Turtle (in Hebrew)

Software in the Java language

Exercise 9:
The program is adapted to:

At the School of Computer Science,
Tel Aviv University

Course: Computer Science

Exercise 9:

Today we showed two ways to reuse existing code in an existing class:

- Inclusion
- Inheritance

The inheriting class inherits all the fields and methods of the base class, as well as the protected methods of the base class.

In the example below, we inherit from the class Turtle, which we saw at the beginning of the semester, and add a new method:

```java
public class SmartTurtle extends Turtle {
    public void drawSquare(int edge) {
        for (int i = 0; i < 4; i++) {
            moveForward(edge);
            turnLeft(90);
        }
    }
}
```

Inheritance of a class

Inheritance of a class usually represents a subset of the objects of the base class.

The inheriting class can override the services inherited:

In order to use the original service (for example, to use the moveForward service on its own)
we can call the service in the following way:

```java
super.methodName(...)
```

In the example below, we define a DrunkTurtle class that inherits from the Turtle class and overrides the moveForward service:

```java
public class DrunkTurtle extends Turtle {
    /**
     * Zigzag forward a specified number of units. At each step
     * the turtle may make a turn of up to 30 degrees.
     * @param units - number of steps to take
     */
    @Override public void moveForward(double units) {
        for (int i = 0; i < units; i++) {
            if (Math.random() < 0.1) {
                turnLeft((int) (Math.random() * 60 - 30));
            }
            super.moveForward(1);
        }
    }
}
```

Inheritance of fields and services of the base class:

- Fields and services are not accessible in the inheriting class.
- In order to allow access to the inheriting classes, we must define the fields as protected.
- Using inheritance should be done with caution, especially when dealing with access hierarchies.
- We use protected only when we plan to create a full inheritance hierarchy and control the class.
At the customer side:
- We introduced the interface `IPoint`, and presented three different implementations for it.
- We showed that customers relying on the interface `IPoint` need only, and do not recognize the implementing classes.
- The use of interfaces saves doubled customer code, since the same code block works correctly with various suppliers (polymorphism).

```java
interface IPoint {
    double x();
    double y();
    double rho();
    double theta();
}
```

```
public class CartesianPoint {
    private double x;
    private double y;
    public CartesianPoint(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public double x() { return x; }
    public double y() { return y; }
    public double rho() { return Math.sqrt(x*x + y*y); }
    public double theta() { return Math.atan2(y,x); }
}
```

```
public class PolarPoint {
    private double r;
    private double theta;
    public PolarPoint(double r, double theta) {
        this.r = r;
        this.theta = theta;
    }
    public double x() { return r * Math.cos(theta); }
    public double y() { return r * Math.sin(theta); }
    public double rho() { return r; }
    public double theta() { return theta; }
}
```

```
private double r;
private double theta;
public SmartPoint(double r, double theta) {
    this.r = r;
    this.theta = theta;
}
```

```
public void rotate(double angle) {
    double currentTheta = Math.atan2(y,x);
    double currentRho = rho();
    x = currentRho * Math.cos(currentTheta+angle);
    y = currentRho * Math.sin(currentTheta+angle);
}
```

```
public void translate(double dx, double dy) {
    x += dx;
    y += dy;
}
```

```
public void rotate(double angle) {
    theta += angle;
}
```

```
public void translate(double dx, double dy) {
    double newX = x() + dx;
    double newY = y() + dy;
    r = Math.sqrt(newX*newX + newY*newY);
    theta = Math.atan2(newY,newX);
}
```

It is difficult to see similarity between implementations in this case. Both basic methods are related to the selected representation for fields.

At the supplier side:
- Instead, in the supplier, the interface `IPoint` saves doubled supplier code.
- "IPoint" receives a class with code by inheritance instead of repeating it. The two suppliers share the same code.
- We try to identify similarity in code between three implementations and centralize these segments in a basic class that will be common to all implementations.

```
interface IPoint {
    public abstract class AbstPoint {
        public abstract void g();
    }
}
```

```
public abstract class A {
    public void f() {
        System.out.println("A.f!!");
    }
    abstract public void g();
}
```

```
public class B extends A {
    public void g() {
        System.out.println("B.g!!");
    }
}
```

```
A a = new A();
public class B extends A {
    public void g() {
        System.out.println("B.g!!");
    }
}
```

```
A a = new B();
```

```
private double x;
private double y;
public CartesianPoint(double x, double y) {
    this.x = x;
    this.y = y;
}
```

```
public double x() {
    return x;
}
public double y() {
    return y;
}
public double rho() {
    return Math.sqrt(x*x + y*y); }
public double theta() {
    return Math.atan2(y,x); }
```

```
private double r;
private double theta;
public PolarPoint(double r, double theta) {
    this.r = r;
    this.theta = theta;
}
```

```
public double x() { return r * Math.cos(theta); }
public double y() { return r * Math.sin(theta); }
public double rho() { return r; }
public double theta() { return theta; }
```

```
private double r;
private double theta;
public SmartPoint(double r, double theta) {
    this.r = r;
    this.theta = theta;
}
```

```
public void rotate(double angle) {
    double currentTheta = Math.atan2(y,x);
    double currentRho = rho();
    x = currentRho * Math.cos(currentTheta+angle);
    y = currentRho * Math.sin(currentTheta+angle);
}
```

```
public void translate(double dx, double dy) {
    x += dx;
    y += dy;
}
```

```
public void rotate(double angle) {
    theta += angle;
}
```

```
public void translate(double dx, double dy) {
    double newX = x() + dx;
    double newY = y() + dy;
    r = Math.sqrt(newX*newX + newY*newY);
    theta = Math.atan2(newY,newX);
}
```

It is difficult to see similarity between implementations in this case, they are related to the selected representation for fields.
Method Overloading & Overriding

public class A {
    public float foo(float a, float b) throws IOException{…}
}

public class B extends A {
    …
}

Which of the following methods can be defined in B:

1. float foo(float a, float b){…}
2. public int foo(int a, int b) throws Exception{…}
3. public float foo(float a, float b) throws Exception{…}
4. public float foo(float p, float q){…}

✓ ✓
Method Overriding

```java
public class A {
    public void print() {
        System.out.println("A");
    }
}

public class B extends A {
    public void print() {
        System.out.println("B");
    }
}

public class C {
    public static void main(...) {
        B b = new B();
        A a = b;
        b.print();
        a.print();
    }
}
```

What is the output?
The output is:

```
B
B
```