CS125 : Introduction to Computer Science

Lecture Notes #5
Boolean Expressions, Simple Conditionals, and Statements

Lecture 5 : Boolean Expressions, Simple Conditionals, and Statements

Boolean Expressions

We have previously discussed the idea of an *expression*, and the idea that every expression evaluates to a single value of a particular type. Up to this point, most of the expressions we have seen have evaluated to type `int` or type `double`. There are many possible expressions you might write, however, and not all of them evaluate to values of type `int` or `double`. For example, the following is an expression – specifically, it's a literal – of type `boolean`:

```
false
```

And, the following is another expression of type `boolean` (which again happens to be a literal of the `boolean` type):

```
true
```

Finally, if we perform the following declaration and initialization:

```
boolean exampleVariable;
exampleVariable = false;
```

then after the above code is run, the following is also an expression of type `boolean`:

```
exampleVariable
```

We call those expressions *boolean expressions* because they are expressions that evaluate to a value of type `boolean`, rather than a value of some other type.

Boolean expressions of greater complexity

Up to this point, the only boolean expressions we have been able to put into our programs are literals of type `boolean`, and variables of type `boolean`, as in the examples we just saw. We had the arithmetic operators to help us create more complex arithmetic expressions, but none of those operators helped to produce boolean values – all of those operators helped perform arithmetic and, as a result, the expressions we wrote that used those operators, evaluated to numerical values.

To form boolean expressions that are more complicated than literals or single variable names, we will need operators that help produce `boolean` values, rather than the arithmetic operators, which help produce numerical values. These operators that produce `boolean` values fall into two groups:

- relational operators – two operands can be any type, but the resultant expression evaluates to a `boolean` value
- logical operators – the operands need to be of type `boolean`, and the resultant expression evaluates to a `boolean` value

We will examine both categories of operators.

Relational Operators

The *relational operators* are operators that perform comparisons; we are trying to see how two values relate to each other. Specifically, we question whether or not two values are related to each other in a particular way. If they are indeed related to each other in that way, the expression evaluates to `true`, and if the two values are *not* related to each other in that way, then the expression evaluates to `false`.

The relational operators are:

- `<` (less than)
- `>` (greater than)
- `<=` (less than or equal to)
- `>=` (greater than or equal to)
- `==` (are equal)
- `!=` (are not equal)

These, like the arithmetic operators, are *binary* operators, meaning they have two operands.

Do not confuse `=` and `==` when you write your code!!! This is a very common mistake. The single equals sign means assignment; it is an action. The double equals sign means "compare for equality"; it asks a question, "are the two operands equal, or not?".

Using these operators, we can create boolean expressions that perform comparisons for us. For example:

- `heightOfPerson > 6.3` (evaluates to `true` when the variable `heightOfPerson` of type `double` holds a value greater than `6.3`, and evaluates to `false` otherwise)
- `examScore <= 91` (evaluates to `true` when the variable `examScore` of type `int` holds a value less than or equal to `91`, and evaluates to `false` otherwise)
- `grade != 'A'` (evaluates to `true` when the variable `grade` of type `char` holds a character other than the capital letter 'A', and evaluates to `false` otherwise)
- `statusFlag == true` (evaluates to `true` when the variable `statusFlag` of type `boolean` holds the value `true`, and evaluates to `false` otherwise)
- `time1 < time2` (assuming `time1` and `time2` are both of type `int`, evaluates to `true` when the value stored in `time1` is less than the value stored in `time2`, and evaluates to `false` otherwise

In general, the two operands of a relational operator tend to be of the same type. However, since both integers and floating-point values are numbers, it *is* possible to compare them to each other using the relational operators (for example, `5 <= 6.1` or `23.00002 == 23`).

Logical Operators

- As we stated earlier, relational operators have assorted types as operands, and produce values of type `boolean`.
- The logical operators also produce values of type `boolean`; however, unlike relational operators, logical operators also must have operands of type `boolean`.
- Logical operators are designed to create complex boolean expressions out of simple boolean expressions.
- The following four operators are the logical operators:

```
||    (or)
&&    (and)
!     (not)
^     (exclusive or, also called xor)
```

- The concepts these operators implement are very common all over computer science.

```
X or Y -- true if either X is true,
                    or Y is true, or both
X and Y -- true only if both X and Y are true
not X   -- returns the opposite of X
X xor Y -- true only when X is true and Y is
   false, or when Y is true and X is false;
   false if X and Y are the same
```

Using these operators, we can create more complex boolean expressions out of simpler boolean expressions such as boolean literals or boolean variables. For example:

- `true && false` (evaluates to `false`, since it is NOT the case that both operands are true
- `true || false` (evaluates to `true`, since AT LEAST ONE operand is true)
- `true ^ false` (evaluates to `true`, since EXACTLY ONE operand is true)
- `!false` (evaluates to `true`, which is the opposite of the operand)

If we had three variables, `val1`, `val2`, and `val3`, each of type `boolean`, and if the first two variables held the value `true` and the `third` held the value `false`, then:

- `val1 && val2` evaluates to `true`
- `val2 && val3` evaluates to `false`
- `val1 || val3` evaluates to `true`
- `val1 ^ val2` evaluates to `false`
- `val1 ^ val3` evaluates to `true`
- `!val3` evaluates to `true`

Using relational and logical operators together

It's important to keep in mind the difference between the relational and logical operators:

- the relational operators produce `boolean` values, but the operands themselves do NOT have to be `boolean` values. For example, `5 < 6.1` is a perfectly legal boolean expression; the operands are not `boolean` values even though the result is
- the logical operators not only produce `boolean` values, but must have `boolean` values as operands as well

As a result, often we have many small `boolean` expressions, each using a relational operator to generate a `boolean` value, and then all the small `boolean` expressions are merged into one large `boolean` expression by using the logical operators. For example, the expression:

```
(x >= 1) && (x <= 100) && (x % 2 == 0)
```

evaluates to `true` whenever `x` is an even integer between `1` and `100`, inclusive. As another example, the following expression:

```
(x == 5) || ((x > 10) && (x < 0))
```

evaluates to `true` only when `x` is `5`, since the second operand of the logical **OR** above can never evaluate to `true`.

The short-circuiting behavior of logical **AND** and logical **OR**

The `&&` and `||` operators are *short-circuiting*; that is, they don't evaluate the second operand of the operator if the answer to the overall expression is already known after evaluating the first operand.

`Example:`

```
(x != 0) && (y/x < 70.2)
```

In the boolean expression above, the first thing to be evaluated is the truth or falsehood of the left side of the `&&` operator. Assume `x` is zero; if so, this expression we are trying to evaluate – namely, `x != 0` – evaluates to `false`. And in that case, we know what the result of the entire expression will be! Since an **AND** expression is only true if both operands are true, and since we know the value of the first operand is `false`, we know the entire **AND** expression *must* evaluate to `false` *regardless* of the value of the second operand. Again, that is assuming that `x` is zero.

On the other hand, if `x` held a non-zero value, then the first expression – the `x != 0` expression – would evaluate to `true`. In that case, the **AND** expression could still evaluate to `true` or to `false`, depending on the result of the evaluation of the second operand. So, because of that, we must evaluate the second operand (`y/x > 70.2`) to know for sure what the result of the **AND** expression is.

This feature might be helpful for two reasons. First of all, any work that we don't have to do, is time we save – so if the machine can avoid having to evaluate the second operand, then that means your program runs a little bit faster since there is less work to do. Also, in the expression above, we don't *want* to evaluate `y/x` if `x` is zero, since the subsequent division-by-zero operation would crash our program. But since the short-circuit property of the `&&` operator prevents that second expression from being evaluated when `x` is zero, our program is safe! So in addition to saving time, the short-circuiting feature can be used as we used it in the example above, to keep us from evaluting a particular expression such as `y/x` if it would be dangerous to do so.

Likewise, for the **OR** operator, if the first operand evaluates to `true`, the second operand won't be evaluated, since the first operand being `true` means the entire expression must be `true` regardless of the value of the second operand.

Note that **XOR** has no short-circuiting ability since you always need to know the values of both operands in order to evaluate the **XOR**, and **NOT** has no short-circuiting ability since there is only one operand to begin with.

Basing results on comparisons – the *conditional*

Up to this point, we've only listed instructions one after the other. But, just listing instructions is not enough!

What we do might depend on certain conditions being true.

ex.: Read a student score, print out whether each has passed (better than 60) or failed (60 or worse).

```
Read score, call it examScore
If examScore is greater than 60,
   print out that the student has passed
Otherwise examScore must be <= 60.
   In this case, print out that
                 the student has failed.
```

In this case, we don't always print "passed" and don't always print "failed". In fact, each time we read an exam score, we perform exactly one of those – we will always print either "passed" or "failed", but never both. How can we make this work?

What we need is a new kind of language statement – the *conditional*, which will run code "conditionally".

The `if` statement

Today we are introducing a new kind of statement – the `if` statement. The form for this statement is as follows:

```
if (condition)
   statement;
```

- `condition` must be a boolean expression
- `statement` is a statement of some kind
- The `statement` is only executed if the `condition` evaluates to `true`. If the `condition` evaluates to `false` instead, then, the statement is skipped over, and the next line of code that executes is whatever is after the `if` statement.

Example:

```
if (grade > 60)
   System.out.println("Passed!");
System.out.println("Done!");
```

If student's grade is greater than 60, then both lines get printed. If student's grade is not greater than 60, than only `"Done!"` gets printed. The line that prints `"Done!` has nothing to do with the `if` statement, so it gets run regardless of whether the condition is `true` or `false`. (Note that we indent the statement that is run conditionally by the `if` statement – that is the proper style.)

Statements

Up to this point, we have used the term "statement" to refer to any one of the following:

- a variable declaration statement
- an assignment statement
- a `System.out.println` statement
- a `System.out.print` statement

Furthermore, if you want to get technical, the `Keyboard.readInt()`, `Keyboard.readChar()`, and `Keyboard.readDouble()` expressions are also statements. Those statements evaluate to values, so they are also expressions, as we discussed before. But we *can* put them in the "statement" category, too, since they accomplish work on their own, whether we do something with the returned value or not.

So, it's all fine and good to make a list of things we call "statements", but what *is* a statement? Well, one vague way to define the term is to say that a statement is one complete unit of work. For example, the expressions `5 - 3 * 2` or `"Sum is : " + 120` evaluate to single values, but what do we *do* with those values? It is not until we put the values inside statements – perhaps by writing the value to a variable (via an assignment statement) or perhaps by printing the value to the screen (via a `System.out.println` statement) – that we've made meaningful use of the expression's value. The expression, on its own, simply calculates the value and then ignores it. The fact that an expression is not automatically a statement, is why the following will compile:

```
public class Example1 {
   public static void main(String[] args) {
      System.out.println(2 + 3 + " is the sum.");
   }
}
```

but the following two examples will not compile:

```
public class Example2 {
   public static void main(String[] args) {
      2 + 3 + " is the sum."
   }
}
```

```
public class Example2 {
   public static void main(String[] args) {
      2 + 3 + " is the sum.";  // <-- adding a ; doesn't make a difference
   }
}
```

You can't just toss `2 + 3 + " is the sum."` into your program; you need to *do* something with the result of that expression. (If you run `Keyboard.readInt()`, you obtain a piece of data from the user, so in that case, you have indeed done something, even if you don't use the value you've inputted – you've taken in an input value from the user so that that input value won't be re-entered the next time your program wants input.)

However, we can come up with a more precise definition of "statement" than "one complete unit of work". Instead, our definition of "statement" will be: a statement is anything we refer to as a statement, i.e. anything that is on our list of things we call statements. We started our list at the top of the previous page; all those statements accomplished different sorts of things, but they were all on the list, so they were all statements.

This widely-ranging definition of "statement" is a good thing! It means that in any code construct where we say a "statement" should go, we can substitute anything we want, as long as it is called a "statement". For example, we have said this is a statement:

```
if (grade > 60)
   System.out.println("Passed!");
```

So, if it is a statement, then why not put it inside an `if` statement, since any statement can go inside an `if` statement? That is, the grammar of an `if`-statement was:

```
if (condition)
   statement;
```

so if the statement is the conditional we listed above, and the condition is `daysInAttendance > 200`, then we get the following:

```
if (daysInAttendance > 200)
   if (grade > 60)
      System.out.println("Passed!");
```

That is legal code! Since the grammar of a conditional had a statement within it, we can put any statement we want there – even another conditional! In a sense, it is as if the simple statements, such as assignments, declarations, and print statements, are primitives – and then statements such as the `if` statement, that can contain other statements within itself, are the means by which we create compositions. This is not so different from expressions, and the way literals and variables were primitive expressions, while more complex expressions were built from simpler expressions. And just as we made use of those expression rules to build expressions that were as complex as we needed, likewise we can make use of statements-that-hold-other-statements, to create statements of whatever complexity we need.

We will explore this idea further in the next lecture.