Lesson 3

Object-Oriented Programming

Before we look at Object-Oriented Programming, let's look at this code:

```
String model = “Subaru”;
String color = “Orange”;
int time = 0;
double position = 0;
double velocity = 3;
double acceleration = 1.3;
```

```
public void Update(int dt){
	for(int i = 0; i < dt; i++){
		velocity += acceleration;
		position += velocity;
		time++;
	}
}
```

-This code is describing a car. The method Update is updating the car’s position after *dt* seconds. This code seems good, but what if we wanted multiple cars, each with their own models, colors, position, velocity , and acceleration?

What if we wanted multiple cars, each with their own models, colors, position, velocity , and acceleration?

Here are some possible solutions:

- Copy-pasting a bunch of variables
- Making arrays to store all the information (don’t worry about this)

Or….

- Use Object Oriented Programming!

Welcome to Object-Oriented Programming! (OOP)

Object-oriented programming means to create several different objects that each have their own unique methods (behaviors), and variables (fields).
Instead of lumping them together into one giant java file, it sections it off into different objects, to take advantage of when we need to code many similar things.
For instance, an object could be a car. With an object, instead of completely rebuilding every new car, we would have some kind of basic framework for a car that we would only need to make modifications of

What Are Classes?

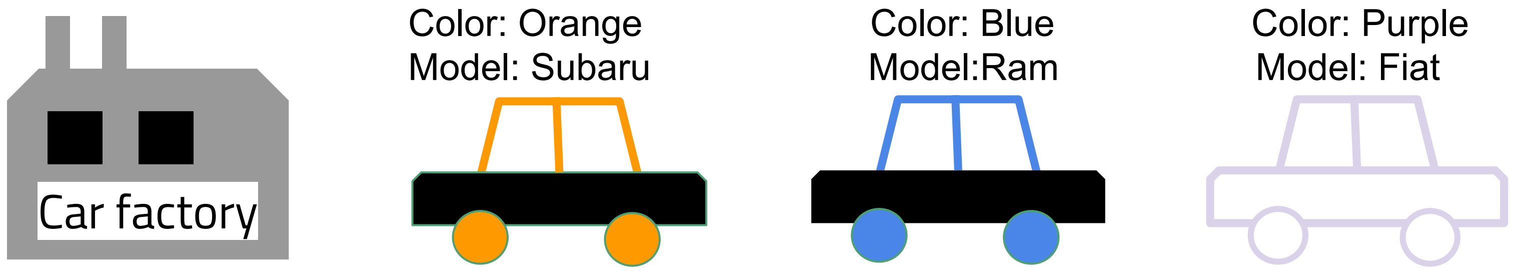
A class is a blueprint for an object. It is here where you write the fields and behaviors that an object will later on have. To get an object, you would need to instantiate a class, and access the methods and variables inside of the class

```
public class <classname> {

    //Code to put inside of class

}
```

Everything that you have been writing so far has been in a class.

What are instances (Or Objects)?

Let’s look at the image below:



Right over here, there are 3 **instances** of cars. Each of them have their own **fields** that they can update. However, how does the factory know what to make? We need to define what a car is if we want to make them.

Constructors

Constructors tell the program how to create an instance of a class. It is in the format of a method, but it has the same name as the class name. It has parameters that specify what values will need to be specified when you instantiate the class (or create the object). Multiple constructors are possible for different amounts of input

```
public Car(String MyModel, String MyColor) {
     Model = MyModel;
     color = MyColor;
}
//To instantiate:
Car niceCar = new Car(“go kart”, “#CC293C”);
//Color is set to user input
```

```
public Car(String MyModel) {
     Model = MyModel;
     color = red;
}
//To instantiate
Car niceCar = new Car(“Tesla”);
//Color is set to red
```

Instantiating the class is like creating a variable, except its type is defined as the class. (eg. `int five = 5`). Just like the int five is being set to 5, the Car carname is being set to an object called Car();

How to create(instantiate) an Object

```
Car niceCar = new Car("Subaru","Orange");
```

This statement creates a new car with the model Subaru, and the color Orange.

```
public Car(String myModel, String myColor){
                model = myModel;
                color = myColor;
        }
```

"Subaru" would replace everywhere that says model and "Orange" would replace everywhere that says color.

(Live coding remake class car)

The Car class

```
public class Car{ //name of class
        //properties of a car
        public String Model;
        public String color;
        /*constructor: a method that has the The exact same name as the class. Immediately called when an
object is first made*/
        public Car(String myModel, String myColor){
                Model = myModel;
                color = myColor;
        }
        //methods
        public void Update(){
                .......
        }
}
```

Here, we define what a car is. By typing class Car{}, you are stating there is an object called Car. At the top of the class, you state what properties a Car has, such as the color.

Grabbing the Fields and Behaviors of a Class

To access the behaviors and fields defined by a class, like car, simply write the name of the field after the name of the instance, separated by a dot. It has the same effect as grabbing methods and variables as if they were in the same file.

```
niceCar.Update(); //Executes Update()
```

```
niceCar.Model; //Gets the value of Model
```

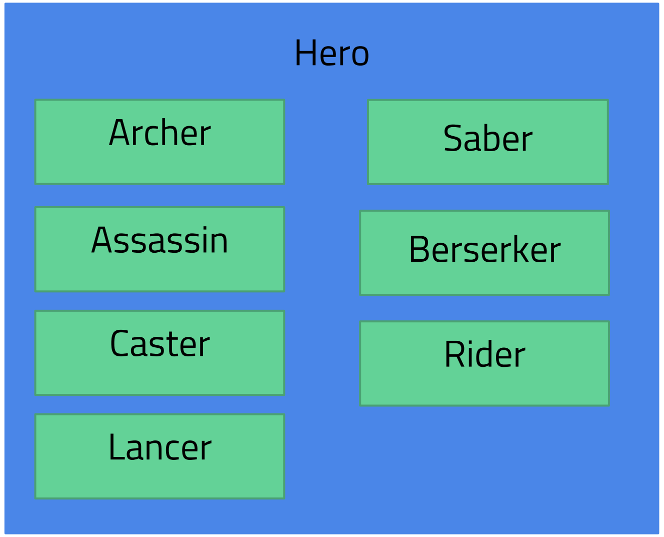
Assignment

Create a bank account class, give it fields like account balance, and interest, and give it behaviors like withdraw money or deposit money, make a constructor that can specify things like name, and their credit score.

Inheritance

What’s Inheritance?

Look at the diagram below:



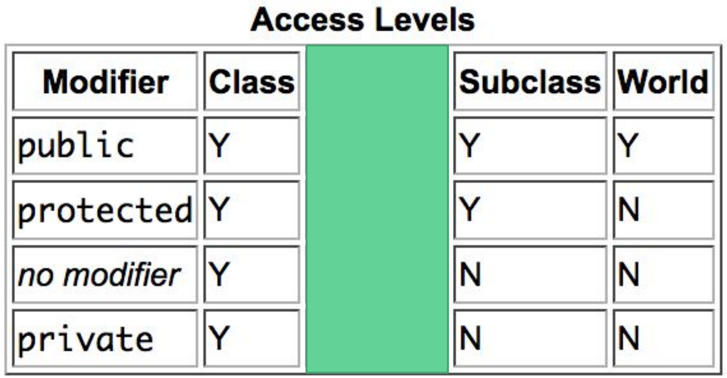
Inheritance is when one class is given access to all the fields and methods of another class. This is useful for when you have a base class that contains code that you would need to use repeatedly. For example, a hero would have the fields of Atk, Def, and Spd, and behaviors to attack and defend. Rather than completely recoding them for every iteration of a hero, we can inherit those methods so that every class can use them. In order to inherit from a class, you need to use the structure ClassA ***extends*** ClassB:

```
public class Archer extends Hero{
        private double range = 0.2;
        public Archer(int myAtk){
                Atk = myAtk;
        }

}
```

Visibility Modifiers

```
public void methodname() {
```

The visibility modifier determines which files would be able to access the method specified. The different modifiers are public, protected, none and private. Public makes it so that any other file can access it, and private makes it so that it cannot be accessed outside of the file. Protected and no modifier specified are in between, limiting the access to only a select group of files.



Access Levels

Modifier	Class		Subclass	World
`public`	Y		Y	Y
`protected`	Y		Y	N
no modifier	Y		N	N
`private`	Y		N	N

Parent and Child Classes

When a class inherits another class, the extended class becomes the parent, and the extending class becomes the child. The child class will contain all of the fields and behaviors of the parent class (as long as they're not private), but not vice versa.

```
public class Burgers {
     String breadAndMeat;
     //cannot access field delicious
}
public class ShakeShack extends Burgers {
     String delicious;
     //can access field breadAndMeat
}
```

```
public class InAndOut extends Burgers {
     //can access field breadAndMeat
     //cannot access field delicious
}

public class SchoolLunch extends ShakeShack{
     //can access field delicious
     //can access field breadAndMeat
}
```

Accessing Parent Methods (This slide is ***super*** cool!)

If you want to access parent class methods & variables, you can use the ***super*** keyword:

In Class Hero:

```
public class Hero{

..

public void a(int x){}

}
```

In Class Archer:

```
public class Archer extends Hero{
public Archer()
        public void a(int x){
                super.a(x+100);
        }
}
```

The a() in Archer is a modified version of the a() in Hero!

New Activity: create Animal, Dog, and Cat classes

Create an Animal class with fields age, size, and color, and a empty method called

makeNoise(String *g*), that prints out *g*;

Create a class dog and class cat with their own methods, bark() and meow(), by using super() for the makeNoise method.

Static Methods and Variables

Static Variables are fields in a class that doesn’t pertain to a single instance of a class.(Do some live coding)

```
public class Person{
        static String leader = “Kevin Cai”;
        /*In this case, you can access the string leader by every single person,
        and it isn’t required to even instantiate a person object. (Person.leader is ok) */
        //This is also the same for static methods. Note: static methods cannot access
        non-static data.
}
```

Static vs. Non Static methods and variables

```
public class Car{

        static int carsMade = 0;

        int ID;

        public Car(int myID){

                ID = myID;

                carsMade++;

        }/* the point is that non-static things like IDs belong to only 1 object, but static
things are unique by themselves.*/

}
```

```
public static void Main (String[] args){

        Car myCar = new Car(Cars.carsMade);

        System.out.println(Cars.carsMade);//out 1

        Car myCarTwo = new Car(Cars.carsMade);

        System.out.println(Cars.carsMade);//out 2

        println(myCar.ID);//out 1;
```

Static VS Non-Static (Cont.)

Initializing fields:	*Initializing fields:*

`Classname.fieldName`	`Classname object = new Classname(params)`
	`object.fieldName`

Creating fields:	*Creating fields:*
`static int fieldName`	`int fieldName`
Initializing methods:	*Initializing methods:*

`Classname.methodName()`	`Classname object = new Classname(params)`
	`object.fieldName`

Creating method:	*Creating method:*
`public static void methodName() {`	`public void methodName() {`
`Classname.nonStaticField //Error`	`object.staticField //no error`
Methods cannot access non-static fields	Methods can access static fields