Object-Oriented Thinking

Introduction to Object-Oriented Thinking

- Moving beyond basic class definition to how to think in an object-oriented way.
- Focuses on design principles and common OOP patterns.
- Key concepts: inheritance, polymorphism, abstract classes, interfaces.

Class Relationships: Association

- Association: A general binary relationship that describes an activity between two classes."Has a" or "uses a" relationship.
- Example: A Student takes a Course. (Many-to-many)
- Represented by a plain line in UML.

Class Relationships: Aggregation

- Aggregation: A special form of association that represents a "has-a" relationship where one object "owns" or contains another, but the contained object can exist independently.
- "Part-of" relationship.
- Example: A Department has Professors. If the department is dissolved, professors still exist.
- Represented by a hollow diamond on the "whole" side.

Class Relationships: Composition

- Composition: A stronger form of aggregation where the contained object cannot exist independently of the containing object. "Part-of" with strong ownership.
- Example: A House has Rooms. If the house is demolished, the rooms cease to exist.
- Represented by a filled diamond on the "whole" side.

Class Relationships: Inheritance

- Inheritance: An "is-a" relationship where a new class (subclass/derived class/child class) is created from an existing class (superclass/base class/parent class).
- The subclass inherits all public and protected members of the superclass.
- Promotes code reuse and establishes a hierarchy.

Defining a Superclass and Subclass

- Use the extends keyword to indicate inheritance.
- Syntax: class Subclass extends Superclass { ... }
- Example:
  class Circle extends GeometricObject { ... }

The Object Class

- All classes in Java implicitly or explicitly extend the java.lang.Object class.
- It is the **root** of the class hierarchy.
- Provides common methods like equals(), toString(), hashCode().

Constructor Chaining

- When a subclass object is created, its constructor implicitly or explicitly calls a superclass constructor.
- The super() keyword is used to call a superclass constructor.
- If super() is not explicitly called, Java automatically inserts a call to the no-arg superclass constructor.

Overriding Methods

- A subclass can provide its own implementation of a method that is already defined in its superclass.
- The method signature (name, parameter list) must be **identical**.
- Use @Override annotation for clarity and compile-time checking.

Invoking Superclass Methods (super keyword)

- The super keyword can be used to explicitly call a method from the superclass that has been overridden in the subclass.
- Syntax:
  super.methodName(arguments);

Polymorphism

- Polymorphism: "Many forms." The ability of an object to take on many forms.
- A reference variable of a superclass type can refer to an object of any of its subclasses.
- The actual method executed is determined at runtime based on the object's actual type.

Dynamic Binding (Runtime Polymorphism)

- The JVM determines which version of an overridden method to execute at runtime, based on the actual type of the object pointed to by the reference variable.

Casting Objects

- Upcasting: Assigning a subclass object to a superclass reference variable (always safe, implicit).
- Downcasting: Assigning a superclass reference variable to a subclass reference variable (requires explicit cast, potentially unsafe, compile-time check for validity).
- Use instanceof operator to check object type before downcasting.

The instanceof Operator

- Used to test whether an object is an instance of a particular class or an instance of a class that implements a particular interface.
- Syntax: object instanceof ClassOrInterface
- Returns true or false.

equals() Method in Object Class

- The default equals() method in Object compares memory addresses (object identity).
- Often needs to be overridden in custom classes to compare object content (values of data fields).

Guidelines for Overriding equals()

- Reflexive: x.equals(x) is true.
- Symmetric: x.equals(y) implies y.equals(x).
- Transitive: If x.equals(y) and y.equals(z), then x.equals(z).
- Consistent: Multiple invocations return the same result (if objects are unchanged).
- x.equals(null) is false.
- Cast Object parameter to the correct type and handle null and instanceof.

The toString() Method

- Returns a string representation of the object.Default implementation in Object provides class name and hash code.

- Usually overridden to provide a meaningful, human-readable description of the object's state.

Abstract Classes and Methods

- Abstract class: A class that cannot be instantiated directly. It serves as a blueprint for other classes. May contain abstract methods.
- Abstract method: A method declared without an implementation (no method body). Must be implemented by concrete (non-abstract) subclasses.
- Declared with the abstract keyword.
- The abstract keyword is a non-access modifier, used for classes and methods
- An abstract class can have both **abstract** and regular **methods**

Example Of abstract class

```
// Abstract class
abstract class Animal {
public abstract void animalSound(); // Abstract method (does not have a body)
public void sleep() {// Regular method
    System.out.println("Zzz");
  }
}
```

- From the example above, it is **not** possible to create an object of the Animal class:
  Animal myObj = new Animal(); // will generate an error

```
// Subclass (inherit from Animal)
class Cat extends Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The cat says: mew mew");
  }
}


      class Main {
  public static void main(String[] args) {
 Cat myCat = new Cat(); // Create a Cat object
    myCat.animalSound();
    myCat.sleep();
  }
}
```

Purpose of Abstract Classes

- Define a common interface for a set of subclasses.
- Enforce that certain methods must be implemented by subclasses.
- Provide partial implementation (some concrete methods) while leaving others abstract.

Interfaces

- An interface is a completely "abstract class" that is used to group related methods with empty bodies
- A class can implement one or more interfaces.
- Interfaces establish a "can-do" or "has-a-capability" relationship.
- Like abstract classes, interfaces cannot be used to create objects (in the example above, it is not possible to create an "Animal" object in the MyMainClass)
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interface methods are by **default abstract** and **public**
- Interface attributes are by default **public**, **static** and **final**
- An interface **cannot contain a constructor** (as it cannot be used to create objects)

```
// interface
interface Animal {
  public void animalSound();
// interface method (does not have a body)
  public void run();
// interface method (does not have a body)
}
class Cat implements Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The cat says: mew mew");
  }
  public void sleep() {
    // The body of sleep() is provided here
    System.out.println("Zzz");
  }
}
```

```
class Main {
  public static void main(String[] args) {
    Cat myCat = new Cat();  // Create a cat object
    myCat.animalSound();
    myCat.sleep();
  }
}
```

Abstract Classes vs. Interfaces

- Abstract Class: "Is-a" relationship, single inheritance, can have constructors, instance variables, concrete methods, abstract methods.

- Interface: "Can-do" relationship, multiple inheritance possible, no instance variables (only constants), all methods implicitly public abstract (pre-Java 8), or default/static.

Polymorphism with Interfaces

- An interface reference variable can refer to any object of a class that implements that interface.
- Example:
  Comparable c = new Circle(5);
- Allows treating diverse objects uniformly based on their shared capabilities.

Designing with Interfaces

- Define common behaviors that multiple unrelated classes might share.
- Decouple the client code from the specific implementation details.
- Example:

  Flyable interface for Bird, Airplane, Kite.

Concept	What It Is	Used To / Purpose	Keywords / Syntax	Notes
Interface	A contract with method signatures	Define what a class should do (not how)	interface, implements	All methods are public abstract by default
Abstract	Incomplete class	Provide base class with some common + abstract parts	abstract class, extends	Can have both abstract & concrete methods
Polymorphism	One interface, many forms	Allow objects to behave differently via the same method	Method Overriding, Overloading	Achieved via inheritance, interface, or override

Cloning Objects

- Creating a duplicate of an existing object.
- Java's Object.clone() method performs a shallow copy.
- For deep copy (duplicating nested objects), you usually need to implement Cloneable interface and override clone().

Immutable Classes (Revisited in Context of OOP Design)

- A good design practice to make classes immutable when their state should not change after creation.
- Enhances thread safety and simplifies reasoning.
- Requires private final fields, no setters, and defensive copying of mutable object fields.

Design Guidelines for Classes

- Cohesion: A class should represent a single logical entity (highly cohesive).
- Consistency: Maintain consistent naming conventions, method signatures.
- Completeness: Provide necessary methods for typical usage.
- Clarity: Well-documented, easy to understand.

Encapsulation (Revisited)

- Crucial for robust OOP design.
- Hides implementation details and provides a clean public interface.
- Changes to internal implementation don't affect external code if the public interface is maintained.

Key Concepts of Object-Oriented Thinking

- Association, Aggregation, Composition: Different forms of "has-a" relationships.
- Inheritance: "Is-a" relationship, code reuse.
- Polymorphism: One interface, multiple implementations.
- Abstract Classes & Interfaces: Defining contracts and common behaviors.
- Design Principles: Cohesion, encapsulation, immutability.

Conclusion

- Chapter 10 moves from syntax to design philosophy.
- Understanding these concepts allows you to build flexible, maintainable, and scalable object-oriented software.
- Practice applying these principles in your code!