Object Oriented Programming notes

Mikkel Helsing Andersen

January 7, 2024

Contents

Chapter 1

Programming concepts

A programming paradigm is a way of conceptualizing what it means to perform computation, and how tasks that are to be carried out on a computer should be structured and organized.

A program has two components data and algorithms.

Imperative paradigm: Also know as algorithmic paradigm, think C. No classes, references to memory, sequentiel running of program/algorithm.

Procedural paradigm: Same as Imperative except it's more modular, meaning code is easily reusable.

Declarative paradig: Computer finds the solution to a problem, think SQL. The data is "permanent".

Functional paradigm: Based on mathematical functions. Data is immutable (Underlying structure cannot be changed). Java allows for lambda functions which allows for creating functional programming.

Logic paradigm: Set the goal and specify the problem, to make the computer solve the problem, instead of writing an algorithm for finding said solution.

Object-oriented par: Objects contains the data and algorithms. Where data is the state of the object. The algorithms allows to change the state of an object. Objects are often described as classes in programming. Think Java, C# etc.

Polymorphism

Overloading Polymorphism

Multiple definition of a function with varying parameters.

```
public class MathUtil {
    public static int max(int n1, int n2) {
        /*code*/
    }
    public static double max(double n1, double n2) {
        /*code*/
    }
    public static int max(int[] num) {
        /*code*/
    }
}
```

Coercion Polymorphism

When a type is implicitly converted.

```
int num = 707;
double d1 = (double)num; // Explicit
double d2 = num; // Implicit
```

Inclusion Polymorphism

Subtype, meaning a type/class that extends or implements a type/class can also be assigned to it's parent type. Think employee extends person can also be assigned to a variable with type person.

Parametric Polymorphism

Lets classes and algorithms be generic without type specification Two conflicting imports

```
List<String> sList = new ArrayList<String>();

public <T> T example(T test) {
    return test;
}

class test<T> {
    /*code*/
}
```

can be used directly as such:

```
package pkg;
import p1.A;
import p2.A; // A compile-time error
class Test {
    A var1; // Which A to use p1.A or p2.A?
    A var2; // Which A to use p1.A or p2.A?
}
```

The alternative is stating it directly when declaring the type:

```
// Test.java
package pkg;
class Test {
    p1.A var1; // Use p1.A
    p2.A var2; // Use p2.A
}
```

Or:

```
// Test.java
package pkg;
import p1.A;
class Test {
    A var1; // Refers to p1.A
    p2.A var2; // Uses the fully qualified name p2.A
}
```

jav

Chapter 2

Classes

Chapter begins: (P. 225[1])

2.1 What is a Java Class

Java Classes has a set of properties and functionalities. Non-static classes can be instantiated. Classes can also work with generic types and inlusion as mentioned in polymorphism and inclusion respectivly. Include the following:

- Fields
- Methods
- Constructors
- Static initializers
- Instance initializers

Basis for creating a class:

```
// <T> is only for generic classes
[modifiers] class class-name <T> {
    // Body of the class goes here
}
// Example
public abstact class Human<T> {
    // An empty body for now
}
```

Static fields in a class are class fields, whereas none static fields are instance variables.

[1] Java Fundamentals

Imports

Import statements that uses the wild(*) modifier are on demand imports, meaning all available classes and functions can be used on demand:

```
import com.java.string.*
```

The Java compiler must resolve the simple name A to its fully qualified name during the compilation process. It searches for a type referenced in a program in the following order:

- The current compilation unit
- Single-type import declarations
- Types declared in the same package
- Import-on-demand declarations

Part I

Design patterns

Chapter 3

SOLID

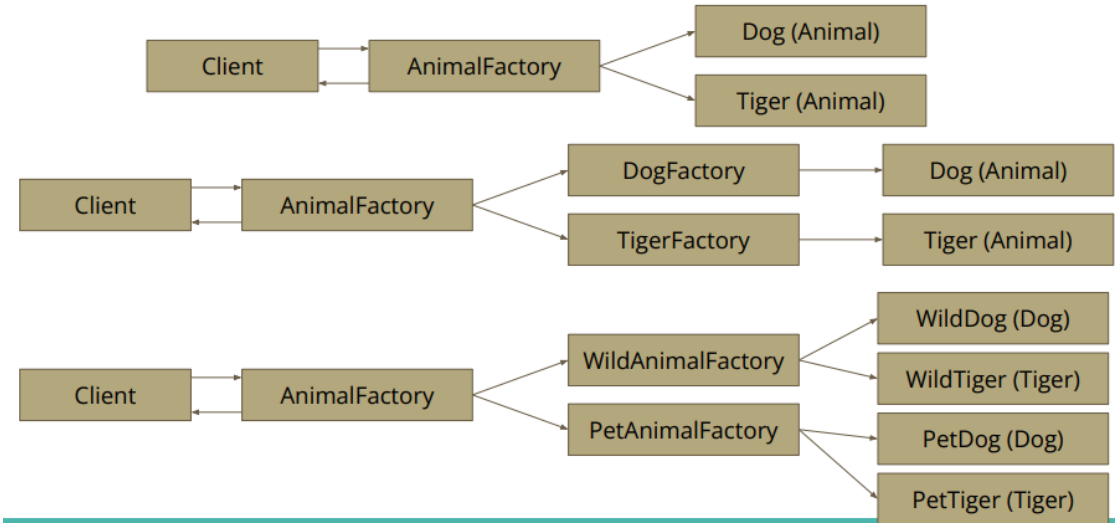
SOLID principles

- Single Responsibility Principle
  - A class should have **only one reason to change**, avoid having methods that do different things
- Open/Closed Principle
  - It should be **possible to expand** the set of operations or fields, but the interface must **remain stable** for others to use
- Liskov Substitution Principle
  - It should be possible to **substitute a parent** type with subtype instance
- Interface Segregation Principle
  - Keep in the (super)type **only the necessary methods**
- Dependency Inversion Principle
  - Types should only **depend on more abstract** types

Chapter 4

Creational Patterns

4.1 Factory



4.1.1 Simple Factory

```
class AnimalFactory {
    public Animal createAnimal(Type animalType) {
        Animal animal = null;
        if (animalType.equals(Type.DOG)) {
            animal = new Dog();
        } else if (animalType.equals(Type.TIGER)) {
            animal = new Tiger();
        } else if (animalType.equals(Type.CAT)) {
            animal = new Cat();
        }
        return animal;
    }
}
```

4.1.2 Factory Pattern

An object with a method that creates new objects using abstraction.

```
// Base factory
abstract class AnimalFactory {
    public abstract Animal createAnimal();
}

// One factory using the base factory
class DogFactory extends AnimalFactory {
    public Animal createAnimal() {
        return new Dog();
    }
}

// Usage of factory
AnimalFactory factory = new DogFactory();
Animal animal = factory.createAnimal();
animal.displayBehavior();
```

4.1.3 Abstract Factory

```
// Base factory
interface AnimalFactory {
    Tiger createTiger();
    Dog createDog();
}

// Wild factory
public class WildAnimalFactory implements AnimalFactory {
    public Tiger createTiger() { ... }
    public Dog createDog() { ... }
}

// Pet factory
public class PetAnimalFactory implements AnimalFactory {
    public Tiger createTiger() { ... }
    public Dog createDog() { ... }
}

// Usage
AnimalFactory factory = new WildAnimalFactory();
Animal animal = factory.createTiger();
animal.displayBehavior();
```

4.2 Builder Pattern

```
// Instantiating class at the beginning
public class Course {
    ...
    public static class CourseBuilder {
        private Course c;
        public CourseBuilder() {
            c = new Course();
        }
        public void setCode(String code); {
            c.code = code;
        }
        public void setTitle(String title) {
            c.title = title;
        }
        public Course getCourse() {
            return c;
        }
    }
}

// Using rigged constructor
public class Course {
    ...
    public static class CourseBuilder {
        private String ccode, ctitle;
        public CourseBuilder() { }
        public void setCode(String code) {
            ccode = code;
        }
        public void setTitle(String title) {
            ctitle = title;
        }
        public Course getCourse() {
            return new Course(ccode, ctitle);
        }
    }
}
```

Chapter 5

Structural Patterns

5.1 Decorator Pattern

Dynamically add responsibilities to an object using composition. The decorator has the same interface as the decorated object, meaning it has the same interaction from the client perspective.

```
abstract class Home {
    public double basePrice;
    public double additionalCost;

    public Home() {
        this.basePrice = 100000.0;
        this.additionalCost = 0.0;
    }

    public abstract double getPrice();
}

class BasicHome extends Home {
    @Override
    public double getPrice() {
        return this.basePrice;
    }
}

// Abstract class for creating decorators
abstact class Luxury extends Home {
    protected Home home;
```

```
    public double luxuryCost;

    public Luxury(Home home) {
        this.home = home;
    }

    @Override
    public double getPrice() {
        return this.home.getPrice();
    }
}

// Decorator
class SwimmingPool extends Luxury {
    public SwimmingPool(Home home) {
        super(home);
        this.luxuryCost = 20000;
    }

    @Override
    public double getPrice() {
        return this.home.getPrice() + this.luxuryCost;
    }
}

// Usage
Home home = new BasicHome;
home.getPrice() // Now 100000
// Add decorator to home
home = SwimmingPool(home);
home.getPrice() // Now 120000
```

5.2 Adapter Pattern

Adapts a class to conform to another class by creating a middleman layer that adapts it.

5.2.1 Object adapter

```
interface RectInterface {
    void aboutMe();
```

```
}

class Rectangle implements RectInterface {
    ....
    @Override
    public void aboutMe() {
        System.out.println("I got 4 corners")
    }
}

interface TriInterface {
    void aboutTriangle();
}

class Triangle implements TriInterface {
    ....
    @Override
    public void aboutTriangle() {
        System.out.println("I got 3 corners")
    }
}

// Adapt triangle to conform to Rectangle
class Adapter implements RectInterface {
    private Triangle triangle;

    public Adapter(Triangle triangle) {
        this.triangle = triangle;
    }

    @Override
    public void aboutMe() {
        this.triangle.aboutTriangle();
    }
}
```

5.2.2 Class adapter

```
// See the code above for context only adapter changes
// Adapt triangle to conform to Rectangle
```

```
class Adapter extends Triangle implements RectInterface {
    @Override
    public void aboutMe() {
        this.aboutTriangle();
    }
}
```

5.3 Composite Pattern

Composite pattern allows the client to handle objects the same, whether they are leafs or nodes. Meaning a leaf can have no children, whereas nodes (Composites) can have children of the same interface. Both leaf and nodes implements the same interface, for cohesive interaction for the client. Composite pattern follows a tree structure.

```
interface Tool {
    private List<Tool> tools;
    private String name;
    public String getName();
    public int getToolCount();
    public void addTool(Tool tool);
}

class Screwdriver implements Tool {
    {
        name = "Screwdriver";
    }

    @Override
    public String getName() {
        return this.name;
    }

    @Override
    public int getToolCount() {
        new Exception("A Screwdriver can't hold other tools");
    }

    @Override
    public void addTool(Tool tool) {
        new Exception("A Screwdriver can't hold other tools");
    }
```

```
}

class ToolBox implements Tool {
    {
        name = "Tool Box";
        tools = new List<Tool>();
    }

    @Override
    public String getName() {
        return this.name;
    }

    @Override
    public int getToolCount() {
        return this.tools.length();
    }

    @Override
    public void addTool(Tool tool) {
        this.tools.append(tool);
    }
}
```

Chapter 6

Behavioral Patterns

6.1 Template Method Pattern

Used to allow customisation of an algorithm within subclasses. The base class sets up the skeleton of an algorithm, but gives some of the responsibility to the subclasses.

```
abstract class LinearSearch() {
    public int search(List<int> list, int searchFor) {
        for (int i = 0; i < list.length(); i++) {
            if (this.compare(list[i], searchFor)) return i;
        }
    }

    public abstract boolean compare(int a, int b);
}

class SimpleSearch extends LinearSearch {
    @Override
    public boolean compare(int a, int b) {
        if (a == b) return true;
        return false;
    }
}
```