Chapter 2: How Branches Work in Java

How Branches Work in Java

A fork in the road

I have to apologize for the picture on the left above, but when I saw it I had to include it as a humorous example of a fork in the road. The one on the right more closely illustrates how branches in programs work.

In Java, statements that make decisions are *branches*, and they have several forms. The most common form is Java's if statement, which can itself take two formats. Here's a definition of the simpler format:

if (<*boolean expression*>) <*body*>

This definition also introduces a form that you'll see often. A term in *angle brackets* (< and >) is a formal notation that represents some programming language feature that we'll define and that can be replaced by any valid example of that feature. For example, <*boolean expression*> represents any expression that results in a boolean value of true or false.

We'll look at how to write Boolean expressions in a minute. Right now I want to finish my description of Java's if statement format.

In that statement, <*body*> represents any single Java statement or any number of Java statements enclosed in *curly brackets* (**{** and **}**). A statement or group of statements in the body of the code execute if and only if the Boolean expression's value is true. Java programmers generally indent the body so that it's clear that it only runs if the if is true.

Here's an example of an if statement.

```
if (x < 0)
    x = -x;
```

In this example, the body is a single statement: "x = -x;". Java will execute the one statement in the body if and only if the value in x is less than zero when the if statement runs. Because this example has only one statement, there's no need to include curly brackets.

Here's a second example:

```
if (x < y) {
    z = x;
    x = y;
    y = z;
}
```

And here's an animation to help you understand how if statements work. Click the red button, and then click **Next Step** to see each step of the process.

Text equivalent start.

When you initialize integer variables x and y to 6 and 9, then the statement if x less than y evaluates to true, since 6 is less than 9. The code in the if block is executed.

Text equivalent stop.

This example checks if x is less than y and if so, swaps their values using z as a temporary holding spot for one of the values. In this case, all three statements in the body will execute if the value of x is less than the value in y when the if statement runs.

This form of the if statement checks a condition and takes action if the condition is true. If the condition is false, the if statement takes no action. But what if we want to do something else when the condition is false? Do we need another if statement? No!

Remember I said the if statement has two formats? The second format tells Java to do one thing if the condition is true and something else if it's false:

```
if (<boolean expression>)
    <body T>
else
    <body F>
```

This form of the if statement executes *<body T>* when *<boolean expression>* is true and executes *<body F>* when the condition is false.

An example of the second form is the following if statement:

```
if (x > y)
    max = x;
else
    max = y;
```

This example takes the larger of the values in variables x and y and stores it in the variable max.

We'll see many if statements in this course because it's one of the most commonly used *control structures* in Java. A Java control structure is any statement that changes the normal sequential order of execution in a program. The three control structures we'll use in this course are *branches* (like if statements), *loops* (which repeat actions, and which we'll discuss in the next lesson), and method calls, which we've already seen.

Now that you know what if statements look like, let's take a closer look at the logical expressions that control which way they branch.

How to Write Boolean Expressions

We saw in the last lesson that Java can use complex arithmetic expressions to calculate numeric results. Java also has Boolean expressions that give us boolean results, and these expressions can be complex too. But let's start with simple comparisons.

Java has several comparison operators we can use to compare values and which give us boolean results, either true or false.

Operator	Meaning
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to
>=	Greater than or equal to
>	Greater than

Java's Comparison Operators

Here are some examples of comparisons and their meanings. In this table, assume that all variables are numeric.

Expression	Meaning
a < b	Is a less than b?
year == 2014	Is the year 2014?
rate >= 4.5	Is the rate greater than or equal to 4.5?

Comparison Expression Examples

The result of each of these comparisons is either true or false. We can use these comparison operators with any of the primitive types in Java, whether those are integer types like the ones in the second row or floating-point like the ones in the third line.

Any of Java's primitive types will work in comparisons as long as both sides of the comparison are compatible. For example, it wouldn't make sense to compare a char type to a double type. You might not get the results you expect!

Okay, now we know how to compare values and get boolean results. Java also gives us operators that allow us to combine boolean values in more complex expressions: *and* (&&), *or* (||), and *not* (!) operators. Here's how they work:

- a && b is true if both a and b are true and false if either is false.
- a || b is true if either a or b is true and false if both are false.
- !a is true if a is false and false if a is true.

Whew, that's confusing! It's easier to see how these work by looking at a *truth table* like the one below. The first two columns give all the possible combinations of values for a and b, and the last three columns give the values of the expressions based on those values.

a	**b**	**a && b**	**a || b**	**!a**
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Truth Table for Boolean Operators

Here are some examples of Boolean expressions using the operators. The first example tests an integer variable named month to see if it is a valid value and returns true if it's between 1 and 12:

(month >= 1) && (month <= 12)

The second example checks integer variables hour and minute to make sure the time is valid for a 12-hour clock (between 1:00 and 12:59):

(hour >= 1) && (hour <= 12) && (minute >= 0) && (minute <= 59)

The last example is a bit more complex:

((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0)

This one tests an integer variable named year to see if its value is a leap year. Leap years occur every fourth year (divisible by four), except for century years (divisible by 100) unless they're also divisible by 400. So 2000 and 2004 were leap years, but 1900 and 1951 weren't.

To do this more easily, we'll use another of Java's arithmetic operators, called *modulus* or just *mod*. The operator is the percent sign (%). It divides numbers and then returns the remainder of the division (what's left over) instead of the quotient. So the operation (5 % 3) gives the result 2 because 5 divided by 3 gives an integer quotient of 1 with a remainder of 2, and (12 % 4) gives 0 because 12 divided by 4 is 3, with nothing left over as a remainder.

This example also uses *nested* parentheses, or sets of parentheses inside other parentheses, to control the order of the tests. The test within the first set of nested parentheses, (year % 4 == 0), evaluates first. Then the test inside the second set, (year % 100 != 0), evaluates. Then the outer set of parentheses ensures that the && operator evaluates next. After that, the test in the last parentheses, (year % 400 == 0), evaluates. The || operator, since it's outside all the parentheses, evaluates last.

So if the year is divisible by 4 but not by 100, we have a leap year, *or* if the year is divisible by 400, we have a leap year. If year contains 1900 or 1951, we'll get false from this expression, but if year is 2000 or 2004, we'll get true.

Text equivalent start.

The prompt says: int year equals. Entering a year in the space next to the prompt and pressing the Run Test button will reveal whether that year is a leap year, highlighting each part of the boolean expression leading to the result as true or false.

Text equivalent stop.

Now that you know how to build logical expressions, let's use them in our temperature conversion project.