Java Programming 1 - Week 11 Notes

Hia Al Saleh

November 14th, 2024

Contents

1 Introduction

In this week's lecture, we are diving deeper into Object-Oriented Programming (OOP), focusing on building super and subclasses, inheritance, and method overriding and overloading. We will use these concepts to create a game in which each character belongs to a specific race (`Orc`, `Elf`, `Human`) and has unique abilities and attributes.

2 Overview of Super and Subclasses

A **superclass** (also called a **base** or **parent class**) is a general class that defines common properties and methods for all its subclasses. A **subclass** (also known as a **derived** or **child class**) inherits the properties and methods from the superclass and can add or override them.

Example:

```
class Character {
    // Common attributes for all characters
    String name;
    int age;
    int health;

    public Character(String name, int age) {
        this.name = name;
        this.age = age;
        this.health = 100; // default health
    }

    public void displayInfo() {
        System.out.println("Name: " + name + ", Age: " +
            age + ", Health: " + health);
    }
}

class Orc extends Character {
    int strength;

    public Orc(String name, int age) {
        super(name, age);
        this.strength = 120; // default orc strength
    }

    public void displayInfo() {
        super.displayInfo();
        System.out.println("Strength: " + strength);
    }
}
```

In this example, `Orc` is a subclass that inherits from the `Character` superclass, adding a specific attribute (`strength`) and overriding the `displayInfo()` method to include it.

3 Creating the `Character` Superclass

The `Character` superclass defines attributes and behaviors that all characters will share, such as name, age, health, and methods to access these attributes. We also define an abstract `communicate()` method, which each subclass will implement differently.

3.1 Attributes

Each `Character` will have the following attributes:

- **Name:** The character's name.
- **Age:** The character's age.
- **Health:** An integer representing health points.
- Other attributes include intelligence, strength, defense, and agility.

3.2 Constructor and Methods

The constructor initializes name and age, while getters and setters allow access to other private attributes.

Example:

```
public abstract class Character {
    private String name;
    private int age;
    private int health;

    public Character(String name, int age) {
        this.name = name;
        this.age = age;
        this.health = 100; // Default health
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public int getHealth() {
```

```
        return health;
    }

    // Abstract method to be implemented by subclasses
    public abstract void communicate();
}
```

4 Creating Subclasses for Different Races

The subclasses (`Orc`, `Elf`, and `Human`) extend `Character` and provide specific implementations of the `communicate()` method and race-specific attributes.

4.1 Orc Subclass

The `Orc` class represents a strong character type with higher-than-average strength but lower intelligence.

Example:

```
public class Orc extends Character {
    private int strength;

    public Orc(String name, int age) {
        super(name, age);
        this.strength = 150;
    }

    @Override
    public void communicate() {
        System.out.println("Grunts and roars");
    }
}
```

4.2 Elf Subclass

The `Elf` class has higher agility and intelligence, suitable for stealth and magic.

Example:

```
public class Elf extends Character {
    private int agility;

    public Elf(String name, int age) {
        super(name, age);
        this.agility = 120;
    }

    @Override
    public void communicate() {
```

```
        System.out.println("Speaks in an elegant, mystical
            language");
    }
}
```

4.3 Human Subclass

The `Human` class is well-balanced across attributes.

Example:

```
public class Human extends Character {
    private int wisdom;

    public Human(String name, int age) {
        super(name, age);
        this.wisdom = 100;
    }

    @Override
    public void communicate() {
        System.out.println("Speaks in a common language");
    }
}
```

5 Creating Arrays of Characters

In Java, we can create arrays of objects to store instances of different subclasses. This allows us to perform actions on each object in a unified way.

Example:

```
public class Game {
    public static void main(String[] args) {
        Character[] characters = {
            new Orc("Gor", 30),
            new Elf("Aelar", 100),
            new Human("John", 25)
        };

        for (Character character : characters) {
            character.displayInfo();
            character.communicate();
        }
    }
}
```

This array contains different types of `Character` objects (an `Orc`, an `Elf`, and a `Human`), and each one can use the `communicate()` method, demonstrating polymorphism.

6 Abstract Classes and Methods

An **abstract class** cannot be instantiated and is meant to be a base class for other classes. In our game, the `Character` class is abstract to ensure no character is created without a specific race.

Abstract Method Example:

```
public abstract class Character {
    public abstract void communicate();
}
```

Polymorphism in Action: In the `Game` class example, polymorphism allows each character type to execute its specific `communicate()` method when called on the array of `Character` objects.

7 Homework and Test Preparation

- Review Chapter 11 of the textbook, focusing on abstract classes and polymorphism.
- Be prepared to demonstrate how inheritance and method overriding work in the context of the game characters.
- Experiment with adding more races, such as `Werewolf` or `Vampire`, each implementing unique `communicate()` behaviors.

8 Next Week

Prepare for a test covering:

- Creating classes, subclasses, and inheritance.
- Method overriding and overloading.
- Abstract classes and polymorphism.