Basic OOP

Stefan Zorcic

OOP or object oriented programming is a style of programming that structures data in objects as the name suggests. The main competitor to OOP is procedural programming, however object oriented programming has a plethora of advantages such as:

- Faster execution speed.
- A clear structure that can be easily understood.
- Less code needs to be written.

As the name suggests object oriented programming uses something called objects. Objects are data that can have a variety of uses as determined by the class. However what are classes? Classes are the blueprints that all objects are made from, if an object is a BMW i8 then the class is a car.

Classes distinguish themselves from each other through attributes and behaviours. Attributes are features that the object will have, for example: gasTank or horsepower. While behaviours are the abilities that the object will have, for example: driving and braking. An implementation is below.

```
public class Car{
    int horsepower;
    int gasTank;

    public void drive(){
        /* implementation not shown */
    }

    public void brake(){
        /* implementation not shown */
    }
}
```

Now that we have a Car class written we can make different car objects with a special method called a constructor. A constructor is the first method to be executed and must have the same name as the class.

*Note Java creates a default constructor however it is good practice to write your own

```
public Car(int horse, int gas){
    horsepower=horse;
    gasTank=gas;
}
```

Now that we have constructor we can finally create our an object with our class.

```
public static void main(String [] args){
   Car BMW = new Car(100,50);
}
```

With the code above we have created an object called BMW from the Car class. The BMW car has 100 horsepower and has 50 litres of gasoline. To check the attributes of the object we can do the syntax:

CLASSNAME.ATTRIBUTE;

For example to check the attributes of the class we can use the code below.

```
public static void main(String [] args){
   Car BMW = new Car(100,50);
   System.out.println("Horsepower: " + BMW.horsepower);
   System.out.println("Gas: " + BMW.gasTank + "L");
}
```

The output of the code segment is as follows:

```
Horsepower: 100
Gas: 50L
```

We can take this concept further to modify attribute(variable) values of the object, for example the code below changes the horsepower of the BMW to 200 and the gas to 20.

```
public static void main(String [] args){
   Car BMW = new Car(100,50);
   BMW.horsepower=200;
   BMW.gasTank=20;
   System.out.println("Horsepower: " + BMW.horsepower);
   System.out.println("Gas: " + BMW.gasTank + "L");
}
```

The output of the code segment is as follows:

```
Horsepower: 200
Gas: 20L
```

Seeing how simple it is to modify the attributes of an object there is a desperate need for something to protect the values. In Java this process of protect the attribute data is called encapsulation. Encapsulation works by adding access modifiers (public, private) to attributes in the class. The private access modifier restrictions access to variables by only allowing methods inside the class to access and modify the value.

```
public class Car{
   private int horsepower;
   int gasTank;

   public Car(int horse, int gas){
      horsepower=horse;
      gasTank=gas;
   }

   public void drive(){
      /* implementation not shown */
   }

   public void brake(){
      /* implementation not shown */
   }
}
```

Changing the class name and trying to change the value of horsepower again now resulting in an error due to the new access modifier.

Code:

```
public static void main(String [] args){
   Car BMW = new Car(100,50);
   BMW.horsepower=200;
   BMW.gasTank=20;
   System.out.println("Horsepower: " + BMW.horsepower);
   System.out.println("Gas: " + BMW.gasTank + "L");
}
```

Result:

```
driver.java:4: error: horsepower has private access in Car
      BMW.horsepower=200;
         ^
driver.java:6: error: horsepower has private access in Car
      System.out.println("Horsepower: " + BMW.horsepower);
                                              ^
```

Finally we need to write get and set methods to be able to now modify the values. However why go through the trouble if we are going to add get/set methods in the end? In actual practice these get/set method would have a lot of code to determine specific values that can be added or specific cases the value can be printed, however this tutorial will not show that as it is out of the scope of OOP.

```
public class Car{
   private int horsepower;
   int gasTank;

   public Car(int horse, int gas){
      horsepower=horse;
      gasTank=gas;
   }

   public int getHorsepower(){
      return horsepower;
   }

   public void setHorsepower(int val){
      horsepower=val;
   }
}
```

Now we can use the get/set method like any other behaviour of the class.

Code:

```
public static void main(String [] args){
   Car BMW = new Car(100,50);
   BMW.setHorsepower(200);
   BMW.gasTank=20;
   System.out.println("Horsepower: " + BMW.getHorsepower());
   System.out.println("Gas: " + BMW.gasTank + "L");
}
```

Output:

```
Horsepower: 200
Gas: 20L
```