Πανεπιστήμιο Κύπρου
Τμήμα Πληροφορικής

ΕΠΛ233: Αντικειμενοστρεφής Προγραμματισμός

Διδάσκων: Δρ. Χαράλαμπος Πουλλής

Εργαστηριακή Άσκηση 4

Πρώτο Μέρος: Θεωρία (διάρκεια 15 λεπτά)

Από την ιστοσελίδα της Java διαβάστε τα ακόλουθα άρθρα:

- Classes http://java.sun.com/docs/books/tutorial/java/javaOO/classes.html
- Objects http://java.sun.com/docs/books/tutorial/java/javaOO/objects.html
- More on classes http://java.sun.com/docs/books/tutorial/java/javaOO/more.html

Απαντήστε στις ακόλουθες ερωτήσεις:

1. Consider the following class:

```
public class IdentifyMyParts {
        public static int x = 7;
        public int y = 3;
}
```

a. What are the class variables?

b. What are the instance variables?

c. What is the output from the following code:

```
IdentifyMyParts a = new IdentifyMyParts();
IdentifyMyParts b = new IdentifyMyParts();
a.y = 5;
b.y = 6;
a.x = 1;
b.x = 2;
System.out.println("a.y = " + a.y);
System.out.println("b.y = " + b.y);
System.out.println("a.x = " + a.x);
System.out.println("b.x = " + b.x);
System.out.println("IdentifyMyParts.x = " + IdentifyMyParts.x);
```

d. Write a class whose instances represent a single playing card from a deck of cards. Playing cards have two distinguishing properties: rank and suit.

e. Write a class whose instances represent a full deck of cards.

f. Write a small program to test your deck and card classes. The program can be as simple as creating a deck of cards and displaying its cards.

Δεύτερο Μέρος: Προγραμματισμός (διάρκεια 1 ώρα)

Inheritance is the capability of a class to use the properties and methods of another class while adding its own functionality. An example of where this could be useful is with an employee records system. You could create a **generic** employee class with states and actions that are common to all employees. Then more **specific** classes could be defined for salaried, commissioned and hourly employees. The generic class is known as the parent (or **superclass** or base class) and the specific classes as children (or **subclasses** or derived classes). The concept of inheritance greatly enhances the ability to **reuse** code as well as making design a much simpler and cleaner process.

The **Object** class is the highest superclass (ie. root class) of Java. All other classes are subclasses (children or descendants) of the Object class. The Object class includes methods such as:

clone()	equals()	copy(Object src)	finalize()	getClass()
hashCode()	notify()	notifyAll()	toString()	wait()

Example:

```
public class Box        {
  // what are the properties or fields
  private int length, width, height;

  // what are the actions or methods
  public void setLength(int p) {length=p;}

  public void setWidth(int p) {width=p;}

  public void setHeight(int p) {height=p;}

  public int displayVolume()
  {System.out.println(length*width*height);}

  public void showObj()
 {System.out.println(“Show object”);}
}
```

Java uses the extends keyword to set the relationship between a parent class and a child class. For example using the Box class:

```
public class GraphicsBox extends Box        {
        // define position locations
           private int left, top;

        GraphicsBox(l,w,h,left,top)   {
            super (l,w,h);
            this.left=left;
            this.top=top;
         }

        public void showObj()
           {System.out.println(super.showObj()+" more stuff here");}

        // override a superclass method
           public int displayVolume()                    {
             System.out.println(length*width*height);
             System.out.println("Location: "+left+", "+top);
            }
}
```

The GraphicsBox class assumes or **inherits** all the properties of the Box class and can now add its own properties and methods as well as override existing methods. Overriding means creating a new set of method statements for the **same** method signature (name, number of parameters and parameter types).

When extending a class constructor you can reuse the superclass constructor and overridden superclass methods by using the reserved word **super**. Note that this reference must come first in the subclass constructor. The reserved word **this** is used to distinguish between the object's property and the passed in parameter.

The reserved word **this** can also be used to reference private constructors which are useful in initializing properties.

Special Note: You cannot override final methods, methods in final classes, private methods or static methods.

As seen from the previous example, the superclass is more general than its subclass(es). The superclass contains elements and properties common to all of the subclasses. The previous example was of a **concrete** superclass that instance objects can be created from. Often, the superclass will be set up as an **abstract** class which does not allow objects of its prototype to be created. In this case, only objects of the subclass are used. To do this the reserved word **abstract** is included in the class definition.

Abstract methods are methods with no body specification. Subclasses **must** provide the method statements for their particular meaning. If the method was one provided by the superclass, it would require overriding in each subclass. And if one forgot to override, the applied method statements may be inappropriate.

```
public abstract class Animal  // class is abstract
{
  private String name;
  public Animal(String nm)      // constructor method
  { name=nm; }
  public String getName()       // regular method
  { return (name); }
  public abstract void speak(); // abstract method - note no {}
}
```

Abstract classes and methods force prototype standards to be followed (ie. they provide templates).

Interfaces are similar to abstract classes but all methods are **abstract** and all properties are **static final**. Interfaces can be inherited (ie. you can have a sub-interface). As with classes the **extends** keyword is used for inheritence.Java does not allow **multiple inheritance** for classes (ie. a subclass being the extension of more than one superclass). An **interface** is used to tie elements of several classes together. Interfaces are also used to separate **design** from **coding** as class method headers are specified but not their bodies. This allows compilation and parameter consistency testing prior to the coding phase. Interfaces are also used to set up unit testing frameworks.

As an example, we will build a **Working** interface for the subclasses of Animal. Since this interface has the method called **work()**, that method must be defined in any class using the Working interface.

```
public interface Working {
  public void work();
}
```

When you create a class that uses an **interface**, you reference the interface with the phrase **implements** “Interface1, Interface2, ...”. “Interface1, Interface2, ...” is one or more interfaces as multiple interfaces are allowed. Any class that implements an interface **must** include code for all methods in the interface. This ensures commonality between interfaced objects.

```
public class WorkingDog extends Dog implements Working
{
  public WorkingDog(String nm)
  {
    super(nm);    // builds parent
  }
  public void work()  // this method specific to WorkingDog
  {
    speak();
    System.out.println("I can herd sheep and cows");
  }
}
```

Overloaded methods are methods with the same **name signature** but either a different number of parameters or different types in the parameter list. For example 'spinning' a number may mean increase it, 'spinning' an image may mean rotate it by 90 degrees. By defining a method for handling each type of parameter you achieve the effect that you want.

Overridden methods are methods that are redefined within an **inherited or subclass**. They have the **same** signature and the subclass definition is used.

Polymorphism is the capability of an action or **method** to do different things based on the object that it is acting upon. This is the third basic principle of object oriented programming. Overloading and overriding are two types of polymorphism . Now we will look at the third type: **dynamic method binding**.

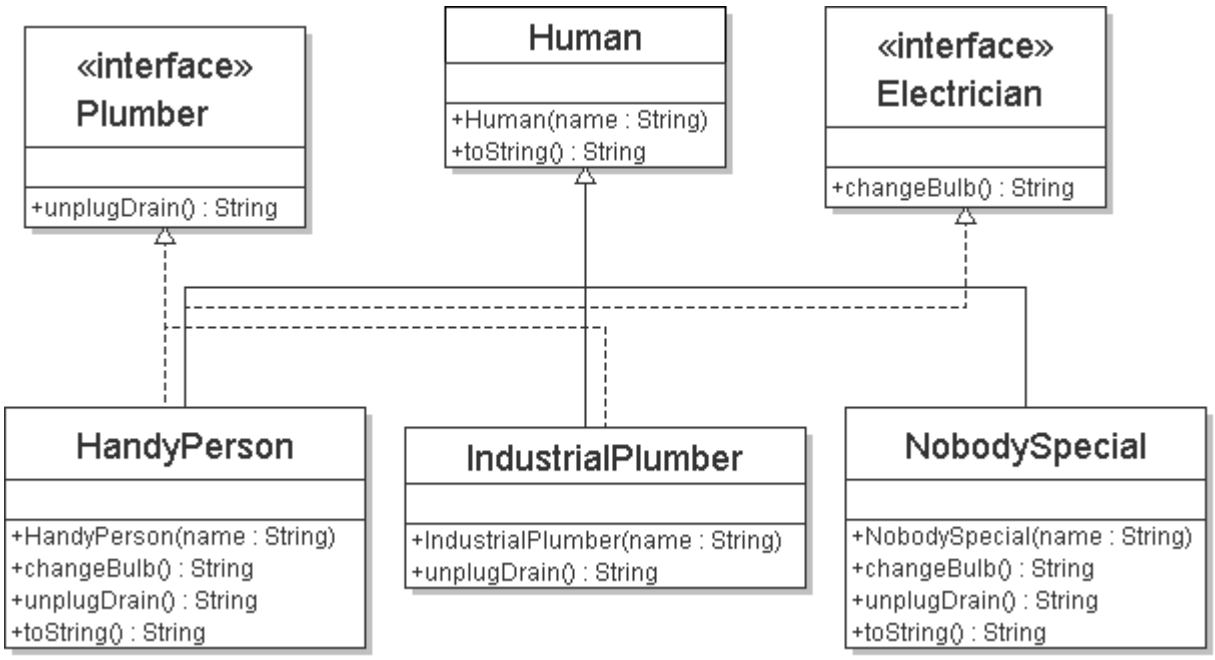
Assume that three subclasses (Cow, Dog and Snake) have been created based on the Animal abstract class, each having their own speak() method.

```
public class AnimalReference
{
  public static void main(String args[])
  Animal ref                // set up var for an Animal
  Cow aCow = new Cow("Bossy"); // makes specific objects
  Dog aDog = new Dog("Rover");
  Snake aSnake = new Snake("Ernie");

  // now reference each as an Animal
  ref = aCow;
  ref.speak();
  ref = aDog;
  ref.speak();
  ref = aSnake;
  ref.speak();
}
```

Notice that although each method reference was to an Animal (but no animal objects exist), the program is able to resolve the correct method related to the subclass object at runtime. This is known as dynamic (or late) method binding.

Πρόγραμμα: Handy Person. Γράψτε κώδικα που να αντιστοιχεί στο πιο κάτω σχεδιάγραμμα.



Ερωτήσεις: What is printed by the following?

If it is erroneous, write C for syntax (compilation) error or R for runtime error (ie, it crashes).

What does this print? _____

```
hp = new HandyPerson("Jill");
System.out.println(hp);
```

What does this print? _____

```
hu = new HandyPerson("Mike");    // Assign HandyPerson to Human
System.out.println(hu);
```

What does this print? _____

```
hp = new Human("Betty");         // Human to HandyPerson
System.out.println(hp);
```

What does this print? _____

```
hu = new HandyPerson("Chris");
System.out.println(hu.changeBulb()); // ChangeBulb
```

What does this print? _____

```
hu = new NobodySpecial("Tom");   // Assign NobodySpecial to Human
System.out.println(hu);
```

What does this print? _____

```
hp = new NobodySpecial("Bill");   // Assign NobodySpecial to HandyPerson
System.out.println(hp);
```

What does this print? _____

```
ns = new HandyPerson("Sally");   // Assign HandyPerson to NobodySpecial
System.out.println(ns);
```

What does this print? _____

```
pl = new Plumber("Andrew");      // Plumber
System.out.println(pl);
```

What does this print? _____

```
pl = new HandyPerson("Nancy");   // HandyPerson to Plumber
hp = (HandyPerson)pl;
System.out.println(hp);
```

What does this print? _____

```
ob = new HandyPerson("Ralf");   // HandyPerson to Object
System.out.println(ob);
```