ENGINEER SERIES

IN JAVA LANGUAGE

2 CH

THINKING IN OBJECTS

ENG : Qusay Khudair

Creativity and Accuracy in Work

AUG PLUS .Edu

Chapter 10

Thinking in Objects

ENG : Qusay Khudair

Immutable Objects and Classes

Definition: Immutable objects are objects whose state cannot be changed once created.

Key Points:

- Fields are private and final. (in the slide private only)
- No setters, only getters.
- Use constructors for setting values.

Example:

```
public final class ImmutableExample {
    private final String name;
    private final int age;

    public ImmutableExample(String name, int age)
{
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

Scope of Variables

Definition: Scope of a variable determines where the variable can be accessed.

Types:

- **Local** : starts from its declaration and continues to the end of the block that contains the variable. A local variable must be initialized explicitly before it can be used.

- **Instance and Static** : entire class , They can be declared anywhere inside a class.

Example:

```
public class VariableScope {
    private int instanceVar; // Instance variable
    private static int staticVar; // Static
variable

    public void method() {
        int localVar = 10; // Local variable
        // Access instanceVar, staticVar, and
localVar
    }
}
```

The this Keyword

Definition: Refers to the current object , or can be said “this” in Java is a keyword that refers to the current object instance.

It can be used to **call current class methods and fields**, **to pass an instance of the current class as a parameter**, and to **differentiate between the local and instance variables and enable a constructor to invoke another constructor of the same class.**.

- Using “**this**” reference can improve code readability and reduce naming conflicts.

Example:

```
public class ThisExample {
    private String Name;

    public ThisExample(String name) {
        this.Name = name; // Using this to
resolve name conflict
    }

public String getName() {
 return Name; }

public static void main(String[] args) {
ThisExample example = new ThisExample("Qusay");
System.out.println(example.getName()); // Output
: Qusay

 } }
```

Reference the Hidden Data Fields

Definition: Using `this` to reference fields hidden by parameters or local variables.

Example:

```
public class HiddenDataFields {
    private int Value;

    public HiddenDataFields(int value) {
        this.Value = value; // 'this.Value'
references the field, 'value' is the parameter
    }
}
```

Calling Overloaded Constructor

Definition: Using `this()` to call another constructor in the same class.

Example:

```
public class ConstructorOverload {
    private int x, y;

    public ConstructorOverload() {
        this(0, 0); // Calling another constructor
    }
    public ConstructorOverload(int x, int y) {
        this.x = x;
        this.y = y; }}
```

Class Abstraction and Encapsulation

Definition:

- Abstraction: Hiding complex implementation details.
  - **Main feature**: reduce complexity, promote maintainability, and also provide clear separation between the interface and its concrete implementation.
  - Abstraction provides access to specific part of data.

Example:

```
// Java program to illustrate the concept of
Abstraction
abstract class Shape {
    String color;
    // these are abstract methods
    abstract double area();
    public abstract String toString();

    // abstract class can have a constructor
    public Shape(String color)
    {
        System.out.println("Shape constructor
called");
        this.color = color;
    }



    // this is a concrete method
    public String getColor() { return color; }}
```

```
class Circle extends Shape {
    double radius;
    public Circle(String color, double radius)
    {
        // calling Shape constructor
        super(color);
        System.out.println("Circle constructor
called");
        this.radius = radius;
    }

    @Override double area()
    {
        return Math.PI * Math.pow(radius, 2);
    }

    @Override public String toString()
    {
        return "Circle color is " + super.color
            + " and area is : " + area();}}

class Rectangle extends Shape {

    double length;
    double width;




    public Rectangle(String color, double length,
                     double width)
    {
```

```
        // calling Shape constructor
        super(color);
        System.out.println("Rectangle constructor
called");
        this.length = length;
        this.width = width;  }

    @Override double area() {
return length * width; }

    @Override public String toString()
    {
        return "Rectangle color is " +
super.color
            + " and area is : " + area();
    }
}

public class Test {
    public static void main(String[] args)
    {
        Shape s1 = new Circle("Red", 2.2);
        Shape s2 = new Rectangle("Yellow", 2, 4);

        System.out.println(s1.toString());
        System.out.println(s2.toString());
    }
}
```

Output

```
Shape constructor called
Circle constructor called
Shape constructor called
```

```
Rectangle constructor called
Circle color is Red and area is : 15.205308443374602
Rectangle color is Yellow and area is : 8.0
```

- Encapsulation: the process or method to contain the information into a single unit and providing this single unit to the user.

- Main feature: Data hiding. It is a common practice to add data hiding in any real-world product to protect it from external world. In OOPs, this is done through specific access modifiers.

  -Encapsulation hides data and the user cannot access same directly (data hiding).

```
// Java program to demonstrate encapsulation

class Encapsulate {

    // private variables declared these can only
be accessed by public methods of class

    private String Name;
    private int Roll;
    private int Age;

    // get method for age to access private
variable Age
    public int getAge() {
return Age; }

    // get method for name to access private
variable Name
    public String getName() { return Name; }
```

```
    // get method for roll to access private
variable Roll
    public int getRoll() { return Roll; }
    // set method for age to access private
variable age
    public void setAge(int newAge) {
Age = newAge; }

    // set method for name to access private
variable Name
    public void setName(String newName)
    {
        Name = newName;
    }

    // set method for roll to access private
variable Roll
    public void setRoll(int newRoll) {
Roll = newRoll; }
}

// Class to access variables of the class
Encapsulate
public class TestEncapsulation {
    public static void main(String[] args)
    {
        Encapsulate obj = new Encapsulate();

        // setting values of the variables
        obj.setName("Qusay");
        obj.setAge(24);
        obj.setRoll(51);
```

```
        // Displaying values of the variables
        System.out.println(" Name: " +
obj.getName());
        System.out.println(" Age: " +
obj.getAge());
        System.out.println("Roll : " +
obj.getRoll());

      // Direct access of Roll is not possible
due to encapsulation
// System.out.println("Roll roll: " + obj.Roll);
    }
}
```

Output

```
Name: Qusay
Age is : 24
Roll: 51
```

Object Composition

Definition: **is a design principle where one class contains an object of another class in its instance variables**. This allows a class to reuse the functionality of another class by containing an instance of that class instead of inheriting from it.

Benefits of Object Composition:

1. **Code Reusability**.
2. **Flexibility and Maintainability**.
3. **Encapsulation**.
4. **Avoids Inheritance Hierarchies**.

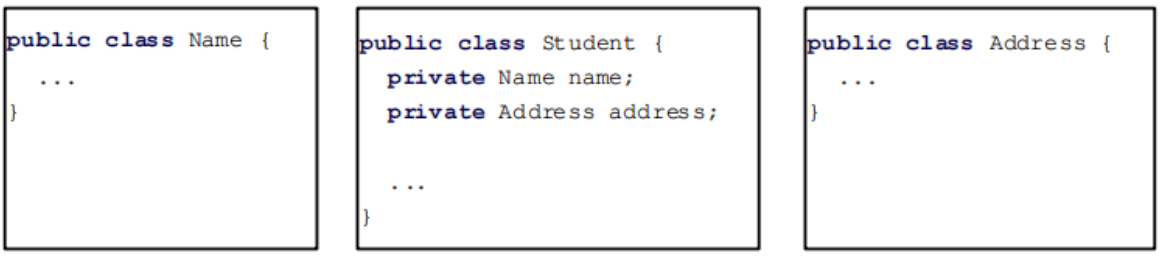
Example:

```
public class Engine {
        private String type;

        public Engine(String type) {
            this.type = type;
        }

        public String getType() {
            return type;
        }
    }

    public class Car {
        private Engine engine;

        public Car(Engine engine) {
            this.engine = engine;
        }

        public Engine getEngine() {
            return engine;
        }
    }
```

Class Representation

Class representation using UML diagrams provides a clear and standardized way to visualize and design classes, their attributes, methods, and relationships. This helps in understanding, communicating, and documenting the structure and behavior of the system.

Example Diagram:



Aggregation or Composition

Since aggregation and composition relationships are represented using classes in similar ways, many texts don't differentiate them and call both compositions.

- Aggregation is a weaker relationship, while composition is a stronger relationship .

Aggregation Between Same Class

Aggregation between the same class in Java refers to a **situation where a class contains references to objects of the same class**. This is useful for modeling hierarchical relationships where an instance of a class can contain other instances of the same class, such as in a tree structure or a linked list.

Ex:

```
import java.util.ArrayList;
import java.util.List;

public class Person {
    private String name;
    private List<Person> friends;

    public Person(String name) {
        this.name = name;
        this.friends = new ArrayList<>();
    }

    public void addFriend(Person friend) {
        friends.add(friend);
    }

    public List<Person> getFriends() {
        return friends;
    }

    public String getName() {
        return name;
    }
```

```
    public static void main(String[] args) {
        Person john = new Person("John");
        Person alice = new Person("Alice");
        Person bob = new Person("Bob");

        john.addFriend(alice);
        john.addFriend(bob);

        System.out.println(john.getName() + "'s
friends:");
        for (Person friend : john.getFriends()) {

System.out.println(friend.getName());} }}
```

- **For More … Follow Me :**

  Eng.Qusay Khudair

- **For Business :**

  Linkedin

  Behance

  Github

- **For My Files and Slides :**

  studocu.com

- **Whatsapp : +972567166452**