Advance Educational Activities Pvt. Ltd.

Unit 3: Object-Oriented Programming – II

3.1.1 Aggregation in Java

Aggregation is a form of association that represents a **"Has-A" relationship** between two classes.
It is a **weaker** form of composition — the **lifecycle of the contained object is independent** of the container.

Definition: Aggregation is when one class contains a reference to another class, but both can exist independently.

Syntax Example

```
class Address {
    String city, state;

    Address(String city, String state) {
        this.city = city;
        this.state = state;
    }
}
class Customer {
    String name;
    Address address; // Aggregation: Customer has an Address

    Customer(String name, Address address) {
        this.name = name;
        this.address = address;
    }

    void showCustomerDetails() {
        System.out.println(name + " lives in " + address.city + ", " + address.state);
    }
}

public class Main {
    public static void main(String[] args) {
        Address addr = new Address("Chennai", "Tamil Nadu");
        Customer c = new Customer("Ravi", addr);
        c.showCustomerDetails();
    }
}
```

Output

```
Ravi lives in Chennai, Tamil Nadu
```

Real-World Analogy

- **Customer ↔ Address**
  - A **Customer** has an **Address**.
  - But if the Customer is deleted, the Address object might still exist in other contexts (like used by multiple customers).
- Another example:
  Company has a **CEO** → If CEO resigns, Company still exists → **Aggregation**.

When to Use Aggregation

- When one class **uses another class**, but doesn't manage its full lifecycle.
- When you want to establish a **modular, reusable structure**.
- To avoid tight coupling.

Best Practices

- Use aggregation to **decouple responsibilities**.
- Avoid deep object nesting unless necessary.
- Combine with interfaces when building extensible systems.

3.1.2 Composition in Java?

Composition is a design principle in Java where one class contains an object of another class, and the **contained object's lifecycle is strictly tied** to the container object.

Definition: Composition is a "Has-A" relationship where the contained object cannot exist without the container.

Key Characteristics of Composition

- Strong form of association.
- If the container is destroyed, the contained objects are also destroyed.
- Provides better **encapsulation** and **control** over the parts.

Syntax Example

Let's take a real-world analogy: A **Car** has an **Engine**, and the engine's life is tied to the car.

```
class Engine {
    void start() {
        System.out.println("Engine started");
    }
}

class Car {
    private Engine engine;  // Composition: Car owns Engine

    Car() {
        engine = new Engine();  // Engine is created inside Car
    }

    void drive() {
        engine.start();
```

```
        System.out.println("Car is moving");
    }
}

public class Main {
    public static void main(String[] args) {
        Car car = new Car();
        car.drive();
    }
}
```

Output

```
Engine started
Car is moving
```

Real-World Analogy

- A **Car** has an **Engine**.
- The **Engine doesn't exist** independently — it's meaningful only as part of the **Car**.
- If the Car is scrapped, the Engine is gone too.

When to Use Composition

- When one class **controls the existence** of another class.
- When building **complex types from smaller types**.
- When you want **tight coupling** to enforce strong dependency.

Composition vs Aggregation

FEATURE	COMPOSITION	AGGREGATION
RELATIONSHIP	Strong "Has-A"	Weak "Has-A"
LIFESPAN	Contained object tied to container	Independent
OWNERSHIP	Exclusive	Shared or reused
DELETION	Deleting the container deletes part	Deleting the container has no effect
EXAMPLE	Car → Engine	Customer → Address

3.2.1 Association in Java

What is Association in Java?

Association is a relationship between two separate classes that are connected through their **objects**.

Definition: Association represents a "uses-a" or "has-a" relationship between two independent classes, where both classes can exist independently.

It is the **most general relationship** in Object-Oriented Programming.

Basic Syntax Example

```
class Customer {
    String name;

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

class Order {
    void placeOrder(Customer customer) {
        System.out.println(customer.name + " placed an order.");
    }
}

public class Main {
    public static void main(String[] args) {
        Customer c = new Customer("Rahul");
        Order o = new Order();
        o.placeOrder(c);  // Association between Order and Customer
    }
}
```

Output:

```
Rahul placed an order.
```

Types of Association in Java

There are mainly **two types** of association:

1. One-Way Association (Unidirectional)

Only one class is aware of the relationship.

Example: Order knows Customer, but Customer doesn't know Order.

2. Two-Way Association (Bidirectional)

Both classes are aware of each other.

Example:

```
class Customer {
    String name;
    Order order;

    void setOrder(Order order) {
        this.order = order;
    }
}
```

```
class Order {
    String id;
    Customer customer;

    void setCustomer(Customer customer) {
        this.customer = customer;
    }
}
```

Now both classes **reference each other**.

Association vs Aggregation vs Composition

FEATURE	ASSOCIATION	AGGREGATION	COMPOSITION
RELATIONSHIP	Uses-a / Has-a	Has-a (weak)	Has-a (strong)
LIFECYCLE	Independent	Contained object is independent	Contained object is dependent
OWNERSHIP	No	Yes (shared)	Yes (exclusive)
EXAMPLE	Student ↔ Course	Customer ↔ Address	Car ↔ Engine

Real-World Analogy

- A **Customer** uses an **Order**.
- A **Student** enrolls in a **Course**.
- Even if the course or order is canceled, the customer/student still exists → **Association**.

Best Practices

- Use association when objects interact but do **not own** each other.
- Clearly decide if the relationship is **unidirectional or bidirectional**.
- Avoid bidirectional association unless **really necessary**, as it increases **coupling**.

Summary

- **Association** is the foundation of object relationships in Java.
- It shows how objects are **related but independent**.
- Leads to better **modular and maintainable** code structure.
- Is the base for **Aggregation** and **Composition**, which are stronger forms.

3.3.1 Inheritance in Java

Definition:

Inheritance is a key feature of Object-Oriented Programming (OOP) where **one class (child/subclass) acquires** the **properties and behaviors** (fields and methods) of **another class (parent/superclass)**.

It promotes **code reusability** and enables **hierarchical classification**.

Real-world Analogy:

Imagine:

- **Father has a house and a car.**
- His **son inherits** both — he can use them and even **add his own bike**.

Similarly in Java:

- A **child class** inherits fields and methods from the **parent class** and can also define its own.

3.3.2 Syntax with Customer Example

Code Explanation

We have:

- A base class Customer that contains **common properties and methods**.
- A derived class PremiumCustomer that **inherits from Customer** using the extends keyword.
- The PremiumCustomer class adds its **own extra property/method**.

Code

```
// Superclass
class Customer {
    String name;
    String email;

    void showDetails() {
        System.out.println("Customer Name: " + name);
        System.out.println("Email: " + email);
    }
}

// Subclass
class PremiumCustomer extends Customer {  // Inheriting from Customer
    int rewardPoints;

    void showRewards() {
        System.out.println("Reward Points: " + rewardPoints);
    }
}
```

Usage

```
public class Main {
    public static void main(String[] args) {
        PremiumCustomer pc = new PremiumCustomer();

        // Inherited properties
        pc.name = "Ravi Kumar";
        pc.email = "ravi@example.com";

        // Own property
        pc.rewardPoints = 1500;

        // Inherited method
        pc.showDetails();
```

```
// Own method
        pc.showRewards();
    }
}
```

Output

```
Customer Name: Ravi Kumar
Email: ravi@example.com
Reward Points: 1500
```

What's Happening Here

ELEMENT	INHERITED FROM CUSTOMER	DEFINED IN PREMIUMCUSTOMER
NAME AND EMAIL	Yes	❌ No
SHOWDETAILS() METHOD	Yes	❌ No
REWARDPOINTS	❌ No	Yes
SHOWREWARDS() METHOD	❌ No	Yes

Key Takeaways

- PremiumCustomer **inherits all non-private** fields and methods from Customer.
- It can use and override inherited members.
- Inheritance is declared using the **extends** keyword.
- Helps with **code reuse** and establishing an **"is-a"** relationship.

PremiumCustomer is-a Customer

3.3.3 Real-world Example:

- Person → topmost base class
- Customer → inherits from Person
- PremiumCustomer → inherits from Customer
- LoyalCustomer → inherits from PremiumCustomer

Constructor Execution Order (with static blocks, initializers, and constructors)

- By default, a child class constructor implicitly calls the parent class's default constructor.

Inheritance Hierarchy:

```
public class InheritanceDemo {

    public static void main(String[] args) {
        LoyalCustomer obj1 = new LoyalCustomer("Raj", "raj@example.com", 5, 2500);
        System.out.println();
        LoyalCustomer obj2 = new LoyalCustomer("Sneha", "sneha@example.com", 10, 8000);
    }

}
```

```
class Person {
    static {
        System.out.println("Person Static Block");
    }
    {
        System.out.println("Person Instance Initializer");
    }
    Person() {
        System.out.println("Person Constructor\n");
    }
}
class Customer extends Person {
    static {
        System.out.println("Customer Static Block");
    }
    {
        System.out.println("Customer Instance Initializer");
    }
    Customer() {
        System.out.println("Customer Default Constructor\n");
    }
    Customer(String name, String email) {
        System.out.println("Customer Parameterized Constructor: " + name + ", " + email
+ "\n");
    }
}

class PremiumCustomer extends Customer {
    static {
        System.out.println("PremiumCustomer Static Block");
    }

    {
        System.out.println("PremiumCustomer Instance Initializer");
    }

    PremiumCustomer() {
        System.out.println("PremiumCustomer Default Constructor\n");
    }

    PremiumCustomer(String name, String email, int discountRate) {
        super(name, email);
        System.out.println("PremiumCustomer Parameterized Constructor: Discount Rate =
" + discountRate + "%\n");
    }
}
```

```
class LoyalCustomer extends PremiumCustomer {
    static {
        System.out.println("LoyalCustomer Static Block\n");
    }

    {
        System.out.println("LoyalCustomer Instance Initializer");
    }

    LoyalCustomer(String name, String email, int discountRate, int rewardPoints) {
        super(name, email, discountRate);
        System.out.println("LoyalCustomer Constructor: Reward Points = " + rewardPoints
+ "\n");
    }
}
```

Output Explanation

- When LoyalCustomer obj1 = new LoyalCustomer(...); is executed:

Static blocks

- Executed **once per class** at the time of **first use**
- From **top to bottom** in hierarchy

```
Person Static Block
Customer Static Block
PremiumCustomer Static Block
LoyalCustomer Static Block
```

Instance Initializers and Constructors

- For every object creation (top to bottom):
    1. Instance initializer block
    2. Constructor

```
Person Instance Initializer
Person Constructor

Customer Instance Initializer
Customer Parameterized Constructor: Raj, raj@example.com

PremiumCustomer Instance Initializer
PremiumCustomer Parameterized Constructor: Discount Rate = 5%

LoyalCustomer Instance Initializer
LoyalCustomer Constructor: Reward Points = 2500
```

For second object (obj2), only instance parts run again (not static):

```
Person Instance Initializer
Person Constructor

Customer Instance Initializer
Customer Parameterized Constructor: Sneha, sneha@example.com

PremiumCustomer Instance Initializer
PremiumCustomer Parameterized Constructor: Discount Rate = 10%

LoyalCustomer Instance Initializer
LoyalCustomer Constructor: Reward Points = 8000
```

Parameterized Constructor Flow

```
LoyalCustomer(String name, String email, int discountRate, int rewardPoints) {
    super(name, email, discountRate); // calls PremiumCustomer
}
PremiumCustomer(String name, String email, int discountRate) {
    super(name, email); // calls Customer
}
Customer(String name, String email) {
    // No further call, calls Person default constructor implicitly
}
```

So, the **parameterized flow propagates upward** through the hierarchy via super() calls.

Order of Execution

ORDER	WHAT HAPPENS	CLASS
1	Static block	Person → Customer → PremiumCustomer → LoyalCustomer
2	Instance block	Top to bottom on every object creation
3	Constructor	Same: top → down

3.3.4 Types of Inheritance in Java:

Inheritance is the mechanism in Java where one class acquires the **properties and behaviors** (fields and methods) of another class using the extends keyword.

Java supports **several types of inheritance**, but due to language design, not all are supported directly (like multiple inheritance through classes).

1. Single Inheritance

One child class inherits from one parent class.

```
class Customer {
    void display() {
        System.out.println("Customer details");
    }
}
```

```
class PremiumCustomer extends Customer {
    void showDiscount() {
        System.out.println("10% Discount");
    }
}
```

PremiumCustomer inherits from Customer.

2. Multilevel Inheritance

👉 A class inherits from a child class, which itself inherited from a parent class.

```
class Person {
    void getName() {
        System.out.println("Person Name");
    }
}

class Customer extends Person {
    void getEmail() {
        System.out.println("Customer Email");
    }
}

class LoyalCustomer extends Customer {
    void getPoints() {
        System.out.println("Reward Points");
    }
}
```

LoyalCustomer inherits from Customer, and Customer inherits from Person.

3. Hierarchical Inheritance

👉 Multiple child classes inherit from a single parent class.

```
class Customer {
    void showDetails() {
        System.out.println("Customer Details");
    }
}

class PremiumCustomer extends Customer {
    void getDiscount() {
        System.out.println("Premium Discount");
    }
}
```

```
class RegularCustomer extends Customer {
    void getCoupon() {
        System.out.println("Regular Coupon");
    }
}
```

PremiumCustomer and RegularCustomer both inherit from Customer.

4. Multiple Inheritance (Not Supported via Classes)

Java **does not support multiple inheritance with classes** due to **ambiguity issues** (Diamond Problem).

```
class A {
    void msg() {
        System.out.println("Class A");
    }
}

class B {
    void msg() {
        System.out.println("Class B");
    }
}
// class C extends A, B { }  ❌ Not allowed
```

Reason: If both A and B have msg(), the compiler won't know which one to use.

How Java Supports Multiple Inheritance: Using Interfaces

```
interface A {
    void msg();
}

interface B {
    void msg();
}

class C implements A, B {
    public void msg() {
        System.out.println("Hello from both A and B");
    }
}
```

No conflict if the method is implemented in C.

5. Hybrid Inheritance (Combination — Only via Interfaces)

👉 Mix of hierarchical and multiple inheritance — possible using **interfaces only**, not classes.

Summary Table

TYPE	DESCRIPTION	JAVA SUPPORT
SINGLE	One child, one parent	Yes
MULTILEVEL	Class inherits from child of another class	Yes
HIERARCHICAL	Multiple classes inherit from same parent	Yes
MULTIPLE	One class inherits from multiple classes	❌ Not directly supported (but via interfaces)
HYBRID	Combination of types	❌ Not directly (can simulate with interfaces)

Things to Remember:

- Constructors are **not inherited**
- Private members of a parent class are **not directly accessible**
- Java supports **only single class inheritance** (to avoid ambiguity)
- Use **super** to refer to the superclass

3.4.1 What is super in Java?

The super keyword in Java is a **reference variable** used to refer to the **immediate parent class object**.

Real-world Analogy

Imagine Customer is a base-level employee, and PremiumCustomer is a manager.

- The manager (child) may **inherit** the same login system (method).
- If the manager wants to **reuse the parent login logic**, they can say: "Hey, use the employee login system" → that's super.login().
- If the manager wants to **initialize** some common fields like employeeId → that's super() calling the parent constructor.

Use Cases of super

It is mainly used for:

1. **Calling the parent class constructor**
2. **Accessing parent class methods**
3. **Accessing parent class fields**

1. Calling Parent Class Constructor

You can use super() to explicitly call a constructor of the parent class from the child class constructor.

Syntax:

```
super(); // must be the first statement in the child constructor
```

Example:

```
class Customer {
    Customer() {
        System.out.println("Customer Constructor");
    }
}
```

```
class PremiumCustomer extends Customer {
    PremiumCustomer() {
        super();  // calls Customer()
        System.out.println("PremiumCustomer Constructor");
    }
}
```

Output:

```
Customer Constructor
PremiumCustomer Constructor
```

2. Accessing Parent Class Methods

If a method in the child class overrides the parent method, you can use super.methodName() to call the parent version.

Example:

```
class Customer {
    void showDetails() {
        System.out.println("Customer details");
    }
}

class PremiumCustomer extends Customer {
    void showDetails() {
        super.showDetails(); // call to parent method
        System.out.println("Premium customer details");
    }
}
```

Output:

```
Customer details
Premium customer details
```

3. Accessing Parent Class Variables

If the child class has a field with the same name as the parent, you can use super.variableName to refer to the parent's version.

Example:

```
class Customer {
    String name = "General Customer";
}

class PremiumCustomer extends Customer {
    String name = "Premium Customer";
```

```
void printNames() {
        System.out.println("Child name: " + name);
        System.out.println("Parent name: " + super.name);
    }
}
```

Output:

```
Child name: Premium Customer
Parent name: General Customer
```

Difference between this and super

KEYWORD	REFERS TO	USED FOR
this	Current class instance	Accessing current class members
super	Immediate parent class	Accessing parent class members

Important Rules

- super() must be the **first statement** in a constructor.
- If you don't use super() explicitly, Java automatically inserts super() for you (if the parent has a default constructor).
- You **cannot use super() and this() in the same constructor**.

Best Practices

- Use super() for **code clarity**, especially when the parent class has **important logic** in the constructor.
- Prefer using super.method() when overriding to **retain original behavior** and extend it.

3.5.1 Method Overriding

Method Overriding occurs when a **child class provides a specific implementation** of a method that is **already defined in its parent class**.

- The method name, return type, and parameters **must match exactly**.
- It enables **runtime polymorphism** (dynamic method dispatch).

Real-World Analogy

Imagine a **base class** Employee that has a method getRole() which returns "General Employee".

In a **child class** Manager, we override getRole() to return "Manager".

So, when calling getRole() on an Employee reference pointing to a Manager object, it returns "Manager" — not the base version.

Syntax Example

```
class Employee {
    void work() {
        System.out.println("Employee is working");
    }
}
```

```
class Developer extends Employee {
    @Override
    void work() {
        System.out.println("Developer writes code");
    }
}
```

Usage Example

```
public class TestOverride {
    public static void main(String[] args) {
        Employee e = new Developer();  // Upcasting
        e.work();  // Output: Developer writes code
    }
}
```

Here, the method call is resolved **at runtime** — demonstrating polymorphism.

Rules for Overriding

RULE	EXPLANATION
SAME METHOD NAME	Must match
SAME RETURN TYPE (OR SUBTYPE - COVARIANT)	Exact or covariant return type
SAME PARAMETER LIST	Must match exactly
CHILD CLASS ONLY	Can only override methods from parent class
ACCESS MODIFIER NOT MORE RESTRICTIVE	Can be same or more accessible
CAN'T OVERRIDE FINAL METHODS	Compilation error
CAN'T OVERRIDE STATIC METHODS	Static methods are hidden, not overridden

Overriding with Access Modifiers

```
class A {
    protected void show() { }
}

class B extends A {
    public void show() { } // Valid (more accessible)
}
Illegal Override Example
class A {
    final void show() { }
}

class B extends A {
    void show() { }  // ❌ Compilation error
}
```

Real-World Example: Customer

```
class Customer {
    void getDiscount() {
        System.out.println("Customer gets 5% discount");
    }
}

class PremiumCustomer extends Customer {
    @Override
    void getDiscount() {
        System.out.println("Premium Customer gets 20% discount");
    }
}

public class Test {
    public static void main(String[] args) {
        Customer c = new PremiumCustomer();
        c.getDiscount();  // Output: Premium Customer gets 20% discount
    }
}
```

Best Practices

- Use @Override annotation for clarity and compile-time checking.
- Don't override methods unnecessarily.
- Prefer overriding only when behavior must change in the subclass.

Summary

FEATURE	DESCRIPTION
PURPOSE	Customize parent behavior in child class
OCCURS AT	Runtime (polymorphism)
MUST MATCH	Method name, signature, return type
KEYWORD	@Override is optional but recommended
CAN'T OVERRIDE	final, private, or static methods

3.6.1 Polymorphism

Polymorphism means **"many forms"** — the ability of a single interface or method to behave **differently based on the context**.

Java supports **polymorphism** in two main ways:

- **Compile-time polymorphism** (Method Overloading)
- **Runtime polymorphism** (Method Overriding)

Real-World Analogy

Imagine the word **"print"**:

- If a **printer** receives it — it prints a document.
- If a **teacher** hears it — they might check test papers.
- If a **developer** writes it in code — it displays output.

Same word, different behaviors depending on **context**. That's polymorphism!

Types of Polymorphism

TYPE	MECHANISM	WHEN RESOLVED	EXAMPLE
COMPILE-TIME POLYMORPHISM	Method Overloading	At Compile Time	print(int), print(String)
RUNTIME POLYMORPHISM	Method Overriding	At Runtime	Overriding draw() in shapes

Compile-Time Polymorphism (Method Overloading)

Multiple methods with **same name** but **different parameters**.

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
    }
}
```

Runtime Polymorphism (Method Overriding)

A **parent class reference** points to a **child class object** and calls an overridden method at **runtime**.

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

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

public class Demo {
    public static void main(String[] args) {
        Animal a = new Dog();
        a.sound();  // Output: Dog barks
    }
}
```

Why Use Polymorphism?

- Increases **code flexibility**
- Supports **dynamic method dispatch**
- Enables **interface-based design**
- Enhances **reusability and scalability**

Real-World Corporate Example: Customer Notifications

```
class Customer {
    void notifyCustomer() {
        System.out.println("Notify customer via SMS");
    }
}

class PremiumCustomer extends Customer {
    void notifyCustomer() {
        System.out.println("Notify premium customer via Email and App");
    }
}

public class App {
    public static void main(String[] args) {
        Customer c1 = new PremiumCustomer();  // Polymorphism
        c1.notifyCustomer();  // Output: Notify premium customer via Email and App
    }
}
```

Key Concepts to Remember

CONCEPT	MEANING
@OVERRIDE	Ensures method is overriding a parent method
DYNAMIC DISPATCH	Decides method at runtime using object type
POLYMORPHIC REFERENCE	Parent type refers to child object
INSTANCEOF	To check actual object type during runtime

Bonus: Polymorphism with Interfaces

```
interface Payment {
    void pay();
}

class CreditCard implements Payment {
    public void pay() {
        System.out.println("Paid with Credit Card");
    }
}
```

```
class UPI implements Payment {
    public void pay() {
        System.out.println("Paid via UPI");
    }
}

public class PaymentApp {
    public static void makePayment(Payment p) {
        p.pay();  // runtime polymorphism
    }

    public static void main(String[] args) {
        makePayment(new CreditCard());
        makePayment(new UPI());
    }
}
```

Summary

FEATURE	DESCRIPTION
POLYMORPHISM	Same method/interface behaves differently
COMPILE-TIME	Method Overloading
RUNTIME	Method Overriding via inheritance/interface
REAL BENEFIT	Flexible, maintainable, scalable, modular code

3.7.1 Abstraction

Abstraction is the process of **hiding internal implementation details** and **showing only the essential features** to the user.

Why? To reduce complexity and increase efficiency.

Real-World Analogy

Remote Control — You press buttons (functions like volume up/down), but you don't see how the signals are sent internally to change the channel or volume.

- The **interface (buttons)** is visible.
- The **implementation (circuit logic)** is hidden.

This is **abstraction**.

What is an Abstract Class?

- A class declared with the abstract keyword.
- **Can have both abstract methods and concrete (implemented) methods.**
- Cannot be instantiated directly.
- Must be extended by a subclass.

```
abstract class Vehicle {
    abstract void start(); // abstract method (no body)

    void fuel() {          // concrete method
        System.out.println("Fueling the vehicle");
    }
}
```

Example: Real-World Corporate — Customer Notification System

```
abstract class NotificationService {
    abstract void notifyCustomer(); // abstract method

    void logNotification() {
        System.out.println("Notification logged");
    }
}

class EmailService extends NotificationService {
    void notifyCustomer() {
        System.out.println("Email sent to customer");
    }
}
```

Usage:

```
public class Main {
    public static void main(String[] args) {
        NotificationService service = new EmailService();
        service.notifyCustomer();      // Output: Email sent to customer
        service.logNotification();     // Output: Notification logged
    }
}
```

Key Points About Abstract Classes

FEATURE	DESCRIPTION
ABSTRACT CLASS	Can’t be instantiated
CONTAINS ABSTRACT METHODS	Must be implemented by subclasses
CAN HAVE CONCRETE METHODS	Unlike interfaces (pre-Java 8)
CONSTRUCTORS ALLOWED	Can have constructors (called via child class constructor)
CAN HAVE FIELDS	Fields + methods — like regular classes

Example with Constructor:

```
abstract class User {
    User() {
        System.out.println("User created");
    }

    abstract void accessDashboard();
}

class Admin extends User {
    Admin() {
        super();  // Calls User constructor
    }

    void accessDashboard() {
        System.out.println("Admin dashboard accessed");
    }
}
```

Summary: Abstract Class vs Interface (Quick Look)

FEATURE	ABSTRACT CLASS	INTERFACE
CAN HAVE METHODS	Both abstract and concrete	All abstract (Java 7 and below)
FIELDS	Yes (with any modifier)	Only public static final
MULTIPLE INHERIT.	No (only one superclass)	Yes (multiple interfaces)
CONSTRUCTORS	Yes	No

Best Practices

- Use abstract classes when you need **shared state or common implementation**.
- Use interfaces when you need **100% abstraction** or **multiple inheritance**.
- Always declare overridden methods with @Override for clarity.

3.8.1 Interface

An **interface** in Java is a **blueprint of a class**. It defines **a contract** that implementing classes must follow.

- All methods in interfaces are **implicitly public and abstract** (until Java 7).
- From **Java 8 onwards**, interfaces can also have **default and static methods**.
- Interfaces enable **100% abstraction** and **multiple inheritance**.

Real-World Analogy

Imagine an **ATM Machine Interface**:

- It has buttons for operations: withdraw(), deposit(), checkBalance().
- The internal implementation differs from bank to bank, but the interface remains the same.

So the **interface = contract**, and each **bank = implementing class**.

Syntax

```
interface Payment {
    void pay();  // implicitly public and abstract
}
Implementation:
class UPI implements Payment {
    public void pay() {
        System.out.println("Paid via UPI");
    }
}
Example: Customer Payment Interface
interface PaymentService {
    void processPayment(double amount);
}

class CardPayment implements PaymentService {
    public void processPayment(double amount) {
        System.out.println("Card Payment of ₹" + amount + " processed.");
    }
}

class NetBankingPayment implements PaymentService {
    public void processPayment(double amount) {
        System.out.println("NetBanking Payment of ₹" + amount + " processed.");
    }
}

public class PaymentDemo {
    public static void main(String[] args) {
        PaymentService p1 = new CardPayment();
        p1.processPayment(1000);

        PaymentService p2 = new NetBankingPayment();
        p2.processPayment(2500);
    }
}
```

Features of Interface

FEATURE	DESCRIPTION
100% ABSTRACTION (JAVA 7)	Only method signatures, no implementation
MULTIPLE INHERITANCE	A class can implement multiple interfaces
METHOD MODIFIERS	All methods are public abstract by default
VARIABLES	All variables are public static final
CANNOT HAVE CONSTRUCTORS	Because interfaces are not instantiated

Interface Features by Java Version

JAVA VERSION	FEATURE
JAVA 8	Default and static methods allowed
JAVA 9	Private methods (helper methods)

Java 8+ Interface Example (with default method)

```
interface Printer {
    void print();

    default void status() {
        System.out.println("Printer is online.");
    }

    static void welcome() {
        System.out.println("Welcome to Printer Services!");
    }
}

class LaserPrinter implements Printer {
    public void print() {
        System.out.println("Laser printing document...");
    }
}

public class InterfaceDemo {
    public static void main(String[] args) {
        Printer printer = new LaserPrinter();
        printer.print();
        printer.status();
        Printer.welcome();
    }
}
```

Interface vs Abstract Class

FEATURE	INTERFACE	ABSTRACT CLASS
ABSTRACTION LEVEL	100% (until Java 7)	Partial or full
METHOD TYPES	abstract, default, static	abstract and concrete
FIELDS	public static final only	Any type
CONSTRUCTORS	Not allowed	Allowed
MULTIPLE INHERITANCE	Yes (implements multiple)	No (only single inheritance)

Best Practices

- Use interfaces for **contracts or capabilities** (e.g., Runnable, Comparable).
- Use interfaces when **multiple classes** need to implement the **same behavior** differently.
- Prefer **default methods** only when adding backward-compatible methods.3.8.2 Functional Interface

A **functional interface** is an interface that **contains exactly one abstract method**.

Functional interfaces are the foundation for using **lambda expressions** in Java.

Key Points:

- It **may have** default and static methods.
- It can be annotated with @FunctionalInterface (optional but recommended).
- Used primarily with **lambda expressions** or **method references**.

Real-World Analogy

Imagine a **switch** that turns ON/OFF a device.
It has **only one job**: trigger a single action.

Similarly, a functional interface allows **one single method** — no ambiguity.

Syntax

```
@FunctionalInterface
interface MessageService {
    void sendMessage(String message);  // Single abstract method
}
You can then implement it like this using a lambda:
public class Main {
    public static void main(String[] args) {
        MessageService service = (msg) -> System.out.println("Sending: " + msg);
        service.sendMessage("Welcome!");
    }
}
```

Why Use @FunctionalInterface Annotation?

- Ensures the interface **has only one abstract method**.
- If more methods are added accidentally, the **compiler will throw an error**.

```
@FunctionalInterface
interface InvalidInterface {
    void action1();
    // void action2(); ❌ This will cause a compile-time error
}
```

Built-in Functional Interfaces (Java 8+)

Java provides many built-in functional interfaces in the java.util.function package:

INTERFACE	ABSTRACT METHOD	PURPOSE
PREDICATE<T>	boolean test(T t)	Tests condition, returns boolean
FUNCTION<T,R>	R apply(T t)	Converts T to R (transformation)
CONSUMER<T>	void accept(T t)	Takes a value, returns nothing
SUPPLIER<T>	T get()	Returns a value, takes nothing
BIFUNCTION<T,U,R>	R apply(T, U)	Takes 2 args, returns 1 value

Example: Using Built-in Functional Interface

```
import java.util.function.Predicate;

public class Test {
    public static void main(String[] args) {
        Predicate<String> isLong = str -> str.length() > 5;
        System.out.println(isLong.test("Hello"));      // false
        System.out.println(isLong.test("Welcome"));    // true
    }
}
```

Functional Interface vs Abstract Class

FEATURE	FUNCTIONAL INTERFACE	ABSTRACT CLASS
NUMBER OF ABSTRACT METHODS	Only one	One or more
CONSTRUCTORS	No	Yes
INHERITANCE	Multiple via interface	Single inheritance only
USAGE	Used with lambdas	Used with inheritance

Best Practices

- Always use @FunctionalInterface to protect your interface from accidental changes.
- Use lambdas only when **only one method** needs to be implemented.
- Combine with Stream API and Collections for clean and powerful code.

3.8.3 Lambda Expression

A **lambda expression** is a **short, anonymous way to implement a functional interface** in Java.

Lambda expression = a concise way to write a method using just input and logic — without creating an entire class.

Syntax

(parameters) -> { body }

Or simplified:

- No parameter: () -> System.out.println("Hello")
- One parameter: x -> x * x
- Multiple parameters: (a, b) -> a + b

Example 1: Without Lambda (Traditional)

```
Runnable r = new Runnable() {
    public void run() {
        System.out.println("Running thread...");
    }
};
new Thread(r).start();
```

With Lambda:

```
Runnable r = () -> System.out.println("Running thread...");
new Thread(r).start();
```

Real-World Analogy

Think of **lambda** as **quick anonymous tasks**:

Like jotting a note ("Remind me to call John") instead of writing a full reminder form.

It's **faster** and **cleaner**.

Real Example: Functional Interface + Lambda

```
@FunctionalInterface
interface Greeting {
    void sayHello();
}

public class Main {
    public static void main(String[] args) {
        Greeting greet = () -> System.out.println("Hello, Customer!");
        greet.sayHello();
    }
}
```

Example 2: Lambda with Parameters

```
@FunctionalInterface
interface Calculator {
    int add(int a, int b);
}

public class Main {
    public static void main(String[] args) {
        Calculator sum = (a, b) -> a + b;
        System.out.println("Sum: " + sum.add(10, 20));
    }
}
```

Common Uses (Java 8+)

Lambda expressions are **widely used** in:

1. **Functional Interfaces** (e.g., Runnable, Comparator, ActionListener)
2. **Streams API**
3. **Collections sorting/filtering**
4. **Event handling in GUIs**

Lambda with Java Built-in Functional Interfaces

```
import java.util.function.Predicate;

public class Test {
    public static void main(String[] args) {
        Predicate<String> isLong = str -> str.length() > 5;
        System.out.println(isLong.test("Hello"));    // false
        System.out.println(isLong.test("Welcome"));  // true
    }
}
```

Advantages of Lambda Expressions

ADVANTAGE	DESCRIPTION
CONCISE CODE	Removes boilerplate for anonymous classes
BETTER READABILITY	Easier to understand logic at a glance
FUNCTIONAL-STYLE CODE	Enables Stream API, functional programming
ENCOURAGES IMMUTABILITY	Lambdas work well with stateless operations

Rules / Restrictions

- Can only be used with **functional interfaces**.
- Cannot declare lambda expressions with multiple abstract methods.
- Cannot throw checked exceptions unless declared in the method signature.

3.8.4 Binding in Java

Binding refers to the process of **connecting a method call to the method definition**.

There are two types:

1. **Static Binding (Early Binding)**
2. **Dynamic Binding (Late Binding)**

1. STATIC BINDING (Early Binding)

Definition:

Static binding occurs at **compile time**.
The type of the object is **determined by the compiler**.

Key Characteristics:

- Happens with private, static, and final methods.
- Method resolution is done **at compile time**.
- Also applies to **method overloading** and **variables**.

Example:

```
class StaticBindingDemo {
    static void show() {
        System.out.println("Static method called");
    }

    public static void main(String[] args) {
        StaticBindingDemo.show();  // Compile-time binding
    }
}
```

Real-World Analogy:

Calling someone at a **landline number**.
You're bound to one device (resolved early).

2. DYNAMIC BINDING (Late Binding)

Definition:

Dynamic binding occurs at **runtime**.
The JVM determines **which method to invoke** based on the **actual object**, not the reference type.

Key Characteristics:

- Happens with **non-static**, **non-final**, **non-private** overridden methods.
- Used in **method overriding**
- Enables **runtime polymorphism**

3.8.5 Dynamic Method Dispatch

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved **at runtime**.

Syntax Example:

```
class Customer {
    void getSupport() {
        System.out.println("General support team.");
    }
}

class PremiumCustomer extends Customer {
    @Override
    void getSupport() {
        System.out.println("Premium support team.");
    }
}
```

```
public class DispatchDemo {
    public static void main(String[] args) {
        Customer c1 = new PremiumCustomer();  // Upcasting
        c1.getSupport();  // Resolved at runtime → Premium support team
    }
}
```

Real World Example: Customer Support System

We have:

- A base interface Support
- Two different implementations: BasicSupport and PremiumSupport
- A dispatcher class that interacts **only through the interface**

Code Example

```
// Step 1: Interface - acts as base type
interface Support {
    void assistCustomer();
}

// Step 2: Implementation 1
class BasicSupport implements Support {
    public void assistCustomer() {
        System.out.println("Basic support: Please wait 24 hours for a response.");
    }
}

// Step 3: Implementation 2
class PremiumSupport implements Support {
    public void assistCustomer() {
        System.out.println("Premium support: Connecting to a live agent immediately.");
    }
}

// Step 4: Dispatcher - works with interface type
public class SupportCenter {
    public static void main(String[] args) {
        Support s;  // Interface reference

        s = new BasicSupport();    // Upcasting
        s.assistCustomer();        // Resolved to BasicSupport at runtime

        s = new PremiumSupport(); // Reassigned
        s.assistCustomer();        // Resolved to PremiumSupport at runtime
    }
}
```

How It Works: Dynamic Dispatch in Action

1. The reference s is of type Support (the interface).
2. At **runtime**, the **actual object type** (BasicSupport or PremiumSupport) decides **which assistCustomer() method to invoke**.
3. This is **dynamic method dispatch**: resolution **happens at runtime**, not at compile time.

Analogy

Think of Support s as a **call center operator** who can connect either to a **basic** or a **premium** team — depending on which team is assigned (at runtime), the experience changes.

Multiple Inheritance Highlight

You can extend this by using **multiple interfaces**:

```
interface Billing {
    void generateInvoice();
}

class EnterpriseSupport implements Support, Billing {
    public void assistCustomer() {
        System.out.println("Enterprise support: Account manager on call.");
    }

    public void generateInvoice() {
        System.out.println("Invoice generated for enterprise client.");
    }
}

public class Main {
    public static void main(String[] args) {
        Support s = new EnterpriseSupport();
        s.assistCustomer();  // dynamic dispatch to EnterpriseSupport

        Billing b = (Billing) s;  // downcasting to access Billing
        b.generateInvoice();      // dynamic dispatch again
    }
}
```

Why This Is the Best Example

- Demonstrates polymorphism via **interface-based multiple inheritance**
- Clearly shows **method resolution at runtime**
- Keeps Java's constraints intact (no class-based multiple inheritance)
- Encourages **decoupled and scalable code design**

Comparison Table

FEATURE	STATIC BINDING	DYNAMIC BINDING
BINDING TIME	Compile time	Runtime
APPLICABLE TO	Static, final, private methods	Overridden instance methods
SPEED	Faster (compiler-resolved)	Slower (JVM decides at runtime)
POLYMORPHISM TYPE	Compile-time polymorphism	Runtime polymorphism
METHOD OVERLOADING	Yes	No
METHOD OVERRIDING	No	Yes

Overloading vs Overriding (Quick Reference)

ASPECT	OVERLOADING	OVERRIDING
BINDING TYPE	Static Binding	Dynamic Binding
INHERITANCE	Not required	Requires inheritance
METHOD SIGNATURE	Must differ	Must be the same
RUNTIME OR COMPILE TIME	Compile time	Runtime

Summary

- **Static Binding**: Determined by the **compiler** (faster, used in method overloading, static/private/final methods).
- **Dynamic Binding**: Determined by the **JVM at runtime** (used in method overriding, polymorphism).

3.8.6 What is Casting in Java?

Casting in Java is converting one type to another.

In the context of **inheritance and polymorphism**, we deal with:

- **Upcasting** → Subclass to Superclass
- **Downcasting** → Superclass to Subclass

1. Upcasting (Widening Reference)

Definition:

Assigning a **subclass object to a superclass reference**.

Safe and implicit — no explicit cast needed.

Syntax:

```
Parent p = new Child();  // Upcasting
```

Example:

```
class Customer {
    void accessAccount() {
        System.out.println("Accessing general account");
    }
}
```

```
class PremiumCustomer extends Customer {
    void accessPremiumSupport() {
        System.out.println("Accessing premium support");
    }
}

public class Main {
    public static void main(String[] args) {
        Customer c = new PremiumCustomer();  // Upcasting
        c.accessAccount();                   // Allowed
        // c.accessPremiumSupport();         ❌ Not allowed (method not in Customer)
    }
}
```

Analogy:

Think of a **PremiumCustomer as a Customer** — every premium customer **is-a** customer.
You can treat them generically.

2. Downcasting (Narrowing Reference)

Definition:

Assigning a **superclass reference to a subclass type**.

Unsafe if done blindly — must ensure object type.
Requires **explicit casting**.

Syntax:

```
Child c = (Child) parentRef;  // Downcasting
```

Example:

```
Customer c = new PremiumCustomer();  // Upcasting
PremiumCustomer pc = (PremiumCustomer) c;  // Downcasting
pc.accessPremiumSupport();  // Allowed
Wrong Downcasting Example (Leads to ClassCastException):
Customer c = new Customer();  // Not actually a PremiumCustomer
PremiumCustomer pc = (PremiumCustomer) c;  // Runtime error!
```

Real-World Analogy

- **Upcasting**: Think of storing a *Bike* in a *Vehicle* parking spot. You're treating the bike as a generic vehicle.
- **Downcasting**: You assume a *Vehicle* is a *Bike* to access kickStart() — but if it's actually a *Car*, it will crash.

How JVM handles it:

Type	When Happens	Needs Cast	Risk of Error?	Polymorphism Enabled
Upcasting	Compile Time	No	No	Yes
Downcasting	Runtime	Yes	Yes (if invalid)	No change

Best Practice

- Always check the actual object type before downcasting:

```
if (c instanceof PremiumCustomer) {
    PremiumCustomer pc = (PremiumCustomer) c;
    pc.accessPremiumSupport();
}
```

3.9.1 final Keyword?

In Java, the final keyword is a **non-access modifier** used to restrict:

1. **Variables** (constants)
2. **Methods** (cannot be overridden)
3. **Classes** (cannot be inherited)

1. final Variable — Value Cannot Change

Use:

- To declare **constants**
- Once assigned, value **cannot be changed**

Example:

```
final int MAX_USERS = 100;
MAX_USERS = 200;  // ❌ Compilation error
```

Real-World Analogy:

A **PAN number** in India — assigned once, cannot be changed.

2. final Method — Cannot Be Overridden

Use:

- Prevent subclasses from **modifying logic**
- Ensures behavior consistency

Example:

```
class Account {
    final void displayBalance() {
        System.out.println("Balance shown");
    }
}

class SavingsAccount extends Account {
    // void displayBalance() { }  // ❌ Error: Cannot override final method
}
```

Real-World Analogy:

A **company policy method** that all departments must use as-is.

3. final Class — Cannot Be Extended

Use:

- To prevent **inheritance**
- Improves **security and immutability**

Example:

```
final class PaymentGateway {
    void processPayment() {
        System.out.println("Payment done");
    }
}
// class CustomGateway extends PaymentGateway { }  // ❌ Error
```

Real-World Analogy:

A **sealed legal document** — no one can modify or extend it.

Additional Notes

Final Reference (for Objects):

```
final Customer c = new Customer();
c.name = "Ravi";          // Allowed (modifying object's state)
c = new Customer();       // ❌ Not allowed (can't reassign)
```

Final with Blank Initialization:

You can assign final variables **later**, but only once (especially in constructors):

```
class Student {
    final int id;

    Student(int id) {
        this.id = id;    // Allowed once
    }
}
```

final vs finally vs finalize()

KEYWORD	USE
FINAL	Restricts variables, methods, or classes
FINALLY	Block used to clean up after try-catch
FINALIZE()	Method called by garbage collector (deprecated)

Best Practices

- Use final for:
    - Constants (static final)
    - Immutable classes
    - Preventing subclass behavior modification

3.10.1 Object Class and Its Methods in Java

Analogy:

Imagine you're in a huge company where every employee (class) must follow some base rules defined by HR. These rules are in a **common handbook**. That **handbook = Object class**. Every class, no matter what department, gets these rules.

What is the Object class?

- The Object class is the **root class** of the Java class hierarchy.
- **Every class** in Java **inherits** from java.lang.Object either **explicitly or implicitly**.

So, when you write:

class MyClass { }

Behind the scenes, it's like:

class MyClass extends Object { }

Why is it important?

It provides a **standard interface** of **commonly used methods** that all Java objects can use (e.g., toString(), equals(), hashCode()).

Common Methods of Object Class

METHOD	PURPOSE
toString()	Returns a string representation of the object
equals(object o)	Compares if two objects are logically equal
hashCode()	Returns a hash code (used in hashing data structures)
getClass()	Returns the runtime class of the object
clone()	Creates and returns a copy of the object (requires Cloneable)
finalize()	Called by GC before object is destroyed (deprecated, rarely used)
wait(), notify(), notifyAll()	Used for thread synchronization

1. toString() Method

By default:

MyClass@15db9742 // ClassName@hexadecimal hash

Customizing:

```
class Student {
    String name;
    int age;

    public String toString() {
        return name + " - " + age;
    }
}
```

Output:

```
Student s = new Student("Alice", 20);
System.out.println(s);  // Prints: Alice - 20
```

2. equals() Method

```
Default: Compares memory address (same as ==)
Override to check logical equality
class Student {
    String name;

    public boolean equals(Object o) {
        Student s = (Student) o;
        return this.name.equals(s.name);
    }
}
```

3. hashCode() Method

Used in hash-based collections (like HashMap, HashSet) to locate objects efficiently.

If equals() is overridden, you **must** also override hashCode().

Example:

```
public int hashCode() {
    return name.length(); // Example: simple custom hash logic
}
```

4. getClass() Method

Returns the **class type** of the object.

```
Student s = new Student();
System.out.println(s.getClass().getName());  // Output: Student
```

5. clone() Method

Used to create a **copy** of an object.

- The class must **implement Cloneable** interface.
- The method must **override clone()**.

```
class Student implements Cloneable {
    String name;

    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

6. finalize() Method (Deprecated)

Called by garbage collector **before** an object is destroyed.

```
protected void finalize() {
    System.out.println("Object is being destroyed.");
}
```

Note: finalize() is deprecated as of Java 9+.

Summary Table

METHOD	NEEDS OVERRIDE?	USE CASE
TOSTRING()	✅	When you want readable object info
EQUALS()	✅	To compare contents, not memory
HASHCODE()	with equals()	Used in hash-based collections
GETCLASS()	❌	To check object's runtime class
CLONE()	✅	When copying objects (rarely used now)
FINALIZE()	❌ (Deprecated)	Cleanup before destruction (not reliable)