Programming via Java Conditional execution

Recall our `MovingBall` program of Figure 6.3, in which a red ball moved steadily down and to the right, eventually moving off the screen. Suppose that instead we want the ball to bounce when it reaches the window's edge. To do this, we need some way to test whether it has reached the edge. The most natural way of accomplishing this in Java is to use the **`if`** statement that we study in this chapter.

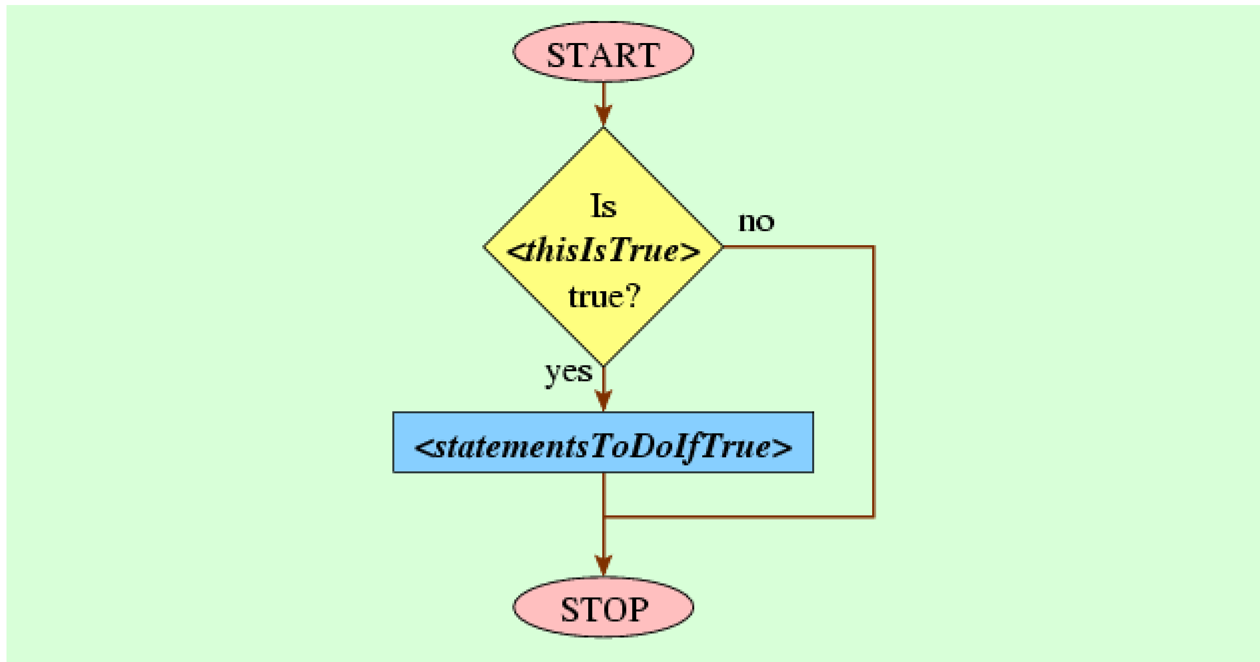
7.1. The `if` statement

An **`if` statement** tells the computer to execute a sequence of statements only *if* a particular condition holds. It is sometimes called a conditional statement, since it allows us to indicate that the computer should execute some statements only in some conditions.

```
if(<thisIsTrue>) {
    <statementsToDoIfTrue>
}
```

When the computer reaches the **`if`** statement, it evaluates the condition in parentheses. If the condition turns out to be **`true`**, then the computer executes the **`if`**statement's body and then continues to any statements following. If the condition turns out to be **`false`**, it skips over the body and goes directly to whatever follows the closing brace. This corresponds to the flowchart in Figure 7.1.

Figure 7.1: A flowchart for the **`if`** statement.



An **if** statement looks a lot like a **while** loop: The only visual difference is that it uses the **if** keyword in place of **while**. And it acts like a **while** loop also, except that after completing the **if** statement's body, the computer does not consider whether to execute the body again.

The following fragment includes a simple example of an **if** statement in action.

```
double num = readDouble();
if(num < 0.0) {
    num = -num;
}
```

In this fragment, we have a variable `num` referring to a number typed by the user. If `num` is less than 0, then we change `num` to refer to the value of `-num` instead, so that`num` will now be the absolute value of what the user typed. But if `num` is not negative, the computer will skip the `num = -num` statement; `num` will still be the absolute value of what the user typed.

7.2. The bouncing ball

We can now turn to our `MovingBall` program, modifying it so that the ball will bounce off the edges of the window. To accomplish this, before each movement, we will test whether the ball has crossed an edge of the window: The movement in the *x* direction should invert if the ball has crossed the east or west side, and the movement in the *y* direction should invert if the ball has crossed the north or south side. The program of Figure 7.2 accomplishes this.

Figure 7.2: The `BouncingBall` program.

```
import acm.program.*;
import acm.graphics.*;
import java.awt.*;

public class BouncingBall extends GraphicsProgram {
    public void run() {
        GOval ball = new GOval(25, 25, 50, 50);
        ball.setFilled(true);
        ball.setFillColor(new Color(255, 0, 0));
        add(ball);

        double dx = 3;
        double dy = -4;
        while(true) {
            pause(40);
            if(ball.getX() + ball.getWidth() >= getWidth()
                    || ball.getX() <= 0.0) {
                dx = -dx;
            }
            if(ball.getY() + ball.getHeight() >= getHeight()
                    || ball.getY() <= 0.0) {
                dy = -dy;
            }
            ball.move(dx, dy);
        }
    }
}
```

If you think about it carefully, the simulation of this program isn't perfect: There will be frames where the ball has crossed over the edge, so that only part of the circle is visible. This isn't a big problem, for a combination of reasons: First, the ball will be over the border only for a single frame (i.e., for only a fleeting few milliseconds), and second, the effect is barely visible because it affects only a few pixels of the ball's total size. Anyway, our eye expects to see a rubber ball flatten against the surface briefly before bouncing back in the opposite direction. If we were to repair this problem, then we would perceive the ball to be behaving more like a billiard ball. In any case, we won't attempt to repair the problem here.

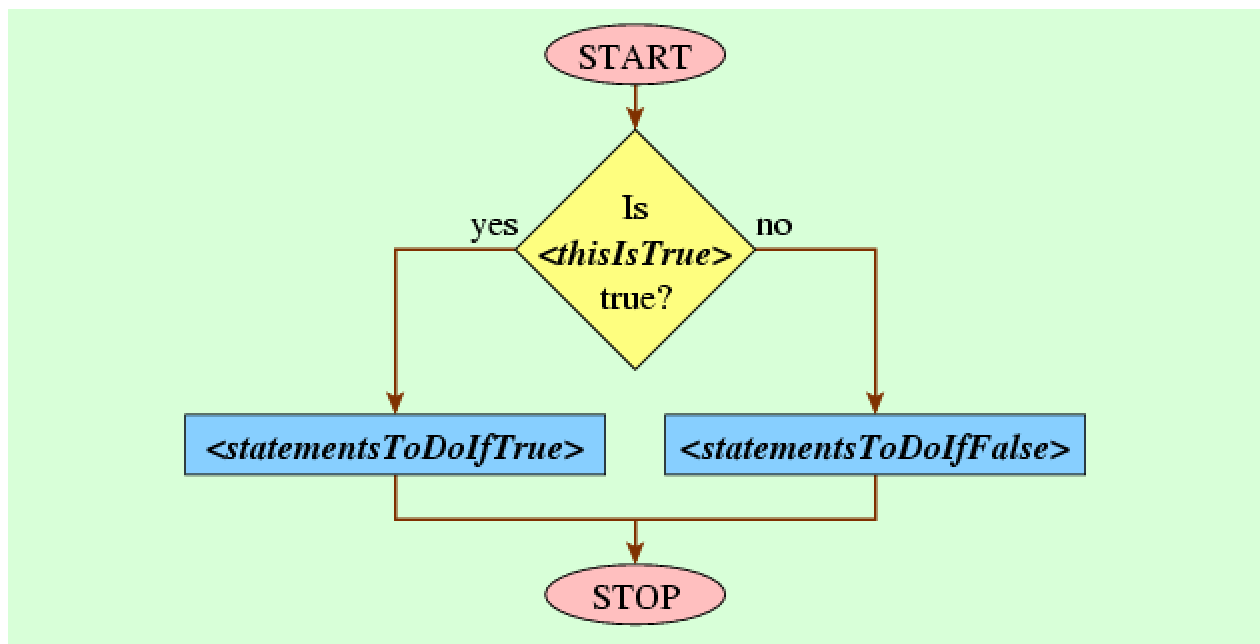
7.3. The `else` clause

Sometimes we want a program to do one thing if the condition is true and another thing if the condition is false. In this case the **`else`** keyword comes in handy.

```
if(<thisIsTrue>) {
    <statementsToDoIfTrue>
} else {
    <statementsToDoIfFalse>
}
```

Figure 7.3 contains a flowchart diagramming this type of statement.

Figure 7.3: A flowchart for the **`else`** clause.



For example, if we wanted to compute the larger of two values typed by the user, then we might use the following code fragment.

```
double first = readDouble();
double second = readDouble();
double max;
if(first > second) {
    max = first;
} else {
    max = second;
}
print(max);
```

This fragment will first read the two numbers `first` and `second` from the user. It then creates a variable `max` that will end up holding the larger of the two. This function says assign the value of `first` to `max` if `first` holds a larger value than `second`; otherwise — it says — assign `max` the value of `second`.

Sometimes it's useful to string several possibilities together. This is possible by inserting **`else if`** clauses into the code.

```
double score = readDouble();
double gpa;
if(score >= 90.0) {
    gpa = 4.0;
} else if(score >= 80.0) {
    gpa = 3.0;
} else if(score >= 70.0) {
    gpa = 2.0;
} else if(score >= 60.0) {
    gpa = 1.0;
} else {
    gpa = 0.0;
}
```

You can string together as many **`else if`** clauses as you want. Having an **`else`** clause at the end is not required.

Note that, if the user types 95 as the computer executes the above fragment, the computer would set `gpa` to refer to 4.0. It would *not* also set `gpa` to refer to 3.0, even though it's true that `score >= 80.0`. This is because the computer checks an **`else if`** clause only if the preceding conditions all turn out to be **`false`**.

As an example using graphics, suppose we want our hot-air balloon animation to include a moment when the hot-air balloon is firing its burner, so it briefly goes up before it resumes its descent. To do this, we'll modify the loop in Figure 6.4 to track which frame is currently being drawn, and we'll use an **`if`** statement with **`else`**clauses to select in which direction the balloon should move for the current frame.

```
int frame = 0; // tracks which frame we are on
while(balloon.getY() + 70 < getHeight()) {
    pause(40);
    if(frame < 100) {         // the balloon is in free-fall
        balloon.move(1, 2);
```

```
    } else if(frame < 200) { // it fires its burner
        balloon.move(1, -1);
    } else {                 // it no longer fires its burner
        balloon.move(1, 1);
    }
    frame++;
}
```

7.4. Braces

When the body of a **`while`** or **`if`** statement holds only a single statement, the braces are optional. Thus, we could write our `max` code fragment from earlier as follows, since both the **`if`** and the **`else`** clauses contain a single statement. (We could also include braces on just one of the two bodies.)

```
double max;
if(first > second)
    max = first;
else
    max = second;
```

I recommend that you include the braces anyway. This saves you trouble later, should you decide to add additional statements into the body of the statement. And it makes it easier to keep track of braces, since each indentation level requires a closing right brace.

In fact, Java technically doesn't have an **`else if`** clause. In our earlier `gpa` examples, Java would actually interpret the **`if`** statements as follows.

```
if(score >= 90.0) {
    gpa = 4.0;
} else
    if(score >= 80.0) {
        gpa = 3.0;
    } else
        if(score >= 70.0) {
            gpa = 2.0;
        } else
            if(score >= 60.0) {
                gpa = 1.0;
            } else {
```

```
            gpa = 0.0;
        }
```

Each **else** clause includes exactly one statement — which happens to be an **if** statement with an accompanying **else** clause in each case. Thus, when we were talking about **else if** clauses, we were really just talking about a more convenient way of inserting white space for the special case where an **else** clause contains a single **if**statement.

7.5. Variables and compile errors

7.5.1. Variable scope

Java allows you to declare variables within the body of a **while** or **if** statement, but it's important to remember the following: A variable is available only from its declaration down to the end of the braces in which it is declared. This region of the program text where the variable is valid is called its scope.

Check out this illegal code fragment.

```
29 if(first > second) {
30     double max = first;
31 } else {
32     double max = second;
33 }
34 print(max); // Illegal!
```

On trying to compile this, the compiler will point to line 34 and display a message saying something like, Cannot find symbol `max`. What's going on here is that the declaration of `max` inside the **if**'s body persists only until the brace closing that **if** statement's body; and the declaration of `max` in the **else**'s body persists only until the brace closing that body. Thus, at the end, outside either of these bodies, `max` does not exist as a variable.

A novice programmer might try to patch over this error by declaring `max` before the **if** statement.

```
double max = 0.0;
if(first > second) {
    double max = first; // Illegal!
} else {
    double max = second; // Illegal!
```

```
}
print(max);
```

The Java compiler should respond by flagging the second and third declarations of `max` with a message like `max` is already defined. In Java, each time you label a variable with a type, it counts as a declaration; and you can only declare each variable name once. Thus, the compiler will understand the above as an attempt to declare two different variables with the same name. It will insist that each variable once and only once.

7.5.2. Variable initialization

Another common mistake of beginners is to use an **`else if`** clause where an **`else`** clause is completely appropriate.

```
46 double max;
47 if(first > second) {
48     max = first;
49 } else if(second >= first) {
50     max = second;
51 }
52 print(max);
```

Surprisingly, the compiler will complain about line 52, with a message like variable `max` might not have been initialized. (Recall that initialization refers to the first assignment of a value to a variable.) The compiler is noticing that the `print` invocation uses the value of `max`, since it will send the variable's value as a parameter. But, it reasons, perhaps `max` may not yet have been assigned a value when it reaches that point.

In this particular fragment, the computer is reasoning that while the **`if`** and **`else if`** bodies both initialize `max`, it may be possible for both conditions to be **`false`**; and in that case, `max` will not be initialized. This may initially strike you as perplexing: Obviously, it can't be that both the **`if`** and the **`else if`** condition turn out **`false`**. But our reasoning hinges on a mathematical fact that the compiler is not itself sophisticated enough to recognize.

You might hope for the compiler to be able to recognize such situations. But it is impossible for the compiler to reliably identify such situations. This results from what mathematicians and computer scientists call the halting problem. The halting problem asks for a program that reads another program and reliably predicts whether that other program will eventually reach its end. Such a program is provably impossible to write.

This result implies that there's no way to solve the initialization problem perfectly, because if we did have such a program, we could modify it to solve the halting problem. Our modification for solving the halting problem is fairly simple: Our modified program would first read the program for which we want to answer the halting question. Then we place a variable declaration at the program's beginning, and then at any point where the program would end we place a variable assignment. And finally we hand the resulting program into the supposed program for determining variable initialization. Out would pop the answer to the halting question for the program under inquiry.

That said, there's nothing preventing the compiler writers from trying at least to identify the easy cases. And you might hope that they'd be thorough enough to identify that given two numbers, the first is either less than, equal to, or greater than the second. But this would inevitably lead to a complex set of rules when the compiler identifies initialization and when not, and the Java designers felt it best to keep the rules simple enough for regular programmers to remember.

The solution in this case is to recognize that, in fact, we don't need the **`else if`** condition: Whenever the **`if`** condition turns out to be **`false`**, we the **`else if`** body to occur. Thus, doing that extra test both adds extra verbiage and slows down the program slightly. If use an **`else`** clause instead of an **`else if`**, the compiler will reason successfully that `max` will be initialized, and it won't complain.

(Novice programmers are sometimes tempted to simply initialize `max` on line 46 to avoid the message. This removes the compiler message, and the program will work. But patching over compiler errors like this is a bad habit. You're better off addressing the problem at its root.)

By the way, Java compilers are some of strictest I've seen about uninitialized variables. It can occassionally be a bit irritating to have to repair a program with an alleged uninitialized variable, when we are certain that the program would run flawlessly as written. The reason the compiler is so aggressive is that it's aggressively trying to help you debug your programs. Suppose, for the sake of argument, that it was a real problem in the program. If the compiler didn't complain, you'd be relying on the test cases to catch the problem. If the test cases didn't catch it, you'd be misled into releasing a erroneous program. Even if they did catch it, you're stuck trying to figure out the cause, which can take quite a while. A problem like an uninitialized variable can be very difficult to track down by trial and error; but with the compiler pointing to it for you, it becomes much easier.

Source: http://www.toves.org/books/java/ch07-if/index.html