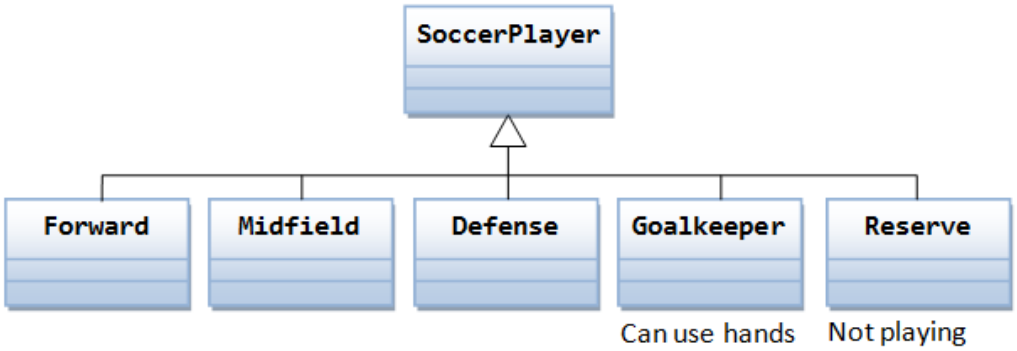
BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
CS F213
LAB-06 [Inheritance, Polymorphism, and Abstract Classes]

AGENDA **DATE: 01/10/2020** **TIME: 02 Hours**

1. Inheritance
2. Polymorphism (method overriding and method overloading, Overriding the toString() method)
3. Abstract Class

1 Inheritance

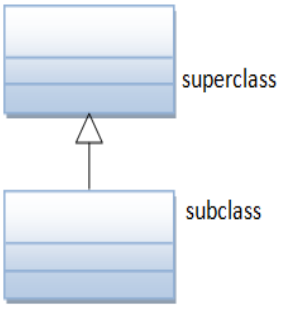
In OOP, we often organize classes in *hierarchy* to *avoid duplication and reduce redundancy*. The classes in the lower hierarchy inherit all the state variables and methods from the higher hierarchies. A class in the lower hierarchy is called a *subclass* (or *derived*, *child*, *extended class*). A class in the upper hierarchy is called a *superclass* (or *base*, *parent class*). For example,



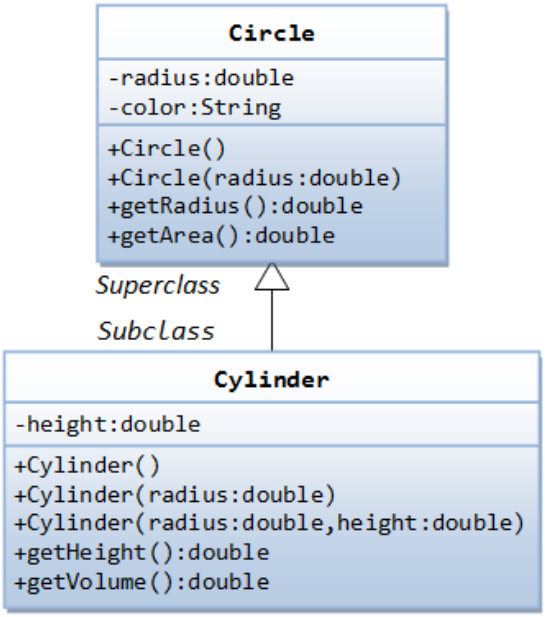
In Java, you define a subclass using the keyword "extends", e.g.,

```
class Cylinder extends Circle {......}
class Goalkeeper extends SoccerPlayer {......}
```

UML Notation: The UML notation for inheritance is a solid line with a hollow arrowhead leading from the subclass to its superclass. By convention, superclass is drawn on top of its subclasses as shown.



1.1 An Example on Inheritance



In this example, we derive a subclass called Cylinder from the superclass Circle. The class Cylinder inherits all the member variables (radius and color) and methods (getRadius(), getArea(), among others) from its superclass Circle. It further defines a variable called height, two public methods - getHeight() and getVolume() and its own constructors.

The Source code as follows:

`Circle.java`

```
1// Define the Circle class
2public class Circle{  // Save as "Circle.java"
3// Private variables
4   private double radius;
5   private String color;
6
7// Constructors (overloaded)
8   public Circle() {                   // 1st Constructor
9      radius = 1.0;
10     color = "red";
11  }
12  public Circle(double r) {           // 2nd Constructor
13     radius = r;
14     color = "red";
15  }
16  public Circle(double r, String c) { // 3rd Constructor
17     radius = r;
18     color = c;
19  }
20
```

```
21// Public methods
22   public double getRadius() {
23      return radius;
24   }
25   public String getColor() {
26      return color;
27   }
28   public double getArea() {
29      return radius*radius*Math.PI;
30   }
  }
```

Cylinder.java

```
 1 // Define Cylinder class, which is a subclass of Circle
 2 public class Cylinder extends Circle {
 3   private double height;   // Private member variable
 4
 5   public Cylinder() {      // constructor 1
 6   super();              // invoke superclass' constructor Circle()
 7      height = 1.0;
 8   }
 9   public Cylinder(double radius, double height) {  // Constructor 2
10      super(radius);   // invoke superclass' constructor Circle(radius)
11      this.height = height;
12   }
13
14   public double getHeight() {
15      return height;
16   }
17   public void setHeight(double height) {
18      this.height = height;
19   }
20   public double getVolume() {
21      return getArea()*height;   // Use Circle's getArea()
22   }
23}
```

A Test Drive Program: TestCylinder.java

```
 1 // A test driver program for Cylinder class
 2 public class TestCylinder {
 3   public static void main(String[] args) {
 4      Cylinder cy1 = new Cylinder();         // Use constructor 1
 5      System.out.println("Radius is " + cy1.getRadius()
 6         + " Height is " + cy1.getHeight()
 7         + " Color is " + cy1.getColor()
 8         + " Base area is " + cy1.getArea()
 9         + " Volume is " + cy1.getVolume());
10      Cylinder cy2 = new Cylinder(5.0, 2.0); // Use constructor 2
```

```
11        System.out.println("Radius is " + cy2.getRadius()
12         + " Height is " + cy2.getHeight()
13         + " Color is " + cy2.getColor()
14         + " Base area is " + cy2.getArea()
15         + " Volume is " + cy2.getVolume());
16   }
17}
```

Keep the "Cylinder.java" and "TestCylinder.java" in the same directory as "Circle.class" (because we are reusing the class Circle). Compile and run the program. The expected output is as follows:

```
Radius is 1.0 Height is 1.0 Color is red Base area is 3.141592653589793 Volume is 3.141592653589793
Radius is 5.0 Height is 2.0 Color is red Base area is 78.53981633974483 Volume is 157.07963267948966
```

1.2 Exercise -

Compile and Execute the following code by completing it as per commented specification given. Write the whole code in file

Ex3Test.java

```
class A { public int a =100; } // End of class A
class B extends A { public int a =80; } // End of class B
class C extends B { public int a =60; } // End of class C
class D extends C { public int a =40; } // End of class D
```

// NOTE : The variable named 'a' used in above classes is the instance field of each class

```
class E extends D{
        public int a =10;
        public void show(){
                int a =0;
                // Write Java statements to display the values of
                // all a's used in this file on System.out
        } // End of show() Method
}// End of class E
class Ex3Test{
        public static void main(String args[]){
                new E().show(); // This is an example of anonymous object
                A a1 = new E();
                D d1 = (D) a1; // what's wrong with this statement?
        }// End of main()
}// End of class EX3Test
```

Upcasting:
((A) this).a;

2. Polymorphism

Polymorphism means—One name many form.

Two ways by which java implements Polymorphism:

- ✓ **Compile time: Overloading. (The discussion on polymorphism in the class pertains here)**
- ✓ **Run time: Overriding.**

2.1 Method Overloading:

In same class, if name of the method remains common but the number and type of parameters are different, then it is called method overloading in Java.

Overloaded methods:

- ✓ appear in the same class or a subclass
- ✓ have the **same name** but,
- ✓ have different **parameter lists**, and,
- ✓ can have different **return types**

Constructor Overloading:

- ✓ Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists.
- ✓ The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

2.1.1 Example for Method Overloading:

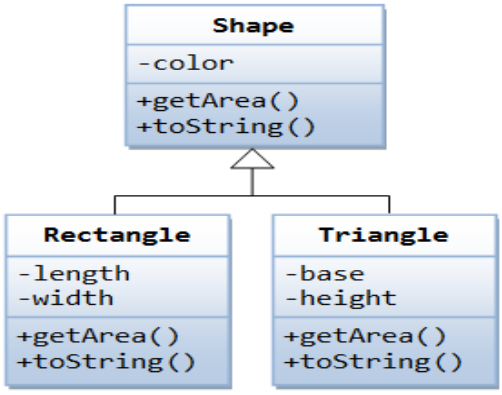
```
class Room{
        double length, breadth, height;

        Room(){//default constructor for class Room
                length=-1;
                breadth=-1;
                height=-1;
        }

        //overloading the constructor
        //3 Parameterized constructor for the class Room
        Room(double l, double b, double h)       {
                length=l;
                breadth=b;
                height=h;
        }

        Room(double len) {// Single parameterized constructor
                length=breadth=height=len;
        }

        double volume()  {
                return length*breadth*height;
        }
}
```

```
// Demonstrating the use of Overloaded constructors
class OverloadConstructors{
        public static void main(String args[]){
                Room a=new Room(20,30,40);
                Room b=new Room();
                Room c=new Room(10);
                double vol;
                vol=a.volume();
                System.out.println("Volume of room a is " + vol);

                vol=b.volume();
                System.out.println("Volume of room b is " + vol);

                vol=c.volume();
                System.out.println("Volume of room c is " + vol);
        }
}
```

2.2 Method Overriding:

- ✓ applies **ONLY** to inherited methods is related to polymorphism
- ✓ object type (**NOT** reference variable type) determines which overridden method will be used at runtime
- ✓ overriding method **MUST** have the same argument list (if not, it might be a case of overloading)
- ✓ overriding method **MUST** have the same return type; the exception is *covariant return* (used as of Java 5) which returns a type that is a subclass of what is returned by the overridden method
- ✓ overriding method **MUST NOT** have more restrictive access modifier, but **MAY** have less restrictive one
- ✓ overriding method **MUST NOT** throw new or broader checked exceptions, but **MAY** throw fewer or narrower checked exceptions or any unchecked exceptions
- ✓ abstract methods **MUST** be overridden
- ✓ final methods **CANNOT** be overridden
- ✓ static methods **CANNOT** be overridden
- ✓ constructors **CANNOT** be overridden

2.2.1 Example for Method Overriding:



Suppose that our program uses many kinds of shapes, such as triangle, rectangle and so on. We should design a superclass called Shape, which defines the public interface (or behaviors) of all the shapes. For example, we would like all the shapes to have a method called getArea(), which returns the area of that particular shape. The Shape class can be written as follow.

Shape.java

```
// Define superclass Shape
public class Shape {
// Private member variable
   private String color;

// Constructor
   public Shape (String color) {
      this.color = color;
   }

   @Override
   public String toString() {
      return "Shape of color=\"" + color + "\"";
   }

// All shapes must has a method called getArea()
   public double getArea() {
      System.err.println("Shape unknown! Cannot compute area!");
      return 0;   // Need a return to compile the program
   }
}
```

We can then derive subclasses, such as Triangle and Rectangle, from the superclass Shape.

Rectangle.java

```
// Define Rectangle, subclass of Shape
public class Rectangle extends Shape {
// Private member variables
   private int length;
   private int width;

// Constructor
   public Rectangle(String color, int length, int width) {
      super(color);
      this.length = length;
      this.width = width;
   }

   @Override
   public String toString() {
      return "Rectangle of length=" + length + " and width=" + width + ", subclass of
                        " + super.toString();}

   @Override
   public double getArea() { return length*width; }
}
```

Triangle.java

```
// Define Triangle, subclass of Shape
public class Triangle extends Shape {
// Private member variables
   private int base;
   private int height;

// Constructor
   public Triangle(String color, int base, int height) {
      super(color);
      this.base = base;
      this.height = height;
   }

   @Override
   public String toString() {
      return "Triangle of base=" + base + " and height=" + height + ", subclass of "
                                                          + super.toString();
   }

   @Override
   public double getArea() {return 0.5*base*height;}
}
```

The subclasses override the getArea() method inherited from the superclass, and provide the proper implementations for getArea().

TestShape.java

In our application, we could create references of Shape, and assigned them instances of subclasses, as follows:

```
// A test driver program for Shape and its subclasses
public class TestShape {
   public static void main(String[] args) {
      Shape s1 = new Rectangle("red", 4, 5);
      System.out.println(s1);
      System.out.println("Area is " + s1.getArea());
      Shape s2 = new Triangle("blue", 4, 5);
      System.out.println(s2);
      System.out.println("Area is " + s2.getArea());
   }
}
```

2.3 Overriding the toString():

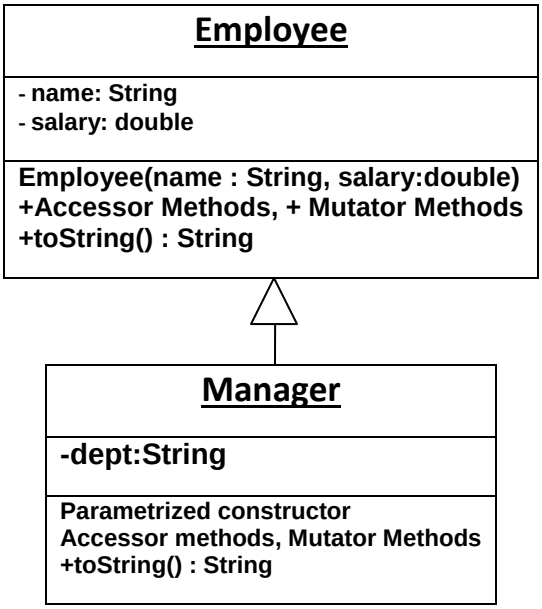
```
/**
 * Java program to demonstrate How to override toString() method in Java.
 * This Java program shows How can you use IDE like Netbeans or Eclipse
 * override toString in Java.
 */

public class Country{
    private String name;
    private String capital;
    private long population;

    public Country(String name){
        this.name = name;
    }

    public String getName(){ return name; }
    public void setName(String name) {this.name = name;}

    public String getCapital() {return capital;}
    public void setCapital(String capital) {this.capital = capital;}

    public long getPopulation() { return population; }
    public void setPopulation(long population) {this.population = population; }

    @Override
    public String toString() {
        return "Country [name="+name + "capital=" + capital + ",
               population=" + population + "]";
    }

   public static void main(String args[]){
            Country India = new Country("India");
            India.setCapital("New Delhi");
            India.setPopulation(1200000000);
            System.out.println(India);
   }
}
```

2.4 Exercise –

Make a class ***Employee*** with attributes
– ***name:String***
– ***salary: double***.
This class supplies
(i) A parameterized constructor
(ii) Accessor and Mutator method(s) for every instance field and
(iii) toString() method which returns the values of instance fields by adding proper heading labels and spaces.

Make a class **Manager** that inherits from Employee and add an instance field named – ***department:String.*** This class also supplies parameterized constructor, accessor and mutator methods and a toString() method that returns the manager's name, department, and salary by adding proper labels and spaces.

The complete UML class diagram representation for these classes is shown below:



You have to write the code as per following description:

1. Write java implementations for classes Employee and Manager.
2. Write a Driver code which creates two Employee and Manager instances each and display their attribute values onstandard output using polymorphism.

3. Abstract Class:

- ✓ An abstract method is a method with only signature (i.e., the method name, the list of arguments and the return type) without implementation (i.e., the method's body).
- ✓ You use the keyword abstract to declare an abstract method.
- ✓ A class containing one or more abstract methods is called an abstract class.
- ✓ An abstract class must be declared with a class-modifier abstract.

3.1 Example for Abstract Class:

Rewrite the Shape class as an `abstract` class, containing an `abstract` method `getArea()` as follows:

Shape.java

```
abstract public class Shape {
// Private member variable
   private String color;

// Constructor
   public Shape (String color) {
      this.color = color;
   }

   @Override
   public String toString() {
      return "Shape of color=\"" + color + "\"";
   }

// All Shape subclasses must implement a method called getArea()
abstract public double getArea();
}
```

Now create instances of the subclasses such as Triangle and Rectangle (Classes which we used previously), and upcast them to Shape, (you cannot create instance of Shape).

```
public class TestShape {
   public static void main(String[] args) {
      Shape s1 = new Rectangle("red", 4, 5);
      System.out.println(s1);
      System.out.println("Area is " + s1.getArea());

      Shape s2 = new Triangle("blue", 4, 5);
      System.out.println(s2);
      System.out.println("Area is " + s2.getArea());

      // Cannot create instance of an abstract class
      Shape s3 = new Shape("green");// Compilation Error!!
   }
}
```

3.2 Exercise –

Define an abstract class ***Worker*** that has an abstract method ***public double computePay()***. Every worker has a ***name*** and a ***salary_rate***. Define two concrete classes ***FullTimeWorker***, and ***HourlyWorker***. A full time worker gets paid the hourly wage for a maximum of240 hours in a month at the rate of Rs. 100/hour. An hourly worker gets paid the hourly wage for the actual number of hours he has worked at the rate of Rs. 50/hour, he is not allowed to work for more than 60 hours in a month. The complete UML class diagram shown below:

You have to write the code as per following specification:

1. Write the java implementations for classes ***Worker***, ***HourlyWorker*** and ***FullTimeWorker***
2. Write a Driver code in the same file ***TestWorker.java***which demonstrates late binding.