Chapter 1 Encapsulation

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. Like how a capsule holds all its medication together. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.

To achieve encapsulation in Java −

a. Declare the variables of a class as private.
b. Provide public setter and getter methods to modify and view the variables values.

Why do we need this ?

Variables by their own nature are not intelligent, they are simple placeholders which will hold any value that you assign to them. If you let a class variable be accessed outside a class, any undesirable change can occur. So, variables must be always accompanied by their more intelligent elders of the family (methods – where you can write logic and perform operations safely), So

```
class Car{
 String brand;}
```

Becomes

```
class Car{
 private String brand;

public void getBrand(){
  return brand;
}

Public void setBrand(String b){
 brand=b;
}
}
```

Chapter 2 Composition

So far, in our course – the class members or member variables that we have used so far are one among the 4 basic data types – int, String, double, float, etc.

But in actuality – classes are built up of other classes. A single class is composed of other sub-classes. For example – A car class may be composed of Wheel class, Engine Class, Steering class etc.

Now, let us re-imagine a kitchen as a composition of other objects

```
class Oven{
  double temperature;
  String brand;
  int numberOfCookTops;

  public Oven(double t, String b, int n){
    temperature =t;
    brand=b;
    numberOfCookTops=n;
  }
}

class Dishwasher{
  String brand;
  int numberOfRacks;
}

public class Kitchen{
  int width, length;
  Oven oven1; //oven1 is the variable which will take actual value - Oven is just type of oven1
 // Dishwasher favotireD;

  public Kitchen(int w, int l, Oven o){
    width=w;
    length=l;
    oven1=o;
  }
}
```

Our kitchen is now composed of an Oven object, Dishwasher object and it's own length and width. Creating a composite object can be tricky at first. But, you can see below how to do this:

```
Oven o = new Oven (100.5, "Samsung", 5);
Kitchen myKitchen = new Kitchen(75,85, o);
```

Chapter 3 Static Variables

Let's assume we have a Condo class with kitchen, bedroom, oven as its members. Whenever we create an object of this Condo class, each object will have its own kitchen, bedroom and it's own oven.

Let's assume we are adding a swimming pool to the building – here even though all condo objects have access to the swimming pool – it does not belong to any specific member. It belongs to all the members or in other words it belongs to the Class. This is a good example for static variables.

Whenever a variable must be shared by all objects or when a variable must belong to a class – we use static variables.

```
class Condo{
 Kitchen kitchen;
 int numOfBedrooms;
 int aptNumber;
 static int swimmingPool;
}
```

So, the above case – pool object is shared by all Condos.

As the Swimmingpool belongs to the class, there's no necessity to create an object to access this variable.

To access a non-static or instance variable, you always need to create an object such as

Condo c1=new Condo();

c1.numOfBedrooms = 3;

To access a static variable, you can either use an object or you can just the use Class name and both will work the same

c1.swimmingPool =2;

Condo.swimmingPool = 3;

Both the above ways work, but accessing static variables with the class name is a preferred method because they belong to the class and not to any specific object.

Chapter 4 Static Methods

Static methods are usually used to work with static variables. Remember this:

a. Non-static methods can use both static as well as non-static variables. But be cautious while using static variables inside a non-static method
b. Static methods can use only static variables. Non-static variables are outside their scope as they belong to specific objects

```
Class Car{
private static int emergency = 911;

  public static void callEmergency(){
    System.out.println("Calling emergency at "+emergency);
  }
}
```

Static methods are used for writing Utility Methods which do not require object creation

```
public class Calculator{

  static double PI = 3.141414;

  public static void areaOfCircle(int r){
    double area = PI*r*r; // exercise with care
    System.out.println("Area of circle "+area);

  }

}
```

In the above scenario, to calculate area of a circle, one does not need to create a Calculator object, because there is no data that will belong to any specific instance. So it's better to have this method as static as it's just a helper method and no object of Calculator class may differ from one another.