the **x** and **y** values of the shape's center point, we follow the pattern of getting the width and height values. We will set the acceptable range for the **x** value to 200 and 800, inclusive, and the **y** value to 100 and 600, inclusive.

Our next task is to modify the **DrawableShape** class so it will be able to draw three different geometric shapes. First we change the constructor to accept the three input values:

```
public DrawableShape(Type sType, Dimension sDim,
                    Point sCenter) {
    type          = sType;
    dimension     = sDim;
    centerPoint  = sCenter;
}
```

The variables **type**, **dimension**, and **centerPoint** are data members for keeping track of necessary information.

Next, we define the data member constants as follows:

```
public static enum Type {ELLIPSE, RECTANGLE, ROUNDED_RECTANGLE}
private static final Dimension DEFAULT_DIMENSION
    = new Dimension(200, 200);
private static final Point DEFAULT_CENTER_PT
    = new Point(350, 350);
```

In the previous step, the **draw** method drew a fixed-size circle. We need to modify it to draw three different geometric shapes based on the value of the data member **type**. We can modify the method to

```
public void draw(Graphics g) {
    g.setColor(Color.blue);
    drawShape(g);
}
```

with the private method **drawShape** defined as

```
private void drawShape(Graphics g) {
    switch (type) {
        case ELLIPSE:
            //code to draw a filled oval comes here
            break;
        case RECTANGLE:
            //code to draw a filled rectangle comes here
            break;
```

5.8 Sample Development—continued

```

        case ROUNDED_RECTANGLE:
            //code to draw a filled rounded rectangle
            //comes here
            break;
    }
}

```

step 3 code

Here's the modified main class **Ch5DrawShape**:

```

import java.awt.*;
import java.util.*;

/*
    Chapter 5 Sample Development: Handle User Input for Shape Type (Step 3)
    The main class of the program.
*/

class Ch5DrawShape {
    . . .

    public void start( ) {
        DrawableShape shape1 = getShape();
        canvas.addShape(shape1);
        canvas.setMovement(DrawingBoard.SMOOTH);
        canvas.setVisible(true);
        canvas.start();
    }

    private DrawableShape getShape( ) {
        DrawableShape.Type type = inputShapeType();
        Dimension dim = inputDimension();
        Point centerPt = inputCenterPoint();
        DrawableShape shape = new DrawableShape(type, dim, centerPt);
        return shape;
    }
}

```

start

getShape


```

private DrawableShape.Type inputShapeType( ) {
    System.out.print("Selection: Enter the Shape number\n" +
        "    1 - Ellipse \n" +
        "    2 - Rectangle \n" +
        "    3 - Rounded Rectangle \n");

    int selection = scanner.nextInt();

    DrawableShape.Type type;

    switch (selection) {

        case 1: type = DrawableShape.Type.ELLIPSE;
                break;

        case 2: type = DrawableShape.Type.RECTANGLE;
                break;

        case 3: type = DrawableShape.Type.ROUNDED_RECTANGLE;
                break;

        default: type = DrawableShape.Type.ELLIPSE;
                 break;

    }

    return type;
}

```

inputShapeType

```

private Dimension inputDimension( ) {
    System.out.print("Enter the width of the shape\n" +
        "between 100 and 500 inclusive: ");

    int width = scanner.nextInt();

    if (width < 100 || width > 500) {
        width = 100;
    }

    System.out.print("Enter the height of the shape\n" +
        "between 100 and 500 inclusive: ");

    int height = scanner.nextInt();

    if (height < 100 || height > 500) {
        height = 100;
    }

    return new Dimension(width, height);
}

```

inputDimension

5.8 Sample Development—continued

```

private Point inputCenterPoint( ) {
    System.out.print("Enter the x value of the center point\n" +
                    "between 200 and 800 inclusive: ");

    int x = scanner.nextInt();

    if (x < 200 || x > 800) {
        x = 200;
    }

    System.out.print("Enter the y value of the center point\n" +
                    "between 100 and 500 inclusive: ");

    int y = scanner.nextInt();

    if (y < 100 || y > 500) {
        y = 100;
    }

    return new Point(x, y);
}
. . .
}

```

inputCenterPoint

The **DrawableShape** class is now modified to this:

```

import java.awt.*;

/*
   Step 3: Draw different shapes
   A class whose instances know how to draw themselves.
*/

class DrawableShape {
    public static enum Type {ELLIPSE, RECTANGLE, ROUNDED_RECTANGLE}

    private static final Dimension DEFAULT_DIMENSION
        = new Dimension(200, 200);
}

```

Data Members

```

private static final Point DEFAULT_CENTER_PT = new Point(350, 350);

private Point    centerPoint;

private Dimension dimension;

private Type     type;

public DrawableShape(Type sType, Dimension sDim, Point sCenter) {

    type      = sType;
    dimension = sDim;
    centerPoint = sCenter;
}

public void draw(Graphics g) {

    g.setColor(Color.blue);

    drawShape(g);

    . . .

public void setType(Type shapeType) {

    type = shapeType;
}

private void drawShape(Graphics g) {
    switch (type) {
        case ELLIPSE:
            g.fillOval(centerPoint.x - dimension.width/2,
                       centerPoint.y - dimension.height/2,
                       dimension.width,
                       dimension.height);

            break;

        case RECTANGLE:
            g.fillRect(centerPoint.x - dimension.width/2,
                       centerPoint.y - dimension.height/2,
                       dimension.width,
                       dimension.height);

            break;

        case ROUNDED_RECTANGLE:
            g.fillRoundRect(centerPoint.x - dimension.width/2,
                            centerPoint.y - dimension.height/2,
                            dimension.width,
                            dimension.height,

```

Constructor

draw

setType

drawShape

5.8 Sample Development—continued

```

        (int) (dimension.width * 0.3),
        (int) (dimension.height * 0.3));
    }
    break;
}
}
}

```

Notice how we add code for handling the case when an invalid number is entered in the **inputShapeType** method. We use the **default** case to set the shape type to **ELLIPSE** if an invalid value is entered. In addition to handling the invalid entries, it is critical for us to make sure that all valid entries are handled correctly. For example, we cannot leave the type undefined or assigned to a wrong value when one of the valid data is entered.

Things to Remember



When we write a selection control statement, we must make sure that all possible cases are handled correctly.

step 3 test

Now we run the program multiple times, trying various shape types, dimensions, and center points. After we verify that everything is working as expected, we proceed to the next step.

Step 4 Development: Allow the User to Select a Color

step 4 design

In the fourth development step, we add a routine that allows the user to specify the color of the selected shape. We adopt the same input style for accepting the shape type as in step 3. We list five different color choices and let the user select one of them by entering the corresponding number. We use a default color when an invalid number is entered. Analogous to the shape selection routine, we will add a method named **inputColor** to the **Ch5DrawShape** class. The structure of this method is identical to that of the input methods, except the return type is **Color**. Using the **inputColor** method, we can define the **getShape** method as follows:

```

private DrawableShape getShape( ) {
    DrawableShape.Type type = inputShapeType();
    Dimension dim = inputDimension();
}

```

```

    Point centerPt = inputCenterPoint();

    Color color = inputColor();

    DrawableShape shape
        = new DrawableShape(type, dim, centerPt, color);

    return shape;
}

```

We make a small extension to the **DrawableShape** class by changing the constructor to accept a color as its fourth argument and adding a data member to keep track of the selected color.

step 4 code

Here's the modified **Ch5DrawShape** class:

```

import java.awt.*;
import java.util.*;

/*
    Chapter 5 Sample Development: Color selection (Step 4)

    The main class of the program.
*/

class Ch5DrawShape {
    . . .

    private DrawableShape getShape( ) {
        DrawableShape.Type type = inputShapeType();
        Dimension dim = inputDimension();
        Point centerPt = inputCenterPoint();
        Color color = inputColor();

        DrawableShape shape
            = new DrawableShape(type, dim, centerPt, color);

        return shape;
    }

    private Color inputColor( ) {
        System.out.print("Selection: Enter the Color number\n" +
            "    1 - Red \n" +
            "    2 - Green \n" +
            "    3 - Blue \n" +
            "    4 - Yellow \n" +
            "    5 - Magenta \n");

        int selection = scanner.nextInt();
    }
}

```

getShape

inputColor

5.8 Sample Development—continued

```

Color color;
switch (selection) {

    case 1: color = Color.red;
           break;

    case 2: color = Color.green;
           break;

    case 3: color = Color.blue;
           break;

    case 4: color = Color.yellow;
           break;

    case 5: color = Color.magenta;
           break;

    default: color = Color.red;
            break;

}

return color;
}

. . .
}

```

The **DrawableShape** class is now modified to this:

```

import java.awt.*;

/*
   Step 4: Adds the color choice

   A class whose instances know how to draw themselves.
*/
class DrawableShape {

    . . .

    private static final Color DEFAULT_COLOR = Color.BLUE;

    . . .
}

```

Data Members

```

private Color    fillColor;

. . .

public DrawableShape(Type sType, Dimension sDim,
                    Point sCenter, Color sColor) {
    type        = sType;
    dimension    = sDim;
    centerPoint = sCenter;
    fillColor    = sColor;
}

public void draw(Graphics g) {
    g.setColor(fillColor);

    drawShape(g);
}

. . .
}

```

Constructor

draw

step 4 test

Now we run the program several times, each time selecting a different color, and we verify that the shape is drawn in the chosen color. After we verify the program, we move on to the next step.

Step 5 Development: Allow the User to Select a Motion Type

step 5
design

In the fifth development step, we add a routine that allows the user to select the motion type. We give three choices to the user: stationary, random, or smooth. The same design we used in steps 3 and 4 is applicable here, so we adopt it for the motion type selection also. Since we adopt the same design, we can ease into the coding phase.

step 5 code

Here's the modified main class **Ch5DrawShape**:

```

import java.awt.*;
import java.util.*;

/*
 * Chapter 5 Sample Development: Color selection (Step 5)
 *
 * The main class of the program.
 */
class Ch5DrawShape {
    . . .
}

```

5.8 Sample Development—continued

```

public void start( ) {
    DrawableShape shapel = getShape();
    canvas.addShape(shapel);
    canvas.setMovement(inputMotionType());
    canvas.setVisible(true);
    canvas.start();
}
. . .
private DrawingBoard.Movement inputMotionType( ) {
    System.out.print("Selection: Enter the Motion number\n" +
        " 1 - Stationary (no movement) \n" +
        " 2 - Random Movement \n" +
        " 3 - Smooth Movement \n" );

    int selection = scanner.nextInt();

    DrawingBoard.Movement type;
    switch (selection) {
        case 1: type = DrawingBoard.Movement.STATIONARY;
                break;
        case 2: type = DrawingBoard.Movement.RANDOM;
                break;
        case 3: type = DrawingBoard.Movement.SMOOTH;
                break;
        default: type = DrawingBoard.Movement.SMOOTH;
                 break;
    }

    return type;
}
. . .
}

```

start

inputMotionType

No changes are required for the **DrawableShape** class, as the **DrawingBoard** class is the one responsible for the shape movement.

step 5 test

Now we run the program multiple times and test all three motion types. From what we have done, we can't imagine the code we have already written in the earlier steps to cause any problems; but if we are not careful, a slight change in one step could cause the code developed from the earlier steps to stop working correctly (e.g., erroneously reusing data members in newly written methods). So we should continue to test all aspects of the program diligently. After we are satisfied with the program, we proceed to the final step.

Step 6 Development: Finalize

program review

We will perform a critical review of the program, looking for any unfinished method, inconsistency or error in the methods, unclear or missing comments, and so forth. We should also not forget to improve the program for cleaner code and better readability. Another activity we can pursue in the final step is to look for extensions.

possible extensions

There are several interesting extensions we can make to the program. First is the morphing of an object. In the current implementation, once the shape is selected, it will not change. It would be more fun to see the shape changes; for example, the width and height of the shape's dimension can be set to vary while the shape is drawn. Another interesting variation is to make a circle morph into a rectangle and morph back into a circle. Second is the drawing of multiple shapes. Third is the variation in color while the shape is drawn. Fourth is the drawing of a text (we "draw" a text on the **Graphics** context just as we draw geometric shapes). You can make the text scroll across the screen from right to left by setting the motion type of **DrawingBoard** to **STATIONARY** and updating the center point value within our **DrawableShape** class. All these extensions are left as exercises.

S u m m a r y

- A selection control statement is used to alter the sequential flow of control.
- The if and switch statements are two types of selection control.
- The two versions of the if statement are if-then-else and if-then.
- A boolean expression contains conditional and boolean operators and evaluates to true or false.
- Three boolean operators in Java are AND (&&), OR (||), and NOT (!).
- DeMorgan's laws state $!(P \ \&\& \ Q)$ and $!P \ || \ !Q$ are equivalent and $!(P \ || \ Q)$ and $!P \ \&\& \ !Q$ are equivalent.
- Logical operators && and || are evaluated by using the short-circuit evaluation technique.
- A boolean flag is useful in keeping track of program settings.
- An if statement can be a part of the then or else block of another if statement to formulate nested if statements.
- Careful attention to details is important to avoid illogically constructed nested if statements.

- When the equality symbol `==` is used in comparing the variables of reference data type, we are comparing the addresses.
- The `switch` statement is useful for expressing a selection control based on equality testing between data of type `char`, `byte`, `short`, or `int`.
- The `break` statement causes the control to break out of the surrounding `switch` statement (*note*: also from other control statements introduced in Chap. 6).
- The standard classes introduced in this chapter are

<code>java.awt.Graphics</code>	<code>java.awt.Point</code>
<code>java.awt.Color</code>	<code>java.awt.Dimension</code>
- The `java.awt.Graphics` class is used to draw geometric shapes.
- The `java.awt.Color` class is used to set the color of various GUI components.
- The `java.awt.Point` class is used to represent a point in two-dimensional space.
- The `java.awt.Dimension` class is used to represent a bounding rectangle of geometric shapes and other GUI components.
- The enumerated constants provide type safety and increase the program readability.

Key Concepts

<ul style="list-style-type: none"> sequential execution control statements if statement boolean expressions relational operators selection statements nested if statements 	<ul style="list-style-type: none"> increment and decrement operators boolean operators switch statements break statements defensive programming content pane of a frame enumerated constants
---	---

Exercises

1. Indent the following if statements properly.
 - a. `if (a == b) if (c == d) a = 1; else b = 1; else c = 1;`
 - b. `if (a == b) a = 1; if (c == d) b = 1; else c = 1;`
 - c. `if (a == b) {if (c == d) a = 1; b = 2; } else b = 1;`
 - d. `if (a == b) {

 if (c == d) a = 1; b = 2; }

 else {b = 1; if (a == d) d = 3;}`
2. Which two of the following three if statements are equivalent?
 - a. `if (a == b)

 if (c == d) a = 1;

 else b = 1;`

- b. `if (a == b) {
 if (c == d) a = 1; }
 else b = 1;`
- c. `if (a == b)
 if (c == d) a = 1;
 else b = 1;`
3. Evaluate the following boolean expressions. For each of the following expressions, assume x is 10, y is 20, and z is 30. Indicate which of the following boolean expressions are always true and which are always false, regardless of the values for x, y, or z.

- a. `x < 10 || x > 10`
- b. `x > y && y > x`
- c. `(x < y + z) && (x + 10 <= 20)`
- d. `z - y == x && Math.abs(y - z) == x`
- e. `x < 10 && x > 10`
- f. `x > y || y > x`
- g. `!(x < y + z) || !(x + 10 <= 20)`
- h. `!(x == y) && (x != y) && (x < y || y < x)`

4. Express the following switch statement by using nested if statements.

```
switch (grade) {
    case 10:
    case 9: a = 1;
           b = 2;
           break;

    case 8: a = 3;
           b = 4;
           break;

    default: a = 5;
            break;
}
```

5. Write an if statement to find the smallest of three given integers without using the min method of the Math class.
6. Draw control flow diagrams for the following two switch statements.

```
switch (choice) {
    case 1: a = 0;
           break;

    case 2: b = 1;
           break;

    case 3: c = 2;
           break;

    default: d = 3;
            break;
}
```

```
switch (choice) {
    case 1: a = 0;

    case 2: b = 1;

    case 3: c = 2;

    default: d = 3;
}
}
```

7. Write an if statement that prints out a message based on the following rules:

If the Total Points Are	Message to Print
≥ 100	You won a free cup of coffee.
≥ 200	You won a free cup of coffee and a regular-size doughnut.
≥ 300	You won a free cup of coffee and a regular-size doughnut and a 12-oz orange juice.
≥ 400	You won a free cup of coffee and a regular-size doughnut and a 12-oz orange juice and a combo breakfast.
≥ 500	You won a free cup of coffee and a regular-size doughnut and a 12-oz orange juice and a combo breakfast and a reserved table for one week.

8. Rewrite the following if statement, using a switch statement.

```

selection = scanner.nextInt( );

if (selection == 0)
    System.out.println("You selected Magenta");

else if (selection == 1)
    System.out.println("You selected Cyan");

else if (selection == 2)
    System.out.println("You selected Red");

else if (selection == 3)
    System.out.println("You selected Blue");

else if (selection == 4)
    System.out.println("You selected Green");

else
    System.out.println("Invalid selection");

```

9. At the end of movie credits you see the year movies are produced in Roman numerals, for example, MCMXCVII for 1997. To help the production staff determine the correct Roman numeral for the production year, write an applet or application that reads a year and displays the year in Roman numerals.

Roman Numeral	Number
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

Remember that certain numbers are expressed by using a “subtraction,” for example, IV for 4, CD for 400, and so forth.

10. Write a program that replies either Leap Year or Not a Leap Year, given a year. It is a leap year if the year is divisible by 4 but not by 100 (for example, 1796 is a leap year because it is divisible by 4 but not by 100). A year that is divisible by both 4 and 100 is a leap year if it is also divisible by 400 (for example, 2000 is a leap year, but 1800 is not).
11. One million is 10^6 and 1 billion is 10^9 . Write a program that reads a power of 10 (6, 9, 12, etc.) and displays how big the number is (Million, Billion, etc.). Display an appropriate message for the input value that has no corresponding word. The table below shows the correspondence between the power of 10 and the word for that number.

Power of 10	Number
6	Million
9	Billion
12	Trillion
15	Quadrillion
18	Quintillion
21	Sextillion
30	Nonillion
100	Googol

12. Write a program `RecommendedWeightWithTest` by extending the `RecommendedWeight` (see Exercise 8 on page 209). The extended program will include the following test:

```

if (the height is between 140cm and 230cm)
    compute the recommended weight
else
    display an error message

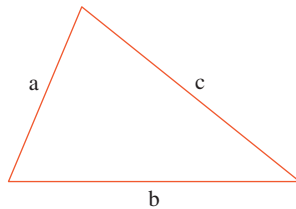
```

13. Extend the `RecommendedWeightWithTest` program in Exercise 12 by allowing the user to enter his or her weight and printing out the message You should exercise more if the weight is more than 10 lb over the ideal weight and You need more nourishment if the weight is more than 20 lb under the recommended weight.
14. Employees at MyJava Lo-Fat Burgers earn the basic hourly wage of \$7.25. They will receive time-and-a-half of their basic rate for overtime hours. In addition, they will receive a commission on the sales they generate while tending the counter. The commission is based on the following formula:

Sales Volume	Commission
\$1.00 to \$99.99	5% of total sales
\$100.00 to \$299.99	10% of total sales
\geq \$300.00	15% of total sales

Write an application that inputs the number of hours worked and the total sales and computes the wage.

15. Using the `DrawingBoard` class, write a screensaver that displays a scrolling text message. The text messages moves across the window, starting from the right edge toward the left edge. Set the motion type to stationary, so the `DrawingBoard` does not adjust the position. You have to adjust the text's position inside your `DrawableShape`.
16. Define a class called `Triangle` that is capable of computing the perimeter and area of a triangle, given its three sides a , b , and c , as shown below. Notice that side b is the base of the triangle.



$$\text{Perimeter} = a + b + c$$

$$\text{Area} = \sqrt{s(s - a)(s - b)(s - c)}$$

$$\text{where } s = \frac{a + b + c}{2}$$

The design of this class is identical to that for the `Ch5Circle` class from Section 5.1. Define a private method `isValid` to check the validity of three sides. If any one of them is invalid, the methods `getArea` and `getPerimeter` will return the constant `INVALID_DIMENSION`.

17. Modify the `Ch5RoomWinner` class so the three dorm lottery cards are drawn vertically. Make the code for drawing flexible by using the `HEIGHT` constant in determining the placement of three cards.

Development Exercises

For the following exercises, use the incremental development methodology to implement the program. For each exercise, identify the program tasks, create a design document with class descriptions, and draw the program diagram. Map out the development steps at the start. Present any design alternatives and justify your selection. Be sure to perform adequate testing at the end of each development step.

18. MyJava Coffee Outlet (see Exercise 25 from Chap. 3) decided to give discounts to volume buyers. The discount is based on the following table:

Order Volume	Discount
≥ 25 bags	5% of total price
≥ 50 bags	10% of total price
≥ 100 bags	15% of total price
≥ 150 bags	20% of total price
≥ 200 bags	25% of total price
≥ 300 bags	30% of total price

Each bag of beans costs \$5.50. Write an application that accepts the number of bags ordered and prints out the total cost of the order in the following style:

```
Number of Bags Ordered: 173 - $ 951.50
      Discount:
                20% - $ 190.30
Your total charge is: $ 761.20
```

19. Combine Exercises 18 and 25 of Chap. 3 to compute the total charge including discount and shipping costs. The output should look like the following:

```
Number of Bags Ordered: 43 - $ 236.50
      Discount:
                5% - $ 11.83
Boxes Used:
      1 Large - $1.80
      2 Medium - $2.00
Your total charge is: $ 228.47
```

Note: The discount applies to the cost of beans only.

20. You are hired by Expressimo Delivery Service to develop an application that computes the delivery charge. The company allows two types of packaging—letter and box—and three types of service—Next Day Priority, Next Day Standard, and 2-Day. The following table shows the formula for computing the charge:

Package Type	Next Day Priority	Next Day Standard	2-Day
Letter	\$12.00, up to 8 oz	\$10.50, up to 8 oz	Not available
Box	\$15.75 for the first pound. Add \$1.25 for each additional pound over the first pound.	\$13.75 for the first pound. Add \$1.00 for each additional pound over the first pound.	\$7.00 for the first pound. Add \$0.50 for each additional pound over the first pound.

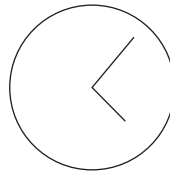
The program will input three values from the user: type of package, type of service, and weight of the package.

21. Ms. Latte's Mopeds 'R Us rents mopeds at Monterey Beach Boardwalk. To promote business during the slow weekdays, the store gives a huge discount. The rental charges are as follows:

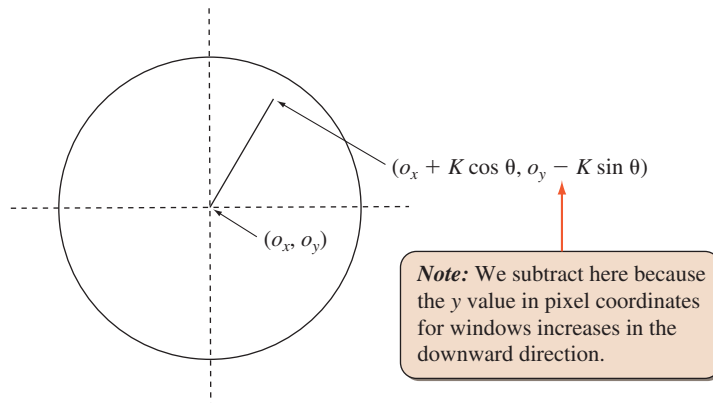
Moped Type	Weekday Rental	Weekend Rental
50cc Mopette	\$15.00 for the first 3 h, \$2.50 per hour after the first 3 h.	\$30.00 for the first 3 h, \$7.50 per hour after the first 3 h.
250cc Mohawk	\$25.00 for the first 3 h, \$3.50 per hour after the first 3 h.	\$35.00 for the first 3 h, \$8.50 per hour after the first 3 h.

Write a program that computes the rental charge, given the type of moped, when it is rented (either weekday or weekend), and the number of hours rented.

22. Write an application program that teaches children how to read a clock. Use `JOptionPane` to enter the hour and minute. Accept only numbers between 0 and 12 for hour and between 0 and 59 for minute. Print out an appropriate error message for an invalid input value. Draw a clock that looks something like this:



To draw a clock hand, you use the `drawLine` method of the `Graphics` class. The endpoints of the line are determined as follows:



The value for constant K determines the length of the clock hand. Make the K larger for the minute hand than for the hour hand. The angle θ is expressed in radians. The angle θ_{\min} of the minute hand is computed as

$$(90 - \text{Minute} \times 6.0) \frac{\pi}{180}$$

and the angle θ_{hr} of the hour hand is computed as

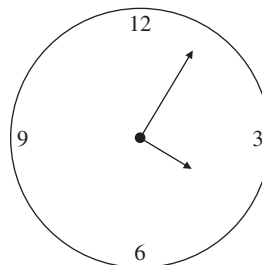
$$\left(90 - \left(\text{Hour} + \frac{\text{Minute}}{60.0}\right) \times 30.0\right) \frac{\pi}{180}$$

where Hour and Minute are input values. The values 6.0 and 30.0 designate the degrees for 1 min and 1 h (i.e., the minute hand moves 6 degrees in 1 min and the hour hand moves 30.0 degrees in 1 h). The factor $\pi/180$ converts a degree into the radian equivalent.

You can draw the clock on the content pane of a frame window by getting the content pane's Graphic object as described in the chapter. Here's some sample code:

```
import javax.swing.*;
import java.awt.*; //for Graphics
...
JFrame win;
Container contentPane;
Graphics g;
...
win = new JFrame();
win.setSize(300, 300);
win.setLocation(100,100);
win.setVisible(true);
...
contentPane = win.getContentPane();
g = contentPane.getGraphics();
g.drawOval(50,50,200,200);
```

23. Extend the application in Exercise 22 by drawing a more realistic, better-looking clock, such as this one:



24. After starting a successful coffee beans outlet business, MyJava Coffee Outlet is now venturing into the fast-food business. The first thing the management decides is to eliminate the drive-through intercom. MyJava Lo-Fat Burgers is the only fast-food establishment in town that provides a computer screen and mouse for its drive-through customers. You are hired as a freelance computer consultant. Write a program that lists items for three menu categories: entree, side dish, and drink. The following table lists the items available for each entry and their prices. Choose appropriate methods for input and output.

Entree		Side Dish		Drink	
Tofu Burger	\$3.49	Rice Cracker	\$0.79	Cafe Mocha	\$1.99
Cajun Chicken	\$4.59	No-Salt Fries	\$0.69	Cafe Latte	\$1.99
Buffalo Wings	\$3.99	Zucchini	\$1.09	Espresso	\$2.49
Rainbow Fillet	\$2.99	Brown Rice	\$0.59	Oolong Tea	\$0.99