✅ Complete Core Java Roadmap (with Learning Flow & Interview Focus)

🔰 Foundation Topics

1. **Data Types, Variables & Keywords** ✅
2. **Operators**
3. **Control Flow (if, else, switch, loops)**
4. **Type Casting & Type Promotion**
5. **OOPs Concepts (Class, Object, Inheritance, Polymorphism, Abstraction, Encapsulation)**

🧱 Class Structure & Advanced Concepts

6. **Constructors (Default, Parameterized, Chaining)**
7. **Static & This Keyword**
8. **Final, Finally, Finalize**
9. **Wrapper Classes & Autoboxing/Unboxing**
10. **Access Modifiers (public, private, etc.)**
11. **Packages & Imports**
12. **Varargs**
13. **Enums**
14. **Annotations (Built-in & Custom)**

📝 String & Array Handling

15. **String, StringBuilder, StringBuffer**
16. **Arrays (1D, 2D, Jagged)**
17. **Immutable Class Design**

⚠️ Exception Handling

18. **Exception Handling (try-catch-finally, throw, throws)**
19. **Custom Exceptions**

📚 Collections & Generics

20. **Collections Framework (List, Set, Map, Queue)**
21. **Iterator, ListIterator**
22. **Generics**
23. **Comparable vs Comparator**

🚦 Threading & Concurrency

24. **Multithreading**
25. **Thread Lifecycle & Methods**
26. **Synchronization, wait/notify**
27. **Executor Framework & Runnable vs Callable**

⚙️ JVM & Performance

28. **JVM Architecture (JDK, JRE, JVM, JIT)**
29. **Memory Areas (Stack, Heap, Method Area, GC)**
30. **Class Loaders**

💾 File & Reflection

31. **File I/O (FileReader, BufferedReader, Scanner)**
32. **Serialization & Deserialization**
33. **Reflection API**
34. **Inner Classes (static, non-static, anonymous)**

🧠 Java 8+ Features (Important for Interviews)

35. **Lambda Expressions**
36. **Streams API**

37. **Functional Interfaces (Predicate, Consumer, Supplier)**
38. **Optional Class**
39. **Date and Time API (java.time)**

Once you're strong in these, you'll be ready for:

- ✅ Spring Boot
- ✅ Hibernate/JPA
- ✅ REST APIs
- ✅ Frontend + DevOps (Full Stack)

Now, let's begin with:

🟢 Topic 1: Data Types, Variables, and Keywords

🔹 1. What are Data Types?

✅ Definition:

Data types tell the compiler **what type of data** a variable can store. Java is **strongly typed**, so every variable must have a type.

📚 Two Categories of Data Types

A. Primitive Data Types (8 types)

Type	Size	Example	Use Case
byte	1 byte	byte b = 10;	Small numbers, memory-saving
short	2 bytes	short s = 20;	Better than byte
int	4 bytes	int i = 100;	Default for whole numbers
long	8 bytes	long l = 100L;	Large whole numbers

Type	Size	Example	Use Case
float	4 bytes	float f = 10.5f;	Decimal numbers
double	8 bytes	double d = 5.6;	Precise decimal values
char	2 bytes	char c = 'A';	Characters like A, B, C
boolean	1 bit	boolean b = true;	true/false values

B. Reference Data Types

- Arrays, Strings, Objects, Classes
- Store reference (address) of actual object in heap

🌍 **Real Life Example:**

java

CopyEdit

```
int age = 25;        // Primitive
String name = "Viraj"; // Reference
```

🔹 **2. What are Variables?**

✅ **Definition:**

A variable is a **container** used to store data. In Java, you must declare it with a data type.

📦 **Types of Variables**

Type	Where Defined	Memory Location	Default Value
Local	Inside method/block	Stack	No default
Instance	Inside class, non-static	Heap	Yes
Static	Inside class with static	Method Area	Yes

💉 **Example:**

java

CopyEdit

```
public class Car {
   int speed = 100; // Instance variable
   static String brand = "Honda"; // Static variable

   public void drive() {
      int gear = 4; // Local variable
      System.out.println("Gear: " + gear);
   }
}
```

🔹 **3. What are Keywords?**

✅ **Definition:**

Keywords are **reserved words** that have predefined meaning in Java. You **cannot use them** as identifiers (variable/class names).

Category	Examples
Data Types	int, char, boolean
Control Flow	if, else, switch, for
OOP	class, interface, extends, implements
Modifiers	public, private, static, final
Exception	try, catch, throw, finally
Misc	this, super, new, return, void

⚙️ **How It Works:**

- Variables are stored based on their type:
    - Local → Stack
    - Instance → Heap

    - Static → Method Area
- Java assigns **default values** only to instance and static variables:
    - int → 0, boolean → false, Object → null

✅ **Benefits:**

- Strongly typed: prevents accidental misuse
- Efficient memory management via variable types
- Clean structure for readability and debugging

❌ **Drawbacks:**

- Slightly more verbose than dynamic languages (like Python)
- Must declare type even for small usage

🎯 **Real World Use:**

- Bank app stores double balance = 5000.75;
- Weather app stores float temperature = 36.5f;
- Login system uses boolean isAuthenticated = true;

💬 **Interview Questions:**

1. What are the 8 primitive types in Java?
2. Difference between primitive and reference data types?
3. What is the default value of a boolean variable?
4. What is the difference between local and instance variable?
5. Can you use keywords as variable names?
6. What happens if you try to use a variable without initializing?

🟢 Topic 2: Operators in Java

🔹 1. What are Operators?

✅ **Definition:**
Operators in Java are **symbols** used to perform operations on variables and values.

✅ **Why we use them:**
To perform actions like addition, comparison, logic checks, increment/decrement, etc.

✅ **How it works:**
Java evaluates expressions based on **precedence** and **associativity** rules.

🔹 2. Types of Operators in Java

Category	Operators	Purpose
1. Arithmetic	+, -, *, /, %	Basic math operations
2. Relational (Comparison)	==, !=, >, <, >=, <=	Compare two values
3. Logical	&&, `	
4. Assignment	=, +=, -=, *=, /=, %=	Assign or update values
5. Unary	+, -, ++, --, !	Operate on a single operand
6. Bitwise	&, `	, ^, ~, <<, >>, >>>`
7. Ternary	? :	Conditional short if-else
8. instanceof	instanceof	Checks if object belongs to a class

✅ 3. Detailed Explanation with Examples

🔹 A. Arithmetic Operators

java

CopyEdit

int a = 10, b = 3;

```
System.out.println(a + b);  // 13
System.out.println(a - b);  // 7
System.out.println(a * b);  // 30
System.out.println(a / b);  // 3 (integer division)
System.out.println(a % b);  // 1 (remainder)
```

🔹 **B. Relational Operators**

java

CopyEdit

```
System.out.println(a > b);   // true
System.out.println(a == b);  // false
System.out.println(a != b);  // true
```

🔹 **C. Logical Operators**

java

CopyEdit

```
boolean x = true, y = false;
System.out.println(x && y);  // false
System.out.println(x || y);  // true
System.out.println(!x);     // false
```

✅ Real-life example: Login system

java

CopyEdit

```
if(username.equals("admin") && password.equals("123")) {
   // login successful
}
```

🔹 **D. Assignment Operators**

java

CopyEdit

```
int x = 10;
x += 5;  // x = x + 5 = 15
x *= 2;  // x = x * 2 = 30
```

🔹 **E. Unary Operators**

java

CopyEdit

```
int a = 5;
System.out.println(++a);  // 6 (pre-increment)
System.out.println(a--);  // 6 (post-decrement, then a becomes 5)
```

🔹 **F. Ternary Operator**

✅ Format:

java

CopyEdit

```
condition ? value_if_true : value_if_false;
```

✅ Example:

java

CopyEdit

```
int age = 20;
String result = (age >= 18) ? "Adult" : "Minor";
System.out.println(result);  // Adult
```

🔹 **G. Bitwise Operators**

✅ Used for **low-level operations** (e.g., performance-critical apps, device control)

java

CopyEdit

```
int a = 5;     // 0101

int b = 3;     // 0011


System.out.println(a & b);  // 0001 = 1 (AND)

System.out.println(a | b);  // 0111 = 7 (OR)

System.out.println(a ^ b);  // 0110 = 6 (XOR)

System.out.println(~a);     // Inverts bits
```

🔹 **H. instanceof Operator**

✅ Checks if an object is of a specific type

java

CopyEdit

```
String s = "hello";

System.out.println(s instanceof String);  // true
```

✅ **4. Benefits of Operators**

- Easy and quick computations
- Essential for conditional logic
- Widely used in algorithms, loops, and calculations

❌ **5. Drawbacks**

- Can lead to complex, unreadable expressions if overused
- Bitwise operations can be tricky and error-prone

🌎 Real-Life Example Use Cases

Use Case	Operators Involved
Login checks	==, &&
EMI calculation	+, *, /
Voting age check	>=, ? :
Game score updates	++, +=

💬 Interview Questions

1. Difference between == and .equals()?
2. What is the output of 10 + 20 + "abc" in Java?
3. What is the use of instanceof?
4. How is && different from &?
5. Write a program using a ternary operator.
6. What happens when dividing by zero?

✅ Topic 3: Control Flow Statements

🔹 What are Control Flow Statements?

Control flow statements in Java **determine the order in which instructions are executed** in a program. They are used to:

- Make decisions
- Repeat actions
- Control the flow based on conditions

🔹 Types of Control Flow Statements in Java

1. **Decision-Making Statements**
    - if
    - if-else
    - if-else-if
    - switch
2. **Looping Statements**
    - for
    - while
    - do-while
3. **Branching Statements**
    - break
    - continue
    - return

🧠 1. Decision-Making Statements

✅ if

java

CopyEdit

```
if (condition) {
   // code to execute if condition is true
}
```

✅ **Example:**

```
java
CopyEdit
int age = 18;
if (age >= 18) {
   System.out.println("Eligible to vote");
}
```

✅ **if-else**

```
java
CopyEdit
if (condition) {
   // true block
} else {
   // false block
}
```

✅ **if-else-if**

```
java
CopyEdit
if (condition1) {
   // block 1
} else if (condition2) {
   // block 2
} else {
```

```
    // default block
}
```

✅ **switch**

java

CopyEdit

```
switch(expression) {
    case value1:
        // code
        break;
    case value2:
        // code
        break;
    default:
        // default code
}
```

🧠 **2. Looping Statements**

✅ **for loop**

Used when the number of iterations is known.

java

CopyEdit

```
for (int i = 0; i < 5; i++) {
    System.out.println("Hello " + i);
}
```

✅ **while loop**

Used when the number of iterations is not known in advance.

java

CopyEdit

```
int i = 0;
while (i < 5) {
   System.out.println("Hello " + i);
   i++;
}
```

✅ **do-while loop**

Executes the code block at least once, then checks condition.

java

CopyEdit

```
int i = 0;
do {
   System.out.println("Hello " + i);
   i++;
} while (i < 5);
```

🧠 **3. Branching Statements**

✅ **break**

Used to **exit from a loop or switch**.

java

CopyEdit

```
for (int i = 1; i <= 5; i++) {
   if (i == 3) break;
   System.out.println(i);
}
```

✅ continue

Skips the current iteration and goes to the next one.

java

CopyEdit

```
for (int i = 1; i <= 5; i++) {
   if (i == 3) continue;
   System.out.println(i);
}
```

✅ return

Exits from the current method and optionally returns a value.

java

CopyEdit

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

🌟 Real Life Examples

Scenario	Java Statement Used
Login: valid credentials?	if-else
Choose user role (admin, user)	switch
Retry password max 3 times	for / while
Skip a banned username	continue
Exit on critical error	break

✅ Benefits

- Makes code logic easy to follow

- Essential for automation, iteration, and decisions
- Reduces code duplication with loops

⚠️ Drawbacks

- Overuse of nested if or switch makes code complex
- Infinite loops if conditions are not managed carefully

🎯 Common Interview Questions

1. Difference between while and do-while?
2. When to use switch over if-else?
3. What happens if we forget break in switch?
4. Can we use continue in a while loop?
5. How to stop an infinite loop?

✅ Topic 4: Arrays and Strings

These two are **fundamental data structures** used in Java. Mastering them is essential for both programming and interview success.

🔶 Part A: Arrays in Java

🔹 What is an Array?

An array is a **collection of elements of the same type**, stored in **contiguous memory locations**.

🧠 Think of it like a row of lockers — each holds a value, and you can access them by index (starting at 0).

🔹 Declaration and Initialization

java

CopyEdit

int[] marks = new int[5]; // Declaration

marks[0] = 90;

int[] scores = {85, 92, 76}; // Initialization

🔹 Types of Arrays

1. **One-Dimensional Array**

java

CopyEdit

int[] numbers = {1, 2, 3};

2. **Multi-Dimensional Array (Matrix)**

java

CopyEdit

int[][] matrix = {

```
  {1, 2},
  {3, 4}
};
```

🔹 **Accessing Elements**

java

CopyEdit

```
System.out.println(scores[0]); // First element
System.out.println(matrix[1][1]); // 4
```

🔹 **Looping Through Array**

java

CopyEdit

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

✅ Enhanced for loop:

java

CopyEdit

```
for (int score : scores) {
  System.out.println(score);
}
```

🔹 **Real-World Examples**

- Store student marks
- Weekly temperature logs
- Game high scores

✅ Benefits of Arrays

- Fixed-size fast access (O(1) by index)
- Memory efficient

⚠️ Drawbacks

- Fixed size — can't grow dynamically
- Must know size in advance

🧠 Common Interview Questions (Arrays)

1. How are arrays stored in memory?
2. Difference between int[] a and int a[]?
3. Can arrays store different data types?
4. What is ArrayIndexOutOfBoundsException?
5. How to reverse an array?

🔶 Part B: Strings in Java

🔹 What is a String?

A String is a **sequence of characters**, treated as an object in Java.

java

CopyEdit

String name = "Viraj";

🔹 String Creation

java

CopyEdit

String s1 = "Java"; // String literal (in String Pool)

String s2 = new String("Java"); // New object in heap

🔹 Important String Methods

java

CopyEdit

```
s.length();         // Length of string
s.charAt(2);        // Character at index
s.equals("Java");   // Content comparison
s.equalsIgnoreCase("java");
s.toUpperCase();    // "JAVA"
s.toLowerCase();    // "java"
s.contains("av");   // true
s.indexOf('v');     // 2
s.substring(1, 3);  // "av"
```

🔹 String Concatenation

java

CopyEdit

```
String fullName = "Viraj" + " Patel";
```

🔹 Immutability

Strings are **immutable** — once created, they can't be changed.

java

CopyEdit

```
String s = "Java";
s.concat("Script"); // Does not change s
System.out.println(s); // Still "Java"
```

✅ Use StringBuilder/StringBuffer for mutable strings.

✅ Benefits of Strings

- Easy to manipulate and use
- Immutable = thread-safe
- Memory efficient via **String Pool**

⚠️ Drawbacks

- Immutable = creates new object on every change
- Performance issues in loops with many string modifications

🔄 StringBuilder vs StringBuffer vs String

Feature	String	StringBuilder	StringBuffer
Mutable	❌ No	✅ Yes	✅ Yes
Thread-Safe	✅ Yes	❌ No	✅ Yes
Performance	Medium	✅ High	Low (sync)

🔥 Real-Life Examples of Strings

- Usernames, passwords
- Messages, logs
- Search inputs

🧠 Common Interview Questions (Strings)

1. Difference between String, StringBuilder, and StringBuffer?
2. Why are strings immutable in Java?
3. How is string stored in memory?
4. What is the String Pool?
5. Write a program to reverse a string.
6. Compare strings using == vs .equals()?

✅ Topic 5: Object-Oriented Programming (OOPs) in Java

This is the **heart of Java**. Mastering OOP is *essential* for real-world application development, backend systems, and cracking Java interviews.

🔹 What is OOP (Object-Oriented Programming)?

OOP is a programming paradigm based on **objects and classes**. It allows you to structure code using **real-world entities** such as Car, BankAccount, Employee, etc.

Java is **100% Object-Oriented** (except for primitive types).

🔹 4 Pillars of OOP in Java

1. **Encapsulation**
2. **Inheritance**
3. **Polymorphism**
4. **Abstraction**

We'll explore each in detail below 👇

✅ 1. Encapsulation

Definition: Wrapping data (variables) and methods (functions) together into a single unit (class), and restricting direct access.

java

CopyEdit

```
public class Person {
  private String name;

  public void setName(String name) {
    this.name = name; // Setter
  }
```

```
    public String getName() {
        return name; // Getter
    }
}
```

🔒 private hides the data. public methods provide controlled access.

✅ **Benefits:**

- Data hiding
- Better control of code
- Easy to debug and maintain

✅ **2. Inheritance**

Definition: When a class (child) **inherits** properties and methods from another class (parent).

java

CopyEdit

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

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

✅ **Types:**

- Single
- Multilevel
- Hierarchical
  (*Note: Java does NOT support multiple inheritance with classes*)

✅ **Benefits:**

- Code reusability
- Easy extension

🛑 **Drawbacks:**

- Tightly coupled classes
- Can lead to complex hierarchies

✅ **3. Polymorphism**

Definition: One action behaves differently in different situations.

➤ **Compile-Time Polymorphism (Method Overloading)**

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

➤ **Runtime Polymorphism (Method Overriding)**

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


class Cat extends Animal {
```

```
  void sound() { System.out.println("Meow"); } // overridden
}
```

✅ **Benefits:**

- Flexibility
- Reduces code duplication

✅ **4. Abstraction**

Definition: Hiding internal details and showing only essential features.

```
java
CopyEdit
abstract class Vehicle {
  abstract void start();
  void stop() {
    System.out.println("Vehicle stopped");
  }
}

class Car extends Vehicle {
  void start() {
    System.out.println("Car started");
  }
}
```

✅ Can be achieved using:

- abstract classes (0–100% abstraction)
- interfaces (100% abstraction)

🔥 **Real Life Examples of OOP**

- **Encapsulation**: ATM – only access balance using PIN

- **Inheritance**: Bird extends Animal
- **Polymorphism**: draw() method for circle, square, triangle
- **Abstraction**: Driving a car – you don't know how engine works

---

✅ Benefits of OOP in Java

- Models real-world easily
- Code reusability and modularity
- Easy to debug and scale

⚠️ Drawbacks

- More complex to design initially
- Over-engineering small problems

🎯 Common Interview Questions

1. What are the 4 pillars of OOP?
2. Difference between abstract class and interface?
3. What is method overloading vs overriding?
4. Why Java doesn't support multiple inheritance?
5. What is encapsulation in real life?
6. Can we create object of abstract class?

✅ Topic 6: Constructors in Java

Constructors are a core part of Java's object-oriented system. Every time you create an object, **a constructor is called** — so understanding this is **essential**.

🔹 What is a Constructor?

A **constructor** is a special method that is automatically called **when an object is created**.

java

CopyEdit

ClassName obj = new ClassName(); // Constructor is called here

---

🔹 Why do we use Constructors?

- To **initialize objects** (set default or custom values)
- To avoid writing initialization code again and again

🔹 Key Rules

- Name of the constructor **must match the class name**
- No return type (not even void)
- Called automatically when the object is created

🔹 Types of Constructors

1. ✅ Default Constructor

Created by Java automatically if no constructor is defined.

java

CopyEdit

```
class Student {
  Student() {
    System.out.println("Default Constructor Called");
```

```
    }
}
```

If you define any constructor, Java **won't** create a default one.

---

2. ✅ Parameterized Constructor

You create this to set custom values during object creation.

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

    Student(String n, int a) {
        name = n;
        age = a;
    }
}
java
CopyEdit
Student s = new Student("Viraj", 22);
```

---

3. ✅ No-Arg Constructor

Same as default, but **you define it manually**.

```
java
CopyEdit
Student() {
    System.out.println("This is a no-arg constructor");
}
```

🔁 Constructor Overloading

You can define **multiple constructors** with different parameters.

java

CopyEdit

```
class Book {
  Book() {
    System.out.println("No-arg");
  }

  Book(String title) {
    System.out.println("Book title is " + title);
  }
}
```

🔁 Difference: Constructor vs Method

Feature	Constructor	Method
Name	Same as class	Any name
Return Type	No return type	Must have return type
Called When	Object is created	Called explicitly
Purpose	Initializes object	Defines behavior

🔐 Private Constructor

Used to **prevent object creation** from outside — often used in **Singleton Design Pattern**.

java

CopyEdit

```
class Test {
   private Test() {
      // Can't create object from outside
   }
}
```

🔥 **Real-Life Example**

ATM System:

java

CopyEdit

ATMUser u = new ATMUser("Viraj", "1234");

Here the constructor sets up the name and PIN during account creation.

✅ **Benefits of Using Constructors**

- Initialize objects with default or custom values
- Simplifies object setup
- Makes code cleaner and more readable

⚠️ **Drawbacks**

- If misused, can create inconsistent objects
- Overloaded constructors can become confusing

🎯 **Common Interview Questions**

1. What is a constructor?
2. Can constructor be private?
3. What happens if no constructor is defined?
4. Difference between constructor and method?
5. What is constructor overloading?

6. Can constructor be static or final?

❌ No, constructors cannot be static, final, or abstract.

✅ Topic 7: static Keyword in Java

🔹 What is static in Java?

The static keyword in Java is used for **memory management**. It means the member (variable or method) belongs to the **class**, not the instance.

This means:

- It gets memory **only once** (class-level)
- All objects of the class **share the same copy**

🔹 Where Can You Use static?

You can apply static to:

1. **Variables**
2. **Methods**
3. **Blocks**
4. **Nested classes**

🔸 1. Static Variable

A **single copy** is shared across all objects.

java

CopyEdit

```
class Student {
   static String college = "IIT"; // shared by all
   String name;

   Student(String name) {
      this.name = name;
   }
}
```

✅ **Usage:**

java

CopyEdit

Student s1 = new Student("Viraj");

Student s2 = new Student("Jay");

System.out.println(Student.college); // IIT

🔸 **2. Static Method**

Can be called **without creating an object**.
It **cannot use non-static variables** directly.

java

CopyEdit

class MathUtils {

  static int square(int x) {

    return x * x;

  }

}

✅ **Usage:**

java

CopyEdit

int result = MathUtils.square(5); // 25

🔸 **3. Static Block**

Runs **once** when the class is loaded.
Used for **initializing static data**.

java

CopyEdit

class InitDemo {

```
  static int a;
  static {
    a = 100;
    System.out.println("Static block executed");
  }
}
```

🔸 **4. Static Class (Nested)**

Only **nested (inner) classes** can be static.

java

CopyEdit

```
class Outer {
  static class Inner {
    void show() {
      System.out.println("Inside static nested class");
    }
  }
}
```

✅ **Usage:**

java

CopyEdit

```
Outer.Inner obj = new Outer.Inner();
obj.show();
```

✅ **Benefits of Static Keyword**

- Memory-efficient (loaded only once)
- Easy access without object creation
- Used in utility/helper classes (Math, Collections, etc.)

⚠️ Limitations/Drawbacks

- Cannot access instance variables/methods directly
- Makes code less flexible (tight coupling)
- Overuse leads to poor OOP design

🎯 Real-Life Example

- Math.sqrt(), System.out.println() — both are static methods
- college field shared by all students
- main() method is static because JVM calls it without object

---

🧠 Interview Questions on static

1. Why is the main method static in Java?
2. Can a static method access instance variables?
3. What is the use of a static block?
4. Can you override static methods?

❌ No, static methods are not polymorphic.

5. Difference between static and non-static members?
6. Can constructors be static?

❌ No, because constructors are used to create objects.

✅ Topic 8: this and super Keywords in Java

These two keywords help in **object context referencing** and **inheritance handling**. They are very commonly asked in interviews and used in real-world code.

🔶 1. this Keyword

🔹 What is this?

this is a reference variable in Java that **refers to the current object** (the object on which the method is being called).

🔹 Use Cases of this

✅ a. To refer to current class instance variables

java

CopyEdit

```
class Student {
   String name;

   Student(String name) {
      this.name = name; // 'this' refers to current object
   }
}
```

✅ b. To call current class methods

java

CopyEdit

```
void show() {
   System.out.println("Hello");
}
```

```
void display() {
   this.show(); // calls show() method
}
```

✅ **c. To call current class constructor**

Using this() constructor call:

java

CopyEdit

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

   Car(String model) {
      this(); // calls default constructor
      System.out.println("Model: " + model);
   }
}
```

✅ **d. To pass current object as a parameter**

java

CopyEdit

```
void print(Student obj) {
   System.out.println(obj.name);
}

void get() {
   print(this); // passes current object
}
```

🎯 Real-Life Example

java

CopyEdit

Student s = new Student("Viraj");

Here, this.name = name helps assign "Viraj" to the current object's variable.

🔶 2. super Keyword

🔹 What is super?

super is a reference variable used to refer to the **immediate parent class object**.

🔹 Use Cases of super

✅ a. To call parent class constructor

java

CopyEdit

```
class Parent {
   Parent() {
      System.out.println("Parent constructor");
   }
}

class Child extends Parent {
   Child() {
      super(); // calls parent constructor
      System.out.println("Child constructor");
   }
}
```

✅ **b. To access parent class method**

java

CopyEdit

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

class Dog extends Animal {
   void sound() {
      super.sound(); // calls Animal's sound()
      System.out.println("Dog barks");
   }
}
```

✅ **c. To access parent class variables**

java

CopyEdit

```
class Parent {
   int x = 10;
}

class Child extends Parent {
   int x = 20;

   void show() {
      System.out.println(super.x); // 10
   }
```

}

✅ **Benefits of this and super**

Keyword	Purpose
this	Access current class variables/methods/constructors
super	Access parent class variables/methods/constructors

⚠️ **Drawbacks**

- Overuse can reduce code clarity
- Only used within inheritance or constructors
- Can be confusing for beginners if misused

🧠 **Interview Questions**

1. What is the difference between this and super?
2. Can we use this() and super() together?

❌ No, only one constructor call is allowed, and it must be the first statement.

3. Why do we use super() in constructors?
4. Can we use super in static context?

❌ No, because super is an instance keyword.

5. Can this and super be used in static methods?

❌ No, because they relate to objects, not classes.

✅ Topic 9: Inheritance in Java (Detailed)

🔹 What is Inheritance?

Inheritance allows a class to acquire (inherit) the **properties and behaviors** (fields and methods) of another class.

It supports the **"is-a"** relationship.

Example: Dog **is-a** Animal.

🔹 Why use Inheritance?

- To achieve **code reusability**
- To build **hierarchical** structures
- To allow **method overriding** (polymorphism)

🔹 Syntax of Inheritance

java

CopyEdit

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

class Child extends Parent {
  void display() {
    System.out.println("Child method");
  }
}
```

✅ Usage:

java

CopyEdit

```
Child c = new Child();
c.show();   // Inherited
c.display(); // Child's own
```

🔸 **Types of Inheritance in Java**

Type	Description
Single Inheritance	One class inherits from another
Multilevel Inheritance	A class inherits from a class which inherits another
Hierarchical Inheritance	Multiple classes inherit from the same parent
Multiple Inheritance (via Interfaces)	A class implements multiple interfaces

❌ **Java doesn't support multiple inheritance with classes** to avoid ambiguity (Diamond Problem).

✅ **1. Single Inheritance**

java

CopyEdit

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

class B extends A {
   void greet() { System.out.println("Hi from B"); }
}
```

✅ **2. Multilevel Inheritance**

java

CopyEdit

```
class A {
   void aMethod() {}
}

class B extends A {
   void bMethod() {}
}

class C extends B {
   void cMethod() {}
}
```

✅ **3. Hierarchical Inheritance**

java

CopyEdit

```
class Animal {
   void eat() { System.out.println("Eating..."); }
}

class Dog extends Animal {
   void bark() { System.out.println("Barking..."); }
}

class Cat extends Animal {
   void meow() { System.out.println("Meowing..."); }
}
```

✅ 4. Multiple Inheritance (via Interface)

java

CopyEdit

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

interface B {
   void display();
}

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

🔹 Method Overriding in Inheritance

Allows child class to provide a specific implementation of a method from parent class.

java

CopyEdit

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

class Child extends Parent {
```

```
 void show() {
     System.out.println("Child Show");
   }
}
```

✅ Access using super.show() if needed.

🔥 **Real-Life Example of Inheritance**

```
java
CopyEdit
class Vehicle {
   void start() {}
}

class Car extends Vehicle {
   void playMusic() {}
}
```

Here, **Car inherits Vehicle** → can start() and playMusic().

✅ **Benefits of Inheritance**

- Reuse of common logic
- Better code organization
- Enables polymorphism
- Reduces redundancy

⚠️ **Drawbacks**

- Tight coupling of parent-child
- Breaks encapsulation (access to internal logic)
- Overuse leads to fragile code structure

- Difficult to modify parent class without affecting child classes

---

🧠 Interview Questions

1. What is inheritance?
2. Types of inheritance supported in Java?
3. Why Java doesn't support multiple inheritance with classes?
4. Difference between extends and implements?
5. What is constructor chaining in inheritance?
6. How can you access parent methods from child class?

✅ Topic 10: Polymorphism in Java (with Real Examples)

🔹 What is Polymorphism?

Polymorphism means **"many forms"**. It allows an object to **take many forms** or behave differently in different contexts.

🔹 Why Use Polymorphism?

- For **code flexibility** and **extensibility**
- To write **generic** and **reusable** code
- Enables **runtime decision-making** on method execution

🔸 Types of Polymorphism in Java

Type	Description
Compile-time (Static)	Method Overloading
Runtime (Dynamic)	Method Overriding via inheritance

✅ 1. Compile-Time Polymorphism (Method Overloading)

Same method name, different parameters.

java

CopyEdit

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

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

```
    }

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

✅ At compile time, Java knows **which method to call** based on arguments.

✅ **2. Runtime Polymorphism (Method Overriding)**

Same method signature, but redefined in **child class**.

java

CopyEdit

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

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

class Cat extends Animal {
    void sound() {
        System.out.println("Cat meows");
    }
```

}

✅ **Dynamic Dispatch**:

java

CopyEdit

Animal a;

a = new Dog(); // prints Dog barks

a.sound();

➡️ JVM decides at **runtime** which method to execute.

🔁 **Difference: Overloading vs Overriding**

Feature	**Overloading**	**Overriding**
Type	Compile-time	Runtime
Class	Same class	Parent-child (Inheritance)
Signature	Must differ	Must be same
Static methods?	Can overload	Cannot override

🔐 **Polymorphism with Interfaces**

java

CopyEdit

interface Shape {

  void draw();

}

class Circle implements Shape {

  public void draw() { System.out.println("Drawing circle"); }

}

```
class Square implements Shape {
    public void draw() { System.out.println("Drawing square"); }
}
```

✅ Usage:

```
java
CopyEdit
Shape s = new Circle();  // polymorphism in action
s.draw();              // Output: Drawing circle
```

🔥 **Real-Life Example**

```
java
CopyEdit
class Payment {
    void pay() {
        System.out.println("Generic payment");
    }
}

class CreditCard extends Payment {
    void pay() {
        System.out.println("Paid using credit card");
    }
}

class UPI extends Payment {
    void pay() {
        System.out.println("Paid using UPI");
    }
```

}

✅ At runtime:

java

CopyEdit

Payment p = new UPI();

p.pay(); // Output: Paid using UPI

✅ **Benefits of Polymorphism**

- Reduces code duplication
- Promotes loose coupling
- Easy to scale and maintain
- Supports dynamic behavior

⚠️ **Drawbacks**

- May lead to unexpected results if misused
- Harder to trace actual method being called
- Slightly slower due to runtime binding

🧠 **Interview Questions**

1. What is polymorphism?
2. Difference between method overloading and overriding?
3. Can you override static or private methods?

❌ No, static methods are class-level.

4. What is dynamic method dispatch?
5. Why is polymorphism important in OOP?
6. How does Java implement polymorphism?

✅ Topic 11: Abstraction in Java

🔹 What is Abstraction?

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

🧠 Think: "What it does, not how it does it."

🔹 Why Use Abstraction?

- To reduce complexity
- To improve security
- To separate **what** from **how**

🔹 Real-Life Example

- **Phone**: You dial a number (you know *what* happens), but you don't know the internal circuitry (*how* it happens).
- **Car**: You drive with pedals and steering, not worrying about the engine mechanism.

✅ How to Achieve Abstraction in Java?

Java provides two ways:

1. **Abstract Class**
2. **Interface**

🔸 1. Abstract Class

- Declared using abstract keyword.
- Can have both **abstract (incomplete)** and **concrete (complete)** methods.
- Cannot be instantiated.

java

CopyEdit

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

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

➕ **Child Class Must Implement Abstract Method**

java

CopyEdit

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

✅ Usage:

java

CopyEdit

```
Animal a = new Dog();
a.makeSound(); // Dog barks
a.eat();      // This animal eats food.
```

🔸 **2. Interface (100% Abstraction)**

- Introduced to provide full abstraction.
- Methods are implicitly public abstract.
- Java 8+ allows **default** and **static** methods.

java

CopyEdit

```
interface Shape {
   void draw(); // abstract method
}
```

➕ **Implementation**

java

CopyEdit

```
class Circle implements Shape {
   public void draw() {
      System.out.println("Drawing circle");
   }
}
```

✅ Usage:

java

CopyEdit

```
Shape s = new Circle();
s.draw(); // Drawing circle
```

🆚 **Abstract Class vs Interface**

Feature	Abstract Class	Interface
Inheritance	Single (with classes)	Multiple (via interfaces)
Methods	Can be abstract + normal	Only abstract (until Java 8+)
Constructor	Yes	❌ No
Access Modifiers	public, protected, etc	Only public
Fields	Can have variables	Only constants (public static final)

✅ **Benefits of Abstraction**

- Hides implementation logic
- Increases security
- Reduces complexity
- Promotes code modularity
- Improves maintainability

⚠️ Drawbacks

- Slightly increases design complexity
- Requires more upfront planning

🧠 Interview Questions

1. What is abstraction?
2. Difference between abstraction and encapsulation?
3. Difference between abstract class and interface?
4. Can we create an object of an abstract class?
5. Can abstract class have constructor?
6. Why do we need interfaces when we have abstract classes?
7. What is default method in interface?

✅ Topic 12: Encapsulation in Java

🔹 What is Encapsulation?

Encapsulation is the process of **wrapping data (variables)** and **code (methods)** together into a **single unit** (class).
It is also known as **data hiding**.

📦 Like putting your code and data in a protective box (class).

🔹 Why Use Encapsulation?

- To protect internal object state
- To achieve **data security**
- To implement **controlled access**

🔹 Real-Life Example

ATM Machine
You press buttons to withdraw money (interface), but you don't see the inner logic of the banking system. The **details are hidden** — only safe access is allowed.

✅ How to Achieve Encapsulation in Java?

1. Make variables **private**
2. Use **public getter and setter methods** to access/update them

---

👷 Example:

java

CopyEdit

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

```
    // Getter for name
    public String getName() {
        return name;
    }

    // Setter for name
    public void setName(String name) {
        this.name = name;
    }

    // Getter for age
    public int getAge() {
        return age;
    }

    // Setter for age
    public void setAge(int age) {
        if (age > 0) {
            this.age = age;
        }
    }
}
```

✅ Usage:

```
java
CopyEdit
Student s = new Student();
s.setName("Viraj");
s.setAge(22);
```

System.out.println(s.getName()); // Viraj

🎯 Key Rules

Element	Access Modifier
Data members (fields)	private
Methods to access data	public

🔄 Difference: Abstraction vs Encapsulation

Feature	Abstraction	Encapsulation
Focus	Hiding implementation complexity	Hiding data
Achieved via	Abstract class / Interface	Class with private fields
Purpose	Show only functionality	Protect internal state

✅ Benefits of Encapsulation

- Data security (hide sensitive data)
- Improves code maintainability
- Easy to add validation logic
- Helps in **loose coupling**

⚠️ Drawbacks

- Might require writing more code (getters/setters)
- Poorly designed accessors can break encapsulation

🧠 Interview Questions

1. What is encapsulation?
2. How is encapsulation implemented in Java?
3. Difference between abstraction and encapsulation?

4. Can you achieve encapsulation without getters/setters?
5. Why should fields be private?

✅ Topic 13: Access Modifiers in Java

🔹 What are Access Modifiers?

Access modifiers are **keywords in Java** used to **control the visibility** (scope) of classes, variables, methods, and constructors.

🔹 Why Use Access Modifiers?

- To **protect data** from unauthorized access
- To **implement encapsulation**
- To define **how much of your class is exposed** to other classes

---

✅ Types of Access Modifiers in Java

Java provides **4 access levels**:

Modifier	Same Class	Same Package	Subclass	Other Packages
private	✅	❌	❌	❌
(default)	✅	✅	❌	❌
protected	✅	✅	✅	❌
public	✅	✅	✅	✅

🔸 1. private

- Accessible **only within the same class**
- Not visible to subclasses or other classes

java

CopyEdit

```
class Person {
    private String name;
```

```
    private void greet() {
        System.out.println("Hello");
    }
}
```

✅ Use case: Sensitive data, internal logic

🔸 **2. Default (No keyword)**

- Accessible within the **same package only**
- Not visible outside the package

```
java
CopyEdit
class Employee {  // default access
    void work() {
        System.out.println("Working...");
    }
}
```

✅ Use case: Package-level modularization

🔸 **3. protected**

- Accessible in:
    - Same package
    - Subclasses (even in different packages)

```
java
CopyEdit
class Vehicle {
    protected void start() {
        System.out.println("Vehicle starting...");
    }
```

}

✅ Use case: Inheritance with controlled access

🔸 **4. public**

- Accessible **from everywhere**

java

CopyEdit

```
public class Main {
   public static void main(String[] args) {
      System.out.println("Public class and method");
   }
}
```

✅ Use case: APIs, services, utility classes

🔐 **Example Summary:**

java

CopyEdit

```
public class Demo {
   private int a = 10;      // private: class-only
   int b = 20;              // default: package
   protected int c = 30;    // protected: package + subclass
   public int d = 40;       // public: everywhere
}
```

✅ **Benefits of Access Modifiers**

- Promotes **encapsulation**
- Protects **data integrity**

- Helps in creating **clean APIs**
- Enables **layered access** to components

⚠️ **Drawbacks**

- Too much restriction can lead to overengineering
- Misuse (e.g. making everything public) can break encapsulation

---

🧠 **Interview Questions**

1. What are access modifiers in Java?
2. Difference between private, protected, and default?
3. Can a top-level class be private?

❌ No. Only public or default.

4. Can protected members be accessed outside package?

✅ Only in subclasses.

5. What's the default access modifier in Java?

✅ Topic 14: static Keyword in Java

🔹 What is static in Java?

The static keyword is used to **indicate that a member (method or variable) belongs to the class itself**, not to instances (objects) of the class.

🔹 Why Use static?

- To **share** common data/methods among all objects
- To avoid creating multiple copies of the same data
- To access members **without creating objects**

🔸 Where Can You Use static?

Used With	Purpose
Static Variable	Shared by all objects (class-level variable)
Static Method	Belongs to the class, can be called without object
Static Block	Runs once when class is loaded
Static Class	Only for **nested classes**, not top-level classes

---

✅ 1. Static Variable

java

CopyEdit

```
class Student {
   int id;
   String name;
   static String college = "ABC College";  // shared by all objects

   Student(int id, String name) {
```

```
        this.id = id;
        this.name = name;
    }

    void display() {
        System.out.println(id + " " + name + " " + college);
    }
}
```

✅ Usage:

```
java
CopyEdit
Student s1 = new Student(1, "Viraj");
Student s2 = new Student(2, "Riya");
s1.display();  // same college printed
s2.display();
```

✅ **2. Static Method**

```
java
CopyEdit
class MathUtil {
    static int square(int x) {
        return x * x;
    }
}
```

✅ Usage:

```
java
CopyEdit
int result = MathUtil.square(5); // No need to create object
```

⚠️ A static method **cannot access non-static members** directly.

✅ **3. Static Block**

- Used for **static initialization logic**
- Executes only once when class is loaded

java

CopyEdit

```
class Test {
  static {
    System.out.println("Static block runs first");
  }

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

🔄 Output:

sql

CopyEdit

```
Static block runs first
Main method runs
```

✅ **4. Static Nested Class**

java

CopyEdit

```
class Outer {
  static class Inner {
    void show() {
```

```
        System.out.println("Inside static nested class");
    }
  }
}
```

✅ Usage:

```
java
CopyEdit
Outer.Inner obj = new Outer.Inner();
obj.show();
```

🧠 **Interview Questions**

1. What is the static keyword used for?
2. Can we override static methods?

❌ No, static methods belong to class.

3. Can static methods access instance variables?

❌ No.

4. When is a static block executed?
5. What is a static nested class?

✅ **Benefits of static**

- Saves memory (only one copy exists)
- Access without creating object
- Useful for utility/helper methods

⚠️ **Drawbacks**

- Less flexible (no access to instance members)
- Can cause design issues if overused

- Harder to test and mock in unit tests

✅ Topic 15: final Keyword in Java

🔹 What is final in Java?

The final keyword is used to **restrict** the user:

- A **final variable** → **can't be changed** (constant)
- A **final method** → **can't be overridden**
- A **final class** → **can't be inherited**

🔒 Think of it like "lock" – once set, it can't be modified.

✅ 1. Final Variable

java

CopyEdit

final int age = 25;

age = 30; // ❌ Error: cannot assign a value to final variable

- Must be initialized **once**
- Often used for **constants**
- If not initialized at declaration, it must be initialized in the **constructor**

java

CopyEdit

```
class Student {
   final int roll;

   Student(int roll) {
      this.roll = roll; // OK
   }
}
```

✅ 2. Final Method

java

CopyEdit

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

class Dog extends Animal {
   // void sound() ❌ Not allowed – can't override final method
}
```

- Useful when you want to **prevent subclasses** from changing the behavior of a method

---

✅ 3. Final Class

java

CopyEdit

```
final class Vehicle {
   void run() {
      System.out.println("Vehicle is running");
   }
}

// class Car extends Vehicle ❌ Not allowed
```

- Can’t be inherited
- Common example: java.lang.String is a final class

🔄 Summary Table

Final On	Restriction Imposed
Variable	Can't be reassigned (constant)
Method	Can't be overridden in subclass
Class	Can't be inherited/extended

✅ Benefits of final

- Ensures **immutability** (for variables/objects)
- Improves **security and design** (avoid accidental overrides)
- Helps with **thread safety** (constant values)
- Compiler optimizations

⚠️ Drawbacks

- Reduces flexibility in inheritance and customization
- Once set, values can't be changed (not suitable for all situations)

🧠 Interview Questions

1. What is the use of the final keyword?
2. Can we reassign a final variable?
3. Can a constructor be final?

❌ No, constructors are never inherited or overridden.

4. Can we override a final method?
5. Why is String class final in Java?

✅ Topic 16: this and super Keywords in Java

🔷 Part 1: this Keyword

🔹 What is this?

this refers to the **current object** of the class.

✅ Uses of this:

Use Case	Purpose
1. Refer current class instance	Resolve confusion between instance variable and parameter
2. Invoke current class constructor	Constructor chaining using this()
3. Pass current object as argument	For method or constructor calls
4. Return current class instance	For method chaining

🔸 Example 1: Resolve variable shadowing

java

CopyEdit

```
class Student {
   int id;

   Student(int id) {
      this.id = id;  // this.id = instance variable, id = parameter
   }
}
```

🔸 **Example 2: Constructor chaining**

java

CopyEdit

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

   Student() {
      this("Default", 0);  // calling another constructor
   }

   Student(String name, int age) {
      this.name = name;
      this.age = age;
   }
}
```

🔸 **Example 3: Returning this**

java

CopyEdit

```
class A {
   A get() {
      return this;
   }
}
```

🔷 Part 2: super Keyword

🔹 What is super?

super refers to the **parent (superclass) object**.

✅ Uses of super:

Use Case	Purpose
1. Access superclass variables	When child class hides parent class variables
2. Call superclass methods	To call overridden method of the parent class
3. Call superclass constructor	Must be the first statement in child constructor

---

🔸 Example 1: Access parent class method

java

CopyEdit

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

class Dog extends Animal {
   void sound() {
      super.sound();  // calls Animal's sound()
      System.out.println("Dog barks");
   }
}
```

🔸 **Example 2: Call parent class constructor**

java

CopyEdit

```
class Animal {
   Animal(String type) {
      System.out.println("Animal type: " + type);
   }
}

class Dog extends Animal {
   Dog() {
      super("Mammal");  // must be first line in constructor
      System.out.println("Dog created");
   }
}
```

🔄 **Summary Table**

Keyword	Refers to	Used for
this	Current class object	Access fields/methods/constructors of same class
super	Parent class object	Access parent fields/methods/constructors

---

✅ **Benefits**

- Helps avoid naming conflicts
- Allows calling specific constructors or methods
- Supports clean inheritance and method overriding

⚠️ **Drawbacks**

- Can be confusing if misused
- Overuse may indicate poor design (e.g. deep inheritance)

🧠 **Interview Questions**

1. What is the use of this keyword?
2. Can this() and super() be used together?

❌ No. Only one constructor call is allowed, and must be the first line.

3. Difference between this and super?
4. Can you return this from a method?
5. Why do we need super() in constructor?

✅ Topic 17: Inheritance in Java

🔹 What is Inheritance?

Inheritance is the process by which **one class (child/subclass) acquires** the properties and behaviors (**fields and methods**) of **another class (parent/superclass)**.

✅ It promotes **code reuse**, helps in **method overriding**, and supports **polymorphism**.

🔹 Why Use Inheritance?

- **Code reusability** – share common code
- To implement **method overriding**
- To support **polymorphism**
- For creating a **hierarchical class structure**

🔹 Real-Life Example

- A Car is a Vehicle
- A Dog is an Animal

We can define common features like start(), stop(), and sound() in the superclass and reuse them in subclasses.

✅ Syntax of Inheritance

java

CopyEdit

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

```
class Child extends Parent {
    void display() {
        System.out.println("Child class method");
    }
}
```

✅ Usage:

```
java
CopyEdit
Child c = new Child();
c.show();    // inherited method
c.display(); // child method
```

✅ Types of Inheritance in Java

Type	Supported in Java?	Example
Single	✅ Yes	One subclass inherits one class
Multilevel	✅ Yes	Class A → Class B → Class C
Hierarchical	✅ Yes	Multiple classes extend same class
Multiple	❌ No	Not directly (via classes)
Hybrid	❌ No	Not directly

🧩 Type Examples

🔸 1. Single Inheritance

```
java
CopyEdit
class Animal {
    void eat() {
```

```
    System.out.println("Eating...");
  }
}

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

🔹 **2. Multilevel Inheritance**

java

CopyEdit

```
class Animal {
  void eat() {}
}

class Dog extends Animal {
  void bark() {}
}

class Puppy extends Dog {
  void weep() {}
}
```

🔹 **3. Hierarchical Inheritance**

java

CopyEdit

```
class Animal {
  void eat() {}
```

}

class Dog extends Animal {}

class Cat extends Animal {}

❌ **Why Java Doesn’t Support Multiple Inheritance (via classes)?**

To **avoid ambiguity** (Diamond Problem)

java

CopyEdit

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

class B {
   void show() {}
}

// class C extends A, B ❌ Error: Java doesn't allow this
```

✅ Use **interfaces** to achieve multiple inheritance behavior

---

🔁 **Method Overriding (With Inheritance)**

java

CopyEdit

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

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

✅ **Benefits of Inheritance**

- Reduces code duplication
- Makes code modular and extensible
- Simplifies maintenance and testing
- Enables polymorphism

⚠️ **Drawbacks**

- Overuse leads to **tight coupling**
- Fragile base class problem: changes in parent may affect child
- Not suitable for all situations (favor composition over inheritance)

🧠 **Interview Questions**

1. What is inheritance?
2. Why does Java not support multiple inheritance with classes?
3. Difference between method overloading and method overriding?
4. Can constructors be inherited in Java?

❌ No, but you can call them using super()

5. What is multilevel vs hierarchical inheritance?

✅ Topic 18: Polymorphism in Java

🔹 What is Polymorphism?

Polymorphism means "**many forms**". In Java, polymorphism allows an object to behave in **multiple ways** depending on the context.

🔄 **One interface → many implementations**

🔹 Why Use Polymorphism?

- To write **flexible and extensible** code
- To achieve **runtime decision-making**
- To implement **method overriding and overloading**
- To support **dynamic behavior**

🔸 Types of Polymorphism in Java

Type	Also Known As	Happens At
Compile-Time	Static / Method Overloading	Compile Time
Runtime	Dynamic / Method Overriding	Runtime

✅ 1. Compile-Time Polymorphism (Method Overloading)

Same method name, **different parameters** (within the same class)

java

CopyEdit

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

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

✅ Usage:

```
java
CopyEdit
Calculator c = new Calculator();
System.out.println(c.add(2, 3));       // Calls int version
System.out.println(c.add(2.5, 3.5));   // Calls double version
```

✅ **2. Runtime Polymorphism (Method Overriding)**

Method in subclass has the **same signature** as in superclass

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

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

✅ Usage:

```
java
```

CopyEdit

```
Animal a = new Dog();  // Upcasting
a.sound();  // Output: Dog barks → resolved at runtime
```

🔸 **Real-Life Example**

Think of a **remote control**:

- One remote (interface)
- Can control TV, AC, Fan (different implementations)

java

CopyEdit

```
interface Remote {
   void pressPowerButton();
}

class TV implements Remote {
   public void pressPowerButton() {
      System.out.println("TV turned ON");
   }
}

class AC implements Remote {
   public void pressPowerButton() {
      System.out.println("AC turned ON");
   }
}
```

✅ Usage:

java

CopyEdit

```
Remote r = new TV();
r.pressPowerButton();  // Output: TV turned ON
```

🔁 Difference: Overloading vs Overriding

Feature	Overloading	Overriding
Class	Same class	Parent and Child class
Parameters	Must be different	Must be same
Return type	Can be different	Should be same or covariant
Access modifier	No restriction	Cannot reduce visibility
Final/Static	Can overload	Cannot override final/static

✅ Benefits of Polymorphism

- Code reusability
- Cleaner and maintainable code
- Supports runtime flexibility
- Easier to scale and modify

⚠️ Drawbacks

- Overuse may lead to **unclear behavior**
- Requires a good understanding of **inheritance and type casting**
- Slight performance cost at runtime (for dynamic binding)

🧠 Interview Questions

1. What is polymorphism?
2. Difference between method overloading and overriding?
3. What is dynamic method dispatch?
4. Can static methods be overridden?

❌ No, they are hidden (not overridden)

5. What is upcasting and downcasting?

✅ Topic 19: Abstraction in Java

🔹 What is Abstraction?

Abstraction is the process of **hiding internal details** and showing only the **essential features** of an object.

💡 Think of it like **controlling a car** – you use the steering, brake, and accelerator without knowing how the engine works inside.

🔹 Why Abstraction?

- To focus on **what** an object does, not **how** it does it
- To reduce complexity
- To build **scalable, loosely-coupled, and maintainable** systems

---

✅ Ways to Achieve Abstraction in Java

Approach	How much abstraction?
Abstract Class	Partial Abstraction (0–100%)
Interface	Full Abstraction (100%) (Before Java 8)

✅ 1. Abstract Class

🔸 Syntax:

java

CopyEdit

```
abstract class Animal {
  abstract void sound(); // no body
  void sleep() {
    System.out.println("Sleeping...");
  }
```

```
}
```

java

CopyEdit

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

🔹 **Key Rules:**

- Can have both **abstract** and **non-abstract methods**
- Can have **constructors**, **fields**, and **static methods**
- Can't be instantiated directly

✅ **2. Interface**

Introduced for **100% abstraction**, now supports **default & static methods** (since Java 8)

🔸 **Syntax:**

java

CopyEdit

```
interface Animal {
  void sound(); // implicitly public & abstract
}
```

java

CopyEdit

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

}

🔹 From Java 8:

java

CopyEdit

```
interface Vehicle {
  default void start() {
    System.out.println("Starting...");
  }
  static void service() {
    System.out.println("Servicing...");
  }
}
```

🔁 **Abstract Class vs Interface**

Feature	Abstract Class	Interface
Methods	Can be abstract or concrete	Only abstract (before Java 8), default/static after
Fields	Can have variables	Only constants (public static final)
Inheritance	Single inheritance	Multiple inheritance supported
Constructor	Can have constructors	❌ No constructors
When to use?	For shared state or partial abstraction	For total abstraction, contracts

---

🔸 **Real-Life Example**

Interface:

java

CopyEdit

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

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

✅ **Benefits of Abstraction**

- Hides complexity, shows relevant details
- Helps in achieving **loose coupling**
- Improves **security** by exposing only necessary things
- Supports **modular design**

⚠️ **Drawbacks**

- Over-abstraction may make code harder to follow
- Too many layers/interfaces may confuse junior developers

🧠 **Interview Questions**

1. What is abstraction?
2. What's the difference between abstraction and encapsulation?
3. Can abstract classes have constructors?
4. What is the difference between abstract class and interface?
5. Can an interface extend a class?

✅ Topic 20: Encapsulation in Java

🔹 What is Encapsulation?

Encapsulation is the technique of **wrapping data (variables)** and **code (methods)** together into a **single unit** (class), and **restricting direct access** to some of the object's components.

🔒 Think of a **capsule**: It keeps the medicine (data) inside and only allows controlled access.

🔹 Why Encapsulation?

- To achieve **data hiding**
- To **protect internal data**
- To make code **more secure**, **maintainable**, and **modular**

✅ How to Achieve Encapsulation in Java?

1. Declare all **fields (variables)** as private
2. Provide **public getter and setter** methods for access

🔸 Example:

java

CopyEdit

```
class Student {
   private String name;  // data hidden

   // Getter method
   public String getName() {
      return name;
   }
```

```
    // Setter method
    public void setName(String name) {
        this.name = name;
    }
}
```

✅ Usage:

```
java
CopyEdit
Student s = new Student();
s.setName("Viraj");
System.out.println(s.getName());  // Output: Viraj
```

🔸 **Real-Life Example**

ATM machine:
You press buttons (methods) to withdraw or deposit money.
You cannot access internal logic or balance storage directly — it's encapsulated.

🔁 **Encapsulation vs Abstraction**

Feature	Encapsulation	Abstraction
Focus	**How** to protect data	**What** to expose to the user
Visibility	Hides internal data using private	Hides internal complexity using methods/interfaces
Implementation	Uses private, getters/setters	Uses abstract class or interface
Example	Class with private fields	Interface with only method declarations

---

✅ **Benefits of Encapsulation**

- Protects data from unauthorized access
- Makes code easy to maintain and change
- Increases code modularity
- Allows validation logic in setters

⚠️ Drawbacks

- May require extra boilerplate code (getters/setters)
- Overuse may lead to too many access methods

🧠 Interview Questions

1. What is encapsulation in Java?
2. How is encapsulation implemented in Java?
3. Difference between abstraction and encapsulation?
4. Why do we use private fields and public getters/setters?
5. Can we access private data from outside a class?

✅ Topic 20: Encapsulation in Java

🔹 What is Encapsulation?

Encapsulation is the technique of **wrapping data (variables)** and **code (methods)** together into a **single unit** (class), and **restricting direct access** to some of the object's components.

🔒 Think of a **capsule**: It keeps the medicine (data) inside and only allows controlled access.

🔹 Why Encapsulation?

- To achieve **data hiding**
- To **protect internal data**
- To make code **more secure**, **maintainable**, and **modular**

✅ How to Achieve Encapsulation in Java?

1. Declare all **fields (variables)** as private
2. Provide **public getter and setter** methods for access

🔸 Example:

java

CopyEdit

```
class Student {
   private String name;  // data hidden

   // Getter method
   public String getName() {
      return name;
   }
```

```
    // Setter method
    public void setName(String name) {
        this.name = name;
    }
}
```

✅ Usage:

```
java
CopyEdit
Student s = new Student();
s.setName("Viraj");
System.out.println(s.getName());  // Output: Viraj
```

🔸 **Real-Life Example**

ATM machine:
You press buttons (methods) to withdraw or deposit money.
You cannot access internal logic or balance storage directly — it's encapsulated.

🔁 **Encapsulation vs Abstraction**

Feature	Encapsulation	Abstraction
Focus	**How** to protect data	**What** to expose to the user
Visibility	Hides internal data using private	Hides internal complexity using methods/interfaces
Implementation	Uses private, getters/setters	Uses abstract class or interface
Example	Class with private fields	Interface with only method declarations

---

✅ **Benefits of Encapsulation**

- Protects data from unauthorized access
- Makes code easy to maintain and change
- Increases code modularity
- Allows validation logic in setters

⚠️ Drawbacks

- May require extra boilerplate code (getters/setters)
- Overuse may lead to too many access methods

🧠 Interview Questions

1. What is encapsulation in Java?
2. How is encapsulation implemented in Java?
3. Difference between abstraction and encapsulation?
4. Why do we use private fields and public getters/setters?
5. Can we access private data from outside a class?

✅ Topic 21: static Keyword in Java

🔹 What is static in Java?

The static keyword is used for **memory management**. When a member (variable or method) is declared static, it belongs to the **class** rather than to any specific instance of the class.

📌 **Static = Shared across all objects of the class**

🔹 Why Use static?

- To **save memory** – only one copy exists for all instances
- For **utility methods** that don't depend on object state
- To access members **without creating an object**

✅ Use Cases of static

Use Case	Description
static variable	Shared variable among all instances
static method	Can be called without creating an object
static block	Runs only once when class is loaded
static class (nested)	Static inner class – doesn't depend on outer class instance

---

🔸 1. Static Variable (a.k.a. Class Variable)

java

CopyEdit

```
class Student {
   int rollNo;
   static String college = "ABC College";  // shared by all objects
```

```
  Student(int r) {
    rollNo = r;
  }

  void show() {
    System.out.println(rollNo + " " + college);
  }
}
```

✅ Usage:

```
java
CopyEdit
Student s1 = new Student(1);
Student s2 = new Student(2);
s1.show();  // 1 ABC College
s2.show();  // 2 ABC College
```

🔸 **2. Static Method**

```
java
CopyEdit
class Utility {
  static int square(int x) {
    return x * x;
  }
}
```

✅ Usage:

```
java
CopyEdit
int result = Utility.square(5);  // Output: 25
```

🔸 **Note:**

- Static methods **cannot use this**
- They **cannot access non-static (instance) variables/methods directly**

🔸 **3. Static Block**

Used to **initialize static data** or **execute code once** during class loading.

java

CopyEdit

```
class Demo {
  static {
    System.out.println("Static block runs first");
  }
}
```

✅ Usage:

java

CopyEdit

```
Demo d = new Demo();  // Static block executes before constructor
```

🔸 **4. Static Nested Class**

A class defined **inside another class** with the static keyword.

java

CopyEdit

```
class Outer {
  static class Inner {
    void show() {
      System.out.println("Static inner class");
    }
  }
```

}

✅ Usage:

java

CopyEdit

Outer.Inner obj = new Outer.Inner();

obj.show();

✅ **Benefits of static**

- Saves memory (single copy)
- Useful for utility/helper methods
- Makes accessing methods/variables easier without objects

⚠️ **Drawbacks**

- Less flexible (you can't override static methods)
- Can't access instance members directly
- Breaks object-oriented principles if overused

🧠 **Interview Questions**

1. What is the purpose of the static keyword in Java?
2. Difference between static and non-static methods?
3. Can static methods access non-static data?
4. What is a static block?
5. Can we override static methods?

🔴 **No**, static methods are not part of runtime polymorphism – they are hidden, not overridden.

✅ Topic 22: final Keyword in Java

🔹 What is final in Java?

The final keyword in Java is a **non-access modifier** used to apply **restrictions** on classes, methods, and variables.

🛡️ Once something is declared final, **you cannot change or override it**.

✅ Where Can final Be Used?

Usage	Effect
final variable	Value **cannot be changed** (constant)
final method	Method **cannot be overridden**
final class	Class **cannot be inherited** (no subclass allowed)

🔸 1. Final Variable

java

CopyEdit

```
final int x = 10;
x = 20;  // ❌ Compilation error – cannot reassign final variable
```

🔹 Final Reference Variable

java

CopyEdit

```
final Student s = new Student();
s = new Student();  // ❌ Not allowed

// BUT you can modify the object's fields
s.setName("Viraj");  // ✅ Allowed
```

🔸 2. Final Method

java

CopyEdit

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

class Dog extends Animal {
   // void sound() {} // ❌ Error: Cannot override final method
}
```

🔸 3. Final Class

java

CopyEdit

```
final class Shape {
   void draw() {
      System.out.println("Drawing shape");
   }
}

// class Circle extends Shape {} // ❌ Not allowed: final class can't be extended
```

🔸 Final with Static

java

CopyEdit

static final double PI = 3.14159;

Used for defining **constants** in Java.

🔄 **Comparison Table: final, finally, finalize**

Keyword	Usage
final	Prevent changes (variable, method, class)
finally	Used in exception handling, always executes
finalize	Method used by garbage collector before destroy (deprecated)

🔸 **Real-Life Example**

- final variable: **Employee ID** – never changes
- final method: **Bank rules** method – should not be overridden
- final class: **Math class** – cannot be extended

✅ **Benefits of final**

- **Ensures immutability** (constants)
- **Protects important methods** from being changed
- **Improves performance** (JVM can optimize final methods)
- Enforces **safe design**

⚠️ **Drawbacks**

- **Less flexibility** (can't extend/override)
- Must be **initialized immediately** or in a constructor (for variables)

🧠 **Interview Questions**

1. What does final mean in Java?
2. Difference between final class and abstract class?

3. Can a constructor be final?

❌ No, constructors cannot be final.

4. Can final methods be overloaded?

✅ Yes, but not overridden.

5. Can final variables be initialized later?

✅ Only if they are **blank final variables** initialized in constructor.

✅ Topic 23: this and super Keywords in Java

🔹 Overview

Keyword	Purpose
this	Refers to the **current class instance**
super	Refers to the **parent class (superclass)**

🔶 1. this Keyword in Java

The this keyword is a reference variable in Java that refers to the **current object** of the class.

🔸 Use Cases of this:

✅ a) To refer current class instance variable

java

CopyEdit

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

  Student(int id, String name) {
    this.id = id;      // differentiates between instance and local variables
    this.name = name;
  }
}
```

✅ b) To invoke current class method

java

CopyEdit

```
void display() {
   System.out.println("Hello");
   this.show();   // same as just calling show()
}

void show() {
   System.out.println("World");
}
```

✅ **c) To invoke current class constructor (constructor chaining)**

```
java
CopyEdit
class Demo {
   Demo() {
      this(5);
      System.out.println("Default constructor");
   }

   Demo(int x) {
      System.out.println("Parameterized constructor: " + x);
   }
}
```

✅ **d) To return the current class instance**

```
java
CopyEdit
class Sample {
   Sample getObj() {
      return this;
   }
```

}

🔶 2. super Keyword in Java

The super keyword refers to the **parent class** (superclass) and is used to access its **methods, variables, or constructors**.

🔸 Use Cases of super:

✅ a) To access parent class variables

java

CopyEdit

```
class Animal {
   String color = "white";
}

class Dog extends Animal {
   String color = "black";

   void printColor() {
      System.out.println(super.color);  // Access parent class variable
   }
}
```

✅ b) To invoke parent class method

java

CopyEdit

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

```
}

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

✅ **c) To invoke parent class constructor**

java

CopyEdit

```
class Animal {
    Animal() {
        System.out.println("Animal constructor");
    }
}

class Dog extends Animal {
    Dog() {
        super();  // calls Animal's constructor
        System.out.println("Dog constructor");
    }
}
```

🔺 If super() is not written explicitly, Java calls it automatically in the subclass constructor.

✅ **this vs super**

Feature	this	super
Refers to	Current class object	Immediate parent class object
Access	Current class variables/methods	Parent class variables/methods
Constructor	Calls current class constructor	Calls parent class constructor
Context	Same class	Inherited class (subclass)

🔸 **Real-Life Analogy**

- this: Talking about **yourself**
  → "I, myself, am responsible"
- super: Referring to your **parent**
  → "I got my discipline from my **parent**"

✅ **Benefits**

- Helps avoid naming conflicts
- Supports **constructor chaining** and **inheritance**
- Makes code **clean**, **modular**, and **organized**

⚠️ **Drawbacks**

- Can be confusing if overused
- Should be used **carefully in constructors** to avoid cyclic calls (this() chaining errors)

🧠 **Interview Questions**

1. What is this in Java?
2. When do you use this() vs super()?
3. Can you use this() and super() together in a constructor?

❌ No, both must be the **first statement**; only one can exist.

4. Can you use super in a static context?

❌ No, super and this can't be used in static methods.

✅ Topic 24: Constructor in Java

🔹 What is a Constructor?

A **constructor** is a **special method** used to **initialize objects**. It is called **automatically** when an object is created using the new keyword.

java

CopyEdit

Student s = new Student(); // Constructor is called

🔸 Key Rules of Constructors:

- Name **must match** the class name
- **No return type**, not even void
- Can be **overloaded**
- Can use this() or super() to call other constructors

🔸 Types of Constructors:

Type	Description
Default Constructor	No arguments; created by Java if none provided
Parameterized	Accepts arguments to initialize object fields
Copy Constructor	Creates a copy of an existing object (manual)

🔹 1. Default Constructor

Created automatically by Java **if no constructor** is defined.

java

CopyEdit

class Student {

  Student() {

```
    System.out.println("Default constructor");
  }
}
```

✅ Usage:

```
java
CopyEdit
Student s = new Student();  // calls default constructor
```

🔹 **2. Parameterized Constructor**

```
java
CopyEdit
class Student {
  String name;

  Student(String name) {
    this.name = name;
  }
}
```

✅ Usage:

```
java
CopyEdit
Student s = new Student("Viraj");
```

🔹 **3. Copy Constructor (User-defined)**

Java does not provide a built-in copy constructor, but you can create one manually.

```
java
CopyEdit
class Student {
```

```
    String name;

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

    Student(Student s) {
        this.name = s.name;
    }
}
```

✅ Usage:

```
java
CopyEdit
Student s1 = new Student("Viraj");
Student s2 = new Student(s1);  // Copy of s1
```

🔹 **Constructor Overloading**

You can define **multiple constructors** in the same class with different parameter lists.

```
java
CopyEdit
class Student {
    Student() {
        System.out.println("Default");
    }

    Student(String name) {
        System.out.println("Name: " + name);
    }
```

```
    Student(String name, int age) {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}
```

🔹 **Constructor Chaining with this()**

java

CopyEdit

```
class Example {
    Example() {
        this(5);  // calling parameterized constructor
        System.out.println("Default");
    }

    Example(int x) {
        System.out.println("Parameterized: " + x);
    }
}
```

✅ **Real-Life Example**

Imagine creating a **Bank Account**.
When the object is created, the constructor assigns an account number, initial balance, etc.

✅ **Benefits**

- Ensures proper initialization
- Supports **overloading** for flexible object creation

- Helps in **code readability and object control**

⚠️ **Drawbacks**

- Constructors cannot be **inherited**
- You must manage **overloaded versions** manually

🔸 **Difference: Constructor vs Method**

Feature	Constructor	Method
Purpose	Initializes object	Defines behavior
Name	Same as class name	Any name
Return Type	No return type	Must have return type (even void)
Call	Called automatically	Called manually

🧠 **Interview Questions**

1. What is a constructor?
2. How is a constructor different from a method?
3. What is constructor overloading?
4. What is a copy constructor?
5. Can a constructor be static or final?

❌ No, constructors cannot be static, final, or abstract.

✅ Topic 26: Polymorphism in Java

🔹 What is Polymorphism?

Polymorphism means **"many forms."**
It allows one entity (like a method or object) to take **multiple forms/behaviors** depending on context.

🔸 Real-Life Analogy:

A **remote** can operate **TV**, **AC**, or **Projector** — same remote, different behavior depending on the device.
Similarly, in Java, the **same method** can behave **differently** depending on how it is used.

🔹 Types of Polymorphism in Java

Type	How it Works	When it Occurs
Compile-time (Static)	Method Overloading	During compilation
Runtime (Dynamic)	Method Overriding (with Inheritance)	At runtime

---

✅ 1. Compile-Time Polymorphism (Method Overloading)

- **Same method name**, but different parameter list.
- Java determines which version to call during **compilation**.

java

CopyEdit

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

  double add(double a, double b) {
```

```
        return a + b;
    }
}
```

✅ Usage:

```
java
CopyEdit
Calculator c = new Calculator();
c.add(5, 6);        // calls int version
c.add(5.5, 6.7);    // calls double version
```

✅ **2. Runtime Polymorphism (Method Overriding)**

- Method in **subclass overrides** the method in **superclass**
- JVM decides which version to call based on the object type — at **runtime**

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

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

✅ Usage:

```
java
```

CopyEdit

```
Animal a = new Dog();  // Upcasting
a.sound();          // Output: Dog barks
```

🔹 **Key Concepts Related to Polymorphism**

Concept	Description
Overloading	Same method name, different parameters
Overriding	Same method name, same parameters, different class
Upcasting	Parent reference to child object
Dynamic Dispatch	JVM chooses method at runtime

✅ **Benefits of Polymorphism**

- **Flexibility**: Code can handle different object types in a unified way
- **Reusability**: Same interface or method can be reused
- **Extensibility**: Easily add new functionality through new classes

---

⚠️ **Drawbacks**

- Can make debugging harder (runtime errors)
- Improper overriding may lead to unexpected behavior

🔸 **Real-Life Use Case**

In a **payment system**:

java

CopyEdit

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

```
public class App {
   public static void main(String[] args) {
      Payment p = new UPI();  // or new Card()
      p.pay();  // Dynamic behavior
   }
}
```

🧠 Interview Questions

1. What is polymorphism in Java?
2. What are the types of polymorphism?
3. What is the difference between overloading and overriding?
4. What is dynamic method dispatch?
5. Can constructors be overloaded? (✅ Yes)
6. Can constructors be overridden? (❌ No, because they are not inherited)

✅ Topic 27: Abstraction vs Encapsulation in Java

This is a **very commonly confused** topic — so I'll make it super simple and crystal clear with definitions, comparisons, examples, benefits, and real-life analogies.

🔹 What is Abstraction?

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

🔍 Focuses on **what an object does**, not **how** it does it.

✅ Real-Life Example (Abstraction):

When you drive a **car**, you just:

- Press the **accelerator** to speed up
- Press the **brake** to stop

But you don't care how the engine, fuel system, or brake mechanism works — that's hidden.

✅ Java Support for Abstraction:

- **Abstract classes**
- **Interfaces**

java

CopyEdit

```
interface Animal {
   void sound(); // no implementation
}


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

```
    }
}
```

🔹 What is Encapsulation?

Encapsulation is the process of **wrapping data (variables) and code (methods)** together as a single unit, and restricting direct access to it.

🔐 Focuses on **how to protect data** using **access modifiers** (private, public, etc.)

✅ Real-Life Example (Encapsulation):

Think of a **capsule** (medicine) — it encapsulates (wraps) all the medicine inside, so you don't get direct access to its ingredients.

✅ Java Support for Encapsulation:

By using:

- private variables
- public getter and setter methods

java

CopyEdit

```
class Student {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

}

🔁 Difference Table: Abstraction vs Encapsulation

Feature	Abstraction	Encapsulation
Definition	Hides internal implementation	Hides internal data using modifiers
Focus	On **what** an object does	On **how** to protect an object's data
Achieved by	Abstract classes, Interfaces	Access modifiers (private, public, etc.)
Goal	Reduce complexity	Increase security
User Level	User sees only necessary info	User can't access internal variables directly
Example	Driving a car (you don't see engine)	Medical capsule (you don't access ingredients)

✅ Benefits of Both

Abstraction	Encapsulation
Simplifies complex systems	Improves data security
Reduces code duplication	Helps maintain control over class fields
Enhances code readability	Improves maintainability

🧠 Interview Questions

1. What is the difference between abstraction and encapsulation?
2. How is abstraction achieved in Java?
3. What are real-life examples of encapsulation?
4. Can you achieve abstraction without encapsulation?

✅ Yes — but good design often combines both.

5. Which one improves **security** more?

👉 **Encapsulation** (hides the data)

✅ Topic 28: Interface vs Abstract Class in Java

🔹 Why Is This Topic Important?

In Java, both **interfaces** and **abstract classes** are used to achieve **abstraction**.
But they work differently and are used in **different design scenarios**.

Let's break it down fully.

🔸 What is an Abstract Class?

An abstract class:

- Is a class that **cannot be instantiated**
- May contain **abstract** (unimplemented) **and non-abstract** (implemented) methods
- Is used when classes share **common behavior**

java

CopyEdit

```
abstract class Animal {
  abstract void sound(); // abstract method

  void eat() {
    System.out.println("This animal eats food."); // concrete method
  }
}
```

✅ You can extend this class and implement its abstract methods.

🔸 What is an Interface?

An interface:

- Is a **blueprint** of a class
- Contains **only abstract methods** (until Java 7)

- From Java 8+, it can also contain:
    - default methods (with body)
    - static methods
- From Java 9+, it can contain **private** methods too

java

CopyEdit

```
interface Vehicle {
   void start(); // public + abstract by default
}
```

✅ A class implements an interface using the implements keyword.

🔁 **Difference Table: Interface vs Abstract Class**

Feature	**Interface**	**Abstract Class**
Keyword	interface	abstract class
Inheritance	Can **only be implemented**	Can be **extended**
Multiple Inheritance	✅ Supported	❌ Not supported
Access Modifiers	Methods are public by default	Can have any (private, protected, etc.)
Method Body	Only default, static, private (since Java 8/9)	Can have regular methods
Constructors	❌ Not allowed	✅ Can have constructors
Variables	public static final only	Can have any type of variable
Use Case	Define **capabilities**	Define **common base behavior**

🔍 **Example: Interface vs Abstract Class**

java

CopyEdit

```
interface Flyable {
    void fly();
}

abstract class Bird {
    void eat() {
        System.out.println("Bird eats seeds");
    }

    abstract void sing();
}

class Sparrow extends Bird implements Flyable {
    public void sing() {
        System.out.println("Sparrow sings");
    }

    public void fly() {
        System.out.println("Sparrow flies");
    }
}
```

✅ Sparrow inherits **common behavior** from Bird
✅ It also adds **capability** from Flyable

🔹 **When to Use What?**

Use Interface When	Use Abstract Class When
You want to specify a **contract** (what to do)	You want to provide **base code** + abstract methods
You need **multiple inheritance**	You need a **partial implementation**
You're working with **unrelated classes**	You're working with **related classes**

✅ Benefits

- Interfaces = **flexibility**, loose coupling
- Abstract classes = **code reuse**, stronger base design

⚠️ Drawbacks

Interface	Abstract Class
Can't provide common implementation (fully)	Can't be used for multiple inheritance

---

🧠 Interview Questions

1. Can we have a constructor in an interface? (❌ No)
2. Can a class implement multiple interfaces? (✅ Yes)
3. Can abstract class have a main() method? (✅ Yes)
4. Can an interface have default methods? (✅ Yes, from Java 8)
5. Can we achieve multiple inheritance in Java? (✅ Yes, using interfaces)

✅ Topic 29: Wrapper Classes in Java

🔹 What Are Wrapper Classes?

Wrapper classes in Java are used to **wrap primitive data types** into **objects**.
Java is an **object-oriented language**, and many frameworks (like collections, generics) **require objects** — not primitives.

✅ Example:

Primitive	Wrapper Class
int	Integer
char	Character
double	Double
boolean	Boolean

So when you write:

java

CopyEdit

```
int a = 5;          // primitive
Integer b = Integer.valueOf(a); // wrapped as object
```

🔸 Why Use Wrapper Classes?

1. To use **primitives as objects** (e.g., in Collections like ArrayList)
2. For utility methods (e.g., parseInt(), compare(), etc.)
3. To take advantage of **autoboxing** and **unboxing**
4. Needed in **generics** (List<Integer>, not List<int> ❌)

🔹 Autoboxing and Unboxing

Java automatically **converts primitives into wrapper objects** (Autoboxing)
and **wrapper objects back to primitives** (Unboxing)

✅ Autoboxing (primitive → object)

java

CopyEdit

```
int num = 10;

Integer obj = num; // auto-converted
```

✅ Unboxing (object → primitive)

java

CopyEdit

```
Integer obj = 15;

int num = obj; // auto-converted back
```

✅ No need to call Integer.valueOf() or intValue() manually — Java does it for you.

🔍 Real-Life Analogy

Imagine you're putting a **gift (primitive)** in a **box (object)** so it can be safely shipped in a system (like collections).
Later, the recipient **opens the box** and uses the gift.

✅ Commonly Used Wrapper Methods

Class	Useful Method	Use
Integer	parseInt("123")	Convert string to int
Boolean	parseBoolean("true")	Convert string to boolean
Double	parseDouble("4.56")	Convert string to double
Character	isLetter('a'), isDigit('3')	Check char type

✅ Benefits of Wrapper Classes

Feature	Benefit
Object support	Can use primitives in collections, generics

Feature	Benefit
Utility methods	parse, compare, valueOf, etc.
Autoboxing	Cleaner and simpler code
Nullability	Wrapper can be null (primitives can't)

⚠️ Drawbacks

Drawback	Explanation
Memory overhead	Objects use more memory than primitives
Performance	Slightly slower than using primitives
NullPointerException	Wrappers can be null, causing potential NPEs

---

🧠 Interview Questions

1. What is a wrapper class in Java?
2. Why do we need wrapper classes?
3. What is autoboxing and unboxing?
4. What's the difference between int and Integer?
5. Can wrapper classes be null?
6. Can we use primitives in collections?

✅ Topic 30: Arrays in Java

🔹 What is an Array?

An **array** is a **collection of similar data types** stored in **contiguous memory locations**.

📦 It acts like a **container** that holds **multiple values** of the **same type** using a **single variable name**.

✅ Example:

java

CopyEdit

```
int[] numbers = {10, 20, 30, 40};

System.out.println(numbers[2]);  // Output: 30
```

🔸 Why Use Arrays?

1. To store **multiple values** in a single variable
2. To **organize data** of the same type
3. To improve **code readability and memory efficiency**

🔹 Types of Arrays

Type	Description	Example
1D Array	Simple list of elements	int[] arr = new int[5];
2D Array	Table (rows & columns)	int[][] matrix = new int[3][3];
Multidimensional	3D, 4D... Arrays	int[][][] arr = new int[2][3][4];

🔸 Array Declaration & Initialization

java

CopyEdit

```
// Declaration
int[] marks;

// Initialization
marks = new int[5];

// Combined
int[] scores = new int[]{10, 20, 30};

// Shortcut
int[] numbers = {1, 2, 3, 4};
```

🔍 Looping Through Arrays

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

// Enhanced for-loop (foreach)
for (int n : numbers) {
   System.out.println(n);
}
```

🔹 Array Properties

- **Fixed size**: You can't change the size after creation
- **Zero-indexed**: Starts from index 0
- **Length**: Use .length (no parentheses) to get size

🔸 **Limitations of Arrays**

Limitation	Explanation
Fixed size	Can't add or remove elements dynamically
Homogeneous data only	Only same data type
No built-in methods	Unlike ArrayList (e.g., no add() or remove())

🔸 **Real-Life Analogy**

Think of an array like a **train** — each coach (index) holds one passenger (value).
You can't change the number of coaches after the train is built.

✅ **Common Interview Coding Examples**

1. Find the sum of array elements

java

CopyEdit

```
int sum = 0;
for (int n : numbers) {
   sum += n;
}
```

2. Find maximum value

java

CopyEdit

```
int max = numbers[0];
for (int n : numbers) {
   if (n > max) max = n;
}
```

🔸 **Array vs ArrayList (Quick Overview)**

Feature	Array	ArrayList
Size	Fixed	Dynamic
Type	Primitive/Object	Only Objects
Methods	No utility methods	Rich API (add(), remove())

🧠 **Interview Questions**

1. What is an array?
2. What are the types of arrays in Java?
3. Can arrays store objects? (✅ Yes)
4. What are the limitations of arrays?
5. How do you find the length of an array?
6. Difference between Array and ArrayList?

✅ Topic 30: Arrays in Java

🔹 What is an Array?

An **array** is a **collection of similar data types** stored in **contiguous memory locations**.

📦 It acts like a **container** that holds **multiple values** of the **same type** using a **single variable name**.

✅ Example:

java

CopyEdit

```
int[] numbers = {10, 20, 30, 40};

System.out.println(numbers[2]);  // Output: 30
```

🔸 Why Use Arrays?

1. To store **multiple values** in a single variable
2. To **organize data** of the same type
3. To improve **code readability and memory efficiency**

🔹 Types of Arrays

Type	Description	Example
1D Array	Simple list of elements	int[] arr = new int[5];
2D Array	Table (rows & columns)	int[][] matrix = new int[3][3];
Multidimensional	3D, 4D... Arrays	int[][][] arr = new int[2][3][4];

🔸 Array Declaration & Initialization

java

CopyEdit

```
// Declaration
int[] marks;

// Initialization
marks = new int[5];

// Combined
int[] scores = new int[]{10, 20, 30};

// Shortcut
int[] numbers = {1, 2, 3, 4};
```

🔍 Looping Through Arrays

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

// Enhanced for-loop (foreach)
for (int n : numbers) {
   System.out.println(n);
}
```

🔹 Array Properties

- **Fixed size**: You can't change the size after creation
- **Zero-indexed**: Starts from index 0
- **Length**: Use .length (no parentheses) to get size

🔸 **Limitations of Arrays**

Limitation	Explanation
Fixed size	Can't add or remove elements dynamically
Homogeneous data only	Only same data type
No built-in methods	Unlike ArrayList (e.g., no add() or remove())

🔸 **Real-Life Analogy**

Think of an array like a **train** — each coach (index) holds one passenger (value).
You can't change the number of coaches after the train is built.

✅ **Common Interview Coding Examples**

1. Find the sum of array elements

java

CopyEdit

```
int sum = 0;
for (int n : numbers) {
   sum += n;
}
```

2. Find maximum value

java

CopyEdit

```
int max = numbers[0];
for (int n : numbers) {
   if (n > max) max = n;
}
```

🔸 **Array vs ArrayList (Quick Overview)**

Feature	Array	ArrayList
Size	Fixed	Dynamic
Type	Primitive/Object	Only Objects
Methods	No utility methods	Rich API (add(), remove())

🧠 **Interview Questions**

1. What is an array?
2. What are the types of arrays in Java?
3. Can arrays store objects? (✅ Yes)
4. What are the limitations of arrays?
5. How do you find the length of an array?
6. Difference between Array and ArrayList?

✅ Topic 31: Strings in Java

🔹 What is a String?

In Java, a **String** is a **sequence of characters**. It is an **object** of the String class, used to represent text.

💡 Strings in Java are **immutable** — once created, they **cannot be changed**.

🔸 Why Use Strings?

1. To represent and manipulate **textual data**
2. Useful for **user input**, **file handling**, **communication**, etc.
3. Java has **built-in methods** for String operations

✅ How to Create a String

java

CopyEdit

```
// Using string literal
String s1 = "Hello";


// Using new keyword
String s2 = new String("Hello");
```

🔸 String Pool (Memory Concept)

- When you create a string using **literals**, it goes into a **String pool** (in heap memory).
- Java **reuses** strings from this pool to **save memory**.

java

CopyEdit

```
String a = "Java";
```

String b = "Java";

System.out.println(a == b); // true — same reference

But with new String():

java

CopyEdit

String a = new String("Java");

String b = new String("Java");

System.out.println(a == b); // false — different objects

🔹 **Common String Methods**

Method	Purpose
length()	Returns number of characters
charAt(index)	Returns character at specified index
substring(start, end)	Extracts part of string
equals(str)	Compares content (case-sensitive)
equalsIgnoreCase(str)	Compares content (case-insensitive)
toUpperCase()	Converts to uppercase
toLowerCase()	Converts to lowercase
trim()	Removes leading/trailing spaces
replace(old, new)	Replaces character/word
contains(str)	Checks if string contains a substring

🔍 **Example:**

java

CopyEdit

String name = "Java Programming";

```
System.out.println(name.length());        // 16
System.out.println(name.charAt(5));       // P
System.out.println(name.substring(0, 4));   // Java
System.out.println(name.toUpperCase());     // JAVA PROGRAMMING
```

🔸 **String Comparison: == vs equals()**

Comparison Type	Use	Compares
==	Reference	Memory location
equals()	Logical	Actual content

java

CopyEdit

```
String s1 = "Java";
String s2 = new String("Java");

System.out.println(s1 == s2);      // false
System.out.println(s1.equals(s2));  // true
```

🔹 **Immutability of Strings**

Once a String object is created, its value **cannot be changed**.

java

CopyEdit

```
String s = "Hello";
s.concat(" World");
System.out.println(s); // "Hello" — original string remains unchanged
```

✅ If you want a new value, you must **assign the result**:

java

CopyEdit

```
s = s.concat(" World");

System.out.println(s); // "Hello World"
```

🔸 Mutable Alternatives to String

Class	Description
StringBuilder	Fast, not thread-safe (preferred)
StringBuffer	Thread-safe (synchronized)

🔍 Real-Life Analogy

Think of a String like a **sealed paper**. Once written, you can't change it.
If you want to change the content, you **create a new paper** (new object).

✅ Benefits of String

- Rich built-in methods
- Memory-efficient with String Pool
- Easy to use and compare
- Immutable = thread-safe in many cases

⚠️ Drawbacks

- **Immutability** can be inefficient for heavy modifications (use StringBuilder)
- Memory may be wasted if too many temporary objects are created

---

🧠 Interview Questions

1. What is a String in Java?
2. What is the difference between == and equals() in Strings?
3. Why are Strings immutable in Java?
4. What is the String pool?

5. Difference between String, StringBuilder, and StringBuffer?
6. Which one is thread-safe: StringBuilder or StringBuffer?

✅ Topic 33: Java Collections Framework

🔹 What is the Java Collections Framework?

The **Java Collections Framework (JCF)** is a set of **classes and interfaces** that provide **built-in data structures** and **algorithms** to store, manage, and manipulate groups of data efficiently.

💡 It is part of java.util package and one of the most commonly used features in Java.

🔸 Why Use Collections?

- To avoid using arrays (which are **fixed in size** and **less flexible**)
- To handle data dynamically (add, remove, sort, search, etc.)
- Provides **ready-to-use implementations** of data structures like lists, sets, queues, maps, etc.

🔹 Core Interfaces in Collections Framework

Interface	Description	Implementations Example
List	Ordered collection with duplicates allowed	ArrayList, LinkedList
Set	No duplicate elements	HashSet, LinkedHashSet, TreeSet
Queue	Elements processed in FIFO order	LinkedList, PriorityQueue
Deque	Double-ended queue	ArrayDeque, LinkedList
Map	Key-value pairs (not part of Collection interface)	HashMap, TreeMap, LinkedHashMap

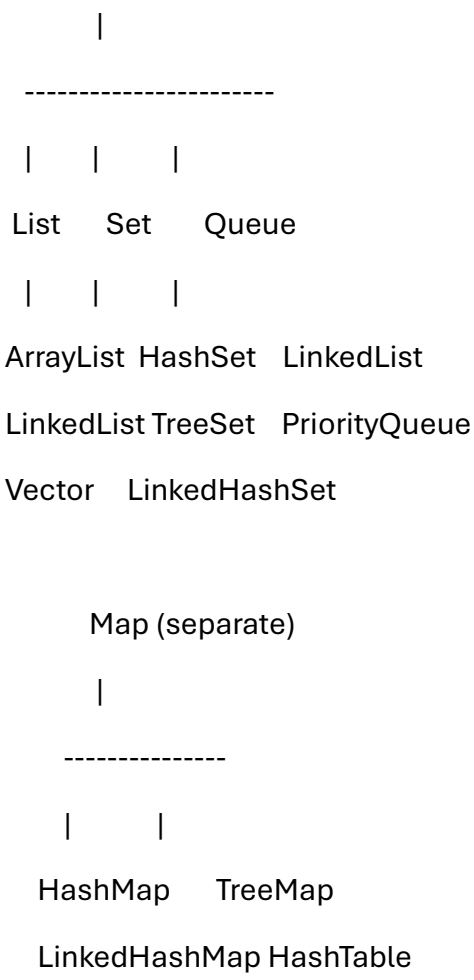
---

🔹 Diagram of Collections Hierarchy

mathematica

CopyEdit

Collection



🔹 **Differences Between Key Interfaces**

Feature	List	Set	Map
Order	Maintains order	No guaranteed order	Keys are unique, unordered
Duplicates	Allowed	Not allowed	Keys: no, Values: yes
Index Access	Yes (get(index))	No	No index-based access

🔹 **Important Implementations**

✅ **List**

- **ArrayList**: Fast for searching, slow for insertion/deletion
- **LinkedList**: Good for insertion/deletion, slow for search
- **Vector**: Thread-safe version of ArrayList (legacy)

✅ Set

- **HashSet**: No duplicates, unordered, uses hash table
- **LinkedHashSet**: Maintains insertion order
- **TreeSet**: Sorted set (uses Red-Black tree)

✅ Map

- **HashMap**: Fastest, unordered
- **LinkedHashMap**: Maintains insertion order
- **TreeMap**: Sorted by keys
- **Hashtable**: Thread-safe, legacy

🔍 Real-Life Example

Scenario: You're developing a student registration system.

- Use a List<Student> to maintain a list of registered students.
- Use a Set<String> to store unique course names.
- Use a Map<Integer, Student> to store students by their ID.

✅ Benefits of Collections Framework

- Ready-to-use data structures
- Reduces programming effort
- Easy to manage and manipulate data
- Built-in sorting and searching
- Interface-based architecture (easy to switch implementations)

⚠️ Drawbacks / Considerations

- Performance depends on which implementation you choose
- May need synchronization for multi-threading
- Some have overhead if used unnecessarily (e.g., TreeMap for unordered data)

🧠 Interview Questions

1. What is the Java Collections Framework?
2. Difference between ArrayList and LinkedList?
3. How does HashMap work internally?
4. What are the differences between Set and List?
5. Why Map is not part of the Collection interface?
6. Difference between HashMap and Hashtable?
7. When would you use a TreeSet?

✅ Topic 34: ArrayList vs LinkedList in Java

🔹 Why Compare ArrayList and LinkedList?

Both ArrayList and LinkedList are part of the List interface in Java. They allow:

- Duplicate elements
- Maintained insertion order
- Dynamic resizing

But their **internal implementations**, **performance**, and **use cases** are quite different.

🔸 Basic Differences

Feature	ArrayList	LinkedList
Internal Structure	Dynamic array	Doubly linked list
Access Time (get/set)	Fast O(1)	Slow O(n)
Insert/Delete (middle)	Slow O(n) (shift elements)	Fast O(1) (if pointer known)
Insert/Delete (end)	Fast O(1) unless resizing occurs	Fast O(1)
Memory Usage	Less memory	More memory (due to node pointers)
Thread-Safe?	❌ Not by default	❌ Not by default

🔹 When to Use What?

Scenario	Use This
Frequent random access (get/set)	ArrayList
Frequent insertion/deletion (especially middle)	LinkedList
Memory efficient with large size	ArrayList

Scenario	Use This
You need a queue-like behavior (FIFO)	LinkedList

🔹 **How They Work Internally**

✅ **ArrayList**

- Backed by a **resizable array**.
- When size exceeds, it creates a new larger array and **copies** all elements.
- Quick access: array[index].

java

CopyEdit

```
ArrayList<String> list = new ArrayList<>();
list.add("Java");
System.out.println(list.get(0)); // O(1)
```

✅ **LinkedList**

- Made up of **nodes**, where each node has:
  - data
  - next pointer
  - prev pointer (in case of doubly linked list)

java

CopyEdit

```
LinkedList<String> list = new LinkedList<>();
list.add("Java");
System.out.println(list.get(0)); // O(n)
```

🔍 **Real-Life Analogy**

- **ArrayList** = Like a row of lockers (indexed). Very fast if you know the locker number.

- **LinkedList** = Like a treasure map, where you go from clue to clue until you find what you want.

✅ Code Example & Performance

java

CopyEdit

```
List<String> arrayList = new ArrayList<>();

List<String> linkedList = new LinkedList<>();


// Add 100,000 elements

long start = System.currentTimeMillis();

for (int i = 0; i < 100000; i++) arrayList.add("A");

long end = System.currentTimeMillis();

System.out.println("ArrayList add: " + (end - start));


start = System.currentTimeMillis();

for (int i = 0; i < 100000; i++) linkedList.add("A");

end = System.currentTimeMillis();

System.out.println("LinkedList add: " + (end - start));
```

➞ Insertion at end is similar, but random access (get(i)) in LinkedList will be much slower.

✅ Advantages

ArrayList	LinkedList
Fast access by index	Fast insertion/deletion in middle
Less memory overhead	Better for frequent add/remove

⚠️ Disadvantages

ArrayList	LinkedList
Slow insert/delete in middle	Slow access by index
Resizing can be costly	More memory usage (pointers)

🧠 Interview Questions

1. Difference between ArrayList and LinkedList?
2. Which is better for inserting elements in the middle?
3. Which is better for accessing elements by index?
4. Why does LinkedList consume more memory?
5. Can you implement your own LinkedList?

✅ Topic 35: HashMap in Java

🔹 What is HashMap?

A **HashMap** is a part of the java.util package. It stores data in **key-value** pairs, where each key must be **unique**, but values can be **duplicate**.

💡 HashMap is one of the most commonly used **data structures** in Java for quick lookups and storage.

🔸 Why Use HashMap?

- To store data where each value is associated with a key (like a dictionary).
- Fast **search**, **insert**, and **delete** operations (average O(1)).
- Example: Student ID (key) → Student Name (value)

🔹 Key Features of HashMap

Feature	Description
Key-Value Storage	Stores data as key → value
Allows null keys/values	1 null key, many null values allowed
Not thread-safe	Use ConcurrentHashMap for safety
No ordering guaranteed	Use LinkedHashMap for insertion order
No duplicate keys allowed	Last value overwrites previous key

🔹 HashMap Syntax

java

CopyEdit

```
Map<Integer, String> map = new HashMap<>();

map.put(101, "John");

map.put(102, "Emma");
```

System.out.println(map.get(101)); // Output: John

🔍 How Does HashMap Work Internally?

✅ Internally uses:

- **Hashing** to find bucket
- **Array of LinkedLists or TreeNodes** (called buckets)
- put(key, value) computes **hash code** of key → finds bucket → stores Entry

➡ Collision Handling:

- Uses **chaining** (LinkedList or TreeNode)
- If multiple keys hash to the same bucket, they are stored in a list

➡ From Java 8 onwards:

- Uses **Red-Black Tree** when number of entries in a bucket exceeds 8 to improve performance

---

🔹 HashMap Methods

Method	Description
put(key, val)	Adds or updates entry
get(key)	Retrieves value by key
remove(key)	Removes entry by key
containsKey(k)	Checks if key exists
containsValue(v)	Checks if value exists
keySet()	Returns all keys
values()	Returns all values
entrySet()	Returns key-value pairs

🔍 Real-Life Analogy

HashMap is like a **real-life dictionary**:

- You (key) → definition (value)
- You look up by key, not value.

✅ **Example Code**

java

CopyEdit

```
import java.util.*;


public class HashMapExample {
   public static void main(String[] args) {
      HashMap<String, String> capital = new HashMap<>();
      capital.put("India", "Delhi");
      capital.put("USA", "Washington DC");
      capital.put("UK", "London");


      System.out.println(capital.get("India")); // Delhi
   }
}
```

✅ **Advantages**

- Fast performance for search, insert, delete (avg O(1))
- Easy key-value storage
- Allows null values
- Dynamically resizable

⚠️ **Drawbacks**

- Not thread-safe (use ConcurrentHashMap for thread safety)
- Unpredictable iteration order

- Hash collisions degrade performance
- Overwrites if key is duplicated

🧠 Interview Questions

1. How does HashMap work internally?
2. What is the time complexity of put/get in HashMap?
3. What is hash collision? How does HashMap handle it?
4. What happens when you insert a duplicate key?
5. Difference between HashMap and Hashtable?
6. How does HashMap improve performance in Java 8?

✅ Topic 36: Hashtable in Java

🔹 What is a Hashtable?

A **Hashtable** is a data structure in Java that stores data as **key-value pairs**, just like a HashMap, but it is **synchronized (thread-safe)** and **older (legacy)**.

💡 It is part of java.util and was present even before the Collections Framework was introduced (Java 1.0).

🔸 Why Use Hashtable?

- When you need a **thread-safe** map in **single-threaded or legacy code**.
- To store and access data using a key in **multi-threaded environments** (before ConcurrentHashMap existed).

🔹 Key Features of Hashtable

Feature	Description
Thread-safe	Yes, all methods are synchronized
Allows null key/value	❌ Neither null key nor null value allowed
Order	No guarantee of insertion or access order
Performance	Slower due to synchronization overhead
Legacy	Part of legacy code, replaced by newer Maps

🔹 Syntax Example

java

CopyEdit

```
import java.util.Hashtable;


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

```
        Hashtable<Integer, String> ht = new Hashtable<>();
        ht.put(1, "Java");
        ht.put(2, "Python");

        System.out.println(ht.get(1)); // Output: Java
    }
}
```

🔹 Hashtable vs HashMap

Feature	HashMap	Hashtable
Thread Safety	❌ Not thread-safe	✅ Synchronized
Null Keys/Values	✅ Allows 1 null key, many null values	❌ Does NOT allow any null
Performance	Fast in single-threaded apps	Slower due to sync
Legacy	Modern	Legacy
Use in new code?	Yes (preferred)	No (not recommended now)

🔍 Internal Working of Hashtable

- Uses **buckets** and **hashing** like HashMap
- On calling put(), it calculates a **hash**, finds the bucket, and stores the key-value entry
- Uses **synchronized methods**, so only one thread can modify the Hashtable at a time

---

🔍 Real-Life Analogy

- Think of Hashtable as a **locked cabinet**:
  Everyone has to wait in line to read or write something.

✅ Useful Methods

Method	Description
put(key, value)	Adds or updates entry
get(key)	Retrieves value
remove(key)	Removes entry
containsKey(key)	Checks if key exists
contains(value)	Checks if value exists
elements()	Returns enumeration of values
keys()	Returns enumeration of keys

✅ Advantages

- Thread-safe by default
- Quick data access (O(1) average)
- Built-in legacy structure for key-value storage

⚠️ Disadvantages

- **Synchronized**: Slower than HashMap in single-threaded applications
- **No nulls** allowed (neither key nor value)
- **Outdated**: Not used in modern Java (use ConcurrentHashMap instead)
- No ordering

🧠 Interview Questions

1. What is Hashtable in Java?
2. How is Hashtable different from HashMap?
3. Why is Hashtable considered legacy?
4. Can you store null keys/values in Hashtable?
5. Is Hashtable thread-safe? How?
6. When should you use ConcurrentHashMap over Hashtable?

✅ Topic 37: ConcurrentHashMap in Java

🔹 What is ConcurrentHashMap?

ConcurrentHashMap is a part of the java.util.concurrent package. It is a **thread-safe**, **high-performance** implementation of the Map interface that allows **concurrent access** without locking the whole map.

💡 It is a modern replacement for Hashtable in multithreaded environments.

🔸 Why Use ConcurrentHashMap?

- To avoid the **performance bottlenecks** of Hashtable
- For **high-concurrency** applications like web servers, caching systems, etc.
- To allow **multiple threads to read and write** safely and efficiently

---

🔹 Key Features

Feature	Description
Thread-safe	✅ Yes, highly concurrent
Null keys/values	❌ Null keys and null values **not allowed**
Locking mechanism	✅ Uses **segment-based (bucket-based) locking** instead of full-map lock
Performance	✅ Better than Hashtable and Collections.synchronizedMap()
Introduced in	Java 5 (java.util.concurrent)

🔹 Syntax Example

java

CopyEdit

```
import java.util.concurrent.ConcurrentHashMap;
```

```
public class Main {
  public static void main(String[] args) {
    ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<>();
    map.put(1, "Java");
    map.put(2, "Python");

    System.out.println(map.get(1)); // Java
  }
}
```

🔍 How It Works Internally

- **Hash-based storage** like HashMap
- Instead of locking the entire map, it **locks only parts (segments)** — this allows **high concurrency**
- From Java 8 onwards, it uses **bucket-level synchronization** with **CAS (Compare-And-Swap)** and **synchronized blocks**

🔹 Real-Life Analogy

Imagine a **library** with multiple sections:

- Hashtable: Locks the entire library — only one person allowed at a time.
- ConcurrentHashMap: Locks **individual sections** — multiple readers and writers can work in different areas at the same time.

✅ Useful Methods

Method	Description
put(key, value)	Inserts or updates a key-value
get(key)	Gets the value for a key
remove(key)	Removes the key-value pair

Method	Description
putIfAbsent(k, v)	Adds only if key is not already present
replace(k, v)	Replaces existing value
forEach()	Performs an action for each entry
computeIfAbsent(k, f)	Computes value only if key is missing

✅ Advantages

- Highly **efficient** in multi-threaded environments
- **Scalable** performance under heavy concurrent access
- Avoids **ConcurrentModificationException**
- **No need for external synchronization**

⚠️ Disadvantages

- Slightly **slower than HashMap** in single-threaded code
- **No null keys or values allowed**
- **Complexity** of internal implementation

🧠 Interview Questions

1. What is ConcurrentHashMap?
2. How is it different from Hashtable?
3. Can you store null in a ConcurrentHashMap?
4. What is segment locking?
5. What happens if two threads put same key?
6. Why use putIfAbsent() instead of put()?

🔄 Quick Comparison

Feature	Hashtable	ConcurrentHashMap
Thread-safe	✅ Yes (synchronized)	✅ Yes (lock-striping)
Null keys/values	❌ Not allowed	❌ Not allowed
Performance	❌ Slower	✅ Much faster
Modern approach	❌ Legacy	✅ Preferred in new code

✅ Topic 38: Iterator in Java

🔹 What is an Iterator?

An **Iterator** is an object that enables you to **traverse (loop)** through a **Collection** (like ArrayList, HashSet, etc.) one element at a time.

💡 It is part of the **Java Collections Framework** and is found in java.util.Iterator.

🔸 Why Use Iterator?

- To **safely iterate** over elements in a collection.
- To **remove elements** during iteration without causing ConcurrentModificationException.
- It provides a **universal way** to iterate through all types of collections.

🔹 Common Use Cases

- Looping through a list, set, or queue
- Removing items safely during iteration
- Avoiding index-based iteration (which is not supported by Set, Queue, etc.)

🔹 Syntax Example

java

CopyEdit

```
import java.util.*;


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

```
        names.add("Charlie");

        Iterator<String> itr = names.iterator();

        while (itr.hasNext()) {
            String name = itr.next();
            System.out.println(name);
        }
    }
}
```

🔹 **Methods of Iterator**

Method	Description
hasNext()	Returns true if next element exists
next()	Returns the next element
remove()	Removes current element from collection

🔍 **Real-Life Analogy**

Imagine a **TV remote** as an iterator:

- hasNext() → Checks if next channel exists
- next() → Goes to the next channel
- remove() → Deletes a channel from the list

🔍 **Internal Working**

- iterator() method from a collection returns an **Iterator object**.
- Internally maintains a **pointer/index** to keep track of the current element.
- Works in **forward direction only**.

🔹 **Using remove() Safely**

java

CopyEdit

```
Iterator<String> itr = names.iterator();
while (itr.hasNext()) {
   String name = itr.next();
   if (name.equals("Bob")) {
      itr.remove(); // Safe removal
   }
}
```

❗ Don’t use list.remove() inside a loop — always use iterator.remove().

🔹 **Limitations of Iterator**

- Only **forward direction** (no backward traversal)
- **No index access**
- Can only be used **once** per traversal
- Cannot **modify** elements directly

✅ **Advantages**

- Works with all collection types
- Prevents ConcurrentModificationException when using remove()
- Clean, simple, and universal traversal

⚠️ **Drawbacks**

- No support for reverse iteration (use ListIterator instead)
- No direct access to index or modify operation

- Can only remove elements, not add or update

🧠 Interview Questions

1. What is an Iterator in Java?
2. How does Iterator differ from ListIterator?
3. How do you safely remove elements during iteration?
4. What exception does next() throw if no element is left?
5. Can Iterator be used with Set?

🔄 Iterator vs ListIterator vs Enumeration

Feature	Iterator	ListIterator	Enumeration
Direction	Forward only	Both (forward/back)	Forward only
Remove support	✅ Yes	✅ Yes	❌ No
Modify support	❌ No	✅ Yes	❌ No
Collections support	All	Only List	Legacy only

✅ Topic 39: ListIterator in Java

🔹 What is ListIterator?

ListIterator is an **interface** in Java that allows **bidirectional** traversal (forward and backward) of a List.
It is an **advanced version** of Iterator and works only with classes that implement the List interface (like ArrayList, LinkedList).

💡 It is defined in java.util and can **add, modify, and remove** elements while iterating.

🔸 Why Use ListIterator?

- To **navigate both forward and backward** through a list
- To **modify elements during iteration**
- To insert new elements while looping

🔹 Key Features

Feature	ListIterator
Direction	✅ Forward & Backward
Element removal	✅ Yes
Element modification	✅ Yes (set() method)
Element addition	✅ Yes (add() method)
Applicable to	Only Lists (ArrayList, LinkedList, etc.)

🔹 Syntax Example

java

CopyEdit

```
import java.util.*;
```

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

        ListIterator<String> itr = names.listIterator();

        // Forward Traversal
        while (itr.hasNext()) {
            System.out.println("Forward: " + itr.next());
        }

        // Backward Traversal
        while (itr.hasPrevious()) {
            System.out.println("Backward: " + itr.previous());
        }
    }
}
```

🔹 **Important Methods**

Method	Description
hasNext()	Checks if there's a next element
next()	Returns the next element
hasPrevious()	Checks if there's a previous element
previous()	Returns the previous element

Method	Description
add(E e)	Adds element at current position
remove()	Removes last returned element
set(E e)	Replaces last returned element

🔍 Real-Life Analogy

Think of ListIterator like a **DVD player remote**:

- next() → Next scene
- previous() → Go back
- set() → Change the current scene
- add() → Insert a new scene

🔍 Internal Working

- Maintains a **cursor/index** to track the current position
- Can move the cursor in **both directions**
- After a call to next() or previous(), you can use remove() or set() safely

---

✅ Advantages

- Bi-directional traversal
- Modify list during iteration
- Works well with all List classes

⚠️ Drawbacks

- Works **only with Lists**
- Slightly more complex than Iterator
- Cannot be used with Set or Queue

🧠 Interview Questions

1. What is ListIterator in Java?
2. How is it different from Iterator?
3. What collections can use ListIterator?
4. Can you modify a list during traversal?
5. Can ListIterator move backward?

🔄 Iterator vs ListIterator

Feature	Iterator	ListIterator
Direction	Forward only	Forward + Backward
Modify elements	❌ No	✅ Yes (set/add/remove)
Collections	All Collections	Only Lists
Add elements	❌ No	✅ Yes
Remove elements	✅ Yes	✅ Yes

✅ Topic 40: Enumeration in Java

🔹 What is Enumeration?

Enumeration is an **interface** in Java that is used to **traverse legacy collection classes** like Vector, Hashtable, etc.

It is similar to Iterator, but is **read-only** and **only supports forward traversal**.

🏷️ Package: java.util

🔸 Why Use Enumeration?

- For working with **legacy collections** (like Vector, Stack, Hashtable)
- To **read elements** one-by-one in forward direction
- To maintain compatibility with **older code**

🔹 Syntax Example

java

CopyEdit

```
import java.util.*;


public class EnumerationExample {
   public static void main(String[] args) {
      Vector<String> vector = new Vector<>();
      vector.add("Java");
      vector.add("Python");
      vector.add("C++");

      Enumeration<String> e = vector.elements();

      while (e.hasMoreElements()) {
```

```
            System.out.println(e.nextElement());
        }
    }
}
```

🔹 Key Methods in Enumeration

Method	Description
hasMoreElements()	Returns true if more elements exist
nextElement()	Returns the next element

🔍 How It Works

- Vector or Hashtable returns an Enumeration object via .elements()
- It uses an **internal index** to keep track of the position
- You can only **read/traverse** — not **modify** the collection

---

🧠 Real-Life Analogy

Think of Enumeration like an **old newspaper archive viewer** —
you can scroll forward, read content, but you **can't delete or change** anything.

✅ Advantages

- Simple and lightweight
- Good for **read-only** traversals
- Useful for **backward compatibility** with legacy code

⚠️ Limitations / Drawbacks

- Works **only with legacy collections** (Vector, Stack, Hashtable)
- **Read-only** (no remove() method)

- **Forward only** (no backward support)
- Replaced by modern Iterator and ListIterator for new code

🔄 Comparison with Iterator

Feature	Enumeration	Iterator
Direction	Forward only	Forward only
Remove support	❌ No	✅ Yes (remove())
Modify collection	❌ No	❌ No (except remove)
Applicable to	Legacy collections	All collections
Introduced in	JDK 1.0	JDK 1.2

🧠 Interview Questions

1. What is Enumeration in Java?
2. Can Enumeration remove elements?
3. How does Enumeration differ from Iterator?
4. Is Enumeration still used today?
5. Where do we find Enumeration in Java?

✅ Summary

When to Use Enumeration?

✓ If you are working with Vector or Hashtable

❌ Avoid in modern applications — use Iterator or ListIterator instead

✅ Topic 41: Generics in Java

🔹 What are Generics?

Generics enable **parameterized types** in Java — allowing you to write code that works with **any type** in a type-safe manner.

💡 Introduced in Java 5, Generics help **eliminate typecasting** and **catch type errors at compile-time**.

🔸 Why Use Generics?

- To ensure **type safety** at compile-time
- To eliminate the need for **explicit casting**
- To write **reusable, generic code**
- To improve **readability and maintainability**

🔹 Real-Life Analogy

Imagine a **reusable lunchbox** where you can specify what to put inside:

- You can create LunchBox<Apple> or LunchBox<Sandwich>, etc.
  This is like Box<T> in Generics.

🔹 Generic Syntax

java

CopyEdit

```
class Box<T> {    // T is a type parameter
  private T item;

  public void setItem(T item) {
    this.item = item;
  }
```

```
    public T getItem() {
        return item;
    }
}
```

🔸 **Usage:**

java

CopyEdit

```
Box<String> stringBox = new Box<>();
stringBox.setItem("Hello");
System.out.println(stringBox.getItem());
```

🔹 **Generic Collections Example**

java

CopyEdit

```
List<String> names = new ArrayList<>();
names.add("Virat");
// names.add(10); ❌ Compile-time error

String player = names.get(0);  // No casting needed
```

🔹 **Benefits of Generics**

Benefit	Explanation
✅ Type Safety	Prevents inserting wrong types
✅ No Casting	Avoids Object to specific type casting
✅ Reusability	Same class or method works for any data type

Benefit	Explanation
✅ Compile-Time Check	Errors caught early during compilation

🔹 **Common Generic Types Used**

Symbol	Meaning
T	Type
E	Element (in collections)
K	Key
V	Value

🔹 **Generic Method Example**

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

🔹 **Bounded Generics (extends / super)**

```
java
CopyEdit
class NumberBox<T extends Number> {
   T value;
}
```

✅ Now T can be Integer, Double, Float, etc.
❌ But T **cannot be String** or other unrelated types.

🔹 Wildcards in Generics

Syntax	Description
<?>	Unknown type
<? extends T>	Upper bound – any subtype of T
<? super T>	Lower bound – T or any supertype of T

🔍 Example with Wildcards

java

CopyEdit

```
public void printList(List<? extends Number> list) {
  for (Number n : list) {
    System.out.println(n);
  }
}
```

⚠️ Drawbacks of Generics

Limitation	Description
Type Erasure	Generic type info is erased at runtime
Cannot create instances of generic type	new T() is not allowed
Cannot use primitive types directly	Must use wrapper classes like Integer, Double
Cannot use instanceof with parameterized types	e.g. if (obj instanceof Box<String>) ❌

🧠 Interview Questions

1. What are Generics in Java?
2. What are the benefits of using Generics?
3. What is type erasure in Java?
4. What is the difference between <T>, <?>, and <? extends T>?
5. Can we use primitive types with Generics?

✅ Summary

- ✅ Introduced in Java 5
- ✅ Improves type safety and readability
- ✅ Works with classes, methods, and collections
- ❌ Cannot use primitives or new T()

✅ Topic 42: Java 8 Features (Lambda, Stream, etc.)

🔹 What is Java 8?

Java 8 is a **major release** that brought **functional programming features** and **new APIs**, transforming how we write Java code.

🗓️ Released in March 2014
🔥 Core aim: Write cleaner, more expressive, and concise code

🔸 Key Java 8 Features You Must Know

Feature	Description
✅ Lambda Expressions	Anonymous function / shorthand for functional interface
✅ Functional Interfaces	Interface with only one abstract method
✅ Stream API	Process collections in a functional style
✅ Method Reference	Shorter syntax for calling methods
✅ Default & Static Methods in Interfaces	Add methods to interfaces without breaking code
✅ Optional Class	Avoid null and handle missing values better
✅ New Date & Time API	Better than old Date/Calendar

🔹 1. Lambda Expressions

🔸 What is it?

A **Lambda** is a **function without a name**, used to provide implementation of a **functional interface**.

🔸 Syntax:

java

CopyEdit

(parameters) -> { body }

🔸 **Example:**

java

CopyEdit

Runnable r = () -> System.out.println("Running thread!");

💡 Equivalent to creating an anonymous inner class for Runnable!

🔹 **2. Functional Interfaces**

An interface with **only one abstract method**.
It may have multiple default or static methods.

🔸 **Example:**

java

CopyEdit

@FunctionalInterface

interface Calculator {

  int add(int a, int b);

}

🔹 **3. Stream API**

Used to **process collections (List, Set, etc.)** in a **declarative & functional** way.

🔸 **Example:**

java

CopyEdit

List<String> names = Arrays.asList("Ram", "Shyam", "John");

names.stream()

  .filter(n -> n.startsWith("S"))

  .forEach(System.out::println);

🔸 **Stream Methods:**

Method	Description
filter()	Filters data based on a condition
map()	Transforms data
forEach()	Loops through each element
collect()	Gathers results into a collection
sorted()	Sorts elements
reduce()	Reduces to a single value (e.g. sum)

🔹 **4. Method References**

Shorthand syntax for **calling an existing method**.

🔸 **Syntax:**

java

CopyEdit

ClassName::methodName

🔸 **Example:**

java

CopyEdit

names.forEach(System.out::println);

🔹 **5. Default and Static Methods in Interfaces**

You can now have **method bodies** in interfaces.

java

CopyEdit

```
interface MyInterface {
   default void greet() {
      System.out.println("Hello from default method");
```

```
    }

    static void info() {
        System.out.println("Static method in interface");
    }
}
```

🔹 6. Optional Class

Used to **handle null values** gracefully and avoid NullPointerException.

🔸 Example:

java

CopyEdit

```
Optional<String> name = Optional.of("Virat");

if (name.isPresent()) {
    System.out.println(name.get());
}
```

🔹 7. New Date & Time API (java.time)

A modern replacement for old Date and Calendar.

🔸 Example:

java

CopyEdit

```
LocalDate today = LocalDate.now();
LocalTime time = LocalTime.now();
LocalDate birthday = LocalDate.of(1998, 12, 15);
```

🔍 Real-Life Use Case

- Lambda: Event handling in GUIs
- Stream: Filtering employees above salary 50k
- Optional: Avoid null checks
- Date API: Scheduling appointments in apps

✅ Advantages

Benefit	Explanation
✅ Less code	No boilerplate with lambdas
✅ Functional programming	More expressive, cleaner code
✅ Better performance (Streams)	Uses lazy evaluation and parallel streams
✅ Type safety (Optional)	Avoids null pointer issues

⚠️ Drawbacks

Drawback	Details
🧠 Learning curve	Functional style may be confusing at first
🔍 Debugging complexity	Lambda stack traces are harder to read
🧪 Harder to test	Lambdas can't easily be unit tested directly

---

🧠 Important Interview Questions

1. What are lambda expressions in Java 8?
2. What is a functional interface?
3. Difference between map and flatMap?
4. What is Optional? Why is it used?
5. How is Stream API better than traditional loops?
6. Explain method reference with an example.

7. What is the advantage of default methods in interfaces?

✅ Topic 43: Functional Interface in Java

🔹 What is a Functional Interface?

A **Functional Interface** is an interface that contains **only one abstract method** (SAM: Single Abstract Method). It can have multiple default or static methods.

💡 Functional Interfaces are the foundation for **Lambda Expressions** in Java.

🔸 Why Do We Need It?

- Enables writing cleaner code using **lambdas**
- Supports **functional programming**
- Makes code **short, readable, and less error-prone**
- Essential in Java 8 **Stream API**, **event handling**, and **callbacks**

🔹 Syntax of Functional Interface

java

CopyEdit

@FunctionalInterface

interface Calculator {

  int add(int a, int b);

}

💡 @FunctionalInterface is optional but recommended.
It prevents the interface from having more than one abstract method (compile-time error).

🔹 Example With Lambda Expression

java

CopyEdit

@FunctionalInterface

```
interface Message {
   void sayHello();
}

public class Test {
   public static void main(String[] args) {
      Message msg = () -> System.out.println("Hello, Java!");
      msg.sayHello();  // Output: Hello, Java!
   }
}
```

🔹 **Built-in Functional Interfaces (From java.util.function package)**

Interface	Abstract Method	Purpose
Predicate<T>	test(T t)	Returns boolean
Function<T, R>	apply(T t)	Transforms value of type T to R
Consumer<T>	accept(T t)	Performs action on T (no return)
Supplier<T>	get()	Supplies a result of type T
BiFunction<T,U,R>	apply(T,U)	Accepts two arguments and returns R

🔹 **Real-Life Analogy**

Think of a **remote control** with only **one button** (e.g., "Power").
This is like a **functional interface** — it performs one specific task but can be used in many contexts.

🔍 **Example of Built-in Functional Interface**

java

CopyEdit

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

public class PredicateExample {
   public static void main(String[] args) {
      Predicate<String> isLong = str -> str.length() > 5;

      System.out.println(isLong.test("Java"));       // false
      System.out.println(isLong.test("JavaScript"));  // true
   }
}
```

✅ **Benefits of Functional Interface**

Benefit	Explanation
✅ Enables Lambdas	You can use lambdas to implement them
✅ Reduces Boilerplate	No need to create anonymous classes
✅ Cleaner code	Makes code easier to read and maintain
✅ Encourages FP	Functional Programming support in Java

⚠️ **Drawbacks / Limitations**

Limitation	Description
❌ Only one abstract method	Not suitable for complex behavior interfaces
❌ Not backward-compatible	Can't use with older Java versions (< Java 8)
🧪 Sometimes hard to test	Lambdas can be less testable/debuggable

---

✅ **Functional Interface vs Interface**

Feature	Functional Interface	Regular Interface
Abstract Methods	Only one (SAM)	One or more
Use Case	Lambdas, Streams, callbacks	General-purpose OOP design
Annotation	@FunctionalInterface (optional)	Not needed
Java Version	Java 8+	Java 1.0+

🧠 Common Interview Questions

1. What is a functional interface in Java?
2. Why is @FunctionalInterface annotation used?
3. What are the built-in functional interfaces in Java?
4. Can a functional interface have default methods?
5. Give a real-life use case of a functional interface.

✅ Summary

- Introduced in Java 8
- Contains **only one abstract method**
- Used with **lambdas**, **streams**, and **callbacks**
- Enables **cleaner functional programming**

✅ Topic 44: Predicate Interface in Java

🔹 What is Predicate in Java?

Predicate<T> is a **functional interface** introduced in Java 8.
It represents a **boolean-valued function** of one argument.

🔍 It's typically used to test conditions (like filtering data).

🔸 Functional Method:

java

CopyEdit

boolean test(T t);

🔸 Package:

java

CopyEdit

import java.util.function.Predicate;

🔹 Syntax Example:

java

CopyEdit

Predicate<String> isLongWord = str -> str.length() > 5;

System.out.println(isLongWord.test("Java")); // false

System.out.println(isLongWord.test("JavaWorld")); // true

🔹 Real-Life Analogy

Think of a **security scanner** that checks if someone has a valid ID.

- If yes → allow entry (return true)

- If no → deny access (return false)

Same with Predicate — it **tests a condition** and returns true or false.

🔹 **Use Case: Filtering Collection Using Predicate**

java

CopyEdit

```
List<String> names = Arrays.asList("Ram", "Shyam", "Sita", "Radha");


Predicate<String> startsWithS = name -> name.startsWith("S");


names.stream()
    .filter(startsWithS)
    .forEach(System.out::println); // Output: Shyam, Sita
```

🔹 **Combining Predicates**

You can chain multiple predicates using:

Method	Description
and()	Both conditions must be true
or()	At least one must be true
negate()	Opposite of the condition

🔸 **Example:**

java

CopyEdit

```
Predicate<String> hasLength = str -> str.length() > 3;
Predicate<String> startsWithR = str -> str.startsWith("R");


Predicate<String> combined = hasLength.and(startsWithR);
```

```
System.out.println(combined.test("Ram"));    // false
System.out.println(combined.test("Radha"));  // true
```

🔹 Benefits of Predicate

Benefit	Explanation
✅ Cleaner logic	Removes need for if-else code
✅ Reusable conditions	Can reuse predicate in multiple places
✅ Used with Streams	Great for filter() operations
✅ Composable	Easy to combine multiple predicates

🔹 Drawbacks / Limitations

Limitation	Details
❌ Only single input	Cannot accept multiple inputs (use BiPredicate for that)
❌ Only boolean return	Can't return anything else (like Object, etc.)

🔍 Predicate vs Function Interface

Feature	Predicate<T>	Function<T, R>
Method	boolean test(T t)	R apply(T t)
Purpose	Test condition	Transform value
Returns	Boolean	Any type (generic R)
Used In	Filters, validations	Mapping, conversions

🧠 Common Interview Questions

1. What is Predicate in Java 8?

2. How is Predicate different from Function?
3. Can Predicate be combined with other predicates?
4. Where is Predicate commonly used?
5. Can Predicate be used outside Streams?

✅ **Summary**

- Predicate<T> is a **functional interface** for testing conditions
- Often used with **Streams** and filter()
- Has useful methods like **and()**, **or()**, and **negate()**
- Returns true or false based on the logic

✅ Topic 46: Consumer Interface in Java

🔹 What is Consumer in Java?

Consumer<T> is a **functional interface** introduced in Java 8.
It takes one argument and **performs an operation**, but **does not return** any result.

🔍 Think of it as a "consumer" of a value — it **uses** the value to do something.

🔸 Functional Method:

java

CopyEdit

void accept(T t);

🔸 Package:

java

CopyEdit

import java.util.function.Consumer;

🔹 Syntax Example:

java

CopyEdit

Consumer<String> printer = str -> System.out.println(str);

printer.accept("Hello Java!"); // Output: Hello Java!

🔹 Real-Life Analogy

Think of a **printer** 🖨️:

- You send it a document (input).
- It prints it (action).

- But it doesn’t give anything back to you (no return value).

This is exactly how a Consumer behaves.

🔹 **Common Use Cases**

Use Case	Example
Print or log data	System.out.println()
Save to DB	dbService.save(obj)
Send email/notification	sendEmail(message)
Used in forEach()	list.forEach(consumer)

🔹 **Example: Using Consumer with List**

java

CopyEdit

```
List<String> names = Arrays.asList("Ram", "Shyam", "Seeta");


Consumer<String> consumer = name -> System.out.println("Hello " + name);


names.forEach(consumer);


// Output:
// Hello Ram
// Hello Shyam
// Hello Seeta
```

🔹 **Chaining Consumers with andThen()**

java

CopyEdit

```
Consumer<String> first = str -> System.out.println("First: " + str);
Consumer<String> second = str -> System.out.println("Second: " + str);

Consumer<String> combined = first.andThen(second);

combined.accept("Java");

// Output:
// First: Java
// Second: Java
```

✅ **Benefits**

Benefit	Explanation
✅ No return needed	Ideal when you just want to **perform an action**
✅ Works well with streams	Especially in forEach()
✅ Chainable	Combine multiple actions using andThen()

⚠️ **Drawbacks**

Drawback	Explanation
❌ No return value	You can't get any result/output
❌ Only one input	For multiple inputs, use BiConsumer

🔍 **Comparison With Other Functional Interfaces**

Interface	Input	Output	Use Case
Consumer<T>	1	void	Print, save, notify, etc.

Function<T,R>	1	1 (R) Transform, convert
Predicate<T>	1	boolean Condition check (true/false)
Supplier<T>	0	1 Produce/generate value

🧠 Common Interview Questions

1. What is the purpose of the Consumer interface in Java?
2. How is it different from Function and Predicate?
3. Can you chain Consumers?
4. How is Consumer used in forEach()?
5. What is the difference between Consumer and BiConsumer?

✅ Summary

- Consumer<T> is a functional interface that **takes an input and performs an action**
- It has no return value
- Mostly used in **logging, saving, or modifying external states**
- Supports chaining via **andThen()**

✅ Topic 47: Supplier Interface in Java

🔹 What is Supplier in Java?

Supplier<T> is a **functional interface** introduced in Java 8.
It **does not take any input** but **returns a result** of type T.

🔍 It is used to **supply or generate values** on demand — typically used for deferred/lazy execution.

🔸 Functional Method:

java

CopyEdit

T get();

🔸 Package:

java

CopyEdit

import java.util.function.Supplier;

🔹 Syntax Example:

java

CopyEdit

Supplier<String> supplier = () -> "Hello from Supplier!";

System.out.println(supplier.get()); // Output: Hello from Supplier!

🔹 Real-Life Analogy

Think of a **water dispenser**.

- You press a button (no input).

- It gives you water (output).
  That's how a Supplier works — **no input**, just **output** when needed.

🔹 **Use Cases in Real Projects**

Use Case	Example
Lazy object creation	Create an object **only when needed**
Supply configuration	Get config values from a file or DB
Fallback values	Provide default when value is missing
Testing/mock data	Generate dummy data in unit tests

🔹 **Example: Supplying Random Values**

java

CopyEdit

```
Supplier<Double> randomSupplier = () -> Math.random();


System.out.println(randomSupplier.get()); // Each call gives a new random number
```

🔹 **Example: Lazy Initialization**

java

CopyEdit

```
Supplier<List<String>> listSupplier = ArrayList::new;


List<String> names = listSupplier.get();  // Creates a new ArrayList when called
```

✅ **Benefits of Supplier**

Benefit	Explanation
✅ Lazy Evaluation	Only creates/gets value **when needed**
✅ Simple Syntax	No input → just a return
✅ Clean Code	Makes deferred value generation more readable

⚠️ Drawbacks of Supplier

Drawback	Explanation
❌ No input accepted	If you need input, use Function or BiFunction
❌ Stateless by default	Doesn't hold state unless you manage it

---

🔍 Comparison With Other Functional Interfaces

Interface	Input	Output	Purpose
Supplier<T>	None	T	Generate/provide a value
Consumer<T>	T	void	Perform action using input
Function<T,R>	T	R	Transform input to output
Predicate<T>	T	boolean	Test condition

🧠 Common Interview Questions

1. What is the Supplier interface in Java?
2. How is Supplier different from Consumer or Function?
3. Give a real-life use case of Supplier.
4. Can Supplier be used in caching or lazy loading?
5. Is Supplier used in multithreaded environments?

✅ Summary

- Supplier<T> is a functional interface used to **supply/generate values**

- It takes **no input** and **returns a value**
- Great for **lazy initialization**, **mocking**, and **default fallbacks**
- Used in scenarios where the value is needed **later or on-demand**

✅ Topic 48: BiPredicate, BiFunction, and BiConsumer in Java

These are **extended versions** of their single-argument counterparts (Predicate, Function, and Consumer), and are used when **two inputs** are involved.

🔷 1. BiPredicate<T, U>

✅ What is BiPredicate?

- It's a functional interface that takes **two inputs** and returns a **boolean** (true/false).
- Think of it as a condition checker with **two arguments**.

🔸 Functional Method:

java

CopyEdit

boolean test(T t, U u);

🔸 Example:

java

CopyEdit

BiPredicate<String, String> equalIgnoreCase =

(s1, s2) -> s1.equalsIgnoreCase(s2);

System.out.println(equalIgnoreCase.test("java", "JAVA")); // true

---

🔹 Real-Life Use Case:

Check if a user-entered password matches stored password:

java

CopyEdit

BiPredicate<String, String> passwordMatch = (actual, entered) -> actual.equals(entered);

✅ **Benefits:**

- Allows comparison or condition check using **two values**
- Simplifies complex logic inside if conditions

⚠️ **Drawbacks:**

- Only returns a boolean; no ability to return data

🔷 **2. BiFunction<T, U, R>**

✅ **What is BiFunction?**

- Takes **two inputs** (T and U) and **returns** a result of type R.
- It's useful for **combining two values** into one.

🔸 **Functional Method:**

java

CopyEdit

R apply(T t, U u);

🔸 **Example:**

java

CopyEdit

BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;

System.out.println(add.apply(10, 20)); // 30

🔹 **Real-Life Use Case:**

Combine first name and last name into full name:

java

CopyEdit

BiFunction<String, String, String> fullName = (first, last) -> first + " " + last;

✅ **Benefits:**

- Combines two objects or values and returns a result
- Commonly used in merging data, string formatting, etc.

⚠️ **Drawbacks:**

- Only handles two inputs — for more, you'd need custom interfaces or chaining

---

🔷 **3. BiConsumer<T, U>**

✅ **What is BiConsumer?**

- Takes **two inputs** and **performs an operation**, returning nothing (like Consumer but with 2 inputs).

🔸 **Functional Method:**

java

CopyEdit

void accept(T t, U u);

🔸 **Example:**

java

CopyEdit

BiConsumer<String, Integer> printNameAndAge =

(name, age) -> System.out.println(name + " is " + age + " years old");

printNameAndAge.accept("John", 30);

🔹 **Real-Life Use Case:**

Logging user activity:

java

CopyEdit

```
BiConsumer<String, String> log = (user, action) ->
   System.out.println("User " + user + " performed: " + action);
```

✅ **Benefits:**

- Useful for logging, storing key-value pairs, and processing dual inputs
- Clean syntax with lambda

⚠️ **Drawbacks:**

- Cannot return a value (for that, use BiFunction)

🔍 **Summary Table**

Interface	Inputs	Output	Use Case Example
BiPredicate	2	boolean	Compare two values
BiFunction	2	1 (any type)	Combine two inputs into one result
BiConsumer	2	void	Perform an action using two inputs

🧠 **Common Interview Questions**

1. What is the difference between Predicate and BiPredicate?
2. When would you use BiFunction over Function?
3. Give a real-world example of BiConsumer.
4. Can BiPredicate be used in stream filtering?
5. Why do we use Bi versions of functional interfaces?

✅ **Summary**

- **BiPredicate** is used for condition checking between two inputs
- **BiFunction** is for combining two inputs into a result
- **BiConsumer** is for performing actions using two inputs (no return)

✅ Topic 49: Method Reference and Constructor Reference in Java 8

🔷 What is Method Reference?

Method Reference is a **short-cut** notation of a **lambda expression** to call a method **directly**.

It helps write **cleaner and more readable code** when the lambda just **calls an existing method**.

🔹 Syntax:

java

CopyEdit

ClassName::methodName

🔹 When to Use?

Whenever your lambda expression looks like this:

java

CopyEdit

(param) -> object.method(param)

You can replace it with:

java

CopyEdit

object::method

🔷 Types of Method References

Type	Syntax	Example
1. Static method	ClassName::staticMethod	Math::sqrt
2. Instance method of a particular object	object::instanceMethod	System.out::println

Type	Syntax	Example
3. Instance method of an arbitrary object of a class	ClassName::instanceMethod	String::toUpperCase
4. Constructor reference	ClassName::new	ArrayList::new

✅ **Type 1: Reference to a Static Method**

java

CopyEdit

```
class Demo {
   public static void sayHello() {
      System.out.println("Hello!");
   }
}

Runnable r = Demo::sayHello;  // instead of () -> Demo.sayHello()
r.run();  // Output: Hello!
```

✅ **Type 2: Reference to an Instance Method of a Particular Object**

java

CopyEdit

```
List<String> names = Arrays.asList("Amit", "Neha", "Kiran");

Consumer<String> printer = System.out::println;  // instead of name ->
System.out.println(name)

names.forEach(printer);
```

✅ **Type 3: Reference to an Instance Method of an Arbitrary Object**

java

CopyEdit

```
Function<String, String> toUpper = String::toUpperCase;


System.out.println(toUpper.apply("java"));  // Output: JAVA
```

✅ **Type 4: Constructor Reference**

java

CopyEdit

```
Supplier<List<String>> listSupplier = ArrayList::new;


List<String> list = listSupplier.get();
list.add("Hello");
System.out.println(list);  // Output: [Hello]
```

🔷 **Constructor Reference with Custom Class**

java

CopyEdit

```
class Person {
   String name;
   Person(String name) { this.name = name; }
}


Function<String, Person> personCreator = Person::new;


Person p = personCreator.apply("Viraj");
System.out.println(p.name);  // Output: Viraj
```

✅ Benefits

Benefit	Explanation
✅ Clean code	Replaces verbose lambda with readable refs
✅ Reusability	Calls already defined methods directly
✅ Less boilerplate	Focus on logic, not syntax
✅ Encourages Functional Style	Works great with streams and lambdas

---

⚠️ Drawbacks

Drawback	Explanation
❌ Limited use	Only works when lambda just calls a method
❌ Can be confusing	For beginners, syntax might feel indirect

🧠 Common Interview Questions

1. What are Method References in Java 8?
2. Difference between lambda and method reference?
3. How many types of method references are there?
4. Can constructor references be used with custom classes?
5. Explain ClassName::methodName syntax with example.

🔍 Real-Life Analogy

Imagine you're calling your friend to ask them to say "Hi".

- **Lambda**: You tell someone: "Call John and tell him to say Hi."
- **Method Reference**: You just **directly pass John's number** and let them handle it.

✅ Summary

Type	Syntax	Use Case
Static	Class::staticMethod	Math::abs, Integer::parseInt
Instance (object)	object::method	System.out::println
Instance (class)	Class::instanceMethod	String::length
Constructor	Class::new	ArrayList::new, Person::new

✅ Topic 50: Stream API in Java 8

🔷 What is the Stream API?

The **Stream API** allows you to **process collections of data** (like List, Set) in a **declarative**, **functional-style** pipeline — instead of traditional loops.

Think of it like a **flow of data** from a source (like a List), through a **pipeline of operations**, to a **result**.

🔹 Package:

java

CopyEdit

import java.util.stream.*;

🔹 Key Characteristics of Streams

Feature	Description
Declarative	Write **what to do**, not **how to do**
Functional	Works well with **lambda expressions**
Lazy Evaluation	Operations are **not executed** until needed
Can be Parallel	Easily **parallelize** data processing

🔹 Common Stream Workflow

java

CopyEdit

collection.stream()

.filter(...)

.map(...)

.sorted(...)

.collect(...);

🔹 **Example**

java

CopyEdit

```
List<String> names = Arrays.asList("Ram", "Shyam", "Ravi", "Ramesh");


List<String> result = names.stream()
                .filter(name -> name.startsWith("R"))
                .collect(Collectors.toList());


System.out.println(result);  // Output: [Ram, Ravi, Ramesh]
```

🔸 **Stream Operations**

🔹 **1. Intermediate Operations (Chained)**

These are **lazy**, return another stream:

- filter()
- map()
- sorted()
- distinct()
- limit(), skip()

🔹 **2. Terminal Operations (Ends Stream)**

These **trigger execution**:

- collect()
- forEach()
- count()
- reduce()
- min(), max()

- anyMatch(), allMatch(), noneMatch()

🔷 Common Stream Methods with Examples

✅ filter()

java

CopyEdit

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5);
nums.stream().filter(n -> n % 2 == 0).forEach(System.out::println);  // 2 4
```

✅ map()

java

CopyEdit

```
List<String> names = Arrays.asList("Amit", "Neha");
names.stream().map(String::toUpperCase).forEach(System.out::println);  // AMIT NEHA
```

✅ sorted()

java

CopyEdit

```
List<Integer> list = Arrays.asList(3, 1, 2);
list.stream().sorted().forEach(System.out::println);  // 1 2 3
```

✅ collect()

java

CopyEdit

```
List<String> result = names.stream()
                .map(String::toLowerCase)
                .collect(Collectors.toList());
```

✅ reduce()

java

CopyEdit

```
int sum = Arrays.asList(1, 2, 3, 4).stream().reduce(0, Integer::sum);
System.out.println(sum);  // 10
```

🔹 Parallel Stream

Easily parallelize large data processing:

```
java
CopyEdit
list.parallelStream().forEach(System.out::println);
```

🔹 Stream vs Collection

Feature	Collection	Stream
Stores data	Yes	No (pipeline only)
Modifies data	Yes	No (read-only)
Eager/Lazy	Eager	Lazy
Can be reused?	Yes	No (stream can be consumed once)

✅ Benefits of Stream API

Benefit	Explanation
✅ Cleaner code	Removes complex loops and if-else
✅ Performance	Supports parallelism and lazy loading
✅ Readability	Functional and fluent style
✅ Chainable operations	Combine multiple filters, maps, etc.

⚠️ Drawbacks

Drawback	Explanation
❌ One-time use	Once consumed, can't reuse stream
❌ Debugging is tricky	Harder to debug chained operations
❌ Might overcomplicate	For small tasks, a loop is simpler

🧠 Common Interview Questions

1. What is the Java Stream API?
2. What are intermediate vs terminal operations?
3. How is map() different from filter()?
4. What is the use of reduce()?
5. What's the difference between Stream and Collection?
6. How can you make a stream parallel?
7. Why streams are lazy?

🔍 Real-Life Analogy

Imagine a water filter pipeline:

- **Input:** Tap water
- **Pipeline:** Filters, UV treatment
- **Output:** Clean water

Same with Stream:

- **Input:** Collection
- **Pipeline:** filter, map, etc.
- **Output:** Transformed collection or result

✅ Summary

- The Stream API simplifies data processing with a **functional and lazy** pipeline.
- You can **filter, map, sort, reduce, and collect** data with simple, readable syntax.

- It supports **parallelism**, **no mutation**, and **reusable logic**.

✅ Topic 51: Optional Class in Java 8 (Handling Nulls Gracefully)

🔷 What is Optional?

- Optional is a **container object** which may or may not contain a non-null value.
- It was introduced to **avoid NullPointerException** and encourage **explicit null checks**.

It's a better alternative to returning null from methods.

🔹 Package:

java

CopyEdit

```
import java.util.Optional;
```

🔷 Why do we use Optional?

Problem	Solution
Returning null leads to runtime NullPointerException	Use Optional.empty() or Optional.of(value)
Forgetting to check for null	Optional forces you to think: **value present or not?**

🔹 Creating Optional Instances

Method	Description
Optional.of(value)	Returns non-empty Optional, throws if value is null
Optional.ofNullable(value)	Returns Optional — empty if value is null
Optional.empty()	Returns an empty Optional

✅ Examples:

java

CopyEdit

```
Optional<String> name1 = Optional.of("Viraj");  // value present
Optional<String> name2 = Optional.ofNullable(null);  // empty
Optional<String> name3 = Optional.empty();  // explicitly empty
```

🔷 Checking Value in Optional

✅ isPresent()

java

CopyEdit

```
Optional<String> name = Optional.of("Java");
if (name.isPresent()) {
   System.out.println(name.get());  // Java
}
```

✅ ifPresent()

java

CopyEdit

```
name.ifPresent(n -> System.out.println("Name: " + n));  // Name: Java
```

🔷 Providing Default Values

✅ orElse()

java

CopyEdit

```
String result = name.orElse("Default Name");
```

✅ orElseGet()

java

CopyEdit

```
String result = name.orElseGet(() -> "From Supplier");
```

🔷 **Throw Exception if Empty**

java

CopyEdit

```
String result = name.orElseThrow(() -> new RuntimeException("Value not present"));
```

🔷 **Transforming Optional Value**

✅ **map() and flatMap()**

java

CopyEdit

```
Optional<String> name = Optional.of("java");

Optional<String> upperName = name.map(String::toUpperCase);

System.out.println(upperName.get());  // JAVA
```

✅ **Real-Life Analogy**

Imagine Optional like a **gift box**:

- Sometimes there's a gift inside (value)
- Sometimes it’s empty (no value)
- You **must check before using it**

🔹 **Real-World Use Case**

Traditional Way (Risky)

java

CopyEdit

```
public String getName(User user) {
   return user.getName();  // might throw NullPointerException
}
```

With Optional

java

CopyEdit

```
public Optional<String> getName(User user) {
   return Optional.ofNullable(user.getName());
}
```

✅ **Benefits of Optional**

Benefit	Description
✅ Avoids null pointer exception	Enforces explicit null handling
✅ Readable code	Self-documenting: return Optional instead of null
✅ Functional style	Works with map(), filter(), etc.
✅ Safer APIs	Reduces bugs by design

⚠️ **Drawbacks**

Drawback	Description
❌ Slightly more verbose	Adds wrapping/unwrapping logic
❌ Shouldn't be used for fields	Best used for method return types, not fields
❌ Overuse leads to complexity	Use wisely — not for everything

---

🧠 **Common Interview Questions**

1. What is the Optional class in Java?
2. How is Optional better than null?
3. Difference between orElse() and orElseGet()?
4. When should we use Optional.empty()?
5. Can you use Optional as a method parameter?

🔍 Summary Table

Method	Purpose
of()	Create Optional with value
ofNullable()	Create Optional that can be empty
empty()	Explicitly empty Optional
get()	Retrieve value (unsafe)
isPresent()	Check if value is present
ifPresent()	Perform action if present
orElse()	Return default value
orElseGet()	Get default from supplier
orElseThrow()	Throw exception if empty
map()	Transform value

✅ Topic 52: Java 8 Date and Time API (java.time)

🔷 Why Was It Introduced?

Before Java 8, date/time handling used classes like Date, Calendar, and SimpleDateFormat, which were:

- **Mutable** (not thread-safe)
- **Confusing** (inconsistent methods)
- **Not ISO compliant**

Java 8 introduced the java.time package:
✅ **Immutable**, ✅ **Thread-safe**, ✅ **ISO-8601 compliant**, ✅ **More readable & powerful**

🔹 Common Classes in java.time

Class	Purpose
LocalDate	Date only (yyyy-MM-dd)
LocalTime	Time only (HH:mm:ss)
LocalDateTime	Date + Time (no timezone)
ZonedDateTime	Date + Time + Timezone
Period	Date-based amount of time (years, months, days)
Duration	Time-based amount (hours, seconds)
DateTimeFormatter	Format/Parse date & time

🔸 Examples

✅ LocalDate

java

CopyEdit

```
LocalDate date = LocalDate.now();  // Current date
```

LocalDate specificDate = LocalDate.of(2023, 10, 5);

System.out.println(date); // 2025-06-12

✅ **LocalTime**

java

CopyEdit

LocalTime time = LocalTime.now(); // Current time

LocalTime specificTime = LocalTime.of(10, 30);

System.out.println(time); // 12:38:22.123

✅ **LocalDateTime**

java

CopyEdit

LocalDateTime dt = LocalDateTime.now();

System.out.println(dt); // 2025-06-12T12:38:22

✅ **Formatting Dates (DateTimeFormatter)**

java

CopyEdit

LocalDate date = LocalDate.now();

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy");

String formattedDate = date.format(formatter);

System.out.println(formattedDate); // 12-06-2025

✅ **Period (Date Difference)**

java

CopyEdit

LocalDate start = LocalDate.of(2022, 1, 1);

```
LocalDate end = LocalDate.now();
Period period = Period.between(start, end);
System.out.println(period.getYears() + " years");
```

✅ Duration (Time Difference)

java

CopyEdit

```
LocalTime t1 = LocalTime.of(10, 0);
LocalTime t2 = LocalTime.of(12, 30);
Duration d = Duration.between(t1, t2);
System.out.println(d.toMinutes());  // 150
```

✅ Benefits of Java 8 Date/Time API

Benefit	Explanation
✅ Immutable	Safe for multi-threading
✅ Easy to use	Clean, chainable APIs
✅ Accurate	Correct handling of leap years, DST
✅ Standard formatting	ISO-8601 by default

⚠️ Drawbacks

Drawback	Description
❌ More verbose	More classes and imports needed
❌ Legacy compatibility	Not directly compatible with Date and Calendar

You can still convert:

java

CopyEdit

```
Date oldDate = new Date();

Instant instant = oldDate.toInstant();

LocalDateTime newDate = instant.atZone(ZoneId.systemDefault()).toLocalDateTime();
```

🧠 Common Interview Questions

1. What are the main classes in java.time?
2. Difference between Period and Duration?
3. How is LocalDate different from Date?
4. Is LocalDateTime thread-safe?
5. How to format and parse dates in Java 8?

🔍 Real-Life Use Case

🗓️ Suppose you're building a leave management system:

- Use Period.between(start, end) to calculate days of leave
- Use LocalDate.now() to get current date
- Format output as "dd-MM-yyyy" using DateTimeFormatter

✅ That’s all you need to master **Date and Time API in Java 8**!

✅ Topic 53: Functional Interfaces & Lambda Expressions

🔷 What is a Functional Interface?

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

💡 Think of it as a contract for a single action.

✅ Declaring a Functional Interface

java

CopyEdit

```
@FunctionalInterface
interface MyFunction {
   void execute();  // Only one abstract method allowed
}
```

✅ @FunctionalInterface annotation is **optional**, but:

- **Helps the compiler** detect mistakes (like adding extra abstract methods)
- **Improves readability** for developers

🔷 Common Predefined Functional Interfaces (in java.util.function)

Interface	Method	Description
Function<T, R>	R apply(T)	Takes one argument, returns result
Predicate<T>	boolean test(T)	Tests condition (true/false)
Consumer<T>	void accept(T)	Performs action on object
Supplier<T>	T get()	Supplies/returns object, no input

🔷 Why Functional Interfaces?

Reason	Explanation
✅ Enable lambda expressions	You can pass behavior, not just data
✅ Support for functional programming	Clean, expressive, and concise
✅ Power tools like Stream API	All stream operations are functional

---

🔷 What is a Lambda Expression?

A **lambda expression** is a shorthand way to implement a functional interface using an anonymous function.

java

CopyEdit

(parameter) -> { logic }

✅ Example 1: Without Lambda

java

CopyEdit

```
MyFunction m = new MyFunction() {
   public void execute() {
      System.out.println("Run without lambda");
   }
};
m.execute();
```

✅ Example 2: With Lambda

java

CopyEdit

```
MyFunction m = () -> System.out.println("Run with lambda");
m.execute();  // Output: Run with lambda
```

✅ Using Java's Built-in Functional Interfaces

Predicate<T>:

java

CopyEdit

```
Predicate<Integer> isEven = x -> x % 2 == 0;

System.out.println(isEven.test(4));  // true
```

Function<T, R>:

java

CopyEdit

```
Function<String, Integer> length = s -> s.length();

System.out.println(length.apply("Java"));  // 4
```

Consumer<T>:

java

CopyEdit

```
Consumer<String> printer = s -> System.out.println(s);

printer.accept("Hello");  // Hello
```

Supplier<T>:

java

CopyEdit

```
Supplier<String> supplier = () -> "Java";

System.out.println(supplier.get());  // Java
```

✅ Real-Life Analogy

🛠 Imagine you want a machine (method) to do some **custom action** like printing, filtering, or transforming.
Instead of writing a new class, you just **pass a behavior** using a lambda.

🧠 Benefits

Benefit	Explanation
✅ Less code	No need for anonymous classes
✅ More readable	Express intent directly
✅ Reusability	Easily pass logic as argument
✅ Stream-friendly	Works with Stream, forEach, etc.

⚠️ Drawbacks

Drawback	Description
❌ Less familiar syntax	Can be confusing for new devs
❌ One-method rule	Functional interfaces must be single-method
❌ Limited debugging	Harder to debug inside lambda

🔍 Real-World Use Case

🎯 Suppose you want to filter a list of employees older than 30:

java

CopyEdit

```
List<Employee> list = ...;


list.stream()
  .filter(e -> e.getAge() > 30)
  .forEach(e -> System.out.println(e.getName()));
```

✅ filter() uses Predicate<T>, forEach() uses Consumer<T>

🧠 Common Interview Questions

1. What is a functional interface in Java?
2. Can an interface with default methods be a functional interface?

3. What is the syntax of a lambda expression?
4. Difference between Predicate and Function?
5. Can we create our own functional interface?

Summary Table

Concept	Example
Functional Interface	@FunctionalInterface with one abstract method
Lambda Expression	(x) -> x * x
Predicate	x -> x > 10
Consumer	x -> System.out.println(x)
Supplier	() -> "Hello"
Function	x -> x.length()

✅ Topic 54: Wrapper Classes, Autoboxing & Unboxing

🔷 What Are Wrapper Classes?

In Java, **primitive types** like int, char, double are **not objects**.
But Java is an **object-oriented language**, and many frameworks (e.g., Collections API) work **only with objects**.

To solve this, Java provides **Wrapper Classes** — they "wrap" primitive types into objects.

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

✅ Example:

java

CopyEdit

```
int a = 10;

Integer obj = Integer.valueOf(a);  // wrapping int into Integer

System.out.println(obj);  // 10
```

🔷 Why Do We Use Wrapper Classes?

Use Case	Reason
✅ Collections	Only store objects, not primitives
✅ Utilities	Methods like parseInt(), compare() etc.
✅ Null handling	Primitives can't be null, wrappers can
✅ Conversions	Easily convert between types

🔷 Autoboxing & Unboxing

Java 5 introduced:

- ✅ **Autoboxing** = Automatic conversion **primitive → wrapper**
- ✅ **Unboxing** = Automatic conversion **wrapper → primitive**

✅ Autoboxing Example:

java

CopyEdit

```
int x = 20;
Integer obj = x;  // Automatically converts to Integer (autoboxing)
```

✅ Unboxing Example:

java

CopyEdit

```
Integer obj = 30;
int y = obj;  // Automatically converts to int (unboxing)
```

🔍 Behind the Scenes:

java

CopyEdit

```
Integer obj = Integer.valueOf(x);  // Autoboxing
int y = obj.intValue();          // Unboxing
```

🔷 Real-Life Use Case

📋 Example: Using a List of numbers

```
java
CopyEdit
List<Integer> list = new ArrayList<>();
list.add(10);  // Autoboxing (int → Integer)
int x = list.get(0);  // Unboxing (Integer → int)
```

🧠 Benefits

Benefit	Explanation
✅ Object features	Wrappers support methods (e.g., compareTo(), parseX())
✅ Null support	Useful in databases and APIs where null is valid
✅ Auto conversions	Reduces boilerplate with autoboxing/unboxing

---

⚠️ Drawbacks

Drawback	Description
❌ Performance hit	Wrappers are heavier than primitives (more memory/CPU)
❌ Null pointer risk	Accessing null wrapper objects can throw NullPointerException
❌ Confusing behavior	== and .equals() behave differently

❗ Example of Confusing Behavior:

```
java
```

CopyEdit

```
Integer a = 127;
Integer b = 127;
System.out.println(a == b);  // true (cached range -128 to 127)

Integer x = 128;
Integer y = 128;
System.out.println(x == y);  // false (outside cache range)

System.out.println(x.equals(y));  // true
```

🧠 Common Interview Questions

1. What is the purpose of wrapper classes in Java?
2. Difference between autoboxing and manual boxing?
3. Why does Integer a = 128; Integer b = 128; a == b return false?
4. What are the drawbacks of using wrappers over primitives?
5. Can a wrapper class be null? Can a primitive?

🔚 Summary Table

Feature	Explanation
Wrapper Class	Object representation of primitive type
Autoboxing	Primitive → Wrapper (automatic)
Unboxing	Wrapper → Primitive (automatic)
Collections	Require wrapper types
Null support	Only wrapper types can be null

✅ Topic 55: Java Modifiers, Keywords, and Cheatsheet

🔷 What are Java Modifiers?

Modifiers in Java are **keywords** used to define the **scope, behavior, and accessibility** of classes, methods, variables, etc.

Java has two types:

Type	Example Keywords
✅ Access Modifiers	public, private, protected, *default*
✅ Non-Access Modifiers	static, final, abstract, synchronized, etc.

🔹 Access Modifiers (Control visibility)

Modifier	Same Class	Same Package	Subclass (other pkg)	Other Pkg
public	✅	✅	✅	✅
protected	✅	✅	✅	❌
default	✅	✅	❌	❌
private	✅	❌	❌	❌

✅ Example:

java

CopyEdit

```
public class Student {
   private int id;         // Accessible only within class
   protected String name;    // Accessible within package + subclasses
   public int age;         // Accessible everywhere
}
```

🔹 **Non-Access Modifiers (Define behavior)**

Modifier	Used With	Purpose
static	Variable, Method	Belongs to class, not object
final	Class, Method, Var	Prevent changes
abstract	Class, Method	Incomplete, must be extended
synchronized	Method, Block	Thread-safe execution
volatile	Variable	Threads always read latest value
transient	Variable	Ignored during serialization
strictfp	Class, Method	Ensures FP accuracy across platforms
native	Method	Calls code written in C/C++

✅ **Examples:**

java

CopyEdit

```
public static void main(String[] args) { }  // Static method


final int x = 10;  // Cannot be changed


abstract class Animal {
   abstract void sound();  // Must be overridden
}


synchronized void update() {
   // only one thread can enter here
}
```

🔷 Java Keywords

Java has **50+ reserved keywords**. You cannot use them as identifiers (variable, class names).

✅ Common Java Keywords:

java

CopyEdit

class, public, private, static, final, void, this, super,

extends, implements, return, new, try, catch, throw, throws,

interface, abstract, package, import, if, else, switch, break,

continue, while, for, do, instanceof, enum, synchronized, native

---

🔷 Real-Life Analogy

- private: Like a personal diary — only you can read.
- public: Like a website — anyone can access it.
- final: Like a constant value (e.g., your birth year — can't change).
- abstract: Like a blueprint — not usable directly until built.

🧠 Common Interview Questions

1. What is the difference between public, protected, private, and default?
2. What is the use of final keyword in Java?
3. Why do we use static methods in Java?
4. Can we override final methods?
5. What does transient mean in serialization?
6. Difference between abstract class and interface?
7. What is the purpose of volatile keyword?
8. What is the role of synchronized in multithreading?

🔍 Mini Cheatsheet (For Quick Revision)

Modifier	Meaning
public	Accessible everywhere
private	Accessible only in class
protected	Accessible in same package + subclasses
default	Accessible in same package
static	Belongs to class, not object
final	Value/method/class can’t change
abstract	Must be implemented by subclass
synchronized	Thread-safe
volatile	Avoid caching by threads
transient	Skip in serialization

✅ **Real-Life Use Case Example:**

Imagine you're building a class BankAccount:

java

CopyEdit

```
public class BankAccount {
   private int balance;        // Not accessible outside
   public final String accountNo; // Can’t be changed after creation
   static int totalAccounts;     // Shared by all objects
}
```

🔚 **Summary**

- Java modifiers define how members are **accessed and behave**.
- You’ll use them **daily**: public static void main, private int id, etc.
- Interviews often test **visibility rules** and **differences** (e.g., final vs abstract).

✅ Topic 56: Exception Handling in Java

🔷 What is an Exception?

An **exception** is an **unwanted or unexpected event** that occurs during the execution of a program and disrupts the normal flow of instructions.

🧠 Think of it like an error situation — e.g., dividing by zero, accessing a null object, file not found, etc.

🔷 Why Exception Handling?

Without handling exceptions:

- Your application **crashes**
- You get **ugly error messages**
- You can't **control** error behavior

🔷 Java Exception Hierarchy

scss

CopyEdit

Throwable

├── Error (serious system errors, unrecoverable)

└── Exception

  ├── Checked Exceptions (compile-time)

  └── Unchecked Exceptions (runtime)

✅ Types of Exceptions

Type	Examples	When?
Checked	IOException, SQLException	Compile-time

Type	Examples	When?
Unchecked	NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException	Runtime

---

🔷 **Syntax of Exception Handling in Java**

java

CopyEdit

```
try {
   // risky code
} catch (ExceptionType e) {
   // handling code
} finally {
   // cleanup code (always executes)
}
```

✅ **Example:**

java

CopyEdit

```
public class Example {
   public static void main(String[] args) {
      try {
         int a = 10 / 0;  // risky
      } catch (ArithmeticException e) {
         System.out.println("Cannot divide by zero!");
      } finally {
         System.out.println("Always executed");
      }
   }
```

}

🔷 Keywords in Exception Handling

Keyword	Purpose
try	Defines a block of code to test for errors
catch	Handles the exception
finally	Code that always runs (e.g., closing files)
throw	Used to explicitly throw an exception
throws	Declares exceptions in method signature

✅ throw Example

java

CopyEdit

```
throw new ArithmeticException("Manual throw");
```

✅ throws Example

java

CopyEdit

```
void readFile() throws IOException {
   // code that might throw IOException
}
```

🔷 Real-Life Analogy

Imagine you're using an ATM:

- **Exception** = You enter a wrong PIN.
- **try block** = You try to withdraw.
- **catch block** = Machine shows “Wrong PIN”.

- **finally block** = Card is returned to you.

🧠 Benefits of Exception Handling

Benefit	Description
✅ Prevents app crash	Keeps program running smoothly
✅ Helps in debugging	Stack trace shows where error occurred
✅ Separates error logic	Clean separation from business logic
✅ Improves code quality	More robust and professional

⚠️ Drawbacks (if misused)

Drawback	Description
❌ Overusing exceptions	May hide logic bugs
❌ Generic catch blocks	Catching Exception hides specific error
❌ No finally or cleanup	May leave resources open (memory leak, file handle)

🔍 Best Practices

✅ Catch specific exceptions
✅ Always close resources (use try-with-resources)
✅ Avoid catching Throwable or Error
✅ Don’t ignore exceptions (empty catch)
✅ Use finally or try-with-resources for cleanup

✅ try-with-resources (Java 7+)

java

CopyEdit

```
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
   String line = br.readLine();
```

```
} catch (IOException e) {
   e.printStackTrace();
}
```

- No need for finally to close the resource!

🧠 Common Interview Questions

1. What is the difference between checked and unchecked exceptions?
2. What is the use of finally block?
3. Can a finally block be skipped?
4. Difference between throw and throws?
5. What happens if an exception is not caught?
6. What are best practices in exception handling?
7. What is try-with-resources?
8. Can we have multiple catch blocks?

🔚 Summary Table

Concept	Summary
Exception	Unexpected event during runtime
Checked Exception	Caught at compile-time
Unchecked Exception	Caught at runtime
try-catch-finally	Structure to handle exceptions
throw / throws	For explicitly creating or declaring exceptions
Best Practice	Catch specific, clean up resources, don't ignore

✅ Topic 57: Multithreading & Concurrency in Java

🔷 What is Multithreading?

Multithreading is the ability of a CPU (or a single core in a multi-core processor) to execute **multiple threads simultaneously**.

- Each thread is a **lightweight process** within a program.
- Java supports multithreading via the Thread class and Runnable interface.

🧠 Why use Multithreading?

Reason	Explanation
✅ **Faster execution**	Multiple tasks run in parallel
✅ **Resource sharing**	Threads share memory
✅ **Better CPU utilization**	Ideal for multi-core processors
✅ **Responsive UIs**	Prevents freezing in GUI apps (e.g., Swing, Android)

🔷 Difference: Process vs Thread

Feature	Process	Thread
Memory	Separate memory	Shared memory
Speed	Slower	Faster
Overhead	High	Low
Communication	Complex	Easy

🔹 How to Create a Thread in Java

Java provides two main ways:

✅ 1. Extending Thread class

java

CopyEdit

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

public class Test {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start(); // starts new thread, calls run()
    }
}
```

✅ **2. Implementing Runnable interface**

java

CopyEdit

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

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

}

✅ **Best Practice:** Prefer Runnable over extending Thread (more flexible).

🔹 **Important Thread Methods**

Method	Purpose
start()	Starts the thread (calls run())
run()	Code to execute
sleep(ms)	Pauses thread
join()	Waits for another thread to finish
yield()	Hints that other threads can run
setPriority()	Set thread priority (1 to 10)

✅ **Example with sleep():**

java

CopyEdit

```
public class SleepExample extends Thread {
  public void run() {
    for (int i = 1; i <= 3; i++) {
      System.out.println(i);
      try {
        Thread.sleep(1000);  // pause 1 second
      } catch (InterruptedException e) {
        System.out.println(e);
      }
    }
  }
}
```

🔷 Concurrency vs Parallelism

Term	Meaning
Concurrency	Tasks start, run, and complete in overlapping time (not always simultaneously)
Parallelism	Tasks literally run at the same time (multi-core CPU)

🔷 Thread Lifecycle

sql

CopyEdit

New → Runnable → Running → Blocked/Waiting → Dead

🔹 Thread Synchronization

🔸 Problem: When two threads access shared data simultaneously, it may cause **race conditions** or data corruption.

🔹 Solution: Use synchronized keyword.

✅ Synchronized Method

java

CopyEdit

```
public synchronized void increment() {
   count++;
}
```

✅ Synchronized Block

java

CopyEdit

```
synchronized(this) {
   // critical section
}
```

🔹 **Volatile Keyword**

- Tells threads to always read the latest value of a variable from **main memory**, not cache.

java

CopyEdit

volatile boolean flag = true;

🔹 **Deadlock in Threads**

When two threads wait for each other to release a lock and **none proceeds**, it's called **deadlock**.

✅ **Example:**

java

CopyEdit

Thread-1: Lock A → Waiting for Lock B

Thread-2: Lock B → Waiting for Lock A

➡️ Avoid deadlocks by using **lock ordering** or **tryLock()**.

🔹 **Java Concurrency API (Advanced)**

Java provides high-level tools in java.util.concurrent package:

Tool	**Description**
ExecutorService	Thread pool manager
Callable	Like Runnable but returns value
Future	Used to get result of Callable
CountDownLatch	Wait for threads to finish
Semaphore	Controls access to a resource
ReentrantLock	Advanced thread lock

✅ Real-Life Analogy

- Threads are like **chefs** in a kitchen.
- If 3 chefs (threads) work on 3 dishes (tasks) at the same time → faster results.
- If two chefs fight over 1 knife (shared resource) → need synchronization!

🧠 Common Interview Questions

1. Difference between Thread and Runnable?
2. What is multithreading?
3. What are thread lifecycle states?
4. How is thread synchronization achieved?
5. What is the use of volatile keyword?
6. How to prevent race conditions?
7. What causes a deadlock? How to avoid it?
8. Difference between synchronized method and block?
9. What is ExecutorService?

🔚 Summary Table

Concept	Explanation
Thread	Lightweight unit of execution
Multithreading	Run multiple threads for better performance
Synchronization	To avoid race conditions
Volatile	Ensures visibility among threads
Deadlock	Two threads block each other
ExecutorService	Manages thread pools