Lecture 12

Superclasses, Subclasses, the word super

Example: A point

\((x, y), x = an\ integer, \ y = an\ integer\)

can be considered as a class. A circle with center \((x, y)\) and radius \(r\)

\((x,y,r), x = an\ integer, \ y = an\ integer, \ r = the\ radius\)

can be considered as an "extension of" a point.

In java, the class for a point could be defined by

```java
public class Point
{
    protected int x, y; //1
    public Point() // 2
    {
        setPoint(0,0);
    }
    public Point(int a, int b) //3
    {
        setPoint(a,b);
    }
    public void setPoint(int a, int b)
    {
        x=a; y=b;
    }
    public int getX()
    {
        return x;
    }
    public int getY()
    {
        return y;
    }
    public String toString()
    {
        return "[" + x + ", " + y + "]";
    }
} // end class Point
```
Comments:

// 1 A protected variable is one that can be accessed only by a class and its subclasses (It can also be accessed by classes in the same package, but we have yet to discuss packages). It is a level of protection between public and private: it is more restrictive than public, less restrictive than private.

// 2 This is a "no-argument" constructor for Point

// 3 This is constructor for Point with two arguments

The class Point does not do much. If we test it out with

```java
public class Test1
{
    public static void main(String args[])
    {
        Point p = new Point();
        Point q = new Point(4,7);
        String blank ="   ";
        System.out.println("\n +blank+ p.toString());
        System.out.println(blank+ q.toString());
    }
} // end class Test1

we get
```

We turn to the problem of constructing a class called Circle, which will be an extension of the class Point. The syntax is
public class Circle extends Point {
    protected double radius;

    public Circle() {
        setRadius(0);
    }

    public Circle(double r, int a, int b) {
        super(a, b); // must come first
        setRadius(r);
    }

    public void setRadius(double r) {
        radius = (r >= 0 ? r : 0.0);
    }

    public double getRadius() {
        return radius;
    }

    public double area() {
        return Math.PI * radius * radius;
    }

    public String toString() {
        return "Center = " +
                "[" +x + ", " +y + "]" +
                "  Radius = " + radius;
    }
}

Some terminology: The class Circle is said to be a subclass of Point. The class Point is the superclass of Circle.

Observe: First, there is no mention in the class Circle of the variables x and y, the variables that will form the center of the circle. They will be "inherited" from the class Point.

The inheritance works this way. The no-parameter constructor of Circle (top right) will implicitly call the no-parameter constructor of its superclass Point and set the center to be (0,0).

The second constructor of Circle will call the second constructor of Point, by means of the "super(a,b)" in the bottom right panel, and set the center to be (a,b).
The class `Point` has two constructors. A "no-parameter" one:

```java
public Point()
{
    setPoint(0,0);
}
```

A two-parameter one that sets the point at \((a,b)\):

```java
public Point(int a, int b)
{
    setPoint(a,b);
}
```

The program makes these decisions about what corresponds to what by comparing the parameter lists of the constructors. The no-parameter constructor of `Circle` calls on the no-parameter constructor of `Point` to determine the center. The three-parameter constructor of `Circle` calls, by means of the `super(a,b)` instruction, the two-parameter constructor of `Point` to determine the center.

To continue, variables \(x\) and \(y\) are not explicitly present in the definition of the class `Circle`. They are however part of the definitions of the circle, and this is done by the calls of `super()`. The call is implicit in the no-parameter case; explicit in the three-parameter case.

We test what we have with a simple application:

```java
public class Test2
{
    public static void main(String args[])
    {
        Circle c, d;
        c = new Circle();
        d = new Circle(3.7, 5, 8);
        System.out.println("\n For circle c:" + c.toString());
        System.out.println(" For circle d:" + d.toString());
    }
}
```

This is the output.
More on "super" and the correspondence of constructors

On page 3, toward the bottom, I remarked that when the compiler reaches no-parameter constructor of Circle it makes an implicit call on the no-parameter constructor of Point. To see this specifically, comment out the no-parameter constructor in Point:

```java
public class Point {
    protected int x, y;
    //public Point(){ setPoint(0,0);}
    public Point(int a, int b){ setPoint(a,b);}
    public void setPoint(int a, int b) { x = a; y = b; }
    public int getX() {return x;}
    public int getY() {return y;}
    public String toString(){return "[ " + x + ", " + y + "]";}
}
``` // end class Point

Then, without any further changes, run the Test2 application. An error occurs, for the method Point() cannot be found

A similar error occurs if the no-parameter constructor is put back and the two-parameter version is commented out. In sum, each of the constructors in the subclass must have a corresponding constructor in the super class

More on "protected"

If the variables of a superclass are declared to be private they cannot be accessed by the subclass. To see an example of this behavior replace the protected variables, x and y, by private ones in the Point class:

```java
public class Point
```
{  
    private int x, y;

    public Point(){ setPoint(0,0);}
    public Point(int a, int b){ setPoint(a,b);}
    public void setPoint(int a, int b) { x = a; y = b;}
    public in getX() {return x;}
    public in getY() {return y;}
    public String toString(){return "[ " + x + ", "+ y + "]";}
}
// end class Point

Save this and then compile the Circle.java class. The compiler then produces an error message, at the toString() method of the Circle class:

![Command Prompt](image)

2:

"2:

Z:\11Pic20\Lecture_12>javac Circle.java
Circle.java:33: Undefined variable: x
    return "Center = " +"["+x +"," +y +"]" +
Circle.java:33: Undefined variable: y
    return "Center = " +"["+x +"," +y +"]" +
2 errors

Inheritance

We shall use the previously defined superclass Point and its extensions to demonstrate the effects of resetting the values of variables in classes and subclasses.

import javax.swing.*;
public class Test3
{
    public static void main(String args[])
    {
        Point p = new Point(72, 85);
        String out;
        out = " x coordinate is "+p.getX();
        out += "\n y coordinate is " + p.getY();

        p.setPoint(-10, 12);
        out +="\n new location of p is "+ p;
        JOptionPane.showMessageDialog(null, out);
        System.exit(0);
    }  // end main()

In this example we reset the value of point to (-10, 12). The output is:
Thus redefining `p.setPoint(-10, 12);` leads to a new definition of point.

Going on to the extension Circle, we create one then reset its radius and center.

```java
import java.text.*;

public class Test4
{
    public static void main(String args[])
    {
        Circle c = new Circle(2.5, 15, 22);

        DecimalFormat precision2;
        precision2 = new DecimalFormat("0.00");

        String out;

        out = "x coordinate = " + c.getX();
        out += 
        y coordinate = " + c.getY();
        out += 
        Radius = " + c.getRadius();

        c.setRadius(3.5);
        c.setPoint(-7, 18);

        out += 
        new location, new radius of c is ";
        out += 
        Area of c is ";
        out += precision2.format(c.area());

        JOptionPane.showMessageDialog(null, out);
        System.exit(0);
    }
}
```

Once again the new values are carried over to the constructor for the circle:
Finally, we extend Circle to a Cylinder and play the same game again:

```java
public class Cylinder extends Circle {
    protected double height;

    // no-argument constructor
    public Cylinder() {
        setHeight(0);
    }

    // four-argument constructor
    public Cylinder(double h, double r, int a, int b) {
        super(r, a, b); // super must come first
        setHeight(h);
    }

    public void setHeight(double h) {
        height = (h >= 0 ? h : 0);
    }

    public double getHeight() {
        return height;
    }

    public double area() {
        return 2*super.area() + 2*Math.PI*height*radius;
    }

    public double volume() {
        return super.area()*height;
    }
}
```
public String toString()
{
    return super.toString() +
            ", Height = " + height;
}
//class cylinder

Testing this out, we have

import javax.swing.*;
import java.text.*;

public class Test5
{
    public static void main(String args[])
    {
        Cylinder c =
                new Cylinder(5.7, 2.4, 2, 5);

        DecimalFormat dos =
                new DecimalFormat("0.00");

        String out;

        out = " x coordinate = ";
        out += c.getX()+"\n";

        out += " y coordinate = ";
        out += c.getY()+"\n";

        out += " Radius = ";
        out += c.getRadius()+ "\n";

        out += " Height = " ;
        out += c.getHeight()+ "\n\n";

        c.setHeight(20);
        c.setRadius(3.5);
        c.setPoint(-7, 18);

        out += " New location, radius and";
        out += " height of c are: \n";
        out += " " + c + "\n";
        out += " area = " +
                dos.format(c.area()) + "\n";
        out += " volume = " +
                dos.format(c.volume())+ "\n";

        JOptionPane.
                showMessageDialog(null, out);
System.exit(0);

} //end main(String args[])

} // end class Test5

This produces another demonstration of inheritance:

"final"

The qualifier final can be applied to variables, methods, and classes.

When applied to a variable it means that the value of that variable cannot be changed by the program. Typical cases might be

    final double TAX = 0.085;
    final int WIN = 0, LOSE = 1;

The value of a final must be stated when it is declared;

A final method is one that cannot be overwritten in a subclass.

A final class is one that cannot be extended. We shall be working with final classes in the next example.
Polymorphism, Abstract Classes

We first will work through an example of the ideas associated with these words. To start we have an abstract class called Employee.

```java
public abstract class Employee {
    private String firstName;
    private String lastName;

    public Employee(String first, String last) {
        firstName = first;
        lastName = last;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

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

    public abstract double earnings();
}
```

This class will be extended to several subclasses such as Boss, CommissionWorker, PieceWorker, and HourlyWorker. Each class will be the same in that it will correspond to one worker who has a first and last name, and there will a method toString() that returns these names.

The extensions will be different in the way the abstract method earnings() is implemented for each type of employee. Note that this method is empty at this point.
Let us go on to the Boss. It is a \textit{final} class, so cannot be extended. It has one variable \texttt{weeklySalary}, which is private.

```java
public final class Boss extends Employee {
    private double weeklySalary;

    public Boss(String first, String last, double s) {
        super(first, last);
        setWeeklySalary(s);
    }

    public void setWeeklySalary(double s) {
        weeklySalary = (s >= 0 ? s : 0);
    }

    public double earnings() {
        return weeklySalary;
    }

    public String toString() {
        return " Boss: " + super.toString();
    }

}// end class Boss
```

Observe that the abstract class \texttt{earnings()} has become concrete. The details of compensating the boss have been spelled out.
We go on to the class `CommissionWorker`. This worker's salary is based on a salary plus a commission on the quantity of elements sold. As before, we have to spell out the earnings() method.

```java
public final class CommissionWorker
    extends Employee
{
    private double salary;
    private double commission;
    private int quantity;

    public CommissionWorker(
        String first, String last,
        double s, double c, int q)
    {
        super(first, last);
        setSalary(s);
        setCommission(c);
        setQuantity(q);
    }

    public void setSalary(double s)
    {
        salary = (s >= 0? s:0);
    }

    public void setCommission(double c)
    {
        commission = (c >= 0? c:0);
    }

    public void setQuantity(int q)
    {
        quantity = (q>=0?q:0);
    }

    public double earnings()
    {
        return salary +
                commission*quantity;
    }

    public String toString()
    {
        return " Commission Worker: "+
                super.toString();
    }
}
// end class CommissionWorker
```
The PieceWorker’s earnings are based solely on a flat rate per item produced.

```java
public final class PieceWorker
    extends Employee {
    private double wagePerPiece;
    private int quantity;

    public PieceWorker(
            String first, String last,
            double w, int q)
    {
        super(first, last);
        setWage(w);
        setQuantity(q);
    }

    public void setWage(double w)
    {
        wagePerPiece = (w > 0?w:0);
    }

    public void setQuantity( int q )
    {
        quantity = (q > 0?q:0);
    }

    public double earnings()
    {
        return quantity*wagePerPiece;
    }

    public String toString()
    {
        return "Piece worker: " +
                super.toString();
    }
}
```
The last worker is an HourlyWorker. The pay for this type of worker is an hourly wage multiplied by the number of hours worked.

```java
public final class HourlyWorker
    extends Employee
{
    private double wage;
    public double hours;

    public HourlyWorker(
        String first, String last,
        double w, double h)
    {
        super(first, last);
        setWage(w);
        setHours(h);
    }

    public void setWage(double w)
    {
        wage = (w > 0?w:0);
    }

    public void setHours(double h)
    {
        hours = ((h >0&& h< 168)? h : 0);
    }

    public double earnings()
    {
        return wage*hours;
    }

    public String toString()
    {
        return "Hourly worker: " +
            super.toString();
    }
}
```

We test this out the whole system with

```java
import javax.swing.*;
import java.text.*;

public class Test
{
    public static void main(String args[])
    {
        Employee ref;
        String out = "";

        Boss b =
        new Boss(" John", "Rigo ", 900.00);

        CommissionWorker c =
        new CommissionWorker(
            " Elizabeth", "Segunda ",
            450, 3.25, 230);

        PieceWorker p =
        new PieceWorker(
            " Alberta", "Wist", 3, 120);

        HourlyWorker h =
        new HourlyWorker(
            " Karen", "Kart ", 15.87, 40);

        DecimalFormat dos =
        new DecimalFormat("0.00");

        ref = b;  //b= boss
        out += ref.toString() +
            " earned $" +
            dos.format(ref.earnings()) +
            "\n";

        ref = c;
        out += ref.toString() +
            " earned $" +
            dos.format(ref.earnings()) +
            "\n";

        ref = p;
        out += ref.toString() +
            " earned $" +
            dos.format(ref.earnings()) +
            "\n";

        ref = h;
        out += ref.toString() +
            " earned $" +
            dos.format(ref.earnings()) +
            "\n";

        JOptionPane.
            showMessageDialog(null, out);
```
System.exit(0);
}
} // end class Test

The output is:

![Image of message box with earned amounts]