Notes on Java Programming

Java Interfaces

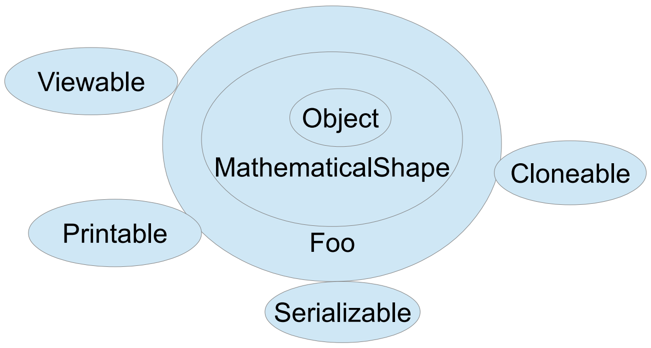
IST JAV 2012

I) Interfaces description

Whereas inheritance indicates what is an object, interfaces are used to capture additional capabilities to an object. For instance if I write the following class,

```
public class Foo extends MathematicalShape
      implements Cloneable, Serializable, Printable, Viewable {
}
```

An instance of this class will have the following possible representation.



The class Foo embeds the behavior of MathematicalShape and Object classes and provides specific abilities on View, Print... These additional abilities are sometimes called additional behaviors which are out of the main goal of the class. The interface mechanism is a way of expressing those abilities independently from the class mechanism.

An interface is a structure that only declares concerns without implementing them. For instance a concerns that describes something that can be cooked should be captured in a Cookable interface. This interface may be expressed like this.

```
package test;
public interface Cookable {
  void describeIngredients();
  void describeRecepies();
  void startCooking();
}
```

"It is permitted, but discouraged as a matter of style, to redundantly specify the public and/or abstract modifier for a method declared in an interface."
http://docs.oracle.com/javase/specs/jls/se7/html/jls-9.html#jls-9.4

II) Interface implementation

We understand that a class that implements this interface should represent something that is Cookable and that must answer to three questions through method invocations. A class that implements an interface must either define the body of each method, or having some of its parent class defining them, or being abstract.

The following three collection of code illustrate this.

```
public class Chicken implements Cookable {
  public void describeIngredients() {
      System.out.println(“Hello”);
  }
  public void describeRecepies() {
    // Do nothing
  }
  public void startCooking() {
     // Do manythings
  }
}
```

```
public abstract class Vegetables implements Cookable {
  public void describeRecepies () {
     // I don't like vegetables
  }
}
```

```
public class KobeBeef extends Beef implements Cookable {
  // Every thing else is defined in the Beef class
  public void startCooking() {
    System.out.println(“Do we really need to cook the beef of kobe ?”);
  }
}
```

III) Interface usage

The interface mechanism is a powerful typing mechanism that enable a clean separation of concerns between independent concept that may be grouped together within a class. They are much like alternative facets of the same object. The *cast* and *instanceof* operators enable to understand what is really our instance.

Some examples :

```
// Direct Object

Object  o  =  new  Object();  //  The  left  and  right  part  of  the  =  sign  are  of  the  same
                         // types.

Object   tmp   =   (Object)   o;   //   We   can   cast   reference   of   some   type   to   another.
                                //    Two    'names'    /    'references'    points    to    the    same
                         // object of class Object

System.out.println(tmp instanceof Object); // tmp and o are instanceof Object class

System.out.println(tmp.toString()); // toString is a direct method of Object class


// Hierarchy object

Object o = new KobeBeef(); // The right part is a child type of the left part. No cast is mandatory
                                          //  But  I  cannot  invoke  any  method  of  the  KobeBeef  type.
                            // Since the o ref is of class Object I can only invoke method from the
                           // class Object. o.toString() is valid. o.startCooking() does not compile

KobeBeef kb = (KobeBeef) o; // Now I have two refs on the KobeBeef instance. One of the kind Object and
                             // one of the kind KobeBeef. Since the left part is more specific than the
                            // right part I had to explicitly cast the reference into the desired type

System.out.println(o                   instanceof                   Object);                   //true
System.out.println(o                  instanceof                  KobeBeef);                  //true
System.out.println(kb                  instanceof                  KobeBeef);                  //true
System.out.println(kb                   instanceof                   Object);                   //true


// Interface Object

Cookable c = new KobeBeef(); // The class is a KobeBeef, but I only want to see it as a Cookable thing

System.out.println(c                   instanceof                   Object);                   //true
System.out.println(c                  instanceof                  KobeBeef);                  //true
System.out.println(c instanceof Cookable); //true
```

A final remark for the moment; *instanceof* and *cast* may be capture with a try/catch code block that captures ClassCastException. The next two codes are functionally equivalent.

```
public void areYouARealBeefOfKobeWithException (Object o) {
  KobeBeef kb = null;
  try {
    kb = (KobeBeef)o;
  } catch(ClassCastException e) {
    System.out.println("Humm "+o+ "seems to be a fake");
  }
}
```

```
function areYouARealBeefOfKobeWithInstanceOf (Object o) {
  KobeBeef kb = null;
  if !(o instanceof KobeBeef ) {
    System.out.println(“Humm “+o+ “seems to be a fake”);
  } else {
    kb = (KobeBeef)o;
  }
}
```

Although these two codes are semantically equivalent, the CastException approach cost much more in Cpu cycle than the instanceof operator. In some cases it may influence your program. The instanceof may also have performance issue when the inheritance graph becomes too complex, typically when reaching 8 levels of depth. http://dl.acm.org/citation.cfm?id=583821

III) Next week exercice

The following code provides a simplified ClassRoom management system.

```
package cr;
public class Student {
  private String name;
  public Student(String name) {
    this.name = name;
  }
}

package cr;
public class ClassRoom {
  public static void main(String[] arg) {
    Student [] students = new Student [10];
    for (int i=0; i<15; i++) {
      students[i] = new Student(String.valueOf(i));
    }
  }
}
```

This code compiles, but exception rises when the student array is filled with new Students. The array [ ] operator is limited to ten entries.

1) Design a class, called Bundle, that must store an undefined number of Students. The class has an internal array structure.
2) Design a second class called Bundle2 that must store an undefined number of any kind of Object instances. The class is designed as a linked List

   http://en.wikipedia.org/wiki/Linked_list#Singly_linked_list.
3) Design a Crowd interface that gather methods from the Bundle and Bundle2 classes, and use it in the ClassRoom example.

You must provide three different projects that contains the various classes for each questions.

Project 1 : Student.java, ClassRoom.java, Bundle.java

Project 2 : Student.java, ClassRoom.java, Bundle2.java

Project 3 : Student.java, ClassRoom.java, Bundle.java, Bundle2.java, Crowd.java

This exercise will be evaluated.