**Sukkur Institute of Business Administration University**

Department of Computer Science

**Object Oriented Programming using Java**

BS – II (CS/AI/SE)

Spring-2024

# Lab # 09: Let's learn about Inheritance and Polymorphism

**Instructor:** Nimra Mughal

## Objectives

After performing this lab, students will be able to understand:

- Inheritance in Java
- Final, super, and this keyword
- Method overriding
- Polymorphism
    - Dynamic
    - Static

## Quick Mind boosters

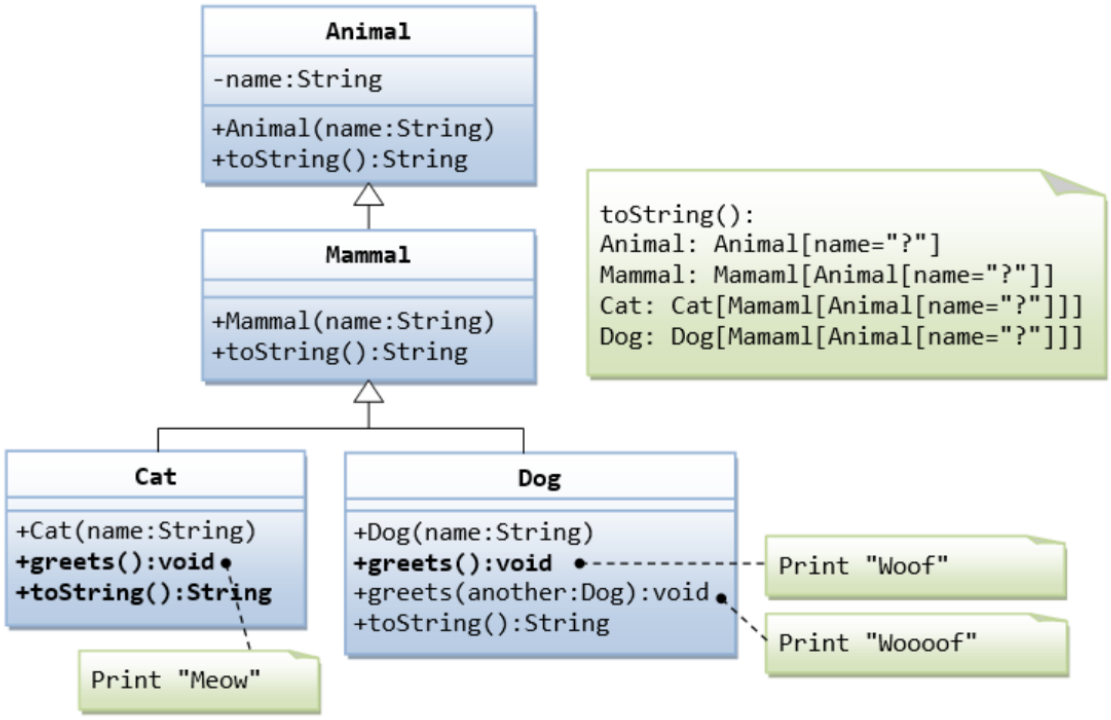
**Answer the following Questions:**

1. Can you restrict a class from inheriting another class?
2. Can we access private instances in derived classes?
3. What is difference between this and super keyword?
4. Why multiple-level inheritance is not allowed in Java?
5. Baki, Do you wanna play **Output Prediction Game** before final exams and upcoming quiz?

## Inheritance

Let's practice exercises of chapter4 and 5 that you could not yet completed last week together. In addition, Let's also solve some exercises from hacker rank.

Can you solve this problem?

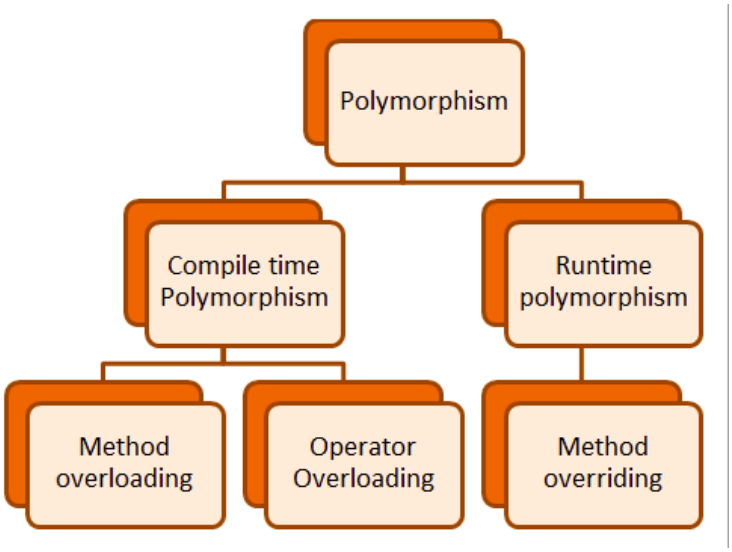



## Polymorphism

The word polymorphism means having many forms. In simple words, that allows us to perform a single action in different way. The word "poly" means many and "morphs" means forms, So it means many forms.

Java supports 2 types of polymorphism:

- static or compile-time
- dynamic or runtime polymorphism




## Static Polymorphism:

```
class DemoOverload{
    public int add(int x, int y){  //method 1
    return x+y;
    }

    public int add(int x, int y, int z){ //method 2
    return x+y+z;
    }

    public int add(double x, int y){ //method 3
    return (int)x+y;
    }

    public int add(int x, double y){ //method 4
    return x+(int)y;
    }
}

class Test{
    public static void main(String[] args){
    DemoOverload demo=new DemoOverload();
    System.out.println(demo.add(2,3));      //method 1 called
```

```
    System.out.println(demo.add(2,3,4));    //method 2 called
    System.out.println(demo.add(2,3.4));    //method 4 called
    System.out.println(demo.add(2.5,3));    //method 3 called
    }
}
```

In the above example, there are four versions of add methods. The first method takes two parameters while the second one takes three. For the third and fourth methods, there is a change of order of parameters. The compiler looks at the method signature and decides which method to invoke for a particular method call at compile time.

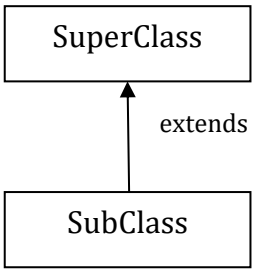
## Dynamic(Runtime) Polymorphism:

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.

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.
- 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();

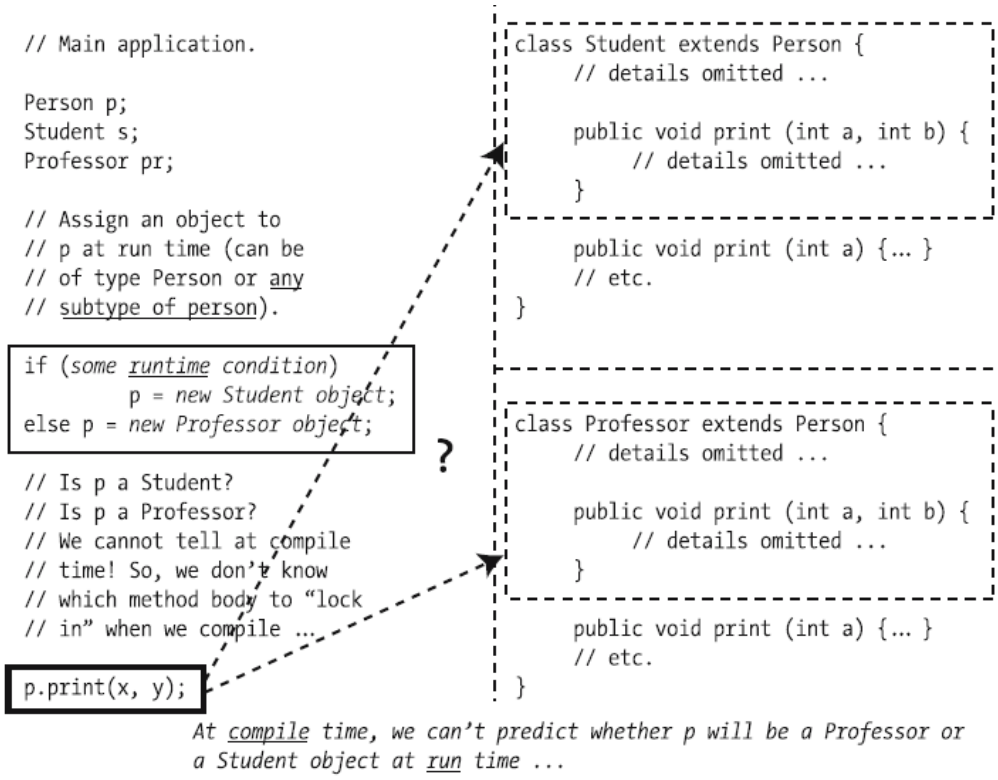
Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

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

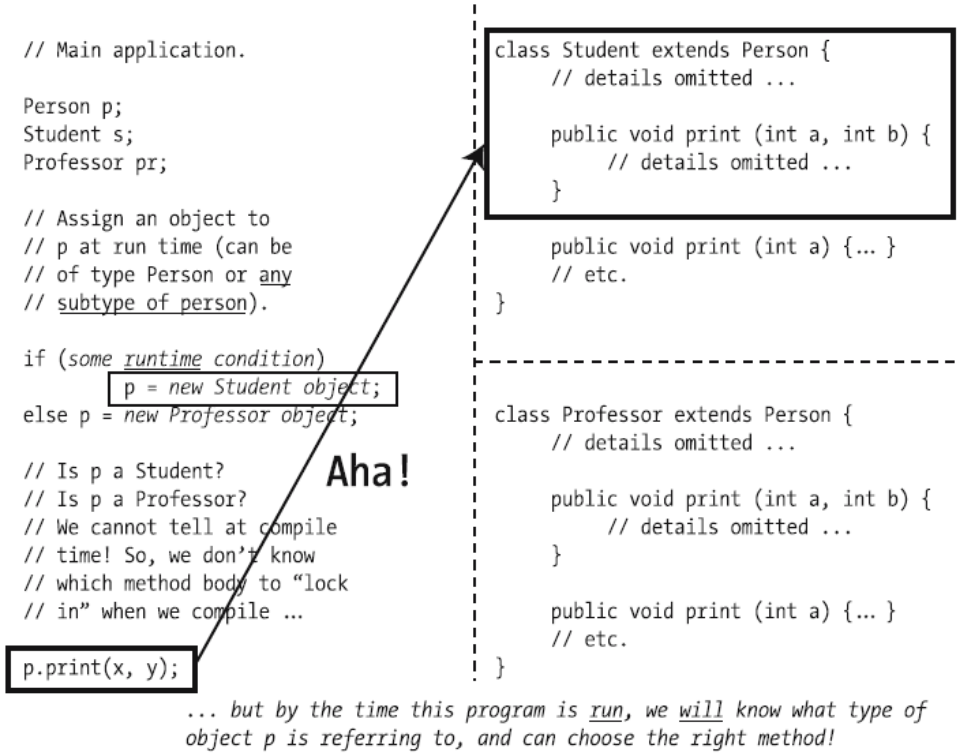
```
class Vehicle{
    public void move(){
    System.out.println("Vehicles can move!!");
    }
}

class MotorBike extends Vehicle{
    public void move(){
    System.out.println("MotorBike can move and accelerate too!!");
    }
}

class Test{
    public static void main(String[] args){
    Vehicle vh=new MotorBike();
    vh.move();    // prints MotorBike can move and accelerate too!!
    vh=new Vehicle();
    vh.move();    // prints Vehicles can move!!
    }
}
```

*Simply, one can say that overridden method is called through a reference of parent class, then type of the object determines which method is to be executed .*

**Compile-time ambiguity as to which overridden method to invoke**



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




# Exercises

**Mega Exercise: Solve Exercises from the HackerRank (OOP track topics) and the link provided.**
https://www.hackerrank.com/domains/java

## Exercise1:

Define a class **named Message** that contains an instance variable of type String named text that stores any textual content for the Message. Create a method named toString that returns the text field and also include a method to set this value.

Next, define a class for **SMS** that is derived from Message and includes instance variables for the recipientContactNo. Implement appropriate accessor(getter) and mutator(setter) methods. The body of the SMS message should be stored in the inherited variable text. Redefine the toString method to concatenate all text fields.

Similarly, define a class for **Email** that is derived from Message and includes an instance variable for the sender, receiver, and subject. The textual contents of the file should be stored in the inherited variable text. Redefine the toString method to concatenate all text fields.

**(A).** Finally, Create sample objects of type Email and SMS in your main method and class. Test your objects bypassing them to the following subroutine that returns true if the object contains the specified keyword in the text property.

```
public static boolean ContainsKeyword(Message messageObject, String keyword) {
    if (messageObject.toString().indexOf(keyword, 0) >= 0)
        return true;
    return false;
}
```

**Output:**

```
SMS: 0300-1231231 How are you bro........?

Email: Sender: reciver@email.com Reciver: sender@email.com
How are you bro........?

Message: This is java
Contains IS keywordtrue
```

**(B).** Extend the code and include a method to encode the final message "This is Java" using an encoding scheme, according to which, each character should be replaced by the character that comes after it. For example, if the message contains character B or b, it should be replaced by C or c accordingly, while Z or z should be replaced with an A or a. If the final message is "This is Java", then the encoded message should be "Uijt jt Kbwb"

**Output:**

```
SMS: 0300-1231231 How are you bro........?

Email: Sender: reciver@email.com Reciver: sender@email.com
How are you bro........?

Message: This is java
Contains IS keywordtrue
"This is Java" encoded as: Uijt jt Kbwb
```

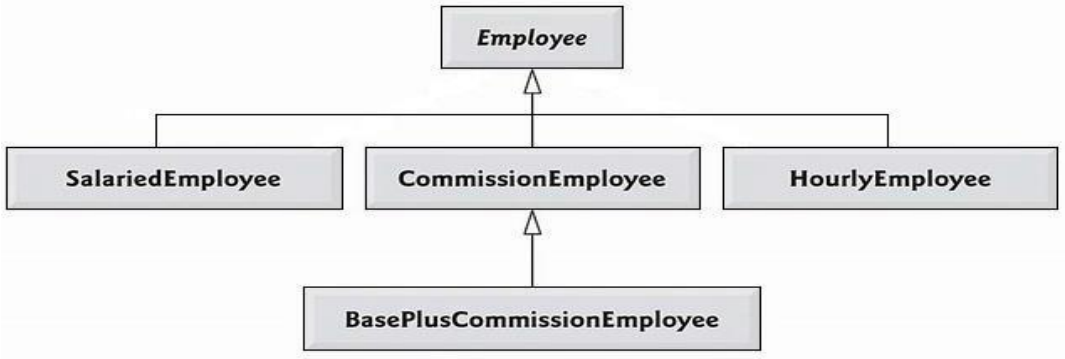
## Exercise 2 (a)

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

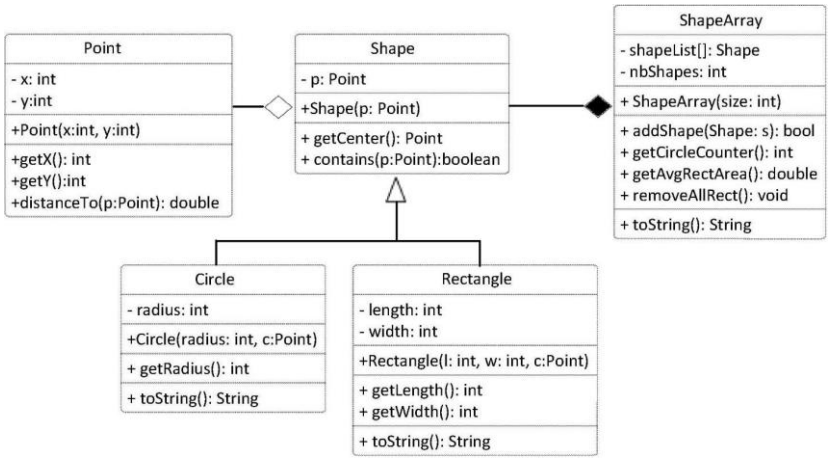
```
public double earnings( ) {  return baseSalary + super.earning();  }
```

## Exercise 2 (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 3 (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 3 (b)

### 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 4

Create a class Maths that contains one method named as display ( ) that display the message as “Hello I am display method of class Maths”. Algebra class is derived from Maths class. It contains one method display( ) that display “Hello I am display method of Algebra”. You have to perform upcasting for creating object and display the display method that show method overriding.

## Exercise 5

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)**