Sofia University
Department of Mathematics and Informatics

Course : Applied OO Programming part 1
Date: May 8, 2020
Student Name:

Lab No. 10

Problem 1a

Write a **Java modular project**. Create the following inheritance hierarchy **Shape-Point- Circle-Cylinder** in a package of a module named ***com.geometry.types***:

A Point has two coordinates – x and y integer values

A Circle is a Point and has a radius (integer value)

A Cylinder is a Circle and has a height (integer value)

Implement ***interface java.lang.Comparable*** in the root class of this inheritance hierarchy as follows

1. Create the respective **UML class diagram** in **IntelliJ** for the above inheritance hierarchy where ***interface Comparable is*** defined as follows:

```
interface Comparable
{
    int compareTo(Object obj);
    // 1.validates obj is not null,
    // if obj is null, then this is greater
    // 2. Uses operator "instanceof " to check the obj reference type
    // make an explicit type conversion when both types are the same
    // if obj reference types are different, then this is greater
    // 3. compares the this reference in the implementation class
    // with the obj reference, according to the above definition
    // for the meaning of the relation greater for the given class
}
```

Implement ***compareTo()*** for Point- Circle- Cylinder objects as follows:

a) a **Point** object P1 is greater than another **Point** object P2, if P1.mX > P2.mX and P1.mY > P2.mY, when P1.mX = P2.mX. (for instance point (1,2) is greater than point (1,1) **) Two points are equal when their coordinates are equal**

b) a **Circle** object C1 is greater than another **Circle** object C2, if the center point of C1 (which is a point object) is greater than the center point of C2 (which is also a point object) and C1.mRadius > C2.mRadius, when the center point of C1 is equal to the center point of C2. **Two circles are equal when their center points and radiuses are equal**

c) a **Cylinder** object C1 is greater than another **Cylinder** object C2, if the circle of C1 (which is a Circle object) is greater than the circle of C2 (which is also a Circle object) and C1.mHeight > C2.mHeight, when the circle of C1 is equal to the circle of C2 **Two cylinders are equal when their circles and heights are equal**

2. Add a module named ***com.geometry.utils*** and write a class ***SelectionSort*** with a ***static*** method ***sortArray*** in a package of that module
   public static void sortArray(Comparable[] arr)
   allowing you to sort (*using the Selection sort algorithm, see Lecture 5)* an array of objects of any kind that implement ***interface Comparable.*** For instance, class ***SelectionSort*** should be able to sort array of ***Circle*** objects or an array of ***Cylinder*** objects that implement ***interface Comparable*** .
3. Write a class ***SelectionSortTest*** in the same package of class ***SelectionSort*** to test class ***SelectionSort*** where class ***SelectionSortTest*** is a Console application. The application class must have a static class member- a reference to an array ***arrComparable*** of type ***Comparable*** with **9** elements. **Employ** ***arrComparable*** to sort these elements with ***sortArray(Comparable[] arr)***:
   a) three **Point**s (the coordinates should be random generated in the interval [10,50]) by assigning the **Point**s to ***arrComparable***.
   b) three **Circle**s (the centers of the circles should be the three points, the radiuses of the circles should be random generated in the interval [10,30]) by assigning the **Circle**s to ***arrComparable***.
   c) three **Cylinder**s (the circles of the circles should be the above defined Circle objects, the heights of the cylinders should be random generated in the interval [10,60]) by assigning the **Cylinder**s to ***arrComparable***.

   Display the objects for each case (a- c) in the respective array sorted by the ***sortArray(Comparable[] arr)*** method of class ***BubbleSort*** in ascending order. You **must** use **overriding of method** toString***()*** and late binding (**polymorphism**) to display each one object.

Use adopted **best practices for naming packages** in modules and **correctly define the module descriptors**.

Problem 1b

Implement the **Business Delegate pattern** for solving Problem 1a, where the ***Client*** if a JavaFX application that display the result of sorting array ***arrComparable*** with elements as above defined by default. The ***SelectionSort*** can serve as a concrete implementation of the ***BusinessService*** interface. For testing purposes add another class that implements the ***BusinessService***, such as ***InsertionSort.*** The JavaFX application should allow the user to ***setServiceType*** by clicking a ***Button*** and display in a ***TextArea*** the execution of the appropriately defined ***doTask***() method of the ***BusinessDelegate***.

Problem No. 2

Create a `Singleton` class in Java.

A `Singleton` is a class that returns the same and the same object everytime it is used. It should have `private` data members, which are initialized by a `private` constructor and a private static reference to a `Singleton` object, instantiated by the `private Singleton` constructor. A reference to the `private static Singleton` is provided through a public `static Singleton getInstance`() method.
Create the `Singleton` class and compare two references of that class to make sure they have the same memory references in a `Console` application.

Problem No. 3a

Create a Java Modular application with a base class with two methods. In the first method, call the second method. Inherit a class and override the second method. Create an object of the derived class, upcast it to the base type, and call the first method. Explain what happens.

Problem No. 3b

Create a Java Modular application with a base class with an `abstract print`( ) method that is overridden in a derived class. The overridden version of the method prints the value of an int variable defined in the derived class. At the point of definition of this variable, give it a nonzero value. In the base-class constructor, call this method. In `main`( ), create an object of the derived type, and then call its `print`( ) method. Explain the results.

Problem No. 3c

Create a Java Modular application with an abstract class with no methods. Derive a `class` and add a method. Create a `static` method that takes a reference to the base class, downcasts it to the derived class, and calls the method.
In `main`( ), demonstrate that it works. Now put the abstract declaration for the method in the base class, thus eliminating the need for the downcast.

Problem No. 4a
Write an **enum type** ***`TrafficLight`*** , whose ***constants*** **(RED, GREEN, YELLOW)** take one parameter – the duration of the light in milliseconds. ***Use***
long tm = System.currentTimeMillis();
to get the current time in milliseconds ***Write a program*** to test the **enum type** ***`TrafficLight`*** , so that it display in a loop the traffic lights text for the time duration set for each traffic light Exit the program test after 90 seconds pass for its start..

Problem No. 4b
Create a ***`Months`*** ***`enumeration`*** in Java

It should be possible to create **only 12** instances of the objects ***JAN, FEB,…, DEC***. There should be a ***toString***() method in that class allowing each month to display the full name "***January"***, ***"February***, …, "***December"*** by referring to the respective object name or by an index in an **array** ***Month***

Months.JAN→ displays "***January***"

Months.Month[[0]→ displays "***January***"

Create the ***Months*** class and test the above class properties in a ***Console*** application.

Problem No. 5

Create three `interface`**s**, each with two methods. **Inherit a new fourth** `interface` from the three, adding a new method `void m()`**. Create a** `class A` **by implementing the fourth interface** and also inheriting from a concrete `class B` with implementation of method `m()` declared in the fourth interface(each of the implementations should write a string on the ***Console*** indicating the method name). Now write four methods, each of which takes one of the four interfaces as an argument(each of the methods should make sure the argument is derive from the respective interface and run the respective interface methods with the so passed argument reference). In `main`(), create an object of your class and pass it to each of the methods. Create a `UML class` diagram and write the definitions of the interfaces and the classes involved in this UML class diagram

Problem No. 6

Consider the following class definitions and describe what would be output by the code segment.

```
public class A {
    public A() { System.out.println("A"); }
}
public class B extends A {
    public B() { System.out.println("B"); }
}
public class C extends B {
    public C() { System.out.println("C"); }
}
          // Determine the output.
A a = new A();
B b = new B();
C c = new C();
```

Optional problems:

a) Prove that the fields in an interface are implicitly **static** and **final**.

b) **Create an interface** containing three methods, in its own package. Implement the interface in a different package.

c) Prove that all the methods in an interface are automatically **public**.

d) **Create three interfaces**, each with two methods. **Inherit a new interface** from the three, adding a new method. **Create a class by implementing the new interface** and also inheriting from a concrete class. Now write four methods, each of which takes one of the four interfaces as an argument. In main(), create an object of your class and pass it to each of the methods

<u>Problem No. 7</u>

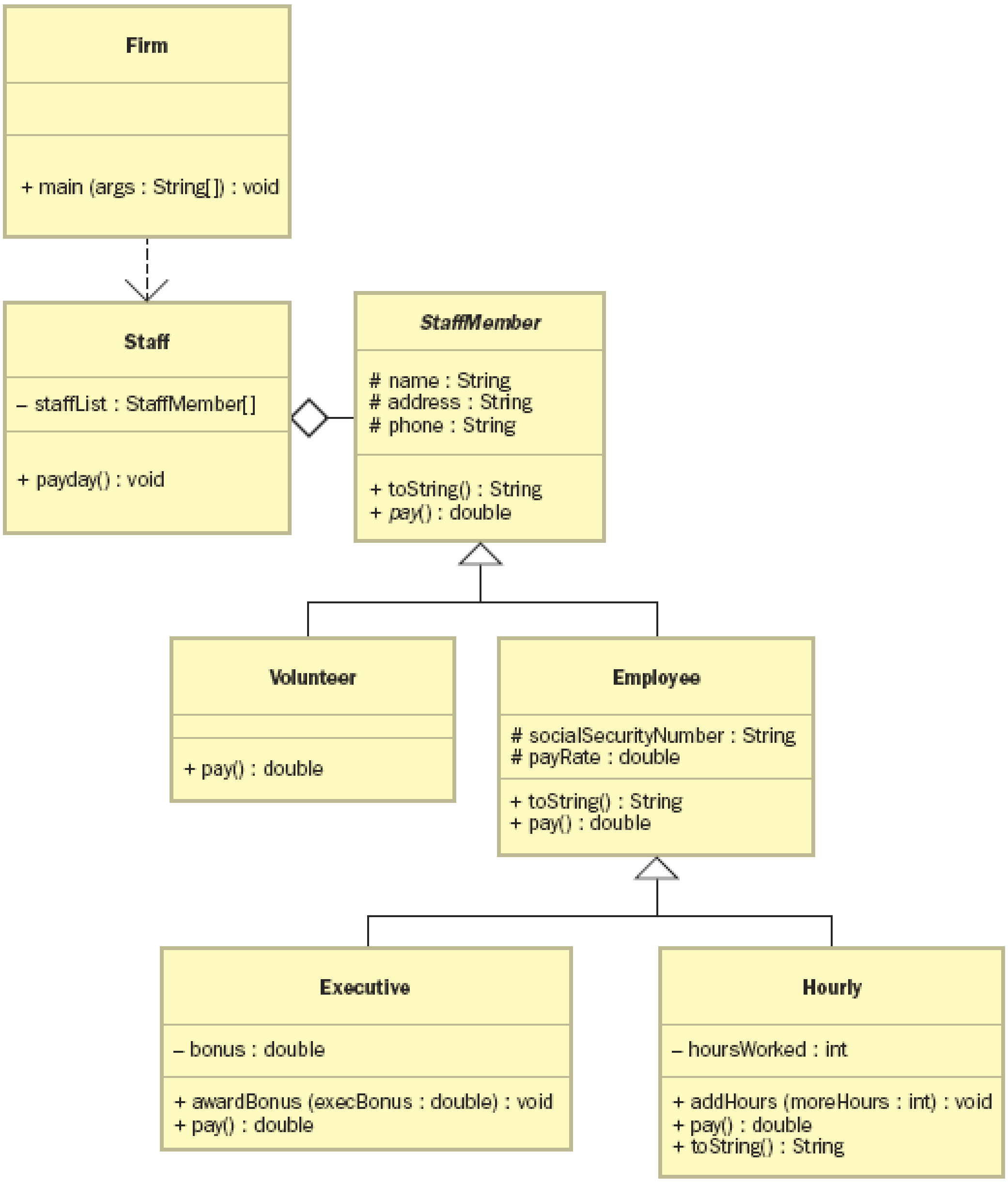
Create a Java Modular application with a class named `Person` and its two subclasses named `Student` and `Employee`. Make `Faculty` and `Staff` subclasses of `Employee`. A person has a name, address, phone number, and email address. A student has a `class status` (freshman, sophomore, junior, or senior). Define the status as a `constant`. An employee has an office, salary, and date hired. Use the `LocalDate` class introduced in Lecture 3 create an object for date hired. A faculty member has office hours and a rank. A staff member has a title. Override the `toString` method in each class to display the class name and the person's name.

Draw the UML diagram for the classes and implement them.

Write a test program **in another Java module** that creates a `Person`, `Student`, `Employee`, `Faculty`, and `Staff`, and invokes their `toString`() methods.

<u>Problem No. 8</u>

Implement the classes in the following UML class diagram in **a Java Modular application**



Consider the class hierarchy shown above.

The classes in it represent various types of employees that might be employed at a particular company. Let's explore an example that uses this hierarchy to pay a set of employees of various types.

The `Firm` class contains a main driver that creates a `Staff` of employees and invokes the `payday` method to pay them all. The program output includes information about each employee and how much each is paid (if anything).

The `Staff` class maintains an `array` of objects that represent individual employees of various kinds. Note that the array is declared to hold `StaffMember` references, but it is actually filled with objects created from several other classes, such as `Executive` and `Employee`. These classes are all descendants of the `StaffMember` class, so the assignments are valid. The `staffList` array is filled with polymorphic references.

The `payday` method of the `Staff` class scans through the list of employees, printing their information and invoking their pay methods to determine how much each employee should be paid. The invocation of the `pay` method is polymorphic, because each class has its own version of the `pay` method.

The `StaffMember` class is `abstract`. It does not represent a particular type of employee and is not intended to be instantiated. Rather, it serves as the ancestor of all employee classes and contains information that applies to all employees. Each employee has a `name`, `address`, and `phone number`, so variables to store these values are declared in the `StaffMember` class and are inherited by all descendants. The `StaffMember` class contains a `toString` method to return the information managed by the `StaffMember` class. It also contains an `abstract` method called `pay`, which takes no parameters and returns a value of type double. At the generic `StaffMember` level, it would be inappropriate to give a definition for this method.

Write a test program **in another Java module** to test inheritance class hierarchy.