INHERITANCE & POLYMORPHISM LAB 3

1. Objectives:

- Understanding and using the Inheritance mechanism of Java (extends keyword)
  - Inheritance
  - Method Overriding
  - superclass
- Viewing the main method of base class:
  - getClass**()**
  - toString()
- Exploring polymorphic behaviors in Java applications
- Defining and using abstract classes and interfaces
- Review instanceof java keyword

2. Inheritance:

2.1. “java.lang.Object”: the root of the class hierarchy

- Inheritance is one of the key concepts of object-oriented design.
- Inheritance allows a class (called “child class” or “derived class” or “sub-class”) to use the attributes and methods of another pre-existing class (called “base class” or “parent class” or “super class”)
- The derived class is adding its own properties and operations. Note that a derived class can only inherit from one single base class.
- By default, all Java classes inherit from the java.lang.Object class. Therefore, all objects (including arrays) implement the methods of this class, listed in below table (reference: http://docs.oracle.com/javase/7/docs/api/)

Modifier and Type	Method and Description
protected Object	clone() Creates and returns a copy of this object.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this one.
protected void	finalize() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	getClass() Returns the runtime class of this Object.
Int	hashCode() Returns a hash code value for the object.
void	notify() Wakes up a single thread that is waiting on this object's monitor.
void	notifyAll() Wakes up all threads that are waiting on this object's monitor.
String	toString() Returns a string representation of the object.

void	**wait**() Causes the current thread to wait until another thread invokes the **notify()** method or the **notifyAll()** method for this object.
void	**wait**(long timeout) Causes the current thread to wait until either another thread invokes the **notify()** method or the **notifyAll()** method for this object, or a specified amount of time has elapsed.
void	**wait**(long timeout, int nanos) Causes the current thread to wait until another thread invokes the **notify()** method or the **notifyAll()** method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

3. Lab Practice:

- Create a new Java project with Eclipse
- Add a new class “Date” with:
  - 3 private attributes: day, month, year of integer type.
  - A constructor with 3 arguments used to initialize the class instance attributes. At this time, the constructor does not check whether input arguments are valid or not.
- Add a TestDate main program. Fill the main method as proposed below:

```
public static void main(String[] args) {

                Date date1 = new Date (5, 2, 2016);
                System.out.println("getClass date1 ->
"+date1.getClass());    }
```

- What’s the result of this program?
- The Date class does not define any getClass() method, so how the “date1” instance can perform the getClass() operation?
- Modify the main method adding the following code:

```
Constructor c[] = date1.getClass().getConstructors();
      for(int i = 0; i < c.length; i++) {
         System.out.println(c[i]);
      }
```

- You should get an error, use eclipse proposal to fix it.
- What’s the result of this program?
- Modify the main method by creating a date2 object as follow:

```
public static void main(String[] args) {
      Date date2 = new Date (5, 2, 2016);
      System.out.println("date2 -> "+date2);
      System.out.println(date2.getClass());
}
```

- What’s the result of this program?
- What would be a more user-friendly output?

- When printing an object using System.out.println(object), the **toString()** method of this object is actually invoked.
- Does the Date class implement a **toString**() method?
- Which implementation is used then?

Sub-classes have the ability to change the implementation of methods inherited from the superclass. The objective is to adapt the inherited behavior to the specificities of a derived class. This concept is called Method Overriding.

Overriding is a way to specialize inherited methods, not to change the semantics.

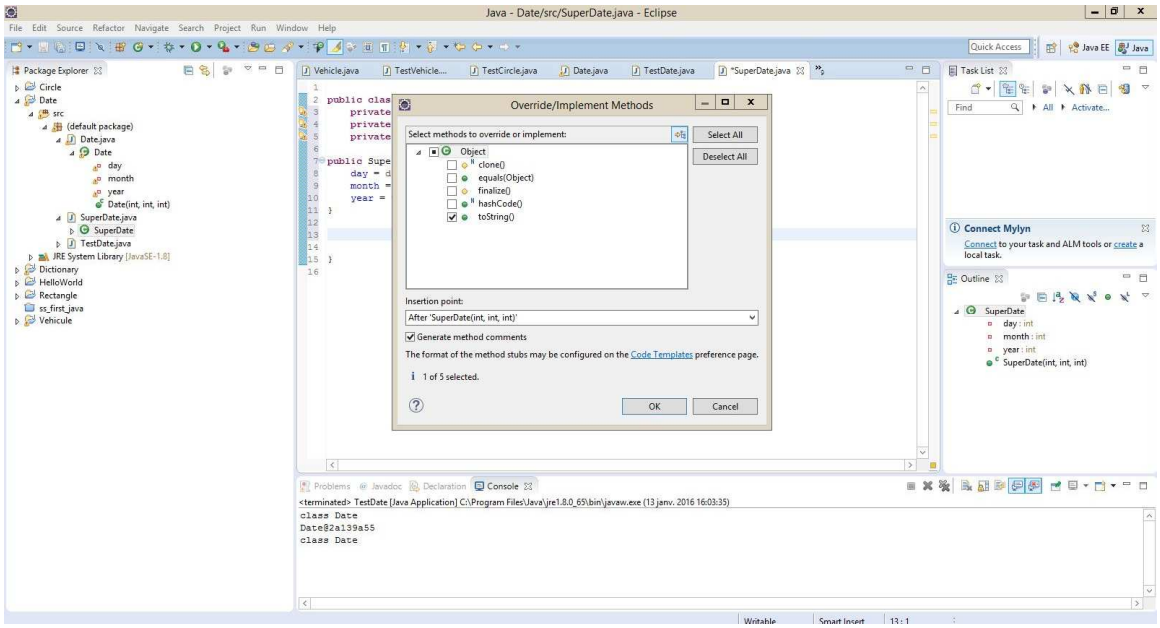
The overriding method must have the same signature (name, number, type and order of parameters) and return type as the method it overrides.

- Copy the Date class, and Rename it SuperDate class
- In the SuperDate class, override the **toString**() method inherited from java.lang.Object, so as to provide a "month/day/year" string representation of Date objects.

```
@Override
public String toString(){//Provide here an implementation ...}
```

- To override you can use eclipse functionality:
    - Right click in SuperDate.java file -> Source -> Override

C:/SSPublic/Public_ImageNote/int001017_capture_20160113_190502.jpg



- Instantiate date3 from the SuperDate class in the TestDate program
- Run the TestDate program again. What's the result now? Is it better?

- Modify the main method as follow:

```
public static void main(String[] args) {
        SuperDate date4 = new Date (5, 2, 2016);
        SuperDate date5 = new Date (5, 2, 2016);
        if(date4.equals(date5))
          System.out.println(date4+" is equal to "+date5);
        else
        System.out.println(date4+" is not equal to "+date5);
```

- What is the result?
- Does Date implement any equals() method?
- Which equals() method is called then? What would be the expected behavior of such equals()
- What operations are applied on Date objects?

- Copy the SuperDate class, and rename it GreatDate class
- Modify the GreatDate class to override the equals() method inherited from java.lang.Object.

```
@Override
public boolean equals(Object obj){
//ss provide here an implementation...
}
```

- Modify the main as follow:

```
GreatDate date6 = new GreatDate (5, 2, 2013);
GreatDate date7 = new GreatDate (5, 2, 2013);

if(date6.equals(date7))
  System.out.println(date7+" is equal to "+date6);


System.out.println(date6.equals(date7));
System.out.println(date6.equals(0));
```

- Run the TestDate program again What's the result now?

3.1.1 The “Square” class:

A Square is a specific type of Rectangle with height = width. Therefore, a Square class is a good candidate to inherit **from** the Rectangle class developed in LAB2

- Create a new Java project “Shape” with Eclipse.
- Copy the Rectangle class from LAB2 to the newly created project.
- Provides an implementation of the toString() method in the Rectangle class.
- Printing the Rectangle object with System.out.println(Object) should result in the following message displayed on the standard output:

  “Rectangle:

  - Id : id value

  - height: height value

  - width : width value

  - perimeter: perimeter value

  - area: area value

- Add a new Square class that inherits from the Rectangle class.

Square class extends the Rectangle class. Rectangle is called the parent class or superclass :

```
public class Square extends Rectangle {

….
}
```

- Add to the Square class a private “side” attribute (the side of the square, type: double) and a constructor to instantiate squares of given side.

```
public Square (double s){
        …

}
```

- Complete the Square class.

- Write a TestShape program. Observe how Square objects automatically benefit from methods defined in the Rectangle class:

…..

```
Square sq1 = new Square(7.0);


double height = sq1.getHeight();
System.out.println("height: "+height);
double width = sq1.getWidth();
System.out.println("width: "+width);
double area = sq1.getArea();
System.out.println("area: "+area);
```

….

- Run the above code. What is the output?

- Test the following code snippet. What is the output?

```
Rectangle rect1 = new Rectangle(4.0, 6.0);
System.out.println("rect1 = "+ rect1);
```

- Test the following code snippet. What is the output and why?

```
Square sq1 = new Square(7.0);
System.out.println("sq1 = "+ sq1);
```

- Explain what should be done to provide a more relevant output to the user. Apply the required modification. Run the above code once again.
- Are the attributes of Rectangle visible in the sub-class Square? Why?

- What's happen if the "id" attribute of Rectangle is now declared as protected?
- Test the following code snippet. What do you observe? Which toString() method is called at runtime?

```
Rectangle rect2 = new Square(4);
System.out.println(rect2);
```

- Test the following code snippet. What do you observe? Explain.

```
Square sq2 = new Rectangle(4,4);
System.out.println(sq2);
```

3.1.2 Abstract classes: The “Shape” class example

Rectangles, squares and circles are three particular geometric shapes.

Therefore, the Rectangle, Square and Circle classes defined before could actually inherit from a generic Shape class.

Geometric shapes support common operations, like computing “area” or “perimeter”, but performed in different ways depending on the kind of shape. For instance, circles and squares are geometric shapes with different area formulas:

- Area for square= side x side
- Area circle= p x R x R

Therefore, a Shape class could declare operations (methods) common to all shapes (e.g. computing area) but would not be able to provide an implementation of it (area formula depending on the actual shape type).

A method declared without an implementation is called an **abstract method**.

A class that contains at least one abstract method must be declared **abstract**.

Shape is then a good candidate for an abstract class.

Derived Classes from an abstract class (the subclasses or child classes) must provide an implementation of abstract methods declared in the superclass (.i.e. the abstract class).

Rule: an abstract class can’t be instantiated.

- Create a new Java Project with Eclipse:”AbstractShape”
- Add an abstract Shape class with:
  - A protected “color” attribute of type java.awt.Color
  - A constructor that initializes the color attribute
  - A public method getColor() that returns the color of the Shape
  - 2 abstract methods :
    - getArea() : objective of this method is to calculate and compute the area of the shape.
    - getPerimeter(): objective of this method is to calculate and compute the perimeter of the shape.

```
public abstract class Shape {
…

}
```

- Design a Rectangle class, a Circle class and a Square class inheriting from the abstract Shape class. These sub-classes shall override the toString() method of java.lang.Object to provide relevant information (color, height/width/side/radius, area, perimeter) when printing instances using System.out.println(Object).

- Add a Test `TestAbsractShape` program. Below is a possible main method for this test program :

```
public class TestAbsractShape {

	public static void main(String[] args) {
		// An array of 9 shapes
		  Shape[] shapes = new Shape[9];
		for(int i = 0; i < 7; i=i+3){
		   shapes[i] = new Circle(4+i, Color.black);
		   shapes[i+1] = new Square(2+i, Color.blue);
		   shapes[i+2] = new Rectangle(1+i,5+i, Color.green);
		  }
		for(int i = 0; i < 9; i++) {
		   System.out.println(shapes[i]);
		  }
		 }
}
```

- What is the type of the “shapes” instance?
- At runtime, which objects are referenced by shapes[0], shapes[1] and shapes[2]?
- Which toString() method is called in the for loop?
- Add in the Test TestAbsractShape program:
  - Shape shapeObj = new Shape(Color.yellow);
- Describe what happens

3.1.3 Interfaces

2.3.1 Defining and using an Interface

- Objects expose their behavior to other objects through their public methods.
- The set of methods of an object form its interface to the external world.
- Most of the time, an Interface consists in a list of related methods grouped together.
- An Interface declares a group of methods but does not provide an implementation of these methods, this will be the responsibility of the class implementing the interface.
- An Interface can be seen as a “contract” between the class that implements it and the user of this class.
- A class claiming to implement a given Interface promises to provide a specific behavior (the one specified by the Interface) to its users.
- The class must then implement all the methods of the interface.

3.1.4 Interfaces Labs

- Create a new Java project with Eclipse and add the following Drawable interface:

```
public interface Drawable {
	/**
	 * Draws an object using ASCII characters
	 */
	public void drawWithASCII();
	/**
	 * Draws an object using dots
	 */
	public void drawWithDots();
}
```

- Create a Rectangle class that implements this interface (you can copy the Rectangle class from previous project).

```
public class Rectangle implements Drawable {
	/**
	 * The height of the rectangle
	 */
	private double height;
	/**
	 * The width of the rectangle
	 */
	private double width;

	/**
	 * Creates a new instance of Rectangle with the given size
	 * @param h height
	 * @param w width
	 */
	public Rectangle(double h, double w) {
		height = h;
		width = w;
	}

	/**
	 * Draw the rectangle on standard
	 * output using ASCII characters
	 */
	public void drawWithASCII() {

	//ss provide here an implementation...
	}
	/**
	 * Draw the rectangle on standard
	 * output using dots(".").
	 */
```

```
    public void drawWithDots() {
  //ss provide here an implementation...
  }
```

- drawWithASCII() method could for instance provide following output for a 5x9 rectangle:

```
.........
.........
.........
```

- Note that the behavior described by this Drawable interface is not specific to rectangle or geometric shapes. Actually, a lot of objects could be drawable and then could implement this Drawable interface as well.

- Add a class Cow that implements the Drawable interface as well.

- Class Cow defines 2 private attributes name of type String and weight of type Integer.

- Class Cow provides the corresponding "getter" methods (getName, getWeight) and an implementation of the drawWithASCII() method from the Drawable interface. drawWithASCII() prints following String on the standard output:

```
class gse4.labs.com.Cow
        (__)
        (oo)
  /-------\/
 / |     ||
*  ||----||
```

Add a TestDrawable program. The program creates a Rectangle object and a Cow object and draws it on standard output.

```
public class TestDrawable {
      /**
       * @param args
       */
      public static void main(String[] args) {
      //ss provide here an implementation...


      }
```

```
}
```

- Add following code snippet to the TestDrawable program. What is the output?

```
Drawable drawable1 = new Rectangle(6, 5);
        System.out.println(drawable1.getClass());
        drawable1.drawWithASCII();

        Drawable drawable2 = new Cow("Cow", 500);
        System.out.println(drawable2.getClass());
        drawable2.drawWithASCII();
```

- What's happen if you try following code?

  ```
  System.out.println(drawable1.getArea());
  ```

- What's happen if you try following code?

  System.out.println((Rectangle)drawable1.getArea());

- What's happen if you try following code?

  System.out.println((Rectangle)drawable2.getArea());

- Add following code snippet to the TestDrawable program. What is the output?

```
if (drawable1 instanceof Rectangle)
      System.out.println(((Rectangle)drawable1).getArea());
if (drawable2 instanceof Cow)
      System.out.println(((Cow)drawable2.getName());
```

3.1.5 Multiple inheritance using Interface

- Java does not allow a class to inherit from more than one base class. For instance, following statement is not supported :

public class Smartphone extends Phone, MediaPlayer, Camera {

- Here, the Smartphone class author likely wants the Smartphone object to behave both as a Phone, a MediaPlayer and a Camera.
- If a class is not allowed to extend several base classes, Java allows a class to inherit from one base class and to implement several interfaces. Therefore, the Smartphone

class author could declare its Smartphone class as follow:

If Phone is a pre-existing class and MediaPlayer, Camcorder are interfaces:

```
public class Smartphone extends Phone implements MediaPlayer,Camera {
      /*Media Player Interface*/
```

```
/*Camera interface*/
```

- Propose a possible MediaPlayer interface (at least3 methods)
- Propose a possible Camera interface (at least 2 methods)
- Propose a possible Phone class
    - Define 2 methods:
        - one that performs a voice call
        - one that send an SMS
    - This is for illustration purpose; implement the methods with simple

```
        System.out.println("I'm the <method_name> method,
<description of the operation>");
    }
```

- Propose a possible Smartphone
    - This is for illustration purpose only, so do not add specific methods.
    - Only implement the interfaces methods with simple
        - `System.out.println` (“`I’m the <method name> method, I’m doing <description of the operation>`”);
- Write a TestSmartphone program to that test the Smartphone class