CS2

Theme:

Object Oriented

Programming

in

Java

Handout : Week10

prepared by N S kumar

Preface

Dear friends,

I am very sorry for the delay in making this week 10 lecture notes. Parkinson manifests in many ways. The work expands to fill the whole time. The expenditure raises to meet the income. So my excuse – my work expanded leaving me no time to make the notes – definitely not pardonable!

Good to know that there were many queries inquiring about the notes.

In this notes, we will discuss about interface, abstract class and Object class.

Look forward to your feedback.

N S Kumar
Visiting Professor
Dept. Of Computer Science and Engineering
PES Institutions.
Email : ask.kumaradhara@gmail.com
Date : 19th April 2015

Interface :

You would have heard of the story of the fox and the stork – each one hosting a feast to the other. The fox serves the soup on a flat plate. The stork serves the soup in a pitcher with a narrow deep opening. The stork does not get the right interface to enjoy its meal when fox serves it. The fox does not get the right interface when the stork serves it. The moral of the story (is this not very important whenever a story is told?) - interface matters the most.

(Advice : Please read Panchatantra stories, Akbar-Birbal stories, Tenali Ramakrishna stories, Aesop fables during the summer time – why not before and between the exams? - would help you in understanding Engineering better).

You may want to check these links.

https://www.google.co.in/search?client=ubuntu&channel=fs&q=panchatantra+fox+and+stork&ie=utf-8&oe=utf-8&gfe_rd=cr&ei=mBAyVc3IH6bv8weTyYG4Cg

or

http://wwriter.hubpages.com/hub/The-fox-and-the-stork

This is another incident my friend told me long time back. In the good old days, the brakes of the two wheelers were applied through a foot lever. The control was not in the hands. Kinetic Honda changed the design sometime in late 80s or early 90s. It seems that some person riding this vehicle was trying to stop the vehicle by applying the brake using his foot – failed – crushed into an oncoming truck and died.

Let us examine a few more examples before we discuss this concept in programming in general and Java in particular.

All Electrical switches appear similar. They have two states -on and off. What happens when we turn a switch on? It depends to what it is connected. It may turn a light on, start a water pump, make a robot dance or ring a bell. I am sure you can think of a number of other possibilities. What if we did not have this concept of a switch. May be, you had to pull a rope or hit at a place with a rod. This interface switch could be connected to any electrical device – each doing something different – but the life of the user becomes simple.

Most of you would know how to drive a car – how to use the three levers A, B, C with the feet. You would know what happens if you press the lever for acceleration and what happens if you press the lever for braking. How would this be in a special car where the position of A and B are interchanged?

By the way, would you know what happens when you press the brake lever? Yes. The vehicle slows down normally. How does it happen? You do not have to know. Internally the wheels might be slowed down by using some brake drum. Would that affect you if this mechanism is replaced by a hydraulic brake system? No. As long the interface does not change, even if the implementation is changed, the user is not affected.

Math.sqrt method takes a real number as argument and returns the square root of it as the result – to be precise – the argument should be greater than or equal to 0 and this function returns the positive root. Would you know how the square root is implemented? Does it matter for you if the algorithm to find the square root is changed by the java library?

What if the Java developers get an interesting idea to make finding the root faster? What if they change the interface of invoking the method? What if you are asked to give two arguments to this method sort – one is the real number whose root you are interested in and the other a possible guess for the root? I am sure you would not like it. All your earlier programs would go for a toss!

Interface is divine. Interface is sacred. It is a protocol. Once defined, it should not be withdrawn. Once defined. It should never be changed. If the interface is changed, the universe will collapse. That is a doomsday.

In programming, a method signature is an interface.
We know what happens when we call Arrays.sort(anArray).
We do not know how it happens
We do not care for how as long as it works.
We enjoy dosa at VidyarthiBhavan – we do not know how they make it – may be it is better we do not know how they make it!

Can we extend this concept to classes?
Can we have a class which commits **what** it will do for the user,
but does **know** how. Somebody should support the "how" of this.

We have already observed this concept while understanding the concept of inheritance. When we use a subclass object, we can be sure that it will support the interface of the superclass- but these could be supported differently by the subclass objects. This superclass object can also provide a default implementation.

What if we do not want to commit even for the default implementation?

Then we use the concept of interface - an amazing concept in programming - ably supported by Java.

An interface in Java says what the user of a class supporting this interface can do. It does not say how these will be implemented by the class supporting this interface. **It says “what” and “not how”.**

It is possible that there could be a number of implementations for the same interface. The client can chose whatever he prefers – can also change from one to another. If you want to eat dosa, you can choose any eatery which supports the interface – Dosable – which makes dosa.

It is equally possible, there could be number of clients using an implementation.

Syllabus of OOPJ is an interface. The teachers implement this interface. You are the clients. But unfortunately, in this case, you can not chose the implementation! The students of H section are tied to my implementation!
In our department, we have already experimented with students choosing the implementation – choose which teacher to trouble!
A day may come when you can choose a different teacher for each topic!

An interface in Java specifies the method signatures and has no default implementation. So, these methods are abstract and also public – are an interface to everybody in the world.

```
// L37/Displayable
public interface Displayable
{
        void disp();
}
```

Here is a concrete class – not abstract, can be instantiated – which says that it will support the interface called Displayable. An interface can have any number of methods. This class implementing the interface should override every method of the interface to become concrete. Otherwise this class also becomes abstract.

```
// L37/Rect.java
public class Rect implements Displayable
{
        private double length;
        private double breadth;
        public Rect(double l, double b)
```

```
	{
		length = l; breadth = b;
	}

	// wrong signature; every method in the interface class
	//	is abstract and public;
	// use @Override to check
	//void disp()

	public void disp()
	{
		System.out.println("length : " + length);
		System.out.println("breadth : " + breadth);
	}
}
```

Here, we have one more concrete class.

//L37/Circle.java

```
public class Circle implements Displayable
{
	private double radius;
	public Circle(double r)
	{
		radius = r;
	}
	// wrong signature; every method in the interface class
	//	is abstract and public;
	// use @Override to check
	//void disp()
	public void disp() {
		System.out.println("radius : " + radius);
	}
}
```

This client creates objects of classes Rect and Circle which in turn implement Displayable and therefore are displayable! We can call methods requiring a Displayable interface and pass objects of Rect or Circle. Conversion of object of classes implementing an interface to an interface reference is trivial. It is similar to conversion of subclass objects to superclass objects.

//L37/Example1.java

```
public class Example1
{
        public static void main(String[] args)
        {
                Rect r = new Rect(20, 10);
                Circle c = new Circle(7);
                foo(r);
                foo(c);
        }
        public static void foo(Displayable d)
        {
                d.disp();
        }
}
```

We should be able to distinguish an interface, an abstract class and a concrete class clearly. We shall now try to answer a few of these questions.

- Can we instantiate an interface directly?
    - NO. you can not have a constructor. There is no default constructor. You can not make one either. It does not commit to any layout at all.
- can we have data members in an interface?
    - We can. But these will for the whole class and will be immutable. In Java terminology, these will be static and final. So, the client of the class has a guarantee about these members. They exist in every class implementing

the interface, can be accessed through the class or the object – no difference though – will never change – as constant as North Star.

- Can we specify that the method of an interface is private?
  - This if allowed is very bad. Definitely NO.
- Can we specify that the method of an interface is protected?
  - Interface should remain an interface for everybody in the world. The answer is a clear NO.
- Can an interface extend an interface?
  - Definitely YES. Dosable may also support Vadeable – together may make a Comboable interface!
- Can a class implement more than one interface?
  - No issue as there are no mutable members in the interface.
- Can a class override a few methods of the interface which it implements?
  - Then the class remains abstract – therefore cannot be instantiated.

Observe **the right and the wrong** statements in the following programs

```
//L37/Example2.java
// A few questions to discuss:
// can we instantiate an interface?
// can we have data members in an interface?
// can we specify that the method of an interface is private?
// can we specify that the method of an interface is protected?
// can an interface extend an interface?
// can a class implement more than one interface?
//              would there be a class if both the interfaces have method
//                      with the same name?
// can a class override a few methods of the interface
//              which it implements?
```

```
public class Example2
{
        public static void main(String[] args)
        {
                //Example2.java:16: error: IWhat is abstract; cannot be instantiated
                //IWhat what = new IWhat();
        }
}

interface IWhat
{
        void foo();
        //Example2.java:26: error: = expected
        //int x; // final : has to be initialized
        int y = 100; // ok; what does this mean? Static final
        //protected void foo1(); // Error; modifier not allowed
}

class CWhat implements IWhat
{
        public void foo() { System.out.println("y : " + y); }
        public void bar(int y)
        {
                //Example2.java:37: error: cannot assign a value to final variable y
                //this.y = y; // Error
        }
}
```

```
interface ITest
{
        void f1();
        void f2();
}

// class has to be abstract as f2 is not implemented!

abstract class CTest implements ITest
{
        public void f1() { }
}

interface I1
{
        void f1(int i);
}

interface I2
{
        void f1(double d);
}

// Error: should also override f1 of I1
/*
class C implements I1, I2
{
        public void f1(double d)
        {
        }
}
*/
```

One last time, we will summarize the characteristics of an interface.

interface :

cannot be instantiated
is like an abstract class (with no default implementation)
all methods are public
all data members are static final

Let us highlight the similarities and the difference between an abstract class and an interface.

Abstract class & Interface:

- both cannot be instantiated
- both commit to commonality of classes extending the abstract class or implementing the interface

Abstract class:

- also provides implementation reuse – provides default implementation
- therefore can have instance fields and instance methods
  - both abstract and concrete

Interface:

- provides an interface reuse - provides no implementation
- methods are all public and abstract
- data members are static and final - can never be changed
- interface provides an invariant about the class implementing it.

A class can extend only one abstract class - as Java does not support multiple inheritance. But a class can have implementations of any number of interfaces.

From the client point of view, the user can use either of these without worrying about the subclass supporting the abstract class or the interface.

Here is an important piece of advice.

"The client should program to the interface and not the implementation"
- Gamma et al

This is an interesting question.

Can a class with all methods implemented also be abstract?

It can be. If creating an object of that class does not make sense in the domain of application, the class can be made abstract.

In the example below which is incomplete, I have illustrated this concept. A bank may support a number of types of Accounts. The commonality is captured in the super class Account. This class would have instances like name of the account holder, account number and so on. To initialize these members, constructor is required. We may also require methods to display and/or modify these fields. But we will never be required to make an object of this class Account directly. We may create objects of SB Account, Current Account and so on – but never an object of Account directly. To avoid the client instantiating this class directly, we may want to make this class abstract even though it may have no abstract methods.

```
abstract class Account
{
	private String name;
	private String acno;
	// there are implementations; but class should not
	//	be instantiable; so make the class abstract
	public void getName()
	{
		// some code
	}
}

        public void setName(String name)
        {
                // some code
        }
        // This may or may not be the reason why this class is abstract!
        abstract public double findInterest();
}

class SBA extends Account
{
        private double minBalance;
        private double rate;
        public double findInterest()
        {
                double res = 0.0;
                // some code
                return res;
        }
}
class FDA extends Account
{
        private double principle;
        private int term;
        private double rate;
        public double findInterest()
        {
                double res = 0.0;
                // some code
                return res;
        }
}
```

A Subhashita (a crude English translation would mean - Well Said) says “as all the rain water flows to the Sea, all the prostrations to different Gods reach one GOD. Basically, this says that there is one and only GOD. (Do not ask me any questions on this – I am agnostic – tending more towards atheistism).
Can we some similar idea in Object Oriented Programming? Can we capture what is all common to all classes and create a super super class from which all classes descend? Or can we create a superclass which provides some default implementation to some interfaces which make sense for every class we create?

This concept is supported in a few Object Oriented Languages – not all. Java does this support point of view. This feature is called **single rooted hierarchy.** All classes in java indirectly extend a super class called Object. Let us examine a few of these interfaces supported by the Object class in the next few examples.

toString Method: Can we pass an object as argument to the println method? This method is overloaded for primitive types. It is also overloaded for Object type. Whenever we pass an object reference, the latter method would be invoked. Now, the question is – how can the println method display an object of user defined class?
This overload of println will try to convert the object reference to a string by calling the method toString on the object. The toString method of the Object class provides a default implementation – outputs the class name and some number. The user defined class can override the method to provide a meaningful conversion of the object to a string.

L39

```
//L39/Example1.java
//	toString
public class Example1
{
	public static void main(String[] args)
	{
		Rect r = new Rect(20, 10);
		Object o = r;
		System.out.println(o);
		System.out.println(r);
	}

}

class Rect // extends Object // implied; can be stated!
{
	private double length;
	private double breadth;
	public Rect(double l, double b)
	{
		length = l; breadth = b;
	}
	// experiment  with or without toString method
	@Override
	public String toString()
	{
		// return "MyRect";
		return length + " : " + breadth + "\n";
	}
}
```

equals method and hashCode method:

Observe the example below. Both r1 and r2 are object references which refer to the Rect object on the heap. But r3 is a different object of the same class with the same values of the attributes. Now, we have to answer the following questions.

Are r1 and r2 same?

Are r1 and r3 same?

r1 == r2 is trivially true as both refer to the same location.

But r1 == r3 becomes false as r1 and r3 do not refer to the same location.

We talk about two different types of equality

- reference semantics

both the objects have the same reference – as in the case of r1 and r2

- value semantics

both the objects may have different references – as in the case of r1 and r3 – but in the eyes of the client, both are same

- clearly it depends on the abstraction of the client
- computer will have to be told what is the meaning of equality in this case

Please observe that reference semantics also implies value semantics.

Java provides a method called equals in the cosmic class Object which by default acts like ==, but the class developer can override to provide a meaningful equality comparison for his class.

But, there is another method which converts a given object to an integer. There are many data structures which store an object at a particular slot based on this value. This concept is called hashing and the method is java is called hashCode.

It is expected that the hashCode will be same if two objects are same.

As java does not know anything about the meaning of an object of our class, hashCode is implemented using reference semantics. If two objects refer to the same object in memory, they will have the same hash code. But if our of idea of equality is different – is based on value semantics – then we should also modify the hashCode and therefore should override this method in our class. If two objects are same according to our logic, then both should emit the same hash code.

```
//L39/Example2.java
// discussion of   Object class
//       equals
//       hashCode
```

There are a few interesting aspects about the method equals.
Let us examine its signature.

```
        @Override
        public boolean equals(Object o)
```

In this example, our idea is to compare two rectangles. We invoke this method with a rectangle. We also pass a rectangle as an argument. Why is it that the parameter is of Object class?
The signature of equals method comes from the Object class. There is no way the Object class could dream of Rect class when its signature was fixed. Any way, converting a Rect to an Object is trivial. The implementor would know which object to expect and therefore can downcast and use the object correctly. Observe the cases considered in the implementation.

- Is the object null? You may wonder why we do not do this on the special reference this.
- Is the object of this Rect class?
- If yes, then downcast and compare.

```
public class Example2
{
        public static void main(String[] args)
        {
                Rect r1 = new Rect(20, 10);
                Rect r2 = r1;
                Rect r3 = new Rect(20, 10);
                Rect r4 = new Rect(30, 15);
                // reference semantics
                System.out.println("r1 == r2 : " + (r1 == r2)); // true
                System.out.println("r1 == r3 : " + (r1 == r3)); // false
                System.out.println("r1 == r4 : " + (r1 == r4)); // false
                // value semantics
                System.out.println("r1 equals r2 : " +
                        r1.equals(r2)); // true
                System.out.println("r1 equals r3 : " +
                        r1.equals(r3)); // true
                System.out.println("r1 equals r4 : " +
                        r1.equals(r4)); // false

                // hashCode should support value semantics
                System.out.println("hashCode r1 : " + r1.hashCode());
                System.out.println("hashCode r2 : " + r2.hashCode());
                System.out.println("hashCode r3 : " + r3.hashCode());
                System.out.println("hashCode r4 : " + r4.hashCode());
        }
}
// A few points to think:
// will this work if we have subclass object of Rect?
// should this work if we pass  object of some other class
//      which also has length and breadth?
```

```
class Rect // extends Object
{
        private double length;
        private double breadth;
        public Rect(double l, double b)
        {
                length = l; breadth = b;

        }
        // understand the signature; why is the parameter
        //      not Rect
        @Override
        public boolean equals(Object o)
        {
                boolean res = false;
                if(o == null || !(o instanceof Rect) )
                {
                        res = false;
                }
                else
                {
                        Rect that = (Rect)o;
                        res = this.length == that.length &&
                                this.breadth == that.breadth;
                }
                return res;
        }
```

```
	// whenever we override the equals method, we should
	//	override the hashCode
	@Override
	public int hashCode()
	{
		// some expression based on length and breadth
		return (int)(length * 100 + breadth);
	}
}
```

clone Method:

There are a few things which are difficult for us to make. You may give a treat to your friends (hopefully you will include me!) - you may make the usual stuffs like chapathi, rice and so on - but bring the sweets from outside. You may not know how to make it or may be difficult to do as well as the experts do.

The question here is how to make a copy of an object. If we copy one reference to another, copy is not made – instead both the references refer to the same object. Copying would require the knowledge how the objects are stored internally – would they have extra members for the reasons of implementation? - would they have made optimization – in having less space or less time to access?

So it is better the job of copying to the expert – in our case – jvm. We would tell jvm we want to copy our object. Let us understand how we should go about doing it.

1. There is a method called clone in the Object class – which is protected. The client cannot use unless it is made public. So, override the method in the subclass and promote the access to public. Make this an interface for the normal client.
2. Tell the jvm that we want to use it. Our class should implement the Cloneable interface. This is a funny interface as it does not have any method in it. The method clone comes from the class Object. Such an

interface used to inform the jvm to provide the required support is called a marker interface.

3. In the clone implementation, make the Object class clone the object.
4. **If the class contains embedded objects, each has to be explicitly cloned as the default clone does copying at one level only.**

```
//L39/Example3.java
// discussion of the  Object class
//	clone
//	we discuss the concept of shallow and deep copy
//				concept of marker interface(not the term used
//				in java documentation)
//	clone is a method of the object class and not
//	of cloneable interface

public class Example3
{
	public static void main(String[] args)
	{
		Rect r1 = new Rect(20, 10);
		Rect r2 = r1;
		Rect r3 = r1.clone();
		System.out.println(r1);
		System.out.println(r2);
		System.out.println(r3);
		r1.setLength(30);
		System.out.println(r1);
		System.out.println(r2);
		System.out.println(r3);
	}
}
```

```
class Rect implements Cloneable
{
        private double length;
        private double breadth;
        public Rect(double l, double b)
        {
                length = l; breadth = b;
        }
        @Override
        public String toString()
        {
                return length + " : " + breadth + "\n";
        }
        public void setLength(double l)
        {
                length = l;
        }
        @Override
        public Rect clone()
        {
                Rect that = null;
                try
                {
                        that = (Rect)super.clone();
                }
                catch(CloneNotSupportedException e)
                {
                        System.out.println(e);
                }
                return that;
        }
}
```

The method clone could fail – so we have to handle the exception CloneNotSupported.

Observe one interesting aspect in the signature of this method. The return type is not Object and is our class. This is something the Object class would never have known. This is allowed in Object Oriented languages in recent times. If the method of the superclass returns a superclass object, the method of the subclass overriding the method of the superclass can return a subclass object.
This is called c**o-variant return type.**

Wrong Designs:

When something can go wrong it will. Murphy manifests in many ways. We learn many things in life based on our mistakes. We can learn from mistakes of others too. Here are some examples regarding how we may go wrong in our class design, Let us observe which cardinal rule is broken by these designs.

1. **A subclass should support every interface of the superclass.**

Inheritance should indicate the relation of "IS-A" between classes.
The room is not a fan. Cannot make a room start-rotating – unless ...

//L40/Example1.java
// Wrong design

// subclass cannot support every method of the superclass
// this is a clearly a case of composition and not inheritance

```
public class Example1
{
	public static void main(String[] args
	{
		Room r = new Room();
		r.startRotation(); // does room rotate?
	}
}

class Fan
{
	public void startRotation()
	{
		System.out.println("starts rotating");
	}
}

class Room extends Fan
{
}
```

2. **subclass objects get a single unnamed object of the superclass.**

If a class requires more than one object, most probably inheritance is a bad idea.

```
//L40/Example2.java
// extending provides a single object of superclass in subclass
// what if the subclass requires more than one object?
// if the subclass has more than one object, interface of one
//	can be accessed directly. Does the reflect the actual condition?



public class Example2
```

```
{

	public static void main(String[] args)
	{
		// line has two points
		Line l = new Line(3, 4, 5, 6);
		l.disp(); // which point?
	}
}

class Point
{
	private int x;
	private int y;
	public Point(int x, int y) { this.x = x; this.y = y; }
	public void disp()
	{
		System.out.println("x : " + x);
		System.out.println("y : " + y);
	}
}

// one point by embedding; cannot be directly accessed
// one point by extending! can be accessed
// In this case, the line has two points – one of the superclass and the other by
 // composition. The client can access the former directly and not the later!!!



class Line extends Point
```

```
{
	private Point other;
	public Line(int x1, int y1, int x2, int y2)
	{
		super(x1, y1);
		other = new Point(x2, y2);

	}
}
```

3. The rule of programming : "prefer composition over inheritance"

In this case, the room has a Fan – it could be any subclass of Fan.

This association can be fixed at runtime.

```
//L40/Example3.java
// limitations of inheritance:
//	- only one object of superclass
//	- every method of superclass should be supported by subclass
//	- binding of subclass to superclass at compile time
// room has fans;
// each fan can be of different type
public class Example3
{
	public static void main(String[] args)
	{
		Room r1 = new Room(new Fan());
		Room r2 = new Room(new OrnamentalFan());
		Room r3 = new Room(new ThreeBladeFan());
		Room r4 = new Room(new FourBladeFan());
		r1.rotateFan();
		r2.rotateFan();
		r3.rotateFan();
		r4.rotateFan();
	}
}

class Fan
{
	public void startRotation()
	{
		System.out.println("starts rotating");
	}
}

class ThreeBladeFan extends Fan
{
	public void startRotation()
	{
		System.out.println("starts rotating 3 blade fan");
	}
}

class FourBladeFan extends Fan
{
	public void startRotation()
	{
		System.out.println("starts rotating 4 blade fan");
	}
}





class OrnamentalFan extends Fan
```

```
{
        public void startRotation()
        {
                System.out.println("starts rotating ornamental fan");
        }
}

// Room has a Fan
// you can also support multiple fans
class Room
{

        private Fan fan; // this can be anytype of fan; no committment here
        // has other things like light, clock ...; not shown!
        public Room(Fan fan)
        {
                this.fan = fan;
        }

        public void rotateFan()
        {
                fan.startRotation();
        }
}
```

4. **Use inheritance judiciously. When in doubt, prefer composition over inheritance.**

A line preferably should have both points by composition unless one of the points is special.

```
//L40/Example4.java
```

```
// line and points

public class Example4
{
	public static void main(String[] args)
	{
		// line has two points
		Line l = new Line(3, 4, 5, 6);
		l.dispOne();
		l.dispTwo();
	}
}

class Point
{
	private int x;
	private int y;

	public Point(int x, int y) { this.x = x; this.y = y; }
	public void disp()
	{
		System.out.println("x : " + x);
		System.out.println("y : " + y);
	}
}
```

```
// both points by embedding
```

```
class Line
{
        private Point one;
        private Point two; // A line has two points
        public Line(int x1, int y1, int x2, int y2)
        {
                one = new Point(x1, y1);
                two = new Point(x2, y2);
        }

        public void dispOne()
        {
                one.disp();
        }

        public void dispTwo()
        {
                two.disp();
        }
}
```

We require interface, abstract class, concrete class, inheritance and composition. We should learn to know what to use, when to use and how to use. Believe me, we require time and effort to master many of these concepts. But, I am sure you can start learning from this very moment.