Procedural $\rightarrow$ **OOP** $2/2$

Tamás Ambrus, István Gansperger

Eötvös Loránd University
ambrus.thomas@gmail.com

OOP

What is Object Oriented Programming?

- Encapsulation
  - binding the data with the code that manipulates it
  - data and the code is safe from external interference
- Inheritance
  - an object can acquire some/all properties of another object
  - supports the concept of hierarchical classification
- Abstraction
  - essential feature without the background details
  - what the object does instead of how it does it
- Polymorphism
  - process objects differently based on their data type
  - one method with different implementation

Inheritance

Inheritance is the mechanism that lets you inherit functionality from other classes.

- while composition (having class fields) means "has a" relationship
    - for example *Rectangle* has an *a* side
- inheritance means "is a" relationship
    - Square **is a** rectangle, but it's not just a regular rectangle, it's a special one.
    - if inheritance did not exist, a lot of code would be duplicated

Inheritance - in a post office

```java
public class Item {
    private final String address;

    public Item(String address) {
        this.address = address;
    }

    public void ship() {
        // ...
    }
}

public class Package extends Item {
    private final double weight;

    public Package(String address, double weight) {
        super(address);
        this.weight = weight;
    }
}
```

Inheritance

The *Package* class would look like this without inheritance:

```java
public class Package {
    private final String address;
    private final double weight;

    public Package(String address, double weight) {
        this.address = address;
        this.weight = weight;
    }

    public void ship() {
        // ...
    }
}
```

The more code lines or subclasses you would need, the more code duplication you would get.

Abstract Classes

Abstract classes give us the possibility to define types that

- **cannot be instantiated**
- already contain implementations
- but in most cases they are not fully implemented

Why is that good for us? We are saved from duplicated code. We only need to fill the **holes** (define the abstract methods) in the subclasses.

Abstract Classes

```java
public abstract class ShapeCalculator {
    protected final List<Shape> shapes;

    public ShapeCalculator(List<Shape> shapes) {
        this.shapes = shapes;
    }

    public abstract double calculate();
}
```

If we needed some Calculator types (max area, sum perimeter, etc.) we could save some information to an abstract class:

- we could work with List in every subclass
- we should override only the calculate method
- nobody could instantiate a ShapeCalculator object even though it exists

Abstract Classes

```java
public class MaxAreaCalculator extends ShapeCalculator {

    public MaxAreaCalculator(List<Shape> shapes) {
        super(shapes);
    }

    @Override
    public double calculate() {
        double max = shapes.get(0).getArea();

        for (int i = 1; i < shapes.size(); i++) {
            double area = shapes.get(i).getArea();
            if (area > max) {
                max = area;
            }
        }

        return max;
    }
}
```

Abstract

Good to know:

- a class can extend only 1 another class
- a class does not have to have an abstract method to be abstract
- abstract methods never have implementation bodies
- a subclass does not have to implement an inherited abstract method - in case of the subclass is abstract too
- abstract classes cannot be instantiated because if you don't know the implementation, how would JVM know?
- by extending a type you inherit all class fields, all methods - except the constructors

Visibility (again)

We have already seen 2 visibility modifiers: **public** and **private**. In reality there are 4 of them in Java, but we commonly use 3:

- **private**: only the defining type can reach it
- **package private** (default): *private* + types in the same package can reach it
- **protected**: *package private* + all subtypes of the defining type can reach it
- **public**: *protected* + everybody can reach it

Interfaces

Interfaces are *similar to* abstract classes. Except:

- they cannot have non-static class fields
- all of their methods are **public** and **abstract** by default
    - from Java 8 methods may have default implementations
- a class can implement any number of interfaces

So interfaces only declare methods but they don't define. In other words interfaces are contracts:

- specify the parameters and return value
- let classes implement the method bodies

Interfaces

Let's refactor the previous *ShapeCalculator* implementation a bit to use interfaces.

```
public interface Calculator {
    double calculate();
}

public abstract class ShapeCalculator implements Calculator {
    protected final List<Shape> shapes;

    public ShapesCalculator(List<Shape> shapes) {
        this.shapes = shapes;
    }
}
```

The *public* and *abstract* keywords are optional, we usually leave them.

Polymorphism

As we have learnt type hierarchy, it's time to learn polymorphism too, they are a nice couple.
The following assignment is valid and common in OOP languages:

```
Calculator calc = new MaxAreaCalculator();
```

MaxAreaCalculator is the subtype of *Calculator*, so we can create a variable whose static type is *Calculator*, while the dynamic type is *MaxAreaCalculator*.

Static and Dynamic Binding

- **Static Binding**
  The binding which can be resolved at compile time by the compiler is known as static or early binding. All the static, private and final methods have always been bound at compile time.
- **Dynamic Binding**
  It's called dynamic or late binding when the compiler is not able to resolve the call/binding at compile time. Overriding is a perfect example of dynamic binding as in overriding both the parent and child classes have the same method.

Polymorphism

```
public interface Shape {
    double getArea();
}

public class Circle implements Shape {
    private double r;

    @Override
    public double getArea() { return r * r * Math.PI; }
}

public class Rectangle implements Shape {
    private double a;
    private double b;

    @Override
    public double getArea() { return a * b; }
}
```

Static and Dynamic Binding

```
public static void draw(Circle c) { System.out.println("Circ"); }
public static void draw(Shape s) { System.out.println("Shape"); }

public static void main(String[] args) {
    Rectangle r = new Rectangle(2, 4);
    Circle c = new Circle(4.5);

    Shape s1 = c;
    Shape s2 = new Rectangle(2, 4);

    double ra = r.getArea(); // area of the rectangle
    double rc = c.getArea(); // area of the circle
    double rs1 = s1.getArea(); // area of the same circle

    // but!

    draw(r); // prints "Shape"
    draw(s2); // prints "Shape"
    draw(c); // prints "Circ"
    draw(s1); // prints what?
}
```

Polymorphism

Good to know:

- polymorphism is when the method call depends on the referenced type of the variable
- the real power of polymorphism is when you work with a collection of objects
- the top super class in Java is Object
- any Java object that is able to pass more than one IS-A test (instanceof) is considered to be polymorphic

References

- http://beginnersbook.com/2013/03/oops-in-java-encapsulation-inheritance-polymorphism-abstraction/
- http://beginnersbook.com/2013/04/java-static-dynamic-binding/
- www.tutorialspoint.com/java/java_polymorphism.htm