Object-Oriented Programming Paradigm CS315

Joie Ann M. Mac

Reference

- Chapter 12 - Support for Object-Oriented Programming
  - Concepts of Programming Languages by Sebesta

Topics

- Introduction
- Object-Oriented Programming
- Support for OOP in Java

Introduction

- Many object-oriented programming (OOP) languages
  - Some support procedural and data-oriented programming (e.g., Ada 95 and C++)
  - Some support functional program (e.g., CLOS)
  - Newer languages do not support other paradigms but use their imperative structures (e.g., Java and C#)
  - Some are pure OOP language (e.g., Smalltalk & Ruby)

Object-Oriented Programming

- Roots in SIMULA 67 but was not fully developed until Smalltalk
- Three major language features:
  - Abstract data types
  - Inheritance
  - Dynamic Binding

Software Reuse

- Productivity increases can come from reuse
  - ADTs are difficult to reuse-always need changes
  - All ADTs are independent and at the same level
- Inheritance allows new classes defined in terms of existing ones, i.e., by allowing them to inherit common parts
- Inheritance addresses both of the above concerns - reuse ADTs after minor changes and define classes in a hierarchy

Object-Oriented Concepts

- ADTs are usually called **classes**
- Class instances are called **objects**
- A class that inherits is a **derived class** or a **subclass**
- The class from which another class inherits is a **parent class** or **superclass**
- Subprograms that define operations on objects are called **methods**

Object-Oriented Concepts

- Calls to methods are called **messages**
- The entire collection of methods of an object is called its **message protocol** or **message interface**
- Messages have two parts--a method name and the destination object
- In the simplest case, a class inherits all of the entities of its parent

java

```java
// Define a Dog class
class Dog {
    void speak() {
        System.out.println("Woof!");
    }
}

// Define a Cat class
class Cat {
    void speak() {
        System.out.println("Meow!");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Cat cat = new Cat();

        // Sending messages to objects
        dog.speak();  // Output: Woof!
        cat.speak();  // Output: Meow!
    }
}
```

Object-Oriented Concepts

- Inheritance can be complicated by access controls to encapsulated entities
  - A class can hide entities from its subclasses
  - A class can hide entities from its clients
  - A class can also hide entities for its clients while allowing its subclasses to see them
- Besides inheriting methods as is, a class can modify an inherited method
  - The new one **overrides** the inherited one
  - The method in the parent is **overridden**

Object-Oriented Concepts

- Three ways a sub class can differ from its parent:
  1. The subclass can add variables and/or methods to those inherited from the parent

```java
// Base class Animal
class Animal {
    // Variable inherited by subclass
    String name;

    // Method inherited by subclass
    void eat() {
        System.out.println(name + " is eating.");
    }
}

// Subclass Dog extending Animal
class Dog extends Animal {
    // New variable specific to Dog
    String breed;

    // New method specific to Dog
    void bark() {
        System.out.println(name + " is barking.");
    }
}
```

Object-Oriented Concepts

- Three ways a sub class can differ from its parent:

  2. The parent class can define some of its variables or methods to have private access, which means they will not be visible in the subclass

```java
// Parent class Person
class Person {
    // Private variable (not accessible from subclasses)
    private String name;

    // Private method (not accessible from subclasses)
    private void setName(String name) {
        this.name = name;
    }

    // Public method to set name (can be used by subclasses)
    public void initializeName(String name) {
        setName(name);
    }

    // Public method to get name (can be used by subclasses)
    public String getName() {
        return name;
    }
}
```

Object-Oriented Concepts

- Three ways a sub class can differ from its parent:

  3. The subclass can modify the behavior of one or more of its inherited methods.

```java
// Define a base class
class Animal {
    void speak() {
        System.out.println("Some generic sound...");
    }
}

// Define a subclass that overrides the speak method
class Dog extends Animal {
    @Override
    void speak() {
        System.out.println("Woof!");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();
        Animal myDog = new Dog();

        myAnimal.speak();  // Output: Some generic sound...
        myDog.speak();     // Output: Woof! (Dog's implementation o
    }
}
```

Object-Oriented Concepts

- There are two kinds of variables in a class:
  - *Class variables* (static variable)- are shared across all instances of a class. Every object can access and modify these variables, and any change made to a class variable affects all instances
  - *Instance variables* – unique to each object of the class. Every object has its own set of instance variables, which can store different values.
- There are two kinds of methods in a class:
  - *Class methods* (static methods) - belong to the class rather than any specific instance
  - *Instance methods* - are methods that operate on the instance (or object) of the class.

```java
class Dog {
    // Class variable (shared by all instances)
    static String species = "Canis Familiaris";

    // Instance variables
    String name;
    String breed;

    // Constructor to initialize instance variables
    public Dog(String name, String breed) {
        this.name = name;
        this.breed = breed;
    }

    // Instance method
    public void bark() {
        System.out.println(name + " is barking!");
    }

    // Class method (static method)
    public static void setSpecies(String newSpecies) {
        species = newSpecies;
    }
}
```

Object-Oriented Concepts

- Single vs. Multiple Inheritance
- One disadvantage of inheritance for reuse:
    - Creates interdependencies among classes that complicate maintenance

Single Inheritance

```java
class Parent {
    void greet() {
        System.out.println("Hello from Parent");
    }
}

class Child extends Parent {
    void farewell() {
        System.out.println("Goodbye from Child");
    }
}

public class Main {
    public static void main(String[] args) {
        Child childInstance = new Child();
        childInstance.greet();    // Output: Hello from Parent
        childInstance.farewell(); // Output: Goodbye from Child
    }
}
```

Multiple Inheritance

```java
interface Parent1 {
    void greet();
}

interface Parent2 {
    void farewell();
}

class Child implements Parent1, Parent2 {
    public void greet() {
        System.out.println("Hello from Parent1");
    }

    public void farewell() {
        System.out.println("Goodbye from Parent2");
    }
}

public class Main {
    public static void main(String[] args) {
        Child childInstance = new Child();
        childInstance.greet();    // Output: Hello from Parent1
        childInstance.farewell(); // Output: Goodbye from Parent2
    }
}
```

Dynamic Binding

- Dynamic binding in Java is primarily used with method overriding, where a method call is resolved at runtime based on the actual object type, rather than the reference type
- Allows software systems to be more easily extended during both development and maintenance

Dynamic Binding Concepts

```java
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();  // Animal reference to Animal object
        Animal myDog = new Dog();        // Animal reference to Dog object (polymo

        myAnimal.sound();  // Output: Animal makes a sound (Static Binding)
        myDog.sound();     // Output: Dog barks (Dynamic Binding at runtime)
    }
}
```

Dynamic Binding Concepts

```java
class Animal {
    void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal1 = new Dog();  // Animal reference to Dog object
        Animal animal2 = new Cat();  // Animal reference to Cat object

        animal1.makeSound();  // Output: Dog barks (Dynamic Binding)
        animal2.makeSound();  // Output: Cat meows (Dynamic Binding)
    }
}
```

Support for OOP in Java

- General Characteristics
  - All data are objects except the primitive types
  - All primitive types have wrapper classes that store one data value
  - All objects are heap-dynamic, are referenced through reference variables, and most are allocated with **`new`**
  - A **`finalize`** method is implicitly called when the garbage collector is about to reclaim the storage occupied by the object

Support for OOP in Java

- Inheritance
  - Single inheritance supported only, but there is an abstract class category that provides some of the benefits of multiple inheritance (**`interface`**)
  - An interface can include only method declarations and named constants, e.g.,

```
public interface Comparable <T> {
        public int comparedTo (T b);
}
```

  - Methods can be **`final`** (cannot be overriden)

Support for OOP in Java

- Evaluation
  - No parentless classes for java
  - Design decisions to support OOP are similar to C++
  - Dynamic binding is used as "normal" way to bind method calls to method definitions
  - Uses interfaces to provide a simple form of support for multiple inheritance

Attendance:

https://forms.gle/CSorAtJCbzSLLpUZ6