Chapter 8

What Java Does (and When)

In This Chapter
- Making decisions with Java statements
- Repeating actions with Java statements

Human thought centers around nouns and verbs. Nouns are the “stuff,” and verbs are the stuff’s actions. Nouns are the pieces, and verbs are the glue. Nouns are, and verbs do. When you use nouns, you say “book,” “room,” or “stuff.” When you use verbs, you say “do this,” “do that,” “tote that barge,” or “lift that bale.”

Java also has nouns and verbs. Java’s nouns include int, JOptionPane, and String, along with Android-specific terms such as Activity, Application, and Bundle. Java’s verbs involve assigning values, choosing among alternatives, repeating actions, and taking other courses of action.

This chapter covers some of Java’s verbs. (In the next chapter, I bring in the nouns.)

Making Decisions

When you’re writing computer programs, you’re continually hitting forks in roads. Did the user type the correct password? If the answer is yes, let the user work; if it’s no, kick the bum out. The Java programming language needs a way to make a program branch in one of two directions. Fortunately, the language has a way: It’s the if statement. The use of the if statement is illustrated in Listing 8-1.
Listing 8-1: Using an if Statement

```java
package com.allmycode.tickets;

import javax.swing.JOptionPane;

public class TicketPrice {

    public static void main(String[] args) {
        String ageString;
        int age;
        String specialShowingString;
        String price;

        ageString = JOptionPane.showInputDialog("Age?");
        age = Integer.parseInt(ageString);

        specialShowingString = JOptionPane.showInputDialog("Special showing (y/n)?");

        if ((age < 18 || 65 <= age) && specialShowingString.equals("n")) {
            price = "$7.00";
        } else {
            price = "$10.00";
        }

        JOptionPane.showMessageDialog(null, price, "Ticket price", JOptionPane.INFORMATION_MESSAGE);
    }
}
```

Listing 8-1 revives a question that I pose originally in Chapter 6: How much should a person pay for a movie ticket? Most people pay $10. But when the movie has no special showings, youngsters (under 18) and seniors (65 and older) pay only $7.

In Listing 8-1, a Java if statement determines a person’s eligibility for the discounted ticket. If this condition is true:

```
(age < 18 || 65 <= age) && specialShowingString.equals("n")
```

the price becomes "$7.00"; otherwise, the price becomes "$10.00". In either case, the code displays the price in a message box. (See Figure 8-1.)
Testing for equality

Java has several ways to test for equality: “Is this value the same as that value?” None of these ways is the first one you’d consider. In particular, to find out whether someone’s age is 35, you don’t write if (age = 35). Instead, you use a double equal sign (==): if (age == 35). In Java, the single equal sign (=) is reserved for assignment. So age = 35 means “Let age stand for the value 35”, and age == 35 means “True or false: Does age stand for the value 35?”

Comparing two strings is a different story. When you compare two strings, you don’t use the double equal sign. Using it would ask a question that’s usually not what you want to ask: “Is this string stored in exactly the same place in memory as that other string?” Instead, you usually ask, “Does this string have the same characters in it as that other string?” To ask the second question (the more appropriate one), use Java’s equals method. To call this equals method, follow one of the two strings with a dot and the word equals, and then with a parameter list containing the other string:

```java
if (specialShowingString.equals("n")) {
```

The equals method compares two strings to see whether they have the same characters in them. In this paragraph’s tiny example, the variable `specialShowingString` refers to a string, and the text "n" refers to a string. The condition `specialShowingString.equals("n")` is true if `specialShowingString` refers to a string whose only character is the letter n.
Java if statements

An if statement has this form:

```java
if (condition) {
  statements to be executed when the condition is true
} else {
  statements to be executed when the condition is false
}
```

In Listing 8-1, the condition being tested is

```java
(age < 18 || 65 <= age) &&
specialShowingString.equals("n")
```

The condition is either true or false — true for youngsters and seniors when there’s no special showing and false otherwise.

Conditions in if statements

The condition in an if statement must be enclosed in parentheses. The condition must be a boolean expression — an expression whose value is either true or false. For example, the following condition is okay:

```java
if (numberOfTries < 17) {
  statements to be executed
}
```

But the strange kind of condition that you can use in other (non-Java) languages — languages such as C++ — is not okay:

```java
if (17) {
  //This is incorrect.
}
```

See Chapter 6 for information about Java’s primitive types, including the boolean type.

Omitting braces

You can omit an if statement’s curly braces when only one statement appears between the condition and the word else. You can also omit braces when only one statement appears after the word else. For example, the following chunk of code is right and proper:

```java
if ((age < 18 || 65 <= age) &&
    specialShowingString.equals("n"))
  price = "$7.00";
else
  price = "$10.00";
```
The code is correct because only one statement (price = "$7.00") appears between the condition and the else, and only one statement (price = "$10.00") appears after the word else.

An if statement can also enjoy a full and happy life without an else part. The following example contains a complete if statement:

```java
price = "$10.00";
if ((age < 18 || 65 <= age) &&
   specialShowingString.equals("n"))
   price = "$7.00";
```

**Compound statements**

An if statement is one of Java’s compound statements because an if statement normally contains other Java statements. For example, the if statement in Listing 8-1 contains the assignment statement price = "$7.00" and the other assignment statement contains price = "$10.00".

A compound statement might even contain other compound statements. In this example:

```java
price = "$10.00";
if (age < 18 || 65 <= age) {
   if (specialShowingString.equals("n")) {
      price = "$7.00";
   }
}
```

one if statement (with the condition age < 18 || 65 <= age) contains another if statement (with the condition specialShowingString. equals("n")).

**A detour concerning Android screen densities**

A device’s screen density is the number of pixels squeezed into each inch of the screen. Older devices and less expensive devices have low screen densities, and newer, more expensive devices compete to have increasingly higher screen densities.

Android supports a wide range of screen densities. It also goes to the trouble of grouping the densities, as I show in Table 8-1.
Table 8-1 Android Screen Densities

<table>
<thead>
<tr>
<th>Name</th>
<th>Acronym</th>
<th>Approximate* Number of Dots per Inch (dpi)</th>
<th>Fraction of the Default Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>DENSITY_LOW</td>
<td>ldpi</td>
<td>120</td>
<td>$\frac{3}{4}$</td>
</tr>
<tr>
<td>DENSITY_MEDIUM</td>
<td>mdpi</td>
<td>160</td>
<td>1</td>
</tr>
<tr>
<td>DENSITY_HIGH</td>
<td>hdpi</td>
<td>240</td>
<td>$\frac{1}{5}$</td>
</tr>
<tr>
<td>DENSITY_XHIGH</td>
<td>xhdpi</td>
<td>320</td>
<td>2</td>
</tr>
<tr>
<td>DENSITY_XXHIGH</td>
<td>xxhdpi</td>
<td>480</td>
<td>3</td>
</tr>
</tbody>
</table>

*When the screen density of a device doesn't match a number in Column 3 of Table 8-1, Android does its best with the existing categories. For example, Android classifies density 265 dpi in the hdpi group.

Fun facts: DENSITY_XHIGH is the same as 1080p high-definition television in the United States. A seldom-used Android density, DENSITY_TV with 213 dpi, represents 720p television.

Screen densities can make a big difference. An image that looks good on a low-density screen might look choppy on a high-density screen. And an image designed for a high-density screen might be much too large for a low-density screen. That’s why, when you create a new application, Android offers to create several different icons for your app. (See Figure 8-2. And I’m sorry, Paul — it’s another cat picture!)

Figure 8-2: One icon; many sizes.
Choosing among many alternatives

A Java if statement creates a fork in the road: The computer chooses between two alternatives. But some problems lend themselves to forks with many prongs. What’s the best way to decide among five or six alternative actions?

For me, multipronged forks are scary. In my daily life, I hate making decisions. (If a problem crops up, I would rather have it be someone else’s fault.) So, writing the previous sections (on making decisions with Java’s if statement) knocked the stuffing right out of me. That’s why my mind boggles as I begin this section on choosing among many alternatives.

To prepare for this section’s example, I created the four icons shown in Figure 8-2. The icons are for four of the densities depicted in Table 8-1. I have a medium-density icon, a high-density icon, an extra-high-density icon, and an extra-extra-high-density icon.

I named each icon cat.png and placed the four icons into four different folders. I added a fifth folder for the ic_dialog_alert.png icon, as shown in Figure 8-3.

The folder structure matches the one you’d see in an Android app. To keep the example simple, I created a plain, old Java program to display the icons. The program is shown in Listing 8-2.

Listing 8-2: Switching from One Icon to Another

```java
package com.allmycode.icons;

import javax.swing.ImageIcon;
import javax.swing.JOptionPane;

public class ShowIcons {
    public static void main(String[] args) {
        String densityCodeString = JOptionPane
(continued)```
Listing 8-2 (continued)

```java
.d.showInputDialog("Density?");

int densityCode =
    Integer.parseInt(densityCodeString);
String iconFileName = null, message = null;

switch (densityCode) {
    case 160:
        iconFileName = "res/drawable-mdpi/cat.png";
        message = "mdpi";
        break;
    case 240:
        iconFileName = "res/drawable-hdpi/cat.png";
        message = "hdpi";
        break;
    case 320:
        iconFileName = "res/drawable-xhdpi/cat.png";
        message = "xhdpi";
        break;
    case 480:
        iconFileName = "res/drawable-xxhdpi/cat.png";
        message = "xxhdpi";
        break;
    default:
        iconFileName = "res/drawable/ic_dialog_alert.png";
        message = "No suitable icon";
        break;
}

ImageIcon icon = new ImageIcon(iconFileName);
JOptionPane.showMessageDialog(null, message,
    "Icon", JOptionPane.INFORMATION_MESSAGE, icon);
```

The code in Listing 8-2 is a standard Oracle Java program. The code illustrates some ideas about Android screen densities, but the program is not an Android application. This program can’t run on an Android device. In Chapter 10, I begin building some examples that run on Android devices.

In Listing 8-2, the program asks the user to enter a screen-density value. If the user types 160, for example, the program responds by displaying my medium-density icon (the image in the cat.png file in my res/drawable-mdpi directory). Two runs of the program are shown in Figure 8-4.

Why the medium-density icon? The program enters the `switch` statement in Listing 8-2. The `switch` statement contains an expression (the value of `densityCode`). The `switch` statement also contains `case` clauses, followed
(optionally) by a default clause. The program compares the value of densityCode with 160 (the number in the first of the case clauses). If the value of densityCode is equal to 160, the program executes the statements after the words case 160.

![Figure 8-4: Running the code shown in Listing 8-2.](image)

In Listing 8-2, the statements after case 160 are

```java
iconFileName = "res/drawable-mdpi/cat.png";
message = "mdpi";
break;
```

The first two statements set the values of iconFileName and message in preparation for the display of a message box. The third statement (the break statement) jumps out of the entire switch statement, skipping past all the other case clauses and past the default clause to get to the last part of the program.

After the switch statement, the statement

```java
ImageIcon icon = new ImageIcon(iconFileName);
```

creates a new icon variable to refer to the image in the iconFileName file. (I have more to say about this kind of statement in Chapter 9.) Finally, the statement

```java
JOptionPane.showMessageDialog(null, message, "Icon", JOptionPane.INFORMATION_MESSAGE, icon);
```

displays the icon image in a message box on the user’s screen. (Refer to Figure 8-4.)
Both the Windows and Macintosh operating systems have directories (also known as folders), and these directories may contain subdirectories, which in turn may contain their own subdirectories. At the bottom of the food chain is the humble file containing a document, an image, a sound, or whatever. On my Windows computer, one of my cat.png files lives in a directory named drawable-hdpi, which is inside a directory named res, which is inside an Eclipse project directory named 08-02. The Eclipse project directory is inside my Eclipse workspace directory, which in turn is inside my Barry directory, which is inside my Users directory, as shown in the sidebar figure. It’s a long chain of stuff leading eventually to a picture of a cat.

When you’re visiting Times Square in New York City, you can say, “I’m walking to the McDonald’s on 34th Street.” You don’t have to say “I’m walking to the McDonald’s on 34th Street in New York City, USA.” In a similar way, my code doesn’t have to refer to the cat.png file by naming a whole bunch of directories and subdirectories. Instead, I can take advantage of the fact that Listing 08-02 is in my 08-02 directory. From the viewpoint of the 08-02 directory, I can refer directly to the res directory, which is contained immediately inside the 08-02 directory. In both the Windows and Macintosh operating system, I can use the forward slash character (/) to point from the 08-02 directory to my cat picture:

`res/drawable-hdpi/cat.png`

But there’s a problem. In a Java string, a single backslash (\) has a special meaning. That special meaning depends on whatever character appears immediately after the backslash. For example, \n stands for “Go to a new line,” \t stands for “Go to the next tab stop,” and \\ stands for “A single backslash.” In Listing 8-2, a double-quoted string such as "res/drawable-mdpi/cat.png" stands for res/drawable-mdpi/cat.png. To the Windows operating system, this double-backslash business is another way to refer to the cat.png file that’s in the drawable-mdpi subdirectory of the res directory.

Once again, if you’re a Mac user, you use a forward slash (/) to separate directory names, and a forward slash has no special meaning inside a Java string. Mac users don’t have to worry about doubling up on slashes.
Take a break

This news might surprise you: The end of a case clause (the beginning of another case clause) doesn’t automatically make the program jump out of the switch statement. If you forget to add a break statement at the end of a case clause, the program finishes the statements in the case clause and then continues executing the statements in the next case clause. Imagine that I write the following code (and omit a break statement):

```java
switch (densityCode) {
    case 160:
        iconFileName = "res/drawable-mdpi/cat.png";
        message = "mdpi";
    case 240:
        iconFileName = "res/drawable-hdpi/cat.png";
        message = "hdpi";
        break;
    ... Etc.
}
```

With this modified code (and with densityCode equal to 160), the program sets iconFileName to "res/drawable-mdpi/cat.png", sets message to "mdpi", sets iconFileName to "res/drawable-hdpi/cat.png", sets message to "hdpi", and, finally, breaks out of the switch statement (skipping past all other case clauses and the default clause). The result is that iconFileName has the value "res/drawable-hdpi/cat.png" (not "res/drawable-mdpi/cat.png") and that message has the value "hdpi" (not "mdpi").

This phenomenon of jumping from one case clause to another in the absence of a break statement) is called fall-through, and, occasionally, it’s useful. Imagine a dice game in which 7 and 11 are instant wins; 2, 3, and 12 are instant losses; and any other number (from 4 to 10) tells you to continue playing. The code for such a game might look like this:

```java
switch (roll) {
    case 7:
    case 11:
        message = "win";
        break;
    case 2:
    case 3:
    case 12:
        message = "lose";
        break;
    case 4:
    case 5:
    case 6:
    case 8:
    case 9:
...
}```
case 10:
    message = "continue";
    break;
default:
    message = "not a valid dice roll";
    break;
}

If you roll a 7, you execute all the statements immediately after case 7 (of which there are none), and then you fall-through to case 11, executing the statement that assigns "win" to the variable message.

Every beginning Java programmer forgets to put a break statement at the end of a case clause. When you make this mistake, don’t beat yourself up about it. Just remember what’s causing your program’s unexpected behavior, add break statements to your code, and move on. As you gain experience in writing Java programs, you’ll make this mistake less and less frequently. (You’ll still make the mistake occasionally, but not as often.)

**The computer selects a case clause**

When you run the code in Listing 8-2, the user doesn’t have to enter the number 160. If the user enters 320, the program skips past the statements in the case 160 clause and then skips past the statements in the 240 clause. The program hits pay dirt when it reaches the case 320 clause, and executes that clause’s statements, making iconFileName be "res/drawable-xhdpi/cat.png" and making message be xhdpi. The case clause’s break statement makes the program skip the rest of the stuff in the switch statement.

**The default clause**

A switch statement’s optional default clause is a catchall for values that don’t match any of the case clauses’ values. For example, if you run the program and the user enters the number 265, the program doesn’t fix on any of the case clauses. (To select a switch statement’s case clause, the value after the word switch has to be an exact match of the value after the word case.) So if densityCode is 265, the program skips past all the case clauses and executes the code in the default clause, making iconFileName be "res/drawable/ic_dialog_alert.png" and making message be "No suitable icon". In this way, the program in Listing 8-2 doesn’t mirror Android’s screen-resolution tricks. (Android uses an existing icon even if the screen’s density doesn’t exactly match one of the numbers 160, 240, 320, or 480.)
The last break statement in Listing 8-2 tells the computer to jump to the end of the switch statement, skipping any statements after the default clause. But look again. Nothing comes after the default clause in the switch statement! Which statements are being skipped? The answer is none. I put a break at the end of the default clause for good measure. This extra break statement doesn’t do anything, but it doesn’t do any harm, either.

Some formalities concerning Java switch statements

A switch statement has the following form:

```java
switch (expression) {
    case constant1:
        statements to be executed when the expression has value constant1
    case constant2:
        statements to be executed when the expression has value constant2
    case ...
    default:
        statements to be executed when the expression has a value different from any of the constants
}
```

You can’t put any old expression in a switch statement. The expression that’s tested at the start of a switch statement must have one of these elements:

- A primitive type: char, byte, short, or int
- A reference type: Character, Byte, Short, or Integer
- An enum type

An enum type is a type whose values are limited to the few that you declare. For example, the line

```java
enum TrafficSignal {GREEN, YELLOW, RED};
```

defines a type whose only values are GREEN, YELLOW, and RED. Elsewhere in your code, you can write
TrafficSignal signal;
signal = TrafficSignal.GREEN;

to make use of the TrafficSignal type.

Starting with Java 7, you can put a String type expression at the start of a switch statement. But the last time I checked, Java 5 or 6 is required for developing Android code. You can’t use Java 7 or later to create an Android app. So with densityCodeString declared to be of type String, you can’t create a switch statement whose first line is switch (display CodeString), and you can’t have a case clause that begins with case "hdpi".

Repeating Instructions Over and Over Again

In 1966, the company that brings you Head & Shoulders shampoo made history. On the back of the bottle, the directions for using the shampoo read, “Lather, rinse, repeat.” Never before had a complete set of directions (for doing anything, let alone shampooing hair) been summarized so succinctly. People in the direction-writing business hailed it as a monumental achievement. Directions like these stood in stark contrast to others of the time. (For instance, the first sentence on a can of bug spray read, “Turn this can so that it points away from your face.” Duh!)

Aside from their brevity, the characteristic that made the Head & Shoulders directions so cool was that, with three simple words, they managed to capture a notion that’s at the heart of all instruction-giving: repetition. That last word, repeat, turned an otherwise bland instructional drone into a sophisticated recipe for action.

The fundamental idea is that when you’re following directions, you don’t just follow one instruction after another. Instead, you make turns in the road. You make decisions (“If HAIR IS DRY, then USE CONDITIONER,”) and you repeat steps (“LATHER-RINSE, and then LATHER-RINSE again.”). In application development, you use decision-making and repetition all the time.
Check, and then repeat

The program in Listing 8-2 is nice (if I say so myself). But the program has its flaws. I expect the user to type a number and for things to go wrong if the user doesn’t type a number, as shown in Figure 8-5. The program doesn’t even like numbers with decimal points.

You should anticipate all kinds of user input. To do that, you have several alternatives. One thing you can do is to dismiss bad input and ask the user for better input — so you might have to repeat your input request over and over again. Listing 8-3 shows you one way to do it.
Listing 8-3:  Look Before You Leap

package com.allmycode.icons;

import javax.swing.ImageIcon;
import javax.swing.JOptionPane;

public class ShowIconsWithWhile {

    public static void main(String[] args) {
        String densityCodeString = JOptionPane.showInputDialog("Density?");

        while ( !densityCodeString.equals("160") &&
               !densityCodeString.equals("240") &&
               !densityCodeString.equals("320") &&
               !densityCodeString.equals("480") ) {
            densityCodeString = JOptionPane
                               .showInputDialog("Invalid input. Try again:");
        }

        int densityCode = Integer.parseInt(densityCodeString);
        String iconFileName = null, message = null;

        switch (densityCode) {
            case 160:
                iconFileName = "res/drawable-mdpi/cat.png";
                message = "mdpi";
                break;
            case 240:
                iconFileName = "res/drawable-hdpi/cat.png";
                message = "hdpi";
                break;
            case 320:
                iconFileName = "res/drawable-xhdpi/cat.png";
                message = "xhdpi";
                break;
            case 480:
                iconFileName = "res/drawable-xxhdpi/cat.png";
                message = "xxhdpi";
                break;
            default:
                iconFileName = "res/drawable/ic_dialog_alert.png";
                message = "No suitable icon";
                break;
        }

        ImageIcon icon = new ImageIcon(iconFileName);
        JOptionPane.showMessageDialog(null, message,
                                        "Icon", JOptionPane.INFORMATION_MESSAGE, icon);
    }
}
A run of the code in Listing 8-3 is shown in Figure 8-6.

The code in Listing 8-3 begins by displaying an input dialog box with the "Density?" message. If the user responds with a value other than 160, 240, 320, or 480, the code dives into its while statement, displaying the message "Invalid input. Try again:" in the input dialog box over and over again. The code continues displaying this input dialog box until the user responds with one of the four valid values — 160, 240, 320, or 480.

In plain language, the while statement in Listing 8-3 says:

```java
while (densityCodeString isn't 160 and densityCodeString isn't 240 and densityCodeString isn't 320 and densityCodeString isn't 480)
{
    get a value for the densityCodeString
}
```

In even plainer language, the while statement says:
while (densityCodeString isn't acceptable) {
    get a value for the densityCodeString
}

The while statement is one of Java’s compound statements. It’s also one of Java’s looping statements because, when executing a while statement, the computer can go into a loop, spinning around and around, executing a certain chunk of code over and over again.

In a looping statement, each go-around is an iteration.

If you stare at Listing 8-3, you might notice this peculiarity: The while statement at the top of the program ensures that the density is either 160, 240, 320, or 480. But toward the end of the program, the switch statement’s default clause provides for the possibility that the density isn’t one of those 160, 240, 320, or 480 values. What gives? The answer is that it never hurts to double-check. You may think that your while statement can spit out only 160, 240, 320, or 480, but you might have forgotten about an unusual scenario that causes the density to be another, strange number. And what happens if another developer (someone trying to improve on your code) messes with your while statement and lets bad density values trickle over to the switch statement? Adding a default clause to a switch statement is never costly, and the default clause always adds an extra layer of protection from errors.

Some formalities concerning Java while statements

A while statement has this form:

```java
while (condition) {
    statements inside the loop
}
```

The computer repeats the statements inside the loop over and over again as long as the condition in parentheses is true:

Check to make sure that the condition is true;
Execute the statements inside the loop.

Check again to make sure that the condition is true;
Execute the statements inside the loop.

Check again to make sure that the condition is true;
Execute the statements inside the loop.

And so on.
At some point, the *while* statement’s condition becomes false. (Generally, this happens because one of the statements in the loop changes one of the program’s values.) When the condition becomes false, the computer stops repeating the statements in the loop. (That is, the computer stops *iterating*.) Instead, the computer executes whatever statements appear immediately after the end of the *while* statement:

```
Check again to make sure that the condition is true;
Execute the statements inside the loop.
Check again to make sure that the condition is true;
Execute the statements inside the loop.
Check again to make sure that the condition is true;
Oops! The condition is no longer true!
Execute the code immediately after the *while* statement.
```

In Listing 8-3, the code

```
int densityCode =
    Integer.parseInt(densityCodeString);
```

comes immediately after the end of the *while* statement.

**Variations on a theme**

Many of the *if* statement’s tricks apply to *while* statements as well. A *while* statement is a compound statement, so it might contain other compound statements. And when a *while* statement contains only one statement, you can omit curly braces. So the following code is equivalent to the *while* statement in Listing 8-3:

```
while ( !densityCodeString.equals("160") &&
    !densityCodeString.equals("240") &&
    !densityCodeString.equals("320") &&
    !densityCodeString.equals("480")    )
    densityCodeString = JOptionPane
        .showInputDialog("Density?");
```

After all, the code

```
densityCodeString = JOptionPane
    .showInputDialog("Density?");
```

is only one (admittedly large) assignment statement.
A `while` statement’s condition might become false in the middle of an iteration, before all the iteration’s statements have been executed. When this happens, the computer doesn’t stop the iteration dead in its tracks. Instead, the computer executes the rest of the loop’s statements. After executing the rest of the loop’s statements, the computer checks the condition (finding the condition to be false) and marches on to whatever code comes immediately after the `while` statement.

The previous icon should come with some fine print. To be painfully accurate, I should point out a few ways for you to stop abruptly in the middle of a loop iteration. You can execute a `break` statement to jump out of a `while` statement immediately. (It’s the same `break` statement that you use in a `switch` statement.) Alternatively, you can execute a `continue` statement (the word `continue`, followed by a semicolon) to jump abruptly out of an iteration. When you jump out with a `continue` statement, the computer ends the current iteration immediately and then checks the `while` statement’s condition. A true condition tells the computer to begin the next loop iteration. A false condition tells the computer to go to whatever code comes after the `while` statement.

**Priming the pump**

Java’s `while` statement uses the policy “Look before you leap.” The computer always checks a condition before executing the statements inside the loop. Among other things, this forces you to prime the loop. When you prime a loop, you create statements that affect the loop’s condition before the beginning of the loop. (Think of an old-fashioned water pump and how you have to prime the pump before water comes out.) In Listing 8-3, the initialization in

```java
String densityCodeString = 
JOptionPane.showInputDialog("Density?");
```

primes the loop. This initialization — the `=` part — gives `densityCodeString` its first value so that when you check the condition `!densityCodeString.equals("160") && ... Etc.` for the first time, the variable `densityCodeString` has a value that’s worth comparing.

Here’s something you should consider when you create a `while` statement: The computer can execute a `while` statement without ever executing the statements inside the loop. For example, the code in Listing 8-3 prompts the user one time before the `while` statement. If the user enters a good density value, the `while` statement’s condition is false. The computer skips past the statement inside the loop and goes immediately to the code after the `while` statement. The computer never displays the `Invalid input. Try again` prompt.
Repeat, and then check

The `while` statement (which I describe in the previous section) is the workhorse of repetition in Java. Using `while` statements, you can do any kind of looping that you need to do. But sometimes it’s convenient to have other kinds of looping statements. For example, occasionally you want to structure the repetition so that the first iteration takes place without checking a condition. In that situation, you use Java’s `do` statement. Listing 8-4 is almost the same as Listing 8-3. But in Listing 8-4, I replace a `while` statement with a `do` statement.

Listing 8-4: Leap before You Look

```java
package com.allmycode.icons;
import javax.swing.ImageIcon;
import javax.swing.JOptionPane;
public class ShowIconsWithDo {
    public static void main(String[] args) {
        String densityCodeString =
            JOptionPane.showInputDialog("Density?");
        do {
            densityCodeString = JOptionPane
                .showInputDialog("Density?");
        } while ( !densityCodeString.equals("160") &&
            !densityCodeString.equals("240") &&
            !densityCodeString.equals("320") &&
            !densityCodeString.equals("480") );
        int densityCode =
            Integer.parseInt(densityCodeString);
        String iconFileName = null, message = null;
        switch (densityCode) {
            case 160:
                iconFileName = "res/drawable-mdpi/cat.png";
                message = "mdpi";
                break;
            case 240:
                iconFileName = "res/drawable-hdpi/cat.png";
                message = "hdpi";
                break;
            case 320:
```

(continued)
Listing 8-4 (continued)

```java
iconFileName = "res/drawable-xhdpi/cat.png";
message = "xhdpi";
break;
case 480:
    iconFileName = "res/drawable-xxhdpi/cat.png";
    message = "xxhdpi";
    break;
default:
    iconFileName = "res/drawable/ic_dialog_alert.png";
    message = "No suitable icon";
    break;
}

ImageIcon icon = new ImageIcon(iconFileName);
JOptionPane.showMessageDialog(null, message, "Icon", JOptionPane.INFORMATION_MESSAGE, icon);
```

With a do statement, the computer jumps right in, takes action, and then checks a condition to see whether the result of the action is what you want. If it is, execution of the loop is done. If not, the computer goes back to the top of the loop for another go-round.

Some formalities concerning Java do statements

A do statement has the following form:

```java
do {
    statements inside the loop
} while (condition)
```

The computer executes the statements inside the loop and then checks to see whether the condition in parentheses is true. If the condition in parentheses is true, the computer executes the statements inside the loop again. And so on.

Java's do statement uses the policy “Leap before you look.” The statement checks a condition immediately after each iteration of the statements inside the loop.
A do statement is good for situations in which you know for sure that
you should perform the loop’s statements at least once. Unlike a while
statement, a do statement generally doesn’t need to be primed. On the
downside, a do statement doesn’t lend itself to situations in which the first
occurrence of an action is slightly different from subsequent occurrences.
For example, with the properly primed while statement in Listing 8-3, the
message in the first input dialog box is Density? and all subsequent messages
say Invalid input. Try again. With the do statement in Listing 8-4, all
input dialog boxes simply say Density?.

**Count, count, count**

This section’s example is a kludge.

*kludge (klooj) n. Anything that solves a problem in an awkward way,
either to fix the problem quickly or (in Chapter 8 of Java Programming For
Android Developers For Dummies) to illustrate a point.*

In fact, after examining this example, you might wonder whether anyone ever
uses the Java feature that’s illustrated in this section. Well, this section’s
feature (the for statement) appears quite frequently in Java programs. Life
is filled with examples of counting loops, and app development mirrors life —
or is it the other way around? When you tell a device what to do, you’re often
telling it to display three lines, process ten accounts, dial a million phone
numbers, or whatever.

For example, to display the first thousand rows of an Android data table, you
might use this Java for statement:

```java
cursor.moveToFirst();
for (int i = 0; i < 999; i++) {
    String _id = cursor.getString(0);
    String name = cursor.getString(1);
    String amount = cursor.getString(2);
    textViewDisplay.append(i + " : "+ _id + " "+
                    name + " "+ amount + "\n");
    cursor.moveToNext();
}
```

Unfortunately, examples involving Android’s data tables and phone numbers
can be quite complicated. Start with a simple example — one that displays
icons in three different sizes. Listing 8-5 has the code.
Listing 8-5: A Loop That Counts

```java
package com.allmycode.icons;
import javax.swing.ImageIcon;
import javax.swing.JOptionPane;
public class ShowIconsWithFor {
    public static void main(String[] args) {
        int densityCode;
        String iconFileName = null, message = null;
        for (int i = 1; i <= 3; i++) {
            densityCode = i * 160;
            switch (densityCode) {
                case 160:
                    iconFileName = "res/drawable-mdpi/cat.png";
                    message = "mdpi";
                    break;
                case 240:
                    iconFileName = "res/drawable-hdpi/cat.png";
                    message = "hdpi";
                    break;
                case 320:
                    iconFileName = "res/drawable-xhdpi/cat.png";
                    message = "xhdpi";
                    break;
                case 480:
                    iconFileName = "res/drawable-xxhdpi/cat.png";
                    message = "xxhdpi";
                    break;
                default:
                    iconFileName = "res/drawable/ic_dialog_alert.png";
                    message = "No suitable icon";
                    break;
            }
            ImageIcon icon = new ImageIcon(iconFileName);
            JOptionPane.showMessageDialog(null, message, "Icon", JOptionPane.INFORMATION_MESSAGE, icon);
        }
    }
}
```
Listing 8-5 declares an `int` variable named `i`. The starting value of `i` is 1. As long as the condition `i <= 3` is true, the computer executes the statements inside the loop and then executes `i++` (adding 1 to the value of `i`). After three iterations, the value of `i` gets to be 4, in which case the condition `i <= 3` is no longer true. At that point, the program stops repeating the statements inside the loop and moves on to execute any statements that come after the `for` statement. (Ha-ha! Listing 8-5 has no statements after the `for` statement!)

In this example, the statements inside the loop include

```java
densityCode = i * 160;
```

which makes `densityCode` be either 160, 320, or 480 (depending on the value of `i`). The loop’s statements also include a big `switch` statement (which creates `icon` and `message` values from the `densityCode`) and a couple of statements to display the icon and the message. The result is the display, one after another, of the three icons for the three densities 160, 320, and 480. Listing 8-5 displays all three icons, one after another, without ever getting input from the user, as shown in Figure 8-7.

---

**Figure 8-7:**
One run of the code in Listing 8-5.
Some formalities concerning Java for statements

A for statement has the following form:

```java
for (initialization ; condition ; update) {
    statements inside the loop
}
```

- An initialization (such as `int i = 1` in Listing 8-5) defines the action to be taken before the first loop iteration.
- A condition (such as `i <= 3` in Listing 8-5) defines the element to be checked before an iteration. If the condition is true, the computer executes the iteration. If the condition is false, the computer doesn’t execute the iteration, and it moves on to execute whatever code comes after the for statement.
- An update (such as `i++` in Listing 8-5) defines an action to be taken at the end of each loop iteration.

You can omit the curly braces when only one statement is inside the loop.

What’s Next?

This chapter describes several ways to jump from one place in your code to another.

Java provides other ways to move from place to place in a program, including enhanced for statements and try statements. But descriptions of these elements don’t belong in this chapter. To understand the power of enhanced for statements and try statements, you need a firm grasp of classes and objects, so Chapter 9 dives fearlessly into the classes-and-objects waters.

I’m your swimming instructor. Everyone into the pool!