Turtle class in Java

Exercise 8: Erasure

Dr. Robi is asked and when he

the School of Computer Science

the University of Tel Aviv

three existing classes.

in the following two ways to reuse existing class

- Inheritance
- Interface

the inheriting class can add new functionality that

wasn't there in the base class, or change functionality

we received.

Dr. Robi in the class

we define a drunk turtle which is a turtle that "staggers" as it moves forward

/**
 * A drunk turtle is a turtle that "stagger" as it moves forward
 */

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);
        }
    }
}
The lectures were given at the Technion in the field of software.

- In the lectures, we introduced the concept of \( IPoint \), and demonstrated various implementations for it.
- It was shown that clients dependent on \( IPoint \) value only, and do not recognize the actual classes implementing it.
- It was demonstrated that using \( IPoint \) saves a double code in the client, since the same code block works correctly with various suppliers (polymorphism).

\[ \langle \text{interface} \rangle \text{IPoint} \langle \text{class} \rangle \]

\[ \text{CartesianPoint} \]
\[ \text{PolarPoint} \]
\[ \text{SmartPoint} \]

- On the supplier side, however, an inheritance mechanism saves a double code in the supplier.
- Instead of reviewing the code of each supplier, we try to identify the common code between the implementations and centralize it in a base class that is shared by all three implementations.

\[ \langle \text{interface} \rangle \text{IPoint} \langle \text{abstract} \rangle \text{AbstPoint} \langle \text{class} \rangle \]

\[ \text{CartesianPoint} \]
\[ \text{PolarPoint} \]
\[ \text{SmartPoint} \]

- Abstract classes - example

- Simple class:

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

- CartesianPoint - PolarPoint

It is difficult to see a similarity between implementations in this case. All four basic methods are closely related to the representation chosen for the fields.

- CartesianPoint - PolarPoint

It is also difficult to see a similarity between implementations here, but there is a strong relationship between the fields.
public double distance(IPoint other) {
    double deltaX = x() - other.x();  
    double deltaY = y() - other.y();  
    return Math.sqrt(deltaX*deltaX + deltaY*deltaY); }

public double distance(IPoint other) {
    double deltaX = x() - other.x();  
    double deltaY = y() - other.y();  
    return Math.sqrt(deltaX*deltaX + deltaY*deltaY); 
}

CartesianPoint P = new CartesianPoint(x, y);  
PolarPoint Q = new PolarPoint(rho, theta);  
Q = new CartesianPoint(Q.x(), Q.y());  
Q = new PolarPoint(Q.rho(), Q.theta());  
Q = new CartesianPoint(Q.x(), Q.y());  
Q = new PolarPoint(Q.rho(), Q.theta());  
Q = new CartesianPoint(Q.x(), Q.y());  
Q = new PolarPoint(Q.rho(), Q.theta());  

CartesianPoint P = new CartesianPoint(x, y);  
PolarPoint Q = new PolarPoint(rho, theta);  
Q = new CartesianPoint(Q.x(), Q.y());  
Q = new PolarPoint(Q.rho(), Q.theta());  
Q = new CartesianPoint(Q.x(), Q.y());  
Q = new PolarPoint(Q.rho(), Q.theta());  
Q = new CartesianPoint(Q.x(), Q.y());  
Q = new PolarPoint(Q.rho(), Q.theta());  

CartesianPoint P = new CartesianPoint(x, y);  
PolarPoint Q = new PolarPoint(rho, theta);  
Q = new CartesianPoint(Q.x(), Q.y());  
Q = new PolarPoint(Q.rho(), Q.theta());  
Q = new CartesianPoint(Q.x(), Q.y());  
Q = new PolarPoint(Q.rho(), Q.theta());  
Q = new CartesianPoint(Q.x(), Q.y());  
Q = new PolarPoint(Q.rho(), Q.theta());  

CartesianPoint P = new CartesianPoint(x, y);  
PolarPoint Q = new PolarPoint(rho, theta);  
Q = new CartesianPoint(Q.x(), Q.y());  
Q = new PolarPoint(Q.rho(), Q.theta());  
Q = new CartesianPoint(Q.x(), Q.y());  
Q = new PolarPoint(Q.rho(), Q.theta());  
Q = new CartesianPoint(Q.x(), Q.y());  
Q = new PolarPoint(Q.rho(), Q.theta());  

CartesianPoint P = new CartesianPoint(x, y);  
PolarPoint Q = new PolarPoint(rho, theta);  
Q = new CartesianPoint(Q.x(), Q.y());  
Q = new PolarPoint(Q.rho(), Q.theta());  
Q = new CartesianPoint(Q.x(), Q.y());  
Q = new PolarPoint(Q.rho(), Q.theta());  
Q = new CartesianPoint(Q.x(), Q.y());  
Q = new PolarPoint(Q.rho(), Q.theta());  

String string1 = new String("Hello, World!");  
string1 = "Hello, World!";  
string1 = "Hello, World!";  
string1 = "Hello, World!";  
string1 = "Hello, World!";  
string1 = "Hello, World!";  
string1 = "Hello, World!";  
string1 = "Hello, World!";
The structure of the class creation process:

1. **Stage One**: Allocating memory for the complete Manager object.
   - Initialize all instance variables to their default values.
   - Call the constructor: `Manager("Joe Smith", "Sales")`.
   - Bind constructor parameters: `n="Joe Smith", d="Sales"`.
   - No explicit `this()` call.
   - Call `super(n)` for `Employee(String)`.
   - Bind constructor parameters: `n="Joe Smith`.
   - Call `this(n, null)` for `Employee(String, Date)`.
     - Bind constructor parameters: `n="Joe Smith", DoB=null`.
     - No explicit `this()` call.
     - Call `super()` for `Object()`.
     - No binding necessary.
     - No `this()` call.
     - No `super()` call (Object is the root).
     - No explicit variable initialization for `Object`.
     - No method body to call.
2. **Execute body**: Initialize explicit `Employee` variables: `salary=15000.00`.
3. **Execute body**: Set `name="Joe Smith"`, `date=null`.
4. **Execute body**: Set `department="Sales"`.