Booleans and Conditional Statements

Introduction

In this article from my free Java 8 Course, I will be discussing booleans and conditional statements in Java.

Booleans

A boolean is a primitive data type that stores one of two values, `true` or `false`. The name comes from the inventor, George Boole who discussed the idea of a `boolean` in great detail. In Java, we can use booleans to create conditions and execute blocks of code based on those conditions. To illustrate this idea, imagine a situation that most of us experience every day. When your alarm goes off in the morning, whether or not you went to sleep early could cause you to decide between getting up or pressing the snooze button. Whether you went to sleep early could be considered a `boolean` value, but for the code to decide which of these two actions you should respond with, you need conditional statements.

Conditional Statements

Conditional statements define conditions that are `true` or `false` and then execute based on whether or not the condition is `true`. Basically, conditions say, “If x is `true`, then execute y”. This logic is called an “if-statement”. Throughout all programming languages this if-statement is the most powerful and important statement, because it allows a program to execute differently every time. For the sake of demonstration, if we created a `boolean isMonday` and a `boolean isRaining`, and set them both to `true`, we could then have an if-statement that checks this and then calls `drinkBeer(),` if both of them are `true`. After all, what else would you do on a rainy Monday? ;-)

```
@Test
public void demonstrateBoolean() {
     boolean isMonday = true;
     boolean isRaining = true;

     if(isMonday && isRaining) {
          drinkBeer();
     }
}
```

Example 1

Checking if both conditions are `true` is done using the “&&” symbol. If both conditions are true, then the `drinkBeer()` method will execute. We could also check if only one of the conditions are `true`:

```
@Test
public void demonstrateBoolean() {
     boolean isMonday = false;
     boolean isRaining = true;

     if(isMonday || isRaining) {
          drinkBeer();
     }
}
```

Example 2

The if-statement in Example 2 says, “If it’s Monday or it’s raining, then drink beer”. The `||`, called a pipe operator, defines an OR operator. Now, if it is raining or it is Monday, the `drinkBeer()` method will be executed.

Short Circuiting

One interesting aspect of compound if-statements is the idea of short circuiting. As we discussed previously, in an AND operator, if both conditions are `true,` the `drinkBeer()` method will execute. However, if the first condition is false, the if-statement will “short circuit” and will not execute the code without checking the second boolean. If the `boolean isMonday` was `true` and the `boolean isRaining` was `false`, you would excitedly note that it’s Monday, but since it wasn’t raining you still couldn’t drink beer.

The same is true for a OR operator. If the first condition is `true`, then checking the second condition is unnecessary, since the code inside the conditional will execute whether or not the second condition is `true`.

Complex If-Statements

Our “if-statements” can also be made much more complex by compounding various conditions. The logic works by evaluating conditions in multiple levels of parentheses and then evaluating conditions in only one set of parentheses. The logic also checks conditions from left to right. Before you read on, see if you can figure out if the `drinkBeer()` method will execute in Example 3.

```
@Test
public void demonstrateBoolean() {
     boolean isMonday = false;
     boolean isRaining = true;
     boolean isTuesday = true;
     boolean isSunny = false;

     if((isMonday && isRaining) || (isTuesday && isSunny)) {
          drinkBeer();
     }
}
```

Example 3

Ok, let's look at the first condition, "`isMonday && isRaining`" - that's `false`. After that you can see that we have a OR operator in between the two sets of conditions, so the if-statement must check the second condition. So let's do that: "`isTuesday && isSunny`". This is also `false`, because it is Tuesday, but it isn't sunny. Since neither condition is `true`, the entire statement is `false` and we can't drink a beer ;-)

Until you fully understand "boolean algebra" and have mastered using conditionals, continue using parentheses to enforce the order of execution you need safely. In short, a conditional is interpreted as follows:

1. Any conditionals inside parentheses
2. Any AND symbols
3. Any OR symbols

Unless you feel very comfortable with conditionals, you should surround all of your conditions in parentheses just to be safe.

The Else Statement

Now I'll introduce you to the counterpart of the "if-statement": the "else statement". Let's say it's not Monday, so we can't drink beer, but we still need to stay hydrated. We could say, "If it's Monday, drink beer; otherwise, drink milk."

```
@Test
public void demonstrateBoolean() {
     boolean isMonday= false;

     if(isMonday) {
          drinkBeer();
     } else {
          drinkMilk();
     }
}
```

Example 4

You might notice that the “else statement” doesn’t have a condition. This is because the “else” executes in all cases where the “if” case doesn’t apply.

The Else-If-Statement

If I were you, I’d get bored with drinking milk six days a week. But at the same time, I don’t want to drink beer more than once a week. This is where the final conditional statement comes into play: the “else-if” statement. The `else if` evaluates a condition if the if-statement is `false`. You can also have multiple “else ifs” that execute if all previous statements are false. At the end of all these statements, you can have your “else” statement that still executes in all other cases, meaning that all of the other statements were `false`. Let’s take a look at an example where on Fridays we drink water:

```
@Test
public void demonstrateBoolean() {
     boolean isMonday= false;
     boolean isFriday= true;

     if(isMonday) {
          drinkBeer();
     }
     else if(isFriday) {
          drinkWater();
     }
     else {
          drinkMilk();
     }
}
```

Example 5

Using Conditionals with Other Primitive Data Types

Not only can we use conditional statements to check if a boolean variable is `true` or `false`, but we can also create a boolean using a condition, and evaluate that. For example, we could have two ints, i and j with the values 4 and 3 respectively. We can compare them using the following symbols:

Symbol	Meaning
>	Greater Than
<	Less Than
>=	Greater Than or Equal
<=	Less Than or Equal
==	Equal
!=	Not Equal

Example 6

You may notice that the operator for equals is a double ‘=’ operator, rather than a single ‘=’. This is because ‘=’ already has a use: it is used for the assignment of values to primitive data types and for assigning Objects to reference variables. Therefore, to avoid confusion for both the programmer and the compiler, == is used to **compare equality.**

‘!=’ means ‘not equal’. Generally, ‘!’ in front of any boolean value will negate its value. So it follows that we'll read, ‘!true’ as ‘not true.’, which is equivalent to false. We read ‘!false’ as ‘not false’, therefore it will be equivalent to true.

If you take a look at the example below, you can see different ways that conditionals can be used to compare values. Obviously, since we know the values assigned to `i` and `j`, this isn’t very helpful, but if these values were dynamically given as a method parameter, then these conditionals would be useful.

```
@Test
public void demonstrateBoolean() {
     int i = 4;
     int j = 3;
     boolean areEqual = (i == j);
     if(i > j) {
          // i is greater than j
     }else if( !(i  >=  j)) {
          // i is not greater than or equal to j
     }else {
          // i is equal to j
     }
```

```
	if(areEqual) {
		// i is equal to j
	}else {
		// i is not equal to j
	}
}
```
Example 7

Applying Conditionals

You may not have the skills to create more complex conditional statements yet, but you can still apply conditionals to some useful examples. Let's say we have our values of `i` and `j`, but now we want to increase the value of `j` if it is Monday. We won't be incrementing in every case; we only do this if our condition is met. We can do other things too, all of which might be useful under certain conditions.

```
@Test
public void demonstrateBoolean() {
	int i = 4;
	int j = 3;
	boolean isMonday = true;
	boolean areEqual = (i == j);
	if(areEqual){
		i = 8
	}
	else if(j > i){
		j = i - 3;
	}

	if(isMonday){
		j++;
	}
}
```
Example 8

Conditionals provide Java code with the means to respond differently depending on different outside conditions.
They are extremely flexible and powerful tools that you will continue to use as you learn more and more Java.

Thanks for reading!