Lab 06: Polymorphism

Objective(s):

1. Polymorphism

1: Polymorphism

The word polymorphism is used in various contexts and describes situations in which something occurs in several different forms. In computer science, it describes the concept that objects of different types can be accessed through the same interface.

Java supports 2 types of polymorphism:

- static or compile-time
- dynamic

Static Polymorphism:

Java, like many other object-oriented programming languages, allows you to implement multiple methods within the same class that use the same name but a different set of parameters (method with different signature). That is called method overloading and represents a static form of polymorphism.

Dynamic Polymorphism:

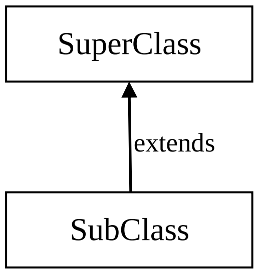
This form of polymorphism doesn't allow the compiler to determine the executed method. The JVM needs to do that at runtime hence also known as Runtime Polymorphism.

Within an inheritance hierarchy, a subclass can override a method of its superclass. That enables the developer of the subclass to customize or completely replace the behavior of that method.

It also creates a form of polymorphism. Both methods, implemented by the super- and subclass, share the same name and parameters (method with same signature) but provide different functionality.

Method overriding is one of the ways in which Java supports Runtime Polymorphism. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

- When an overridden method is called through a superclass reference, Java determines which version (superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
- At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed
- A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time.



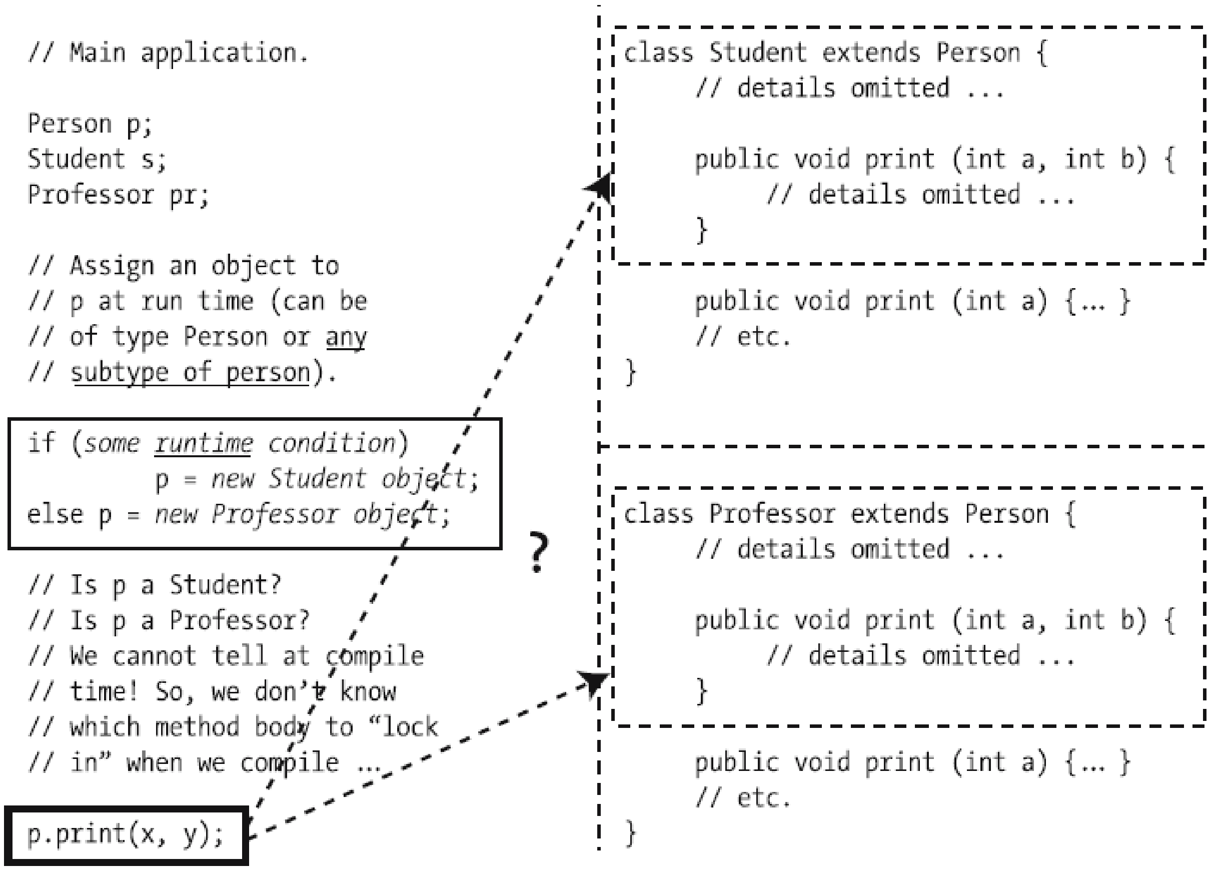
Upcasting

SuperClass obj = new SubClass();

Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

```
class A{}
class B extends A{}
A a = new B();//upcasting
```

Compile-time ambiguity as to which overridden method to invoke



```
// Main application.

Person p;
Student s;
Professor pr;

// Assign an object to
// p at run time (can be
// of type Person or any
// subtype of person).

if (some runtime condition)
          p = new Student object;
else p = new Professor object;

// Is p a Student?
// Is p a Professor?
// We cannot tell at compile
// time! So, we don't know
// which method body to "lock
// in" when we compile ...

p.print(x, y);
```

```
class Student extends Person {
     // details omitted ...

     public void print (int a, int b) {
          // details omitted ...
     }

     public void print (int a) {... }
     // etc.
}
```

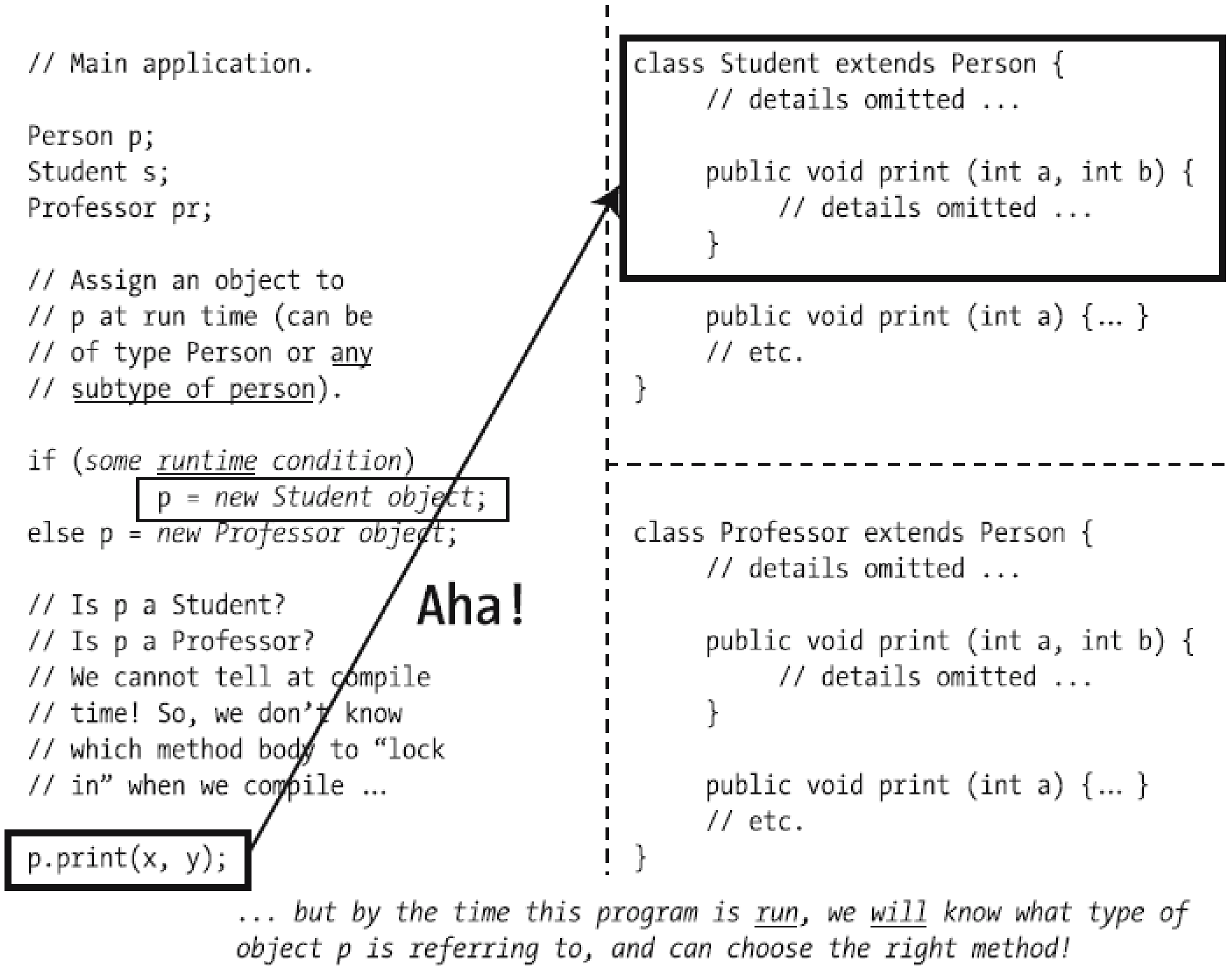
```
class Professor extends Person {
     // details omitted ...

     public void print (int a, int b) {
          // details omitted ...
     }

     public void print (int a) {... }
     // etc.
}
```

At compile time, we can't predict whether p will be a Professor or a Student object at run time ...

Polymorphism at work (all known as run-time binding)



```
// Main application.

Person p;
Student s;
Professor pr;

// Assign an object to
// p at run time (can be
// of type Person or any
// subtype of person).

if (some runtime condition)
      p = new Student object;
else p = new Professor object;

// Is p a Student?
// Is p a Professor?
// We cannot tell at compile
// time! So, we don't know
// which method body to "lock
// in" when we compile ...

p.print(x, y);
```

```
class Student extends Person {
     // details omitted ...

     public void print (int a, int b) {
          // details omitted ...
     }

     public void print (int a) {… }
     // etc.
}
```

```
class Professor extends Person {
     // details omitted ...

     public void print (int a, int b) {
          // details omitted ...
     }

     public void print (int a) {… }
     // etc.
}
```

... but by the time this program is run, we will know what type of object p is referring to, and can choose the right method!

Lab Tasks:

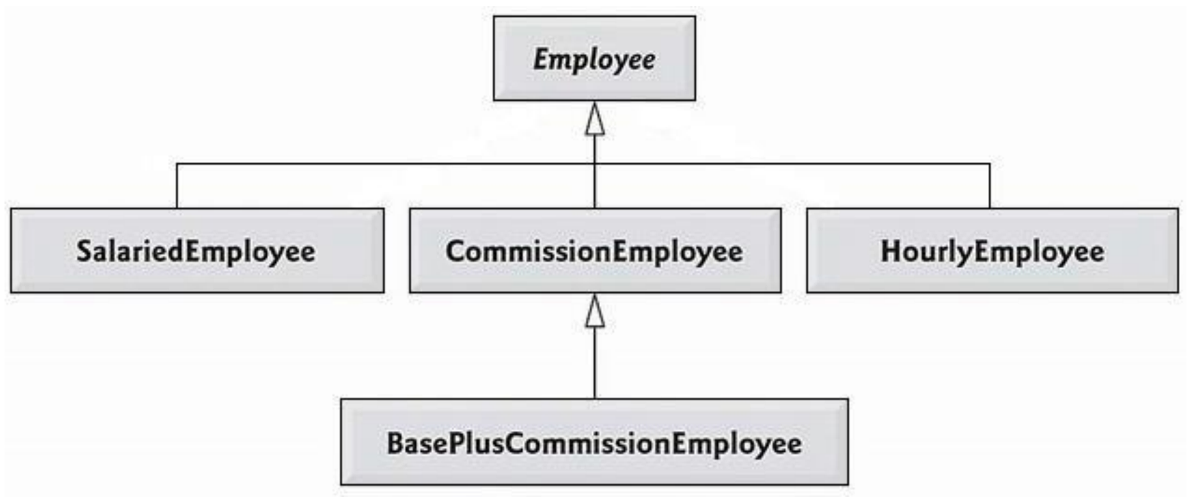
Exercise 1(a)

Create a payroll system using **classes**, **inheritance** and **polymorphism**

Four types of employees paid weekly

1. Salaried employees: fixed salary irrespective of hours
2. Hourly employees: 40 hours salary and overtime (> 40 hours)
3. Commission employees: paid by a percentage of sales
4. Base-plus-commission employees: base salary and a percentage of sales

The information know about each employee is his/her first name, last name and national identity card number. The reset depends on the type of employee.



Step by Step Guidelines

Step 1: Define Employee Class

- Being the base class, Employee class contains the common behavior. Add firstName, lastName and CNIC as attributes of type String
- Provide getter & setters for each attribute
- Write default & parameterized constructors

- Override **toString**() method as shown below

```
public String toString( ) {
return firstName + " " + lastName + " CNIC# " + CNIC ;   }
```

- Define **earning()** method as shown below

```
public double earnings( ) { return 0.00;   }
```

Step 2: Define SalariedEmployee Class

- Extend this class from Employee class.
- Add **weeklySalary** as an attribute of type double
- Provide **getter** & **setters** for this attribute. Make sure that **weeklySalary** never sets to **negative** value. (use if)
- Write **default** & **parameterize** constructor. Don't forget to call default & parameterize constructors of Employee class.
- Override **toString**() method as shown below

```
public String toString( ) {     return "\nSalaried employee: " +
super.toString(); }
```

- Override **earning**() method to implement class specific behavior as shown below

```
public double earnings( ) {   return weeklySalary;   }
```

Step 3: Define HourlyEmployee Class

- Extend this class from Employee class.
- Add **wage** and **hours** as attributes of type double
- Provide **getter** & **setters** for these attributes. Make sure that **wage** and **hours** never set to a negative value.
- Write default & parameterize constructor. Don't forget to call default & parameterize constructors of Employee class.
- Override **toString**() method as shown below

```
public String toString( ) { return "\nHourly employee: " +
super.toString(); }
```

- Override **earning**() method to implement class specific behaviour as shown below

```
public double earnings( ) { if (hours <= 40){ return wage * hours;
} else{ return 40*wage + (hours-40)*wage*1.5; } }
```

Step 4: Define CommissionEmployee Class

- Extend this class form Employee class.
- Add **grossSales** and **commissionRate** as attributes of type double

- Provide **getter** & setters for these attributes. Make sure that grossSales and commissionRate never set to a negative value.
- Write default & parameterize constructor. Don't forget to call default & parameterize constructors of Employee class.
- Override **toString**() method as shown below

  ```
  public String toString( ) {  return "\nCommission employee: " +
  super.toString(); }
  ```
- Override **earning**() method to implement class specific behaviour as shown below

  ```
  public double earnings( ) { return grossSales * commisionRate; }
  ```

Step 5: Define BasePlusCommissionEmployee Class

- Extend this class form **CommissionEmployee** class not from Employee class. Why? Think on it by yourself
- Add **baseSalary** as an attribute of type double
- Provide **getter** & **setters** for these attributes. Make sure that **baseSalary** never sets to negative value.
- Write default & parameterize constructor. Don't forget to call default & parameterize constructors of Employee class.
- Override **toString**() method as shown below

  ```
  public String toString( ) {  return "\nBase plus Commission employee: " +
  super.toString();  }
  ```
- Override **earning**() method to implement class specific behaviour as shown below

  ```
  public double earnings( ) {  return baseSalary + super.earning();  }
  ```

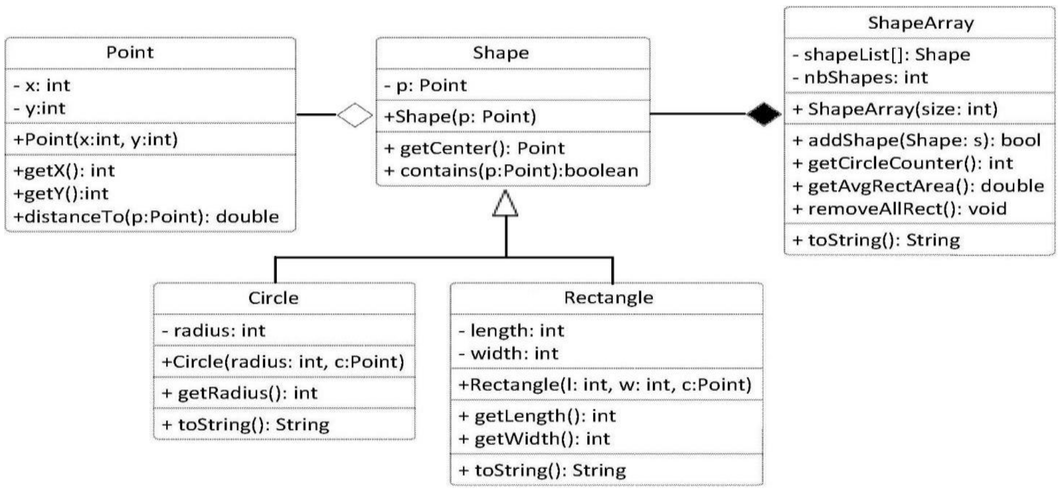
Exercise 1(b)

Step 6: Putting it all Together

```
public class PayRollSystemTest {
    public static void main (String [] args) {
    Employee firstEmployee = new SalariedEmployee("Muhammad" ,"Ali","11111-
1111", 800.00 );
    Employee secondEmployee = new CommissionEmployee("Tarwan" ,"Kumar",
"222-22-2222", 10000, 0.06 );
    Employee thirdEmployee = new BasePlusCommissionEmployee("Fabeeha",
"Fatima", "333-33-3333", 5000 , 0.04 , 300 );
```

```
    Employee fourthEmployee = new HourlyEmployee( "Hasnain" , "Ali", "444-44-
4444" , 16.75 , 40 );
    // polymorphism: calling toString() and earning() on Employee's reference
    System.out.println(firstEmployee);
    System.out.println(firstEmployee.earnings());
    System.out.println(secondEmployee);
    System.out.println(secondEmployee.earnings());
    System.out.println(thirdEmployee);
    // performing downcasting to access & raise base salary
    BasePlusCommissionEmployee currentEmployee =
    (BasePlusCommissionEmployee) thirdEmployee;
    double oldBaseSalary = currentEmployee.getBaseSalary();
    System.out.println( "old base salary: " + oldBaseSalary) ;
     currentEmployee.setBaseSalary(1.10 * oldBaseSalary);
    System.out.println("new base salary with 10% increase is:"+
currentEmployee.getBaseSalary());
    System.out.println(thirdEmployee.earnings() );
    System.out.println(fourthEmployee);
    System.out.println(fourthEmployee.earnings() );
    } // end main
} // end class
```

Exercise 2(a)



Implement classes: Shape, Circle and Rectangle based on the class diagram and description below:

Class Point implementation is given as follow:

```
class Point {
     private int x;  private int y;
     public Point(int x, int y) {  this.x = x;  this.y = y;  }
     public int getX() { return x;}
     public int getY() { return y;}
     public double distanceTo(Point p) {
     return Math.sqrt((x-p.getX())*(x-p.getX())+  (y-p.getY())*(y-
p.getY()));
     }
     public String toString() {  return "("+x+", "+y+")";  }
}
```

Class **Shape** has:

- An attributes of type Point, specifies the center of the shape object.
- A constructor that allows to initialize the center attribute with the value of the passed parameter
- A method that takes an object of type Point as a parameter and returns true if the point resides within the shape's area, and false otherwise.

Class **Circle** has:

- An attribute of type integer specifies the radius measure of the circle
- A constructor that takes a Point parameter to initialize the center and an integer parameter to initialize the radius
- A getRadius method to return the value of the attribute radius
- An overriding version of toString method to return the attribute values of a Circle object as String

Class **Rectangle** has:

- Two integer attributes represents the length and width of the Rectangle object
- A constructor to initialize the center, length and width attribute for a new Rectangle object
- Methods getLength and getWidth returns the values of attributes length and width respectively
- An overriding version of toString method to return the attribute values of a Rectangle object as a String

Class **ShapesArray**

- displayrectsinfo() display all rectangles information
- getCirclecounter():int return the number of circles
- getAvgAreas():double return the average area of all shapes
- removeallrect() delete all rectangles

Exercise 2(b)

Step 6: Putting it all Together

Implementation TestShape as given.

create ShapesArray object with size=20

Display these options

1. add new shape
    a. for rectangle (ask for details)
    b. for circle (ask for details)
2. display all rectangles
3. display the average shapes area
4. display the number of circles
5. remove all rectangles
6. exit

Exercise 3

Following is an example of abstract class:

```
abstract class Player
{
    String name;
    abstract void setName(String str);
    String getName()
    {
        return name;
    }
}
```

If you try to create an instance of this class like the following line you will get an error:

```
Player player = new Player();
```

In order to avoid this error solve this problem to create an instance?

Notice that setName method is abstract too and has no body. That means you must implement the body of that method in the child class.

We have provided the abstract Player class. Now you have to write the FootballPlayer class. Your class mustn't be public.

Now create a Main class. In the Main class create an instance of a class called FootballPlayer.

Get an input from user for a name to enter.

Then print that name.

Sample Input: Aariz Ali

Sample Output:

Football player name is: Aariz Ali

Exercise 4

Create a Parent Class name Animal with instance variable **name**, **age** and **gender**, also a method name **ProduceSound()**.

1. Create child classes of Animal Dog, Frog, Kitten and Tomcat. Dog, Frog, Cats are animal. Kittens are female cats and Tomcats are male cats. Define useful constructors and methods.
2. Modify the ProduceSound() method inherited by child class by its type "e.g for Dog ProduceSound("Bow wow")".
    - *Hint:* method OverRiding will be used(MayBe just a keyWord would be used and everything else would be same as parent class).
3. Create an array of different kind of animals and calculate the average age of each kind of animals. **(hint: you can use instanceOf method for this task)**

Exercise 5

(a) Create a class named Movie that can be used with your video rental business. The Movie class should track the Motion Picture Association of America (MPAA) rating (e.g., Rated G, PG-13, R), ID Number, and movie title with appropriate accessor and mutator methods. Also create an equals() method that overrides Object 's equals() method, where two movies are equal if their ID number is identical. Next, create three additional classes named Action , Comedy , and Drama that are derived from Movie. Finally, create an overridden method named calcLateFees that takes as input the number of days a movie is late and returns the late fee for that movie. The default late fee is \$2/day. Action movies have a late fee of \$3/day, comedies are \$2.50/day, and dramas are \$2/day. Test your classes from a main method.

(b) Extend the previous problem with a Rental class. This class should store a Movie that is rented, an integer representing the ID of the customer that rented the movie, and an integer indicating how many days late the movie is. Add a method that calculates the late fees for the rental. In your main method, create an array of type Rental filled with sample data of all types of movies. Then, create a method named lateFeesOwed that iterates through the array and returns the total amount of late fees that are outstanding.