CS125 : Introduction to Computer Science

Lecture Notes #6
Compound Statements, Scope, and Advanced Conditionals

Lecture 6 : Compound Statements, Scope, and Advanced Conditionals

Multiple statements based on the same condition

Given what we have seen so far, if we wanted three different statements to run based on the truth or falsehood of the same condition, we would need three different conditionals:

```
if (x > 5)
   a = a + 2;

if (x > 5)
   b = b + 3;

if (x > 5)
   c = c + 1;
```

In the above code, if `x` is indeed greater than `5`, then all three assignment statements will be run, and if `x` is NOT greater than `5`, then none of the assignment statements will be run.

However, we are being wasteful in two ways here. First of all, we had to write the same condition three different times. If we had some means of collecting those three assignment statements together, we might be able to do something like this:

```
if (x > 5)
   DO ALL OF THESE:
       a = a + 2;
       b = b + 3;
       c = c + 1;
   OK NOW YOU ARE DONE
```

and thus only write out the condition once. If there were 1000 statements that should all run depending on whether or not `x` was greater than `5`, rather than just three statements that depend on that condition like in the above example, then we'd save even more work by being able to group all those statements into one conditional statement under one copy of the condition, instead of typing the same condition 1000 times (once in each of 1000 different conditional statements).

In addition, if we've *listed* the condition three times in our code, then it is being *evaluated* three times as well. By limiting the number of times we list the same condition, we limit the number of times it has to be evaluated – thus reducing the work our program is requiring the processor to do.

For both of these reasons, we would like a way to group statements together, as we did above with our "DO ALL OF THESE/OK NOW YOU ARE DONE" remarks. We would like a way to make a conditional statement, conditionally run *many* statements, rather than just one.

To accomplish this, we will introduce another kind of statement – the *compound statement*.

Compound Statements

The compound statement is created by taking any collection of other statements, listing them one after the other in the order you want, and then enclosing the entire collection of statements within a set of curly braces. For example:

```
{
   int a;
   a = 2;
   System.out.println(a);
}
```

We can consider the above five lines of code to be one statement, since even though there are three smaller statements (on the second, third, and fourth lines of the example above), they are all included within a pair of curly braces. In places where we need a single statement, the above qualifies – it is considered only one statement. The fact that it happens to be composed of many smaller statements is irrelevant, because the curly braces have the effect of gathering all the smaller statements together into one logical unit.

To run this single statement, you just run each of the smaller statements within it, one after the other, in sequence. So, in most situations, there is no difference at all between writing a program with the above code in it, and writing three separate statements on their own:

```
int a;
a = 2;
System.out.println(a);
```

since in both cases, we will run the declaration, the assignment, and the print statement, one after the other, in sequence.

There are only two situations where converting a series of individual statements into a single, compound statement, is a helpful thing to do. One of these situations involves *scope*, and we'll talk about that later in this packet. The other situation is the one we were just talking about at the start of this packet – by treating a number of statements as one complete unit, we can then include that one complete unit as the one statement inside a conditional, and thus enable a single conditional statement to conditionally execute many smaller statements together, instead of only one smaller statement. (We'll see an example in just a moment.)

Once again, we are seeing here that a good definition of "statement" is, "Anything that we call a statement, is a statement". It's a good definition because there really isn't any similarity between the different things that we call a statement. Each one basically represents a single "chunk" of work, but even that is a vague definition. So rather than trying to define "statement", we just make a list of everything that is called a statement. What we have found already, though, is that having a list of different things that we call a statement is very useful, since it means whenever we need a statement, we can use anything appropriate from that list. Now, we can add the "compound statement" to that list of things that we can call "a statement", and thus we can use a "compound statement" anywhere we might need a statement in our program.

Conditionals with Compound Statements

Just as the compound statement had other statements nested within it, likewise the `if`-statement has another statement nested within it. And, as with the compund statement, we can choose any statement from our "list of statements" to go in that position. For example, if we chose our example compound statement from the previous page, to be the statement nested within an `if`-statement, then we might get the following:

```
   if (daysInAttendance > 200)
   {
      int a;
      a = 2;
      System.out.println(a);
   }
```

In the above example, if the condition is true, all three lines inside the curly braces are run, and if the condition is false, none of the three lines inside the curly braces are run.

Similarly, to solve the problem we introduced at the start of this packet, we simply need to enclose each of the three assignment statements, together inside a compound statement:

```
if (x > 5)
{
   a = a + 2;
   b = b + 3;
   c = c + 1;
}
```

In both cases, if the condition is true, all the smaller statements within the compound statement are run, and if the condition is false, none of the smaller statements within the compound statement are run.

By using this technique, we can make a single condition affect whether or not an enormous amount of code is run. For example, if there were 1000 smaller statements within a compound statement, and we then put that compound statement within a conditional statement, then those 1000 lines together would either all get run, or else none of those 1000 lines would run – in either case, as the result of evaluating a single condition.

Scope

The *scope* of a variable is the part of the program in which that variable can be used, the section of the code for which that particular variable is defined and accessible.

Part of what determines the scope of a variable is what *block* it is declared in. For now, a block can be thought of as the collection of code within a set of curly braces. For example, creating a compound statement creates a block, since now you've got the assorted smaller statements grouped within a set of curly braces.

Variables declared in a block are usable only until the end of that block. Things will get more complex when we start discussing methods, but for now – as we deal with code where everything is inside the `main()` method – we can think of our program as a bunch of blocks nested within other blocks.

The outermost block is the `main()` method itself – all the code inside the open- and close-braces of the `main()` method. In the absence of any additional blocks inside `main()`, a variable declared inside `main()` is usable from the point of its declaration, to the end of the block it was declared in – i.e. the end of the `main()` method.

But now, put another block inside the `main()` method:

```
public static void main(String[] args)
{
   int i = 0;
   {
      int x;
      x = 1;
      System.out.print(i);
      i++;
   }
   // System.out.println(x);
}
```

A variable such as `i` is declared in the outermost block – inside `main()` but not inside any other block. So, it is usable everywhere in `main()` from the point of its declaration, down to the end of `main()` at the final closing brace.

But, the variable `x` is declared inside a block that is *inside* `main()`, or, more formally, inside a block that is *nested* in `main()`. The rule is that a variable declared inside a block goes out of scope at the end of the block. And so, `x` is declared inside the nested block and thus goes out of scope as soon as the closing brace of the nested block is reached. So `x` and any value it was holding effectively vanish once the end of the nested block is reached. As a result, the program would not compile if we uncommented the output line that is currently commented out. At that point in the code, there *isn't* a variable `x`, because that variable went out of scope when the nested block ended and thus no longer exists.

The rule is as follows:

- As always, variables cannot be used before they are declared.
- Variables declared in outer blocks can be used inside any block nested inside that outer block, and any changes made to the variable inside that inner, nested block, are still in effect when the nested block ends and we return to the outer block. (In our example, the variable `i` is usable anywhere inside that compound statement, and after the compound statement is over, the variable `i` holds the value `1`, since the increment that happpened to the variable `i` within the compound statement, permanently affects `i`. The value of `i` does NOT reset to `0` when the compound statement ends.)
- Variables declared inside an inner block are not accesssible to the outer block – i.e. once you leave the inner block and you are back in the outer block inside which the inner block was nested, the variable you declared inside the inner block has gone out of scope and cannot be used by the outer block. (In our example, the variable `x` is destroyed at the close of the compound statement and does not exist past that point, which is why the commented-out print statement would not compile if we uncommented it – it is the same as if we tried printing the value of the expression `x` when the variable `x` had never been declared at all.)

One advantage of this is that, if a variable is out of scope, its memory is now free for some other variable to use. There are other advantages as well, involving properly organizing what variable names we are using – those advantages will become a bit clearer as we learn additional Java syntax and as our programs become more complex.

Scope in conditionals

From the standpoint of scope, we can consider the following two statements to be equivalent:

```
if (grade > 60)                 if (grade > 60) {
   int i = 5;                      int i = 5;
                                }
```

That is, if you only have a single-line statement nested within the `if`-statement, it is still treated as if there were a set of curly braces around it, and thus the scope of the variable `i` above will end at the end of the closing curly brace, whether you actually put the curly braces into your code or not. Or in other words, in the above left example, the variable `i` is NOT in scope once the `if`-statement has concluded; anything declared within the `if`-statement is out of scope once the `if`-statement has concluded.

If you actually have a compound statement within your `if`-statement, then the scope rules work exactly as we said they did for compound statements earlier. For example:

```
// same with an if statement as with a loop

int y = 5;  <--- y declared outside block
if (y < 10)
{
   int x = 9; <--- x declared inside block
   System.out.print(x); <-- x can be used here
   y = y + 7;   <--- y can be used here
}
System.out.println(y);  <--- will print 12;
                       not only can y be used
                       here but whatever
                       alterations made to y
                       inside the block are
                       still in effect here
System.out.println(x);  <--- ERROR! x went
                       out of scope with the
                       close brace of the if
                       statement and can't be
                       used here
```

The `if-else` statement

Imagine we had exactly one of two statements we wanted to execute, depending on whether a condition was `true` or `false`:

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

Only one of those conditions can be true, so only one of `"Passed!"` or `"Failed!"` will be printed (though, as before, `"Done!"` gets printed no matter what). There is a more convenient form for this, which is as follows:

```
if (condition)
   statement1;
else
   statement2;
```

- Again, the `condition` is a boolean expression
- When the `condition` is `true`, `statement1` is executed and `statement2` is skipped over.
- When the `condition` is `false`, `statement1` is skipped over and `statement2` is executed.

Example:

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

If the grade is greater than 60, then the output will be:

```
Passed!
Done!
```

On the other hand, if the grade is not greater than 60, then the output will be:

```
Failed!
Done!
```

This is exactly like the example on the top of the previous slide...only now we needed only one statement to do it, and we didn't need to evaluate conditions twice, just once.

As we stated before, whenver the form of these constructs requires a `statement` somewhere, you can substitute as simple a statement or as complex a statement as you like. This means, for example, that since the `if-else` statement needs a `statement` after the condition, and another after the `else` keyword, you can put very complex statements in those spots...such as another `if-else`.

```
   if (grade >= 90)
      System.out.println("gets an A.");
   else
      if (grade >= 80)    // but < 90
         System.out.println("gets a B.");
      else
         if (grade >= 70)    // but < 80
            System.out.println("gets a C.");
         else
            System.out.println("below C.");
```

It is possible to list these statements *very* compactly if there are many nested `else` cases. The above can instead be written as:

```
// the above:
   if (grade >= 90)
      System.out.println("gets an A.");
   else
      if (grade >= 80)     // but < 90
         System.out.println("gets a B.");
      else
         if (grade >= 70)    // but < 80
            System.out.println("gets a C.");
         else
            System.out.println("below C.");


// compactly:
   if (grade >= 90)
      System.out.println("gets an A.");
   else if (grade >= 80)   // but < 90
      System.out.println("gets a B.");
   else if (grade >= 70)   // but < 80
      System.out.println("gets a C.");
   else      // grade < 70
      System.out.println("below C.");
```

What happens here?

```
if (grade >= 80)
   if (grade >= 90)
      System.out.println("gets an A.");
else
   System.out.println("lower than B.");
```

The `else` is paired with the *nearest* unmatched `if`. So, the computer reads it as:

```
if (grade >= 80)
   if (grade >= 90)
      System.out.println("gets an A.");
   else
      System.out.println("lower than B.");
```

This is known as a **dangling else** and is an error that can be very hard to track down. So, be careful! To correct this, you can either put in an "empty statement" – i.e. a semicolon by itself...

```
if (grade >= 80)
   if (grade >= 90)
      System.out.println("gets an A.");
   else
      ;
else
   System.out.println("lower than B.");
```

...or (better!) remove ambiguity and make the nested `if` a clearer statement by using braces...

```
if (grade >= 80)
{
   if (grade >= 90)
      System.out.println("gets an A.");
}
else
   System.out.println("lower than B.");
```

...or else (best of all) rearrange the logic.

```
if (grade < 80)
   System.out.println("lower than B.");
else if (grade >= 90)
   System.out.println("gets an A.");
```