Islamic University of Technology

Visual Programming Lab

CSE 4402

Basics of OOP

Author:
Ishmam Tashdeed
Sabrina Islam
CSE, IUT

Contents

1 Object Oriented Programming

Object-Oriented Programming (OOP) is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts. The main pillars of OOP are:

- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

1.1 Class

In object-oriented programming, a class is a blueprint from which individual objects are created. Using classes, you can create multiple objects with the same behavior instead of writing their code multiple times. This includes classes for objects occurring more than once in your code. An example of creating a class in Java is:

```java
// Create a Student class

public class Student {
   // Declaring attributes
   String name;
   int rollNo;
   String section;

   // initialize attributes in constructor
   Student(String name, int rollNo, String section){
      this.name= name;
      this.rollNo = rollNo;
      this.section = section;
   }

   //Getter methods
   public String getName() {
       return this.name;
   }

   public int getRoll() {
       return this.rollNo;
```

```
    }

    public String getSection() {
        return this.section;
    }

    // print details
    public void printDetails() {
        System.out.println("Student Details:");
        System.out.println(this.getName());
        System.out.println(this.getRoll());
        System.out.println(this.getSection());
    }
}
```

In Java, if you do not specify an access modifier, the member (class, method, or variable) gets **private** access by default.

1.2 Object

An Object is a basic unit of Object-Oriented Programming that represents real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. The objects are what perform your code, they are the part of your code visible to the viewer/user. An example of creating an object in Java is:

```
// Create student objects
Student student1 = new Student("Robert", 1, "A");

// Print student details
student1.printDetails();
```

1.3 Inheritance

In object-oriented programming, inheritance is a process by which we can reuse the functionalities of existing classes to new classes. In the concept of inheritance, there are two terms base (parent) class and derived (child) class. When a class is inherited from another class (base class), it (derived class) obtains all the properties and behaviors of the base class.

```java
// base class for all students
public class Person {
   String name;

   Person(String name){
      this.name = name;
   }
}

// create a Student class
public class Student extends Person {
   // Declaring attributes
   int rollNo;
   String section;

   // initialize attributes
   Student(String name, int rollNo, String section){
      super(name);
      this.rollNo = rollNo;
      this.section = section;
   }
   // print details
   public void display() {
      System.out.print("Student Details: ");
      System.out.println(this.name+ ", " + this.rollNo + ", "
    + section);
   }
```

1.3.1 Multiple-Inheritance

Multiple inheritance, where a class can inherit from multiple parent classes, is not directly supported in Java. This design choice is primarily to avoid the complexities and ambiguities that can arise from multiple inheritance, most notably the ”diamond problem.” The diamond problem occurs when a class inherits from two classes that have a common ancestor, leading to potential conflicts in method resolution.

```java
// First Parent Class
class Parent1 {
    void fun() { System.out.println("Parent1"); }
}

// Second Parent Class
```

```java
class Parent2 {
    void fun() { System.out.println("Parent2"); }
}

// Inheriting Properties of
// Parent1 and Parent2
// Will throw an error
class Test extends Parent1, Parent2  // This is not allowed !
{
    // main method
    public static void main(String args[])
    {
        // Creating instance of Test
        Test t = new Test();

        t.fun();
    }
}
```

1.4 Polymorphism

Polymorphism refers to the ability of object-oriented programming languages to differentiate between entities with the same name efficiently. This is done by Java with the help of the signature and declaration of these entities. The ability to appear in many forms is called polymorphism. Polymorphism in Java is mainly of 2 types as mentioned below:

- Method Overloading
- Method Overriding

1.4.1 Method Overloading

Also, known as compile-time polymorphism, is the concept of Polymorphism where more than one method share the same name with different signature (Parameters) in a class. The return type of these methods may or may not be same.

```java
// Parent Class
class Parent {
    // Overloaded method (compile-time polymorphism)
    public void func() {
        System.out.println("Parent.func()");
    }

    // Overloaded method (same name, different parameter)
    public void func(int a) {
        System.out.println("Parent.func(int): " + a);
    }
}

public class Main {
    public static void main(String[] args) {
        Parent parent = new Parent();

        // Method Overloading (compile-time)
        parent.func();
        parent.func(10);
    }
}
```

1.4.2 Method Overriding

Also, known as run-time polymorphism, is the concept of Polymorphism where method in the child class has the same name, return-type and parameters as in parent class. The child class provides the implementation in the method already written.

```java
// Parent Class
class Parent {
    public void func(int a) {
        System.out.println("Parent.func(int): " + a);
    }
}

// Child Class
class Child extends Parent {
    // Overrides Parent.func(int) (runtime polymorphism)
    @Override
    public void func(int a) {
        System.out.println("Child.func(int): " + a);
    }
}

public class Main {
    public static void main(String[] args) {
        Child child = new Child();

        // Dynamic dispatch
        Parent polymorphicObj = new Child();

        // Method Overriding (runtime)
        child.func(20);

        // Polymorphism in action
        polymorphicObj.func(30);
    }
}
```

1.4.3 Operator Overloading

Java does not support operator overloading in the same way as some other programming languages like C++. Operator overloading allows the redefinition of built-in operators, such as +, -, *, / to work with user-defined types. While Java doesn't allow custom operator overloading, it achieves similar functionality through method overloading. Method overloading allows defining multiple methods with the same name but different parameters within the same class. This enables different behaviors based on the types or number of arguments passed.

1.5 Abstraction

In object-oriented programming, an abstraction is a technique of hiding internal details and showing functionalities. **Abstract classes** and **interfaces** are used to achieve abstraction in Java. The real-world example of an abstraction is a Car, the internal details such as the engine, process of starting a car, process of shifting gears, etc. are hidden from the user, and features such as the start button, gears, display, break, etc are given to the user. When we perform any action on these features, the internal process works.

```
// Abstract class representing a Vehicle (hiding
   implementation details)
abstract class Vehicle {
    // Abstract methods (what it can do)
    abstract void accelerate();
    abstract void brake();
    // Concrete method (common to all vehicles)
    void startEngine() {
        System.out.println("Engine started!");
    }
}

// Concrete implementation (hidden details)
class Car extends Vehicle {
    @Override
    void accelerate() {
        System.out.println("Car: Pressing gas pedal...");
        // Hidden complex logic: fuel injection, gear
   shifting, etc.
    }

    @Override
    void brake() {
        System.out.println("Car: Applying brakes...");
        // Hidden logic: hydraulic pressure, brake pads, etc.
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle myCar = new Car();
        myCar.startEngine();
        myCar.accelerate();
        myCar.brake();
    }
}
```

1.5.1 Interface

An interface in Java is a blueprint of a behavior. A Java interface contains static constants and abstract methods. It is used to simulate scenarios with multiple inheritance. By default, variables in an interface are public, static, and final. Interfaces primarily define methods that other classes must implement.

```
// Interface declared
interface Vehicle {
    // public, static and final variables
    final int a = 10;

    // Abstract methods (what it can do)
    abstract void accelerate();
    abstract void brake();
}

// Implements the interface
class Car implements Vehicle {
    public void accelerate() {
        System.out.println("Car: Pressing gas pedal...");
    }

    public void brake() {
        System.out.println("Car: Applying brakes...");
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();

        myCar.accelerate();
        myCar.brake();
    }
}
```

1.6 Encapsulation

Encapsulation is a process of binding the data members (attributes) and methods together. The encapsulation restricts direct access to important data. The best example of the encapsulation concept is making a class where the data members are private and methods are public to access through an object. In this case, only methods can access those private data. Encapsulation can be achieved by declaring all the variables in a class as private and writing public methods in the class to set and get the values of the variables.

```
public class Student{
    // Private attributes
    private int id;
    private String name;
    // Constructor
    Student(int id, String name){
        this.id = id;
        this.name = name;
    }

    // Setter methods
    public void setId(int id){
        this.id = id;
    }

    public void setName(String name){
        this.name = name;
    }

    //Getter methods
    public int getId(){
        return this.id;
    }

    public String getId(){
        return this.name;
    }
}
```

2 Tasks

2.1 Task 1

Suppose you are developing a zoo monitoring system that monitors different animals like lions, elephants, and parrots. Each animal has a name, age, and a unique animal ID in the format: ”animal type - serial number” (e.g., Lion - 1). All animals can be fed using a feed() method and can make a sound using makeSound(), but each displays species-specific messages like “Lion is eating meat” or “Parrot is chirping.” Each of the animal has some functionalities and attributes:

- Lions have attributes like size and whether they are alpha; they can `roar()` and `hunt()`, hunting alone or in a group will depend being Alpha.

- Elephants have trunk length and weight, with methods like `sprayWater()` and `walk()`, walking speed is affected by weight.
- Parrots can talk and have a vocabulary size; they can `speak()` phrases and `fly()`, flying distance will change based on age.

2.2 Task 2

Create a custom stack class that takes a list of numbers from the user. The user defines the size of the list. Implement the following methods for:

- Stack Push (Pushes an integer into the stack and prints the list).
- Stack Pop (Removes and returns the element on the top of the stack).
- Max value (Returns the maximum value currently in the stack).
- Min value (Returns the minimum value currently in the stack).
- Size Returns the number of elements currently in the stack.