Identifiers in Java

Identifiers in Java are the names used to identify variables, methods, classes, packages, and interfaces. They must follow certain rules and conventions to be valid.

Rules for Identifiers in Java

Allowed Characters:

Can contain letters (`A-Z`, `a-z`), digits (`0-9`), underscore (`_`), and dollar sign (`$`).

Cannot contain special characters (`@, #, %, ^, &, *`, etc.) except `_` and `$`.

Cannot be a Java Keyword:

Example: `int, class, static, public` cannot be used as identifiers.

Must Begin with a Letter, `_`, or `$`:

Valid: `myVar`, `_temp`, `$price`

Invalid: `1stVar`, `@home` (Cannot start with a digit or special character)

Case-Sensitive:

`MyVariable` and `myvariable` are different identifiers.

No White Spaces Allowed:

`my Variable` is invalid; should be `myVariable`.

No Length Limit (But Should Be Meaningful):

Java has no strict length limit, but identifiers should be concise and readable.

Examples

☑ Valid Identifiers:

```
int age;
String $name;
double _salary;
char myChar;
```

❌ Invalid Identifiers:

```
int 123number;  // Cannot start with a number
```

```
double price#;  // Cannot contain special characters
String class;   // Cannot be a Java keyword
float my value; // Cannot contain spaces
```

Naming Conventions (Best Practices)

Class Names: Should start with an uppercase letter (PascalCase).

```
class StudentDetails { }
```

Variable and Method Names: Start with a lowercase letter and use camelCase.

```
int studentAge;
void calculateTotal() { }
```

Constants: Use uppercase letters with underscores.

```
final double PI_VALUE = 3.14159;
```

Package Names: Use all lowercase.

```
package com.example.project;
```

Modifiers in Java

Modifiers in Java are keywords that change the behavior of classes, methods, and variables. They are categorized into **Access Modifiers** and **Non-Access Modifiers**.

---

1. Access Modifiers

Access modifiers define the **visibility** or **scope** of classes, methods, and variables.

Modifier	Class	Package	Subclass	World
public	☑ Yes	☑ Yes	☑ Yes	☑ Yes

Modifier	Class	Package	Subclass	World
protected	☑ Yes	☑ Yes	☑ Yes	✗ No
default *(no modifier)*	☑ Yes	☑ Yes	✗ No	✗ No
private	☑ Yes	✗ No	✗ No	✗ No

a) `public`

Accessible from **anywhere** in the program.

Can be applied to classes, methods, and variables.

Example:

```
public class Car {
    public String brand = "Toyota"; // Public variable
}
```

b) `private`

Accessible **only within the same class**.

Used for **encapsulation** (data hiding).

Example:

```
class Car {
    private String brand = "Toyota"; // Private variable
}
```

c) `protected`

Accessible **within the same package** and **in subclasses** (even in different packages).

Example:

```
class Car {
    protected String brand = "Toyota"; // Protected variable
}
```

d) Default (No Modifier)

Accessible **only within the same package**.

If no modifier is specified, it is considered **default**.

Example:

```
class Car {
    String brand = "Toyota"; // Default access
}
```

2. Non-Access Modifiers

Non-access modifiers provide **additional functionality** to variables, methods, and classes.

Modifier	Applied To	Purpose
`static`	Methods, Variables	Belongs to the class instead of an object
`final`	Classes, Methods, Variables	Prevents modification (constant values, no method overriding, no subclassing)
`abstract`	Classes, Methods	Defines an abstract class or method (to be implemented in a subclass)
`synchronized`	Methods, Blocks	Ensures thread safety
`volatile`	Variables	Ensures variable changes are visible across threads
`transient`	Variables	Prevents variable serialization
`strictfp`	Methods, Classes	Ensures floating-point consistency across platforms
`native`	Methods	Calls non-Java (e.g., C/C++) code

a) `static`

Used for **class-level** variables and methods.

Example:

```
class Car {
    static int wheels = 4; // Static variable
}
```

b) `final`

Final variable: Cannot be modified (constant).

```
final int MAX_SPEED = 120;
```

Final method: Cannot be overridden.

```
class Vehicle {
    final void start() { System.out.println("Starting..."); }
}
```

Final class: Cannot be extended.

```
final class Vehicle { }
```

c) `abstract`

Abstract class: Cannot be instantiated.

```
abstract class Animal {
    abstract void makeSound(); // Abstract method
}
```

Abstract method: Must be implemented by a subclass.

```
class Dog extends Animal {
    void makeSound() { System.out.println("Bark"); }
}
```

d) `synchronized`

Used for **thread safety**.

Example:

```
class BankAccount {
    synchronized void deposit(int amount) { /* synchronized code */ }
}
```

e) `volatile`

Ensures changes to a variable are visible across multiple threads.

Example:

```
class Example {
    volatile int counter = 0;
}
```

f) `transient`

Used to exclude variables from **serialization**.

Example:

```
class Employee implements Serializable {
    transient int SSN; // Will not be serialized
}
```

g) `strictfp`

Ensures consistent floating-point calculations across platforms.

Example:

```
strictfp class Calculator { }
```

h) `native`

Used for methods implemented in **C/C++**.

```
class NativeExample {
    native void sayHello();
}
```

Constructors in Java

A **constructor** in Java is a special method used to initialize objects. It is called automatically when an object of a class is created.

---

1. Characteristics of Constructors

- ☑ **Same name as the class**
- ☑ **No return type (not even `void`)**
- ☑ **Called automatically when an object is created**
- ☑ **Can be overloaded**

---

2. Types of Constructors

a) Default Constructor (No-Argument Constructor)

Automatically provided if no constructor is defined.

Used to initialize default values.

Example:

```
class Car {
    Car() { // Default Constructor
        System.out.println("Car is created");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car(); // Calls default constructor
    }
}
```

Output:

```
Car is created
```

---

b) Parameterized Constructor

Used to **initialize objects with specific values**.

Example:

```
class Car {
    String brand;
    int speed;

    Car(String b, int s) { // Parameterized Constructor
        brand = b;
        speed = s;
    }

    void display() {
        System.out.println("Brand: " + brand + ", Speed: " + speed);
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car("Toyota", 120);
        myCar.display();
    }
}
```

Output:

Brand: Toyota, Speed: 120

---

c) Copy Constructor (User-Defined)

Used to create a new object by **copying values from another object**.

Java **does not** provide a built-in copy constructor like C++, but we can create one manually.

Example:

```
class Car {
    String brand;
    int speed;

    Car(String b, int s) { // Parameterized Constructor
        brand = b;
        speed = s;
    }

    Car(Car c) { // Copy Constructor
        brand = c.brand;
        speed = c.speed;
    }

    void display() {
        System.out.println("Brand: " + brand + ", Speed: " + speed);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car("BMW", 150);
        Car car2 = new Car(car1); // Copy constructor
        car2.display();
    }
}
```

Output:

Brand: BMW, Speed: 150

---

d) Private Constructor

Used in **Singleton Pattern** (restricts object creation).

Example:

```
class Singleton {
    private static Singleton instance;

    private Singleton() { } // Private Constructor

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}

public class Main {
    public static void main(String[] args) {
        Singleton obj = Singleton.getInstance(); // Only one instance allowed
    }
}
```

---

3. Constructor Overloading

Multiple constructors in a class with **different parameters**.

Example:

```
class Car {
    String brand;
    int speed;

    // Default Constructor
```

```
    Car() {
        brand = "Unknown";
        speed = 0;
    }

    // Parameterized Constructor
    Car(String b, int s) {
        brand = b;
        speed = s;
    }

    void display() {
        System.out.println("Brand: " + brand + ", Speed: " + speed);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car(); // Calls default constructor
        Car car2 = new Car("Honda", 130); // Calls parameterized constructor

        car1.display();
        car2.display();
    }
}
```

Output:

Brand: Unknown, Speed: 0

Brand: Honda, Speed: 130

---

4. `this()` and `super()` in Constructors

a) Using `this()`

Calls **another constructor** in the **same class**.

Example:

```
class Car {
    String brand;
    int speed;

    Car() {
        this("DefaultBrand", 100); // Calls parameterized constructor
    }

    Car(String brand, int speed) {
        this.brand = brand;
        this.speed = speed;
    }

    void display() {
        System.out.println("Brand: " + brand + ", Speed: " + speed);
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car(); // Calls default constructor, which calls parameterized constructor
        myCar.display();
    }
}
```

Output:

Brand: DefaultBrand, Speed: 100

---

b) Using `super()`

Calls the **constructor of the parent class**.

Example:

```
class Vehicle {
```

```
    Vehicle() {
        System.out.println("Vehicle constructor called");
    }
}

class Car extends Vehicle {
    Car() {
        super(); // Calls parent class constructor
        System.out.println("Car constructor called");
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car(); // Calls parent constructor first
    }
}
```

Output:

```
Vehicle constructor called
Car constructor called
```

---

5. Differences Between Constructor and Method

Feature	Constructor	Method
Name	Same as class name	Any valid identifier
Return Type	No return type (not even `void`)	Must have a return type
Invocation	Called automatically when an object is created	Called explicitly
Purpose	Initializes an object	Defines behavior

Summary

☑ **Constructors initialize objects automatically.**
☑ **Types:** Default, Parameterized, Copy, Private.
☑ **Can be overloaded using different parameters.**

☑ **`this()` calls another constructor in the same class.**
☑ **`super()` calls the parent class constructor.**

Overloading in Java

Overloading in Java allows multiple methods or constructors in the same class to have the **same name but different parameters** (number, type, or order of parameters).

---

1. Method Overloading

Method overloading occurs when **multiple methods** in the same class have the **same name but different parameter lists**.

Rules for Method Overloading

☑ Methods must have the **same name**.
☑ Must have **different number, type, or order of parameters**.
☑ **Return type does not matter** (Overloading is based on parameters, not return type).

Example 1: Method Overloading

```
class MathOperations {
   // Method 1: Adds two numbers
   int add(int a, int b) {
      return a + b;
   }

   // Method 2: Adds three numbers
   int add(int a, int b, int c) {
      return a + b + c;
   }

   // Method 3: Adds two double numbers
   double add(double a, double b) {
      return a + b;
   }
```

```
}

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

        System.out.println(obj.add(5, 10));        // Calls method 1
        System.out.println(obj.add(5, 10, 15));    // Calls method 2
        System.out.println(obj.add(5.5, 2.5));     // Calls method 3
    }
}
```

```
15
30
8.0
```

Changing Parameter Order:

```
class Display {
    void show(int a, String b) {
        System.out.println("Integer: " + a + ", String: " + b);
    }

    void show(String b, int a) { // Different order
        System.out.println("String: " + b + ", Integer: " + a);
    }
}

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

```
        obj.show(10, "Java");  // Calls first method
        obj.show("Hello", 20); // Calls second method
    }
}
```

Output:

Integer: 10, String: Java

String: Hello, Integer: 20

2. Constructor Overloading

Constructor Overloading allows multiple constructors in the same class with different parameter lists.

Example: Constructor Overloading

```
class Car {
    String brand;
    int speed;

    // Default Constructor
    Car() {
        brand = "Unknown";
        speed = 0;
    }

    // Constructor with one parameter
    Car(String b) {
        brand = b;
        speed = 100;
    }

    // Constructor with two parameters
```

```
    Car(String b, int s) {
        brand = b;
        speed = s;
    }

    void display() {
        System.out.println("Brand: " + brand + ", Speed: " + speed);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car();          // Calls default constructor
        Car car2 = new Car("Honda");   // Calls constructor with one parameter
        Car car3 = new Car("BMW", 200); // Calls constructor with two parameters

        car1.display();
        car2.display();
        car3.display();
    }
}
```

Brand: Unknown, Speed: 0

Brand: Honda, Speed: 100

Brand: BMW, Speed: 200

3. Can We Overload `main()` Method?

Yes! **The `main()` method can be overloaded**, but only the standard `main(String[] args)` is used by Java as the entry point.

Example: Overloading `main()`

```
public class Main {
```

```
    public static void main(String[] args) {
        System.out.println("Main method with String[] args");
        main(10);
        main("Overloaded Main");
    }

    public static void main(int a) {
        System.out.println("Main method with int: " + a);
    }

    public static void main(String str) {
        System.out.println("Main method with String: " + str);
    }
}
```

Main method with String[] args

Main method with int: 10

Main method with String: Overloaded Main

Note: Java only executes `main(String[] args)`. Other `main()` methods must be called explicitly.

4. Method Overloading vs. Method Overriding

Feature	Method Overloading	Method Overriding
Definition	Same method name in the same class but different parameters.	Same method name and parameters in parent and child class.
Return Type	Can be different.	Must be the same or a subtype.
Access Modifiers	No restriction.	Cannot reduce visibility (e.g., `public` method in parent must stay `public`).
Keyword Used	No special keyword.	Uses `@Override` annotation.
Where Defined	Same class.	Parent and child class.

Summary

☑ **Overloading allows multiple methods or constructors with the same name but different parameters.**
☑ **Method overloading is based on parameter number, type, or order (not return type).**
☑ **Constructor overloading allows multiple ways to initialize an object.**
☑ **We can overload the `main()` method, but Java calls only the `main(String[] args)`.**

Polymorphism in Java

Polymorphism means **"many forms"** and allows **a single interface (method, function, or object) to represent different behaviors**.

Types of Polymorphism in Java

Compile-time Polymorphism (Method Overloading)

Runtime Polymorphism (Method Overriding)

---

1. Compile-time Polymorphism (Method Overloading)

- **Definition**: Multiple methods in the same class with the **same name but different parameters**.
- **Determined at compile-time** by the Java compiler.

Example: Method Overloading

```
class MathOperations {
    // Method 1: Adds two integers
    int add(int a, int b) {
        return a + b;
    }

    // Method 2: Adds three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method 3: Adds two double values
    double add(double a, double b) {
        return a + b;
```

```
    }
}

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

        System.out.println(obj.add(5, 10));       // Calls method 1
        System.out.println(obj.add(5, 10, 15));    // Calls method 2
        System.out.println(obj.add(5.5, 2.5));     // Calls method 3
    }
}
```

15

30

8.0

2. Runtime Polymorphism (Method Overriding)

- **Definition**: A **child class provides a specific implementation of a method that is already defined in the parent class**.
- **Determined at runtime** by the Java Virtual Machine (JVM).

Rules for Method Overriding

☑ **Same method name, same parameters.**
☑ **Return type must be the same (or a subclass – called covariant return type).**
☑ **The child class method must have the same or higher access level than the parent class.**
☑ **The method in the parent class must be `non-final` and `non-static`.**

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

class Dog extends Animal {
```

```
    // Overriding the makeSound() method
    @Override
    void makeSound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog(); // Runtime Polymorphism
        myAnimal.makeSound(); // Calls Dog's makeSound()
    }
}
```

Dog barks

Explanation: Even though `myAnimal` is of type `Animal`, at runtime the `makeSound()` method of `Dog` is executed.

3. Upcasting and Dynamic Method Dispatch

- **Upcasting**: Storing a child class object in a parent class reference.
- **Dynamic Method Dispatch**: Overridden methods are resolved at runtime based on the actual object type.

Example: Upcasting and Dynamic Method Dispatch

```
class Parent {
    void show() {
        System.out.println("Parent's show()");
    }
}

class Child extends Parent {
    @Override
    void show() {
```

```
        System.out.println("Child's show()");
    }
}

public class Main {
    public static void main(String[] args) {
        Parent obj = new Child(); // Upcasting
        obj.show(); // Calls Child's show() due to dynamic method dispatch
    }
}
```

```
Child's show()
```

4. Difference Between Overloading and Overriding

Feature	Method Overloading	Method Overriding
Definition	Multiple methods with the same name but different parameters in the same class.	A subclass provides a specific implementation of a method in the parent class.
Return Type	Can be different.	Must be the same or a subclass (covariant return type).
Parameters	Must be different (number, type, or order).	Must be exactly the same.
Access Modifier	No restrictions.	Cannot reduce visibility (`public` in the parent must stay `public`).
Static Methods	Can be overloaded.	Cannot be overridden.
Final Methods	Can be overloaded.	Cannot be overridden.

5. Polymorphism with Interfaces

- **Interfaces** allow different classes to have different implementations of the same method.

Example: Interface Polymorphism

```
interface Animal {
```

```
    void makeSound();
}

class Cat implements Animal {
    public void makeSound() {
        System.out.println("Cat meows");
    }
}

class Dog implements Animal {
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

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

        myAnimal = new Cat();
        myAnimal.makeSound(); // Calls Cat's makeSound()

        myAnimal = new Dog();
        myAnimal.makeSound(); // Calls Dog's makeSound()
    }
}
```

Cat meows Dog barks

6. Can We Override Static and Final Methods?

❌ **Static methods** cannot be overridden but can be hidden.
❌ **Final methods** cannot be overridden.

Example: Static Method Hiding

```
class Parent {
    static void display() {
        System.out.println("Static method in Parent");
    }
}

class Child extends Parent {
    static void display() { // This is method hiding, NOT overriding
        System.out.println("Static method in Child");
    }
}

public class Main {
    public static void main(String[] args) {
        Parent obj = new Child();
        obj.display(); // Calls Parent's static method (not Child's)
    }
}
```

Static method in Parent

🔹 **Explanation:** Static methods belong to the class, not objects, so method hiding occurs instead of overriding.

7. Summary

Polymorphism Types

Type	Example
Compile-time (Method Overloading)	Same method name, different parameters. Resolved at compile-time.
Runtime (Method Overriding)	Same method in parent and child class. Resolved at runtime.

Key Concepts

☑ **Method Overloading** - Same method name, different parameters.
☑ **Method Overriding** - Same method name and parameters, different implementations in parent and child class.
☑ **Upcasting & Dynamic Method Dispatch** - Parent class reference calls overridden methods in child class at runtime.
☑ **Polymorphism with Interfaces** - Different classes implementing the same interface behave differently.
☑ **Static Methods are Hidden, Not Overridden.**
☑ **Final Methods Cannot Be Overridden.**

Interface Implementation in Java

In Java, an **interface** is a blueprint for a class that defines a set of **abstract methods** (methods without a body) and **constants**. It allows multiple classes to follow the same behavior while supporting **multiple inheritance**.

1. Defining an Interface

- Use the `interface` keyword.
- Methods in an interface are **implicitly `public` and `abstract`**.
- Variables in an interface are **implicitly `public`, `static`, and `final`**.

Example: Creating an Interface

```
interface Animal {
    void makeSound(); // Abstract method (no body)
}
```

2. Implementing an Interface

- A class **implements** an interface using the `implements` keyword.
- The implementing class **must provide method definitions** for all interface methods.

Example: Implementing an Interface

```
// Defining an interface
interface Animal {
    void makeSound(); // Abstract method
}

// Implementing the interface in the Dog class
class Dog implements Animal {
    public void makeSound() {
```

```
        System.out.println("Dog barks");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeSound(); // Calls overridden method
    }
}
```

Dog barks

- **Note:** The method **must be declared `public`** in the implementing class because interface methods are implicitly `public`.

---

3. Multiple Interfaces (Multiple Inheritance)

- A class can **implement multiple interfaces** (since Java **does not** support multiple class inheritance).

Example: Multiple Interface Implementation

```
// First interface
interface Animal {
    void eat();
}

// Second interface
interface Pet {
    void play();
}

// Class implementing multiple interfaces
class Dog implements Animal, Pet {
```

```
    public void eat() {
        System.out.println("Dog eats");
    }

    public void play() {
        System.out.println("Dog plays");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat();  // Calls Animal's eat()
        myDog.play(); // Calls Pet's play()
    }
}
```

```
Dog eats
Dog plays
```

☑ **Multiple interface implementation allows Java to achieve multiple inheritance.**

---

4. Interface with Default and Static Methods (Java 8+)

- **Default methods** in interfaces **have a body** and **do not require overriding**.
- **Static methods** belong to the interface and cannot be overridden.

Example: Default and Static Methods in Interface

```
interface Vehicle {
    void start(); // Abstract method

    // Default method (can be overridden)
    default void fuelType() {
```

```
        System.out.println("Fuel type: Petrol/Diesel");
    }

    // Static method (cannot be overridden)
    static void serviceInfo() {
        System.out.println("Vehicle service every 6 months.");
    }
}

// Implementing the interface
class Car implements Vehicle {
    public void start() {
        System.out.println("Car starts");
    }

    // Overriding the default method
    public void fuelType() {
        System.out.println("Fuel type: Electric");
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.start();
        myCar.fuelType(); // Calls overridden method

        Vehicle.serviceInfo(); // Calls static method of interface
    }
}
```

Car starts

Fuel type: Electric

Vehicle service every 6 months.

☑ **Default methods allow interfaces to evolve without breaking existing implementations.**
☑ **Static methods in interfaces cannot be overridden.**

---

5. Interface Inheritance (Extending Interfaces)

- An interface **can extend another interface** using `extends`.
- A class implementing a sub-interface must implement all methods.

Example: Interface Inheritance

```
interface Animal {
    void eat();
}

// Interface extending another interface
interface WildAnimal extends Animal {
    void hunt();
}

// Implementing the sub-interface
class Tiger implements WildAnimal {
    public void eat() {
        System.out.println("Tiger eats");
    }

    public void hunt() {
        System.out.println("Tiger hunts");
    }
}
```

```
public class Main {
   public static void main(String[] args) {
      Tiger t = new Tiger();
      t.eat();
      t.hunt();
   }
}
```

Tiger eats

Tiger hunts

☑ **Interface inheritance allows interface reuse and better abstraction.**

6. Interface vs. Abstract Class

Feature	Interface	Abstract Class
Methods	Only abstract (Java 8+ allows default/static methods).	Can have both abstract and concrete methods.
Variables	`public`, `static`, `final` only.	Can have instance variables (not necessarily final).
Constructors	No constructors.	Can have constructors.
Multiple Inheritance	A class can implement multiple interfaces.	A class can extend only one abstract class.
Access Modifiers	Methods are always `public`.	Methods can be `private`, `protected`, or `public`.
Usage	Used to define a contract (behavior).	Used when a class provides some base behavior.

7. Functional Interfaces (Java 8)

- A **functional interface** has **only one abstract method**.
- Used in **lambda expressions**.
- Marked with `@FunctionalInterface` annotation.

Example: Functional Interface with Lambda Expression

```
@FunctionalInterface
interface Greetings {
```

```
    void sayHello(String name);
}

public class Main {
    public static void main(String[] args) {
        // Implementing using lambda expression
        Greetings greet = (name) -> System.out.println("Hello, " + name);

        greet.sayHello("John"); // Calls the lambda function
    }
}
```

Hello, John

8. Summary

☑ **Interfaces** define a contract for classes to implement.
☑ A **class implements an interface** using `implements`.
☑ Java **supports multiple interfaces** but only **single inheritance** for classes.
☑ **Default and static methods (Java 8+)** allow method implementation inside interfaces.
☑ **Functional interfaces** support lambda expressions.

Encapsulation in Java

Encapsulation is one of the four fundamental OOP concepts in Java. It refers to the **bundling of data (variables) and methods (functions) together within a class**, while restricting direct access to some of the object's details.

Key Principles of Encapsulation

☑ Data is hidden using **private** access modifiers.
☑ Data can only be accessed through **public getter and setter methods**.
☑ It improves **security** and **data integrity** by preventing unauthorized modifications.
☑ It allows **controlled access** to variables.

---

1. Example of Encapsulation

Encapsulated Class

```
class Person {
```

```
    // Private variables (data hiding)
    private String name;
    private int age;

    // Public getter method for name
    public String getName() {
        return name;
    }

    // Public setter method for name
    public void setName(String name) {
        this.name = name;
    }

    // Public getter method for age
    public int getAge() {
        return age;
    }

    // Public setter method for age (validating age)
    public void setAge(int age) {
        if (age > 0) {
            this.age = age;
        } else {
            System.out.println("Age must be positive.");
        }
    }
}

public class Main {
    public static void main(String[] args) {
```

```
        Person p = new Person();

        // Setting values using setters
        p.setName("John");
        p.setAge(25);

        // Getting values using getters
        System.out.println("Name: " + p.getName());
        System.out.println("Age: " + p.getAge());
    }
}
```

Name: John

Age: 25

☑ **Encapsulation prevents direct access to private data** while allowing controlled access through getter and setter methods.

---

2. Benefits of Encapsulation

- Data Hiding

Internal data is **hidden from external access**, preventing unintended modifications.

- Data Validation

Setter methods **validate inputs** before assigning values.

```
public void setAge(int age) {
    if (age > 0) { // Ensuring only valid ages are set
        this.age = age;
    } else {
        System.out.println("Invalid age!");
    }
}
```

- Maintainability & Flexibility

Changes to internal implementation **don't affect external code** using the class.

- Security

Prevents unauthorized access and modification of important data.

---

3. Encapsulation with Read-Only and Write-Only Properties

Read-Only Class (No Setters)

```
class Employee {
    private String empID = "E123"; // Private data

    // Getter only (No Setter)
    public String getEmpID() {
        return empID;
    }
}

public class Main {
    public static void main(String[] args) {
        Employee e = new Employee();
        System.out.println("Employee ID: " + e.getEmpID()); // Only reading
    }
}
```

☑ **This makes the variable read-only. It cannot be changed outside the class.**

Write-Only Class (No Getters)

```
class BankAccount {
    private double balance;

    // Setter only (No Getter)
    public void setBalance(double amount) {
        if (amount >= 0) {
            balance = amount;
        } else {
            System.out.println("Invalid amount!");
```

```
    }
  }
}

public class Main {
  public static void main(String[] args) {
    BankAccount account = new BankAccount();
    account.setBalance(5000); // Can set value but cannot read it
  }
}
```

☑ **This makes the variable write-only. It cannot be read outside the class.**

---

4. Encapsulation vs Data Hiding

Feature	Encapsulation	Data Hiding
Definition	Wrapping data and methods in a single unit.	Restricting access to internal details of a class.
Purpose	Improve maintainability and modularity.	Protect sensitive data from external modification.
Implementation	Achieved through private variables and public getter/setter methods.	Achieved using access modifiers (private/protected).

---

5. Access Modifiers & Encapsulation

Modifier	Class	Package	Subclass	World
private	☑	✗	✗	✗
default (no modifier)	☑	☑	✗	✗
protected	☑	☑	☑	✗
public	☑	☑	☑	☑

☑ **Encapsulation is achieved by using `private` variables and `public` methods**.

---

6. Encapsulation in Real-Life Scenarios

- Example: Bank Account System

```
class BankAccount {
    private double balance; // Private balance

    // Constructor to initialize balance
    public BankAccount(double balance) {
        if (balance >= 0) {
            this.balance = balance;
        } else {
            System.out.println("Invalid initial balance.");
            this.balance = 0;
        }
    }

    // Getter method for balance
    public double getBalance() {
        return balance;
    }

    // Deposit method (Validates amount)
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println("Deposited: " + amount);
        } else {
            System.out.println("Invalid deposit amount!");
        }
    }

    // Withdraw method (Validates withdrawal)
    public void withdraw(double amount) {
```

```
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            System.out.println("Withdrawn: " + amount);
        } else {
            System.out.println("Insufficient funds or invalid amount!");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        BankAccount myAccount = new BankAccount(1000); // Initial balance

        myAccount.deposit(500);
        myAccount.withdraw(300);
        System.out.println("Current Balance: " + myAccount.getBalance());
    }
}
```

Deposited: 500

Withdrawn: 300

Current Balance: 1200

☑ **Encapsulation ensures that the balance cannot be accessed directly. All transactions go through controlled methods.**

---

7. Summary

Feature	**Encapsulation**
Definition	Wrapping data and methods together while restricting direct access.
Implemented Using	Private variables and public getter/setter methods.

Feature	Encapsulation
Benefits	Data security, validation, maintainability, and flexibility.
Examples	Bank account balance, Student details, Employee records.

---

8. Why Use Encapsulation?

✔ **Prevents direct modification of data (data hiding).**
✔ **Allows controlled access via getters/setters.**
✔ **Improves security and maintains integrity.**
✔ **Enhances code reusability and maintainability.**
✔ **Encourages abstraction and modular design.**

Inheritance in Java

Inheritance is one of the core concepts of **Object-Oriented Programming (OOP)** in Java. It allows a class to **inherit** fields and methods from another class, enabling **code reusability** and **hierarchical relationships**.

---

1. What is Inheritance?

☑ **Inheritance** allows a class (**child/subclass**) to acquire properties (fields) and behaviors (methods) from another class (**parent/superclass**).
☑ Promotes **code reusability** and **reduces redundancy**.
☑ Allows **method overriding** to modify inherited behavior.

Syntax: Using `extends` Keyword

```
class Parent {
   // Parent class properties and methods
}

class Child extends Parent {
   // Child class inherits Parent class
}
```

2. Types of Inheritance in Java

Java supports the following types of inheritance:

1️⃣ **Single Inheritance**
2️⃣ **Multilevel Inheritance**
3️⃣ **Hierarchical Inheritance**
4️⃣ **Hybrid Inheritance (Achieved using Interfaces)**

❌ **Multiple Inheritance** (via classes) is **not** supported in Java to avoid ambiguity.

---

3. Single Inheritance

- A child class inherits from a **single parent class**.
- Child class can use all **non-private** members of the parent class.

Example of Single Inheritance

```
// Parent class
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

// Child class inheriting Animal class
class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat();  // Inherited method
        myDog.bark(); // Child's own method
    }
}
```

This animal eats food.

Dog barks.

☑ **Child class gets access to `eat()` method from the parent class.**

---

4. Multilevel Inheritance

- A class inherits from another **child class**, forming a chain.

Example of Multilevel Inheritance

```
// Parent class
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

// Intermediate child class
class Mammal extends Animal {
    void breathe() {
        System.out.println("Mammal breathes.");
    }
}

// Child class inheriting Mammal class
class Dog extends Mammal {
    void bark() {
        System.out.println("Dog barks.");
    }
}

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

```
        myDog.eat();    // Inherited from Animal
        myDog.breathe(); // Inherited from Mammal
        myDog.bark();   // Own method
    }
}
```

This animal eats food.

Mammal breathes.

Dog barks.

☑ **Dog inherits properties from both Animal and Mammal classes.**

---

5. Hierarchical Inheritance

- **Multiple child classes** inherit from a **single parent class**.

Example of Hierarchical Inheritance

```
// Parent class
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

// Child class 1
class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks.");
    }
}

// Child class 2
class Cat extends Animal {
```

```
    void meow() {
        System.out.println("Cat meows.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat();
        myDog.bark();

        Cat myCat = new Cat();
        myCat.eat();
        myCat.meow();
    }
}
```

```
This animal eats food.
Dog barks.
This animal eats food.
Cat meows.
```

6. Multiple Inheritance (Not Supported)

- Java **does not** support **multiple inheritance** with classes (to avoid ambiguity).
- **Example of ambiguity in multiple inheritance:**

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

```
class B {
   void show() {
      System.out.println("Class B");
   }
}
```

// ❌ This is NOT allowed in Java

```
class C extends A, B {  // Error: Multiple inheritance not supported
}
```

Solution: Java allows **multiple inheritance using interfaces** instead.

7. Hybrid Inheritance (Using Interfaces)

- A mix of **single, multilevel, and multiple inheritance**.
- **Achieved using interfaces**.

Example of Hybrid Inheritance

```
// Parent interface 1
interface Vehicle {
    void start();
}

// Parent interface 2
interface Engine {
    void fuelType();
}

// Child class implementing multiple interfaces
class Car implements Vehicle, Engine {
    public void start() {
        System.out.println("Car starts.");
    }

    public void fuelType() {
        System.out.println("Fuel type: Petrol.");
```

```
    }
}

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

Car starts.

Fuel type: Petrol.

☑ **Car inherits behavior from multiple interfaces.**

---

8. `super` Keyword in Inheritance

- **`super`** is used to refer to the **parent class's methods and constructors**.

Using `super` to Call Parent's Method

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

class Dog extends Animal {
  void makeSound() {
    super.makeSound(); // Calls parent class method
    System.out.println("Dog barks.");
  }
}
```

```
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeSound();
    }
}
```

Animal makes a sound.

Dog barks.

9. Method Overriding in Inheritance

- **Child class redefines parent class method** for customized behavior.
- Must have **same method signature** as the parent method.

Example: Method Overriding

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

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

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();
        myAnimal.makeSound(); // Calls overridden method in Dog class
    }
}
```

Dog barks.

☑ **`@Override` annotation ensures the method correctly overrides the parent's method.**

---

10. Summary

Feature	**Description**
What is Inheritance?	Acquiring properties and methods from another class.
Keyword Used	`extends` (for classes), `implements` (for interfaces).
Types	Single, Multilevel, Hierarchical, Hybrid (via interfaces).
Method Overriding?	Yes, using `@Override`.
Multiple Inheritance?	Not supported (but can be achieved via interfaces).
Key Benefit	Code reusability and maintainability.

---

11. Why Use Inheritance?

✔ **Reduces code duplication** (reusability).
✔ **Simplifies code structure** (hierarchy).
✔ **Supports polymorphism** (method overriding).
✔ **Enhances code maintenance**.

Method Overriding in Java

Method Overriding is a feature in Java that allows a subclass to **provide a specific implementation** of a method already defined in its superclass. It enables **runtime polymorphism** and allows a subclass to modify the behavior of its inherited methods.

---

1. What is Method Overriding?

☑ A **subclass provides a specific implementation** of a method that is already defined in its **superclass**.
☑ The method in the **subclass must have the same signature** (method name, parameters, and return type) as in the superclass.
☑ It allows **dynamic method dispatch** (runtime polymorphism).
☑ The **`@Override` annotation** is used to ensure correct overriding.

---

2. Rules for Method Overriding

- **Method must have the same name, return type, and parameters as the parent method.**
- **Access modifier in subclass must be the same or more permissive** (e.g., `protected` → `public`).
- **Methods declared `private`, `final`, or `static` cannot be overridden.**
- The overridden method should be **inherited from a superclass**.
- The overriding method can throw **fewer or no exceptions** but **not more** than the superclass method.

---

3. Basic Example of Method Overriding

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

// Child class overriding the method
class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog(); // Upcasting
        myAnimal.makeSound(); // Calls overridden method in Dog class
    }
}
```

Dog barks.

☑ The `makeSound()` method in the `Dog` class **overrides** the one in `Animal`.
☑ **Upcasting (`Animal myAnimal = new Dog();`) allows dynamic method dispatch.**

---

4. Using `super` to Call Parent Class Method

- **`super.methodName()`** can be used to **call the parent class's version** of the overridden method.

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

class Dog extends Animal {
    @Override
    void makeSound() {
        super.makeSound(); // Calls parent class method
        System.out.println("Dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeSound();
    }
}
```

```
Animal makes a sound.
Dog barks.
```

☑ **`super.makeSound()` calls the parent's method before executing the child's method.**

---

5. Access Modifiers in Overriding

Parent Class Modifier	Allowed in Child Class?
`private`	❌ Cannot be overridden
`default`	☑ `default`, `protected`, `public`
`protected`	☑ `protected`, `public`
`public`	☑ `public` (Cannot reduce visibility)

Example of Access Modifier Rule

```
class Parent {
    protected void show() {
        System.out.println("Parent class method.");
    }
}

class Child extends Parent {
    @Override
    public void show() { // Access modifier changed from protected to public
        System.out.println("Child class method.");
    }
}

public class Main {
    public static void main(String[] args) {
        Child obj = new Child();
        obj.show();
    }
}
```

☑ **You can increase visibility (`protected` → `public`) but not decrease it.**
❌ **Changing `public` to `protected`/`private` would cause an error.**

---

6. Overriding with Different Return Type (Covariant Return)

- **Child class method can return a subclass of the parent method's return type**.

Example: Covariant Return Type

```
class Parent {
    Parent getInstance() {
        return new Parent();
    }
}

class Child extends Parent {
    @Override
    Child getInstance() { // Covariant return type
        return new Child();
    }
}

public class Main {
    public static void main(String[] args) {
        Child obj = new Child();
        obj.getInstance();
    }
}
```

☑ **Child class returns `Child` instead of `Parent`, which is allowed.**

---

7. Static Methods Cannot Be Overridden (Method Hiding)

- **Static methods belong to the class, not instances, so they cannot be overridden.**
- If a subclass defines a static method with the same signature, it **hides** the superclass method instead.

Example of Method Hiding

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

```
}

class Child extends Parent {
    static void show() { // Method hiding, not overriding
        System.out.println("Static method in Child");
    }
}

public class Main {
    public static void main(String[] args) {
        Parent obj = new Child();
        obj.show(); // Calls Parent's method (not overridden)
    }
}
```

Static method in Parent

☑ **Static methods do not support runtime polymorphism (overriding).**

---

8. Final Methods Cannot Be Overridden

- A method marked as `final` **cannot** be overridden by a subclass.

Example

```
class Parent {
    final void display() {
        System.out.println("Final method in Parent");
    }
}

class Child extends Parent {
    // ❌ Compilation Error: Cannot override final method
    // void display() {
    //     System.out.println("Trying to override");
    // }
```

```
}

public class Main {
    public static void main(String[] args) {
        Child obj = new Child();
        obj.display();
    }
}
```

☑ **Final methods are meant to be preserved as they are.**

---

9. Method Overriding with Exception Handling

- The **overriding method cannot throw a broader checked exception** than the parent method.

Valid Example

```
class Parent {
    void show() throws IOException { // Checked Exception
        System.out.println("Parent method");
    }
}

class Child extends Parent {
    @Override
    void show() throws FileNotFoundException { // Allowed (subclass exception)
        System.out.println("Child method");
    }
}
```

☑ **Subclass can throw the same or a more specific exception.**

---

10. Summary

Feature	Method Overriding
Purpose	Allow a subclass to provide a specific implementation of a method from its parent class.
Runtime Polymorphism?	☑ Yes
Keyword Used?	`@Override` annotation (optional but recommended)
Access Modifier Rule?	Cannot reduce visibility (e.g., `public` → `protected`)
Final Methods?	❌ Cannot be overridden
Static Methods?	❌ Cannot be overridden (Method Hiding instead)
Private Methods?	❌ Cannot be overridden
Exceptions Rule?	☑ Subclass can throw same or **narrower** checked exceptions

---

11. Why Use Method Overriding?

✔ **Allows dynamic method dispatch (runtime polymorphism).**
✔ **Enables different behaviors in subclasses using the same method.**
✔ **Supports code reusability and maintainability.**
✔ **Improves flexibility in designing object-oriented systems.**

Cohesion in Java

Cohesion is a key principle in **Object-Oriented Programming (OOP)** that refers to **how closely related and focused the responsibilities of a class or method are**. A highly cohesive class or method **performs a single, well-defined task**, making the code **more maintainable, reusable, and easier to understand**.

---

1. What is Cohesion?

☑ **Cohesion measures the degree to which the elements of a module/class belong together.**
☑ **High cohesion** means a class has a **single, well-defined purpose**.
☑ **Low cohesion** means a class has **too many unrelated responsibilities**, making it **harder to maintain and modify**.

Analogy:

📌 A **highly cohesive** class is like a **sharp knife** 🚀 —designed for a **single purpose**, making it efficient.
📌 A **low cohesive** class is like a **Swiss Army knife** 🛠️ —it does **many things but is not specialized**.

---

2. Types of Cohesion

Cohesion can be categorized as **low to high** based on how strongly related the functionalities in a class or module are.

Low Cohesion (Bad Design)

- 🔹 **Coincidental Cohesion:** Unrelated tasks grouped randomly.
- 🔹 **Logical Cohesion:** Grouped by logic but not necessarily related.
- 🔹 **Temporal Cohesion:** Methods run at the same time but serve different purposes.

High Cohesion (Good Design)

- 🔹 **Functional Cohesion:** Every method contributes to a single function.
- 🔹 **Sequential Cohesion:** Methods work in sequence where one method's output is the next method's input.
- 🔹 **Communicational Cohesion:** Methods operate on the same data.

---

3. Example of Low Cohesion (Bad Design)

🔴 **Problem:** The `Utility` class below **performs multiple unrelated tasks** (logging, file handling, and math operations).
🔴 **Issue:** Difficult to maintain, modify, and test.

```
class Utility {
    void logMessage(String message) {
        System.out.println("Logging: " + message);
    }

    void saveToFile(String data) {
        System.out.println("Saving data to file: " + data);
    }

    int addNumbers(int a, int b) {
        return a + b;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Utility util = new Utility();
        util.logMessage("App started");
        util.saveToFile("User data");
        System.out.println(util.addNumbers(5, 10));
    }
}
```

Problems with Low Cohesion:

❌ **Unrelated responsibilities** in one class.
❌ **Difficult to maintain** (e.g., changing logging affects file handling).
❌ **Violates Single Responsibility Principle (SRP)** in **SOLID principles**.

4. Example of High Cohesion (Good Design)

🔵 **Solution:** Split `Utility` into **highly cohesive** classes based on functionality.

```
// Handles only logging-related tasks
class Logger {
    void logMessage(String message) {
        System.out.println("Logging: " + message);
    }
}

// Handles only file-related tasks
class FileHandler {
    void saveToFile(String data) {
        System.out.println("Saving data to file: " + data);
    }
}

// Handles only mathematical operations
```

```
class MathOperations {
    int addNumbers(int a, int b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Logger logger = new Logger();
        FileHandler fileHandler = new FileHandler();
        MathOperations mathOps = new MathOperations();

        logger.logMessage("App started");
        fileHandler.saveToFile("User data");
        System.out.println(mathOps.addNumbers(5, 10));
    }
}
```

Benefits of High Cohesion:

☑ **Each class has a single, well-defined responsibility.**
☑ **Easier to maintain, test, and extend.**
☑ **Follows the Single Responsibility Principle (SRP).**

---

5. Cohesion vs Coupling

Aspect	**Cohesion (High is Good)**	**Coupling (Low is Good)**
Definition	How focused a class/module is on a single task.	How dependent a class/module is on others.
Goal	Increase cohesion to make modules self-contained.	Reduce coupling to make modules independent.
Best Practice	Design each class with a single responsibility.	Minimize dependencies between classes.
Example	`Logger` class should only handle logging.	Avoid one class depending heavily on another.

☑ **Good Design = High Cohesion + Low Coupling**

---

6. Key Takeaways

✔ **Cohesion refers to how well elements in a class are related to each other.**
✔ **High cohesion is preferred** because it makes code **easier to maintain, test, and reuse**.
✔ **Low cohesion leads to spaghetti code**, making it hard to understand and modify.
✔ **Follow the Single Responsibility Principle (SRP) to ensure high cohesion.**
✔ **Strive for high cohesion and low coupling for better software design.**

Casting in Java

Casting in Java refers to **converting one data type into another**. It is used when we need to convert variables between different types, either within primitive types or between objects.

---

1. Types of Casting in Java

Casting is mainly divided into **two categories**:

1 **Primitive Casting** (for primitive data types)
2 **Object Casting** (for reference types, i.e., objects)

---

2. Primitive Type Casting

- Converts **one primitive type** into another.
- Java supports **two types** of primitive casting:

(i) Implicit Casting (Widening) – Automatic

☑ **Smaller data type → Larger data type** (No data loss)
☑ **Happens automatically** (no explicit syntax needed)

From (Smaller Type)	To (Larger Type)

`byte` → `short` → `int` → `long` → `float` → `double`

Example of Implicit Casting (Widening)

```
public class Main {
    public static void main(String[] args) {
        int num = 10;
        double d = num; // Implicit casting (int to double)
        System.out.println(d); // Output: 10.0
```

```
    }
}
```

(ii) Explicit Casting (Narrowing) – Manual

❌ **Larger data type → Smaller data type** (Possible data loss)
❌ **Must be done explicitly using `(type)`**

From (Larger Type) To (Smaller Type)

`double` ⟶ `float` ⟶ `long` ⟶ `int` ⟶ `short` ⟶ `byte`

```
public class Main {
    public static void main(String[] args) {
        double d = 10.5;
        int num = (int) d; // Explicit casting (double to int)
        System.out.println(num); // Output: 10 (Decimal part is lost)
    }
}
```

🚨 **Data loss occurs because narrowing removes decimal values**.

---

3. Object Type Casting (Reference Type Casting)

- Used when **working with objects** and **inheritance**.
- Can be of **two types**:

(i) Upcasting (Widening)

☑ **Subclass object → Superclass reference**
☑ **Happens automatically**
☑ **Allows accessing only superclass methods**

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

class Dog extends Animal {
```

```
    void bark() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a = new Dog(); // Upcasting (Subclass → Superclass)
        a.makeSound(); // Allowed
        // a.bark(); ❌ Not allowed (because 'a' is treated as an Animal)
    }
}
```

(ii) Downcasting (Narrowing)

❌ **Superclass reference → Subclass reference**
❌ **Needs explicit casting**
❌ **May throw `ClassCastException` if not used correctly**

```
public class Main {
    public static void main(String[] args) {
        Animal a = new Dog(); // Upcasting
        Dog d = (Dog) a; // Downcasting (Explicitly cast back to Dog)
        d.bark(); // ☑ Now we can call Dog's specific method
    }
}
```

🚨 **If downcasting is done incorrectly, it will throw `ClassCastException`.**

---

4. `instanceof` Operator – Safe Downcasting

To avoid `ClassCastException`, use `instanceof` before downcasting:

```
if (a instanceof Dog) {
    Dog d = (Dog) a; // Safe downcasting
    d.bark();
}
```

☑ **Prevents runtime errors by checking type compatibility.**

---

5. Summary Table

Type of Casting	Conversion	Automatic?	Data Loss?
Implicit (Widening) Primitive Casting	Small → Large	☑ Yes	❌ No
Explicit (Narrowing) Primitive Casting	Large → Small	❌ No	☑ Yes
Upcasting (Object Casting)	Subclass → Superclass	☑ Yes	❌ No
Downcasting (Object Casting)	Superclass → Subclass	❌ No (Needs explicit cast)	❌ No (If checked with `instanceof`)

---

6. Key Takeaways

✔ **Primitive Casting** is used for converting primitive types.
✔ **Widening (Implicit)** is safe, while **Narrowing (Explicit)** may cause data loss.
✔ **Object Casting** is used for **inheritance-based polymorphism**.
✔ **Upcasting happens automatically**, while **Downcasting requires explicit casting**.
✔ **Use `instanceof` before downcasting** to avoid runtime errors.

Arrays in Java

An **array** in Java is a **fixed-size, ordered collection** of elements of the **same data type**. Arrays are used to store multiple values in a single variable instead of declaring separate variables for each value.

---

1. Declaring and Initializing Arrays

(i) Declaration

In Java, an array is declared using square brackets `[]`:

dataType[] arrayName;

(ii) Instantiation (Memory Allocation)

After declaration, an array must be initialized using the `new` keyword:

arrayName = new dataType[size];

(iii) Declaration + Initialization (Shortcut)

int[] numbers = new int[5]; // Creates an integer array of size 5

Alternatively, you can directly assign values:

int[] numbers = {10, 20, 30, 40, 50}; // Declares and initializes in one step

2. Accessing Array Elements

Each element in an array is accessed using **indexing**, starting from `0`.

```
public class Main {
   public static void main(String[] args) {
      int[] numbers = {10, 20, 30, 40, 50};
      System.out.println(numbers[0]); // Output: 10
      System.out.println(numbers[3]); // Output: 40
   }
}
```

🚨 **Accessing an invalid index (`numbers[5]`) results in `ArrayIndexOutOfBoundsException`.**

3. Modifying Array Elements

You can modify array elements by assigning new values:

```
numbers[1] = 25;  // Changing the second element (index 1) to 25
System.out.println(numbers[1]); // Output: 25
```

4. Array Length Property

Java arrays have a `length` property to get the **number of elements**:

```
System.out.println(numbers.length); // Output: 5
```

5. Looping Through Arrays

(i) Using a `for` Loop

```
for (int i = 0; i < numbers.length; i++) {
   System.out.println(numbers[i]);
}
```

(ii) Using an Enhanced `for` Loop (for-each loop)

```
for (int num : numbers) {
    System.out.println(num);
}
```

☑ **Simpler syntax but doesn't provide access to indexes.**

---

6. Multidimensional Arrays

Java supports **2D and multi-dimensional arrays**, represented as arrays of arrays.

(i) Declaring a 2D Array

```
int[][] matrix = new int[3][3]; // Creates a 3x3 matrix
```

(ii) Initializing a 2D Array

```
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

(iii) Accessing 2D Array Elements

```
System.out.println(matrix[1][2]); // Output: 6
```

(iv) Iterating Through a 2D Array

```
for (int i = 0; i < matrix.length; i++) {
    for (int j = 0; j < matrix[i].length; j++) {
        System.out.print(matrix[i][j] + " ");
    }
    System.out.println();
}
```

7. Arrays vs ArrayList

Feature	**Array**	**ArrayList**
Size	Fixed size (cannot change)	Dynamic size (resizable)

Feature	Array	ArrayList
Type	Can store primitives & objects	Only stores objects (e.g., `Integer`, `String`)
Performance	Faster for fixed-size data	Slower due to resizing overhead
Usage	Used for performance-sensitive applications	Used when flexible size is needed

Example of `ArrayList` (Alternative to Arrays)

```
import java.util.ArrayList;


public class Main {
   public static void main(String[] args) {
      ArrayList<Integer> list = new ArrayList<>();
      list.add(10);
      list.add(20);
      list.add(30);
      System.out.println(list.get(1)); // Output: 20
   }
}
```

8. Key Takeaways

✔ Arrays **store multiple values** of the **same type**.
✔ Array **indexing starts from 0**.
✔ Arrays have **fixed sizes** (use `ArrayList` for dynamic resizing).
✔ Use **loops** (`for`, `for-each`) to iterate over arrays.
✔ **Multidimensional arrays** allow storing tabular data.

Wrapper Classes in Java

Wrapper classes in Java **convert primitive data types into objects** and provide additional functionalities. They are part of the `java.lang` package.

---

1. Why Use Wrapper Classes?

Convert Primitives to Objects (Needed for Collections like `ArrayList`)

Provide Utility Methods (e.g., `Integer.parseInt()`, `Double.valueOf()`)

Support Autoboxing & Unboxing (Automatic conversion between primitive and object)

---

2. List of Wrapper Classes

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

---

3. Creating Wrapper Objects

(i) Using Constructor (Deprecated)

Integer obj1 = new Integer(10); // Deprecated in Java 9

(ii) Using `valueOf()` Method (Recommended)

Integer obj2 = Integer.valueOf(10); // ☑ Recommended

4. Autoboxing and Unboxing

(i) Autoboxing (Primitive → Wrapper)

Integer num = 100; // Automatically converts int to Integer

(ii) Unboxing (Wrapper → Primitive)

int x = num; // Automatically converts Integer to int

5. Wrapper Class Methods

Wrapper classes provide useful utility methods:

(i) `parseXxx()` - Convert String to Primitive

int num = Integer.parseInt("100"); // String to int

```
double d = Double.parseDouble("10.5");  // String to double
```

(ii) `valueOf()` - Convert String to Wrapper Object

```
Integer obj = Integer.valueOf("200");  // String to Integer object
```

(iii) `toString()` - Convert Primitive to String

```
String s = Integer.toString(123);  // int to String
```

(iv) `xxxValue()` - Convert Wrapper to Another Type

```
Double d = 10.5;
int x = d.intValue();  // Convert Double to int
```

Example Program

```
public class WrapperExample {
    public static void main(String[] args) {
        // Autoboxing
        Integer intObj = 50;

        // Unboxing
        int intVal = intObj;

        // Using parseInt and valueOf
        int num1 = Integer.parseInt("123");
        Integer num2 = Integer.valueOf("456");

        // Printing results
        System.out.println("Autoboxed Integer: " + intObj);
        System.out.println("Unboxed int: " + intVal);
        System.out.println("Parsed int: " + num1);
        System.out.println("ValueOf Integer: " + num2);
    }
}
```

```
Autoboxed Integer: 50
Unboxed int: 50
```

Parsed int: 123

ValueOf Integer: 456

7. Key Takeaways

✔ **Wrapper classes** convert **primitives into objects**.
✔ **Autoboxing** (Automatic primitive → Wrapper) & **Unboxing** (Wrapper → Primitive).
✔ Use **`parseXxx()` for String to primitive conversion**.
✔ Use **`valueOf()` for String to Wrapper conversion**.
✔ Useful in **Collections, Streams, and Generics**.

Garbage Collection in Java 🗑️ ♻️

Garbage Collection (GC) in Java **automatically manages memory** by removing unused objects to free up space. This helps prevent memory leaks and improves performance.

---

1. Why Garbage Collection?

☑ **Automatic Memory Management** – No need to manually free memory like in C/C++.
☑ **Prevents Memory Leaks** – Removes unreachable objects.
☑ **Enhances Performance** – Frees up heap memory automatically.

---

2. How Does Garbage Collection Work?

Java's **JVM (Java Virtual Machine)** has a **Garbage Collector (GC)** that automatically identifies and removes objects **no longer reachable** by the program.

🔹 **Heap Memory**:

Young Generation (Eden, Survivor Space) – New objects are allocated here.

Old Generation (Tenured) – Long-lived objects are moved here.

Permanent Generation (MetaSpace in Java 8+) – Stores class metadata.

🔹 **Garbage Collection Process:**

1️⃣ **Mark** – Identifies unreachable objects.
2️⃣ **Sweep** – Removes unreferenced objects.
3️⃣ **Compact** – Rearranges memory to optimize performance.

---

3. Types of Garbage Collectors in Java

Java provides different garbage collectors for various needs:

GC Type	Description
Serial GC	Simple, uses a single thread (Best for small apps).
Parallel GC	Uses multiple threads for faster performance.
G1 (Garbage-First) GC	Balances performance and efficiency (Default in Java 9+).
ZGC (Z Garbage Collector)	Low-latency, scalable for large heaps (Java 11+).

---

4. When Does Garbage Collection Happen?

✔ **When JVM needs memory** (Not predictable)
✔ **Calling `System.gc()`** (Only a suggestion, not guaranteed)
✔ **Object Becomes Unreachable** (No references remain)

---

5. How to Make Objects Eligible for GC?

(i) Nullifying References

```
Object obj = new Object();
obj = null; // Now eligible for GC
```

(ii) Reassigning References

```
String s1 = new String("Hello");
String s2 = new String("World");
s1 = s2; // "Hello" is now eligible for GC
```

(iii) Objects Inside Methods

```
void someMethod() {
    Employee emp = new Employee(); // emp is local
} // emp is eligible for GC after method exits
```

---

6. `finalize()` Method (Deprecated in Java 9)

- **`finalize()` was used before GC removed an object**, but it's unreliable and **deprecated**.

Example:

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

☑ **Better Alternative:** Use `try-with-resources` or `close()` method for cleanup.

---

7. Forcing Garbage Collection (Not Recommended)

```
System.gc(); // Requests garbage collection (not guaranteed)
```

⚠️ **JVM decides whether to run GC, so this is just a request!**

---

8. Example: Demonstrating Garbage Collection

```
public class GarbageCollectionExample {
    public static void main(String[] args) {
        GarbageCollectionExample obj1 = new GarbageCollectionExample();
        GarbageCollectionExample obj2 = new GarbageCollectionExample();

        obj1 = null; // Eligible for GC
        obj2 = null; // Eligible for GC

        System.gc(); // Request for Garbage Collection
    }

    @Override
    protected void finalize() {
        System.out.println("Garbage Collected!");
    }
}
```

☑ **Possible Output (Not guaranteed):**

```
Garbage Collected!
Garbage Collected!
```

---

9. Key Takeaways

✔ **Automatic memory management** – No manual deallocation.
✔ **JVM removes unreachable objects** to free memory.
✔ **Several GC types exist** (Serial, Parallel, G1, ZGC).
✔ **Use `System.gc()` cautiously** – JVM decides when to collect.
✔ **Avoid using `finalize()`**, use **try-with-resources** instead.

Exception Handling in Java

Exception handling in Java is a mechanism used to handle runtime errors, ensuring the program continues execution without crashing unexpectedly.

---

1. What is an Exception?

An **exception** is an event that occurs during the execution of a program, disrupting the normal flow. It can be caused by various reasons such as invalid user input, file not found, network failures, database connection issues, etc.

---

2. Exception Hierarchy in Java

All exception classes are subclasses of `Throwable`, which has two direct subclasses:

- **Exception**: Used for conditions that a program should handle.
- **Error**: Indicates serious problems that should not be handled by a program (e.g., `OutOfMemoryError`).

Hierarchy:

```
Throwable
|
├── Exception
|   ├── IOException
|   ├── SQLException
|   ├── ClassNotFoundException
|   ├── RuntimeException
|       ├── NullPointerException
|       ├── ArithmeticException
|       ├── IndexOutOfBoundsException
|       ├── IllegalArgumentException
|
└── Error
    ├── OutOfMemoryError
    ├── StackOverflowError
    ├── VirtualMachineError
```

3. Types of Exceptions

A. Checked Exceptions (Compile-time)

Checked exceptions are checked at compile-time and must be handled using `try-catch` or `throws`.

- **Examples**:
  - `IOException`
  - `SQLException`
  - `ClassNotFoundException`

```
import java.io.*;


public class CheckedExceptionExample {
   public static void main(String[] args) {
      try {
         FileReader file = new FileReader("nonexistent.txt"); // File not found
      } catch (FileNotFoundException e) {
         System.out.println("File not found: " + e.getMessage());
      }
   }
}
```

B. Unchecked Exceptions (Runtime)

Unchecked exceptions occur at runtime and are **not required** to be handled.

- **Examples**:
  - `NullPointerException`
  - `ArithmeticException`
  - `ArrayIndexOutOfBoundsException`

```
public class UncheckedExceptionExample {
   public static void main(String[] args) {
      int a = 10, b = 0;
```

```
        System.out.println(a / b); // ArithmeticException: Division by zero
    }
}
```

4. Exception Handling Mechanisms

A. Using try-catch

```
public class TryCatchExample {
    public static void main(String[] args) {
        try {
            int num = 10 / 0;  // Risky code
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        }
    }
}
```

B. Using finally Block

The `finally` block is always executed, whether an exception occurs or not.

```
public class FinallyExample {
    public static void main(String[] args) {
        try {
            int arr[] = {1, 2, 3};
            System.out.println(arr[5]); // ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Index out of bounds!");
        } finally {
            System.out.println("This block will always execute.");
        }
    }
```

```
}
```

C. Using throws Keyword

Used to declare exceptions in the method signature.

```
public class ThrowsExample {
    static void checkFile() throws FileNotFoundException {
        FileReader file = new FileReader("file.txt");
    }

    public static void main(String[] args) {
        try {
            checkFile();
        } catch (FileNotFoundException e) {
            System.out.println("File not found exception caught.");
        }
    }
}
```

D. Using throw Keyword

Used to explicitly throw an exception.

```
public class ThrowExample {
    static void validateAge(int age) {
        if (age < 18) {
            throw new IllegalArgumentException("Not eligible to vote");
        } else {
            System.out.println("Eligible to vote");
        }
    }

    public static void main(String[] args) {
```

```
        validateAge(16);
    }
}
```

5. Custom Exception Handling

Java allows users to define custom exceptions by extending the `Exception` class.

```
class MyException extends Exception {
    public MyException(String message) {
        super(message);
    }
}


public class CustomExceptionExample {
    static void checkNumber(int num) throws MyException {
        if (num < 0) {
            throw new MyException("Negative numbers are not allowed");
        }
    }

    public static void main(String[] args) {
        try {
            checkNumber(-5);
        } catch (MyException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

6. Best Practices for Exception Handling

☑ **Catch only required exceptions** – Don't use `catch(Exception e)` unless necessary.
☑ **Use specific exception types** – Helps in debugging.
☑ **Always close resources** – Use `try-with-resources` for closing resources automatically.

☑ **Never ignore exceptions** – Avoid empty `catch` blocks.
☑ **Use meaningful exception messages** – Helps in debugging.
☑ **Avoid unnecessary checked exceptions** – Use checked exceptions only when the caller can handle them.

```
try (FileReader file = new FileReader("data.txt")) {
   // File operations
} catch (IOException e) {
   System.out.println("Error reading file: " + e.getMessage());
}
```

Conclusion

- Java provides robust exception-handling mechanisms using `try-catch`, `finally`, `throw`, `throws`, and custom exceptions.
- Proper exception handling ensures better error management and a smooth user experience.
- Always follow best practices for writing maintainable and readable code.

Concurrency & Threading in Java

Concurrency in Java allows multiple tasks to run in parallel, improving performance and efficiency. Java provides built-in support for multi-threading, enabling developers to create and manage threads effectively.

---

1. What is a Thread?

A **thread** is a lightweight subprocess that executes a task. Each Java program has at least one thread, the **main thread**, which starts execution.

Thread Lifecycle

A thread goes through several states:

1. **New** – Created but not started.
2. **Runnable** – Ready to run but waiting for CPU time.
3. **Blocked** – Waiting to acquire a lock.
4. **Waiting** – Waiting indefinitely for another thread's signal.
5. **Timed Waiting** – Waiting for a specified time.
6. **Terminated** – Completed execution.

---

2. Creating Threads in Java

Java provides two ways to create threads:

A. Extending `Thread` Class

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running...");
    }

    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start(); // Starts the thread
    }
}
```

- **Drawback**: Since Java doesn't support multiple inheritance, extending `Thread` limits class flexibility.

B. Implementing `Runnable` Interface (Recommended)

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Runnable thread is running...");
    }

    public static void main(String[] args) {
        Thread t1 = new Thread(new MyRunnable());
        t1.start();
    }
}
```

- **Why preferred?** It allows extending another class while still running a thread.

---

3. Thread Methods

Method	Description
`start()`	Starts the thread
`run()`	Defines the task for the thread
`sleep(ms)`	Puts the thread to sleep
`join()`	Waits for a thread to finish execution
`isAlive()`	Checks if the thread is still running
`setPriority(int)`	Sets thread priority (1-10)

Example: Using `join()`

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " - " + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

        t1.start();
        t1.join(); // Waits for t1 to finish before starting t2
        t2.start();
    }
}
```

4. Thread Synchronization

When multiple threads access shared resources, **race conditions** and **inconsistencies** can occur. **Synchronization** ensures only one thread accesses a resource at a time.

A. Using `synchronized` Keyword

```
class SharedResource {
    synchronized void printNumbers(int n) { // Only one thread can access this method at a time
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " - " + (n * i));
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }
    }
}

class MyThread extends Thread {
    SharedResource obj;

    MyThread(SharedResource obj) {
        this.obj = obj;
    }

    public void run() {
        obj.printNumbers(5);
    }
}

public class SyncExample {
```

```
    public static void main(String[] args) {
        SharedResource obj = new SharedResource();
        MyThread t1 = new MyThread(obj);
        MyThread t2 = new MyThread(obj);

        t1.start();
        t2.start();
    }
}
```

- **Benefits**: Prevents race conditions and ensures consistency.
- **Drawback**: Can lead to reduced performance due to thread waiting.

B. Using `synchronized` Block

Instead of locking an entire method, a **synchronized block** locks only a specific section.

```
class SharedResource {
    void printNumbers(int n) {
        synchronized (this) { // Locks only this block
            for (int i = 1; i <= 5; i++) {
                System.out.println(Thread.currentThread().getName() + " - " + (n * i));
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    System.out.println(e.getMessage());
                }
            }
        }
    }
}
```

- **Benefit**: Increases performance by allowing partial locking.

---

5. Deadlock in Java

A **deadlock** occurs when two threads are waiting on each other to release a lock.

Example of Deadlock

```
class DeadlockExample {
    static final Object lock1 = new Object();
    static final Object lock2 = new Object();

    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            synchronized (lock1) {
                System.out.println("Thread 1: Holding lock1...");
                try { Thread.sleep(100); } catch (InterruptedException e) {}

                synchronized (lock2) {
                    System.out.println("Thread 1: Acquired lock2");
                }
            }
        });

        Thread t2 = new Thread(() -> {
            synchronized (lock2) {
                System.out.println("Thread 2: Holding lock2...");
                try { Thread.sleep(100); } catch (InterruptedException e) {}

                synchronized (lock1) {
                    System.out.println("Thread 2: Acquired lock1");
                }
            }
        });
```

```
        t1.start();
        t2.start();
    }
}
```

- **Solution**: Avoid nested locks, use **lock ordering**, or use `tryLock()`.

Generics & Collections in Java

Java **Generics** and **Collections Framework** provide powerful tools for managing data structures efficiently while ensuring **type safety**.

---

1. Generics in Java

Generics allow defining **classes, interfaces, and methods** with type parameters, ensuring **compile-time type safety**.

A. Why Use Generics?

☑ Avoids `ClassCastException` (No need for explicit typecasting).
☑ Ensures **type safety** (Errors detected at compile-time).
☑ Provides **code reusability**.

B. Creating a Generic Class

```
class Box<T> {
    private T value;

    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }

    public static void main(String[] args) {
```

```
        Box<Integer> intBox = new Box<>();
        intBox.setValue(100);
        System.out.println("Value: " + intBox.getValue());

        Box<String> strBox = new Box<>();
        strBox.setValue("Hello");
        System.out.println("Value: " + strBox.getValue());
    }
}
```

- **Here, `T` is a type parameter** that allows storing any data type.

C. Generic Methods

A method can use generics without making the entire class generic.

```
class Utility {
    public static <T> void printArray(T[] array) {
        for (T item : array) {
            System.out.print(item + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        Integer[] intArray = {1, 2, 3};
        String[] strArray = {"A", "B", "C"};

        printArray(intArray);
        printArray(strArray);
    }
}
```

D. Bounded Type Parameters

You can restrict the generic type to a specific hierarchy using `extends`.

```
class MathUtils<T extends Number> {
    private T num;

    public MathUtils(T num) {
        this.num = num;
    }

    public double square() {
        return num.doubleValue() * num.doubleValue();
    }

    public static void main(String[] args) {
        MathUtils<Integer> intObj = new MathUtils<>(5);
        MathUtils<Double> doubleObj = new MathUtils<>(4.5);

        System.out.println(intObj.square());
        System.out.println(doubleObj.square());
    }
}
```

- **Here, `T extends Number` ensures only numeric types are allowed.**

---

2. Java Collections Framework

The **Collections Framework** provides **dynamic data structures** like Lists, Sets, Maps, and Queues.

3. List Interface (Ordered, Allows Duplicates)

A **List** maintains insertion order and allows duplicates.

A. `ArrayList` (Dynamic Resizable Array)

☑ Fast random access (`O(1)`)
☑ Slower insertions/deletions (`O(n)`)

```
import java.util.*;


public class ArrayListExample {
   public static void main(String[] args) {
      List<String> names = new ArrayList<>();
      names.add("Alice");
      names.add("Bob");
      names.add("Charlie");


      System.out.println("Names: " + names);
   }
}
```

B. `LinkedList` (Doubly Linked List)

☑ Fast insertions/removals (`O(1)`)
☑ Slower random access (`O(n)`)

```
import java.util.*;


public class LinkedListExample {
   public static void main(String[] args) {
      List<Integer> numbers = new LinkedList<>();
      numbers.add(10);
      numbers.add(20);
      numbers.add(30);


      System.out.println("Numbers: " + numbers);
   }
}
```

4. Set Interface (No Duplicates, Unordered)

A **Set** does not allow duplicate elements.

A. `HashSet` (Unordered, Fast)

☑ Uses a **Hash Table**
☑ No guaranteed order
☑ `O(1)` for add, remove, contains

```
import java.util.*;


public class HashSetExample {
   public static void main(String[] args) {
      Set<String> colors = new HashSet<>();
      colors.add("Red");
      colors.add("Blue");
      colors.add("Green");
      colors.add("Red"); // Duplicate (ignored)


      System.out.println("Colors: " + colors);
   }
}
```

B. `TreeSet` (Sorted, No Duplicates)

☑ Uses **Red-Black Tree**
☑ Maintains **sorted order**
☑ `O(log n)` operations

```
import java.util.*;


public class TreeSetExample {
   public static void main(String[] args) {
      Set<Integer> numbers = new TreeSet<>();
      numbers.add(50);
      numbers.add(20);
```

```
        numbers.add(30);


        System.out.println("Sorted Set: " + numbers);
    }
}
```

5. Map Interface (Key-Value Pairs, No Duplicate Keys)

A **Map** stores key-value pairs, where keys must be unique.

A. `HashMap` (Unordered, Fast)

☑ **Key-value storage**
☑ **Fast lookups** (`O(1)`)

```
import java.util.*;


public class HashMapExample {
    public static void main(String[] args) {
        Map<Integer, String> students = new HashMap<>();
        students.put(1, "Alice");
        students.put(2, "Bob");
        students.put(3, "Charlie");


        System.out.println("Students: " + students);
    }
}
```

B. `TreeMap` (Sorted by Keys)

☑ Uses **Red-Black Tree**
☑ Maintains **sorted order of keys** (`O(log n)`)

```
import java.util.*;


public class TreeMapExample {
```

```
    public static void main(String[] args) {
        Map<String, Integer> scores = new TreeMap<>();
        scores.put("Alice", 90);
        scores.put("Bob", 85);
        scores.put("Charlie", 92);

        System.out.println("Sorted Scores: " + scores);
    }
}
```

6. Queue Interface (FIFO – First In First Out)

Queues **process elements in order**.

A. `PriorityQueue` (Sorted Queue)

☑ Maintains elements **in natural order**
☑ Uses a **Heap** (`O(log n)`)

```
import java.util.*;

public class PriorityQueueExample {
    public static void main(String[] args) {
        Queue<Integer> pq = new PriorityQueue<>();
        pq.add(30);
        pq.add(10);
        pq.add(20);

        while (!pq.isEmpty()) {
            System.out.println(pq.poll()); // Retrieves and removes elements in sorted order
        }
    }
}
```

7. Iterating Collections

A. Using `forEach` (Lambda)

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.forEach(System.out::println);
```

B. Using Iterator

```
Iterator<String> iterator = names.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

8. Comparable vs Comparator

A. `Comparable` (Natural Ordering)

```
class Student implements Comparable<Student> {
    int id;
    String name;

    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public int compareTo(Student other) {
        return Integer.compare(this.id, other.id);
    }
}
```

B. `Comparator` (Custom Ordering)

```
class NameComparator implements Comparator<Student> {
    public int compare(Student s1, Student s2) {
```

```
        return s1.name.compareTo(s2.name);
    }
}
```

Conclusion

- **Generics** provide type safety and reusability.
- **Lists** allow duplicates; **Sets** do not.
- **Maps** store key-value pairs.
- **Queues** follow FIFO.
- **Use proper iterators** for efficient iteration.