

Statements and Control Flow

The program examples presented until now have executed from top to bottom without making any decisions. In this chapter, we have programs select among two or more alternatives. We also demonstrate how to write programs that repeatedly execute the same sequence of instructions. Both instructions to computers and instructions in everyday life are filled with conditional and iterative statements. A conditional instruction for your microwave oven might say “*If* you wish to defrost press the defrost button; otherwise, press the full power button.” An iterative instruction for baking a loaf of bread might say “Let the dough rise in a warm place *until* it has doubled in size.” Conditional and iterative statements are controlled by boolean expressions. A boolean expression is either true or false. “If it is raining, wear your raincoat” is an instruction given by many parents and is followed if it is true that it is raining. In Java, expressions that evaluate as true or false are of type `boolean`. To direct the flow of control properly you need to learn how to write `boolean` expressions.

3.1 EXPRESSION, BLOCK, AND EMPTY STATEMENTS

Java has many kinds of *statements*. Most of the statements that we have shown have specified the evaluation of an expression. We’ll soon look at statements that select between two alternatives and statements that repeat many times. Before doing that, we need to look more closely at the statements that we have been using. The normal flow of instructions in Java is to execute the statements of the program in sequential order from top to bottom.

All the statements used so far have been either *variable declaration statements* or *expression statements*. Variable declaration statements begin with a type, such as `int` or `String`, and end with a semicolon, as in

```
int width, height, area;
String hello = "Hello, world!";
double size = 1.5, x;
```

The first declares three variables of type `int`. The second declares one variable of type `String` and initializes it. The third declares and initializes the variable `size`, but not the variable `x`. Declaration statements start with a type and are followed by a comma separated by a list of variables. The variables may be initialized by using the equals sign followed typically by a literal. In Java all variables need to be declared.

Expression statements are formed by adding a semicolon to the end of an expression. Expressions are basic to performing computations. Not all expressions are valid in expression statements. The two types of expressions used so far that are valid in expression statements are assignment expressions and method call expressions. An *assignment expression* is any expression involving the assignment operator. A *method call expression* does not involve an assignment operator. The following are examples of expression statements.

```
area = width * height; //simple assignment statement
System.out.println(...); //method call expression
```

A statement used for grouping a number of statements is a block. A *block* is a sequence of one or more statements enclosed by braces. A block is itself a statement. A simple example is

```
{
    x = 1;
    y = 2 * x + 1;
    System.out.println(y);
    System.out.println(x);
}
```

Statements inside a block can also be blocks. The inside block is called an *inner block*, which is *nested* in the *outer block*. An example is

```
{ //outer block
    x = 1;
    { //inner block
        y = 2;
        System.out.println(y);
    } //end of inner block
    System.out.println(x);
}
```

This example merely demonstrates the syntax of a block; we wouldn't normally put a block inside another block for no reason. Most nested blocks involve declaration statements that create local variables. A simple example of a block with declarations is

```

{
    int i = 5 + j;
    //i is created in this block, j is from elsewhere
    ...
} //end of block i disappears

```

In this example the `int` variable `i` is created when this block is executed. When this block is started, `i` is placed in memory with its initial value calculated as 5 plus the value of `j`. When the block is exited, the variable disappears.

Blocks are not terminated by semicolons. Rather they are terminated by a closing brace, also called the right brace. Recall that the semicolon, when used, is part of the statement, not something added to the statement. For example, the semicolon turns an expression into a statement. Understanding this will make it much easier for you to create syntactically correct programs with the new statement types that we introduce in this chapter.

3.1.1 Empty Statement

The simplest statement is the *empty statement*, or *null statement*. It is just a semicolon all by itself and results in no action. A semicolon placed after a block is an empty statement and is irrelevant to the program's actions. The following code fragment produces exactly the same result as the nested block example in the preceding section. The string of semicolons simply create seven empty statements following the inner block.

```

{
    x = 1;
    {
        y = 2;
        System.out.println(y);
        };;;;;;
    System.out.println(x);
}

```



3.2 BOOLEAN EXPRESSIONS

A *boolean expression* is any expression that evaluates to either true or false. Java includes a primitive type `boolean`. The two simplest boolean expressions are the boolean literals `true` and `false`. In addition to these two literals, boolean values result from expressions involving either relational operators for comparing numbers or logical operators that act on boolean values.

3.2.1 Relational and Equality Operators

All conditional statements require some boolean expression to decide which execution path to follow. Java uses four *relational operators*: less than, `<`; greater than, `>`; less than or equal, `<=`; and greater than or equal, `>=`. Java also contains two equality

operators: equal, `==`; and not equal, `!=`. They can be used between any two numeric values. The equality operators may also be used when comparing nonnumeric types. They are listed in the following table.

Operator	Name	Example
<code><</code>	Less than	<code>10 < 20</code> is true.
<code>></code>	Greater than	<code>10 > 20</code> is false.
<code>==</code>	Equal	<code>10 == 20</code> is false.
<code><=</code>	Less than or equal	<code>10 <= 10</code> is true.
<code>>=</code>	Greater than or equal	<code>11 >= 10</code> is true.
<code>!=</code>	Not equal	<code>10 != 20</code> is true.

The relational operators can be used in assignment to boolean variables, as in

```
int i = 3, j = 4;
boolean flag;
flag = 5 < 6;           //flag is now true
flag = (i == j);        //flag is now false
flag = (j + 2) <= 6;     //flag is now true
```

3.2.2 Logical operators

Once you have a boolean value, either stored in a variable representing a primitive boolean value (for example, `boolean done = false;`) or as the result of an expression involving a relational operator (for example, `(x < y)`), you can combine these boolean values by using the logical operators. Java provides three logical operators, “and,” “or,” and “not.” The meaning of these operators is given in the following table.

Operator	Name	Description	Example—Assume x is 10 and y is 20
<code>&&</code>	and	The expression <code>x && y</code> is true if both x AND y are true and false otherwise.	<code>(x < 20) && (y < 30)</code> is true.
<code> </code>	or	The expression <code>x y</code> is true if either x OR y (or both) is true and false otherwise.	<code>(x < 20) (y > 30)</code> is true.
<code>!</code>	not	The expression <code>!x</code> is true if x is false and false otherwise.	<code>!(x < 20)</code> is false.

For example, if you wanted to determine whether a person in a database was an adult but not a senior citizen, you could check if their age was greater than or equal to 18 *and* their age was less than 65. The following Java code fragment will print out “full fare adult is true” if this condition is met; otherwise, it prints “full fare adult is false”.

```
boolean b = (ageOfPerson >= 18 && ageOfPerson < 65);
System.out.println("full fare adult is " + b);
```

For an example of the use of “or,” consider the opposite situation as above where you wanted to find out if a reduced fare was appropriate. You might write

```
b = (ageOfPerson < 18 || ageOfPerson >= 65);
System.out.println("reduced fare is " + b);
```

The logical operators `&&` and `||` use *short-circuit evaluation*. In the preceding example of a logical “and” expression, if the `ageOfPerson` were 10, then the test for `ageOfPerson < 65` would be omitted. Used partly for efficiency reasons, this approach is helpful when the second part of such an expression could lead to an undesirable result, such as program termination.

As with other operators, the relational, equality, and logical operators have rules of precedence and associativity that determine precisely how expressions involving these operators are evaluated, as shown in the following table.

Operator Precedence and Associativity	
Operators	Associativity
() ++ (postfix) -- (postfix)	Left to right
+ (unary) - (unary) ++ (prefix) -- (prefix) !	Right to left
* / %	Left to right
+ -	Left to right
< <= > >=	Left to right
== !=	Left to right
&&	Left to right
	Left to right
= += -= *= /= etc.	Right to left

Note that with the exception of the boolean unary operator negation, the relational, boolean, and equality operators have lower precedence than the arithmetic operators. Only the assignment operators have lower precedence.



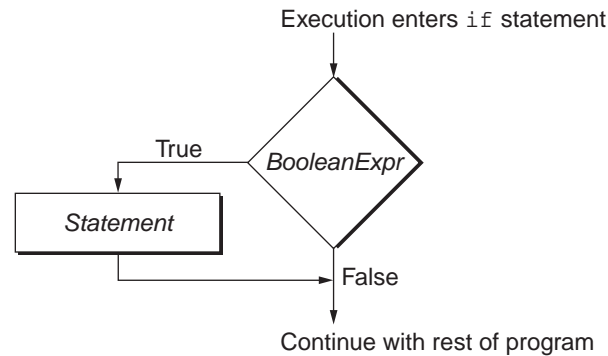
3.3 THE if STATEMENT

Computers make decisions by evaluating expressions and executing different statements based on the value of the expression. The simplest type of decision is one that can have only two possible outcomes, such as go left versus go right or continue versus stop. In Java, we use *boolean expressions* to control decisions that have two possible outcomes.

The *if statement* is a conditional statement. An if statement has the general form

```
if ( BooleanExpr )  
    Statement
```

If the expression *BooleanExpr* is true, then the statement, *Statement*, is executed; otherwise, *Statement* is skipped. *Statement* is called the *then statement*. In some programming languages, but not Java, then is used to signal the then statement. After the if statement has been executed, control passes to the next statement. The flow of execution can skip around *Statement* as shown in the following diagram.



Note the absence of semicolons in the general form of the if statement. Recall that the semicolon, when required, is part of the *Statement* and is not used to separate statements, as in

```
if (temperature < 32)  
    System.out.println("Warning: Below Freezing!");  
    System.out.println("It's " + temperature + "degrees");
```

The message `Warning: Below Freezing!` is printed only when the temperature is less than 32. The second print statement is always executed. This example has a semicolon at the end of the if statement because the statement inside the if statement is an expression statement that ends with a semicolon.

When the *Statement* inside an if statement is a block, you get if statements that look like

```
if (temperature < 32)  
{  
    System.out.println("Warning Warning Warning!");  
    System.out.println("Warning: Below Freezing!");  
    System.out.println("Warning Warning Warning!");  
}
```

Here you can see the importance of the block as a means of grouping statements. In this example, what otherwise would be three separate statements are grouped, and all are executed when the boolean expression is true. The formatting shown of the if statement with a block as the *Statement* aligns vertically with the braces of the block statement. An alternative formatting—and the one that we use—places the opening brace on the same line as the keyword if and then aligns the closing brace with the keyword as shown here.

```
if (temperature < 32) {  
    System.out.println("Warning Warning Warning!");  
    System.out.println("Warning: Below Freezing!");  
    System.out.println("Warning Warning Warning!");  
}
```

At the end of this chapter, we discuss further which style to choose.

3.3.1 Problem Solving with the if statement

A different number is initially placed in each of three boxes, labeled *a*, *b*, and *c*, respectively. The problem is to rearrange or sort the numbers so that the final number in box *a* is less than that in box *b* and that the number in box *b* is less than that in box *c*. Initial and final states for a particular set of numbers are as follows.

	Before		After
<i>a</i>	17	<i>a</i>	6
<i>b</i>	6	<i>b</i>	11
<i>c</i>	11	<i>c</i>	17

Pseudocode for performing this sorting task involves the following steps.

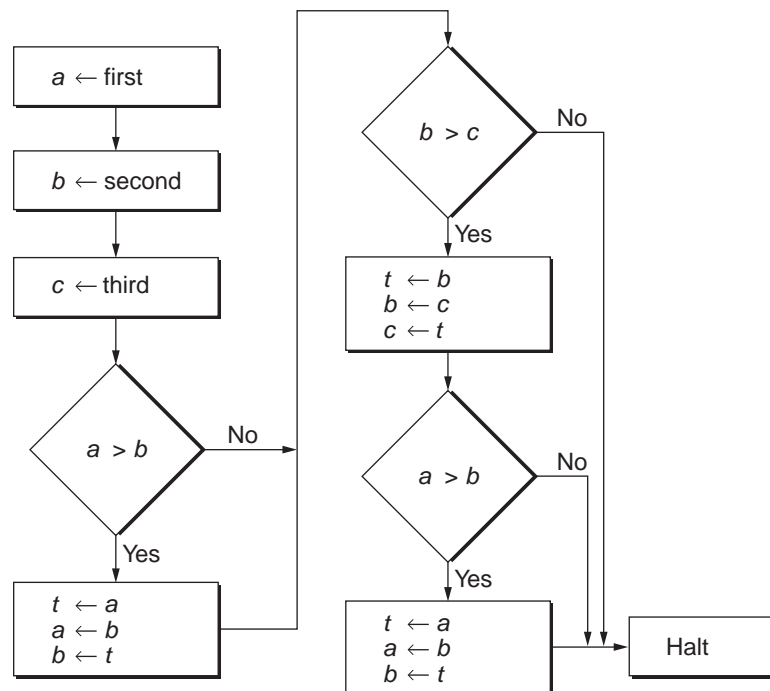
PSEUDOCODE FOR THREE-NUMBER SORT

1. Place the first number in box *a*.
2. Place the second number in box *b*.
3. Place the third number in box *c*.
4. If the number in *a* is not larger than the number in *b*, go to step 6.
5. Interchange the number in *a* with that in *b*.
6. If the number in *b* is larger than the number in *c*, then go to step 7; otherwise, halt.
7. Interchange the numbers in *b* and *c*.
8. If the number in *a* is larger than that in *b*, then go to step 9; otherwise, halt.
9. Interchange the numbers in *a* and *b*.
10. Halt.

Let's execute this pseudocode with the three specific numbers previously given: 17, 6, and 11, in that order. We always start with the first instruction. The contents of the three boxes at various stages of execution are shown in the following table.

Box	Step 1	Step 2	Step 3	Step 5	Step 7
a	17	17	17	6	6
b		6	6	17	11
c			11	11	17

To execute step 1, we place the first number, 17, in box *a*; similarly, at the end of instruction 3, the 6 has been inserted into box *b*, and box *c* contains the 11. As 17 is larger than 6, the condition tested in step 4 is false, and we proceed to instruction 5; this step switches the values into boxes *a* and *b* so that box *a* now contains the 6 and box *b* has the 17. Step 6 has now been reached, and we compare the number in box *b* (17) to that in box *c* (11); 17 is greater than 11, so a transfer is made to step 7. The numbers in boxes *b* and *c* are then interchanged so that box *b* has the 11 and box *c* has the 17. The test in step 8 fails (6 is not larger than 11) and the computation then halts. The three numbers have been sorted in ascending sequence (i.e., $6 < 11 < 17$). You should convince yourself by bench testing this algorithm with other values of *a*, *b*, and *c* that the computation described by the pseudocode will work correctly for any three numbers. A flowchart of the sorting algorithm is shown in the following diagram.



Note that we decomposed the operation of interchanging two numbers into three more primitive instructions. Box *t* is used as temporary storage to hold intermediate results. In order to interchange or switch the two numbers *a* and *b*, we first temporarily store one of the numbers, say, *a*, in *t* ($t \leftarrow a$); next the other number is stored in *a* ($a \leftarrow b$), and, last, the first number is placed in *b* ($b \leftarrow t$). Note that the instruction sequence “ $a \leftarrow b$; $b \leftarrow a$ ” will not interchange *a* and *b* because the first instruction effectively destroys the old

value in *a*. In computer terms, the labeled boxes are analogous to memory or storage areas that can contain values.

Next we code in Java the pseudocode version of our sorting program.

```
// SortInput.java - sort three numbers
import tio.*; // use the package tio

class SortInput {
    public static void main (String[] args) {
        int a, b, c, t;

        System.out.println("type three integers:");
        a = Console.in.readInt();
        b = Console.in.readInt();
        c = Console.in.readInt();
        if (a > b) {
            t = a;
            a = b;
            b = t;
        }
        if (b > c) {
            t = b;
            b = c;
            c = t;
        }
        if (a > b) {
            t = a;
            a = b;
            b = t;
        }
        System.out.print("The sorted order is : ");
        System.out.println(a + ", " + b + ", " + c);
    }
}
```

DISSECTION OF THE **SortInput** PROGRAM

❑ `int a, b, c, t;`

This program declares four integer variables. The variables **a**, **b**, and **c** are inputs to be sorted, and **t** is to be used for temporary purposes, as described in the pseudocode.

❑ `System.out.println("type three integers:");`

This line is used to *prompt* the user to type the three numbers to be sorted. Whenever a program is expecting the user to do something, it should print out a prompt telling the user what to do.

```
❏ a = Console.in.readInt();  
  b = Console.in.readInt();  
  c = Console.in.readInt();
```

The method call expression `Console.in.readInt()` is used to obtain the input from the keyboard. Three separate integers need to be typed. The values read will be stored in the three variables.

```
❏ if (a > b) {  
    t = a;  
    a = b;  
    b = t;  
}  
  if (b > c) {  
    t = b;  
    b = c;  
    c = t;  
  }  
  if (a > b) {  
    t = a;  
    a = b;  
    b = t;  
  }
```

The `if` statements and resulting assignments are Java notation for the same actions described in the sort flow chart. To comprehend these actions you need to understand why the interchange or swapping of values between two variables, such as `a` and `b`, requires the use of the temporary `t`. Also note how the three assignments are grouped as a block, allowing each `if` expression to control a group of actions.

```
❏ System.out.print("The sorted order is : ");  
  System.out.println(a + ", " + b + ", " + c);  
}
```

If the input values were 10, 5, and 15, the output would be

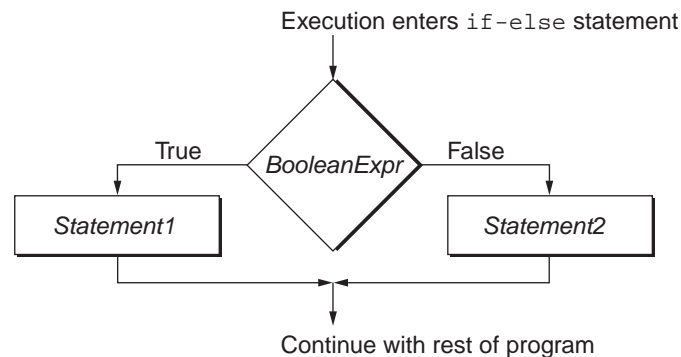
The sorted order is : 5, 10, 15

3.4 THE if-else STATEMENT

Closely related to the if statement is the if-else *statement*. An if-else statement has the following general form:

```
if ( BooleanExpr ) Statement1 else Statement2
```

If the expression *BooleanExpr* is true, then *Statement1* is executed and *Statement2* is skipped; if *BooleanExpr* is false, then *Statement1* is skipped and *Statement2* is executed. *Statement2* is called the else statement. After the if-else statement has been executed, control passes to the next statement. The flow of execution branches and then rejoins, as shown in the following diagram.



Consider the following code:

```
if (x < y)
    min = x;
else
    min = y;
System.out.println("min = " + min);
```

If $x < y$ is true, then *min* will be assigned the value of *x*; if it is false, then *min* will be assigned the value of *y*. After the if-else statement is executed, *min* is printed.

As with the if statement, either branch of an if-else statement can contain a block, as shown in the following example.

```
if (temperature < 32) {
    System.out.println("Warning Warning Warning!");
    System.out.println(32 - temperature + "(F) below Freezing!");
    System.out.println("Warning Warning Warning!");
}
else {
    System.out.println("It's " + temperature +
        "degrees fahrenheit.");
}
```

COMMON PROGRAMMING ERROR

A semicolon is used to change an expression to an expression statement. All statements do not end in semicolons, and extraneous semicolons can cause subtle errors. Look carefully at the following code fragment. What is printed when it executes if *x* is 3 and *y* is 4?

```
if (x < y);  
    System.out.println("The smaller is " + x);  
if (y < x);  
    System.out.println("The smaller is " + y);
```

The answer is

```
The smaller is 3  
The smaller is 4
```

Note the extra semicolon after the close parenthesis in each `if` statement. The indentation is misleading: All four lines should be indented the same to reflect what is actually going to happen. The true branch of each `if` statement is the empty statement `;`. The example code is syntactically correct but semantically incorrect.

Another common error is to be misled by indentation and to try to include two statements in one branch without using a block. Consider this example.

```
if (temperature < 32)  
    System.out.print("It is now");  
    System.out.print(32 - temperature);  
    System.out.println(" below freezing.");  
System.out.println("It's " + temperature + "degrees");
```

A user might mistakenly think that, if *temperature* is 32 or above, the code will execute only the last print statement displaying *temperature*. That is almost certainly what was intended. Unfortunately, if *temperature* is 32 or above, only the first `print()` would be skipped. If *temperature* is 45 the following confusing message would be printed:

```
-13 below freezing.  
It's 45 degrees
```


The then statement for the outer if-else statement is a block containing another if-else statement.

COMMON PROGRAMMING ERROR

In algebra class you came across expressions such as $18 \leq \text{age} < 65$. This expression tempts many new programmers to write

```
if (18 <= age < 65) ...
```

In evaluating the expression, Java first evaluates $(18 \leq \text{age})$ —let's call it *part1*—which is either true or false. It then tries to evaluate $(\text{part1} < 65)$, which is an error because relational operators are used to compare two numbers, not a boolean and a number. The only good thing about this type of mistake is that the compiler will catch it. Our example earlier for printing full fare adult showed the proper way to handle this situation.

3.4.2 if-else-if-else-if ...

In the preceding example, the nesting was all done in the then statement part of the if-else statement. Now we nest in the else statement part of the if-else statement:

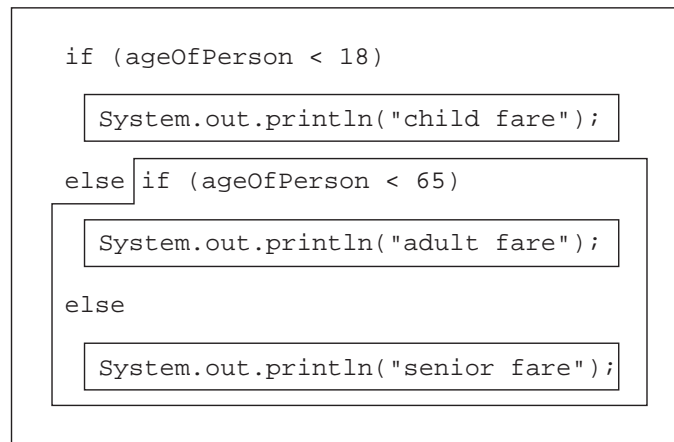
```
if (ageOfPerson < 18)
    System.out.println("child fare");
else {
    if (ageOfPerson < 65)
        System.out.println("adult fare");
    else
        System.out.println("senior fare");
}
```

The braces are not needed; we added them only for clarity. This form is so common that experienced programmers usually drop the braces. In addition, the else and the following if are usually placed on the same line, as in

```
if (ageOfPerson < 18)
    System.out.println("child fare");
else if (ageOfPerson < 65)
    System.out.println("adult fare");
else
    System.out.println("senior fare");
```

Note that the second if statement is a *single* statement that constitutes the else branch of the first statement. The two forms presented are equivalent. You should be sure that you understand why they are. If you need to, look back at the general form of the if statement and recall that an entire if-else statement is a statement itself. This fact is

illustrated in the following figure, wherein each statement is surrounded by a box. As you can see, there are five different statements.



Sometimes this chain of `if-else-if-else-if-else...` can get rather long, tedious, and inefficient. For this reason, a special construct can be used to deal with this situation when the condition being tested is of the right form. If you want to do different things based on distinct values of a *single* expression, then you can use the `switch` statement, which we discuss in [Section 3.8 The switch Statement](#).

3.4.3 The Dangling else Problem

When you use sequences of nested `if-else` statements, a potential problem can arise as to what `if` an `else` goes with, as for example in

```
if (Expression1)
    if (Expression2)
        Statement1
else
    Statement2
```

The indenting suggests that *Statement2* is executed whenever *Expression1* is false; however, that isn't the case. Rather, *Statement2* is executed only when *Expression1* is true and *Expression2* is false. The proper indenting is

```
if (Expression1)
    if (Expression2)
        Statement1
    else
        Statement2
```

The rule used in Java is that an `else` is always matched with the nearest preceding `if` that doesn't have an `else`. To cause *Statement2* to be executed whenever *Expression1* is false, as suggested by the first indenting example, you can use braces to group the statements as shown.

```
if (Expression1) {  
    if (Expression2)  
        Statement1  
}  
else  
    Statement2
```

The braces are like parentheses in arithmetic expressions that are used to override the normal precedence rules. The nested, or inner, `if` statement is now inside a block that prevents it from being matched with the `else`.



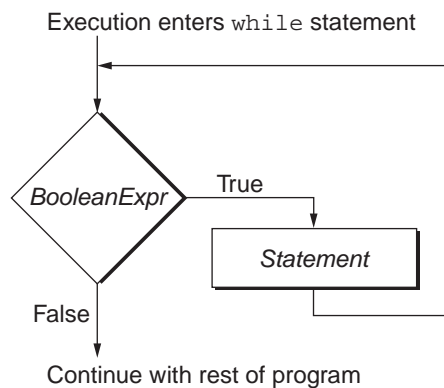
3.5 THE `while` STATEMENT

We have added the ability to choose among alternatives, but our programs still execute each instruction at most once and progress from the top to the bottom. The *while statement* allows us to write programs that run repeatedly.

The general form of a `while` statement in Java is

```
while ( BooleanExpr )  
    Statement
```

Statement is executed repeatedly as long as the expression *BooleanExpr* is true, as shown in the following flowchart.



This flowchart is the same as the one for the `if` statement, with the addition of an arrow returning to the test box.

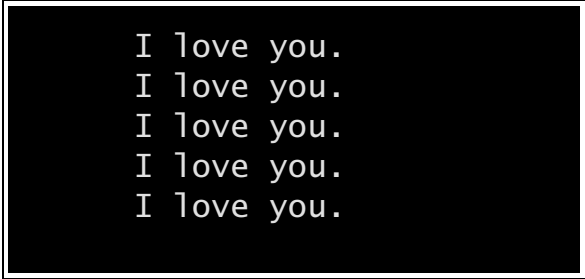
Note that *Statement* may not be executed. That will occur when *BooleanExpr* evaluates to false the first time. Note also, that like the description of the `if-else` statement, there are no semicolons in the general form of the `while` statement. The semicolon, if any, would be part of the statement that follows the parenthesized boolean expression.

A simple example for use on Valentine's Day is

```
// Valentine.java - a simple while loop
class Valentine {
    public static void main(String[] args) {
        int howMuch = 0;

        while (howMuch++ < 5)
            System.out.println("I love you.");
    }
}
```

The output is



```
I love you.
I love you.
I love you.
I love you.
I love you.
```

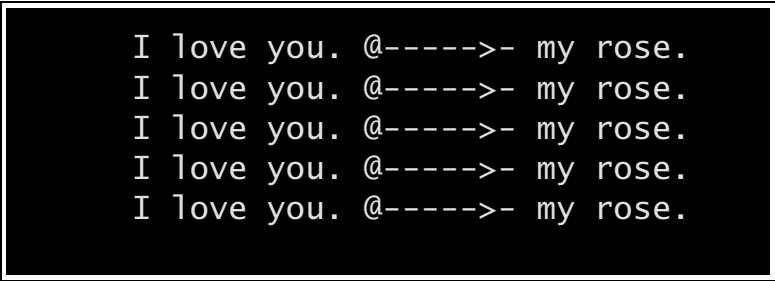
When the body of the loop is a block you get

```
while ( BooleanExpr ) {
    Statement1
    Statement2
    ...
}
```

Modifying our earlier example gives

```
int howMuch = 0;
while (howMuch++ < 5) {
    System.out.print("I love you.");
    System.out.println {" @----->- my rose."};
}
```

which results in the output



```
I love you. @----->- my rose.
I love you. @----->- my rose.
I love you. @----->- my rose.
I love you. @----->- my rose.
I love you. @----->- my rose.
```

Most while statements are preceded by some initialization statements. Our example initialized the loop counting variable `howMuch`.

3.5.1 Problem Solving with Loops

Suppose that you wanted to have a program that could read in an arbitrary number of nonzero values and compute the average. If you were to use pseudocode, the program might look like the following.

PSEUDOCODE FOR AVERAGE, USING `goto`

1. Get a number.
2. If the number is 0 go to step 6.
3. Add the number to the running total.
4. Increment the count of numbers read in.
5. Go back to step 1.
6. Divide the running total by the count of numbers in order to get the average.
7. Print the average.

In this pseudocode, step 5 goes back to the beginning of the instruction sequence, forming a loop. In the 1950s and 1960s many programming languages used a *goto statement* to implement step 5 for coding such a loop. These `goto` statements resulted in programs that were hard to follow because they could jump all over the place. Their use was sometimes called *spaghetti code*. In Java there is no `goto` statement. Java includes `goto` as a keyword that has no use so that the compiler can issue an error message if it is used inadvertently by a programmer familiar with C or C++. Today *structured control constructs* can do all the good things but none of the bad things that you could do with `goto` statements.

A *loop* is a sequence of statements that are to be repeated, possibly many times. The preceding pseudocode has a *loop* that begins at step 1 and ends at step 5. Modern programs require that you use a construct to indicate explicitly the beginning and the end of the loop. The structured equivalent, still in pseudocode, might look like the following.

PSEUDOCODE FOR AVERAGE, WITHOUT USING `goto`

```
get a number
while the number is not 0 do the following:
    add the number to the running total
    increment the count of numbers read in
    get a number
(when the loop exits)
divide the running total by the count of numbers to
    get the average
print the average
```

The loop initialization in this case is reading in the first number. Somewhere within a loop, typically at the end, is something that prepares the next iteration of the loop—getting a new number in our example. So, although not part of the required syntax, `while` statements generally look like the following

```

Statementinit      // initialization for the loop
while ( BooleanExpr ) {
    Statement1
    Statement2
    ...
    Statementnext // prepare for next iteration
}

```

The while statement is most often used when the number of iterations isn't known in advance. This situation might occur if the program was supposed to read in values, processing them in some way until a special value called a *sentinel* was read in. Such is the case for our “compute the average” program: It reads in numbers until the sentinel value 0 is read in.

We can now directly translate into Java the pseudocode for computing an average.

```

// Average.java - compute average of input values
import tio.*;

public class Average {
    public static void main(String[] args) {
        double number;
        int count = 0;
        double runningTotal = 0;
        // initialization before first loop iteration
        System.out.println("Type some numbers, " +
            "the last one being 0");
        number = Console.in.readDouble();

        while (number != 0) {
            runningTotal = runningTotal + number;
            count = count + 1;
            // prepare for next iteration
            number = Console.in.readDouble();
        }
        System.out.print("The average of the ");
        System.out.print(count);
        System.out.print(" numbers is ");
        System.out.println(runningTotal / count);
    }
}

```

DISSECTION OF THE **Average** PROGRAM

```
❑ double number;  
  int count = 0;  
  double runningTotal = 0;
```

The variable **number** is used to hold the floating point value typed in by the user. No initial value is given in the declaration because it gets its initial value from user input. The variable **count** is used to count the number of nonzero values read in. Counting is done efficiently and accurately with whole numbers, so the variable **count** is an **int** initialized to zero. The variable **runningTotal** is used to accumulate the sum of the numbers read in so far and is initialized to zero. Similar to assignment statements, these are declaration statements with initial values. Later we show how declaration statements can be used but assignment statements can't.

```
❑ System.out.println("Type some numbers, " +  
  "the last one being 0");  
  number = Console.in.readDouble();
```

A message should always be printed to prompt the user when the user is expected to enter data. We could have initialized **number** when it was declared, but it is central to the loop so we initialized it just before entering the loop. These two statements correspond to the first line of our pseudocode. All the preceding code is supporting syntax required by Java.

```
❑ while (number != 0) {
```

The loop will continue *while* the value stored in the variable **number** is not 0. The value 0 is used to detect the end of the loop. It acts as a *sentinel* to keep the loop from running forever. A *sentinel* is an important technique for terminating loops correctly.

```
❑     runningTotal = runningTotal + number;  
      count = count + 1;  
      number = Console.in.readDouble();  
  }
```

The first statement in the loop body would look strange in a mathematics class unless **number** was 0. Recall that the symbol **=** is not an equals sign; it is the assignment operator. It means: Evaluate the expression on the right and then assign that value to the variable on the left. The result in this case is to add the value of **number** to the value of **runningTotal** and store the result in **runningTotal**. This type of expression is common in computer programs: It computes a new value for a variable, using an expression that includes the old value of the variable. Similarly, we increment the value of **count**. The last statement in the body of the loop gets ready for the next

iteration by reading in a new value from the user and storing it in `number`. This action overwrites the old value, which is no longer needed. At this point the computer will go back and evaluate the boolean expression to determine whether the loop should execute again (i.e., `number` isn't 0) or continue with the rest of the program.

❑ `System.out.println(runningTotal / count);`

When the user finally types a 0, the program prints the answer and exits. We have already used the `print()` and `println()` methods with a string and with a number. Here we show that the number can be an expression. Recall that the symbol `/` is the symbol for division. If the first number typed is 0, this program will terminate by printing

The average of the 0 numbers is NaN

The symbol NaN stands for *not a number* and results from dividing zero by zero. A better solution might be to test, using an `if-else` statement, for this special case and print a more appropriate message. We leave that for you to do as an exercise.

3.6 THE for STATEMENT

You can use the `while` construct to create any loop you will ever need. However, Java, like most modern programming languages, provides two alternative ways to write loops that are sometimes more convenient. The *for statement* is a looping statement that captures the initialization, termination test, and iteration preparation all in one place at the top of the loop. The general form of the *for statement* is

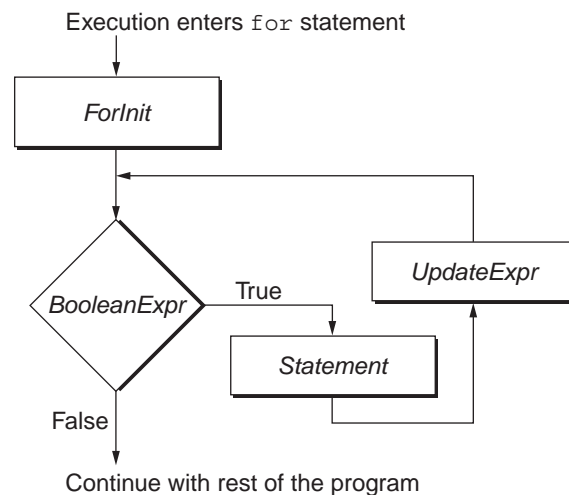
```
for ( ForInit; BooleanExpr ; UpdateExpr )  
    Statement
```

As part of the syntax of the *for statement*, the semicolons are used to separate the three expressions. Note the absence of a semicolon between *UpdateExpr* and the closing parenthesis.

EXECUTION OF A FOR STATEMENT

1. *ForInit* is evaluated *once*—before the body of the loop.
2. *BooleanExpr* is tested *before each iteration* and, if true, the loop continues (as in the `while` statement).
3. The loop body *Statement* is executed.
4. *UpdateExpr* is evaluated at the *end of each iteration*.
5. Go to step 2.

The following diagram also shows the flow of execution.



Compare this general form and the flow with that of the `while` statement. In the `for` statement, the initialization expression represented by *ForInit* and the iteration expression represented by *UpdateExpr* frequently involve the use of an assignment operator, but one is not required.

The `for` statement applies most naturally to situations involving going around a loop a specific number of times. For example, if you wanted to print out the square roots of the numbers 1 through 10 you could do so as follows.

```
// SquareRoots.java - print square roots of 1 - 10
public class SquareRoots {
    public static void main(String[] args) {
        int i;
        double squareRoot;

        for (i = 1; i <= 10; i++) {
            squareRoot = Math.sqrt(i);
            System.out.println("the square root of " + i +
                               " is " + squareRoot);
        }
        System.out.println("That's All!");
    }
}
```

DISSECTION OF THE **SquareRoots** PROGRAM

❑ `for (i = 1; i <= 10; i++) {`

The expression `i = 1` is evaluated only once—before the loop body is entered. The expression `i <= 10` is evaluated before *each* execution of the loop body. The expression `i++` is evaluated at the *end of each* loop body evaluation. Recall that `i++` is shorthand for `i = i + 1`.

❑

```
    squareRoot = Math.sqrt(i);
    System.out.println("the square root of " + i +
                      " is " + squareRoot);
}
```

The body of the loop is one statement—in this case a block. The braces are part of the block syntax, not part of the syntax of a `for` statement.

❑ `System.out.println("That's All!");`

When the loop exits, program execution continues with this statement.

We can redo the square root printing loop by using a `while` statement as follows.

```
// SquareRoots2.java - replace for with while
public class SquareRoots2 {
    public static void main(String[] args) {
        int i;
        double squareRoot;
        i = 1;                      // initialization-expr
        while (i <= 10) {
            squareRoot = Math.sqrt(i);
            System.out.println("the square root of " + i +
                              " is " + squareRoot);
            i++;                    // iteration-expr
        }
        System.out.println("That's All!");
    }
}
```

In both versions of this program, the same sequence of steps occurs.

USING LOOPS

1. The variable `i` is initialized to 1.
2. The boolean expression `i <= 10` is tested, and if true, the loop body is executed. If the expression is false, the loop statement is completed and execution continues with the next statement in the program, the final print statement.
3. After executing the print statement within the loop, the statement incrementing the variable `i` is executed, in preparation for the next loop iteration.
4. Execution continues with step 2.

3.6.1 Local Variables in the for Statement

The *ForInit* part of the for statement can be a local declaration. For example, let's write a loop that reads characters and prints them as uppercase.

```
char c;
for (int i = 1; i <= 10; i++) {
    c = (char)Console.in.readChar();
    if ( c >= 'a' && c <= 'z')
        c = (char)(c - ('a' - 'A'));
    System.out.print(c);
}
```

In this case, the `int` variable `i` declared in the for statement is limited to the for statement, including the body of the loop. The variable's existence is tied to the for statement code; outside the for statement it disappears. If you want the variable `i` to exist independently of the for statement you must declare it before the for statement. It then is available throughout the block containing the for statement. The variable is said to have *local scope*. We discuss scope further in [Section 4.4 Scope of Variables](#).

In the preceding example, we used a property of the character codes for the English alphabet. The property is that the difference between any lowercase letter and its uppercase equivalent is always the same. You compute that difference with the expression `('a' - 'A')`. You could just as easily use the expression `('z' - 'Z')` or any other pair of matching uppercase and lowercase letters. Also, recall that when you use `Console.in.readChar()` from `tio` it returns an `int` value and hence a cast is needed. When you use `(c - ('a' - 'A'))` a cast to `char` is necessary because the resulting expression is converted to the wider `int` type.



3.7 THE break AND continue STATEMENTS

Two special statements,

`break;` `and` `continue;`

interrupt the normal flow of control. The `break` statement causes an exit from the innermost enclosing loop. The `break` statement also causes a `switch` statement to terminate. (We discuss the `switch` statement in [Section 3.8 The `switch` Statement](#).) In the following example, a test for a negative argument is made. If the test is true, then a `break` statement is used to pass control to the statement immediately following the loop.

```
while (true) {                                //seemingly an infinite loop
    System("Enter a positive integer:");
    n = Console.in.readInt();
    if (n < 0)
        break;                                // exit loop if n is negative
    System.out.print("squareroot of " + n);
    System.out.println(" = " + Math.sqrt(n));
}

// break jumps here
```

This use of the `break` statement is typical. What would otherwise be an infinite loop is made to terminate upon a given condition tested by the `if` expression.

The `continue` statement causes the current iteration of a loop to stop and causes the next iteration of the loop to begin immediately. The following code adds `continue` to the preceding program fragment in order to skip processing of negative values.

```
// BreakContinue.java - example of break and continue
import tio.*;

class BreakContinue {
    public static void main(String[] args) {
        int n;

        while (true) {                        //seemingly an infinite loop
            System.out.print("Enter a positive integer ");
            System.out.print("or 0 to exit:");
            n = Console.in.readInt();
            if (n < 0)
                continue;
        }
    }
}
```

```

        if (n == 0)
            break;           // exit loop if n is 0
        if (n < 0)
            continue;        //wrong value
        System.out.print("squareroot of " + n);
        System.out.println(" = " + Math.sqrt(n));
        //continue lands here at end of current iteration
    }
    //break lands here
    System.out.println("a zero was entered");
}
}

```

The output of this program, assuming that the user enters the values 4, 21, 9, and 0, is

```

os-prompt>java BreakContinue
Enter a positive integer or 0 to exit:4
squareroot of 4 = 2.0
Enter a positive integer or 0 to exit:-1
Enter a positive integer or 0 to exit:9
squareroot of 9 = 3.0
Enter a positive integer or 0 to exit:0
a zero was entered
os-prompt>

```

The continue statement may only occur inside for, while, and do loops. As the example shows, continue transfers control to the end of the current iteration, whereas break terminates the loop.

The break and continue statements can be viewed as a restricted form of a goto statement. The break is used to go to the statement following the loop and the continue is used to go to the end of the current iteration. For this reason, many programmers think that break and continue should be avoided. In fact, many uses of break and continue can be eliminated by using the other structured control constructs. For example, we can redo the BreakContinue example, but without break and continue, as follows.

```

// NoBreakContinue.java - avoiding break and continue
import tio.*;

class NoBreakContinue {
    public static void main(String[] args) {
        int n;
    }
}

```

```

        System.out.print("Enter a positive integer ");
        System.out.print("or 0 to exit:");
        n = Console.in.readInt();
        while (n != 0) {
            if (n > 0) {
                System.out.print("squareroot of " + n);
                System.out.println(" = " + Math.sqrt(n));
            }
            System.out.print("Enter a positive integer ");
            System.out.print("or 0 to exit:");
            n = Console.in.readInt();
        }

        System.out.println("a zero was entered");
    }
}

```

The loop termination condition is now explicitly stated in the `while` statement and not buried somewhere inside the loop. However, we did have to repeat the prompt and the input statement—once before the loop and once inside the loop. We eliminated `continue` by changing the `if` statement to test for when the square root could be computed instead of testing for when it could not be computed.



3.8 THE switch STATEMENT

The `switch` statement can be used in place of a long chain of `if-else-if-else-if-else` statements when the condition being tested evaluates to an integer numeric type. Suppose that we have an integer variable `dayOfWeek` that is supposed to have a value of 1 through 7 to represent the current day of the week. We could then print out the day as

```


if (dayOfWeek == 1)
    System.out.println("Sunday");
else if (dayOfWeek == 2)
    System.out.println("Monday");
else if (dayOfWeek == 3)
    System.out.println("Tuesday");
else if (dayOfWeek == 4)
    System.out.println("Wednesday");
else if (dayOfWeek == 5)
    System.out.println("Thursday");
else if (dayOfWeek == 6)
    System.out.println("Friday");
else if (dayOfWeek == 7)
    System.out.println("Saturday");
else
    System.out.println("Not a day number " + dayOfWeek);

```

An alternative is to use a `switch` statement as follows.


```
switch (dayOfWeek) {  
    case 1:  
        System.out.println("Sunday");  
        break;  
    case 2:  
        System.out.println("Monday");  
        break;  
    case 3:  
        System.out.println("Tuesday");  
        break;  
    case 4:  
        System.out.println("Wednesday");  
        break;  
    case 5:  
        System.out.println("Thursday");  
        break;  
    case 6:  
        System.out.println("Friday");  
        break;  
    case 7:  
        System.out.println("Saturday");  
        break;  
    default:  
        System.out.println("Not a day number " + dayOfWeek);  
}
```

Unlike the `if-else` and looping statements described earlier in this chapter, the braces are part of the syntax of the `switch` statement. The controlling expression in parentheses following the keyword `switch` must be of integral type. Here it is the `int` variable `dayOfWeek`. After the expression has been evaluated, control jumps to the appropriate case label. All the constant integral expressions following the case labels must be unique. Typically, the last statement before the next case or `default` label is a `break` statement. If there is no `break` statement, then execution “falls through” to the next statement in the succeeding case. Missing `break` statements are a frequent cause of error in `switch` statements. For example, if the `break;` was left out of just case 1 and if `dayOfWeek` was 1, the output would be



Sunday
Monday

instead of just



Sunday

There may be at most one `default` label in a `switch` statement. Typically, it occurs last, although it can occur anywhere. The keywords `case` and `default` can't occur outside a `switch`.

Taking advantage of the behavior when `break`; is not used, you can combine several cases as shown in the following example.

```
switch (dayOfWeek) {  
    case 1:  
    case 7:  
        System.out.println("Stay home today!");  
        break;  
    case 2:  
    case 3:  
    case 4:  
    case 5:  
    case 6:  
        System.out.println("Go to work.");  
        break;  
    default:  
        System.out.println("Not a day number " + dayOfWeek);  
        break;  
}
```

Note that the `break` statement—not anything related to the case labels—causes execution to continue after the `switch` statement. The `switch` statement is a structured `goto` statement. It allows you to go to one of several labeled statements. It is then combined with the `break` statement, which does another `goto`, in this case “go to the statement after the `switch`.”

THE EFFECT OF A `switch` STATEMENT

1. Evaluate the `switch` expression.
2. Go to the case label having a constant value that matches the value of the expression found in step 1; or, if a match is not found, go to the `default` label; or, if there is no `default` label, terminate `switch`.
3. Continue executing statements in order until the end of `switch` is reached or a `break` is encountered.
4. Terminate `switch` when a `break` statement is encountered, or terminate `switch` by “falling off the end.”



3.9 USING THE LAWS OF BOOLEAN ALGEBRA

Boolean expressions can frequently be simplified or more efficiently evaluated by converting them to logically equivalent expressions. Several rules of Boolean algebra are useful for rewriting boolean expressions.

Laws of Boolean Algebra	
Commutative law	<i>a or b equals b or a</i> <i>a and b equals b and a.</i>
Distributive <i>and</i> law	<i>a and (b or c) equals</i> <i>(a and b) or (a and c).</i>
Distributive <i>or</i> law	<i>a or (b and c) equals</i> <i>(a or b) and (a or c).</i>
Double negation	<i>not not a equals a.</i>
DeMorgan's laws	<i>not(a and b) equals (nota or notb)</i> <i>not(a or b) equals (not a and not b)</i>

In the expression `x || y`, `y` will not be evaluated if `x` is true. The value of `y` in that case doesn't matter. For this reason, it may be more efficient to have the first argument of a boolean *or* expression be the one that most often evaluates to true. This is the basis for short-circuit evaluation of the logical *or* operator.

```
while (a < b || a == 200) {
    ...
}
while (a == 200 || a < b) {
    ...
}
```

For the two controlling expressions, which order is more likely to be efficient? Note that by the commutative law both `while` conditions are equivalent.



3.10 PROGRAMMING STYLE

We follow Java professional style throughout the programming examples. Statements should readily display the flow of control of the program and be easy to read and follow. Only one statement should appear on a line. Indentation should be used consistently to set off the flow of control. After a left brace, the next line should be indented, showing that it is part of that compound statement. There are two conventional brace styles in Java. The style we follow in this book is derived from the C and C++ professional programming community. In this style, an opening or left brace stays on the same line as the beginning of a selection or iteration statement. The closing or right brace lines up under the keyword that starts the overall statement. For example,

```
if (x > y) {
    System.out.println("x is larger " + x);
    max = x;
}
```

```
while (i < 10) {
    sum = sum + i;
    i++;
}
```

An alternative acceptable style derives from the Algol60 and Pascal communities. In this style each brace is kept on a separate line. In those languages the keywords `begin` and `end` were used to set off compound statements and they were placed on their own lines. For example,

```
if ( x > y)
{
    System.out.println("x is larger " + x);
    max = x;
}
```

However, be consistent and use the same style as others in your class, group, or company. A style should be universal within a given community. A single style simplifies exchanging, maintaining, and using each others' code.



SUMMARY

- ❑ An expression followed by a semicolon is an expression statement. Expression statements are the most common form of statement in a program. The simplest statement is the empty statement, which syntactically is just a semicolon.
- ❑ A group of statements enclosed by braces is a block. A block can be used anywhere a statement can be used. Blocks are important when you need to control several actions with the same condition. This is often the case for selection or iteration statements such as the `if-else` statement or the `for` statement, respectively.
- ❑ All statements don't end with a semicolon. The only statements covered in this chapter that end with a semicolon are expression statements and declaration statements.
- ❑ The general form of an `if` statement is

```
if ( BooleanExpr )
    Statement
```

- ❑ The general form of an `if-else` statement is

```
if ( BooleanExpr )
    Statement1
else
    Statement2
```

- ❑ When nesting an `if` statement inside an `if-else` statement or vice-versa, the `else` will always be matched with the closest unmatched `if`. This precedence can be

overridden with a block statement to enclose a nested `if` statement or `if-else` statement.

- ❑ The general form of the `while` statement is

```
while ( BooleanExpr )  
    Statement
```

- ❑ The general form of the `for` statement is

```
for ( ForInit ; BooleanExpr ; UpdateExpr )  
    Statement
```

- ❑ Java includes the usual logical operators *and*, *or*, and *not*.
- ❑ The `break` statement can be used to terminate abruptly the execution of either a `switch` statement or a looping statement. When a `break` is executed inside any of those statements, the enclosing `switch` or loop statement is immediately terminated and execution continues with the statement following the `switch` or loop.
- ❑ The `continue` statement is used only inside a looping statement. When a `continue` is executed, the remaining portion of the surrounding loop body is skipped and the next iteration of the loop is begun. In the case of the `while` statement, the loop termination expression is tested as the next operation to be performed after the `continue`. In the case of the `for` statement, the update expression is evaluated as the next operation, followed by the loop termination test.
- ❑ The `switch` statement is an alternative to a sequence of `if-else-if-else...` statements. The `switch` statement can be used only when the selection is based on an integer-valued expression.



REVIEW QUESTIONS

1. True or false? An expression is a statement.
2. How do you turn an expression into a statement?
3. True or false? All statements end with a semicolon. If false, give an example to show why.
4. True or false? An `if` statement is always terminated with a semicolon. If false, give an example to show why.
5. What are the values of the following Java expressions?

```
true && false  
true || false
```

6. Write a Java expression that is true whenever the variable `x` is evenly divisible by both 3 and 5. Recall that `(x % y)` is zero if `y` evenly divides `x`.
7. What is printed by the following program fragment?

```

x = 10;
y = 20;
if ( x < y)
    System.out.println("then statement executed");
else
    System.out.println("else statement executed");
    System.out.println("when is this executed?");

```

8. What is printed by the following program, if 3 and 12 are entered as x and y? Now if you change the order and enter 12 and 3, what is printed?

```

// PrintMin.java - print the smaller of two numbers
import tio.*;

class PrintMin {
    public static void main(String[] args) {
        System.out.println("Type two integers.");
        int x = Console.in.readInt();
        int y = Console.in.readInt();

        if (x < y)
            System.out.println("The smaller is " + x);
        if (y < x)
            System.out.println("The smaller is " + y);
        if (x == y)
            System.out.println("They are equal.");
    }
}

```

9. For the declarations shown, fill in the value of the expression or enter **illegal**.

```
int  a = 2, b = 5, c = 0, d = 3;
```

Expression	Value
b % a	
a < d	
(c != b) && (a > 3)	
a / b > c	
a * b > 2	

10. What is printed by the following program fragment? How should it be indented to reflect what is really going on?

```
x = 10;
y = 20;
z = 5;
if ( x < y)
if ( x < z)
System.out.println("statement1");
else
System.out.println("statement2");
System.out.println("statement3");
```

11. How many times will the following loop print testing?

```
int i = 10;
while (i > 0) {
    System.out.println("testing");
    i = i - 1;
}
```

12. How many times will the following loop print testing?

```
int i = 1;
while (i != 10) {
    System.out.println("testing");
    i = i + 2;
}
```

13. What is printed by the following loop? See [Review Question 6](#).

```
int i = 1;
while (i <= 100) {
    if (i % 13 == 0)
        System.out.println(i);
    i = i + 1;
}
```

14. Rewrite the loop in the previous question using a for statement.

15. Rewrite the following loop, using a while statement.

```
for (i = 0; i < 100; i++) {
    sum = sum + i;
}
```

16. True or false? Anything that you can do with a while statement can also be done with a for statement and vice-versa.

17. How many times will the following loop go around? This is a trick question—look at the code fragment carefully. It is an example of a common programming error.

```
int i = 0;
while (i < 100) {
    System.out.println(i*i);
}
```

18. What is printed by the following program?

```
// Problem18.java
class Problem18 {
    public static void main(String[] args) {
        int i, j = 0;

        for (i = 1; i < 6; i++)
            if (i > 4)
                break;
            else {
                j = j + i;
                System.out.println("j= " + j + " i= " + i);
            }
        System.out.println("Final j= " + j + " i= " + i);
    }
}
```



EXERCISES

1. Write a program that asks for the number of quarters, dimes, nickels, and pennies you have. Then compute the total value of your change and print the amount in the form \$X.YY. See [Exercise 16 in Chapter 2 Program Fundamentals](#).
2. Write a program that reads in two integers and then prints out the larger one. Use [Review Question 8](#), as a starting point and make the necessary changes to class `PrintMin` to produce class `PrintMax`.
3. Modify the class `Average` from this chapter to print a special message if the first number entered is 0.
4. Write a program that reads in four integers and then prints out `yes` if the numbers were entered in increasing order and prints out `no` otherwise.
5. Write a program that prompts for the length of three line segments as integers. If the three lines could form a triangle, the program prints "Is a triangle." Otherwise, it prints "Is not a triangle." Recall that the sum of the lengths of *any* two sides of a triangle must be greater than the length of the third side. For example, 20, 5, and 10 can't be the lengths of the sides of a triangle because $5 + 10$ is not greater than 20.
6. Write a program that tests whether the formula $a^2 + b^2 = c^2$ is true for three integers entered as input. Such a triple is a *Pythagorean triple* and forms a right-angle triangle with these numbers as the lengths of its sides.

7. An operator that is mildly exotic is the conditional operator `? :`. This operator takes three arguments and has precedence just above the assignment operators, as for example in

```
s = (a < b)? a : b;  
// (a < b) true then s assigned a else s assigned b
```

Rewrite the code for class `PrintMin`, in [Review Question 8](#), using this operator to eliminate the first two `if` statements. We chose not to use this operator because it is confusing to beginning programmers. It is unnecessary and usually can readily and transparently be replaced by an `if` statement.

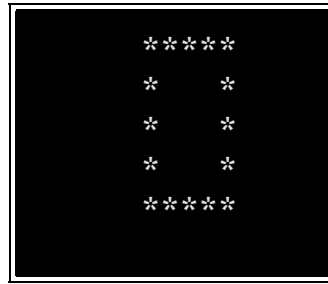
8. Write a program that reads in integers entered at the terminal until a value of 0 is entered. A *sentinel* value is used in programming to detect a special condition. In this case the sentinel value is used to detect that no more data values are to be entered. After the sentinel is entered, the program should print out the number of numbers that were greater than 0 and the number of numbers that were less than 0.
9. Write a program that reads in integers entered at the terminal until a sentinel value of 0 is entered. After the sentinel is entered, the program should print out the smallest number, other than 0, that was entered.
10. Rewrite [Exercise 8](#) to print the largest number entered before the sentinel value 0 is entered. If you already did that exercise, only a few changes are needed to complete this program. Now code a further program that ends up printing both the smallest or minimum value found and the largest or maximum value found.
11. Rewrite [Exercise 5](#) to continue to test triples until the sentinel value 0 is entered.
12. Write a program that reads in characters and prints their integer values. Use the *end-of-file* value -1 as a guard value to terminate the character input. For Unix systems, the end-of-file character can be generated by hitting Ctrl+D on the keyboard. For Windows systems, the end-of-file character can be generated by hitting Ctrl+Z on the keyboard. Note again that `readChar()` returns an `int` value. This action allows you to test the end-of-file value. If you convert this value to a `char`, negative numbers would not be representable. Explain why.
13. Write a program to input values as in [Exercise 12](#). This time run it with the input taken as a file through *redirection*. If the compiled Java program is *TestChar.class*, then the command

```
java TestChar < myFile
```

will use `myFile` for input. Redirection is possible with Unix or the Windows console window.

14. Write a program to print every even number between 0 and 100. Modify the program to allow the user to enter a number *n* from 1 through 10 and have the program print every *n*th number from 0 through 100. For example, if the user enters 5, then 0 5 10 15 20...95 100 are printed.

15. Write a program that will print out a box drawn with asterisks, as shown.



Use a loop so that you can easily draw a larger box. Modify the program to read in a number from the user specifying how many asterisks high and wide the box should be.

16. Write a program that reads in numbers until the same number is typed twice in a row. Modify it to go until three in a row are typed. Modify it so that it first asks for “how many in a row should I wait for?” and then it goes until some number is typed that many times. For example, for two in a row, if the user typed “1 2 5 3 4 5 7” the program would still be looking for two in a row. The number 5 had been typed twice, but not in a row. If the user then typed 7, that would terminate the program because two 7s were typed, one directly after the other.
17. Write a program that prints all the prime numbers in 2 through 100. A prime number is an integer that is greater than 1 and is divisible only by 1 and itself. For example, 2 is the only even prime. Why?

PSEUDOCODE FOR FINDING PRIMES

```

for n = 2 until 100
  for i = 2 until the square root of n
    if n % i == 0 the number is divisible by i
      otherwise n is prime
  
```

Can you explain or prove why the inner-loop test only needs to go up to the square root of n ?

18. Write a program that prints the first 100 prime numbers. Use the same algorithm as in [Exercise 17](#), but terminate the program upon finding the 100th prime. Assume that the search needs to go no farther than $n = 10,000$.
19. Write a program that generates all Pythagorean triples (see [Exercise 6](#)) whose small sides are no larger than n . Try it with $n \leq 200$. (*Hint*: Use two for loops to enumerate possible values for the small sides and then test to determine whether the result is an integral square.
20. Write a program that gives you a different message each day of the week. Use a switch statement. Take as input an integer in the range 1 through 7. For example, if 6 means Friday, the message might say, Today is Friday, tGif. If the user inputs a number other than 1 through 7, have the default issue an appropriate message.
21. Write a program that gives you a fortune based on an astrological sign. Use a switch statement to structure the code.

22. Write a program that generates an approximation of the real number e . Use the formula

$$e \approx 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots + \frac{1}{k!} + \dots$$

where $k!$ means k factorial $= 1 * 2 * \dots * k$. Keep track of term $1/k!$ by using a double. Each iteration should use the previous value of this term to compute the next term, as in

$$T_{k+1} = T_k \times \frac{1}{k+1}$$

Run the computation for 20 terms, printing the answer after each new term is computed.

23. Write your own pseudorandom number generator. A pseudorandom sequence of numbers appears to be chosen at random. Say that all the numbers you are interested in are placed in a large fishbowl and you reach in and pick out one at a time without looking where you are picking. After reading the number, you replace it and pick another. Now you want to simulate this behavior in a computation. You can do so by using the formula $X_{n+1} = (aX_n + c) \bmod m$. Let a be 3,141,592,621, c be 1, and m be 10,000,000,000 (see Knuth, *Seminumerical Algorithms*, Addison-Wesley 1969, p. 86). Generate and print the first 100 such numbers as long integers. Let $X_1 = 1$.



APPLET EXERCISE

Redo any one of the first ten exercises in this chapter but use an applet for input and output. You are to do so by modifying the following applet. Recall that the Applet Exercises in [Chapter 2, Program Fundamentals](#), introduced a special kind of Java program called an *applet*. Among other things, an applet may be used to create graphical output such as plots or diagrams. In order to be useful, most programs, including applets, need some way to receive input. For the regular applications that we've created so far, we've used `Console.in.readInt()`, etc., to read values from the console. This exercise introduces you to the use of a `TextField` object to get input from an applet. For this exercise we need to introduce two more methods: `init()` and `actionPerformed()`. As with the earlier applet exercise, for now you need only to concentrate on the body of the methods `init()` and `actionPerformed()`, treating the surrounding code as a template to be copied verbatim. The following applet reads two numbers and then displays their sum.

```
/* <applet code="AppletSum.class"
   width=420 height=100></applet> */
// AppletSum.java - Text input with an applet
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```

public class AppletSum extends JApplet
    implements ActionListener {
    JTextField inputOne = new JTextField(20);
    JTextField inputTwo = new JTextField(20);
    JTextField output = new JTextField(20);
    public void init() {
        Container pane = getContentPane();

        pane.setLayout(new FlowLayout());
        pane.add(new JLabel("Enter one number."));
        pane.add(inputOne);
        pane.add(
            new JLabel("Enter a number and hit return."));
        pane.add(inputTwo);
        pane.add(new JLabel("Their sum is:"));
        pane.add(output);
        inputTwo.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e) {
        double first, second, sum;

        first = Double.parseDouble(inputOne.getText());
        second = Double.parseDouble(inputTwo.getText());
        sum = first + second;
        output.setText(String.valueOf(sum));
    }
}

```

DISSECTION OF THE **AppletSum** PROGRAM

❑ `import java.awt.*;`
`import java.awt.event.*;`
`import javax.swing.*;`
`public class AppletSum extends JApplet`
 `implements ActionListener`

For now this code is all part of the template for an applet. The only part that you should change is to replace **AppletSum** with the name of your applet. The **import** statements tell the Java compiler where to find the various classes used in this program, such as **JTextField** and **JApplet**. The **extends JApplet** phrase is how we indicate that this class is defining an applet instead of a regular application. The **implements ActionListener** phrase is needed if we want to have our program do something when the user has finished entering data.

```
❑   JTextField inputOne = new JTextField(20);  
    JTextField inputTwo = new JTextField(20);  
    JTextField output = new JTextField(20);
```

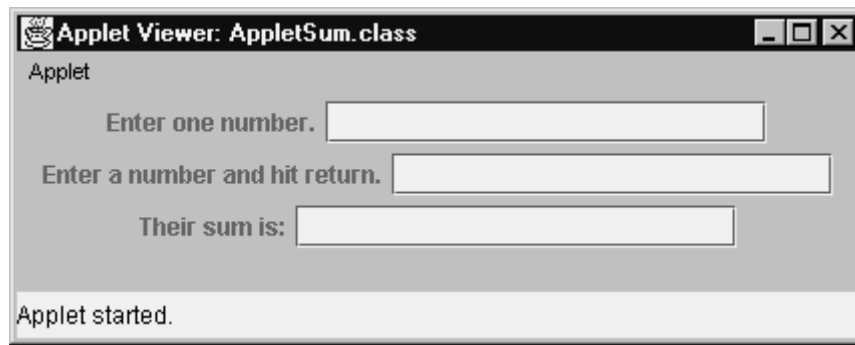
`JTextField` is a standard Java class. A `JTextField` is a graphical user interface component in which the user can enter text and the program can then read the text entered by the user. This program will create three text fields—two for the input values and one to display the sum of the first two. Note that these variables aren't declared inside any method. These variables need to be referenced by both the methods in this applet. You should think of them as part of the applet class `AppletSum` and not part of any one method. (We discuss such declarations in [Section 6.5 Static Fields and Methods](#).)

```
❑   public void init() {  
        Container pane = getContentPane();  
        pane.setLayout(new FlowLayout());
```

The method `init()` is rather like `main()` for an applet. This method is called to initialize the applet. The first thing we do in the method is get the content pane. The *content pane* is the part of the applet used to display the various graphical components that we add to the applet. The `setLayout()` method is used to adjust the content pane so that when components are added, they are arranged in lines, like words of text in a word processor. The components are allowed to “flow” from one line to the next. For now, just always include the two statements in your `init()` method.

```
❑   pane.add(new JLabel("Enter one number."));  
    pane.add(inputOne);  
    pane.add(  
        new JLabel("Enter a number and hit return."));  
    pane.add(inputTwo);  
    pane.add(new JLabel("Their sum is:"));  
    pane.add(output);
```

The method `add()` is used to add some labels and text fields to the content pane of the applet. `JLabel` is a standard Java class that is used to place labels in graphical user interfaces. Unlike with a `JTextField`, the user can't enter text in a `JLabel` when the applet is running. The order in which we add all the components to the applet is important. The components are placed in the applet, like words in a word processor. That is, the components will be displayed from left to right across the applet until the next component won't fit, in which case a new line of components is started. Later, you'll learn more about controlling the placement of components, but for now just add them in the desired order and then adjust the width of the applet to get the desired appearance. Here is what the applet will look like.



❑ `inputTwo.addActionListener(this);`

The last statement in the `init()` method says that the `actionPerformed()` method in the applet referred to by `this` should be called when Return is hit in the text field, `inputTwo`.

❑

```
public void actionPerformed(ActionEvent e) {  
    double first, second, sum;  
    first = Double.parseDouble(inputOne.getText());  
    second = Double.parseDouble(inputTwo.getText());  
    sum = first + second;  
    output.setText(String.valueOf(sum));  
}
```

This method is called each time the user hits Return with the cursor placed in the text field `inputTwo`. The method gets the text that was entered in the first two text fields by invoking their `getText()` methods. The resulting strings are then converted to floating point numbers, using the standard Java method `Double.parseDouble()`. Next the two numbers are added and stored in the variable `sum`. The floating point value stored in `sum` is converted to a `String`, using `String.valueOf()`, which can be used to convert any primitive value to a `String`. Finally the method `setText()` from the class `JTextField` is used to set the value of the text displayed in the `JTextField` object `output`.