LESSON 3
CONTROL STRUCTURES, PART I

INTRODUCTION

In this lesson, we'll discuss the different types of control structures in Java, and take a look at a few examples. Also, we will look at the **modulus operator,** and some **compound operators**. The `String` data type is also introduced.

3.1 CONTROL STRUCTURES

The phrase **control structure** refers to how the source code is organized to control how the different statements execute (or don't execute.)

There are three fundamental categories of control structures, one of which we've seen already without knowing it.

- Sequential (Linear)
- Selection (Branching, Decision)
- Iterative (Repetition)

3.1.1 LOGICAL OPERATORS AND COMPARISON OPERATORS

Before we can explore the selection and iterative control structures, we must understand other types of operators, in addition to the arithmetic operators we learned about the last lesson.

Comparison operators are probably those you are familiar with from basic arithmetic:

Symbol	Name/Description
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Is equal to
!=	Is NOT equal to

Logical operators are operators that evaluate to **Boolean** values, which just means that they evaluate to either **true** or **false**. Sometimes (but not always) these are used in conditions in selection and iterative control structures.

Symbol	Name	Meaning
&&	Logical AND	A **binary** operator (takes two **operands**) that evaluates to true if and only if *both* operands are true. For example: X > 5 && X < 10 would only evaluate to true if the value X was strictly in between 5 and 10.
||	Logical OR	A binary operator that evaluates to true if and only if at least one of the operands is true. For example: Age > 10 || Weight > 100 would evaluate to true if the value of Age is greater than 10 OR the Weight is greater than 100, OR even both. As long as *one* is true, the entire statement is true
!	Logical NOT	A unary operator (takes only one operator) that negates (reverses) the truth value of its operand. For example: !(x > 10) returns true if x is NOT greater than 10. In other words x > 10 is evaluated first, and then the ! applies to the entire statement and reverses its truth value.

3.1.2 Sequential Control Structures

We've already seen **sequential control structure**, in which code is simply executed one statement after another. For example, from our last lesson, we had code such as:

```
        System.out.println("Difference: " + diff);
        System.out.println("Product: " + prod);
       System.out.println("Quotient: " + quot);
```

All three of these lines, which print information to the console, execute one right after the other.

3.1.3 Selection (Decision) Control Structures

Selection control structures are structures in which blocks of code are either executed, or not, depending on a particular **condition** or conditions.

There are three categories of selection control structures:

- Single selection control structures
  - `if` statements
- Double selection control structures
  - `if – else` statements
- Multiple selection control structures
  - `switch` statements

We'll take a look at the `if` single selection control structure and the closely related `if-else` double selection control structure in this lesson.

A **single selection control structure,** which is the `if` statement in Java, either executes something or doesn't. This requires us to use a comparison of some sort. The best way to observe this is to see it in action.

```
1   public class L2_ControlStructures {
2
3         public static void main(String[] args)
4         {
5               int x = 10;
6               int y = 15;
7
8               if(x > y)
9               {
10                    System.out.println("x is greater than y");
11              }
12
13              System.out.println("This will execute no matter what.");
14        }
15  }
```

Code 3.1-1

The output is:

```
This will execute no matter what.
```

Line 8 uses an `if` statement and compares x and y using the greater than (>) operator. Since x is *not* greater than y, the code block that the if statement applies to does not execute.

Technically, we can simplify the above code if there is only one statement to be executed. Specifically,

```
if(x > y)
{
     System.out.println("x is greater than y");
}
```

Can be shortened to:

```
if(x > y)
     System.out.println("x is greater than y");
```

This shortened syntax however, ***only works if there is only one statement to execute***. I almost always use the original syntax with the curly braces, { and }, because it is less likely there will be mistakes made. So even if there is only one statement to be executed, I still use the curly braces. That is often a matter of style, and some people consider the curly brace syntax with only one executable statement to be too verbose. I consider it a trade-off and worth the extra couple keystrokes.

The **`if-else` double selection statement** is useful when you want to select between two possible courses of action depending on a condition.

```
1   public class L2_ControlStructures {
2
3         public static void main(String[] args)
4         {
5               int x = 10;
6               int y = 15;
7
8               if(x > y)
9               {
10                    System.out.println("x is greater than y");
11              }
12              else
13              {
14                    System.out.println("x is not greater than y");
15              }
16
17              System.out.println("This will execute no matter what.");
18        }
19  }
```

Code 3.1-2

Since x is not greater than y, the code in the else block (line 14) will execute.

Thus, the output is:

```
x is not greater than y
This will execute no matter what.
```

Note that we can also use **cascading if-else statements** to simulate a multiple-selection scenario. For example:

```
1  public class L2_ControlStructures {
2
3        public static void main(String[] args)
4        {
5              int x = 10;
6              int y = 15;
7
8              if(x > y)
9              {
10                   System.out.println("x is greater than y");
11             }
12             else if (x == y)
13             {
14                   System.out.println("x is equal to y");
15             }
16             else   //x < y
17             {
18                   System.out.println("x is less than y");
19             }
20
21             System.out.println("This will execute no matter what.");
22       }
23 }
```

Code 3.1-3

The output from this is the same as the previous, but you'll notice three possibilities. The first is on line 8. The second is on line 12, this time, if the values of x and y are equal. And finally, the condition on line 16 is implied to be x < y, because there are no other possibilities. An `else` without any `if` keyword and a condition is a catch-all, or default for the remainder of possible values.

3.1.4 Iterative (Repetition) Control Structures

Sometimes we want a statement or set of statements to execute more than one time. Instead of writing the statement over and over again, we can use an **iterative** (**repetition**) **control structure**, often realized in the form of a **looping statement**.

There are three primary iterative control structures in Java:

- `while` loops
- `do-while` loops
- `for` loops
    - Also, a modification of the `for` loop exists called `for-each`

In this lesson, we'll only concern ourselves with the while loop. An example will help us here.

```
1  public class L2_ControlStructures {
2
3          public static void main(String[] args)
4          {
5
6                  int counter = 0;
7
8                  while(counter < 10)
9                  {
10                         System.out.println("In the loop!");
11                         counter = counter + 1;
12                 }
13         }
14 }
```

Code 3.1-4

In the above code, the `while` loop and its body is present on lines 8-12. Notice that with a `while` loop, we often need a `counter` of some sort, which we declare on line 6, initializing it to 0. Then, on line 8, we have the while keyword and the **loop continuation condition**. What this particular loop continuation condition says is, "As long as `counter` is less than 10, keep doing what is in the code block of the body of this loop."

Now, we have to move from the current value of our `counter` variable toward the termination condition. If we don't update this variable, we get what is called an **infinite loop**, meaning a loop that keeps going until you close the program, restart the system, or until memory runs out or some other bad program state occurs and the operating system terminates the process. It will effectively freeze the program.

Thus, on line 11 we **increment** the value of `counter` by one. The syntax we use in Code 3.1-4 says to "take the value of `counter` and add 1 to it, then store it back in the `counter` variable."

There are however, shorthand versions of this `counter` incrementing that we can use.

Since using counters and specifically, adding 1 to a `counter` is so common in programming, there is actually a special unary operator for that purpose. The following code would be more common that the code in Code 3.1-4.

```
1  public class L2_ControlStructures {
2
3          public static void main(String[] args)
4          {
5
6                  int counter = 0;
7
8                  while(counter < 10)
9                  {
10                         System.out.println("In the loop!");
11                         counter++;
12                 }
13         }
14 }
```

Code 3.1-5

Notice on line 11, we have the syntax

```
counter++;
```

This is called the **post increment operator**, and the only thing it does is add 1 to its operand, in this case, `counter`.

There are four different operators like this, known as increment and **decrement operators**.

Operator (as applied to an operand)	Name/Description
w++;	Post-increment operator. Returns the value of w, and then increments w by 1.
++w;	Pre-increment operator. Increments w by 1, then returns the value of w.
w--;	Post-decrement operator. Returns the value of w, then decrements w by 1.
--w;	Pre-decrement operator. Decrements w by 1, then returns the value of w.

Be careful to note the difference between the pre- and post- versions of the operators above. Check out what the difference between the following code segments is:

```
1  public class L2_ControlStructures {
2
3        public static void main(String[] args)
4        {
5
6              int someNum = 5;
7              int anotherNum1 = someNum++;
8              int anotherNum2 = ++someNum;
9
10             System.out.println("AnotherNum1 = " + anotherNum1);
11             System.out.println("AnotherNum2 = " + anotherNum2);
12
13       }
14 }
```

Code 3.1-6

As a related issue, there is another way to add 1 to a variable. We can use the **compound assignment operator for addition**:

```
counter += 1;
```

In this case, however, we can put any number as the right-hand operand, since this operator is a binary operator. Compound assignment operators exist for many different arithmetic operators:

Compound operator	Corresponding regular operator	Description/Name
+	+=	Addition. x += y; Adds y to the variable x and

		stores the sum in x.
-	-=	Subtraction. x -= y; Subtracts y from x and stores that difference in x.
*	*=	Multiplication. x *= y; Multiplies x by y and stores the product in x.
/	/=	Division. x /= y; Divides x by y and stores the quotient in x.
%	%/	**Modulus**. x %/ y; Divides x by y and stores the **remainder** in x.

An important new thing that we notice in the above table is the **modulus operator**. The modulus operator by itself, or in its compound form, returns the **remainder** (also sometimes called the **residue**) of a division operation. While the division operator (/) finds the **quotient**, the modulus finds the remainder.

3.2 THE `String` DATA TYPE (AND A LITTLE BIT ON MEMORY)

The String class allows us to create objects that hold text strings. There are also some very helpful methods associated with this class.

```
1   public class L2_ControlStructures {
2
3         public static void main(String[] args)
4         {
5
6               String name = "John Baugh";
7
8               System.out.println("Hello there, " + name);
9
10        }
11  }
```

So, just like **primitive data types** like int and double can hold primitive data like integers and real numbers, respectively, a String object can hold a string.

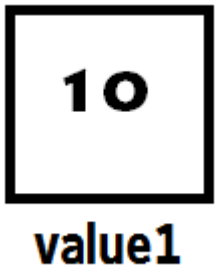
There is a distinction that is important, however, with how the memory is reserved and addressed with primitive data types versus **class data types**.

A variable of a primitive data type *directly* holds the value you assign it.

So, consider the following code:

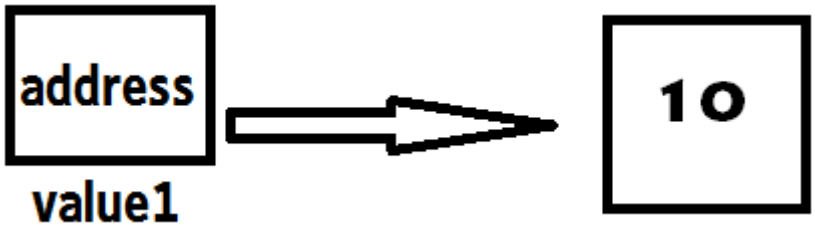
```
int value1 = 10;
```

Then the memory looks like this:



The variable `value1` directly refers to the memory location.

However, with a class data type, the variable you use refers to a **memory address** which then refers to a value *indirectly*.



In this case, **address** would be some memory location, and then when the value is to be obtained, the system checks that address and gets the value.

The distinction should be noted, but for our purposes right now, we really don't have to worry too much about how the memory is reserved. In Java, a String variable can be accessed in code essentially the same way a primitive type would be.

3.2.1 Some Helpful Methods of the String Class

Here are some handy methods of the String class, and what they do:

Method	Name/Description
length	Returns the length of the string. In other words, this method returns the number of characters in the string.
toLowerCase	Returns a new string that is the lowercase version of the string that the calling object contains.
toUpperCase	Returns a new string that is the uppercase version of the string that the calling object contains.

All three of these methods can be demonstrated as follows:

```
1   public class L2_ControlStructures {
2
3         public static void main(String[] args)
4         {
5
6               String name = "John Baugh";
7
8               int numChars = name.length();
9               String lowerName = name.toLowerCase();
10              String upperName = name.toUpperCase();
11
12              System.out.println("Original String: " + name);
13              System.out.println("Number of chars: " + numChars);
14              System.out.println("Lower version: " + lowerName);
15              System.out.println("Upper version: " + upperName);
16
17        }
18  }
```

The output is as follows:

```
Original String: John Baugh
Number of chars: 10
Lower version: john baugh
Upper version: JOHN BAUGH
```

EXERCISES

1. Write a program to print out the counter variable itself in a while loop, from zero (0) up to twenty (20).
Hint: Consider using the <= instead of just <. Also, make sure to change the counter's value in the body of the loop!

2. Modify the program in Exercise 1 so that if the counter at that iteration is *even*, "<counter> is even", and if it's odd, "<counter> is odd" are printed out. Note that <counter> should be replaced by a value of counter at that iteration.
Hint: Combine the iterative control structure (while loop) with a selection control structure (if, or if-else)

3. Declare a string with your name in it. Then, write a program to loop 3 times. At each iteration, it should do something different:

- If counter is 0, print out the original string (e.g, name)
- If the counter is 1, print out the name in all upper case
- If the counter is 2, print out the name in all lower case