# ITSC 1212 Module 2 Lab

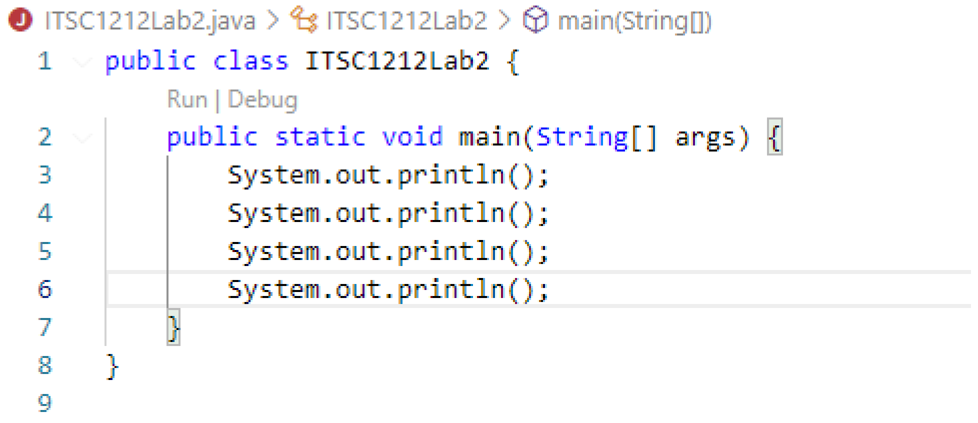
In this lab, we'll be building off of what we learned last week by introducing new types of variables and mathematical expressions.

**Concepts covered in this lab:**
- Primitive variables
- Casting
- Modulo

**Required files or links**
- N/A

## Part A: Exploring More Primitive types

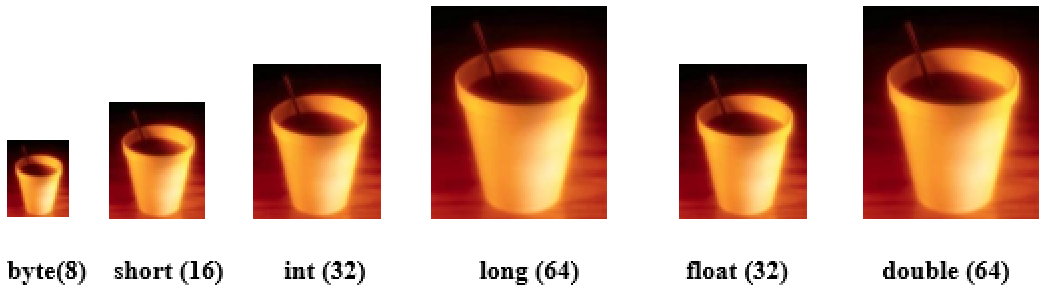
1. In Lab 1, we introduced you to *int* and *double* variables. However, Java has eight total primitive types, and although you will rarely use most of them, it is important to at least know what they are.
2. Create a new file in your ITSC1212 folder called ITSC1212Lab2.java. Give it a main method the same way we did in Lab 1.
3. Create several empty print line statements. We will use these to demonstrate the output and interactions of different primitive types in future steps, and creating those now will help us save time. At this point, your class should look something like this:



```
public class ITSC1212Lab2 {
    public static void main(String[] args) {
        System.out.println();
        System.out.println();
        System.out.println();
        System.out.println();
    }
}
```

4. Of the 8 primitive types, 6 of those are used for numerical values. They are: byte, short, int, long, float, double. Byte, short, int, and long hold whole numbers, float and double hold real numbers (things with decimal points). The reason we have 6 different types just for numbers, and the best way to think of them in an

abstract sense, is to imagine them as containers. Each different type holds the same thing, but the difference is in the size of the container. The size of the container is actually the number of bits each type can hold in memory, which affects the highest number those variables can hold. You can think of it like this:



5. You will almost certainly use int and double for most of your numerical needs. The other types, especially byte and short, have very limited use cases these days because of the large amount of memory most computers have. For instance, a byte container can only hold numbers between -128 and 127 which limits its usefulness, but can save on space when memory is at a premium.
6. Back in your Java class, create three short variables: x, y, and z. Initialize them to values 0, 2000, and 100000 respectively. Make sure you add these before the print statements we have since we will be referencing them there.
7. You'll notice that for z, we have red squigglies indicating an error. What does the error message say, and what do you think it's trying to tell you?
8. You can fix this error by making z an *int*, which increases the container size enough to contain that particular value.
9. In the first print statement that you wrote in step 3, add these three variables together inside the parentheses. Before you attempt to run your code, stop and think: will this work? And if it does, what type of variable would be needed to store the resulting value? Make a note of your answer, or write it below

______________________________________________________________

______________________________________________________________

10. The last two primitive variable types to explore are *char* and *boolean*. Unlike the previous types of primitives, these hold much different types of data. A *char*, for example, holds a single character of text.

11. Create two *char* variables in your class like so. Note that the value of a char is placed between single quotes.

```
char a = 'a';
char b = 'b';
```

12. Put these variables inside the empty print statements and run your code. Experiment with different values if you like, but more than likely there won't be any surprises.
13. One quirk of *char* variables is how they are stored in memory. A *char* is stored as Unicode, which means it is stored as a numerical value that is interpreted and displayed as a character. To demonstrate that, change one of your print statements so that your two char variables are added to one another.

```
char a = 'a';
char b = 'b';

//System.out.println(x + y + z);
System.out.println(a + b);
```

14. What value is printed and why?

15. The last primitive type to discuss is called a *boolean*. A *boolean* holds either one of two values: true or false. Create a boolean variable and initialize it to true.

```
boolean boo = true;
```

16. Notice that the value "true" here is in blue text, which means that it is a reserved keyword.
17. Modify your remaining print statements so they look like this:

```
    System.out.println(x < y);
    System.out.println(y > z);
    System.out.println(z < a);
```

18. Before you run your program, try to guess what value will be printed for each of these lines based on their values.
19. In the previous lab, we introduced you to arithmetic expressions using mathematical operators such as + or *. The operators you just used are called *relational operators*, because they examine the relationship between two values and give us either true or false depending on the comparison of those values. Fill

in the chart below, explaining the function of each relational operator. If you don't recognize one, test it with the code you have written to determine what it is doing.

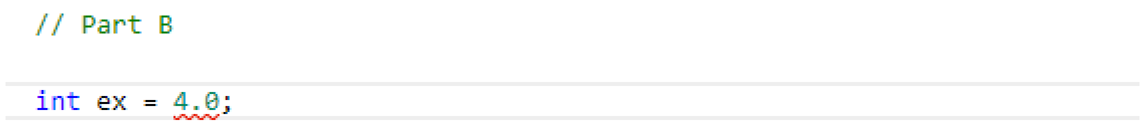
| Operator | Function |
|---|---|
| > | |
| >= | |
| < | |
| <= | |
| == | |
| != | |

20. When you are done, show the instructor or TA your chart and code to get credit for this section

## Part B: Casting

To start this section, comment-out all the code from Part A, and create a new comment to denote the start of this section.

1. In Java, variables are hard-typed, which means that a variable can only hold the types of values compatible with the type it is declared. For instance, if we declare a variable X as an int, it can only ever hold whole number values.
2. But this is not to say that one type of value cannot be converted to another. The process by which we do this is called *casting*, and although it is a fairly simple process it can still be a stumbling block for new coders. To start, let's create a new *int* variable and attempt to set it to a real number value like so:



```
// Part B

int ex = 4.0;
```

3. Notice from the squiggles that we have an error. Specifically, the error text tells us we have a type mismatch because an int variable cannot hold a real number.
4. We can, however, use casting to convert this real number value to a whole number. To cast something to a new type, we simply put the desired type in

parentheses BEFORE the value we are attempting to convert, like so:

```java
int ex = (int)4.0;
System.out.println(ex);
```

5. Add in the print statement seen above, and run your code. What was the value of *ex*? If you change 4.0 to 4.9, is the same thing printed? Why?
6. Copy and run the code seen below.
7. As you can see, the placement of the (int) casting is extremely important and must be done with care. Can you determine why each answer is so different?

```java
double x = 8.8;
double y = 2.2;

System.out.println(x / y);
System.out.println(x / (int) y);
System.out.println((int)x / y);
System.out.println((int)(x / y));
```

8. Now that you understand how to cast numbers, let's demonstrate how casting can be combined with arithmetic operations to do some clever things.
9. Copy the following code into your program and run your code. (It may help to comment out the earlier parts of this section)

```java
double cost = 10.10;
double paid = 12.33;
double change = paid - cost;
System.out.println("The change owed on this transation is: " + change);
```

10. When you run this code, you should see this as the output

```
The change owed on this transation is: 2.2300000000000004
```

11. This sort of thing happens from time to time with arithmetic involving decimal places in code, and is an unfortunate quirk of how these numbers are stored in memory. But one of the things we are accustomed to when dealing with dollar amounts is to only see two decimal places, and it would be helpful to be able to eliminate those extra numbers so we see this in the format we expect.
12. Using the provided pseudocode below, use casting to eliminate those pesky decimal places until we only see the number in the expected format.
    a. Multiply change by 100 to move the decimal point over 2 places
    b. Convert the change to a whole number to drop the remaining decimal points

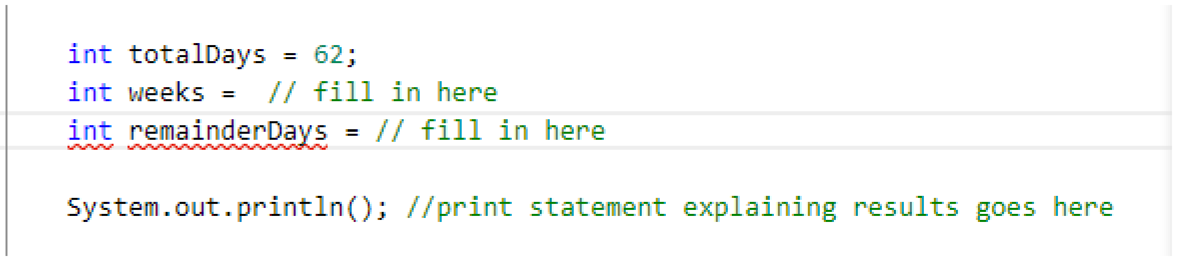
    c. Convert back to a double to reintroduce the decimal points
    d. Divide the change by 100 to put the decimal point back in its original place.

13. Once this is working the way you think it should, test the process with a series of different numbers for cost and paid. If you broke this process down into multiple lines, figure out how to combine these steps into a single line of code.
14. When this conversion is working to your satisfaction, show your code and it's output to the instructor or a TA to get credit for this section.

## Part C: Modulo

Start this section by commenting out the previous section, and adding a new comment header.

1. There is one more operator we need to practice, and it is the one you are likely least familiar with and can be hard to appreciate: the modulo operator (%). The modulo operator, typically referred to as "mod" for short, gives the remainder of a division operation. So this means that:
    a. 10 % 5 = 0
    b. 14 % 5 = 4
    c. 7 % 5 = 2
    d. 2 % 5 = 2
2. What the modulo operator does is fairly straightforward, but it's usefulness can be hard to appreciate at first glance. But consider this: given a number of days, let's say 62, we can easily convert that into weeks and days using a combination of division and modulo like so:
    a. First, divide 62 by 7 to get the number of weeks (it's important to remember here that in Java, integer division always yields whole numbers. We don't need to worry about this step yielding a decimal point)
    b. Then, 62 % 7 will give us the number of days remaining after we subtract the whole weeks from 62.
    c. This gives us a final answer of 8 weeks and 6 days.
3. Use the follow code template to convert the pseudocode steps above into Java

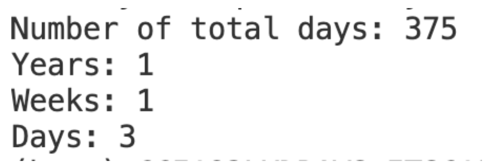


```
int totalDays = 62;
int weeks =  // fill in here
int remainderDays = // fill in here

System.out.println(); //print statement explaining results goes here
```

*Refer back to Part B Step 9 for how you might format the print statement using variables*

4. One thing to notice in this template code is that by using variables to store our various values, we create steps that will work no matter what change we make to the *totalDays* variable.
5. Test your code with several different values of totalDays, making sure to verify each result. Be sure to also test at least one day value less than 6.
6. Once you feel confident that you understand this process, use the same logic and process to expand this to include years, weeks and days. Your output at the end should look something like this:



```
Number of total days: 375
Years: 1
Weeks: 1
Days: 3
```

7. Show your work to an instructor or TA to receive feedback and credit for this section.

**Looking ahead**: In the remaining lab time, you should take a look through Project 1.