
5 Conditionals and Loops

Chapter Objectives

- Define the flow of control through a method.
- Explore boolean expressions that can be used to make decisions.
- Perform basic decision-making using `if` statements.
- Discuss issues pertaining to the comparison of certain types of data.
- Execute statements repetitively using `while` loops.
- Discuss the concept of an iterator object and use one to read a text file.
- Introduce the `ArrayList` class.
- Explore more GUI controls and events.

All programming languages have statements that allow you to make decisions to determine what to do next. Some of those statements allow you to repeat a certain activity multiple times. This chapter discusses key Java statements of this type and explores issues related to the comparison of data and objects. It begins with a discussion of boolean expressions, which form the basis of any decision. The Graphics Track sections of this chapter explore some new controls and events.

5.1 Boolean Expressions

The order in which statements are executed in a running program is called the *flow of control*. Unless otherwise specified, the execution of a program proceeds in a linear fashion. That is, a running program starts at the first programming statement and moves down one statement at a time until the program is complete. A Java application begins executing with the first line of the `main` method and proceeds step by step until it gets to the end of the `main` method.

Invoking a method alters the flow of control. When a method is called, control jumps to the code defined for that method. When the method completes, control returns to the place in the calling method where the invocation was made, and processing continues from there.

Within a given method, we can alter the flow of control through the code by using certain types of programming statements. Statements that control the flow of execution through a method fall into two categories: conditionals and loops.

Key Concept

Conditionals and loops allow us to control the flow of execution through a method.

A **conditional statement** ⓘ is sometimes called a *selection statement*, because it allows us to choose which statement will be executed next. The conditional statements in Java are the `if` statement, the `if-else` statement, and the `switch` statement. We explore the `if` statement and the `if-else` statement in this chapter and cover the `switch` statement in **Chapter 6** 📖.

Each decision is based on a **boolean expression** ⓘ (also called a *condition*), which is an expression that evaluates to either true or false. The result of the expression determines which statement is executed next. The following is an example of an `if` statement:

```
if (count > 20)
    System.out.println("Count exceeded");
```

Key Concept

An `if` statement allows a program to choose whether to execute a particular statement.

The condition in this statement is `count > 20`. That expression evaluates to a boolean (true or false) result. Either the value stored in `count` is greater than 20 or it's not. If it is, the `println` statement is

executed. If it's not, the `println` statement is skipped and processing continues with whatever code follows it.

The need to make decisions like this comes up all the time in programming situations. For example, the cost of life insurance might be dependent on whether the insured person is a smoker. If the person smokes, we calculate the cost using a particular formula; if not, we calculate it using another. The role of a conditional statement is to evaluate a boolean condition (whether the person smokes) and then to execute the proper calculation accordingly.


A **loop** ⓘ, or **repetition statement** ⓘ, allows us to execute a programming statement over and over again. Like a conditional, a loop is based on a boolean expression that determines how many times the statement is executed.

Key Concept

A loop allows a program to execute a statement multiple times.


For example, suppose we wanted to calculate the grade point average of every student in a class. The calculation is the same for each student; it is just performed on different data. We would set up a loop

that repeats the calculation for each student until there are no more students to process.

Java has three types of loop statements: the `while` statement, the `do` statement, and the `for` statement. Each type of loop statement has unique characteristics that distinguish it from the others. We cover the `while` statement in this chapter and explore `do` loops and `for` loops in **Chapter 6** .

The boolean expressions on which conditionals and loops are based use equality operators, relational operators, and logical operators to make decisions. Before we discuss the conditional and loop statements in detail, let's explore these operators.

Equality and Relational Operators

The `==` and `!=` operators are called **equality operators**.  They test whether two values are equal or not equal, respectively. Note that the equality operator consists of two equal signs side by side and should not be mistaken for the assignment operator that uses only one equal sign.

The following `if` statement prints a sentence only if the variables `total` and `sum` contain the same value:


```
if (total == sum)

    System.out.println("total equals sum");
```

Likewise, the following `if` statement prints a sentence only if the variables `total` and `sum` do *not* contain the same value:

```
if (total != sum)

    System.out.println("total does NOT equal sum");
```

Java also has several *relational operators* that let us decide relative ordering between values. Earlier in this section, we used the greater than operator (`>`) to decide if one value was greater than another. We can ask similar questions using various operators. In Java, relational operators are greater than (`>`), less than (`<`), greater than or equal to (`>=`), and less than or equal to (`<=`). **Figure 5.1**  lists the Java equality and relational operators.


Operator	Meaning
<code>==</code>	equal to
<code>!=</code>	not equal to
<code><</code>	less than
<code><=</code>	less than or equal to
<code>></code>	greater than
<code>>=</code>	greater than or equal to

Figure 5.1 Java equality and relational operators

The equality and relational operators have precedence lower than the arithmetic operators. Therefore, arithmetic operations are evaluated first, followed by equality and relational operations. As always, parentheses can be used to explicitly specify the order of evaluation.

We'll see more examples of relational operators as we examine conditional and loop statements throughout this chapter.

Logical Operators


In addition to the equality and relational operators, Java has three *logical operators* that produce boolean results. They also take boolean operands. **Figure 5.2**  lists and describes the logical operators.

Operator	Description	Example	Result
!	logical NOT	! a	true if a is false and false if a is true
&&	logical AND	a && b	true if a and b are both true and false otherwise
	logical OR	a b	true if a or b or both are true and false otherwise

Figure 5.2 Java logical operators

The `!` operator is used to perform the *logical NOT* operation, which is also called the *logical complement*. The logical complement of a boolean value yields its opposite value. That is, if a boolean variable called `found` has the value false, then `!found` is true. Likewise, if


`found` is true, then `!found` is false. The logical NOT operation does not change the value stored in `found`.

A logical operation can be described by a *truth table* that lists all possible combinations of values for the variables involved in an expression. Because the logical NOT operator is unary, there are only two possible values for its one operand: true or false. **Figure 5.3**  shows a truth table that describes the `!` operator.

a	!a
false	true
true	false

Figure 5.3 Truth table describing the logical NOT operator

The `&&` operator performs a *logical AND* operation. The result is true if both operands are true, but false otherwise. Compare that to the result of the *logical OR* operator (`||`), which is true if one or the other or both operands are true, but false otherwise.

The AND and OR operators are both binary operators since each uses two operands. Therefore, there are four possible combinations to consider: both operands are true, both are false, one is true and the other false, and vice versa. **Figure 5.4**  depicts a truth table that shows both the `&&` and `||` operators.

a	b	a && b	a b
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Figure 5.4 Truth table describing the logical AND and OR operators

The logical NOT has the highest precedence of the three logical operators, followed by logical AND, then logical OR.

Consider the following `if` statement:

```
if (!done && (count > MAX))
    System.out.println("Completed.");
```

Under what conditions would the `println` statement be executed? The value of the boolean variable `done` is either true or false, and the NOT operator reverses that value. The value of `count` is either greater than `MAX` or it isn't. The truth table in [Figure 5.5](#) breaks down all of the possibilities.

done	count > MAX	!done	!done && (count > MAX)
false	false	true	false
false	true	true	true
true	false	false	false
true	true	false	false

Figure 5.5 A truth table for a specific condition

An important characteristic of the `&&` and `||` operators is that they are “short-circuited.” That is, if their left operand is sufficient to decide the boolean result of the operation, the right operand is not evaluated. This situation can occur with both operators, but for different reasons. If the left operand of the `&&` operator is false, then the result of the operation will be false no matter what the value of the right operand is. Likewise, if the left operand of the `||` is true, then the result of the operation is true no matter what the value of the right operand is.

Key Concept

Logical operators are often used to construct sophisticated conditions.

Sometimes you can capitalize on the fact that the operation is short-circuited. For example, the condition in the following `if` statement will

not attempt to divide by zero if the left operand is false. If `count` has the value zero, the left side of the `&&` operation is false; therefore, the whole expression is false and the right side is not evaluated.

```
if (count != 0 && total/count > MAX)
    System.out.println("Testing.");
```

You should consider carefully whether or not to rely on these kinds of subtle programming language characteristics. Not all programming languages work the same way. As we have stressed before, you should favor readability over clever programming tricks. Always strive to make it clear to any reader of the code how the logic of your program works.

Self-Review Questions

(see answers in [Appendix L](#) )

SR 5.1 What is meant by the flow of control through a program?

SR 5.2 What type of conditions are conditionals and loops based on?

SR 5.3 What are the equality operators? The relational operators? The logical operators?

SR 5.4 Given the following declarations, what is the value of each of the listed boolean expressions?

```
int value1 = 5, value2 = 10;  
boolean done = true;
```

- a. `value1 <= value2`
- b. `(value1 + 5) >= value2`
- c. `value1 < value2 / 2`
- d. `value2 != value1`
- e. `!(value1 == value2)`
- f. `(value1 < value2) || done`
- g. `(value1 > value2) || done`
- h. `(value1 < value2) && !done`
- i. `done || !done`
- j. `((value1 > value2) || done) && (!done || (value2 > value1))`

SR 5.5 What is a truth table?

SR 5.6 Assuming `done` is a `boolean` variable and `value` is an `int` variable, create a truth table for the expression:


```
(value > 0 ) || !done
```

SR 5.7 Assuming `c1` and `c2` are `boolean` variables, create a truth table for the expression:

```
(c1 && !c2) || (!c1 && c2)
```


5.2 The `if` Statement

We've used a basic `if` statement in earlier examples in this chapter. Let's now explore it in detail.

An *if statement* consists of the reserved word `if` followed by a boolean expression, followed by a statement. The condition is enclosed in parentheses and must evaluate to true or false. If the condition is true, the statement is executed and processing continues with the next statement. If the condition is false, the statement is skipped and processing continues immediately with the next statement. **Figure 5.6**  shows this processing.

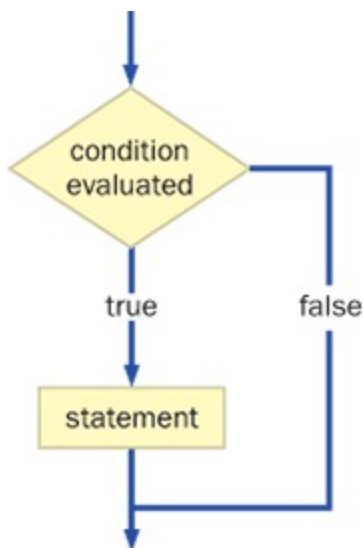


Figure 5.6 The logic of an `if` statement

Key Concept


Proper indentation is important for human readability; it shows the relationship between one statement and another.

Consider the following example of an `if` statement:

```
if (total > amount)
    total = total + (amount + 1);
```

In this example, if the value in `total` is greater than the value in `amount`, the assignment statement is executed; otherwise, the assignment statement is skipped.

Note that the assignment statement in this example is indented under the header line of the `if` statement. This communicates that the assignment statement is part of the `if` statement; it implies that the `if` statement governs whether the assignment statement will be executed. Although this indentation is extremely important for the human reader, it is ignored by the compiler.

The example in [Listing 5.1](#)  reads the age of the user and then makes a decision as to whether to print a particular sentence based on the age that is entered. The `Age` program echoes the age value

that is entered in all cases. If the age is less than the value of the constant `MINOR`, the statement about youth is printed. If the age is equal to or greater than the value of `MINOR`, the `println` statement is skipped. In either case, the final sentence about age being a state of mind is printed.

Listing 5.1

```
//*****

//  Age.java      Author: Lewis/Loftus
//
//  Demonstrates the use of an if statement.
//*****

import java.util.Scanner;

public class Age
{
    //-----
    ---
    //  Reads the user's age and prints comments accordingly.
    //-----
    ---
    public static void main(String[] args)
    {
```

```
final int MINOR = 21;

Scanner scan = new Scanner(System.in);
System.out.print("Enter your age: ");
int age = scan.nextInt();

System.out.println("You entered: " + age);

if (age < MINOR)
    System.out.println("Youth is a wonderful thing.
Enjoy.");

System.out.println("Age is a state of mind.");
}
```

Output

```
Enter your age: 40
You entered: 40
Age is a state of mind.
```

Let's look at a few more examples of basic `if` statements. The following `if` statement causes the variable `size` to be set to zero if

its current value is greater than or equal to the value in the constant `MAX`:

```
if (size >= MAX)
    size = 0;
```

The condition of the following `if` statement first adds three values together, then compares the result to the value stored in `numBooks`:

```
if (numBooks < stackCount + inventoryCount + duplicateCount)
    reorder = true;
```

If `numBooks` is less than the other three values combined, the boolean variable `reorder` is set to `true`. The addition operations are performed before the less than operator, because the arithmetic operators have a higher precedence than the relational operators.

Assuming `generator` refers to an object of the `Random` class, the following `if` statement examines the value returned from a call to `nextInt` to determine a random winner:

```
if (generator.nextInt(CHANCE) == 0)
    System.out.println("You are a randomly selected winner!");
```

The odds of this code picking a winner are based on the value of the `CHANCE` constant. That is, if `CHANCE` contains 20, the odds of winning are 1 in 20. The fact that the condition is looking for a return value of 0 is arbitrary; any value between 0 and `CHANCE-1` would have worked.

The `if-else` Statement

Sometimes we want to do one thing if a condition is true and another thing if that condition is false. We can add an *else clause* to an `if` statement, making it an *if-else statement*, to handle this kind of situation. The following is an example of an `if-else` statement:

```
if (height <= MAX)
    adjustment = 0;
else
    adjustment = MAX - height;
```

If the condition is true, the first assignment statement is executed; if the condition is false, the second assignment statement is executed. Only one or the other will be executed, because a boolean condition evaluates to either true or false. Note that proper indentation is used again to communicate that the statements are part of the governing `if` statement.

Key Concept

An `if-else` statement allows a program to do one thing if a condition is true and another thing if the condition is false.

If Statement




An `if` statement tests the boolean Expression and, if true, executes the first Statement. The optional `else` clause identifies the Statement that should be executed if the Expression is false.

Examples:

```
if (total < 7)
    System.out.println("Total is less than 7.");
if (firstCh != 'a')
    count++;
```

```
        else  
            count = count / 2;
```

The `Wages` program shown in [Listing 5.2](#)  uses an `if-else` statement to compute the proper payment amount for an employee.

Listing 5.2

```
/*******  
  
//  Wages.java      Author: Lewis/Loftus  
//  
//  Demonstrates the use of an if-else statement.  
/*******  
  
  
import java.text.NumberFormat;  
import java.util.Scanner;  
  
public class Wages  
{  
    //-----  
    -----  
    //  Reads the number of hours worked and calculates wages.  
    //-----
```

```
-----  
  
public static void main(String[] args)  
{  
    final double RATE = 8.25;    // regular pay rate  
    final int STANDARD = 40;    // standard hours in a work  
week  
  
    Scanner scan = new Scanner(System.in);  
  
    double pay = 0.0;  
  
    System.out.print("Enter the number of hours worked: ");  
    int hours = scan.nextInt();  
    System.out.println();  
  
    // Pay overtime at "time and a half"  
    if (hours > STANDARD)  
        pay = STANDARD * RATE + (hours-STANDARD) * (RATE *  
1.5);  
    else  
        pay = hours * RATE;  
  
    NumberFormat fmt = NumberFormat.getCurrencyInstance();  
    System.out.println("Gross earnings: " + fmt.format(pay));  
}  
}
```

Output

```
Enter the number of hours worked: 46

Gross earnings: $404.25
```


In the `Wages` program, if an employee works over 40 hours in a week, the payment amount takes into account the overtime hours. An `if-else` statement is used to determine whether the number of hours entered by the user is greater than 40. If it is, the extra hours are paid at a rate one and a half times the normal rate. If there are no overtime hours, the total payment is based simply on the number of hours worked and the standard rate.

Let's look at another example of an `if-else` statement:

```
if (roster.getSize() == FULL)
    roster.expand();
else
    roster.addName(name);
```

This example makes use of an object called `roster`. Even without knowing what `roster` represents, or from what class it was created, we can see that it has at least three methods: `getSize`, `expand`, and

`addName`. The condition of the `if` statement calls `getSize` and compares the result to the constant `FULL`. If the condition is true, the `expand` method is invoked (apparently to expand the size of the roster). If the roster is not yet full, the variable `name` is passed as a parameter to the `addName` method.

The program in **Listing 5.3**  instantiates a `Coin` object, flips the coin by calling the `flip` method, then uses an `if-else` statement to determine which of two sentences gets printed based on the result.

Listing 5.3

```
//*****  
  
//  CoinFlip.java      Author: Lewis/Loftus  
//  
//  Demonstrates the use of an if-else statement.  
//*****  
  
public class CoinFlip  
{  
    //-----  
    -----  
    //  Creates a Coin object, flips it, and prints the  
    results.  
    //-----  
    -----  
}
```

```
public static void main(String[] args)
{
    Coin myCoin = new Coin();

    myCoin.flip();

    System.out.println(myCoin);

    if (myCoin.isHeads())
        System.out.println("You win.");
    else
        System.out.println("Better luck next time.");
}
```

Output

```
Tails
Better luck next time.
```

The `Coin` class is shown in [Listing 5.4](#). It stores two integer constants (`HEADS` and `TAILS`) that represent the two possible states of the coin, and an instance variable called `face` that represents the current state of the coin. The `Coin` constructor initially flips the coin by calling the `flip` method, which determines the new state of the coin

by randomly choosing a number (either 0 or 1). The `isHeads` method returns a `boolean` value based on the current face value of the coin. The `toString` method uses an `if-else` statement to determine which character string to return to describe the coin. The `toString` method is automatically called when the `myCoin` object is passed to `println` in the `main` method.

Listing 5.4

```
//*****

//  Coin.java      Author: Lewis/Loftus
//
//  Represents a coin with two sides that can be flipped.
//*****

public class Coin
{
    private final int HEADS = 0;
    private final int TAILS = 1;

    private int face;

    //-----
    -----
    //  Sets up the coin by flipping it initially.
```

```
//-----
```

```
public Coin()
```

```
{
```

```
    flip();
```

```
}
```

```
//-----
```

```
// Flips the coin by randomly choosing a face value.
```

```
//-----
```

```
public void flip()
```

```
{
```

```
    face = (int) (Math.random() * 2);
```

```
}
```

```
//-----
```

```
// Returns true if the current face of the coin is heads.
```

```
//-----
```

```
public boolean isHeads()
```

```
{
```

```
    return (face == HEADS);
```

```
}
```

```
//-----
```

```
-----  
    // Returns the current face of the coin as a string.  
    //-----  
-----  
    public String toString()  
    {  
        String faceName;  
        if (face == HEADS)  
            faceName = "Heads";  
        else  
            faceName = "Tails";  
  
        return faceName;  
    }  
}
```

Using Block Statements

We may want to do more than one thing as the result of evaluating a boolean condition. In Java, we can replace any single statement with a *block statement*. A block statement is a collection of statements enclosed in braces. We've used these braces many times in previous examples to enclose method and class definitions.

The program called `Guessing`, shown in [Listing 5.5](#), uses an `if-else` statement in which the statement of the `else` clause is a block

statement.

Listing 5.5

```
//*****

//  Guessing.java      Author: Lewis/Loftus
//
//  Demonstrates the use of a block statement in an if-else.
//*****

import java.util.*;

public class Guessing
{
    //-----
    -----
    //  Plays a simple guessing game with the user.
    //-----
    -----
    public static void main(String[] args)
    {
        final int MAX = 10;
        int answer, guess;

        Scanner scan = new Scanner(System.in);
        Random generator = new Random();
```

```
        answer = generator.nextInt(MAX) + 1;

        System.out.print("I'm thinking of a number between 1 and
"
        + MAX + ". Guess what it is: ");

        guess = scan.nextInt();

        if (guess == answer)
            System.out.println("You got it! Good guessing!");
        else
        {
            System.out.println("That is not correct, sorry.");
            System.out.println("The number was " + answer);
        }
    }
}
```

Output

```
I'm thinking of a number between 1 and 10. Guess what
it is: 7
That is not correct, sorry.
The number was 5
```

If the guess entered by the user equals the randomly chosen answer, an appropriate acknowledgment is printed. However, if the answer is incorrect, two statements are printed, one that states that the guess is wrong and one that prints the actual answer. A programming project at the end of this chapter expands the concept of this example into the Hi-Lo game.

Note that if the block braces were not used, the sentence stating that the answer is incorrect would be printed if the answer was wrong, but the sentence revealing the correct answer would be printed in all cases. That is, only the first statement would be considered part of the `else` clause.

Remember that indentation means nothing except to the human reader. Statements that are not blocked properly can lead to the programmer making improper assumptions about how the code will execute. For example, the following code is misleading:

```
if (depth >= UPPER_LIMIT)
    delta = 100;
else
    System.out.println("WARNING: Delta is being reset to ZERO");
    delta = 0;    // not part of the else clause!
```

The indentation (not to mention the logic of the code) implies that the variable `delta` is reset to zero only when `depth` is less than `UPPER_LIMIT`. However, without using a block, the assignment

statement that resets `delta` to zero is not governed by the `if-else` statement at all. It is executed in either case, which is clearly not what is intended.



Examples using conditionals.

A block statement can be used anywhere a single statement is called for in Java syntax. For example, the `if` portion of an `if-else` statement could be a block, or the `else` portion could be a block (as we saw in the `Guessing` program), or both parts could be block statements. For example:

```
if (boxes != warehouse.getCount())
{
    System.out.println("Inventory and warehouse do NOT match.");
    System.out.println("Beginning inventory process again!");
    boxes = 0;
}
else
{
    System.out.println("Inventory and warehouse MATCH.");
    warehouse.ship();
}
```

```
}
```

In this `if-else` statement, the value of `boxes` is compared to a value obtained by calling the `getCount` method of the `warehouse` object (whatever that is). If they do not match exactly, two `println` statements and an assignment statement are executed. If they do match, a different message is printed and the `ship` method of `warehouse` is invoked.

Nested `if` Statements

The statement executed as the result of an `if` statement could be another `if` statement. This situation is called a *nested if*. It allows us to make another decision after determining the results of a previous decision. The program in [Listing 5.6](#), called `MinOfThree`, uses nested `if` statements to determine the smallest of three integer values entered by the user.

Listing 5.6

```
//*****  
  
//  MinOfThree.java      Author: Lewis/Loftus  
//
```

```
// Demonstrates the use of nested if statements.

//*****

import java.util.Scanner;

public class MinOfThree
{
    //-----
    -----
    // Reads three integers from the user and determines the
    smallest
    // value.
    //-----
    -----
    public static void main(String[] args)
    {
        int num1, num2, num3, min = 0;

        Scanner scan = new Scanner(System.in);

        System.out.println("Enter three integers: ");
        num1 = scan.nextInt();
        num2 = scan.nextInt();
        num3 = scan.nextInt();

        if (num1 < num2)
            if (num1 < num3)
```

```

        min = num1;
    else
        min = num3;
    else
        if (num2 < num3)
            min = num2;
        else
            min = num3;

    System.out.println("Minimum value: " + min);
}
}

```

Output

```

Enter three integers:
45  22  69
Minimum value: 22

```

Carefully trace the logic of the `MinOfThree` program, using various input sets with the minimum value in all three positions, to see how it determines the lowest value.

An important situation arises with nested `if` statements. It may seem that an `else` clause after a nested `if` could apply to either `if` statement. For example:

```
if (code == 'R')  
    if (height <= 20)  
        System.out.println("Situation Normal");  
    else  
        System.out.println("Bravo!");
```

Is the `else` clause matched to the inner `if` statement or the outer `if` statement? The indentation in this example implies that it is part of the inner `if` statement, and that is correct. An `else` clause is always matched to the closest unmatched `if` that preceded it. However, if we're not careful, we can easily mismatch it in our mind and misalign the indentation. This is another reason why accurate, consistent indentation is crucial.

Key Concept

In a nested `if` statement, an `else` clause is matched to the closest unmatched `if`.

Braces can be used to specify the `if` statement to which an `else` clause belongs. For example, if the previous example should have been structured so that the string "`Bravo!`" is printed if `code` is not

equal to 'R', we could force that relationship (and properly indent) as follows:

```
if (code == 'R')
{
    if (height <= 20)
        System.out.println("Situation Normal");
}
else
    System.out.println("Bravo!");
```

By using the block statement in the first `if` statement, we establish that the `else` clause belongs to it.

Self-Review Questions

(see answers in [Appendix L](#) )

SR 5.8 What output is produced by the following code fragment given the assumptions below?

```
if (num1 < num2)
    System.out.print(" red ");
if ((num1 + 5) < num2)
    System.out.print(" white ");
else
```

```
System.out.print(" blue ");  
System.out.println(" yellow ");
```

- a. Assuming the value of `num1` is 2 and the value of `num2` is 10?
- b. Assuming the value of `num1` is 10 and the value of `num2` is 2?
- c. Assuming the value of `num1` is 2 and the value of `num2` is 2?

SR 5.9 How do block statements help us in the construction of conditionals?

SR 5.10 What is a nested `if` statement?

SR 5.11 For each assumption, what output is produced by the following code fragment?

```
if (num1 >= num2)  
{  
    System.out.print(" red ");  
    System.out.print(" orange ");  
}  
if ((num1 + 5) >= num2)  
    System.out.print(" white ");  
else  
    if ((num1 + 10) >= num2)  
    {  
        System.out.print(" black ");
```

```
        System.out.print(" blue ");  
    }  
    else  
        System.out.print(" yellow ");  
    System.out.println(" green ");
```

- a. Assuming the value of `num1` is 5 and the value of `num2` is 4?
- b. Assuming the value of `num1` is 5 and the value of `num2` is 12?
- c. Assuming the value of `num1` is 5 and the value of `num2` is 27?

SR 5.12 Write an expression that will print a message based on the value of the int variable named `temperature`. If `temperature` is equal to or less than 50, it prints “It is cool.” on one line and “Dress warmly.” on the next. If `temperature` is greater than 80, it prints “It is warm.” on one line and “Dress coolly.” on the next. If `temperature` is in between 50 and 80, it prints “It is pleasant.” on one line and “Dress pleasantly.” on the next.

5.3 Comparing Data

When comparing data using boolean expressions, it's important to understand some nuances that arise depending on the type of data being examined. Let's look at a few key situations.

Comparing Floats

An interesting situation occurs when comparing floating point data. Two floating point values are equal, according to the `==` operator, only if all the binary digits of their underlying representations match. If the compared values are the results of computation, it may be unlikely that they are exactly equal even if they are close enough for the specific situation. Therefore, you should rarely use the equality operator (`==`) when comparing floating point values.

A better way to check for floating point equality is to compute the absolute value of the difference between the two values and compare the result to some tolerance level. For example, we may choose a tolerance level of `0.00001`. If the two floating point values are so close that their difference is less than the tolerance, then we are willing to consider them equal. Comparing two floating point values, `f1` and `f2`, could be accomplished as follows:

```
if (Math.abs(f1 - f2) < TOLERANCE)
    System.out.println("Essentially equal.");
```

The value of the constant `TOLERANCE` should be appropriate for the situation.

Comparing Characters

We know what it means when we say that one number is less than another, but what does it mean to say one character is less than another? As we discussed in [Chapter 2](#), characters in Java are based on the Unicode character set, which defines an ordering of all possible characters that can be used. Because the character `'a'` comes before the character `'b'` in the character set, we can say that `'a'` is less than `'b'`.

We can use the equality and relational operators on character data. For example, if two character variables `ch1` and `ch2` hold two characters, we might determine their relative ordering in the Unicode character set with an `if` statement as follows:

```
if (ch1 > ch2)
    System.out.println(ch1 + " is greater than " + ch2);
else
```

```
System.out.println(ch1 + " is NOT greater than " + ch2);
```

Key Concept

The relative order of characters in Java is defined by the Unicode character set.

The Unicode character set is structured so that all lowercase alphabetic characters ('a' through 'z') are contiguous and in alphabetical order. The same is true of uppercase alphabetic characters ('A' through 'Z') and characters that represent digits ('0' through '9'). The digits precede the uppercase alphabetic characters, which precede the lowercase alphabetic characters. Before, after, and in between these groups are other characters. See the chart in [Appendix C](#) for details.

Remember that a character and a character string are two different types of information. A `char` is a primitive value that represents one character. A character string is represented as an object in Java, defined by the `String` class. While comparing strings is based on comparing the characters in the strings, the comparison is governed by the rules for comparing objects.

Comparing Objects

The Unicode relationships among characters make it easy to sort characters and strings of characters. If you have a list of names, for instance, you can put them in alphabetical order based on the inherent relationships among characters in the character set.

However, you should not use the equality or relational operators to compare `String` objects. The `String` class contains a method called `equals` that returns a `boolean` value that is true if the two strings being compared contain exactly the same characters and is false otherwise. For example:

```
if (name1.equals(name2))  
    System.out.println("The names are the same.");  
else  
    System.out.println("The names are not the same.");
```

Assuming that `name1` and `name2` are `String` objects, this condition determines whether the characters they contain are an exact match. Because both objects were created from the `String` class, they both respond to the `equals` message. Therefore, the condition could have been written as `name2.equals(name1)`, and the same result would occur.

Key Concept

The `compareTo` method can be used to determine the relative order of strings.

It is valid to test the condition `(name1 == name2)`, but that actually tests to see whether both reference variables refer to the same `String` object. For any object, the `==` operator tests whether both reference variables are aliases of each other (whether they contain the same address). That's different from testing to see whether two different `String` objects contain the same characters.

Keep in mind that a string literal (such as `"Nathan"`) is a convenience and is actually a shorthand technique for creating a `String` object. An interesting issue related to string comparisons is the fact that Java creates a unique object for string literals only when needed. That is, if the string literal `"Hi"` is used multiple times in a method, only one `String` object is created to represent it. Therefore, the conditions of both `if` statements in the following code are true:

```
String str = "software";  
if (str == "software")  
    System.out.println("References are the same");  
if (str.equals("software"))  
    System.out.println("Characters are the same");
```

The first time the string literal "software" is used, a `String` object is created to represent it and the reference variable `str` is set to its address. Each subsequent time the literal is used, the original object is referenced.

To determine the relative ordering of two strings, use the `compareTo` method of the `String` class. The `compareTo` method is more versatile than the `equals` method. Instead of returning a `boolean` value, the `compareTo` method returns an integer. The return value is negative if the `String` object through which the method is invoked precedes (is less than) the string that is passed in as a parameter. The return value is zero if the two strings contain the same characters. The return value is positive if the `String` object through which the method is invoked follows (is greater than) the string that is passed in as a parameter. For example:

```
int result = name1.compareTo(name2);
if (result < 0)
    System.out.println(name1 + " comes before " + name2);
else
    if (result == 0)
        System.out.println("The names are equal.");
    else
        System.out.println(name1 + " follows " + name2);
```

Keep in mind that comparing characters and strings is based on the Unicode character set (see [Appendix C](#)). This is called a *lexicographic ordering*. If all alphabetic characters are in the same case (upper or lower), the lexicographic ordering will be alphabetic ordering as well. However, when comparing two strings, such as "able" and "Baker", the `compareTo` method will conclude that "Baker" comes first because all of the uppercase letters come before all of the lowercase letters in the Unicode character set. A string that is the prefix of another, longer string is considered to precede the longer string. For example, when comparing the two strings "horse" and "horsefly", the `compareTo` method will conclude that "horse" comes first.

Self-Review Questions

(see answers in [Appendix L](#))


SR 5.13 Why must we be careful when comparing floating point values for equality?

SR 5.14 How do we compare strings for equality?

SR 5.15 Write an `equals` method for the `Die` class of [Section 4.2](#). The method should return `true` if the `Die` object it is invoked on has the same `facevalue` as the `Die` object passed as a parameter, otherwise it should return `false`.

SR 5.16 Assume the `String` variables `s1` and `s2` have been initialized. Write an expression that prints out the two strings on separate lines in lexicographic order.

5.4 The `while` Statement

As we discussed in the introduction of this chapter, a repetition statement (or loop) allows us to execute another statement multiple times. A *while statement* is a loop that evaluates a boolean condition just as an `if` statement does and executes a statement (called the *body* of the loop) if the condition is true. However, unlike the `if` statement, after the body is executed, the condition is evaluated again. If it is still true, the body is executed again. This repetition continues until the condition becomes false; then processing continues with the statement after the body of the `while` loop. **Figure 5.7**  shows this processing.

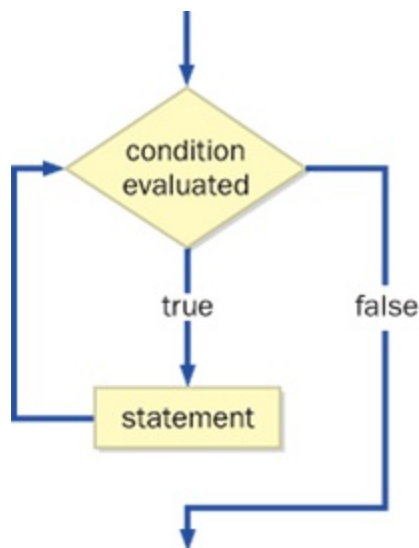
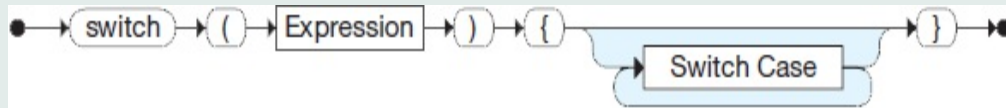


Figure 5.7 The logic of a `while` loop

While Statement



The `while` loop repeatedly executes the specified Statement as long as the boolean Expression is true. The Expression is evaluated first; therefore the Statement might not be executed at all. The Expression is evaluated again after each execution of Statement until the Expression becomes false.

Example:

```
while (total > max)
{
    total = total / 2;
    System.out.println("Current total: " +
total);
}
```


The following loop prints the values from 1 to 5. Each iteration through the loop prints one value, then increments the counter.

```
int count = 1;
while (count <= 5)
{
    System.out.println(count);
    count++;
}
```

Key Concept

A `while` statement executes the same statement until its condition becomes false.

Note that the body of the `while` loop is a block containing two statements. The entire block is repeated on each iteration of the loop.

Let's look at another program that uses a `while` loop. The `Average` program shown in **Listing 5.7**  reads a series of integer values from the user, sums them up, and computes their average.

Listing 5.7

```
//*****
```

```

// Average.java      Author: Lewis/Loftus
//
// Demonstrates the use of a while loop, a sentinel value,
// and a
// running sum.
//*****

import java.text.DecimalFormat;
import java.util.Scanner;

public class Average
{
    //-----
    -----
    // Computes the average of a set of values entered by the
    user.
    // The running sum is printed as the numbers are entered.
    //-----
    -----

    public static void main(String[] args)
    {
        int sum = 0, value, count = 0;
        double average;

        Scanner scan = new Scanner(System.in);
        System.out.print("Enter an integer (0 to quit): ");
        value = scan.nextInt();

```

```
        while (value != 0)    // sentinel value of 0 to terminate
loop
    {
        count++;

        sum += value;

        System.out.println("The sum so far is " + sum);

        System.out.print("Enter an integer (0 to quit): ");
        value = scan.nextInt();
    }

    System.out.println();

    if (count == 0)
        System.out.println("No values were entered.");
    else
    {
        average = (double)sum / count;

        DecimalFormat fmt = new DecimalFormat("0.###");
        System.out.println("The average is " +
fmt.format(average));
    }
}
}
```

Output

```
Enter an integer (0 to quit): 25
The sum so far is 25
Enter an integer (0 to quit): 164
The sum so far is 189
Enter an integer (0 to quit): -14
The sum so far is 175
Enter an integer (0 to quit): 84
The sum so far is 259
Enter an integer (0 to quit): 12
The sum so far is 271
Enter an integer (0 to quit): -35
The sum so far is 236
Enter an integer (0 to quit): 0

The average is 39.333
```


We don't know how many values the user may enter, so we need to have a way to indicate that the user has finished entering numbers. In this program, we designate zero to be a *sentinel value* that indicates the end of the input. The `while` loop continues to process input values until the user enters zero. This assumes that zero is not one of the valid numbers that should contribute to the average. A sentinel value must always be outside the normal range of values entered.

Note that in the `Average` program, a variable called `sum` is used to maintain a *running sum*, which means it is the sum of the values entered thus far. The variable `sum` is initialized to zero, and each value read is added to and stored back into `sum`.



Examples using `while` loops.

We also have to count the number of values that are entered so that after the loop concludes we can divide by the appropriate value to compute the average. Note that the sentinel value is not counted. Consider the unusual situation in which the user immediately enters the sentinel value before entering any valid values. The `if` statement at the end of the program avoids a divide-by-zero error.

Let's examine yet another program that uses a `while` loop. The `WinPercentage` program shown in **Listing 5.8**  computes the winning percentage of a sports team based on the number of games won.

Listing 5.8

```
//*****
```

```

// WinPercentage.java      Author: Lewis/Loftus
//
// Demonstrates the use of a while loop for input validation.
//*****

```

```

import java.text.NumberFormat;
import java.util.Scanner;

public class WinPercentage
{
    //-----

    // Computes the percentage of games won by a team.
    //-----

    public static void main(String[] args)
    {
        final int NUM_GAMES = 12;
        int won;
        double ratio;
        Scanner scan = new Scanner(System.in);

        System.out.print("Enter the number of games won (0 to "
            + NUM_GAMES + "): ");

        won = scan.nextInt();

```

```

        while (won < 0 || won > NUM_GAMES)
        {
            System.out.print("Invalid input. Please reenter: ");
            won = scan.nextInt();
        }

        ratio = (double)won / NUM_GAMES;

        NumberFormat fmt = NumberFormat.getPercentInstance();

        System.out.println();
        System.out.println("Winning percentage: " +
            fmt.format(ratio));
    }
}

```

Output

```

Enter the number of games won (0 to 12): -5
Invalid input. Please reenter: 13
Invalid input. Please reenter: 7

Winning percentage: 58%

```

We use a `while` loop in the `WinPercentage` program to *validate the input*, meaning we guarantee that the user enters a value that we

consider to be valid. In this example, that means that the number of games won must be greater than or equal to zero and less than or equal to the total number of games played. The `while` loop continues to execute, repeatedly prompting the user for valid input, until the entered number is indeed valid.

We generally want our programs to be *robust*, which means that they handle potential problems as elegantly as possible. Validating input data and avoiding errors such as dividing by zero are situations that we should consciously address when designing a program. Loops and conditionals help us recognize and deal with such situations.

Infinite Loops

It is the programmer's responsibility to ensure that the condition of a loop will eventually become false. If it doesn't, the loop body will execute forever, or at least until the program is interrupted. This situation, referred to as an **infinite loop**, ⓘ is a common mistake.

The following is an example of an infinite loop:

```
int count = 1;
while (count <= 25)    // Warning: this is an infinite loop!
{
    System.out.println(count);
    count = count - 1;
```

```
}
```

If you execute this loop, you should be prepared to interrupt it. On most systems, pressing the Control-C keyboard combination (hold down the Control key and press C) terminates a running program.

In this example, the initial value of `count` is `1`, and it is decremented in the loop body. The `while` loop will continue as long as `count` is less than or equal to `25`. Because `count` gets smaller with each iteration, the condition will always be true, or at least until the value of `count` gets so small that an underflow error occurs. The point is that the logic of the code is clearly wrong.

Let's look at some other examples of infinite loops:

Key Concept

We must design our programs carefully to avoid infinite loops.

```
int count = 1;
while (count != 50)    // infinite loop
    count += 2;
```

In this code fragment, the variable `count` is initialized to 1 and is moving in a positive direction. However, note that it is being incremented by 2 each time. This loop will never terminate because `count` will never equal 50. It begins at 1 and then changes to 3, then 5, and so on. Eventually it reaches 49, then changes to 51, then 53, and continues forever.

Now consider the following situation:

```
double num = 1.0;
while (num != 0.0)    // infinite loop
    num = num - 0.1;
```

Once again, the value of the loop control variable seems to be moving in the correct direction. And, in fact, it seems like `num` will eventually take on the value `0.0`. However, this loop is infinite (at least on most systems), because `num` will never have a value *exactly* equal to `0.0`. This situation is similar to one we discussed earlier in this chapter when we explored the idea of comparing floating point values in the condition of an `if` statement. Because of the way the values are represented in binary, minute computational errors occur internally, making it problematic to compare two floating point values for equality.

Nested Loops

The body of a loop can contain another loop. This situation is called a *nested loop*. Keep in mind that for each iteration of the outer loop, the inner loop executes completely. Consider the following code fragment. How many times does the string "Here again" get printed?

```
int count1, count2;  
count1 = 1;  
while (count1 <= 10)  
{  
    count2 = 1;  
    while (count2 <= 50)  
    {  
        System.out.println("Here again");  
        count2++;  
    }  
    count1++;  
}
```


The `println` statement is inside the inner loop. The outer loop executes 10 times, as `count1` iterates between 1 and 10. The inner loop executes 50 times, as `count2` iterates between 1 and 50. For each iteration of the outer loop, the inner loop executes completely. Therefore the `println` statement is executed 500 times.

As with any loop situation, we must be careful to scrutinize the conditions of the loops and the initializations of variables. Let's

consider some small changes to this code. What if the condition of the outer loop were `(count1 < 10)` instead of `(count1 <= 10)`? How would that change the total number of lines printed? Well, the outer loop would execute 9 times instead of 10, so the `println` statement would be executed 450 times. What if the outer loop were left as it was originally defined, but `count2` were initialized to 11 instead of 1 before the inner loop? The inner loop would then execute 40 times instead of 50, so the total number of lines printed would be 400.

Let's look at another example that uses a nested loop. A *palindrome* is a string of characters that reads the same forward or backward. For example, the following strings are palindromes:

- radar
- drab bard
- ab cde xxxx edc ba
- kayak
- deified
- able was I ere I saw elba

Note that some palindromes have an even number of characters, whereas others have an odd number of characters. The `PalindromeTester` program shown in [Listing 5.9](#)  tests to see whether a string is a palindrome. The user may test as many strings as desired.

Listing 5.9



```

//*****

// PalindromeTester.java      Author: Lewis/Loftus
//
// Demonstrates the use of nested while loops.
//*****

import java.util.Scanner;

public class PalindromeTester
{
    //-----
    -----
    // Tests strings to see if they are palindromes.
    //-----
    -----
    public static void main(String[] args)
    {
        String str, another = "y";
        int left, right;

        Scanner scan = new Scanner(System.in);

        while (another.equalsIgnoreCase("y")) // allows y or Y
        {
            System.out.println("Enter a potential palindrome:");
            str = scan.nextLine();

```

```

        left = 0;

        right = str.length() - 1;

        while (str.charAt(left) == str.charAt(right) && left
< right)
        {
            left++;
            right--;
        }

        System.out.println();

        if (left < right)

            System.out.println("That string is NOT a
palindrome.");
        else

            System.out.println("That string IS a palindrome.");

        System.out.println();

        System.out.print("Test another palindrome (y/n)? ");

        another = scan.nextLine();
    }
}
}

```

Output

```
Enter a potential palindrome:
radar

That string IS a palindrome.

Test another palindrome (y/n)? y
Enter a potential palindrome:
able was I ere I saw elba

That string IS a palindrome.

Test another palindrome (y/n)? y
Enter a potential palindrome:
abccddcba

That string IS a palindrome.

Test another palindrome (y/n)? y
Enter a potential palindrome:
abracadabra

That string is NOT a palindrome.

Test another palindrome (y/n)? n
```

The code for `PalindromeTester` contains two loops, one inside the other. The outer loop controls how many strings are tested, and the

inner loop scans through each string, character by character, until it determines whether the string is a palindrome.

The variables `left` and `right` store the indexes of two characters. They initially indicate the characters on either end of the string. Each iteration of the inner loop compares the two characters indicated by `left` and `right`. We fall out of the inner loop when either the characters don't match, meaning the string is not a palindrome, or when the value of `left` becomes equal to or greater than the value of `right`, which means the entire string has been tested and it is a palindrome.

Note that the following phrases would not be considered palindromes by the current version of the program:

- A man, a plan, a canal, Panama.
- Dennis and Edna sinned.
- Rise to vote, sir.
- Doom an evil deed, liven a mood.
- Go hang a salami; I'm a lasagna hog.

These strings fail our current criteria for a palindrome because of the spaces, punctuation marks, and changes in uppercase and lowercase. However, if these characteristics were removed or ignored, these strings read the same forward and backward. Consider how the program could be changed to handle these situations. These modifications are included as a programming project at the end of the chapter.

The `break` and `continue` Statements

Java includes two statements that affect the processing of conditionals and loops. When a `break` statement is executed, the flow of execution transfers immediately to the statement after the one governing the current flow. For example, if a `break` statement is executed within the body of a loop, the execution of the loop is stopped and the statement following the loop is executed. It “breaks” out of the loop.

In [Chapter 6](#), we’ll see that using the `break` statement is usually necessary when writing `switch` statements. However, it is never necessary to use a `break` statement in a loop. An equivalent loop can always be written without it. Because the `break` statement causes program flow to jump from one place to another, using a `break` in a loop is not good practice. You can and should avoid using the `break` statement in a loop.

A *continue statement* has a related effect on loop processing. The `continue` statement is similar to a `break`, but the loop condition is evaluated again, and the loop body is executed again if it is still true. Like the `break` statement, the `continue` statement can always be avoided in a loop, and for the same reasons, it should be.

Self-Review Questions

(see answers in [Appendix L](#) )

SR 5.17 What is an infinite loop? Specifically, what causes it?

SR 5.18 What output is produced by the following code fragment?

```
int low = 0, high = 10;
while (low < high)
{
    System.out.println(low);
    low++;
}
```

SR 5.19 What output is produced by the following code fragment?

```
int low = 10, high = 0;
while (low <= high)
{
    System.out.println(low);
    low++;
}
```

SR 5.20 What output is produced by the following code fragment?

```
int low = 0, high = 10;
```

```
while (low <= high)
{
    System.out.println(low);
    high = high - low;
}
```

SR 5.21 What output is produced by the following code fragment?

```
int low = 0, high = 10, mid;
while (low <= high)
{
    mid = low;
    while (mid <= high)
    {
        System.out.print(mid + " ");
        mid++;
    }
    System.out.println();
    low++;
}
```

SR 5.22 Assume the `int` variable `value` has been initialized to a positive integer. Write a `while` loop that prints all of the positive divisors of `value`. For example, if `value` is 28, it prints divisors of 28: 1 2 4 7 14 28

SR 5.23 Assume the `int` variable `value` has been initialized to a positive integer. Write a `while` loop that prints all of the positive divisors of each number from one to `value`. For example, if `value` is 4, it prints

divisors of 1: 1

divisors of 2: 1 2

divisors of 3: 1 3


divisors of 4: 1 2 4

5.5 Iterators

An *iterator* is an object that has methods that allow you to process a collection of items one at a time. That is, an iterator lets you step through each item and interact with it as needed. For example, your goal may be to compute the dues for each member of a club or print the distinct parts of a URL. The key is that an iterator provides a consistent and simple mechanism for systematically processing a group of items. Since it is inherently a repetitive process, it is closely related to the idea of loops.

Key Concept

An iterator is an object that helps you process a group of related items.

Technically an iterator object in Java is defined using the `Iterator` interface, which is discussed in [Chapter 7](#) . For now, it is simply helpful to know that such objects exist and that they can make the processing of a collection of items easier.

Every iterator object has a method called `hasNext` that returns a `boolean` value indicating if there is at least one more item to process.

Therefore, the `hasNext` method can be used as a condition of a loop to control the processing of each item. An iterator also has a method called `next` to retrieve the next item in the collection to process.

There are several classes in the Java standard class library that define iterator objects. One of these is `Scanner`, a class we've used several times in previous examples to help us read data from the user. The `hasNext` method of the `Scanner` class returns true if there is another input token to process. And, as we've seen previously, it has a `next` method that returns the next input token as a string.

The `Scanner` class also has specific variations of the `hasNext` method, such as the `hasNextInt` and `hasNextDouble` methods, which allow you to determine if the next input token is a particular type. Likewise, as we've seen, there are variations of the `next` method, such as `nextInt` and `nextDouble`, which retrieve values of specific types.

When reading input interactively from the standard input stream, the `hasNext` method of the `Scanner` class will wait until there is input available, then return true. That is, interactive input read from the keyboard is always thought to have more data to process—it just hasn't arrived yet until the user types it in. That's why in previous examples we've used special sentinel values to determine the end of interactive input.

However, the fact that a `Scanner` object is an iterator is particularly helpful when the scanner is being used to process input from a source that has a specific end point, such as processing the lines of a data file or processing the parts of a character string. Let's examine an example of this type of processing.

Reading Text Files

Suppose we have an input file called `urls.inp` that contains a list of URLs that we want to process in some way:


`www.google.com`

`www.linux.org/info/gnu.html`

`thelyric.com/calendar/`

`www.cs.vt.edu/undergraduate/about`

`youtube.com/watch?v=EHCRimwRGLs`

The program shown in [Listing 5.10](#)  reads the URLs from this file and dissects them to show the various parts of the path. It uses a `Scanner` object to process the input. In fact, it uses multiple `Scanner` objects—one to read the lines of the data file and another to process each URL string.

Listing 5.10

```

//*****

//  URLDissector.java      Author: Lewis/Loftus
//
//  Demonstrates the use of Scanner to read file input and
//  parse it
//  using alternative delimiters.
//*****

import java.util.Scanner;
import java.io.*;

public class URLDissector
{
    //-----
    //  Reads urls from a file and prints their path
    //  components.
    //-----

    public static void main(String[] args) throws IOException
    {
        String url;
        Scanner fileScan, urlScan;
        fileScan = new Scanner(new File("urls.inp"));

        //  Read and process each line of the file

```

```

        while (fileScan.hasNext())
        {
            url = fileScan.nextLine();
            System.out.println("URL: " + url);

            urlScan = new Scanner(url);
            urlScan.useDelimiter("/");

            // Print each part of the url
            while (urlScan.hasNext())
                System.out.println("  " + urlScan.next());

            System.out.println();
        }
    }
}

```

Output

```


URL: www.google.com
www.google.com
URL: www.linux.org/info/gnu.html
    www.linux.org
    info
    gnu.html
URL: thelyric.com/calendar/
thelyric.com

```

```
calendar
URL: www.cs.vt.edu/undergraduate/about
    www.cs.vt.edu
    undergraduate
    about
URL: youtube.com/watch?v=EHCRimwRGLs
    youtube.com
    watch?v=EHCRimwRGLs
```

There are two `while` loops in this program, one nested within the other. The outer loop processes each line in the file, and the inner loop processes each token in the current line.

The variable `fileScan` is created as a scanner that operates on the input file named `urls.inp`. Instead of passing `System.in` into the `Scanner` constructor, we instantiate a `File` object that represents the input file and pass it into the `Scanner` constructor. At that point, the `fileScan` object is ready to read and process input from the input file.

If for some reason there is a problem finding or opening the input file, the attempt to create a `File` object will throw an `IOException`, which is why we've added the `throws IOException` clause to the `main` method header. (Processing I/O exceptions is discussed further in [Chapter 11](#) 

The body of the outer `while` loop will be executed as long as the `hasNext` method of the input file scanner returns true—that is, as long as there is more input in the data file to process. Each iteration through the loop reads one line (one URL) from the input file and prints it out.

For each URL, a new `Scanner` object is set up to parse the pieces of the URL string, which is passed into the `Scanner` constructor when instantiating the `urlScan` object. The inner `while` loop prints each token of the URL on a separate line.

Recall that, by default, a `Scanner` object assumes that white space (spaces, tabs, and new lines) is used as the delimiters separating the input tokens. That works in this example for the scanner that is reading each line of the input file. However, if the default delimiters do not suffice, as in the processing of a URL in this example, they can be changed.

Key Concept

The delimiters used to separate tokens in a `Scanner` object can be explicitly set as needed.

In this case, we are interested in each part of the path separated by the slash (/) character. A call to the `useDelimiter` method of the scanner sets the delimiter to a slash prior to processing the URL string.

If you want to use more than one alternate delimiter character, or if you want to parse the input in more complex ways, the `Scanner` class can process patterns called *regular expressions*, which are discussed in [Appendix H](#).

Self-Review Questions

(see answers in [Appendix L](#))

SR 5.24 Devise statements that create each of the following `Scanner` objects.

- One for interactive input, which reads from `System.in`.
- One that reads from the file “info.dat”.
- One that reads from the `String` variable `infoString`.

SR 5.25 Assume the `Scanner` object `fileScan` has been initialized to read from a file. Write a `while` loop that calculates the average number of characters per line of the file.


5.6 The `ArrayList` Class

Now that we have a loop in our arsenal of programming statements, let's introduce a very useful class for managing a set of objects. The `ArrayList` class is part of the `java.util` package of the Java standard class library. An `ArrayList` object stores a list of objects and allows you to refer to each one by an integer *index* value. We will often use loops to scan through the objects in the list and deal with them in one way or another.

Internally, the `ArrayList` class manages the list using a programming construct called an **array** ⓘ (hence the name). Arrays are discussed in detail in **Chapter 8** ☐, but we don't have to know the details of arrays to make use of an `ArrayList` object. The `ArrayList` class is part of the Java Collections API, a group of classes that serve to organize and manage other objects. We discuss collection classes further in **Chapter 13** ☐.

Key Concept

An `ArrayList` object stores a list of objects and lets you access them using an integer index.

Figure 5.8  lists several methods of the `ArrayList` class. You can add and remove elements in various ways, determine if the list is empty, and obtain the number of elements currently in the list, among several other operations.

```
ArrayList<E>()  
    Constructor: creates an initially empty list.  
  
boolean add(E obj)  
    Inserts the specified object to the end of this list.  
  
void add(int index, E obj)  
    Inserts the specified object into this list at the specified index.  
  
void clear()  
    Removes all elements from this list.  
  
E remove(int index)  
    Removes the element at the specified index in this list and returns it.  
  
E get(int index)  
    Returns the object at the specified index in this list without removing it.  
  
int indexOf(Object obj)  
    Returns the index of the first occurrence of the specified object.  
  
boolean contains(Object obj)  
    Returns true if this list contains the specified object.  
  
boolean isEmpty()  
    Returns true if this list contains no elements.  
  
int size()  
    Returns the number of elements in this list.
```

Figure 5.8 Some methods of the `ArrayList<E>` class.

Note that the `ArrayList` class refers to having elements of type `E`. That is a *generic type* (the `E` stands for element), which is determined when an `ArrayList` object is created. So you don't just create an `ArrayList` object, you create an `ArrayList` object that will store a particular type of object. The type parameter for a given object is written in angle brackets after the class name. So we can talk about an `ArrayList<String>` object that manages a list of `String` objects, or an `ArrayList<Book>` that manages a list of `Book` objects.

You can create an `ArrayList` without specifying the type of element, in which case the `ArrayList` stores `Object` references, which means that you can put any type of object in the list. This is usually not a good idea. The point of being able to commit to storing a particular type in a given `ArrayList` object lets the compiler help you check that only the appropriate types of objects are being stored in the object.

Key Concept


When an `ArrayList` object is created, you specify the type of element that will be stored in the list.

The index values of an `ArrayList` begin at 0, not 1. So conceptually, for example, an `ArrayList` of `String` objects might be managing the

following list:

```
0 "Bashful"  
1 "Sleepy"  
2 "Happy"  
3 "Dopey"  
4 "Doc"
```

Also note that an `ArrayList` stores references to objects. You cannot create an `ArrayList` that stores primitive values such as an `int`. But that's where wrapper classes come to the rescue again. For example, you can create an `ArrayList<Integer>` or an `ArrayList<Double>` as appropriate.

The program shown in [Listing 5.11](#)  instantiates an `ArrayList<String>` called `band`. The method `add` is used to add several `String` objects to the end of the `ArrayList` in a specific order. Then one particular string is deleted and another is inserted at a particular index. As with any other object, the `toString` method of the `ArrayList` class is automatically called whenever it is sent to the `println` method, which prints all of the elements surrounded by square brackets. The while loop at the end of the program explicitly prints each element on a separate line.

Listing 5.11

```

//*****

// Beatles.java      Author: Lewis/Loftus
//
// Demonstrates the use of a ArrayList object.
//*****

import java.util.ArrayList;

public class Beatles
{
    //-----

    // Stores and modifies a list of band members.
    //-----

    public static void main(String[] args)
    {
        ArrayList<String> band = new ArrayList<String>();

        band.add("Paul");
        band.add("Pete");
        band.add("John");
        band.add("George");

        System.out.println(band);

        int location = band.indexOf("Pete");

        band.remove(location);
    }
}

```

```
        System.out.println(band);  
        System.out.println("At index 1: " + band.get(1));  
        band.add(2, "Ringo");  
  
        System.out.println("Size of the band: " + band.size());  
        int index = 0;  
        while (index < band.size())  
        {  
            System.out.println(band.get(index));  
            index++;  
        }  
    }  
}
```

Output

```
[Paul, Pete, John, George]  
[Paul, John, George]  
At index 1: John  
Size of the band: 4  
Paul  
John  
Ringo  
George
```

Self-Review Questions

(see answers in [Appendix L](#) )

SR 5.26 What are the advantages of using an `ArrayList` object?

SR 5.27 What type of elements does an `ArrayList` hold?

SR 5.28 Write a declaration for a variable named `dice` that is an `ArrayList` of `Die` objects.

SR 5.29 What output is produced by the following code fragment?

```
ArrayList<String> names = new ArrayList<String>();  
names.add("Andy");  
names.add("Betty");  
names.add(1, "Chet");  
names.add(1, "Don");  
names.remove(2);  
System.out.println(names);
```

Summary of Key Concepts

- Conditionals and loops allow us to control the flow of execution through a method.
- An `if` statement allows a program to choose whether to execute a particular statement.
- A loop allows a program to execute a statement multiple times.
- Logical operators are often used to construct sophisticated conditions.
- Proper indentation is important for human readability; it shows the relationship between one statement and another.
- An `if-else` statement allows a program to do one thing if a condition is true and another thing if the condition is false.
- In a nested `if` statement, an `else` clause is matched to the closest unmatched `if`.
- The relative order of characters in Java is defined by the Unicode character set.
- The `compareTo` method can be used to determine the relative order of strings.
- A `while` statement executes the same statement until its condition becomes false.
- We must design our programs carefully to avoid infinite loops.
- An iterator is an object that helps you process a group of related items.

- The delimiters used to separate tokens in a `Scanner` object can be explicitly set as needed.
- An `ArrayList` object stores a list of objects and lets you access them using an integer index.
- When an `ArrayList` object is created, you specify the type of element that will be stored in the list.
- A single event handler can be used to process events generated by multiple controls.
- A character font applied to a `Text`, `Label`, or `Button` object is represented by the `Font` class.
- The `HBox` and `VBox` layout panes arrange their nodes in a single row or column, respectively.
- A group of radio buttons provide a set of mutually exclusive options.

Exercises

EX 5.1 What happens in the `MinOfThree` program if two or more of the values are equal? If exactly two of the values are equal, does it matter whether the equal values are lower or higher than the third?

EX 5.2 What is wrong with the following code fragment? Rewrite it so that it produces correct output.

```
if (total == MAX)
    if (total < sum)
        System.out.println("total == MAX and < sum.");
else
    System.out.println("total is not equal to MAX");
```

EX 5.3 What is wrong with the following code fragment? Will this code compile if it is part of an otherwise valid program? Explain.

```
if (length = MIN_LENGTH)
    System.out.println("The length is minimal.");
```

EX 5.4 What output is produced by the following code fragment?

```
int num = 87, max = 25;
if (num >= max*2)
    System.out.println("apple");
    System.out.println("orange");
System.out.println("pear");
```

EX 5.5 What output is produced by the following code fragment?

```
int limit = 100, num1 = 15, num2 = 40;
if (limit <= limit)
{
    if (num1 == num2)
        System.out.println("lemon");
        System.out.println("lime");
}
System.out.println("grape");
```

EX 5.6 Put the following list of strings in lexicographic order as if determined by the `compareTo` method of the `String` class. Consult the Unicode chart in [Appendix C](#) .

```
"fred"
"Ethel"
"?-?-?-?"
" { ( [ ] ) } "
```



```
"Lucy"  
"ricky"  
"book"  
"*****"  
"12345"  
"  "  
"HEPHALUMP"  
"bookkeeper"  
"6789"  
";+6?"  
"^^^^^^^^^^"  
"hephalump"
```

EX 5.7 What output is produced by the following code fragment?

```
int num = 0, max = 20;  
while (num < max)  
{  
    System.out.println(num);  
    num += 4;  
}
```

EX 5.8 What output is produced by the following code fragment?

```
int num = 1, max = 20;
```

```

while (num < max)
{
    if (num%2 == 0)
        System.out.println(num);
    num++;
}

```

EX 5.9 What is wrong with the following code fragment? What are three distinct ways it could be changed to remove the flaw?

```

count = 50;
while (count >= 0)
{
    System.out.println(count);
    count = count + 1;
}

```

EX 5.10 Write a `while` loop that verifies that the user enters a positive integer value.

EX 5.11 Write a code fragment that reads and prints integer values entered by a user until a particular sentinel value (stored in `SENTINEL`) is entered. Do not print the sentinel value.

EX 5.12 Write a method called `maxOfTwo` that accepts two integer parameters and returns the larger of the two.

EX 5.13 Write a method called `larger` that accepts two floating point parameters (of type `double`) and returns true if the first parameter is greater than the second, and false otherwise.

EX 5.14 Write a method called `evenlyDivisible` that accepts two integer parameters and returns true if the first parameter is evenly divisible by the second, or vice versa, and false otherwise. Return false if either parameter is zero.

EX 5.15 Write a method called `isAlpha` that accepts a character parameter and returns true if that character is either an uppercase or lowercase alphabetic letter.

EX 5.16 Write a method called `floatEquals` that accepts three floating point values as parameters. The method should return true if the first two parameters are equal within the tolerance of the third parameter.

EX 5.17 Write a method called `isIsosceles` that accepts three integer parameters that represent the lengths of the sides of a triangle. The method returns true if the triangle is isosceles but not equilateral (meaning that exactly two of the sides have an equal length), and false otherwise.

EX 5.18 Would it be better to use check boxes or radio buttons to determine the following? Explain.

- a. Your favorite book genre.
- b. Whether to make your profile visible or not.
- c. Which image format to use (jpg, png, or gif).
- d. Which programming languages you know.

Programming Projects

PP 5.1 Write a program that reads an integer value from the user representing a year. The purpose of the program is to determine if the year is a leap year (and therefore has 29 days in February) in the Gregorian calendar. A year is a leap year if it is divisible by 4, unless it is also divisible by 100 but not 400. For example, the year 2003 is not a leap year, but 2004 is. The year 1900 is not a leap year because it is divisible by 100, but the year 2000 is a leap year because even though it is divisible by 100, it is also divisible by 400. Produce an error message for any input value less than 1582 (the year the Gregorian calendar was adopted).

PP 5.2 Modify the solution to the previous project so that the user can evaluate multiple years. Allow the user to terminate the program using an appropriate sentinel value. Validate each input value to ensure it is greater than or equal to 1582.

PP 5.3 Write a program that determines and prints the number of odd, even, and zero digits in an integer value read from the keyboard.

PP 5.4 Write a program that plays the Hi-Lo guessing game with numbers. The program should pick a random number between 1 and 100 (inclusive), then repeatedly prompt the user to guess the number. On each guess, report to the user that he or she is correct or that the guess is high or low. Continue accepting guesses until the user guesses correctly or chooses

to quit. Use a sentinel value to determine whether the user wants to quit. Count the number of guesses and report that value when the user guesses correctly. At the end of each game (by quitting or a correct guess), prompt to determine whether the user wants to play again. Continue playing games until the user chooses to stop.




Developing a solution of **PP**


PP 5.5 Create a modified version of the `PalindromeTester` program so that the spaces, punctuation, and changes in uppercase and lowercase are not considered when determining whether a string is a palindrome. *Hint:* These issues can be handled in several ways. Think carefully about your design.


PP 5.6 Using the `Coin` class defined in this chapter, design and implement a driver class called `FlipRace` whose `main` method creates two `Coin` objects, then continually flips them both to see which coin first comes up heads three flips in a row. Continue flipping the coins until one of the coins wins the race, and consider the possibility that they might tie. Print the results of each turn, and at the end print the winner and total number of flips that were required.

PP 5.7 Write a program that plays the Rock-Paper-Scissors game against the computer. When played between two people, each person picks one of three options (usually shown by a hand gesture) at the same time, and a winner is determined. In the game, Rock beats Scissors, Scissors beats Paper, and Paper beats Rock. The program should randomly choose one of the three options (without revealing it), then prompt for the user's selection. At that point, the program reveals both choices and prints a statement indicating if the user won, the computer won, or if it was a tie. Continue playing until the user chooses to stop, then print the number of user wins, losses, and ties.

PP 5.8 Design and implement an application that simulates a simple slot machine in which three numbers between 0 and 9 are randomly selected and printed side by side. Print an appropriate statement if all three of the numbers are the same, or if any two of the numbers are the same. Continue playing until the user chooses to stop.

PP 5.9 Modify the `Die` class from [Chapter 4](#)  so that the `setFaceValue` method does nothing if the parameter is outside of the valid range of values.

PP 5.10 Modify the `Account` class from [Chapter 4](#)  so that it performs validity checks on the deposit and withdraw operations. Specifically, don't allow the deposit of a negative number or a withdrawal that exceeds the current balance. Print appropriate error messages if these problems occur.

PP 5.11 Using the `PairOfDice` class from [PP 4.9](#) , design and implement a class to play a game called Pig. In this game, the user competes against the computer. On each turn, the

current player rolls a pair of dice and accumulates points. The goal is to reach 100 points before your opponent does. If, on any turn, the player rolls a 1, all points accumulated for that round are forfeited and control of the dice moves to the other player. If the player rolls two 1s in one turn, the player loses all points accumulated thus far in the game and loses control of the dice. The player may voluntarily turn over the dice after each roll. Therefore, the player must decide to either roll again (be a pig) and risk losing points, or relinquish control of the dice, possibly allowing the other player to win. Implement the computer player such that it always relinquishes the dice after accumulating 20 or more points in any given round.

PP 5.12 Design and implement a program to process golf scores. The scores of four golfers are stored in a text file. Each line represents one hole, and the file contains 18 lines. Each line contains five values: par for the hole followed by the number of strokes each golfer used on that hole. Store the totals for par and the players in an `ArrayList`. Determine the winner and produce a table showing how well each golfer did (compared to par).

PP 5.13 Design and implement a program that compares two text input files, line by line, for equality. Print any lines that are not equivalent.

PP 5.14 Design and implement a program that counts the number of integer values in a text input file. Produce a table listing the values you identify as integers from the input file.

PP 5.15 Design and implement a program that counts the number of punctuation marks in a text input file. Produce a

table that shows how many times each symbol occurred.

PP 5.16 Write a JavaFX application that allows the user to pick a set of pizza toppings using a set of check boxes. Assuming each topping cost 50 cents, and a plain pizza costs \$10, display the cost of the pizza.

PP 5.17 Write a JavaFX application that allows the user to select a color out of five options provided by a set of radio buttons. Change the color of a displayed square accordingly.

PP 5.18 Write a JavaFX application that displays the drawing of a traffic light. Allow the user to select the state of the light (stop, caution, or go) from a set of radio buttons.

PP 5.19 Write a JavaFX application that allows the user to display the image of one of the Three Stooges (Moe, Larry, or Curly) based on a radio button choice.

Software Failure Therac-25

What Happened?



Radiation therapy machines deliver precise amounts of targeted radiation.

The Therac-25 was a radiation therapy machine used to deliver targeted electron or X-ray beams in order to destroy cancerous

tissue. While in use, hundreds of patients were given proper treatments using this device. But in six documented cases from 1985 to 1987, the Therac-25 delivered an overdose of radiation resulting in severe disability and death.

In a typical treatment, the patient lies down and the operator adjusts the machine to target the appropriate area of the body. The operator sets the parameters of the treatment on the machine's computer console and pushes a button to deliver the radiation. Patients are told that a typical side effect is minor skin discomfort similar to that of a mild sunburn.

In the accident cases, some patients reported feeling a “tremendous force of heat” or an “electric tingling shock.” In one case, the patient lost the use of her shoulder and arm and had to have her left breast removed because of the radiation damage. Several others died of radiation poisoning.

The amount of radiation delivered is measured in rads (radiation absorbed dose). A standard treatment is around 200 rads. It's estimated that the accidents caused 20,000 rads to be administered.

What Caused It?

The operators were told the Therac-25 had so many safety precautions that it would be “virtually impossible” to overdose a patient. But part of the software used in the Therac-25 was reused from an earlier version of the machine that had included many hardware-based safety precautions. Thus, the hardware

features had masked problems with the underlying software. The safety features of the Therac-25 were dominantly software based, and the lurking problems emerged.

It turned out that if the operator mistyped a parameter and then corrected it in a particular way, the software allowed the machine to deliver the maximum radiation dose without diffusing it properly. It was such a strange error that technicians testing the machine failed to reproduce the problem. At one point, one of the machines that had clearly caused an overdose was put back into service after technicians could find nothing wrong with it.

Analysts say that the software problems were only part of the reason the accidents occurred. While there were fundamental programming errors, there was also inadequate attention to safety issues in general. And one reason the machines were used for so long was that the problems were not reported as accurately and thoroughly as they should have been.

Lessons Learned

Software safety is the dominant issue in this case. When human lives are on the line, it's difficult to imagine not doing everything possible to ensure that your software is as robust as possible. It comes down to risk analysis. How much are you willing to risk that a particular piece of software you're developing still contains an error? For many applications, a problem might cause inconvenience to the user and may have

business implications, but other applications literally have people's lives at stake.

This case is also an example of the difficulty of isolating the problem. For hundreds of patients, the Therac-25 provided excellent treatment. One of the initial complaints was dismissed without proper investigation or reporting. Later, when a problem had clearly occurred, it could not be replicated. This was an example of a truly exceptional situation—one that does not occur usually or under normal circumstances.

Source: computingcases.org, *IEEE Computer*

6 More Conditionals and Loops

Chapter Objectives

- Examine conditional processing using `switch` statements.
- Discuss the conditional operator.
- Examine alternative repetition statements: the `do` and `for` loops.
- Draw graphics with the aid of conditionals and loops.
- Apply transformations to graphical objects.

*In **Chapter 5**, we examined the `if` statement for making decisions and the `while` statement for looping. This chapter explores several additional statements in Java for performing similar tasks. In particular, it examines the `switch` conditional statement, as well as the `do` and `for` loops. These alternative statements differ in key details, and any particular situation may lend itself to the use of one over another. The Graphics Track sections of this chapter explore the use of conditionals and loops to control our graphics and examine the role of transformations.*

6.1 The switch Statement

Another conditional statement in Java is called the *switch statement*, which causes the executing program to follow one of several paths based on a single value. Similar logic could be constructed with multiple `if` statements, but in the cases where it is warranted, a `switch` statement usually makes code easier to read.

The `switch` statement evaluates an expression to determine a value and then matches that value with one of several possible cases. Each case has statements associated with it. After evaluating the expression, control jumps to the statement associated with the first case that matches the value. Consider the following example:

```
switch (idChar)
{
    case 'A':
        aCount = aCount + 1;
        break;
    case 'B':
        bCount = bCount + 1;
        break;
    case 'C':
        cCount = cCount + 1;
        break;
```

```
default:  
    System.out.println("Error in Identification Character.");  
}
```

First, the expression is evaluated. In this example, the expression is a simple `char` variable called `idChar`. Execution then transfers to the first statement after the case value that matches the result of the expression. Therefore, if `idChar` contains an `'A'`, the variable `aCount` is incremented. If it contains a `'B'`, the case for `'A'` is skipped and processing continues where `bCount` is incremented. Likewise, if `idChar` contains a `'C'`, that case is processed.

Key Concept


A `switch` statement matches a character or integer value to one of several possible cases.

If no case value matches that of the expression, execution continues with the optional *default case*, indicated by the reserved word `default`. If no default case exists, no statements in the `switch` statement are executed and processing continues with the statement after the `switch` statement. It is often a good idea to include a default case, even if you don't expect it to be executed.

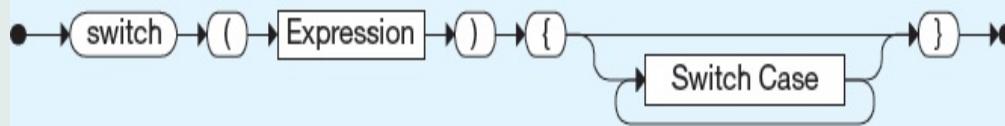
When a `break` statement is encountered, processing jumps to the statement following the `switch` statement. A `break` statement is usually used to break out of each case of a `switch` statement. Without a `break` statement, processing continues into the next case of the `switch`. Therefore, if the `break` statement at the end of the `'A'` case in the previous example was not there, both the `aCount` and `bCount` variables would be incremented when the `idChar` contains an `'A'`. Usually, we want to perform only one case, so a `break` statement is almost always used. Occasionally, though, the “pass through” feature comes in handy.

Key Concept

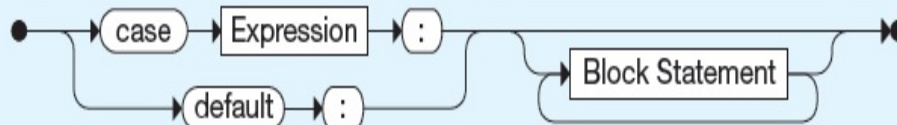
A `break` statement is usually used at the end of each case alternative of a `switch` statement.

You’ll remember that the `break` statement was briefly mentioned in **Chapter 5** , because it could be used in other types of loops and conditionals. We warned then that such processing is usually unnecessary and its use is considered by many developers to be bad practice. The `switch` statement is the exception to this guideline. Using a `break` statement is the only way to make sure the code for only one case is executed.

Switch Statement



Switch Case




The `switch` statement evaluates the initial Expression and matches its value with one of the cases. Processing continues with the Statement corresponding to that case. The optional `default` case will be executed if no other case matches.

Example:

```
switch (numValues)
{
    case 0:
        System.out.println("No values were
entered.");
        break;
    case 1:
        System.out.println("One value was
entered.");
        break;
```

```
        case 2:
            System.out.println("Two values were
entered.");
            break;
        default:
            System.out.println("Too many values were
entered.");
    }
```

The expression evaluated at the beginning of a `switch` statement must be of type `char`, `byte`, `short`, `int`, or an enumerated type. As of Java 7, a `switch` can also be performed on a `String`. But a switch expression cannot be a boolean or a floating point value. Furthermore, the value of each case must be a constant; it cannot be a variable or other expression. This limits the situations in which a `switch` statement is appropriate. But when it is appropriate, it can make the code easier to read and understand.

Note also that the implicit boolean condition of a `switch` statement is based on equality. The expression at the beginning of the statement is compared to each case value to determine which one it equals. A `switch` statement cannot be used to determine other relational operations (such as less than), unless some preliminary processing is done. For example, the `GradeReport` program in **Listing 6.1**  prints a comment based on a numeric grade that is entered by the user.

Listing 6.1

```
//*****

//  GradeReport.java      Author: Lewis/Loftus
//
//  Demonstrates the use of a switch statement.
//*****

import java.util.Scanner;

public class GradeReport
{
    //-----

    //  Reads a grade from the user and prints comments
    accordingly.
    //-----

    public static void main(String[] args)
    {
        int grade, category;

        Scanner scan = new Scanner(System.in);

        System.out.print("Enter a numeric grade (0 to 100): ");
        grade = scan.nextInt();
```

```
        category = grade / 10;

        System.out.print("That grade is ");

        switch (category)
        {
            case 10:
                System.out.println("a perfect score. Well done.");
                break;
            case 9:
                System.out.println("well above average.
Excellent.");
                break;
            case 8:
                System.out.println("above average. Nice job.");
                break;
            case 7:
                System.out.println("average.");
                break;
            case 6:
                System.out.println("below average. You should see
the");
                System.out.println("instructor to clarify the
material "
                                + "presented in class.");
                break;
            default:
```

```
System.out.println("not passing.");
```

```
}
```

```
}
```

```
}
```

Output

```
Enter a numeric grade (0 to 100): 86  
That grade is above average. Nice job.
```

In `GradeReport`, the category of the grade is determined by dividing the grade by 10 using integer division, resulting in an integer value between 0 and 10 (assuming a valid grade is entered). This result is used as the expression of the `switch`, which prints various messages for grades 60 or higher and a default sentence for all other values.

Note that any `switch` statement could be implemented as a set of nested `if` statements. However, nested `if` statements can be difficult for a human reader to understand and are error prone to implement and debug. But because a `switch` can evaluate only equality, sometimes nested `if` statements are necessary. It depends on the situation.

Self-Review Questions

(see answers in [Appendix L](#) )

SR 6.1 When a Java program is running, what happens if the expression evaluated for a `switch` statement does not match any of the case values associated with the statement?

SR 6.2 What happens if a case in a `switch` statement does not end with a `break` statement?

SR 6.3 What is the output of the `GradeReport` program if the user enters 72? What if the user enters 46? What if the user enters 123?

SR 6.4 Transform the following nested `if` statement into an equivalent `switch` statement.

```
if (num1 == 5)
    myChar = 'W';
else
    if (num1 == 6)
        myChar = 'X';
    else
        if (num1 == 7)
            myChar = 'Y';
        else
            myChar = 'Z';
```

6.2 The Conditional Operator

The Java *conditional operator* is similar to an `if-else` statement in some ways. It is a *ternary operator* because it requires three operands. The symbol for the conditional operator is usually written `?:`, but it is not like other operators in that the two symbols that make it up are always separated. The following is an example of an expression that contains the conditional operator:

```
(total > MAX) ? total + 1 : total * 2;
```

Preceding the `?` is a boolean condition. Following the `?` are two expressions separated by the `:` symbol. The entire conditional expression returns the value of the first expression if the condition is true, and returns the value of the second expression if the condition is false.

Key Concept

The conditional operator evaluates to one of two possible values based on a boolean condition.

Keep in mind that this example is an expression that returns a value. The conditional operator is just that, an operator, not a statement that stands on its own. Usually, we want to do something with that value, such as assign it to a variable:

```
total = (total > MAX) ? total + 1 : total * 2;
```

The distinction between the conditional operator and a conditional statement can be subtle. In many ways, the `?:` operator lets us form succinct logic that serves as an abbreviated `if-else` statement. The previous statement is functionally equivalent to, but sometimes more convenient than, the following:

```
if (total > MAX)
    total = total + 1;
else
    total = total * 2;
```

Let's look at a couple more examples. Consider the following declaration:

```
int larger = (num1 > num2) ? num1 : num2;
```

If `num1` is greater than `num2`, the value of `num1` is returned and used to initialize the variable `larger`. If not, the value of `num2` is returned and used to initialize `larger`. Similarly, the following statement prints the smaller of the two values:

```
System.out.println("Smaller: " + ((num1 < num2) ? num1 : num2));
```

The conditional operator is occasionally helpful to evaluate a short condition and return a result. It is not a replacement for an `if-else` statement, however, because the operands to the `?:` operator are expressions, not necessarily full statements. Even when the conditional operator is a viable alternative, you should use it carefully because it may be less readable than an `if-else` statement.

Self-Review Questions

(see answers in [Appendix L](#) )

SR 6.5 What is the difference between a conditional operator and a conditional statement?

SR 6.6 Write a declaration that initializes a `char` variable named `id` to `'A'` if the `boolean` variable `first` is true and to `'B'` otherwise.

SR 6.7 Express the following logic in a succinct manner using the conditional operator.

```
if (val <= 10)
    System.out.println("The value is not greater than
10.");
else
    System.out.println("The value is greater than 10.");
```

6.3 The do Statement

Remember from [Chapter 5](#) that the `while` statement first examines its condition, then executes its body if that condition is true. The *do statement* is similar to the `while` statement except that its termination condition is at the end of the loop body. Like the `while` loop, the `do` loop executes the statement in the loop body until the condition becomes false. The condition is written at the end of the loop to indicate that it is not evaluated until the loop body is executed. Note that the body of a `do` loop is always executed at least once. [Figure 6.1](#) shows this processing.

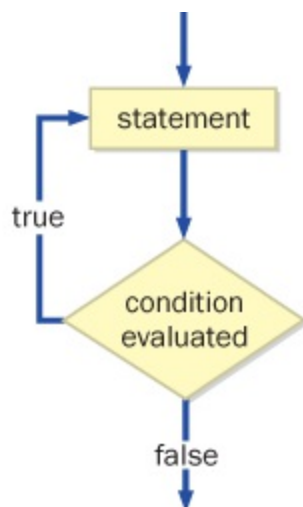


Figure 6.1 The logic of a `do` loop

The following code prints the numbers from 1 to 5 using a `do` loop. Compare this code with the similar example in [Chapter 5](#) that uses

a `while` loop to accomplish the same task.

```
int count = 0;
do
{
    count++;
    System.out.println(count);
}
while (count < 5);
```

Note that the `do` loop begins simply with the reserved word `do`. The body of the `do` loop continues until the *while clause* that contains the boolean condition that determines whether the loop body will be executed again. Sometimes it is difficult to determine whether a line of code that begins with the reserved word `while` is the beginning of a `while` loop or the end of a `do` loop.

Key Concept

A `do` statement executes its loop body at least once.

Let's look at another example of the `do` loop. The program called `ReverseNumber`, shown in **Listing 6.2**, reads an integer from the user and reverses its digits mathematically.

Listing 6.2

```
//*****

//  ReverseNumber.java      Author: Lewis/Loftus
//
//  Demonstrates the use of a do loop.
//*****

import java.util.Scanner;

public class ReverseNumber
{
    //-----
    -----
    //  Reverses the digits of an integer mathematically.
    //-----
    -----

    public static void main(String[] args)
    {
        int number, lastDigit, reverse = 0;
```

```
Scanner scan = new Scanner(System.in);

System.out.print("Enter a positive integer: ");

number = scan.nextInt();

do
{
    lastDigit = number % 10;
    reverse = (reverse * 10) + lastDigit;
    number = number / 10;
}
while (number > 0);

System.out.println("That number reversed is " +
reverse);
}
}
```

Output

```
Enter a positive integer: 2896
That number reversed is 6982
```

Do Statement



The `do` loop repeatedly executes the specified Statement as long as the boolean Expression is true. The Statement is executed at least once, then the Expression is evaluated to determine whether the Statement should be executed again.

Example:

```
do
{
    System.out.print("Enter a word:");
    word = scan.next();
    System.out.println(word);
}
while (!word.equals("quit"));
```

The `do` loop in the `ReverseNumber` program uses the remainder operation to determine the digit in the 1's position, then adds it into the

reversed number, then truncates that digit from the original number using integer division. The `do` loop terminates when we run out of digits to process, which corresponds to the point when the variable `number` reaches the value zero. Carefully trace the logic of this program with a few examples to see how it works.

If you know you want to perform the body of a loop at least once, then you probably want to use a `do` statement. A `do` loop has many of the same properties as a `while` statement, so it must also be checked for termination conditions to avoid infinite loops.

Self-Review Questions

(see answers in [Appendix L](#) )

SR 6.8 Compare and contrast a `while` loop and a `do` loop.

SR 6.9 What output is produced by the following code fragment?

```
int low = 0, high = 10;
do
{
    System.out.println(low);
    low++;
} while (low < high);
```

SR 6.10 What output is produced by the following code fragment?

```
int low = 10, high = 0;
do
{
    System.out.println(low);
    low++;
} while (low <= high);
```

SR 6.11 Write a `do` loop to obtain a sequence of positive integers from the user, using zero as a sentinel value. The program should output the sum of the numbers.

6.4 The `for` Statement

The `while` and the `do` statements are good to use when you don't initially know how many times you want to execute the loop body. The *for statement* is another repetition statement that is particularly well suited for executing the body of a loop a specific number of times that can be determined before the loop is executed.


Key Concept

A `for` statement is usually used when a loop will be executed a set number of times.

The following code prints the numbers 1 through 5 using a `for` loop, just as we did using other loop statements in previous examples:

```
for (int count=1; count <= 5; count++)  
    System.out.println(count);
```

The header of a `for` loop contains three parts separated by semicolons. Before the loop begins, the first part of the header, called

the *initialization*, is executed. The second part of the header is the boolean condition, which is evaluated before the loop body (like the `while` loop). If true, the body of the loop is executed, followed by the execution of the third part of the header, which is called the *increment*. Note that the initialization part is executed only once, but the increment part is executed after each iteration of the loop. **Figure 6.2**  shows this processing.

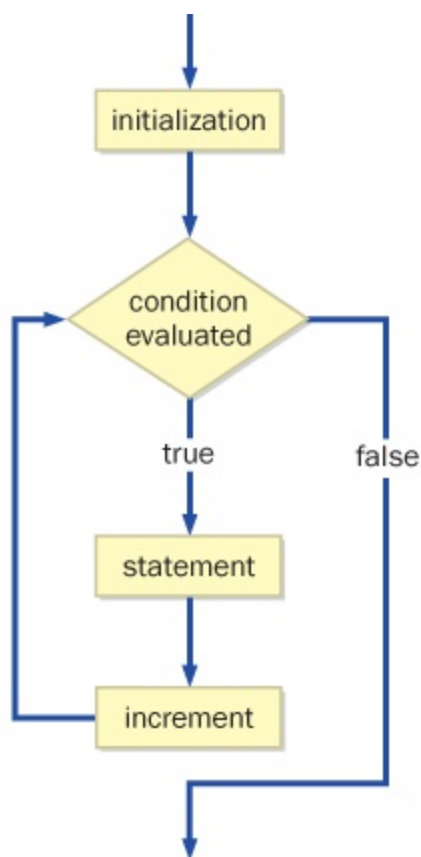


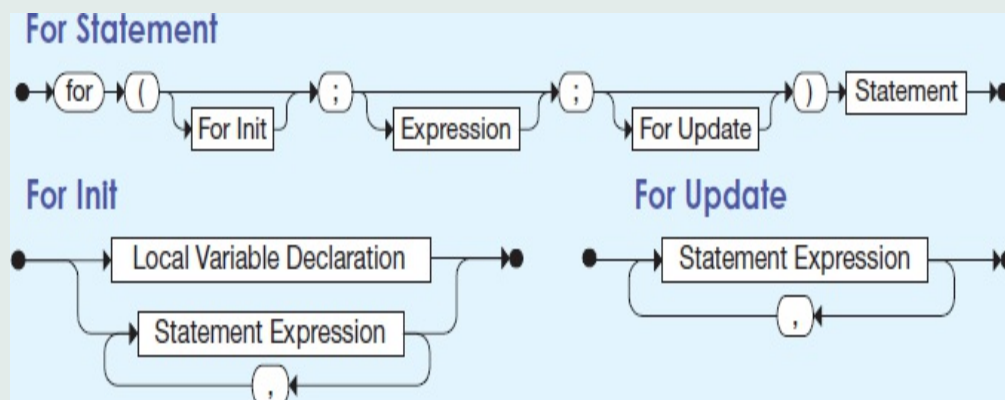
Figure 6.2 The logic of a `for` loop

A `for` loop can be a bit tricky to read until you get used to it. The execution of the code doesn't follow a "top to bottom, left to right" reading. The increment code executes after the body of the loop even though it is in the header.



Examples using `for` loops.

In this example, the initialization portion of the `for` loop header is used to declare the variable `count` as well as to give it an initial value. We are not required to declare a variable there, but it is common practice in situations where the variable is not needed outside of the loop. Because `count` is declared in the `for` loop header, it exists only inside the loop body and cannot be referenced elsewhere. The loop control variable is set up, checked, and modified by the actions in the loop header. It can be referenced inside the loop body, but it should not be modified except by the actions defined in the loop header.



The `for` statement repeatedly executes the specified Statement as long as the boolean Expression is true. The For Init portion of the header is executed only once, before the loop begins. The For Update portion executes after each execution of Statement.

Examples:

```
for (int value=1; value < 25; value++)  
    System.out.println(value + " squared is " +  
value*value);  
  
for (int num=40; num > 0; num-=3)  
    sum = sum + num;
```

The increment portion of the `for` loop header, despite its name, could decrement a value rather than increment it. For example, the following loop prints the integer values from 100 down to 1:

```
for (int num = 100; num > 0; num--)  
    System.out.println(num);
```

In fact, the increment portion of the `for` loop can perform any calculation, not just a simple increment or decrement. Consider the program shown in [Listing 6.3](#), which prints multiples of a particular value up to a particular limit.

Listing 6.3

```
//*****

//  Multiples.java      Author: Lewis/Loftus
//
//  Demonstrates the use of a for loop.
//*****

import java.util.Scanner;

public class Multiples
{
    //-----
    -----
    //  Prints multiples of a user-specified number up to a
    user-
    //  specified limit.
    //-----
    -----

    public static void main(String[] args)
    {
```

```
final int PER_LINE = 5;

int value, limit, mult, count = 0;

Scanner scan = new Scanner(System.in);

System.out.print("Enter a positive value: ");
value = scan.nextInt();

System.out.print("Enter an upper limit: ");
limit = scan.nextInt();

System.out.println();

System.out.println("The multiples of " + value + "
between " +
                    value + " and " + limit + " (inclusive)
are:");

for (mult = value; mult <= limit; mult += value)
{
    System.out.print(mult + "\t");

    // Print a specific number of values per line of
    output
    count++;

    if (count % PER_LINE == 0)
        System.out.println();
}
}
```

Output

```
Enter a positive value: 7
Enter an upper limit: 400

The multiples of 7 between 7 and 400 (inclusive) are:
7      14      21      28      35
42     49     56     63     70
77     84     91     98     105
112    119    126    133    140
147    154    161    168    175
182    189    196    203    210
217    224    231    238    245
252    259    266    273    280
287    294    301    308    315
322    329    336    343    350
357    364    371    378    385
392    399
```

The increment portion of the `for` loop in the `Multiples` program adds the value entered by the user after each iteration. The number of values printed per line is controlled by counting the values printed and then moving to the next line whenever `count` is evenly divisible by the `PER_LINE` constant.

The `Stars` program in [Listing 6.4](#) shows the use of nested `for` loops. The output is a triangle shape made of asterisk characters. The outer loop executes exactly 10 times. Each iteration of the outer loop prints one line of the output. The inner loop performs a different number of iterations depending on the line value controlled by the outer loop. Each iteration of the inner loop prints one star on the current line. Writing programs that print variations on this triangle configuration are included in the programming projects at the end of the chapter.

Listing 6.4

```
//*****

// Stars.java      Author: Lewis/Loftus
//
// Demonstrates the use of nested for loops.
//*****

public class Stars
{
    //-----

    // Prints a triangle shape using asterisk (star)
    characters.

    //-----
    -----
}
```

```
public static void main(String[] args)
{
    final int MAX_ROWS = 10;

    for (int row = 1; row <= MAX_ROWS; row++)
    {
        for (int star = 1; star <= row; star++)
            System.out.print("*");

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

Output

```
*
**
***
****
*****
*****
*****
*****
*****
*****
*****
```

The for-each Loop

A variation of the `for` statement, often called the *for-each loop*, is particularly helpful in situations that involve iterators. In [Chapter 5](#), we discussed that some objects are considered to be iterators, which have `hasNext` and `next` methods to process each item from a group. If an object has implemented the `Iterable` interface, then we can use a variation of the `for` loop to process items using a simplified syntax.

An `ArrayList` object is an `Iterable` object. Therefore, for example, if `library` is an `ArrayList<Book>` object (that is, an `ArrayList` that manages `Book` objects), we can use a `for` loop to process each `Book` object in the collection as follows:


Key Concept

The for-each version of a `for` loop simplifies the processing of all elements in an `Iterable` object.

```
for (Book myBook : library)
    System.out.println(myBook);
```

This code can be read as follows: for each `Book` in `library`, print the book object. The variable `myBook` takes the value of each `Book` object in the collection in turn, and the body of the loop can process it appropriately. That succinct for-each loop is essentially equivalent to the following:

```
Book myBook;
while (bookList.hasNext())
{
    myBook = bookList.next();
    System.out.println(myBook);
}
```

This version of the `for` loop can also be used on arrays, which are discussed in [Chapter 8](#) . We use the for-each loop as appropriate in various situations throughout the rest of the book.

Comparing Loops

The three basic loop statements (`while`, `do`, and `for`) are functionally equivalent. Any particular loop written using one type of loop can be written using either of the other two loop types. Which type of loop we use depends on the situation.

As we mentioned earlier, the primary difference between a `while` loop and a `do` loop is when the condition is evaluated. The body of a `do` loop is executed at least once, whereas the body of a `while` loop will not be executed at all if the condition is initially false. Therefore, we say that the body of a `while` loop is executed zero or more times, but the body of a `do` loop is executed one or more times.

A `for` loop is like a `while` loop in that the condition is evaluated before the loop body is executed. We generally use a `for` loop when the number of times we want to iterate through a loop is fixed or can be easily calculated. In many situations, it is simply more convenient to separate the code that sets up and controls the loop iterations inside the `for` loop header from the body of the loop.

Key Concept

The loop statements are functionally equivalent. Which one you use should depend on the situation.

Self-Review Questions

(see answers in [Appendix L](#) )

SR 6.12 When would we use a `for` loop instead of a `while` loop?

SR 6.13 What output is produced by the following code fragment?

```
int value = 0;
for (int num = 10; num <= 40; num += 10)
{
    value = value + num;
}
System.out.println(value);
```

SR 6.14 What output is produced by the following code fragment?

```
int value = 0;
for (int num = 10; num < 40; num += 10)
{
    value = value + num;
}
System.out.println(value);
```

SR 6.15 What output is produced by the following code fragment?

```
int value = 6;
for (int num = 1; num <= value; num ++)
```

```
{  
    for (int i = 1; i <= (value - num); i++)  
        System.out.print(" ");  
    for (int i = 1; i <= ((2 * num) - 1); i++)  
        System.out.print("*");  
    System.out.println();  
}
```

SR 6.16 Assume `die` is a `Die` object (as defined in [Section 4.2](#)). Write a code fragment that will roll `die` 100 times and output the average value rolled.

Summary of Key Concepts

- A `switch` statement matches a character or integer value to one of several possible cases.
- A `break` statement is usually used at the end of each case alternative of a `switch` statement.
- The conditional operator evaluates to one of two possible values based on a boolean condition.
- A `do` statement executes its loop body at least once.
- A `for` statement is usually used when a loop will be executed a set number of times.
- The for-each version of a `for` loop simplifies the processing of all elements in an `Iterable` object.
- The loop statements are functionally equivalent. Which one you use should depend on the situation.
- A transformation changes the visual presentation of a node.
- Use the same scaling factor on both axes to keep a node in proportion to the original.
- A transformation applied to a group or pane is applied automatically to all nodes in the container.

Exercises

EX 6.1 How many iterations will the following for loops execute?

- a. `for (int i = 0; i < 20; i++) { }`
- b. `for (int i = 1; i <= 20; i++) { }`
- c. `for (int i = 5; i < 20; i++) { }`
- d. `for (int i = 20; i > 0; i--) { }`
- e. `for (int i = 1; i < 20; i = i + 2) { }`
- f. `for (int i = 1; i < 20; i *= 2) { }`

EX 6.2 What output is produced by the following code fragment?

```
for (int num = 0; num <= 200; num += 2)
    System.out.println(num);
```

EX 6.3 What output is produced by the following code fragment?

```
for (int val = 200; val >= 0; val -= 1)
    if (val % 4 != 0)
        System.out.println(val);
```

EX 6.4 Transform the following `while` loop into an equivalent `do` loop (make sure it produces the same output).

```
int num = 1;
while (num < 20)
{
    num++;
    System.out.println(num);
}
```

EX 6.5 Transform the `while` loop from the previous exercise into an equivalent `for` loop (make sure it produces the same output).

EX 6.6 Write a `do` loop that verifies that the user enters an even integer value.

EX 6.7 Write a `for` loop to print the odd numbers from 1 to 99 (inclusive).

EX 6.8 Write a `for` loop to print the multiples of 3 from 300 down to 3.

EX 6.9 Write a code fragment that reads 10 integer values from the user and prints the highest value entered.

EX 6.10 Write a code fragment that determines and prints the number of times the character `'a'` appears in a `String` object called `name`.

EX 6.11 Write a code fragment that prints the characters stored in a `String` object called `str` backward.

EX 6.12 Write a code fragment that prints every other character in a `String` object called `word` starting with the first character.

EX 6.13 Write a method called `powersOfTwo` that prints the first 10 powers of 2 (starting with 2). The method takes no parameters and doesn't return anything.

EX 6.14 Write a method called `alarm` that prints the string `"Alarm!"` multiple times on separate lines. The method should accept an integer parameter that specifies how many times the string is printed. Print an error message if the parameter is less than 1.

EX 6.15 Write a method called `sum100` that returns the sum of the integers from 1 to 100, inclusive.

EX 6.16 Write a method called `sumRange` that accepts two integer parameters that represent a range. Issue an error message and return zero if the second parameter is less than the first. Otherwise, the method should return the sum of the integers in that range (inclusive).

EX 6.17 Write a method called `countA` that accepts a `String` parameter and returns the number of times the character `'A'` is found in the string.

EX 6.18 Write a method called `reverse` that accepts a `String` parameter and returns a string that contains the characters of the parameter in reverse order. Note that there is a method in the `String` class that performs this operation, but for the sake of this exercise, you are expected to write your own.

EX 6.19 In the `Bullseye` program, what is the fill color of the innermost circle before it is changed to red? Explain.

EX 6.20 Given the way the `Boxes` program is written, what color will a rectangle be if it is both narrow and short? Explain.

EX 6.21 Rewrite the `if` statement used in the `Boxes` program so that if a box is both narrow and short, its fill color will be orange. Otherwise, keep narrow boxes yellow and short boxes green.

EX 6.22 Write code that will shift a `Rectangle` named `rec` 50 pixels right and 10 pixels down, rotate it 45 degrees clockwise, and display it at half its original size.

EX 6.23 Write code that will invert (turn upside down) a `ImageView` named `pic` and display it at twice its original size.

EX 6.24 What happens when you apply a transformation to a group?

Programming Projects

PP 6.1 Write a program that reads an integer value and prints the sum of all even integers between 2 and the input value, inclusive. Print an error message if the input value is less than 2. Prompt accordingly.

PP 6.2 Write a program that reads a string from the user and prints it one character per line.

PP 6.3 Write a program that produces a multiplication table, showing the results of multiplying the integers 1 through 12 by themselves.



Developing a solution of **PP**

PP 6.4 Write a program that prints the first few verses of the traveling song “One Hundred Bottles of Beer.” Use a loop such that each iteration prints one verse. Read the number of verses to print from the user. Validate the input. The following are the first two verses of the song:

100 bottles of beer on the wall

100 bottles of beer

If one of those bottles should happen to fall

99 bottles of beer on the wall

99 bottles of beer on the wall

99 bottles of beer

If one of those bottles should happen to fall

98 bottles of beer on the wall

PP 6.5 Using the `PairOfDice` class from **PP 4.9**, write a program that rolls a pair of dice 1000 times, counting the number of box cars (two sixes) that occur.

PP 6.6 Using the `Coin` class defined in **Chapter 5**, write a program called `CountFlips` whose `main` method flips a coin 100 times and counts how many times each side comes up. Print the results.

PP 6.7 Create modified versions of the `Stars` program to print the following patterns. Create a separate program to produce each pattern. *Hint:* Parts b, c, and d require several loops, some of which print a specific number of spaces.

a.

```
*****
*****
*****
*****
*****
*****
*****
```

**

*

b.

*

**

c.

**

*

d.

*

```
*****  
*****  
*****  
*****  
***  
*
```

PP 6.8 Write a program that prints a table showing a subset of the Unicode characters and their numeric values. Print five number/character pairs per line, separated by tab characters. Print the table for numeric values from 32 (the space character) to 126 (the ~ character), which corresponds to the printable ASCII subset of the Unicode character set. Compare your output to the table in [Appendix C](#). Unlike the table in [Appendix C](#), the values in your table can increase as they go across a row.

PP 6.9 Write a program that reads a string from the user, then determines and prints how many of each lowercase vowel (a, e, i, o, and u) appear in the entire string. Have a separate counter for each vowel. Also count and print the number of nonvowel characters.

PP 6.10 Write a program that prints the verses of the song “The Twelve Days of Christmas,” in which each verse adds one line. The first two verses of the song are:

On the 1st day of Christmas my true love gave to me

A partridge in a pear tree.

On the 2nd day of Christmas my true love gave to me

Two turtle doves, and

A partridge in a pear tree.

Use a `switch` statement in a loop to control which lines get printed. *Hint:* Order the cases carefully and avoid the `break` statement. Use a separate `switch` statement to put the appropriate suffix on the day number (1st, 2nd, 3rd, etc.). The final verse of the song involves all 12 days, as follows:
On the 12th day of Christmas, my true love gave to me

Twelve drummers drumming,

Eleven pipers piping,

Ten lords a-leaping,

Nine ladies dancing,

Eight maids a-milking,

Seven swans a-swimming,

Six geese a-laying,

Five golden rings,

Four calling birds,

Three French hens,

Two turtle doves, and

A partridge in a pear tree.

PP 6.11 Write a JavaFX application that draws 20 horizontal, evenly spaced parallel lines of random length.

PP 6.12 Write a JavaFX application that displays an 8*8 checkerboard with 64 squares, alternating black and white.

PP 6.13 Write a JavaFX application that draws 10 concentric circles of random radius.

PP 6.14 Write a JavaFX application that draws 100 circles of random color and random size in random locations. Ensure that the entire circle appears in the visible area of the scene.

PP 6.15 Write a JavaFX application that displays 10,000 very small circles (radius of 1 pixel) in random locations within the visible area. Fill the dots on the left half of the scene red and the dots on the right half of the scene green. Use the `getWidth` method of the scene to help determine the halfway point.

PP 6.16 Write a JavaFX application that displays a brick wall pattern in which each row of bricks is offset from the row above and below it.

PP 6.17 Write a JavaFX application called `Quilt` that displays a quilt made up of two alternating square patterns. Define a class called `QuiltSquare` that represents a pattern of your choice. Allow the constructor of the `QuiltSquare` class to vary some characteristics of the pattern, such as its color scheme. Instantiate two `QuiltSquare` objects and incorporate them in a checkerboard layout in the quilt.

PP 6.18 Write a JavaFX application that draws 10 circles of random radius in random locations. Leave all circles unfilled except for the largest circle, which should be filled with a translucent red (30% opaque). If multiple circles have the same

largest size, fill any one of them. Hint: keep track of the largest circle as you generate them; then change its fill color at the end.

PP 6.19 Write a JavaFX application that displays the same square image four times, once right side up, then on its right side, then upside down, and finally, on its left side.

PP 6.20 Write a JavaFX application that displays a series of ellipses with the same center point. Each one should be slightly rotated, creating a pinwheel effect. Use a loop to create the shapes.

PP 6.21 Write a JavaFX application that displays a group of four alien spaceships in space. Define the spaceship once in a separate class, made up of whatever shapes you'd like. Then create four of them to be displayed, adjusting the position and size of each ship to make it appear that some are closer than others. Include a field of randomly generated stars (small white dots) behind the ships.