GLOBAL EDITION

Java™ Software Solutions

Foundations of Program Design

NINTH EDITION

Lewis • Loftus

Pearson

Digital Resources for Students

Your new textbook provides 12-month access to digital resources that may include VideoNotes (step-by-step video tutorials on programming concepts), source code, web chapters, quizzes, and more. Refer to the preface in the textbook for a detailed list of resources.

Follow the instructions below to register for the Companion Website for John Lewis and William Loftus' *Java™ Software Solutions,* Ninth Edition, Global Edition.

1. Go to www.pearsonglobaleditions.com/Lewis
2. Select your textbook and click Companion Website.
3. Click Register and follow the on-screen instructions to create a login name and password.

Use a coin to scratch off the coating and reveal your access code.
Do not use a sharp knife or other sharp object as it may damage the code.

Use the login name and password you created during registration to start using the online resources that accompany your textbook.

IMPORTANT

This prepaid subscription does not include access to Pearson MyLab Programming, which is available at www.myprogramminglab.com for purchase.

This access code can only be used once. This subscription is valid for 12 months upon activation and is not transferable. If the access code has already been revealed, it may no longer be valid.

For technical support, go to https://support.pearson.com/getsupport 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:

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

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

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.");
```

KEY CONCEPT

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

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.

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?

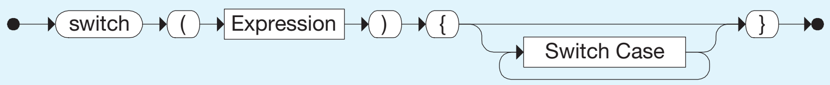
SR 5.15 Write an equals method for the Die class of Section 4.2. The method should return true if the Die object it is invoked on has the same faceValue as the Die object passed as a parameter, otherwise it should return false.

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.

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);
}
```

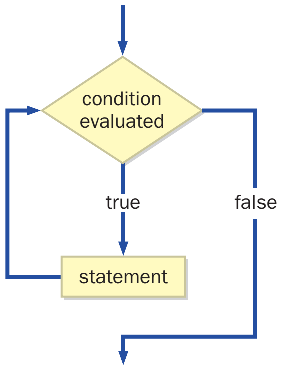


FIGURE 5.7 The logic of a `while` loop

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.

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

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);
```

LISTING 5.7 *continued*

```
      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
```