Class 1

Goals

1. Variables
2. `if` statements
3. Methods
4. Testing our code
5. Commenting/Good practices

Variables

What are variables? If you know some Algebra or Pre-Algebra, you know that we use variables in math to represent numbers in our equations. Essentially, a variable is an unknown value that we are trying to solve for. In programming, variables are also used to represent values.

In programming, we use variables all the time to store information that we would want to access later. There are various forms of information we could store. For example, we may want to store numbers, words, or even a variety of these things. The type of information we want to store is usually called the `type` of the variable. For example, the variable storing the phrase "sit vis vobiscum", would be of `type` string (in java, words are referenced as strings).

Some key words that you need to be aware of when discussing variables:

1. **Declaration**
2. **Initalization**
3. **type**

We have previous defined type, but we still need to define declaration and initialization.

A **declaration** is defined as the act of creating a new variable, but not assigning it a value. In Java, we declare variables as follows:

type name

Say that we wanted to make a variable to store an integer. We would do that by doing:

```java
int anIntegerVariable;
```

In this case, we have **declared** a new variable named `anIntegerVariable` of **type** `int`.

Exercise: Look at the following variables and then break down them down like we did for `anIntegerVariable`.

```java
    String myName;
    double pi;
    int aName;
    float someNum;
```

So far, we have only covered how to declare a variable. This means that all of our variables are set to `null` and we can't use them in our code. In order to assign values to our variables we must **intialize** them. We can initialize variables by setting them equal to some value. For example:

```java
    String myName;//declaring the variable
    myName = "Aravind";//initializing the variable
```

We first declared the variable (`myName`) and then we assigned it the value "Aravind". This is an example of initalizing a variable. It is important to note that in java, variables accept their assignments from the right. This means that

```java
    String myName;
    "Aravind" = myName;
```

will throw an error. This is the case for most programming languages you will encounter.

Exercise: Declare and initialize a variable for each of the following types:

```java
int
float
double
String
```

It gets annoying to declare and initialize a variable on different lines. It's much easier to do both at the same time. We can easily do this by:

```
    int myAge = 17 //declaring and initializing a variable at the same time
```

Note that it is common practice to declare and initialize variables on the same line. There are situations where it may not be possible to do both, but when you can, you should.

Now that we know about variables and how to use them, we can start doing some basic tasks in java. I'm a strong believer in application, so many of my example will come in the form of code. Here is the first example:

```
int num1 = 10;
int num2 = 20;
int num3 = num1 * num2;//What are we doing here?
int num4 = num3 / num1;//Is num4 always equal to num2?
```

In this example, we are introduced to something we haven't seen before: defining a variable in terms of other variables. In our example, `num3` is defined to be the product of `num1` and `num2`.Although we will always know the numerical values of `num1` and `num2`, we won't always know the numerical value of `num3`. Sure when `num1` and `num2` are small, we can calculate the value of `num3`, we won't be able to do so when `num1` and `num2` are large. The beauty of this situation is that we always know what the value of `num3` is in relation to that of `num1` and `num2`, but we may not always know the numerical value.

Exercise: Initialize two `String` variables with your first and last name. Make a third variable that will store your entire name, but you must define the third variable in terms of the first two.

`if` Statements

Just like in real life, we need to be able to make decisions in our code. What if we only want the robot to turn when a button is pushed or if a joystick is being used? In situations like this, we can use `if` statements to control the flow of our code. The way that an `if` statement works is pretty simple: we put a condition in the code, and if that condition is met, then we run a different set of code. Here is how we would construct an `if` statement:

```
    if(/*condition goes here*/){
        //code goes here
    }
```

A condition can be any logical operation that yields a **boolean**. A **boolean** can only either be true or false. Some examples of boolean operators in java are $>=$ ($\geq$), $<=$ ($\leq$), and $==$.

The code within an `if` statement will execute if and only if the condition is met (true). Say for example we have the following code:

```
int a = 10;
int b = 12;
int c = a + b;

if(c > 20){
    System.out.println("Greater than 20!");
}
```

Let's take a close look at what the code is doing in this situation. We declared three `int` variables: a,b, and c. Next, we used an `if` statement to check if c was greater than 20. Since `c = 12 + 10`, we know that `c` = 22. This means that `c > 20` and that the condition is satisfied. This means that the code would print "Greater than 20!".

Exercise: What would happen if a = 5 and b = 10? Would the code print "Greater than 20!"?

So know how to write simple `if` statements now, but what if we want the code to do something else if the condition is not met. Say for example, that we want the code to tell us if the value of `c` is less than 20? We can use the `else` statement for these situations. An `else` statement is used after an `if` statement like so:

```
    if(/*condition here*/){
        //run this code if the condition was met
    } else{
        //run this code if the condition was not met
    }
```

So if the condition in the `if` statement is not satisfied, then the code in the else statement will run. Think of it as an ultimatum, if something does not happen, then this will happen. If you use an `if-else` block of code, then you can always be sure that either the `if` statement code will run or the `else` statement code will run. Let's expand our previous example to print out if `c` is less than 20:

```
    int a = 10;
    int b = 12;
    int c = a + b;

    if(c > 20){
        System.out.println("Greater than 20!");
    } else{
        System.out.println("Less than 20")
    }
```

Now, if the value of `c` is less than 20, the code will print "Less than 20".

Exercise: What happens if `c` = 20? Does the code do anything at all?

Thinking about the last exercise probably gave you a headache since it shows that we have a fault in our logic. If c = 20, then the code will print out "Less than 20" because that is how an `if-else` code block works. Since 20 is not less than 20, we go to the else and print out a false statement. This is a good example of "dumb programming". Computers are dumb by design and they can only do what you tell them to. It is up to you to determine edge cases like this in you code and to accommodate for them. Situations like this arise very often and they can cause your code to not work like it should.

Luckily, we have a solution to this problem. We can use the `else if` command to check to see if c = 20. The `else if` comes right after an `if` statement and before an `else` statement. Basically an `else if` says that if the condition in the `if` is not met, then check to see if the condition in the `else if` is met. If that condition is also not satisfied, then continue on to the `else` statement. Here is an example of how an `else if` would work:

```
    if(/*condition a*/){
        //code to run if condition a is met
    } else if{/* condition b*/}{
        //code to run if condition a not met, but condition b is met
    } else{
        //code to run neither conditions are met
    }
```

Now that we know how an `else if` statement works, let's modify our code from before to fix the bug that we encountered:

```
    int a = 10;
    int b = 12;
    int c = a + b;

    if(c > 20){
        System.out.println("Greater than 20!");
    } else if{c = 20}{
        System.out.println("Equal to 20");
    } else{
        System.out.println("Less than 20");
    }
```

Using this final version, we have fixed all of our bugs.

Exercise: Make a program that will compare two numbers to each other and print out which one is greater. If they are equal, print that the two numbers are equal. Make sure to use variables.

Methods

Now that we have a solid understand of variables and `if` statements we can begin putting them together in larger segments of code. Since Java is an object-oriented programming language, we can create blocks of code called methods. Basically, a method accepts some input or a list of **parameters** and then **returns** a single piece of information. In java the type of the information that the method will return is pre-defined, but there are other languages where this is not the case. It is also important to note that a method can only return information one time and can only return one piece of information. Let's look at how we would declare a method and break it down into simple terms:

```
public int mysteryMethod(int a, String b){
    //code goes here
}
```

- **Modifier**: public
- **return type**: int
- **Parameters**: (int a, String b)

A **modifier** describes the different forms that a method can take. Making a method public means that it can be accessed from anywhere. You can call it from another class and it can always be accessed. We will mostly be working with methods that are public, so it is important that you understand how to create public methods.

Previously I said that Java requires you return a pre-defined type. The **return type** is where you define what type you will be returning. Say that you want your function to return an `int` value, then you would make the **return type** int. In this situation, our `mysteryMethod` will return an `int`.

Methods are powerful since they are a set of instructions that we can apply to any code. **Parameters** are considered to be "input" and we need them to be defined each time the method is called. In this situation, whenever `mysteryMethod` is called, it need to be given an `int` variable and `String` variable. If we define a method to require **parameters**, it is mandatory that you provide those parameters when you call the method, otherwise, you will get an error.

Let's write a method that compares two numbers:

```java
public int compareTo(int a, int b){
    if(a == b){
        return 0;
    } else if{a > b}{
        return 1;
    } else{
        return -1;
    }
}
```

Let's break this method down and look at what it is doing. There are a few essential questions you need to ask yourself when reading a method to determine its function:

1. What is the name of the method?
2. What is the return type of the method?
3. What is the modifier of the method?
4. What are the parameters and how are they used?
5. What is the method trying to accomplish (read the code within the method)?

The first four questions have to do with how we declare the method.

Exercise: Answer the first four questions about the sample method.

Alright now we are faced with the question of what this method doing? If we look closely at the code we are essentially just comparing two numbers and returning either 1, 0, or -1 depending on the relationship between a and b.

When looking at this method, we are faced with a keyword that we have not seen before: `return`. Previously, I said that every method can only return a single piece of information. The `return` keyword ensures that you are giving a value back when the method is called. I understand this is a little abstract, but try to follow me through this example. Say that we have the following method:

```
public int addThis(int a, int b){
    return a+b;
}
```

Suppose that we call this method in the following manner:

```
int sum = addThis(7,9);
```

This expression is the equivalent of saying:

```
int sum = 16;
```

Hopefully you understand what the `return` keyword does. If you don't feel free to ask me a question the next time you see me.

Exercise: Make a method that accepts two integers. If the sum of the integers are greater than 50, return the sum of the integers. If the product of the two numbers is greater than 20 return the product of the two integers. If neither of these conditions is met, return 0.

Testing our Code

We currently have all the tools we need to be to able to write some simple code. Before we can begin writing larger methods and making cool code, we need to learn a little bit about how classes are organized.

Say that we make a new **class** in Android Studio named `test`. We can expect to see the following in the class:

```
public class test{

}
```

We really can't run any code with just this template. We need to add a special method in the class in order to run code. It should look like:

```
public class test{

    //methods and stuff go here

    public static void main(String[] args){

        //tester code goes here

    }

}
```