Club Gen-Y PROGRAMMING DOMAIN Seniors- Mod 1

Date: 27-11-2016

To be submitted by 03-12-2016

PART-I

Inheritance, Abstract Classes and Interfaces

What are Classes & Objects – A Brief Intro.

A Blank Answer-Sheet.

That's a class. Bizarre right? Think about it. We take 8 copies of this questionnaire. Now there are 8 of you and each one fills one.

Everyone will have slightly different answer, and thus each answer-sheet will be now different.

The simple idea of **Class** is a that blank Answer-Sheet. It has no value. Just a structure to contain your answers. Once its filled it has value- hence a Useful **Object.** So we can call a **Class as an Object Factory- one which defines the shape and nature of the object.** Hence we also call *Class as a template of an object* or an *Object as an Instance of a Class*. This object can have a useful work to do which we call **Functions or Methods**.

How does a Class look like?

Something like this:

```
class Pokemon{
	String name;
	String type;
	int HP;
	int Attack;
	int Defence;
	String SplAtk;

	public Pokemon(String n, String t, int h, int a, int d, String sa){
		this.name = n;
		this.type = t;
		this.HP = h;
		this.Attack = a;
		this.Defence = d;
		this.SplAtk = sa;
	}

	public void doSpecialAttack(){
		System.out.println(name+" is Launching: "+SplAtk);
	}
}
```


And Creating an object for Class *Pokemon* is like:

```
Pokemon Pikachu = new Pokemon("Pikachu","Electric",400,200,300,"Iron Tail");
```

And now let's call the method of Pokemon- *doSpecialAttack( )*.

```
Pikachu.doSpecialAttack();
```

Gives us the output:

```
	Pikachu is Launching: Iron Tail
** new is a keyword used to create a "new" instance of a class.
```

We also used `Pokemon()` which is a Constructor. This is used to initialize every new object with new values. Basically you're a constructor when you're filling the Questionnaire!
Here we used a parameterised Constructor that means every time we create an object we have to initialize by passing values to this Constructor. We can overload Constructors like we can overload methods i.e. by passing different parameters. No parameters – Default Constructor.

We used another keyword – `this` which specifies in a method or Constructor that we're currently referring to the object being the caller now. In other words, it refers to the *Current Object*.
So calling Pikachu's name, we initialize it as `this.name = "Pikachu".`

TASK 1:

CREATE A CLASS MOVIE WITH SUITABLE ATTRIBUTES AND METHODS AND INITIALIZE TWO OBJECTS AND CALL THEIR METHODS IN THE MAIN().

Understanding Static

Static is a OOP concept and Java has static data, methods, blocks and classes too.

Sometimes we need to access certain resources of a Class which are common to all AKA Shared Resources. Something like keeping a count would require a variable which keeps a track how many objects are created. This count can be accessed by any object and it should not lose its previous value for any new object. So ***our class member is now independent or static***. A static member hence requires no reference. In Java, any method or variable can be static.

Something to REMEMBER!!

- All instances of a class share the same **static** variable.
- **Static** methods can only call other **static** methods.
- **Static** methods can only directly access **static** data members.
- **Static** methods cannot refer to **this** or **super** keywords at all.

Static Block??

A **static** Block is a block of code which is executed exactly once for any program when the class is first loaded. It is the first block to execute for any class. All our **static** variables can be initialized in this block.

The following Election and Voter classes will further clear the idea.

```
class Election {
        private static int partyA;  //static variables
        private static int partyB;

        static {                    //static block
                partyA = 0;
                partyB = 0;
        }
```

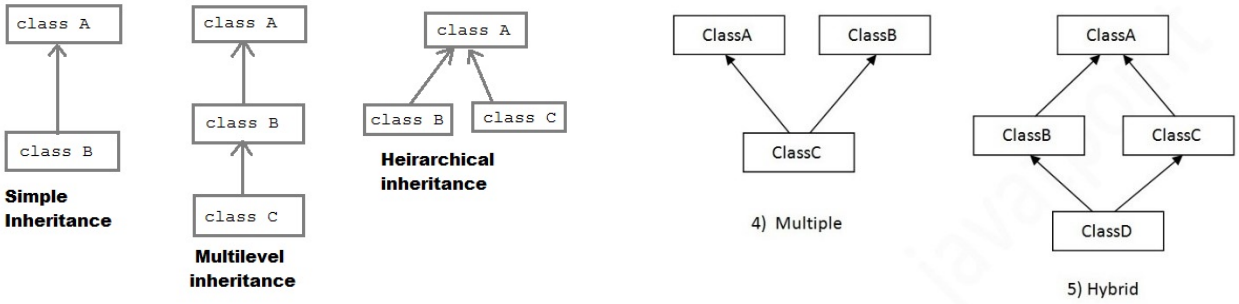
```
	public static void votePartyA(){ //static methods
		partyA++;
	}

	public static void votePartyB(){
		partyB++;
	}

	public void showResults(){
		System.out.println("Party A: "+partyA);
		System.out.println("Party B: "+partyB);
	}
}

public class Voter {
	String Name;
	int Age;
	String Voter_ID;
	static Election e = new Election();

	Voter(){
	}
	Voter(String n, int a, String vid){
		this.Name = n;
		this.Age = a;
		this.Voter_ID = vid;
	}

	public static void castVote(char p){
		if(p=='A')
			Election.votePartyA();
		else if(p=='B')
			Election.votePartyB();
		else
			System.out.println("Please Vote Fairly!!");
	}

	public static void main(String args[]){
		Voter A = new Voter("Aviral", 19, "V123435");
		Voter B = new Voter("Tanya", 21, "V124G21");
		Voter C = new Voter("Akash", 28, "V101244");
		A.castVote('A');
		B.castVote('B');
		C.castVote('A');
		e.showResults();
	}
}
```


As we can clearly see all instances of *Election* class share the static variables *partyA* and *partyB* and they can be only accessed by static methods and initialized by a static block. Also a static method only uses static objects, hence a static object being created.

Static classes can only be inner(nested) classes which are implemented as we implement data members. No outer class can be static. Refer: static class in Java

Inheritance – Story of parent & child

One of the most important concepts why we use OOPs concept is Inheritance. It tells about how we have a very general class with common characteristics for a set of other very specific classes. Such a general class is called a **Super-Class** and the classes which are specific but inherit the common features are called **Sub-Classes.** We also term them as *Parent and Child Classes.* As they resemble the behaviour of the same.



Java supports Simple, Multi-level and Hierarchical Inheritance. It partially supports Multiple and Hybrid Inheritance from Java 8; about which we will explore soon.

So how do we inherit? We actually extend from classes. One class can only extend from a single class which is *Single Inheritance*. However multiple classes can extend from a Class which is *Hierarchical Inheritance.*

Some points to remember:

- A **Subclass** can never access **private members** of a **Superclass.**
- We can always use a **Superclass object** to reference a **Subclass object**.
- A **Subclass** can override the **Superclass** constructor or methods provided no method is private (**Runtime Polymorphism**).

Using super keyword

Whenever a Subclass needs to refer to **its immediate Superclass,** it can do so by using the keyword super. Keyword super has two usages: First it is used as `super();` which means we want to access the Superclass constructor. Secondly we can use it to access the *data members* without creating a Superclass object like `super.member;` . It works like *this* keyword but especially for immediate Superclass.

To demonstrate these concepts have a look:

```
class Animal {
	String name;

	Animal(String n){
		name = n;
	}
	public void talk(){
		System.out.println(name+" can talk");
	}
```

```
	public void move() {
		System.out.println(name+" can move.");
	}
}

class Mammal extends Animal {
	Mammal(String n){
		super(n);
	}
	public void move(){ //method override
		System.out.println(name+" can walk or run");
	}
}

class Birds extends Animal {
	Birds(String n){
		super(n);
	}
	public void move(){ //method override
		System.out.println(name+" can fly");
	}
}

public class Dog extends Mammal{
	Dog(String n){
		super(n);
	}
	public void talk(){ //method override
		System.out.println(name+" can bark");
	}

	public static void main(String args[]){
		Animal a = new Animal("Cat");
		Animal b = new Birds("Eagle");
		Animal c = new Dog("Tommy");

		a.move();
		a.talk();

		b.move();

		c.move();
		c.talk();
	}
}
```


Each of the identical functions of the Superclass Animal will now produce different results for different Subclasses as they are inherited and overridden by them.

Output:
Cat can move.
Cat can talk
Eagle can fly
Tommy can walk or run
Tommy can bark

Dynamic Method Dispatch (DMD)

It is one of the most important concepts of Java which forms its basis at method overriding feature. It means the calls for a method is resolved at runtime rather than during compile-time. *Thus a method behaves based upon the type of object it is being referred to.* To show this concept;

```
class A {
	void callme(){
		System.out.println("I'm inside A");
	}
}

class B extends A {
	void callme(){		//overridden
		System.out.println("I'm inside B");
	}
}

class C extends A {
	void callme(){		//overridden
		System.out.println("I'm inside C");
	}
}

public class Dispatch {
	public static void main (String args[]){
		A a = new A(); //object of A
		B b = new B(); //object of B
		C c = new C(); //object of C

		A r; //reference object

		r = a;
		r.callme();

		r = b;
		r.callme();

		r = c;
		r.callme();

	}
}
```



```
Output:
I'm inside A
I'm inside B
I'm inside C
```

Method overriding has many practical applications. When you use a method in *Area* Class: findArea(), instead of overloading it by writing it different times in a superclass we can override it at its respective *Shape* Subclasses and produce desired results.

final keyword is used to declare constants in Java. It can be used also to prevent inheritance. If used with any method, we cannot override the method. If used with a class, we cannot extend it.

TASK 2:

USE THE CLASS MOVIE AND CREATE A DATA-MEMBER MOVIESWATCHED WHICH COUNTS A TOTAL NUMBER OF MOVIES WATCHED. CREATE SUB-CLASSES HORROR_MOVIE AND ROMCOM_MOVIE OVERRIDING A METHOD CALLED EXPERIENCE() OF MOVIE CLASS USING DMD.

Abstract Classes and Interfaces:

So as we observed in previous cases, there are situations when we don't need to define a method of superclass specifically which is to be overridden. Generally, the superclass has no role of using those methods. In such cases we can use a concept of ***Abstract Classes and methods.***

An Abstract class always as Abstract methods which are to be overridden mandatorily. For example `findArea()` method can be an Abstract Method of Abstract Class `Shapes` and it can be overridden in class `Triangle` which extends the class `Shapes`. All we need to do is declare the Abstract method in the Abstract class and Define the method in the Derived class.

```
abstract class Shapes {
	double dim1;
	double dim2;

	Shapes (double a, double b){
		dim1 = a;
		dim2 = b;
	}
	abstract double findArea(); //abstract method
}

class Rectangle extends Shapes {
	Rectangle (double a, double b){
		super(a,b);
	}
	double findArea() {
		return dim1 * dim2;
	}
}

class Triangle extends Shapes {
	Triangle (double a, double b){
		super(a,b);
	}

	double findArea(){
		return dim1 * dim2/2;
	}
}

public class ShapeArea{
	public static void main(String args[]){
		Rectangle r = new Rectangle(10,9);
		Triangle t = new Triangle(8,6);
		Shapes ref;

		ref = r;
		System.out.println("Area of Rectangle is: "+ref.findArea());
```

```
            ref = t;
            System.out.println("Area of Triangle is: "+ref.findArea());
      }
}
```

```
Output:
Area of Rectangle is: 90.0
Area of Triangle is: 24.0
```

Interfaces

Now that we know *Abstract Classes* it will be fairly easier to understand ***Interfaces*** and its use in Java. Using interfaces, we can fully abstract a class from its implementation. *This means All the methods of an interface is purely Abstract unlike Abstract classes where we can have Concrete methods (which are defined in class) as well. Also in an* ***Interface*** *all the* ***data-members are final*** *in nature.* We need to override all the methods of the interface when we implement them.

Interfaces are *implemented* by other classes. Here comes the catch, remember we discussed multiple inheritance is partially-possible in Java? Well it is done by implementing multiple interfaces in a single class. So basically while inheriting you can extend a class and implement multiple interfaces. Since interfaces are not the typical Classes; this scheme is partial.

How to define an Interface?

```
      interface Callback {
            void call (int param);
      }
      class Client implements Callback {
            public void call (int p) {
                  System.out.println("Client is calling: "+p);
            }
      }
```


The class which implements an interface can also partially implement it by not overriding all of its methods. Then this class must be made abstract. Just like Superclass objects we can also use interface objects as reference for its implementations' objects.

```
      public class Test {
            public static void main(String args[]){
                  Callback c = new Client();
                  c.call(19);
            }
      }
```

We can also implement Nested interfaces. Learn more from: Nested Interface
Example of an interface with data members:

```
import java.util.Random;
interface SharedConstants {
      int NO = 0;
      int YES = 1;
      int MAYBE = 2;
      int LATER = 3;
      int SOON = 4;
      int NEVER = 5;
}
```

```
class Question implements SharedConstants {
      Random rand = new Random();
      int ask() {
            int prob = (int) (100 * rand.nextDouble());
            if (prob < 30)
                  return NO; // 30%
            else if (prob < 60)
                  return YES; // 30%
            else if (prob < 75)
                  return LATER; // 15%
            else if (prob < 98)
                  return SOON; // 13%
            else
                  return NEVER; // 2%
      }
}
class AskMe implements SharedConstants {
   static void answer(int result) {
            switch(result) {
                  case NO:
                        System.out.println("No");
                        break;
                  case YES:
                        System.out.println("Yes");
                        break;
                  case MAYBE:
                        System.out.println("Maybe");
                        break;
                  case LATER:
                        System.out.println("Later");
                        break;
                  case SOON:
                        System.out.println("Soon");
                        break;
                  case NEVER:
                        System.out.println("Never");
                        break;
            }
      }
      public static void main(String args[]) {
            Question q = new Question();
            answer(q.ask());
            answer(q.ask());
            answer(q.ask());
            answer(q.ask());
      }
}
```


DID YOU KNOW?

Chef is an esoteric programming language in which programs look like cooking recipes. The variable names are usually named as common foods and stacks are called “mixing bowls” or “baking dishes”!