



## Sukkur Institute of Business Administration University

Department of Computer Science

BS – II (CS/SE/AI) Spring 2024

### Object Oriented Programming

### Lab # 08: To become familiar with Polymorphism, and Packages

**Instructor:** Engr. Zainab Umair Kamangar

<b>Lab Report Rubrics</b> (Add the points in each column, then add across the bottom row to find the total score)					<b>Total Marks</b>
S.No	Criterion	0.5	0.25	0.125	
1	Accuracy	<input type="checkbox"/> Desired output	<input type="checkbox"/> Minor mistakes	<input type="checkbox"/> Critical mistakes	
2	Timing	<input type="checkbox"/> Submitted within the given time	<input type="checkbox"/> 1 day late	<input type="checkbox"/> More than 1 day late	

#### Submission Profile

Name:

Submission date (dd/mm/yy):

Enrollment ID:

Receiving authority name and signature:

Comments:

\_\_\_\_\_  
Instructor Signature

**Note:** Submit this lab hand-out in the next lab with attached solved activities and exercises

## Objectives

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

- Polymorphism
- Packages

## 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.

**Real life example of polymorphism:** A person at the same time can have different characteristic. Like a woman at the same time is a mother, a daughter or an employee. So the same person posses different behavior in different situations. This is called polymorphism. In computer science, it describes the concept that objects of different types can be accessed through the same interface.

*Polymorphism in Java only occurs when there are one or more classes or objects related to each other by inheritance (method overriding).*

Java supports 2 types of polymorphism:

- static or compile-time
- dynamic or runtime polymorphism

### 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.

At compile time, Java knows which method to invoke by checking the method signatures. So, this is called compile time polymorphism or static binding.

```
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;  
    }  
}
```

```

        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 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.

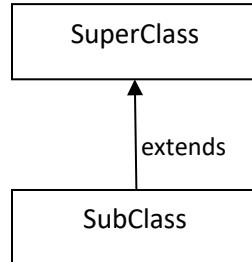
Within an inheritance hierarchy, a subclass can override a method of its superclass. That enables the developer of the subclass to customize or completely replace the behavior of that method.

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
```

```
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

```
// 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);
```

Aha!

```
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!*

## Package

A **package** is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of packages as being similar to different folders on your computer.

*You might keep HTML pages in one folder, images in another, and scripts or applications in yet another.*

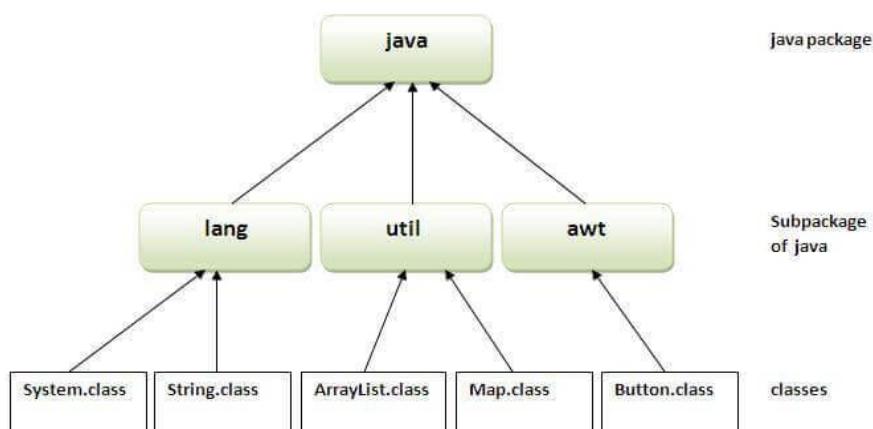
Because software written in the Java programming language can be composed of hundreds or thousands of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages.

The Java platform provides an enormous class library (a set of packages) suitable for use in your own applications. This library is known as the "Application Programming Interface", or "API" for short.

Package in java can be categorized in two form, built-in package and user-defined package. There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

### Advantages of using Java Packages:

1. **Better organization:** Java package is used to categorize the classes and interfaces so that they can be easily maintained. As in large java projects where we have several hundreds of classes, it is always required to group the similar types of classes in a meaningful package name so that you can organize your project better and when you need something you can quickly locate it and use it, which improves the efficiency.
2. **Reusability:** While developing a project in java, we often feel that there are few things that we are writing again and again in our code. Using packages, you can create such things in form of classes inside a package and whenever you need to perform that same task, just import that package and use the class.
3. **Name conflicts:** Java package removes naming collision. For example; We can define two classes with the same name in different packages so to avoid name collision, we can use packages



## Types of packages in Java

- *User defined package*: The package we create is called user-defined package.
- *Built-in package*: The already defined package like `java.io.*`, `java.lang.*` etc are known as built-in packages.

## Example

A class `Calculator` is created inside a package name `letmecalculate`. To create a class inside a package, declare the package name in the first statement in your program.

A class can have only one package declaration.

`Calculator.java` file created inside a package `letmecalculate`

```
package letmecalculate;

public class Calculator {
    public int add(int a, int b){
        return a+b;
    }
} // save this file as Calculator.java. This class is used to create a package as well
  as to perform addition
```

Now lets see how to use this package in another program.

```
import letmecalculate.Calculator;

public class Demo{
    public static void main(String args[]){
        Calculator obj = new Calculator();
        System.out.println(obj.add(100, 200));
    }
} // save this file as Demo.java
```

## Steps for compiling and running java package

1. Access path of the folder/directory where you have saved the `Calculator` file  
For example : `C:\Users\HP\Desktop\OOP`, it suggests that `Calculator` file is on desktop in `OOP` folder
2. `javac -d . Calculator.java` \\ compile the package
3. `javac Demo.java`
4. `java Demo`

## Access package from another package

There are three ways to access the package from outside the package.

1. Import `package.classname;`
2. Fully qualified name.
3. Import `package.*;`

## **1. Using package.classname**

If you import package.classname then only declared class of this package will be accessible.

### **Example:**

```
import letmecalculate.Calculator;
```

## **2. Using fully qualified name**

If you use fully qualified name then only declared class of this package will be accessible.

Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

**It is generally used when two packages have same class name** e.g. java.util and java.sql packages contain Date class.

### **Example:**

```
public class Demo{  
    public static void main(String args[]){  
        letmecalculate.Calculator obj = new letmecalculate.Calculator();  
        System.out.println(obj.add(100, 200));  
    }  
}
```

## **3. Using package.\***

If you use package.\* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package

### **Example:**

```
Import java.util.*;
```

## Exercises

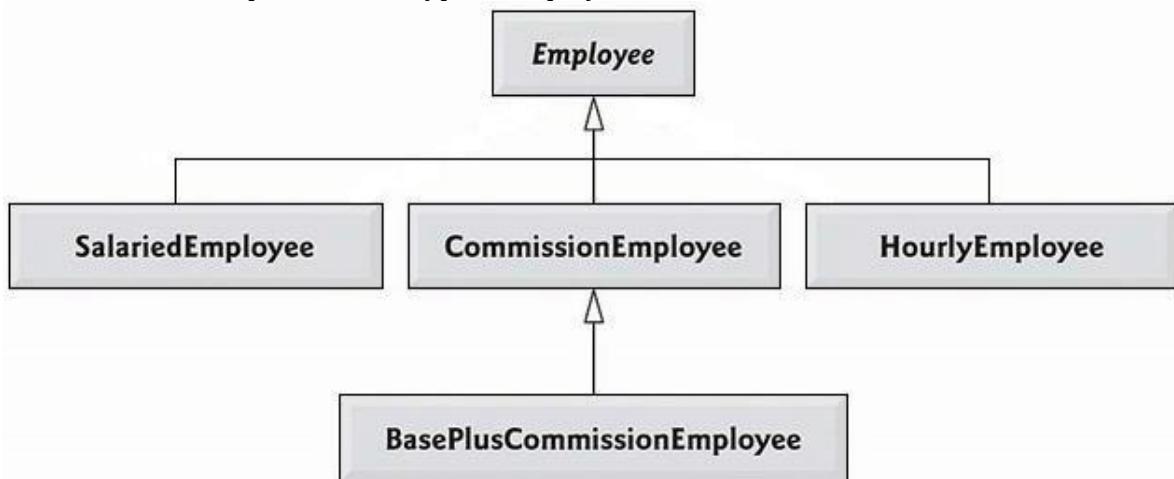
### Question: 1 (a)

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 known about each employee is his/her first name, last name and national identity card number. The rest 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(); }
```

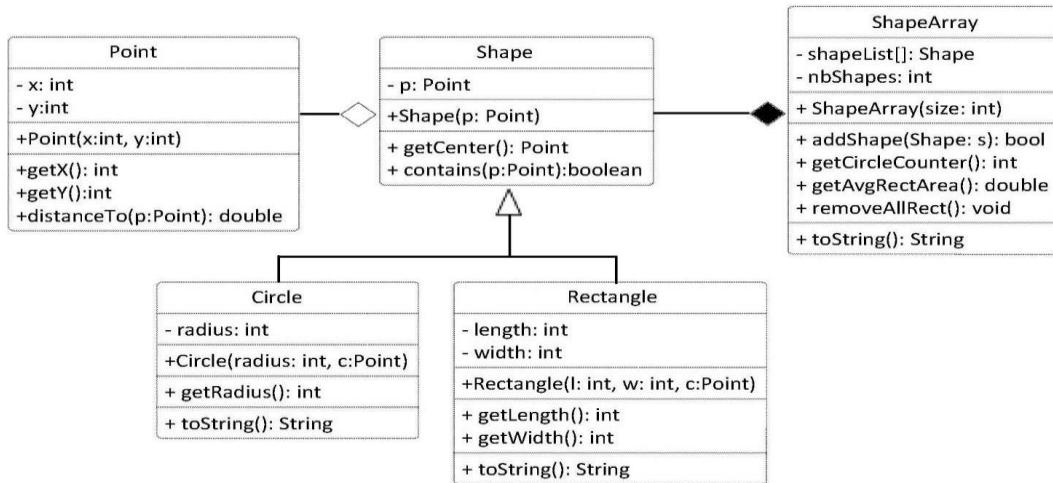
### Question: 1 (b)

#### 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
```

## Question: 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

### Question: 2 (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

### Question: 3

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.

### Question: 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**)

### **Question: 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.

### **Question: 6 (Packages)**

Your creativity

1. Create three packages, think of them yourself
2. Put two different classes in each package
3. Import all three of these packages in class named `PackagePractice`, you will have access to 6 classes
4. Call methods of these 6 classes and use them in `PackagePractice`