# CMS 3145 Week 5 Class Notes: Chapter 8, Control Statements

**OBJECTIVES:**

1. **Review programming conditional constructs in Java (if/else and switch)**
2. **Review logical and relational operators**
3. **Understand string comparisons methods**
4. **Review While and For loops**
5. **Code break and continue statements**
6. **Use static methods**

**Sequence, Selection, and iteration**

All programs use these 3 constructs.

* Sequence:
  + **Input** comes before **processing** which comes before **output**.
  + Lines of code must be written in the correct order.
* Selection:
  + If statements and Switch statements guide the flow of the program based on changing input.
  + Without selection the program would do the exact same thing every time it is run.
* Iteration:
  + Allows the program to repeat a set of code as many times as needed.

**“Selection” or Conditional Branching**: **If** statements provide the means to execute a series of program instructions based on a defined *condition* being **True** or **False.**

General forms of the If conditional statements are:

**if (*condition*)**

Single statement;

**if (*condition*)**

{

Multiple statements;

Multiple statements;

## }

**if (*condition*)**

{Block of code}

**else**

{Block of code}

i**f (*condition*)**

{Block of code}

**else If**

{Block of code}

**else if**

{Block of code}

IF statements can be **nested** (also called **embedded**)**.**  However one should be careful to maintain code structures that are clear and easily followed. Nested IF statements should be indented or identified in a clear manner.

**The “Condition”**

A **condition** is any code that evaluates as a Boolean (true or false).

There are 6 **relational operators** used in conditional statements. Use these operators for **Primitive** data types.

Equality ==

Inequality !=

Greater Than >

Less Than <

Greater Than or Equal >=

Less Than or Equal <=

Since Strings are **objects**, they must use the String class “equals” method to compare the content of the string. Using the **equality** **operator** (==) on string objects will determine if the objects are the same, not the content of the objects.

The **logical operators** () are used to combine together two Boolean Expressions.

|  |  |  |
| --- | --- | --- |
| And | & | both expressions are evaluated and must be true for the combined condition to be true |
| Or | | | both expressions are evaluated |
| Not | ! | Reverse the value |
| And | && | both expressions must be true and the 2nd is evaluated only if the first is true (a.k.a. “Short-circuit operator”) |
| Or | || | Either expressions can be true and the 2nd is evaluated only if the first is false (a.k.a. “Short-circuit operator”) |

**Switch Statement**

The **Switch** construct is a conditional construct for testing more than 2 alternatives. It can do the same thing as a set of If statements but it can be easier to read and maintain.

The “**switch expression**” is either a string or integer variable.

The “**label**” is a possible value of the “**switch expression**”.

This is the official syntax for the switch statement:

Switch (switch Expression)

{

Case label1:

Statements1;

break;

Case label2: 🡨 optional

Statements2;

break;

}

An equivalent If statement is:

if (switch Expression==label1)

{Statements1}

else if (switch Expression==label2)

{Statements2}

The statements that go inside of the blocks of code for each case or condition should be unique. If each block of code ends with the same line of code then that line of code should be removed from the block and put after the switch or if statement. If the line of code,

System.out.print(productDescription);

is found in every case then it is redundant and will be put after the switch or if statement.

**“Iterations” or Loops:** **Loops** will run one or more lines of code over and over again. **While** loops are good for when we don’t know the number of times a loop will run, the **For** loop is used when we do know the number of times a loop will be run.

**While Loops**

There are two kinds of While loops:

* While loop a.k.a. pre-test
* Do While loop a.k.a. post –test

As long as the condition is true the loop will keep running.

The “while loop” has a test condition at the beginning of the loop. It is possible that the code in the loop is never run if the condition fails.

The do-while loop has the while statement test a condition after the statements are run. Thus the statements are guaranteed to run at least once. A **counter** variable is often used to count how many times the while loop is run. The **counter** is also used to track how many rows of an array have data in them (locationArray[counter] = input;).

**For Loops**

The **For** loop must use a counter variable that is an integer.

The **For** loop has three inputs that determine the number of times the loop executes:

* Initialization Expression
* Boolean Expression
* Increment Expression

Where does the counter variable start?

The Initialization Expression sets the first value of the counter variable. The counter variable can be defined before the loop or as part of the “Initialization Expression”.

Where does the counter variable stop?

The Boolean Expression determines when the loop stops. When this expression becomes false the loop stops.

What size steps does the counter variable take?

The Increment Expression determines how much a counter variable changes each time a loop occurs. Normally we count by ones, but we could count by any value: 2’s, 3’s, 5’s, 10’s, etc. We can also count down by decrementing the counter.

**General Loop concepts**

We can nest one loop inside of another.

We can use a ‘**break**’ statement to leave a loop before the Boolean conditions are meet. The **break** bypasses the **Boolean Expression**. The ‘**continue**’ statement is used to skip any remaining lines inside the loop and go to the next increment of the counter variables.

With many embedded loops, the ‘**labeled’** break or continue statement will take us to a specific point in the embedded loop hierarchy.

Counters and Accumulators

Loops are useful for adding many values into a variable for later calculations. For example the price of each item on a grocery list is summed together for the subtotal. This kind of variable is called an **accumulator**. An integer variable that holds how many items are purchased is called a **counter**. The average price would be the accumulator / counter.

**Static methods**

The methods that are accessed through a class are considered **Static Methods**. The Math class, discussed in chapter 3, has the min() and max() static methods. We can create methods in the class of our program.

By using the **static** keyword the method is available through the class. Remember that the purpose of the method is to make a contained module of code that can be called from multiple places. That is the code can easily be reused. It makes it easier to read and maintain the code.

If the method is declared private than the static method can only be called from inside the class. If the method is public we must specify the method with the class name. For example Math.max(5,20) uses the name of the class, Math, when calling the method.

In later chapters we will create object or instance methods. All methods either return a value (the return type equals a *data type*) like a **function**, or don’t return a value (return type is *void*) like a **sub procedure**. Also all methods can have any number of parameters in the definition.

When a method is called, every *parameter* must have a corresponding *argument*.

In the definition of the **max** method, ‘a’ and ‘b’ are *parameters* of the integer data type.

[**max**](http://docs.oracle.com/javase/8/docs/api/java/lang/Math.html#max-int-int-)(int a, int b)

When the method is called, ‘5’ and ‘20’ are *arguments*.

Math.max(5,20)