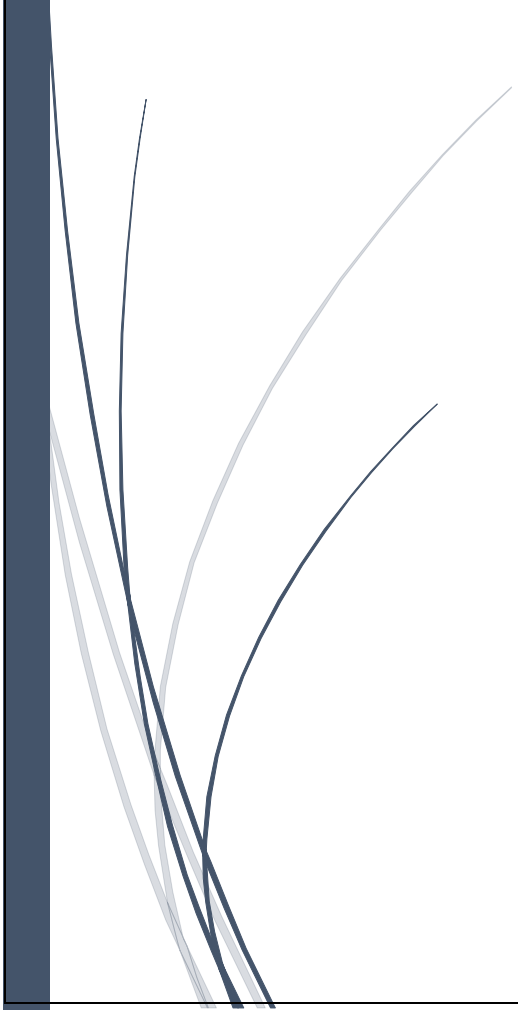
Java Full Portion

Basic - Advanced



Subbiah Pillai P

1. What is Java?

Java is a **programming language** used to build:

- Mobile apps (like Android apps),
- Websites (like Amazon, IRCTC),
- Desktop software (like Calculator),
- Games,
- ATM software, etc.

🔹 **Features of Java:**

- **Simple** – Easy to learn and use
- **Object-Oriented** – Based on real-world objects (we will learn soon)
- **Platform-Independent** – You write code once, it can run on Windows, Linux, Mac, etc.
- **Secure** – It has many built-in security features
- **Robust** – Java can handle errors well and avoids crashing

💡 **Real World Example:**

Imagine you're writing a book. If you write it in English, any English reader can read it. Same way, Java code (bytecode) can be read by any computer using JVM.

2. What is JVM (Java Virtual Machine)?

JVM is the **engine** that runs Java programs.

- It takes the .class file (bytecode)
- Converts it into **machine code**
- Runs the program on your system (Windows, Mac, Linux)

🔸 **Why JVM is Special:**

- JVM is different for Windows, Linux, etc.
- But it understands the same bytecode
- That's why Java is **platform-independent**

💡 **Example**:

Like a translator who reads one language (bytecode) and speaks in your local language (machine code).

3. What is Bytecode?

Bytecode is a **special code** created by Java compiler.

🔸 **How is it created?**

1. You write a Java file → MyProgram.java
2. You compile it → creates MyProgram.class (this is bytecode)

- .java → is your code
- .class → is the bytecode

🔸 **JVM reads .class file and runs it**

💡 Think like this:

- You write a recipe in English (.java)
- You translate it into a cooking script (.class)
- The kitchen (JVM) reads that cooking script and makes food (runs your code)

4. What is JIT (Just-In-Time Compiler)?

JIT is a small part inside JVM. It helps your program run **faster**.

- While JVM is running the program,
- JIT converts bytecode into machine code quickly
- It stores it in memory so the code doesn't need to be translated again

💡 Think like this:

If you repeat a task many times, JIT helps JVM to **remember it and reuse**, which saves time.

5.What is JRE (Java Runtime Environment)?

JRE is needed to **run Java programs**.

🔸 **JRE Contains:**

- ✅ JVM (Java Virtual Machine)
- ✅ Library Files (pre-written Java code)

💡 Example:

If your friend made a Java game, and you just want to **play** it, you only need **JRE**. You don’t need JDK if you're not writing code.

6. What is JDK (Java Development Kit)?

JDK is a full setup to **create** and **run** Java programs.

🔸 **JDK Contains:**

- ✅ JRE (Java Runtime Environment)
- ✅ Compiler (javac)
- ✅ Tools (for debugging, documentation, etc.)

🔸 **What you can do with JDK:**

- Write Java programs
- Compile them (turn your code into machine language)
- Run them

💡 Example:
You install **JDK** if you're a **Java developer**.

Complete (Real-Time Flow):

1. You write a Java file: Hello.java
2. You compile it: javac Hello.java → creates Hello.class
3. JVM takes Hello.class
4. JIT helps run it faster
5. You see the result on screen!

🔚 Summary Chart:

Term	Meaning	Use	Part of
Java	Programming language	Create apps	—
JDK	Java Development Kit	Create + Run code	Contains JRE + Compiler
JRE	Java Runtime Environment	Run code	Contains JVM
JVM	Java Virtual Machine	Runs bytecode	Inside JRE
JIT	Just-In-Time Compiler	Runs faster	Inside JVM
Bytecode	Compiled Java code (.class)	JVM reads it	Output of compilation

Java Syntax

Syntax means **rules** you must follow while writing Java code.

✅ **Basic Java Syntax Rules:**

- Java code must be inside a **class**
- Code runs from the **main method**
- Every statement ends with **semicolon ;**
- Curly brackets { } are used to group code
- Java is **case-sensitive** (Example: Main and main are not the same)

What is a Data Type?

- **Data type** means:
  ➡️ What type of value you are storing in a variable.
  Example:
- int age = 25;
- Here, int is the **data type** — it tells Java to store a **whole number**.

Types of Data Types in Java

- Java has **two main types** of data types:

  1. Primitive Data Types

  2. Non-Primitive Data Types

  Primitive Data Types (8 types)

  These are **basic** and **fixed-size** data types in Java.

Data Type	Size	Example	Used For
`byte`	1 byte	`100`	Small numbers
`short`	2 bytes	`32000`	Medium numbers
`int`	4 bytes	`100000`	Big whole numbers
`long`	8 bytes	`1234567890L`	Very large numbers
`float`	4 bytes	`3.14f`	Decimal numbers
`double`	8 bytes	`45.9876`	More accurate decimals
`char`	2 bytes	`'A'`, `'5'`, `'@'`	Single character
`boolean`	1 bit	`true`, `false`	Yes/No conditions

Non-Primitive Data Types

These are more advanced. Not fixed-size. We can create or customize them.

Type	Example
`String`	`"Hello"`
Arrays	`{1, 2, 3}`
Classes	`Student, Book, Car`
Interfaces	Like contracts for classes
Objects	Real things created from class

Variables in Java

A **variable** is a **container** used to store values.

```
int age = 25;                    // age is variable
float price = 99.99f;            // price is variable
char grade = 'A';                // grade is variable
boolean isPassed = true;         // isPassed is variable
String name = "Subbiah";         // name is variable
```

Rules for Variable Names:

- Must start with a letter (A–Z or a–z), or (_ or $)
- Cannot start with a number
- Cannot use Java keywords like int, class as names

Operators:

Operators are special symbols in Java that perform operations on variables and values. They are fundamental to programming logic.

1. **Arithmetic Operators**
2. **Relational Operators**
3. **Logical Operators**

4. **Assignment Operators**
5. **Unary Operators**
6. **Bitwise Operators**
7. **Ternary Operators**

✅ 1. Arithmetic Operators

Used for performing basic mathematical operations.

Operator	Description	Example	Result
`+`	Addition	`10 + 5`	15
`-`	Subtraction	`10 - 5`	5
`*`	Multiplication	`10 * 5`	50
`/`	Division	`10 / 2`	5
`%`	Modulus (Remainder)	`10 % 3`	1

🧪 Example:

java Copy Edit

```java
int a = 15, b = 4;
System.out.println("Addition: " + (a + b));
System.out.println("Division: " + (a / b));
System.out.println("Remainder: " + (a % b));
```

✅ 2. Relational (Comparison) Operators

Used to compare two values. Returns a boolean result (`true` or `false`).

Operator	Description	Example	Result
`==`	Equal to	`5 == 5`	true
`!=`	Not equal to	`5 != 3`	true
`>`	Greater than	`10 > 3`	true
`<`	Less than	`5 < 8`	true
`>=`	Greater or equal	`5 >= 5`	true
`<=`	Less or equal	`3 <= 4`	true

🧪 Example:

java Copy Edit

```java
int x = 7, y = 10;
System.out.println(x == y);  // false
System.out.println(x < y);   // true
```

✅ 3. Logical Operators

Used to combine multiple boolean expressions.

Operator	Description	Example	Result
`&&`	Logical AND	`true && false`	false
`		`	`	Logical OR
`!`	Logical NOT	`!true`	false

🧪 Example:

java Copy Edit

```java
int age = 25;
boolean hasLicense = true;

if (age >= 18 && hasLicense) {
    System.out.println("Eligible to drive");
}
```

✅ 4. Assignment Operators

Used to assign values to variables. Some operators combine assignment with arithmetic.

Operator	Example	Equivalent To
`=`	`a = 5`	—
`+=`	`a += 3`	`a = a + 3`
`-=`	`a -= 2`	`a = a - 2`
`*=`	`a *= 4`	`a = a * 4`
`/=`	`a /= 5`	`a = a / 5`
`%=`	`a %= 2`	`a = a % 2`

🧪 Example:

java Copy Edit

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

✅ 5. Unary Operators

These operate on a single operand.

Operator	Description	Example
`+`	Unary plus	`+a`
`-`	Unary minus	`-a`
`++`	Increment	`++a` / `a++`
`--`	Decrement	`--a` / `a--`
`!`	Logical complement	`!true`

🧪 Example:

```java
int a = 5;
System.out.println(++a); // Pre-increment: 6
System.out.println(a--); // Post-decrement: 6 (then a becomes 5)
```

✅ 6. Bitwise Operators (Advanced)

Used for bit-level operations on integers.

Operator	Description	Example
`&`	Bitwise AND	`a & b`
`	`	`	Bitwise OR
`^`	Bitwise XOR	`a ^ b`
`~`	Bitwise Complement	`~a`
`<<`	Left shift	`a << 2`
`>>`	Right shift	`a >> 2`

🧪 Example:

```java
int a = 5;  // 0101
int b = 3;  // 0011
System.out.println(a & b);  // 0001 = 1
```

✅ 7. Ternary Operator

A shorthand for `if-else`.

Syntax:

java Copy Edit

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

🧪 Example:

java Copy Edit

```
int marks = 75;
String result = (marks >= 50) ? "Pass" : "Fail";
System.out.println(result); // Output: Pass
```

✅ Typecasting

🔹 What is Typecasting?

Typecasting means **converting one data type to another**.

Java has **two types** of typecasting:

1. Implicit Typecasting (Widening)

Java automatically converts **small → big type**

java Copy Edit

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

✅ Example:

java Copy Edit

```
int a = 10;
double b = a; // implicit conversion (int → double)

System.out.println(b); // 10.0
```

2. Explicit Typecasting (Narrowing)

You manually convert **big → small type**

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

✅ **Example:**

```java
double x = 10.75;
int y = (int) x;  // manual casting: double to int

System.out.println(y); // 10 (decimal part is removed)
```

🔔 Important Notes:

- Narrowing may cause **data loss**
- Widening is **safe**

Input in Java

Java doesn't have cin or scanf like C/C++. Instead, we use:

✅ **a. Scanner (Most commonly used)**

✅ **b. BufferedReader (Faster, used when handling large input)**

🔸 Import Scanner Class:

```java
import java.util.Scanner;
```

🔸 Create Scanner Object:

```java
Scanner sc = new Scanner(System.in);
```

🔷 Common Input Methods in Scanner

Method	Description	Example Input
`nextInt()`	Reads an `int` value	`42`
`nextDouble()`	Reads a `double` value	`3.14`
`next()`	Reads a single word (`String`)	`hello`
`nextLine()`	Reads a full line (`String`)	`hello world`
`nextBoolean()`	Reads a `boolean` value (`true` / `false`)	`true`
`nextFloat()`	Reads a `float` value	`2.5f`
`nextLong()`	Reads a `long` value	`9876543210`
`nextShort()`	Reads a `short` value	`120`
`nextByte()`	Reads a `byte` value	`12`

🔶 Special: How to read a character using `Scanner`?

➡️ There is **no direct method** like `nextChar()`

But you can read a `String` and take the first character:

✅ Example:

```java
System.out.print("Enter a character: ");
char ch = sc.next().charAt(0); // reads first character of the input
System.out.println("You entered: " + ch);
```

💡 `next()` returns a word (string), and `charAt(0)` gets the first character.

🔴 Important Note: `nextLine()` trap

Problem:

java

Copy Edit

```java
int age = sc.nextInt();    // Reads number
String name = sc.nextLine(); // Tries to read string → Skips input
```

➡️ **Why?**

Because after reading `int`, the **Enter key (`\n`) is still in the input buffer**, and `nextLine()` reads it as an **empty string**.

✅ Solution:

Add an extra `sc.nextLine();` after `nextInt()`

🔁 Example with fix:

java

```java
System.out.print("Enter age: ");
int age = sc.nextInt();
sc.nextLine(); // clear the leftover newline

System.out.print("Enter name: ");
String name = sc.nextLine(); // now it works fine

System.out.println("Name: " + name + ", Age: " + age);
```

🧪 Full Example – All Types

java

```java
import java.util.Scanner;

public class ScannerDemo {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // int input
        System.out.print("Enter age: ");
        int age = sc.nextInt();

        sc.nextLine(); // clear newline

        // String input (full line)
        System.out.print("Enter full name: ");
        String name = sc.nextLine();

        // char input
        System.out.print("Enter gender (M/F): ");
        char gender = sc.next().charAt(0);

        // double input
        System.out.print("Enter weight: ");
        double weight = sc.nextDouble();

        System.out.println("\n--- Output ---");
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Gender: " + gender);
        System.out.println("Weight: " + weight);
    }
}
```

🔍 Sample Input:

mathematica

```
Enter age: 25
Enter full name: Subbiah Pillai
Enter gender (M/F): M
Enter weight: 70.5
```

✅ b. BufferedReader

Faster than `Scanner`, but a bit more complex. Needs `IOException` handling.

🔹 Import:

```java
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
```

🔹 Create Object:

```java
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

🔹 Reading Input:

BufferedReader only reads **String**, so you need to convert it if needed:

```java
String str = br.readLine(); // reads full line
int num = Integer.parseInt(str); // convert to int
```

🧪 Example:

```java
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class BufferedReaderExample {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        System.out.print("Enter your city: ");
        String city = br.readLine();

        System.out.print("Enter your pin code: ");
        int pin = Integer.parseInt(br.readLine());

        System.out.println("City: " + city + ", Pincode: " + pin);
    }
}
```

🔷 Scanner vs BufferedReader in Java

Feature	`Scanner`	`BufferedReader`
📦 Package	`java.util`	`java.io`
🧠 Simplicity	Very easy to use	Slightly more complex
💡 Input Methods	Built-in methods to read `int`, `double`, etc.	Only reads `String`; must convert manually
🚀 Performance	Slower for large inputs	Faster (more efficient)
📋 Use Case	Beginners, small apps	Competitive programming, large inputs
🔄 Parsing	Built-in (`nextInt()`, `nextDouble()`)	Must use `Integer.parseInt()`, etc.
⚠️ Exceptions	No need to handle checked exceptions	Requires `IOException` handling
🧵 Thread Safety	Not thread-safe	Thread-safe

✅ Comments in Java

🔸 What is a Comment?

A **comment** is a line of text in code that **Java will ignore** during execution.

📌 Used to:

- Explain what the code is doing
- Add notes/reminders
- Disable code temporarily (for debugging)

✅ Comments:

- `//` – single-line
- `/* */` – multi-line
- `/** */` – documentation (for APIs)

✅ Java Naming Conventions

Java has **rules + best practices** for naming classes, variables, methods, etc.

📌 **Why use naming conventions?**

- Improves readability
- Makes code look clean and standard
- Required for large team projects and interview

🔷 Java Naming Guidelines Table:

Type	Convention	Example
Class	PascalCase	`EmployeeData`, `StudentRecord`
Interface	PascalCase	`Serializable`, `Runnable`
Method	camelCase	`getSalary()`, `printReport()`
Variable	camelCase	`employeeName`, `totalAmount`
Constant	UPPER_SNAKE_CASE	`MAX_VALUE`, `PI`
Package	lowercase (dot format)	`com.wipro.project`

✅ 1. `if`, `else if`, `else` – Full Explanation

🔹 Syntax:

java

```java
if (condition) {
    // Executes if condition is true
} else if (anotherCondition) {
    // Executes if anotherCondition is true
} else {
    // Executes if no condition is true
}
```

🔹 Flow Diagram:

```lua
          condition1?
          /     \
       true     false
      /            \
 execute          condition2?
                    /    \
                 true    false
                 /         \
           execute      else block
```

✅ Example 1: Basic if-else

```java
int age = 20;

if (age >= 18) {
    System.out.println("Eligible to vote");
} else {
    System.out.println("Not eligible");
}
```

✅ Example 2: if-else-if ladder

```java
int marks = 85;

if (marks >= 90) {
    System.out.println("Grade: A");
} else if (marks >= 80) {
    System.out.println("Grade: B");
} else if (marks >= 70) {
    System.out.println("Grade: C");
} else {
    System.out.println("Grade: D");
}
```

✅ Example 3: Nested if (if inside if)

```java
int age = 25;
boolean hasLicense = true;

if (age >= 18) {
    if (hasLicense) {
        System.out.println("You can drive");
    } else {
        System.out.println("Get a license first");
    }
} else {
    System.out.println("Too young to drive");
}
```

✅ 2. `switch` Statement – Full Explanation

🔹 Use Case:

When you have **many conditions based on the same variable**, use `switch`.

🔹 Syntax:

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

✅ Example:

```java
int day = 3;

switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    default:
        System.out.println("Invalid day");
}
```

✅ Mini Calculator Using switch in Java

```
import java.util.Scanner;

public class MiniCalculator {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);

      // Input first number
      System.out.print("Enter first number: ");
      double num1 = sc.nextDouble();

      // Input second number
      System.out.print("Enter second number: ");
      double num2 = sc.nextDouble();
```

```
// Input operator
System.out.print("Choose operation (+, -, *, /, %): ");
char operator = sc.next().charAt(0);

double result;

// Switch based on operator
switch (operator) {
   case '+':
      result = num1 + num2;
      System.out.println("Result: " + result);
      break;
   case '-':
      result = num1 - num2;
      System.out.println("Result: " + result);
      break;
   case '*':
      result = num1 * num2;
      System.out.println("Result: " + result);
      break;
   case '/':
      if (num2 == 0) {
         System.out.println("Error: Division by zero");
      } else {
         result = num1 / num2;
         System.out.println("Result: " + result);
      }
      break;
   case '%':
```

```
            if (num2 == 0) {
                System.out.println("Error: Modulus by zero");
            } else {
                result = num1 % num2;
                System.out.println("Result: " + result);
            }
            break;
        default:
            System.out.println("Invalid operator!");
    }

    sc.close(); // good practice to close Scanner
  }
}
```

🧪 Sample Run:

```sql
Enter first number: 10
Enter second number: 2
Choose operation (+, -, *, /, %): *
Result: 20.0
```

✅ Supported types in `switch`:

Java allows:

Data Type	Supported?
`int`	✅
`byte`, `short`, `char`	✅
`String`	✅ (since Java 7)
`enum`	✅
`long`, `float`, `double`, `boolean`	❌ not supported

✅ Example: Switch with String

```java
String fruit = "Mango";

switch (fruit) {
    case "Apple":
        System.out.println("Apple is red");
        break;
    case "Mango":
        System.out.println("Mango is yellow");
        break;
    default:
        System.out.println("Unknown fruit");
}
```

🔷 Loops in Java

Loops help you run a block of code multiple times.

🔑 **Three types of loops:**

1. **for loop**
2. **while loop**
3. **do-while loop**

✅ 1. for Loop

🔹 **Definition:**

The for loop is used when number of iterations is known.

🔹 **Syntax:**

```
for (initialization; condition; update)
{
   // block of code
}
```

Flow:

1. **Initialize variable**
2. **Check condition**
3. **Run block**
4. **Update (increment/decrement)**
5. **Repeat until condition is false**

✅ Example:

```java
public class ForExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Hello " + i);
        }
    }
}
```

🔹 Output:

```nginx
Hello 1
Hello 2
Hello 3
Hello 4
Hello 5
```

✅ 2. while Loop

🔹 **Definition:**

The while loop is used when you **don't know how many times** the loop should run — only when a **condition is true**.

🔹 **Syntax:**

```
while (condition) {
   // block of code
}
```

🔹 **Flow:**

1. Check condition
2. If true → run block
3. Go back and check again
4. Stops when condition is false

✅ Example:

```java
public class WhileExample {
    public static void main(String[] args) {
        int i = 1;
        while (i <= 5) {
            System.out.println("Hi " + i);
            i++;
        }
    }
}
```

🔹 Output:

```nginx
Hi 1
Hi 2
Hi 3
Hi 4
Hi 5
```

✅ 3. do-while Loop

🔹 **Definition:**

do-while is like while — but it **runs the block at least once**, even if the condition is false.

🔹 **Syntax:**

```
do {
   // block of code
} while (condition);
```

✅ Example:

```java
public class DoWhileExample {
    public static void main(String[] args) {
        int i = 1;
        do {
            System.out.println("Welcome " + i);
            i++;
        } while (i <= 5);
    }
}
```

🔹 Output:

```nginx
Welcome 1
Welcome 2
Welcome 3
Welcome 4
Welcome 5
```

❗ Difference between `while` and `do-while`

Feature	`while`	`do-while`
Condition check	Before loop body	After loop body
Minimum execution	0 times (if false)	1 time (always runs once)
Use case	When pre-check is needed	When at least 1 run is needed

🔷 break, continue, and return in Java

These are used to change the normal flow of execution inside loops or methods.

✅ 1. break

🔹 **Definition:**

break is used to exit from a loop (or switch) immediately, even if the loop condition is still true.

🔹 **Usage:**

- **To stop a loop early**
- **To exit a switch case**

✅ Example: `break` in a loop

```java
for (int i = 1; i <= 10; i++) {
    if (i == 5) {
        break;  // stop loop when i == 5
    }
    System.out.println(i);
}
```

🔹 Output:

```
1
2
3
4
```

➡️ Loop stops at `i == 5`

✅ 2. continue

🔹 **Definition:**

continue skips the current iteration of the loop and jumps to the next iteration.

🔹 **Usage:**

- When you want to skip some values but not break the loop

✅ Example: `continue` in a loop

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

🔹 Output:

```
1
2
4
5
```

➡️ `3` is skipped (not printed), but loop continues.

✅ 3. return

🔹 **Definition:**

return is used to exit from a method and optionally return a value to the caller.

🔹 **Usage:**

- To stop execution inside a method
- To send a result to the place where the method was called

✅ Example 1: `return` with no value (void method)

```java
public static void greet(String name) {
    if (name == null) {
        return; // exit early if name is null
    }
    System.out.println("Hello, " + name);
}
```

✅ Example 2: `return` with value

```java
public static int add(int a, int b) {
    return a + b; // returns result to caller
}

public static void main(String[] args) {
    int sum = add(10, 5);
    System.out.println("Sum: " + sum);
}
```

🔹 **Output:**

```makefile
Sum: 15
```

Object-Oriented Programming (OOP) in Java:

✅ 1. What is a Class?

🔸 **Definition:**

A class is like a blueprint or template that defines the structure and behaviour of objects.

It contains:

- **Fields (variables) — data**
- **Methods (functions) — actions**

✅ **Example:**

```java
public class Car {
    // Fields / properties
    String brand;
    int speed;

    // Method / behavior
    void drive() {
        System.out.println(brand + " is driving at " + speed + " km/h");
    }
}
```

This class `Car` defines what every car **has** (brand, speed) and what it **does** (drive).

✅ 2. What is an Object?

🔸 **Definition:**

An **object** is a real entity **created from a class**.

You can create **many objects** from the same class.

🔹 **Example:**

```java
Car myCar = new Car();
```

Now `myCar` is an object of the class `Car`.

🔷 Key Terms

Term	Meaning
Class	Template or design for objects
Object	Instance of a class
Field	Variable inside a class
Method	Function inside a class
new	Keyword to create object

Constructor in Java

🔸 **Definition:**

A constructor is a special method that runs automatically when you create an object using new.

🔹 **Purpose:**

- Initialize objects (set default values)
- Looks like a method, but name = class name
- No return type (not even void)

✅ Syntax:

```java
class ClassName {
    ClassName() {
        // constructor code
    }
}
```

✅ Example 1: Default Constructor

```java
public class Student {
    Student() {  // constructor
        System.out.println("Student object created!");
    }

    public static void main(String[] args) {
        Student s1 = new Student(); // constructor will run
    }
}
```

🔹 Output:

```csharp
Student object created!
```

✅ Example 2: Parameterized Constructor

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

    Student(String n, int a) {
        name = n;
        age = a;
    }

    void show() {
        System.out.println("Name: " + name + ", Age: " + age);
    }

    public static void main(String[] args) {
        Student s1 = new Student("Subbiah", 25);
        s1.show();
    }
}
```

🔹 Output:

```yaml
Name: Subbiah, Age: 25
```

this Keyword

🔸 **Definition:**

this refers to the current object (the object that is calling the method or constructor).

🔹 **Use Cases:**

1. To refer to current class variables
2. To call current class methods
3. To call another constructor inside same class

Example 1: Resolving variable name conflict

```java
public class Student {
    String name;

    Student(String name) {
        this.name = name;  // "this.name" refers to class variable, "name" is parameter
    }

    void show() {
        System.out.println("Name: " + name);
    }

    public static void main(String[] args) {
        Student s = new Student("Ravi");  // creating object with value "Ravi"
        s.show();  // calling method to display name
    }
}
```

✅ **Output:**

```
makefile

Name: Ravi
```

🧠 **Explanation:**

- When you do Student s = new Student("Ravi");,
  the constructor is called with "Ravi" as the parameter.
- this.name = name; assigns "Ravi" to the class variable name.
- s.show(); prints "Name: Ravi".

✅ Example 2: Calling another constructor using this()

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

    // Default constructor
    Student() {
        this("Unknown", 0); // calls the parameterized constructor
    }

    // Parameterized constructor
    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }

    public static void main(String[] args) {
        Student s1 = new Student(); // uses default constructor
        s1.display(); // displays the initialized values
    }
}
```

✅ Output:

```yaml
Name: Unknown, Age: 0
```

🔁 Summary:

- this() is used to **call another constructor inside the same class**.
- Must be the **first statement** in the constructor.
- Helps to **reuse constructor logic** and avoid code duplication.

super Keyword

🔸 **Definition:**

super refers to the **parent class (superclass)**

🔹 **Use Cases:**

1. To access **parent class methods or variables**
2. To call **parent class constructor**

✅ Example 1: Access parent class variable

```java
class Animal {
    String sound = "Some sound";
}

class Dog extends Animal {
    String sound = "Bark";

    void printSound() {
        System.out.println(super.sound); // parent class variable
        System.out.println(this.sound);  // current class variable
    }
}

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

🔹 **Output:**

```rust
Some sound
Bark
```

✅ Example 2: Call parent class constructor

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

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

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

🔹 Output:

```kotlin
Animal constructor
Dog constructor
```

🧠 Summary

Keyword	Used For
`constructor`	Initializes object (same name as class)
`this`	Refers to current class object
`super`	Refers to parent class (superclass)

✅ What is Inheritance?

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

🧠 Why use inheritance?

- **Code Reusability**: You don't need to rewrite the same code in multiple classes.
- **Improved Structure**: You can organize shared code in parent classes.
- **Polymorphism**: Allows method overriding and dynamic method dispatch.

✅ 1. Single Inheritance

👉 One child inherits from one parent.

java

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

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

public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();  // from Animal
        d.bark(); // from Dog
    }
}
```

✅ Output:

nginx

```
Animal eats
Dog barks
```

✅ 2. Multi-level Inheritance

👉 A class inherits from a class which itself inherited from another class.

java

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

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

class Puppy extends Dog {
    void weep() {
        System.out.println("Puppy weeps");
    }
}
public class Main {
    public static void main(String[] args) {
        Puppy p = new Puppy();
        p.eat();   // from Animal
        p.bark();  // from Dog
        p.weep();  // own method
    }
}
```

✅ Output:

nginx

```
Animal eats
Dog barks
Puppy weeps
```

✅ 3. Hierarchical Inheritance

👉 Multiple child classes inherit from one parent class.

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

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

class Cat extends Animal {
    void meow() {
        System.out.println("Cat meows");
    }
}
public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();
        d.bark();

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

✅ Output:

```nginx
Animal eats
Dog barks
Animal eats
Cat meows
```

❌ 4. Multiple Inheritance (Not allowed with classes)

Java doesn't support **multiple inheritance with classes** to avoid **ambiguity**.

java Copy Edit

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

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

// ❌ Not allowed in Java
// class C extends A, B { }
```

✅ Instead, Java supports multiple inheritance using **interfaces** (we'll see that in abstraction).

✅ Types of Inheritance in Java:

Type	Structure	Supported in Java?
Single Inheritance	A → B	✅ Yes
Multi-level Inheritance	A → B → C	✅ Yes
Hierarchical Inheritance	A → B, A → C	✅ Yes
Multiple Inheritance (by classes)	A, B → C	❌ Not supported via classes (but supported via interfaces)

🧠 Important Notes:

1. Every class in Java **implicitly extends Object** class.
2. Constructor of parent class is **automatically called** before child constructor.
3. **Private members are not inherited.**
4. Use super to access parent methods or constructors.
5. Use @Override to override parent methods in child.

✅ What is Polymorphism?

Polymorphism means "**many forms**".

In Java, it allows:

- A single function/method name to behave differently depending on the **context** (input, object, etc.)

🔸 Types of Polymorphism in Java:

Type	Also Known As	Happens At...
Compile-time Polymorphism	Method Overloading	Compile time
Runtime Polymorphism	Method Overriding	Runtime

✅ 1. Compile-Time Polymorphism (👉 Method Overloading

🔹 What is Method Overloading?

- When **multiple methods** have the **same name**, but **different parameters** (type, number, or order) in the **same class**.
- The method call is resolved **at compile time**.

✅ Example 1: Method Overloading

java

```java
public class Calculator {

    // Method with 2 int parameters
    int add(int a, int b) {
        return a + b;
    }

    // Method with 3 int parameters
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method with 2 double parameters
    double add(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculator calc = new Calculator();
```

```
        Calculator calc = new Calculator();

        // Call method with 2 ints
        System.out.println("add(2, 3) = " + calc.add(2, 3));

        // Call method with 3 ints
        System.out.println("add(1, 2, 3) = " + calc.add(1, 2, 3));

        // Call method with 2 doubles
        System.out.println("add(2.5, 3.5) = " + calc.add(2.5, 3.5));
    }
}
```

✅ Output:

```
csharp

add(2, 3) = 5
add(1, 2, 3) = 6
add(2.5, 3.5) = 6.0
```

✅ 2. Runtime Polymorphism (👉Method Overriding)

🔹 What is Method Overriding?

- When a **child class** provides its own **version** of a method that is already present in the **parent class**.
- Method is chosen **at runtime** using the **actual object type**, not reference type.

✅ Example 2: Method Overriding

```
java

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

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

```java
// Another child class
class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("Cat meows");
    }
}

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

        // Parent reference, child object → runtime polymorphism
        Animal a1 = new Dog();
        Animal a2 = new Cat();
        Animal a3 = new Animal();

        // Method call decided at runtime
        a1.makeSound();  // Dog's version
        a2.makeSound();  // Cat's version
        a3.makeSound();  // Animal's version
    }
}
```

✅ Output:

```
css

Dog barks
Cat meows
Animal makes a sound
```

❗ Difference Between Overloading vs Overriding

Feature	Overloading	Overriding
Definition	Same method name, different params	Same method name + params
Class type	Same class	Parent → Child classes
Binding type	Compile-time	Runtime
Return type	Can be different (if signature changes)	Must be same or covariant
Annotation used	❌ Not needed	✅ `@Override` recommended

✅ What is Abstraction?

Abstraction means **hiding complex details** and **showing only the necessary parts** to the user.

- It helps to reduce complexity.
- Focus only on what an object **does**, not **how it does** it.

✅ How do we achieve abstraction in Java?

Two ways:

1. **Abstract Classes**
2. **Interfaces**

1️⃣ Abstract Class

🔸 What is an abstract class?

- A class declared with keyword abstract.
- Can have **abstract methods** (methods without body) and **concrete methods** (with body).
- Cannot create an object of abstract class directly.
- Child class **must implement** abstract methods (unless child class is abstract too).

✅ Syntax of abstract class:

```java
abstract class ClassName {
    abstract void abstractMethod();  // no method body

    void concreteMethod() {
        // method body
    }
}
```

✅ Example: Abstract Class

```java
abstract class Animal {
    // Abstract method (no body)
    abstract void sound();

    // Concrete method
    void sleep() {
        System.out.println("Animal is sleeping");
    }
}

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

Copy

```java
public class Main {
    public static void main(String[] args) {
        // Animal a = new Animal(); // ❌ Error! Can't create object of abstract class

        Dog d = new Dog();  // Create Dog object
        d.sound();          // Calls implemented method
        d.sleep();          // Calls concrete method
    }
}
```

✅ Output:

```csharp
Dog barks
Animal is sleeping
```

Copy

🔹 Explanation (in comments):

- abstract void sound(); means all subclasses must provide their own sound() implementation.
- sleep() method has a body and can be used by subclasses directly.
- You **cannot create object** of abstract class directly.
- The subclass (Dog) **implements** the abstract method sound().

2️⃣ Interface

🔸 What is an interface?

- A **100% abstract type** (before Java 8).
- Can contain **only abstract methods** (before Java 8) and **constants**.
- Java 8+ allows default and static methods in interfaces.
- A class can **implement multiple interfaces** (helps achieve multiple inheritance).

✅ Syntax of interface:

```java
interface InterfaceName {
    void method1();  // abstract by default
}
```

✅ Example: Interface

```java
interface Printable {
    void print();  // abstract method
}

class Document implements Printable {
    public void print() {
        System.out.println("Printing document...");
    }
}

public class Main {
    public static void main(String[] args) {
        Printable p = new Document();
        p.print();
    }
}
```

✅ Output:

```javascript
Printing document...
```

3️⃣ Difference between Abstract Class and Interface

Feature	Abstract Class	Interface
Methods	Abstract + Concrete allowed	Only abstract (before Java 8), default and static methods allowed after Java 8
Multiple Inheritance	No (Java does not allow)	Yes (a class can implement multiple interfaces)
Variables	Can have instance variables	Only `public static final` constants
Constructor	Can have constructor	No constructor
Inheritance Keyword	`extends`	`implements`
Usage	When classes share common behavior with some differences	When you want to specify capability/contract

🔹 Before Java 8 (i.e., Java 7 and below):

- Interfaces can **only have abstract methods**
- All methods are:
  - `public abstract` (by default)
  - **No method body**

```java
interface Printable {
    void print();  // abstract, no body
}
```

✅ Valid in Java 7 and before.

🔹 From Java 8 onward:

Now **interfaces can have methods with body**, BUT **only in two cases**:

1. Default Methods → has a body

```java
interface Printable {
    default void show() {
        System.out.println("This is a default method in interface");
    }
}
```

➡️ `default` methods are used to add new functionality to interfaces without affecting the old implementation.

❌ What is not allowed:

- **Normal methods with body** (without `default` or `static`) are NOT allowed.

```java
interface WrongExample {
    void hello() {
        System.out.println("Not allowed!"); // ❌ Compile error
    }
}
```

✅ Example: Interface with all 3 method types (Java 8+)

```java
interface Test {
    void show(); // abstract method

    default void defaultMethod() {
        System.out.println("This is default method");
    }

    static void staticMethod() {
        System.out.println("This is static method");
    }
}

class Demo implements Test {
    public void show() {
        System.out.println("Implemented abstract method");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Demo d = new Demo();
        d.show();           // calls abstract method
        d.defaultMethod();  // calls default method
        Test.staticMethod(); // calls static method via interface
    }
}
```

✅ Output:

```pgsql
Implemented abstract method
This is default method
This is static method
```

✅ What is Encapsulation?

Encapsulation is the process of **wrapping data (variables)** and **methods** together as a single unit (class), and **restricting direct access** to some components of an object.

🔑 Key Features of Encapsulation:

Feature	Description
🔐 Data hiding	Keep variables `private` so they can't be accessed directly
🧩 Controlled access	Provide public `getter` and `setter` methods to access/update
✅ Improves security	Prevents unauthorized access
🔄 Easy to modify	Internal code can be changed without affecting outside classes

✅ Real-Life Example:

Think of a **bank account**:

- You cannot access the **balance** directly.
- You must use methods like getBalance() and deposit(amount).

✅ Example: Java Encapsulation

java

```java
// Class with private data
public class Student {
    // 🔐 private variables (not accessible directly)
    private String name;
    private int age;

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

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

    // 🟢 Public getter for 'age'
    public int getAge() {
        return age;
    }
```

```
    // 🟢 Public setter for 'age'
    public void setAge(int age) {
        if (age > 0) {  // Validating before setting
            this.age = age;
        }
    }
}
```

✅ Main class to use Student:

```java
public class Main {
    public static void main(String[] args) {
        Student s = new Student();

        // 🟦 Set values using setters
        s.setName("Subbiah");
        s.setAge(25);

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

✅ Output:

```makefile
Name: Subbiah
Age: 25
```

✅ Summary:

Concept	Example
Private variables	`private String name;`
Public getters	`getName()`
Public setters	`setName(String name)`
Benefit	Protects and secures data

🔐 Access Modifiers in Java

Access modifiers **control the visibility** of:

- **Classes**
- **Methods**
- **Variables**
- **Constructors**

They decide **where a member can be accessed from** (within same class, package, or outside).

✅ Types of Access Modifiers

Java provides **4 access modifiers**:

Modifier	Within Same Class	Same Package	Subclass (Other Pkg)	Outside Pkg
`private`	✅ Yes	❌ No	❌ No	❌ No
default	✅ Yes	✅ Yes	❌ No	❌ No
`protected`	✅ Yes	✅ Yes	✅ Yes	❌ No
`public`	✅ Yes	✅ Yes	✅ Yes	✅ Yes

✅ 1. `private`

- Accessible **only within the same class**.
- ❌ Not visible outside the class, not even in a subclass.

```java
class Person {
    private String name = "Subbiah";

    private void showName() {
        System.out.println(name);
    }
}
```

➡️ You **cannot** access `name` or `showName()` outside `Person` class.

✅ 2. *default* (no modifier)

- If **no access modifier** is written, it's called **default access**.
- Accessible **within the same package only**.

```java
class Student {
    int rollNumber = 101;  // default access
    void show() {
        System.out.println("Roll No: " + rollNumber);
    }
}
```

➡️ You can access this from any class in the **same package**, but not from outside packages.

✅ 3. `protected`

- Accessible:
  - In the **same package**
  - In **subclasses** (even if in another package)

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

class Dog extends Animal {
    void bark() {
        sound(); // ✅ allowed
    }
}
```

➡️ You can use `protected` when you want to allow access to child classes.

✅ 4. `public`

- Can be accessed from **anywhere** (other class, other package, subclass, etc.)
- Class or method is **globally available**

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

➡️ You can use `Car` and `start()` method from **any package/project**.

🔢 Arrays in Java

Including:

- ✅ 1D Arrays (One-Dimensional)
- ✅ 2D Arrays (Two-Dimensional)
- ✅ Jagged Arrays (Irregular 2D arrays)

✅ What is an Array?

An **array** is a **collection of elements** (same type), stored in **consecutive memory locations**.

- Fixed size (you decide size at creation)
- Index-based (starts from 0)
- Can be 1D, 2D, or more

🔹 Syntax of Declaring Arrays:

```java
datatype[] arrayName = new datatype[size];
```

1️⃣ One-Dimensional (1D) Array

✅ Declaration + Initialization:

```java
int[] arr = new int[5];          // declare + create array of size 5
int[] nums = {10, 20, 30, 40};   // declare + initialize
```

✅ Access Elements:

```java
System.out.println(nums[0]);     // 10
```

✅ Full Example:

```java
public class Main {
    public static void main(String[] args) {
        int[] marks = {85, 90, 76, 88};

        // Loop to access elements
        for (int i = 0; i < marks.length; i++) {
            System.out.println("marks[" + i + "] = " + marks[i]);
        }
    }
}
```

✅ Output:

```
marks[0] = 85
marks[1] = 90
marks[2] = 76
marks[3] = 88
```

2️⃣ Two-Dimensional (2D) Array

A 2D array is like a **matrix** (rows and columns).

✅ Declaration + Initialization:

```java
int[][] matrix = new int[2][3]; // 2 rows, 3 columns

int[][] table = {
    {1, 2, 3},
    {4, 5, 6}
};
```

✅ Access Elements:

```java
System.out.println(table[0][2]); // 3
```

✅ Full Example:

```java
public class Main {
    public static void main(String[] args) {
        int[][] arr = {
            {10, 20, 30},
            {40, 50, 60}
        };

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

✅ Output:

```
10 20 30
40 50 60
```

3️⃣ Jagged Arrays (Irregular 2D Arrays)

A jagged array is an array of arrays **with different lengths per row**.

✅ Declaration:

```java
int[][] jagged = new int[3][];  // 3 rows, columns not defined yet

// Define each row with different length
jagged[0] = new int[2]; // 2 columns
jagged[1] = new int[4]; // 4 columns
jagged[2] = new int[3]; // 3 columns
```

✅ Assign values:

```java
jagged[0][0] = 10;
jagged[0][1] = 20;
// and so on...
```

✅ Full Example:

```java
public class Main {
    public static void main(String[] args) {
        int[][] jagged = {
            {1, 2},
            {3, 4, 5, 6},
            {7, 8, 9}
        };

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

✅ Output:

```
1 2
3 4 5 6
7 8 9
```

✅ Summary Table:

Array Type	Shape	Example
1D Array	Single row (linear)	`{10, 20, 30}`
2D Array	Grid (matrix)	`{{1,2,3}, {4,5,6}}`
Jagged Array	Uneven row lengths	`{{1,2}, {3,4,5}, {6}}`

🔁 Array Sorting and Searching

✅ Import Required Utility Class:

For sorting and searching, we use:

```java
import java.util.Arrays;
```

1️⃣ Array Sorting

🔹 Syntax:

```java
Arrays.sort(arrayName);
```

By default, it sorts in **ascending order**.

✅ Example: Sort 1D Array in Ascending

```java
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] nums = {5, 2, 8, 1, 9};

        Arrays.sort(nums);  // Ascending sort

        System.out.println("Sorted array:");
        for (int n : nums) {
            System.out.print(n + " ");
        }
    }
}
```

✅ Output:

```php
Sorted array:
1 2 5 8 9
```

✅ Sort in Descending Order (manual way):

Java's Arrays.sort() does not sort primitives in descending directly. We can:

- Use reverse loop after sort
- OR use Collections.reverseOrder() with Integer[]

```java
import java.util.Arrays;
import java.util.Collections;

public class Main {
    public static void main(String[] args) {
        Integer[] nums = {5, 2, 8, 1, 9};

        Arrays.sort(nums, Collections.reverseOrder());

        System.out.println("Descending order:");
        for (int n : nums) {
            System.out.print(n + " ");
        }
    }
}
```

✅ Output:

```css
Descending order:
9 8 5 2 1
```

2️⃣ Array Searching

✅ 1. Linear Search (Normal search)

- Works on **unsorted or sorted arrays**
- Time complexity: O(n)

✅ Example: Linear Search

java

```java
public class Main {
    public static void main(String[] args) {
        int[] nums = {4, 7, 1, 9, 5};
        int key = 9;
        boolean found = false;

        for (int i = 0; i < nums.length; i++) {      //Linear search start here
            if (nums[i] == key) {
                System.out.println("Found at index: " + i);
                found = true;
                break;
            }
        }

        if (!found) {
            System.out.println("Not found");
        }
    }
}
```

✅ Output:

pgsql

```
Found at index: 3
```

✅ 2. Binary Search (Only for sorted arrays)

- Much faster: O(log n)
- Uses Arrays.binarySearch(array, key)

✅ Example: Binary Search

```java
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] nums = {2, 4, 6, 8, 10, 12};

        int key = 8;
        int index = Arrays.binarySearch(nums, key);

        if (index >= 0) {
            System.out.println("Found at index: " + index);
        } else {
            System.out.println("Not found");
        }
    }
}
```

✅ Output:

```pgsql
Found at index: 3
```

❗ Important Note:

- Array **must be sorted** before binary search
- If not sorted → **wrong result or -1**

🔷 1. String in Java (`java.lang.String`)

✅ What is a String?

- A **sequence of characters**, like `"Hello"`.
- **Immutable** → once created, **cannot be changed**.

```java
String s = "Java"; // String literal
String s2 = new String("Java"); // using new keyword
```

✅ Example:

```java
public class Main {
    public static void main(String[] args) {
        String s1 = "Hello";
        String s2 = "Hello";

        System.out.println(s1 == s2);        // true (same memory)
        System.out.println(s1.equals(s2));   // true (same content)

        String s3 = new String("Hello");
        System.out.println(s1 == s3);        // false (different memory)
        System.out.println(s1.equals(s3));   // true
    }
}
```

✅ Output:

```arduino
true
true
false
true
```

🔧 Important String Methods:

java

```java
String str = "Java Programming";

str.length();           // 16
str.charAt(5);          // 'P'
str.toUpperCase();      // "JAVA PROGRAMMING"
str.toLowerCase();      // "java programming"
str.contains("gram");   // true
str.startsWith("Java"); // true
str.endsWith("ing");    // true
str.replace("Java", "C++"); // "C++ Programming"
str.substring(5);       // "Programming"
```

🔁 Why Strings are Immutable?

- Stored in **String Pool**
- Shared among multiple references
- Changing one string would affect all → Java avoids this

⚠️ Problem with String (slow in loops):

java

```java
String result = "";
for (int i = 0; i < 1000; i++) {
    result += i;  // ❌ creates new object every time – very slow
}
```

➡️ That's why we use **StringBuilder** or **StringBuffer**

🔷 2. StringBuilder in Java (java.lang.StringBuilder)

✅ What is StringBuilder?

- Used to **modify strings** without creating new objects.
- It is **mutable**.
- **Faster** than String in loops.
- **Not thread-safe** (means not safe for multithreaded apps).

✅ Example:

```java
public class Main {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Hello");
        sb.append(" World");  // appends string
        sb.insert(5, " Java"); // inserts at index
        sb.delete(0, 5); // deletes chars from index 0 to 4

        System.out.println(sb);  // Output: " Java World"
    }
}
```

🔧 Important StringBuilder Methods:

```java
sb.append("text");    // add at end
sb.insert(2, "X");    // insert at position
sb.delete(1, 3);      // remove chars from index 1 to 2
sb.reverse();         // reverse the string
sb.replace(0, 2, "Hi"); // replace chars from 0-1
```

🔷 3. StringBuffer in Java (java.lang.StringBuffer)

- Same as **StringBuilder**
- But it is **thread-safe**
- Slightly **slower** than StringBuilder

✅ Example:

```java
public class Main {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hi");
        sb.append(" Everyone");
        System.out.println(sb);  // Hi Everyone
    }
}
```

✅ Summary Table

Feature	String	StringBuilder	StringBuffer
Mutability	❌ Immutable	✅ Mutable	✅ Mutable
Thread-Safe	❌ No	❌ No	✅ Yes
Speed	❌ Slow	✅ Fast	⚠️ Slower than SB
Package	`java.lang`	`java.lang`	`java.lang`
Used For	Constants	Fast text building	Safe multithreading

🧠 Stack vs Heap Memory in Java

Java has two main memory areas used during program execution:

Feature	🧱 Stack Memory	🗃️ Heap Memory
🧩 What is it?	Temporary memory for method calls	Memory for storing objects
🧮 Used For	Local variables, method calls	All objects and their instance variables
🔁 Lifespan	Short-term (method-based)	Long-term (until object is garbage collected)
🧵 Thread Safety	Yes (Each thread has its own)	No (Shared among all threads)
📌 Memory Allocation	Automatically managed (LIFO)	Needs garbage collection
🧹 Cleanup	Auto when method exits	Done by **Garbage Collector**
📌 Location	Inside RAM	Inside RAM
🔥 Common Error	`StackOverflowError`	`OutOfMemoryError`

✅ Diagram (Visual Representation)

less

```
Main Memory (RAM)
├── Stack (LIFO)
│   ├── Method 1: local variables
│   ├── Method 2: local variables
│   └── ...
│
└── Heap
    ├── Object A (e.g., new Student())
    ├── Object B (e.g., new String())
    └── ...
```

✅ Code Example to Understand Stack vs Heap

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

public class Main {
    public static void main(String[] args) {
        int a = 10;                     // 🧱 stored in Stack
        Student s = new Student();      // 📦 object created in Heap
        s.id = 101;
        s.name = "Subbiah";

        System.out.println(s.name);     // Stack has reference to Heap object
    }
}
```

📌 Breakdown:

Variable	Stored In	Why?
`a = 10`	Stack	Local primitive
`s` (reference)	Stack	Reference variable
`new Student()`	Heap	Object is always in heap
`s.id, s.name`	Heap	Inside object → lives in heap

🔄 Lifecycle of Stack & Heap

🧱 Stack:

- Follows **LIFO (Last-In-First-Out)** structure
- Each method call → new stack frame
- When method ends → frame removed automatically

🗃️ Heap:

- Objects remain in memory until:
  - No reference is pointing to them
  - Then **Garbage Collector** removes it

🔥 Errors:

1. `StackOverflowError`

Occurs when too many nested method calls (e.g., infinite recursion):

```java
public void call() {
    call(); // Stack will overflow
}
```

2. `OutOfMemoryError`

Occurs when too many objects are created, or heap is full:

```java
while (true) {
    new Student(); // keeps creating without clearing
}
```

💡 Real-Life Analogy:

- **Stack** → Chef's to-do list. Takes one task, completes, and removes it.
- **Heap** → Kitchen storage. All ingredients and utensils (objects) are kept and shared.

✅ What is Garbage Collection?

Garbage Collection (GC) in Java is the process of **automatically freeing memory** by **removing unused objects** from the **Heap**.

👉 Java takes care of memory management for you!

🧠 Why is it needed?

In languages like C/C++, the programmer must manually release memory using free() or delete.

But in Java:

- When an object is **no longer used** (no references),
- It becomes **eligible for garbage collection**
- JVM's **Garbage Collector** will delete it to free memory.

📦 Example:

```java
Student s = new Student();  // s refers to object in heap
s = null;                   // now object has no reference → eligible for GC
```

✅ When does GC happen?

- **Automatically** by the JVM
- You **can request**, but **cannot force** GC

```java
System.gc(); // suggests JVM to run GC
```

🧪 Example Program:

```java
class Student {
    String name;

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

    @Override
    protected void finalize() {
        System.out.println(name + " object is garbage collected");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("Subbiah");
        Student s2 = new Student("Arun");

        s1 = null;  // s1 no longer points to Subbiah
        s2 = null;  // s2 no longer points to Arun

        System.gc();  // request JVM for garbage collection
    }
}
```

✅ Output (may vary):

```csharp
Subbiah object is garbage collected
Arun object is garbage collected
```

📝 *Note: finalize() method is not guaranteed to run immediately. It's called by JVM before object is collected.*

🔍 Key Points for Interview:

Concept	Description
GC is automatic	Handled by JVM, not programmer
Only heap memory	GC works in heap, not stack
Unreferenced object	Eligible for GC
`System.gc()`	Suggest GC (not guarantee)
`finalize()` **method**	Called before object removal (deprecated in Java 9+)

📌 Example of Unreferenced Objects:

1. Nullify reference:

```java
Student s = new Student("X");
s = null; // eligible
```

2. Reassign reference:

```java
Student s1 = new Student("Y");
Student s2 = new Student("Z");
s1 = s2; // now old object Y is unreferenced
```

🧠 Real-life Analogy:

- Heap = Warehouse
- Objects = Boxes
- Stack references = Labels
- If label is removed → box becomes untracked → garbage collector removes it

🔹 finalize() Method

✅ What is it?

- A **special method** in Java called by the **Garbage Collector before** destroying an object.
- Used to **clean up resources** (like closing files, DB connections) — but not recommended in modern Java (deprecated in Java 9).

🧪 Example:

```java
class Student {
    String name;

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

    @Override
    protected void finalize() throws Throwable {
        System.out.println(name + " is being garbage collected");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("Subbiah");
        s1 = null;

        System.gc();  // Suggests GC
    }
}
```

✅ Output (may vary):

```csharp
Subbiah is being garbage collected
```

⚠️ Note:

- You **cannot force** when GC will run.
- `finalize()` is **deprecated** → use **try-with-resources** or `Cleaner`.

🔹 2. `static` Keyword

✅ Purpose:

The `static` keyword means:

- **Belongs to the class**, not to any object.
- Can be accessed **without creating an object**.

✅ You can use `static` for:

Used With	Meaning
Variable	Shared across all objects (only one copy)
Method	Called without object
Block	Runs once when class is loaded
Class (nested)	Acts like a utility class inside outer class

🧪 Example:

```java
class Student {
    int id;
    static String college = "IIT"; // shared by all

    Student(int id) {
        this.id = id;
    }

    static void showCollege() {
        System.out.println("College: " + college);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student(101);
        Student s2 = new Student(102);

        Student.showCollege(); // ✅ no need to use object
    }
}
```

✅ Output:

```makefile
College: IIT
```

🔹 3. `final` Keyword

✅ Purpose:

The `final` keyword is used to **make something unchangeable**.

Used With	Meaning
Variable	Value **cannot be changed** after assign
Method	Cannot be **overridden** in subclass
Class	Cannot be **inherited** (no subclassing)

🧪 Example 1: Final Variable

```java
public class Main {
    public static void main(String[] args) {
        final int speed = 90;
        // speed = 100; ❌ Error: cannot assign again

        System.out.println(speed);
    }
}
```

✅ Output:

```
90
```

🧪 Example 2: Final Method

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

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

🧪 Example 3: Final Class

```java
final class Vehicle {
    void type() {
        System.out.println("Vehicle type");
    }
}

// class Car extends Vehicle {} ❌ Error: Cannot inherit final class
```

🔁 Summary Table

Keyword	Used With	Meaning
`finalize()`	Object	Called before GC removes object
`static`	Variable/Method	Belongs to class, not object
`final`	Var/Method/Class	Makes it constant / non-overridable / sealed

✅ Exception Handling in Java

💡 What is an Exception?

An **exception** is a **runtime error** that can stop your program if not handled.

Examples:

- Dividing by zero
- Accessing invalid array index
- Using null reference

❗ Types of Exceptions

Type	Example	Checked/Unchecked
`ArithmeticException`	Divide by zero	Unchecked
`ArrayIndexOutOfBoundsException`	Invalid index	Unchecked
`NullPointerException`	Accessing null object	Unchecked
`IOException`, `SQLException`	File/DB error	Checked

🔐 Why Exception Handling?

To **handle errors gracefully** and keep program running.

Instead of crashing:

```java
try {
    // risky code
} catch (Exception e) {
    // handle error
} finally {
    // always runs
}
```

✅ Example: `try-catch-finally`

```java
public class Main {
    public static void main(String[] args) {
        try {
            int a = 10, b = 0;
            int c = a / b;  // ❌ This causes ArithmeticException
            System.out.println("Result: " + c); // skipped
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero: " + e);
        } finally {
            System.out.println("Finally block always runs.");
        }

        System.out.println("Program continues...");
    }
}
```

✅ Output:

```vbnet
Cannot divide by zero: java.lang.ArithmeticException: / by zero
Finally block always runs.
Program continues...
```

🔁 Explanation of Each Block:

- **try**
  - Put **risky code** here
  - Like division, file opening, etc.
- **catch**
  - Handles the **exception**
  - You can print message, log it, or do alternative logic
- **finally**
  - Always runs
  - Used to **close resources** (file, DB, etc.)
  - Even if exception occurs or not

📌 Example 2: No exception occurs

```java
public class Main {
    public static void main(String[] args) {
        try {
            int a = 10, b = 2;
            System.out.println("Result: " + (a / b));  // ✅ no error
        } catch (ArithmeticException e) {
            System.out.println("Error: " + e);
        } finally {
            System.out.println("Finished");
        }
    }
}
```

✅ Output:

```makefile
Result: 5
Finished
```

❗ Common Interview Questions

1. Can we have multiple catch blocks?

✅ Yes:

```java
try {
    // code
} catch (ArithmeticException e) {
    // handle divide
} catch (ArrayIndexOutOfBoundsException e) {
    // handle index
}
```

2. Can we write `try` without `catch`?

✅ Yes, but you must use `finally`:

```java
try {
   // code
} finally {
   // clean up
}
```

---

3. Can we write `catch` without `try`?

❌ No. `catch` must come after `try`.

---

4. What happens if exception not caught?

Java will:

- Print stack trace
- Terminate the program

🔥 Summary

Keyword	Meaning
`try`	Write risky code here
`catch`	Handle the exception
`finally`	Always executes (clean-up logic)
`throw`	Manually throw exception
`throws`	Declare exception in method

🎯 throw vs throws in Java

They both deal with **exceptions**, but they are **completely different** in usage and purpose.

Feature	`throw`	`throws`
🔹 What is it?	Used to **actually throw** an exception	Used to **declare** that a method might throw an exception
🔹 Where used?	Inside a method	In method **signature**
🔹 Followed by?	Exception **object**	Exception **class name**
🔹 How many?	Only **one object**	Can declare **multiple exceptions**
🔹 Purpose	To **create and throw** exception	To **declare and pass** exception responsibility
🔹 Example	`throw new ArithmeticException();`	`void myMethod() throws IOException`
🔹 Type	Runtime or Checked exception	Usually for **checked** exceptions

✅ 1. `throw` – to actually throw an exception

Used to **manually throw** an exception from your code.

🔧 Syntax:

```java
throw new ExceptionType("message");
```

🧪 Example:

```java
public class Main {
    public static void main(String[] args) {
        int age = 17;

        if (age < 18) {
            throw new ArithmeticException("Not eligible to vote");  // 👉 THROWS EXCEPTION NOW
        }

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

✅ Output:

```pgsql
Exception in thread "main" java.lang.ArithmeticException: Not eligible to vote
```

✅ 2. `throws` – to declare that a method might throw

Used in the **method header** to say:

> "This method might throw an exception, caller must handle it."

🔧 Syntax:

```java
void myMethod() throws IOException, SQLException {
    // risky code
}
```

🧪 Example with `throws`:

```java
import java.io.*;

public class Main {

    // 👇 Method says: I may throw IOException
    public static void readFile() throws IOException {
        FileReader fr = new FileReader("test.txt");
        fr.read();
        fr.close();
    }

    public static void main(String[] args) {
        try {
            readFile();  // 👉 You must handle the exception here
        } catch (IOException e) {
            System.out.println("File not found or error reading.");
        }
    }
}
```

✅ Real-World Difference Summary

Situation	Use `throw`	Use `throws`
You want to manually cause error	✅ `throw`	❌
A method might cause error (like file not found)	❌	✅ `throws IOException`
You catch inside the same method	✅ Throw inside try/catch	❌
You pass error to calling method	❌	✅ Use `throws`

🔹 1. ✅ Checked vs Unchecked Exceptions

✅ Checked Exceptions (Compile-Time)

Feature	Description
Type	**Checked** by compiler
When it occurs	**Compile time**
Must handle (try/throws)?	✅ Yes – must handle using `try-catch` or `throws`
Examples	`IOException`, `SQLException`, `FileNotFoundException`

🔧 Example:

```java
import java.io.*;

public class Main {
    public static void main(String[] args) {
        try {
            FileReader fr = new FileReader("abc.txt"); // ❗ Checked exception
        } catch (FileNotFoundException e) {
            System.out.println("File not found");
        }
    }
}
```

🧠 *If you don't use try-catch or throws, compiler gives an error.*

❌ Unchecked Exceptions (Runtime)

Feature	Description
Type	**Not checked** by compiler
When it occurs	**Runtime only**
Must handle (try/throws)?	❌ Not mandatory
Examples	`NullPointerException`, `ArithmeticException`, `ArrayIndexOutOfBoundsException`

🔧 Example:

```java
public class Main {
    public static void main(String[] args) {
        int a = 10 / 0; // ❗ Unchecked exception at runtime
    }
}
```

🧠 *Code compiles fine, but crashes during execution.*

🔸 2. Custom Exceptions in Java

When **built-in exceptions** are not enough, we can create our **own exception classes**.

✅ How to Create Custom Exception?

1. **Extend the Exception class** → for Checked exception
2. **Extend RuntimeException** → for Unchecked exception

🛠️ Example 1: Custom Checked Exception

java

```java
// Step 1: Create custom exception class
class AgeException extends Exception {
    AgeException(String message) {
        super(message);
    }
}

// Step 2: Use it
public class Main {
    static void checkAge(int age) throws AgeException {
        if (age < 18) {
            throw new AgeException("Age must be 18 or above");
        } else {
            System.out.println("Valid age");
        }
    }
    public static void main(String[] args) {
        try {
            checkAge(16);
        } catch (AgeException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

✅ Output:

javascript

```
Error: Age must be 18 or above
```

🛠️ Example 2: Custom Unchecked Exception

```java
class InvalidPinException extends RuntimeException {
    InvalidPinException(String message) {
        super(message);
    }
}

public class ATM {
    public static void main(String[] args) {
        int pin = 123;
        if (pin != 456) {
            throw new InvalidPinException("Invalid PIN entered");
        }

        System.out.println("Access granted");
    }
}
```

✅ Output:

```cpp
Exception in thread "main" InvalidPinException: Invalid PIN entered
```

🔁 Summary

Concept	Custom Exception Type	Extends
Checked Custom Exception	Must handle with try	`Exception`
Unchecked Custom	Runtime exception	`RuntimeException`
Can we give message?	✅ Yes, using constructor + `super()`	

✅ Wrapper Classes in Java

➡️ Integer, Double, Character, etc.
➡️ **Autoboxing & Unboxing**

🔷 What are Wrapper Classes?

Java is **object-oriented**, but **primitive types** (`int` , `double` , `char` , etc.) are **not objects**.

👉 Wrapper classes **wrap** primitive types into **objects**.

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

🔧 Example: Without Wrapper Class

```java
int a = 10;
```

✅ `a` is a **primitive**, cannot be used in collections like `ArrayList` .

🔧 Example: With Wrapper Class

```java
Integer a = Integer.valueOf(10);
```

✅ Now `a` is an **object**, and can be stored in an `ArrayList` , passed to functions, etc.

🔁 Why Use Wrapper Classes?

Reason	Benefit
Needed for **Collections**	Collections store only objects
Needed for **Object methods**	`.equals()`, `.compareTo()`, etc.
Conversion from **String to number**	`Integer.parseInt("123")`
Enable **null** values	`Integer x = null;` is valid

✅ Autoboxing & Unboxing

🔷 What is Autoboxing?

👉 **Automatically converts** primitive → object (wrapper class)

```java
int a = 10;
Integer b = a;  // autoboxing: primitive to object
```

🔷 What is Unboxing?

👉 **Automatically converts** object → primitive

```java
Integer x = 100;
int y = x;     // unboxing: object to primitive
```

🧪 Full Example:

java

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        // Autoboxing
        int a = 10;
        Integer obj = a; // primitive → object
        System.out.println("Autoboxed Integer: " + obj);

        // Unboxing
        Integer b = 20;
        int val = b; // object → primitive
        System.out.println("Unboxed int: " + val);

        // Using in collection
        ArrayList<Integer> list = new ArrayList<>();
        list.add(5);      // autoboxing
        list.add(15);     // autoboxing
        list.add(25);     // autoboxing

        System.out.println("ArrayList: " + list);
    }
}
```

✅ Output:

yaml

```yaml
Autoboxed Integer: 10
Unboxed int: 20
ArrayList: [5, 15, 25]
```

✅ Important Wrapper Methods

Method	Example	Output
`Integer.parseInt("123")`	converts String → int	`123`
`Integer.toString(123)`	int → String	`"123"`
`Double.parseDouble("10.5")`	String → double	`10.5`
`Character.isDigit('5')`	check if char is digit	`true`
`Boolean.parseBoolean("true")`	String → boolean	`true`

🔥 Interview Question

🧠 Can you store `int` in an `ArrayList<int>`?

❌ No → Use `ArrayList<Integer>`

✅ Java automatically does **autoboxing**

✅ Summary

Concept	Description
Wrapper Class	Object version of primitive type
Autoboxing	Primitive → Object (done automatically)
Unboxing	Object → Primitive
Needed For	Collections, null values, method arguments

✅ Java Collections Framework

✅ 1. What is Java Collections Framework?

Java Collections Framework (JCF) is a set of interfaces and classes that provide powerful ways to store, retrieve, and manipulate groups of objects (collections).

🔧 Why Use It?

- No need to write your own data structure logic
- Provides reusable, efficient implementations (List, Set, Map, Queue)
- Helps you organize, search, sort, and manage data easily

✅ 2. `Collection` vs `Collections`

Feature	`Collection`	`Collections` (with 's')
Type	Interface	Utility Class
Package	`java.util.Collection`	`java.util.Collections`
Purpose	Root interface for List, Set, Queue	Helper class for operations like `sort()`, `reverse()` etc.
Inheritance	Extended by List, Set, Queue	Not extended, used directly
Example	`List<String> list = new ArrayList<>();`	`Collections.sort(list);`

✅ 3. List Interface (Ordered, Allows Duplicates)

The **List** interface:

- Allows **duplicate elements**
- Maintains **insertion order**
- Supports **index-based access** (e.g., `.get(0)`)

```java
List<String> list = new ArrayList<>();
list.add("Apple");
list.add("Banana");
System.out.println(list.get(0)); // Apple
```

🔹 3.1 ArrayList

- Backed by a **dynamic array**
- Fast for **searching** (random access)
- Slow for **inserting/deleting** in middle

```java
List<String> list = new ArrayList<>();
list.add("A");
list.add("B");
System.out.println(list); // [A, B]
```

🔹 3.2 LinkedList

- **Doubly linked list**
- Good for **inserting/deleting** in the middle
- Slower for random access

```java
List<String> list = new LinkedList<>();
list.add("X");
list.add("Y");
System.out.println(list); // [X, Y]
```

🔹 3.3 Vector (Legacy)

- Synchronized (thread-safe)
- Slower than ArrayList in single-threaded programs

```java
Vector<String> v = new Vector<>();
v.add("Java");
v.add("C++");
System.out.println(v); // [Java, C++]
```

🔹 3.4 Stack (extends Vector)

- LIFO (Last In First Out)
- Uses `push()`, `pop()`, `peek()` methods

```java
Stack<String> stack = new Stack<>();
stack.push("A");
stack.push("B");
System.out.println(stack.pop()); // B
```

✅ Quick Comparison of List Types:

Feature	ArrayList	LinkedList	Vector	Stack
Ordered?	✅ Yes	✅ Yes	✅ Yes	✅ Yes
Duplicates?	✅ Yes	✅ Yes	✅ Yes	✅ Yes
Thread-safe?	❌ No	❌ No	✅ Yes	✅ Yes
Best for	Search	Insert/Delete	Thread Safety	LIFO ops

✅ 1. What is Set?

- Set is a child interface of Collection
- Used to store **unique elements only**
- **No indexing**, so no .get(index) like in List

✅ 2. HashSet

- **Doesn't maintain order**
- **Allows one null**
- **Fastest performance**
- Based on **Hashing**

```java
Set<String> hs = new HashSet<>();
hs.add("B");
hs.add("A");
hs.add("C");
hs.add("A"); // duplicate – won't be added
System.out.println(hs);  // Output may be [A, B, C] or [C, A, B]
```

✅ 3. LinkedHashSet

- **Maintains insertion order**
- Allows one null
- Slower than HashSet (due to linked list inside)

```java
Set<String> lhs = new LinkedHashSet<>();
lhs.add("B");
lhs.add("A");
lhs.add("C");
System.out.println(lhs); // Output: [B, A, C]
```

✅ 4. TreeSet

- Stores elements in **sorted (ascending)** order
- **Does NOT allow null**
- Slowest among the three
- Internally uses **Red-Black Tree**

```java
Set<String> ts = new TreeSet<>();
ts.add("Banana");
ts.add("Apple");
ts.add("Cherry");
System.out.println(ts);  // Output: [Apple, Banana, Cherry]
```

✅ 5. Comparison Table

Feature	HashSet	LinkedHashSet	TreeSet
Order Maintained	❌ No	✅ Yes	✅ Sorted order
Duplicates	❌ Not allowed	❌ Not allowed	❌ Not allowed
Null Allowed	✅ Yes (one)	✅ Yes (one)	❌ No null
Performance	✅ Fastest	⚠️ Moderate	❌ Slow (sorted)
Internal Structure	Hash Table	Hash + LinkedList	Tree (RB Tree)

✅ What is Map?

A Map stores data in **key-value pairs** where:

- Keys must be **unique**
- Values can be **duplicated**
- You use get(key) to fetch values

Map is **not a child of Collection**.

🔹 1. HashMap

- ✅ Fastest Map type
- ❌ No order maintained
- ✅ Allows **1 null key**, many null values
- Backed by **Hash Table**

```java
Map<Integer, String> map = new HashMap<>();
map.put(101, "Java");
map.put(102, "Python");
map.put(null, "C++");
System.out.println(map);
```

📄 Output: Order not guaranteed

```
{null=C++, 101=Java, 102=Python}
```

🔹 2. LinkedHashMap

- ✅ Maintains **insertion order**
- ✅ Allows **1 null key**
- 🔁 Slightly slower than `HashMap`

```java
Map<Integer, String> map = new LinkedHashMap<>();
map.put(101, "Apple");
map.put(102, "Banana");
System.out.println(map);
```

📄 Output: `{101=Apple, 102=Banana}`

🔹 3. TreeMap

- ✅ Maintains keys in **sorted (ascending)** order
- ❌ Does **not allow null key** (throws `NullPointerException`)
- ✅ Allows null values
- Internally uses **Red-Black Tree**

```java
Map<Integer, String> map = new TreeMap<>();
map.put(105, "Zebra");
map.put(103, "Lion");
map.put(101, "Ant");
System.out.println(map);
```

📄 Output: `{101=Ant, 103=Lion, 105=Zebra}`

✅ Comparison Table

Feature	HashMap	LinkedHashMap	TreeMap
Order	❌ No	✅ Insertion Order	✅ Sorted Keys
Null key	✅ One allowed	✅ One allowed	❌ Not allowed
Null values	✅ Allowed	✅ Allowed	✅ Allowed
Performance	✅ Fastest	⚠️ Slightly slower	❌ Slowest
Internal structure	Hash Table	Hash + LinkedList	Red-Black Tree

✅ How to Loop Map Entries

```java
for (Map.Entry<Integer, String> entry : map.entrySet()) {
    System.out.println(entry.getKey() + " = " + entry.getValue());
}
```

Or using lambda:

```java
map.forEach((k, v) -> System.out.println(k + ": " + v));
```

🔷 What is Queue?

- A Queue is a **collection that stores elements in order**, typically **First In First Out (FIFO)**.
- It is part of the **Java Collections Framework**.
- Defined in java.util.Queue interface.

✅ Common Operations in a Queue:

Method	Description
`add(e)`	Adds element, throws error if queue is full
`offer(e)`	Adds element, returns false if full (safe)
`remove()`	Removes head, throws error if empty
`poll()`	Removes head, returns null if empty
`element()`	Returns head, throws error if empty
`peek()`	Returns head, returns null if empty

> 🔸 Prefer `offer()`, `poll()`, and `peek()` for safe operations

🔷 Subinterfaces & Implementations:

```kotlin
Queue (interface)
  |
  +-- Deque (interface for double-ended queue)
        |
        +-- ArrayDeque
  |
  +-- PriorityQueue
```

✅ 1. `PriorityQueue`

- Stores elements based on **natural ordering** or **custom comparator**
- **Not FIFO**
- **Duplicates allowed**
- **Null not allowed**
- **Not thread-safe**

```java
Queue<Integer> pq = new PriorityQueue<>();
pq.add(30);
pq.add(10);
pq.add(20);
System.out.println(pq);  // Sorted: [10, 30, 20] (internal view)
System.out.println(pq.poll());  // 10 (lowest priority removed)
```

✅ 2. `ArrayDeque`

- **Double-ended queue** → can insert/remove from both ends
- **Faster** than `LinkedList` for queue operations
- **No nulls allowed**
- Allows **stack** and **queue** behavior

```java
Deque<String> dq = new ArrayDeque<>();
dq.add("A");
dq.addLast("B");
dq.addFirst("Z");

System.out.println(dq);     // [Z, A, B]
System.out.println(dq.pollFirst()); // Z
System.out.println(dq.pollLast());  // B
```

🔁 Queue vs Stack

Feature	Queue	Stack
Type	FIFO	LIFO
Java class	Queue / Deque	Stack
Method used	`add`, `poll`	`push`, `pop`

✅ Summary Table

Feature	PriorityQueue	ArrayDeque
FIFO	❌ No (priority)	✅ Yes
Ordered	✅ Sorted	✅ Insertion order
Null allowed?	❌ No	❌ No
Thread-safe?	❌ No	❌ No
Allows Duplicates?	✅ Yes	✅ Yes
Allows stack ops?	❌ No	✅ Yes

✅ Additional Info

✅ 1. What is this line doing?

```java
Collection<String> fruits = new ArrayList<>();
```

This means:

- You are **declaring a reference** of **type** `Collection`
- And **creating an object** of type `ArrayList`
- The object stores `String` **type elements**

👉 So, you're writing code to the **interface** (`Collection`) instead of the class (`ArrayList`)

✅ 2. Why use interface type (`Collection`)?

This is a **best practice** in Java called **"Program to Interface"**, which means:

🟢 You use:

```java
Collection<String> list = new ArrayList<>();
```

🔴 Not:

```java
ArrayList<String> list = new ArrayList<>();
```

✅ Advantage:

- You can **easily switch** to another implementation like `LinkedList`, `Vector`, etc., without changing the rest of your code.

```java
Collection<String> fruits = new LinkedList<>();
```

✅ 3. Can you use `get()` with this?

Even though the object is an `ArrayList`, the **reference type is** `Collection`, and the `Collection` interface **does not have** `get(index)` **method**.

So this will give **compile error**:

```java
Collection<String> fruits = new ArrayList<>();
fruits.add("Apple");
System.out.println(fruits.get(0)); // ❌ Compile Error
```

> 🔴 **Why?** Because `Collection` interface doesn't define `get()`.

✅ 4. How to fix it?

If you need to use `get(index)`, then use `List` as the reference type:

```java
List<String> fruits = new ArrayList<>();
fruits.add("Apple");
System.out.println(fruits.get(0)); // ✅ Apple
```

Because `List` interface **does have** `get()`.

✅ 5. Summary Table:

Reference Type	Object Type	Can use `get(index)`?	Notes
`Collection<String>`	`ArrayList<>`	❌ No	Only methods from `Collection` interface
`List<String>`	`ArrayList<>`	✅ Yes	Full list methods
`ArrayList<String>`	`ArrayList<>`	✅ Yes	Tightly coupled to implementation

✅ 1. Iterator

- Used to **loop through collections**
- Works with **Set**, **List**, **Queue**
- Can **remove** elements while iterating
- Only works **forward**

🔧 **Example:**

java Copy Edit

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Apple");
        names.add("Banana");
        names.add("Cherry");

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

        while (it.hasNext()) {
            String val = it.next();
            System.out.println(val);  // 🔸 Output: Apple Banana Cherry (each in a new line)
        }
    }
}
```

✅ 2. ListIterator

- Works only on **List** (ArrayList, LinkedList)
- Can go **forward and backward**
- Can **add, remove, update** during traversal

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("One");
        list.add("Two");
        list.add("Three");

        ListIterator<String> lit = list.listIterator();

        System.out.println("-- Forward --");
        while (lit.hasNext()) {
            System.out.println(lit.next());  // 🔸 Output: One Two Three
        }

        System.out.println("-- Backward --");
        while (lit.hasPrevious()) {
            System.out.println(lit.previous());  // 🔸 Output: Three Two One
        }
    }
}
```

✅ 3. Enumeration (Legacy - old version)

- Used only for **legacy classes** like Vector, Hashtable
- **Read-only**
- Only **forward** direction

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Vector<String> v = new Vector<>();
        v.add("X");
        v.add("Y");
        v.add("Z");

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

        while (e.hasMoreElements()) {
            System.out.println(e.nextElement());  //  🔸 Output: X Y Z
        }
    }
}
```

🔍 Comparison Table

Feature	Iterator	ListIterator	Enumeration
Direction	➡️ Forward	⬅️➡️ Forward & Backward	➡️ Forward only
Works on	List, Set, Queue	Only List	Only legacy (Vector, Hashtable)
Modify Collection?	✅ remove()	✅ add, remove, set	❌ No modification
Null-safe?	✅ Yes	✅ Yes	✅ Yes

✅ Comparable vs Comparator in Java

Feature	`Comparable`	`Comparator`
Package	`java.lang`	`java.util`
Sorting logic	**Inside the class**	**Outside the class**
Method	`compareTo(Object o)`	`compare(Object o1, Object o2)`
Used for	**Default/natural** sorting	**Custom** sorting
Affects original	Yes (class must implement it)	No (can use on same class multiple times)
Flexibility	Low (one logic)	High (multiple custom logics)

🔹 1. `Comparable` Example (Natural Sorting)

📄 Sort students by `rollNo` (ascending)

java

```java
import java.util.*;

class Student implements Comparable<Student> {
    int rollNo;
    String name;

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

    // Natural sorting by rollNo
    public int compareTo(Student s) {
        return this.rollNo - s.rollNo;
    }
    public String toString() {
        return rollNo + " - " + name;
    }
}

public class Main {
    public static void main(String[] args) {
        List<Student> list = new ArrayList<>();
        list.add(new Student(103, "Ravi"));
        list.add(new Student(101, "Arun"));
        list.add(new Student(102, "Kumar"));

        Collections.sort(list);  // uses compareTo()
        System.out.println("Sorted by rollNo:");
        for (Student s : list) {
            System.out.println(s);  // 🔸 Output: 101-Arun, 102-Kumar, 103-Ravi
        }
    }
}
```

🔹 2. `Comparator` Example (Custom Sorting)

📄 Sort students by `name` alphabetically

java

```java
import java.util.*;

class Student {
    int rollNo;
    String name;

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

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

class NameComparator implements Comparator<Student> {
    public int compare(Student s1, Student s2) {
        return s1.name.compareTo(s2.name);  // Alphabetical order
    }
}

public class Main {
    public static void main(String[] args) {
        List<Student> list = new ArrayList<>();
        list.add(new Student(103, "Ravi"));
        list.add(new Student(101, "Arun"));
        list.add(new Student(102, "Kumar"));

        Collections.sort(list, new NameComparator());  // uses compare()
        System.out.println("Sorted by name:");
        for (Student s : list) {
            System.out.println(s);  // 🔸 Output: 101-Arun, 102-Kumar, 103-Ravi
        }
    }
}
```

✅ Sorting and Filtering Examples

✅ Java Program: Student Sorting & Filtering

java

```java
import java.util.*;

class Student {
    int rollNo;
    String name;
    int age;
    double marks;

    Student(int rollNo, String name, int age, double marks) {
        this.rollNo = rollNo;
        this.name = name;
        this.age = age;
        this.marks = marks;
    }

    public String toString() {
        return rollNo + " - " + name + " - Age: " + age + " - Marks: " + marks;
    }
}
```

🔹 Comparator: Sort by Marks (Descending)

java

```java
class SortByMarksDesc implements Comparator<Student> {
    public int compare(Student s1, Student s2) {
        return Double.compare(s2.marks, s1.marks);  // Descending
    }
}
```

🔹 Comparator: Sort by Age (Ascending)

java

```java
class SortByAge implements Comparator<Student> {
    public int compare(Student s1, Student s2) {
        return Integer.compare(s1.age, s2.age);
    }
}
```

🔹 Main Program with Filtering

java

```java
public class Main {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student(101, "Arun", 20, 85.5));
        students.add(new Student(102, "Ravi", 22, 60.2));
        students.add(new Student(103, "Divya", 21, 95.0));
        students.add(new Student(104, "Kumar", 23, 72.8));

        // Sort by marks (Descending)
        Collections.sort(students, new SortByMarksDesc());
        System.out.println("\n-- Sorted by Marks (Desc) --");
        for (Student s : students) {
            System.out.println(s);
        }

        // Sort by age (Ascending)
        Collections.sort(students, new SortByAge());
        System.out.println("\n-- Sorted by Age (Asc) --");
        for (Student s : students) {
            System.out.println(s);
        }

        // Filter: Marks > 75
        System.out.println("\n-- Students with Marks > 75 --");
        for (Student s : students) {
            if (s.marks > 75) {
                System.out.println(s);
            }
        }
    }
}
```

✅ Output:

yaml

```yaml
-- Sorted by Marks (Desc) --
103 - Divya - Age: 21 - Marks: 95.0
101 - Arun - Age: 20 - Marks: 85.5
104 - Kumar - Age: 23 - Marks: 72.8
102 - Ravi - Age: 22 - Marks: 60.2

-- Sorted by Age (Asc) --
101 - Arun - Age: 20 - Marks: 85.5
103 - Divya - Age: 21 - Marks: 95.0
102 - Ravi - Age: 22 - Marks: 60.2
104 - Kumar - Age: 23 - Marks: 72.8

-- Students with Marks > 75 --
101 - Arun - Age: 20 - Marks: 85.5
103 - Divya - Age: 21 - Marks: 95.0
```

✅ Java 8 Major Features (Quick Summary)

Feature	Description
✅ Lambda Expressions	Anonymous function (no name)
✅ Functional Interfaces	Interface with **one abstract method**
✅ Stream API	Functional operations on collections
✅ Default & Static Methods	In interfaces
✅ Method References	Shorthand for lambda
✅ Optional Class	Avoid null pointer exception
✅ Date & Time API	New time package (`java.time`)

🔧 Example 1: Custom Functional Interface

java

```java
@FunctionalInterface
interface Greeting {
    void say(); // abstract method
}

public class Main {
    public static void main(String[] args) {
        Greeting g = () -> System.out.println("Hello, Functional Interface!");
        g.say();  // 🔸 Output: Hello, Functional Interface!
    }
}
```

🔧 Example 2: With Parameters

java

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

public class Main {
    public static void main(String[] args) {
        Calculator c = (a, b) -> a + b;
        System.out.println("Sum: " + c.add(10, 20));  // 🔸 Output: Sum: 30
    }
}
```

✅ Built-in Functional Interfaces (in `java.util.function`)

Interface	Method	Use Case Example
`Function<T,R>`	`R apply(T t)`	Convert one type to another
`Predicate<T>`	`boolean test(T t)`	Condition checking
`Consumer<T>`	`void accept(T t)`	Perform action
`Supplier<T>`	`T get()`	Supply data (no input)

✅ What is a Lambda Expression?

Lambda Expression = **Anonymous method** (a method without name)

It is used to implement **functional interfaces** in a **shorter, cleaner** way.

🔷 Syntax:

```java
(parameter) -> { method body }
```

- ✅ No need for method name
- ✅ No need for class
- ✅ Used with functional interfaces (like Runnable, Comparator, etc.)

🔧 Example 1: Without Lambda vs With Lambda

❌ Traditional Way:

```java
interface Greeting {
    void sayHello();
}

public class Main {
    public static void main(String[] args) {
        Greeting g = new Greeting() {
            public void sayHello() {
                System.out.println("Hello World");
            }
        };
        g.sayHello();
    }
}
```

🔧 Example 2: With Parameters

java

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

public class Main {
    public static void main(String[] args) {
        Calculator c = (a, b) -> a + b;
        System.out.println("Sum: " + c.add(10, 20));  // 🔹 Output: Sum: 30
    }
}
```

🔧 Example 3: Using Lambda with Thread

java

```java
public class Main {
    public static void main(String[] args) {
        Runnable r = () -> System.out.println("Running in a thread");
        Thread t = new Thread(r);
        t.start();  // 🔹 Output: Running in a thread
    }
}
```

🔧 Example 4: Using Lambda with Collections

java

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("Apple", "Banana", "Orange");

        list.forEach(item -> System.out.println(item));
        // 🔹 Output: Apple, Banana, Orange (each on new line)
    }
}
```

✅ Benefits of Lambda Expressions

- ✅ Less Code
- ✅ Easy to read and maintain
- ✅ Useful in functional programming
- ✅ Enables use of **Stream API**

🚨 Rules

- Lambda can only be used with **functional interfaces** (one abstract method)
- Parentheses can be removed if 1 parameter: `x -> x * x`
- Braces `{}` are optional if single line body

⚠️ Common Mistakes

❌ You cannot use lambda with multiple abstract methods
❌ You cannot use lambda with normal interfaces

✅ What is Stream API?

- Stream API helps you **process collections** (List, Set) in a **functional style**
- Doesn't modify the original list
- You can use operations like:
  filter(), map(), collect(), forEach(), count(), sorted(), etc.

🔹 Stream API Core Operations

Method	Description
`forEach()`	Loop through all elements
`filter()`	Select items based on condition
`map()`	Convert each element to something else
`collect()`	Collect result into List, Set, etc.
`sorted()`	Sort elements
`count()`	Count how many items match condition
`distinct()`	Remove duplicates

🔧 Sample List for Testing:

```java
List<String> fruits = Arrays.asList("Apple", "Banana", "Mango", "Banana", "Orange");
```

✅ 1. `forEach()` – print all items

```java
fruits.stream().forEach(fruit -> System.out.println(fruit));
```

🔸 **Output:**

```mathematica
Apple
Banana
Mango
Banana
Orange
```

✅ 2. `filter()` – only items starting with "B"

```java
fruits.stream()
      .filter(fruit -> fruit.startsWith("B"))
      .forEach(System.out::println);
```

🔸 **Output:**

```nginx
Banana
Banana
```

✅ 3. `map()` – convert all to uppercase

```java
fruits.stream()
      .map(f -> f.toUpperCase())
      .forEach(System.out::println);
```

🔸 **Output:**

```nginx
APPLE
BANANA
MANGO
BANANA
ORANGE
```

✅ 4. `collect()` – remove duplicates and collect to List

```java
List<String> uniqueFruits = fruits.stream()
                                  .distinct()
                                  .collect(Collectors.toList());
System.out.println(uniqueFruits);
```

🔸 **Output:**

```csharp
[Apple, Banana, Mango, Orange]
```

✅ 5. `sorted()` – natural order sort

```java
fruits.stream()
      .sorted()
      .forEach(System.out::println);
```

🔸 **Output:**

```mathematica
Apple
Banana
Banana
Mango
Orange
```

✅ 6. `count()` – count how many times “Banana” appears

```java
long count = fruits.stream()
                   .filter(f -> f.equals("Banana"))
                   .count();
System.out.println("Banana count: " + count);
```

🔸 **Output:**

```yaml
Banana count: 2
```

✅ Summary:

- `filter()` = condition check
- `map()` = transformation
- `forEach()` = loop
- `collect()` = collect to list
- `count()` = counting matching data
- `sorted()` = sorting
- `distinct()` = remove duplicates

✅ What is Method Reference in Java?

A **method reference** is a **shortcut for calling a method** using `::` operator.

It is used to **replace a lambda expression** when:

- The lambda just calls a method
- It **refers to an existing method**

🔹 Syntax:

```java
ClassName::methodName
or
object::methodName
```

✅ Types of Method References

Type	Syntax	Example
1. Reference to static method	`ClassName::staticMethod`	`Math::sqrt`
2. Reference to instance method	`object::instanceMethod`	`str::toLowerCase`
3. Reference to instance method of arbitrary object	`ClassName::instanceMethod`	`String::toUpperCase`
4. Constructor reference	`ClassName::new`	`Employee::new`

🔧 Example 1: Static Method Reference

java

```java
class Utility {
    public static void sayHello() {
        System.out.println("Hello from static method!");
    }
}

public class Main {
    public static void main(String[] args) {
        Runnable r = Utility::sayHello;  // method reference to static method
        r.run();  //  🔸 Output: Hello from static method!
    }
}
```

🔧 Example 2: Instance Method Reference

java

```java
class Message {
    void print() {
        System.out.println("This is an instance method!");
    }
}

public class Main {
    public static void main(String[] args) {
        Message msg = new Message();
        Runnable r = msg::print;
        r.run();  //  🔸 Output: This is an instance method!
    }
}
```

🔧 Example 3: Reference to Arbitrary Object Method

java

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("apple", "orange", "banana");

        // Using method reference to print each in upper case
        names.stream()
             .map(String::toUpperCase)
             .forEach(System.out::println);
    }
}
```

🔸 **Output:**

nginx

```
APPLE
ORANGE
BANANA
```

🔧 Example 4: Constructor Reference

java

```java
interface MyInterface {
    Student create(String name);
}

class Student {
    String name;

    Student(String name) {
        this.name = name;
        System.out.println("Student created: " + name);
    }
}

public class Main {
    public static void main(String[] args) {
        MyInterface ref = Student::new;  // constructor reference
        Student s = ref.create("Arun");  // 🔹 Output: Student created: Arun
    }
}
```

✅ Summary

Lambda Expression	Method Reference Equivalent
`() -> System.out.println("Hi")`	`System.out::println`
`(x) -> x.toUpperCase()`	`String::toUpperCase`
`(x) -> new Student(x)`	`Student::new`

💡 When to Use?

- When lambda is **just calling a method**, use method reference.
- It makes code **more readable**.

✅ 1. `Optional<T>` — To Avoid NullPointerException

`Optional` is a container object which may or may not contain a value.

```java
Optional<String> name = Optional.of("Arun");
System.out.println(name.get());  // Output: Arun

Optional<String> empty = Optional.empty();
System.out.println(empty.isPresent()); // Output: false

String result = empty.orElse("Default Name");
System.out.println(result); // Output: Default Name
```

✅ 2. `Predicate<T>` — Condition Testing (returns `true/false`)

Used to test a condition.

```java
Predicate<Integer> isEven = n -> n % 2 == 0;
System.out.println(isEven.test(10)); // true
System.out.println(isEven.test(5));  // false
```

Real-time filtering example:

```java
List<String> names = Arrays.asList("Arun", "Ravi", "Divya");

names.stream()
     .filter(name -> name.startsWith("A"))
     .forEach(System.out::println);  // Output: Arun
```

✅ 3. `Function<T, R>` — One input, One output

It transforms an input `T` to output `R`.

```java
Function<String, Integer> strLength = str -> str.length();
System.out.println(strLength.apply("Hello"));  // Output: 5
```

Real-time mapping:

```java
List<String> list = Arrays.asList("Apple", "Banana");

list.stream()
    .map(str -> str.length())
    .forEach(System.out::println);  // Output: 5, 6
```

✅ 4. `Consumer<T>` — Takes input, returns nothing

Used when you want to **perform some action** (like printing, storing), no return.

```java
Consumer<String> display = s -> System.out.println("Name: " + s);
display.accept("Arun");  // Output: Name: Arun
```

Real-time loop:

```java
List<Integer> list = Arrays.asList(10, 20, 30);
list.forEach(i -> System.out.println("Value: " + i));
```

✅ 5. `Supplier<T>` — No input, only output

It just supplies/generates a value.

```java
Supplier<String> supplier = () -> "Java Rocks!";
System.out.println(supplier.get());  // Output: Java Rocks!
```

📋 Summary Table

Interface	Method	Input	Output	Use
`Predicate`	`test()`	T	boolean	Check condition
`Function`	`apply()`	T	R	Convert/transform data
`Consumer`	`accept()`	T	void	Perform operation, no return
`Supplier`	`get()`	none	T	Supply value/object

Java 8 Date and Time API — specifically:

- ✅ LocalDate
- ✅ LocalTime
- ✅ LocalDateTime
- ✅ Period
- ✅ Duration

These classes are found in the package java.time.*

🗓️ 1. `LocalDate` – Date only (yyyy-MM-dd)

```java
import java.time.LocalDate;

public class Main {
    public static void main(String[] args) {
        LocalDate today = LocalDate.now();
        System.out.println("Today: " + today);  // Output: Today: 2025-07-28

        LocalDate dob = LocalDate.of(1998, 10, 15);
        System.out.println("DOB: " + dob);  // Output: DOB: 1998-10-15

        System.out.println("Year: " + dob.getYear());     // 1998
        System.out.println("Month: " + dob.getMonth());   // OCTOBER
        System.out.println("Day: " + dob.getDayOfMonth()); // 15
    }
}
```

⏰ 2. `LocalTime` – Time only (HH:mm:ss)

```java
import java.time.LocalTime;

public class Main {
    public static void main(String[] args) {
        LocalTime now = LocalTime.now();
        System.out.println("Current Time: " + now);  // Output: Current Time: 11:34:56.123

        LocalTime custom = LocalTime.of(9, 45);
        System.out.println("Custom Time: " + custom);  // Output: Custom Time: 09:45
    }
}
```

🕓 3. `LocalDateTime` – Date and Time

```java
import java.time.LocalDateTime;

public class Main {
    public static void main(String[] args) {
        LocalDateTime current = LocalDateTime.now();
        System.out.println("Now: " + current);  // Output: 2025-07-28T11:35:00.456
    }
}
```

⏳ 4. `Period` – Difference between two `LocalDate` (in years, months, days)

java Copy Edit

```java
import java.time.LocalDate;
import java.time.Period;

public class Main {
    public static void main(String[] args) {
        LocalDate start = LocalDate.of(2020, 1, 1);
        LocalDate end = LocalDate.now();

        Period period = Period.between(start, end);
        System.out.println("Years: " + period.getYears());
        System.out.println("Months: " + period.getMonths());
        System.out.println("Days: " + period.getDays());
    }
}
```

🔸 **Output Example:**

makefile

```
Years: 5
Months: 6
Days: 27
```

🕐 5. Duration – Difference between two LocalTime or LocalDateTime (in hours, minutes, seconds)

```java
import java.time.LocalTime;
import java.time.Duration;

public class Main {
    public static void main(String[] args) {
        LocalTime start = LocalTime.of(10, 30); // 10:30 AM
        LocalTime end = LocalTime.now();        // Assume current time is 13:15:30 (1:15:30 PM)

        Duration duration = Duration.between(start, end);
        System.out.println("Hours: " + duration.toHours());
        System.out.println("Minutes: " + duration.toMinutes());
        System.out.println("Seconds: " + duration.getSeconds());
    }
}
```

📃 **Output (Assuming current time is 13:15:30):**

yaml

```yaml
Hours: 2
Minutes: 165
Seconds: 9900
```

📌 Summary

Class	Purpose	Example Output
`LocalDate`	Only date	`2025-07-28`
`LocalTime`	Only time	`11:35:12.789`
`LocalDateTime`	Date + Time	`2025-07-28T11:35:12.789`
`Period`	Difference in years/months	`2 Years 4 Months 10 Days`
`Duration`	Difference in time (hours)	`3 hours 45 mins`

🌐 What is Multithreading?

Multithreading means **running multiple parts of a program at the same time (concurrently)**.

For example: downloading a file and showing progress bar both at the same time.

🌀 Thread Lifecycle

State	Meaning
NEW	Thread is created but not started
RUNNABLE	Thread is ready to run
RUNNING	Thread is executing
BLOCKED	Waiting to acquire lock
WAITING	Waiting for another thread
TERMINATED	Finished execution

✅ How to Create Threads

🔹 1. Extend `Thread` class

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

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

🟩 Output:

```arduino
Thread running using Thread class
```

🔧 Important Thread Methods

Method	Description
`start()`	Starts a new thread
`run()`	Logic of the thread goes here
`sleep(ms)`	Pauses thread for milliseconds
`join()`	Waits for thread to complete
`isAlive()`	Checks if thread is still running

🔐 `synchronized` Keyword

Used to prevent two threads from accessing shared data at the same time (race condition).

```java
class Counter {
    int count = 0;

    synchronized void increment() {
        count++;
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Counter c = new Counter();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) c.increment();
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) c.increment();
        });

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Final Count: " + c.count);
    }
}
```

🟩 Output:

```yaml
Final Count: 2000
```

🔁 `wait()` and `notify()`

Used for **inter-thread communication** like producer-consumer.

java

```java
class Shared {
    boolean ready = false;

    synchronized void produce() throws InterruptedException {
        System.out.println("Producing...");
        Thread.sleep(1000);
        ready = true;
        notify(); // wakes up waiting thread
    }

    synchronized void consume() throws InterruptedException {
        while (!ready) wait();  // waits for notify
        System.out.println("Consuming...");
    }
}
```

```java
public class Main {
    public static void main(String[] args) throws Exception {
        Shared s = new Shared();

        Thread producer = new Thread(() -> {
            try { s.produce(); } catch (Exception e) {}
        });

        Thread consumer = new Thread(() -> {
            try { s.consume(); } catch (Exception e) {}
        });

        consumer.start();
        producer.start();
    }
}
```

🟩 Output:

```
Producing...
Consuming...
```

⏳ `join()` Example

Wait for one thread to complete before continuing.

java

```java
public class Main {
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            for (int i = 1; i <= 3; i++) {
                System.out.println("T1: " + i);
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 1; i <= 3; i++) {
                System.out.println("T2: " + i);
            }
        });

        t1.start();
        t1.join();  // wait for t1 to finish
        t2.start(); // t2 starts after t1 completes
    }
}
```

🟩 Output:

makefile

```
T1: 1
T1: 2
T1: 3
T2: 1
T2: 2
T2: 3
```

📌 Summary

- ✅ `Thread` = unit of work
- ✅ `Runnable` = better for sharing logic
- ✅ `synchronized` = prevent data conflict
- ✅ `wait()` / `notify()` = for communication
- ✅ `join()` = wait for thread to finish

🧠 What is Thread Safety?

Thread safety means:

"When multiple threads access a shared resource (like a variable, file, or object), it behaves correctly and predictably."

In thread-safe code, you will **not get wrong or corrupted results** even when multiple threads access it at the same time.

❌ Problem Without Thread Safety

Let's say two threads update the same variable at the same time:

```java
class Counter {
    int count = 0;

    void increment() {
        count++; // not thread-safe
    }
}
```

Example:

```java
public class UnsafeCounterDemo {
    public static void main(String[] args) throws InterruptedException {
        Counter c = new Counter();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) c.increment();
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) c.increment();
        });

        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Final count: " + c.count);
    }
}
```

🔴 Expected Output: 2000

⚠️ Real Output: Varies like 1782, 1954, etc.

👉 Because count++ is **not atomic** (it's actually 3 operations: read → add → write).

✅ How to Make Code Thread Safe

🔐 1. Using `synchronized` Keyword

Locks a block or method so that only one thread can execute it at a time.

```java
class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

This prevents multiple threads from updating `count` simultaneously.

🧮 2. Using `AtomicInteger` (from `java.util.concurrent.atomic`)

This class supports **atomic operations** like `incrementAndGet()` safely across threads.

```java
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    AtomicInteger count = new AtomicInteger(0);

    void increment() {
        count.incrementAndGet();
    }

    int getCount() {
        return count.get();
    }
}
```

✅ Faster and lighter than `synchronized` in most cases.

🪛 3. Using `volatile` keyword (for visibility, not atomicity)

java Copy Edit

```java
class Flag {
    volatile boolean running = true;

    void stop() {
        running = false;
    }
}
```

🔸 `volatile` ensures all threads **see the latest value** of the variable from memory — no caching.

⚠️ Not for compound operations like `count++`.

🧺 4. Thread-safe Collections

Use these when multiple threads access collections:

Safe Collection	Type
`Vector`	List
`Collections.synchronizedList()`	List
`ConcurrentHashMap`	Map
`CopyOnWriteArrayList`	List (read-heavy use)

📌 Summary

Technique	Use When
`synchronized`	Critical section, simple locking
`AtomicInteger`	Fast atomic operations
`volatile`	Shared flags (true/false)
`Thread-safe collections`	Lists, maps in multi-threaded context
`ExecutorService`	Managing thread pools

❌ What is Deadlock?

Deadlock is a situation where **two or more threads are blocked forever**, waiting for each other to release the lock.

In simple words:

👉 Thread A holds Lock 1 and waits for Lock 2.

👉 Thread B holds Lock 2 and waits for Lock 1.

👉 Both are stuck forever!

🧠 Real-Life Analogy

Imagine:

- 🧍 Person A has 🔑 Key1 and wants 🔑 Key2 to enter a room.
- 🧍 Person B has 🔑 Key2 and wants 🔑 Key1.
- Both refuse to let go. They **wait forever**.

💻 Code Example of Deadlock

java

```java
class A {
    synchronized void methodA(B b) {
        System.out.println("Thread-1: locked A");

        try { Thread.sleep(100); } catch (Exception e) {}

        System.out.println("Thread-1: trying to lock B...");
        b.last();  // wants B's lock (held by Thread-2)
    }

    synchronized void last() {
        System.out.println("Inside A.last()");
    }
}

class B {
    synchronized void methodB(A a) {
        System.out.println("Thread-2: locked B");

        try { Thread.sleep(100); } catch (Exception e) {}
```

```
        System.out.println("Thread-2: trying to lock A...");
        a.last();  // wants A's lock (held by Thread-1)
    }

    synchronized void last() {
        System.out.println("Inside B.last()");
    }
}

public class DeadlockExample {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();

        Thread t1 = new Thread(() -> a.methodA(b));
        Thread t2 = new Thread(() -> b.methodB(a));

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

🟧 Output:

```
mathematica

Thread-1: locked A
Thread-2: locked B
Thread-1: trying to lock B...
Thread-2: trying to lock A...
// ❗ Now stuck – DEADLOCK!
```

🔄 How Deadlock Happens

4 Conditions of Deadlock (all must be true):

Condition	Meaning
1. Mutual Exclusion	A resource (lock) is held by only one thread
2. Hold and Wait	Thread holds one lock and waits for another
3. No Preemption	Locks cannot be forcibly taken
4. Circular Wait	Each thread waits for the next in a loop

✅ If any one is broken → Deadlock won't happen.

✅ How to Prevent Deadlock

✅ 1. Lock Ordering

Always acquire locks in the same order in all threads.

```java
class NoDeadlock {
    private final Object lock1 = new Object();
    private final Object lock2 = new Object();

    void method1() {
        synchronized (lock1) {
            synchronized (lock2) {
                System.out.println("No Deadlock: method1");
            }
        }
    }

    void method2() {
        synchronized (lock1) {  // same order
            synchronized (lock2) {
                System.out.println("No Deadlock: method2");
            }
        }
    }
}
```

✅ 2. Try-Lock (Advanced – `ReentrantLock` with timeout)

```java
import java.util.concurrent.locks.*;

class SafeLock {
    private final Lock lock1 = new ReentrantLock();
    private final Lock lock2 = new ReentrantLock();

    void method() {
        try {
            if (lock1.tryLock() && lock2.tryLock()) {
                System.out.println("Safe execution");
            }
        } finally {
            if (lock1.tryLock()) lock1.unlock();
            if (lock2.tryLock()) lock2.unlock();
        }
    }
}
```

✅ 3. Avoid Nested Locks

Avoid using one lock **inside another lock**, if possible.

✅ 4. Use Timed Waiting

Example: tryLock(timeout) in ReentrantLock

🧪 Detecting Deadlocks (Advanced)

Use Java tools like:

- 🔍 `jconsole`
- 🔍 `jvisualvm`
- 🧰 `ThreadMXBean` (in code)

📃 Summary

Topic	Description
Deadlock	Two or more threads wait forever on each other
Cause	Locks acquired in inconsistent order
Fixes	Lock ordering, tryLock, avoid nested locks
Tools	jconsole, visualvm, ThreadMXBean

🏁 What is a Race Condition?

A Race Condition occurs when two or more threads access shared data and try to change it at the same time. Because thread execution order is unpredictable, the final result becomes inconsistent.

🔥 Example of Race Condition

Let’s say two threads are trying to increment a counter variable from 0 to 1000.

🚨 Code Without Thread Safety (Risk of Race Condition):

java

```java
class Counter {
    int count = 0;  // shared variable among threads

    void increment() {
        count++;  // NOT atomic (read, increment, write)
    }
}

public class RaceConditionDemo {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        // Thread 1 increments count 1000 times
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();  // shared access
            }
        });

        // Thread 2 increments count 1000 times
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();  // shared access
            }
        });

        t1.start();  // starts thread-1
        t2.start();  // starts thread-2

        t1.join();   // wait for thread-1 to finish
        t2.join();   // wait for thread-2 to finish

        //  ❗ Expected result: 2000
        // 🔴 But you may get incorrect result due to Race Condition
        System.out.println("Final Count: " + counter.count);
    }
}
```

🟧 Sample Output:

yaml

```yaml
Final Count: 1982
```

🛑 Sometimes you may get:

- 1956
- 1990
- 2000 ✅ (rare, but possible)
- 1883 ❌

Because count++ is **not atomic**, both threads may:

- Read the same value at the same time
- Increment it
- Write back an outdated value

🔬 Why `count++` Is Not Safe?

It looks simple, but it's actually:

```java
int temp = count;   // read
temp = temp + 1;    // increment
count = temp;       // write back
```

If two threads do this at the same time → one value will be lost.

✅ Solution 1: Make it Thread-Safe using `synchronized`

```java
class SafeCounter {
    int count = 0;

    synchronized void increment() {
        count++;  // locked to one thread at a time
    }
}
```

🔐 `synchronized` makes sure **only one thread can execute** the method at a time.

✅ Solution 2: Use `AtomicInteger`

java

```java
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    AtomicInteger count = new AtomicInteger(0);  // thread-safe class

    void increment() {
        count.incrementAndGet();  // atomic operation
    }

    int getCount() {
        return count.get();
    }
}
```

✅ Comparison of Outputs:

Approach	Output (Expected)
❌ Unsafe Counter	1867 (varies)
✅ Synchronized	2000
✅ AtomicInteger	2000

📃 Summary:

Concept	Description
Race Condition	Threads accessing and modifying shared data simultaneously (unsafe)
Cause	Lack of synchronization or atomicity
Solution	`synchronized`, `AtomicInteger`, thread-safe collections
Real Danger	In banking apps, ticket booking, etc. → critical data corruption possible

✅ ExecutorService, Callable, and Future in Java

Java provides a modern and powerful way to handle threads using the java.util.concurrent package. The main concepts here are:

1. ExecutorService

What is it?

ExecutorService is an interface used to manage and control thread pools. It separates **task submission** from **thread creation and execution**.

Common Implementations:

```java
Executors.newFixedThreadPool(3);       // Fixed 3-thread pool
Executors.newSingleThreadExecutor();   // Only 1 thread
Executors.newCachedThreadPool();       // Dynamic thread creation
```

Basic Example:

```java
ExecutorService executor = Executors.newFixedThreadPool(2);

executor.execute(() -> {
    System.out.println("Task running in thread: " + Thread.currentThread().getName());
});

executor.shutdown(); // Always shutdown after use
```

2. Callable

What is it?

`Callable<T>` is a functional interface like `Runnable`, but:

- Returns a result (with type `T`)
- Can throw checked exceptions

Syntax:

```java
Callable<String> task = () -> {
    return "Hello from Callable";
};
```

🔄 3. Future

What is it?

`Future<T>` is used to **hold the result** of a `Callable` task that's being executed by an ExecutorService.

Key Methods:

- `get()` → waits for result
- `get(timeout)` → waits for result with timeout
- `isDone()` → true if task completed
- `cancel(true/false)` → cancel the task

🔧 Full Example with Output and Comments

java

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

public class CallableFutureExample {
    public static void main(String[] args) throws Exception {
        // 🔧 Step 1: Create ExecutorService
        ExecutorService executor = Executors.newSingleThreadExecutor();

        // 🧠 Step 2: Create Callable task
        Callable<String> task = () -> {
            Thread.sleep(1000); // Simulate delay
            return "✅ Task Completed by: " + Thread.currentThread().getName();
        };

        // 🚀 Step 3: Submit Callable and get Future object
        Future<String> future = executor.submit(task);

        System.out.println("⏳ Main thread continues...");

        // 📦 Step 4: Get the result (waits if needed)
        String result = future.get(); // Blocks until task is done
        System.out.println(result);  // Output the result

        // 🛑 Step 5: Always shutdown the executor
        executor.shutdown();
    }
}
```

🟢 Output:

arduino

```
⏳ Main thread continues...
✅ Task Completed by: pool-1-thread-1
```

✅ When to Use `ExecutorService + Callable + Future`

These 3 together help you manage **multi-threading with results** and **asynchronous tasks** efficiently.

📌 1. When You Need to Return a Result from a Thread

Unlike `Runnable`, `Callable` allows threads to **return a result**.

🧪 Example:

```java
Callable<Integer> task = () -> 10 + 20;  // returns 30
Future<Integer> result = executor.submit(task);
System.out.println(result.get());  // Output: 30
```

📌 2. When You Want to Handle Exceptions from Threads

`Callable.call()` can throw **checked exceptions** (unlike `Runnable.run()`).

🧪 Use case: File reading, DB access, API calls

```java
Callable<String> task = () -> {
    if (new Random().nextBoolean()) throw new IOException("Failed");
    return "Success";
};
```

📌 3. When You Want to Run Multiple Tasks in Parallel and Wait for All

Use `invokeAll()` to run multiple tasks **simultaneously** and wait for all to finish.

🧪 Example:

```java
List<Callable<String>> tasks = List.of(() -> "A", () -> "B", () -> "C");
List<Future<String>> results = executor.invokeAll(tasks);
```

📌 4. When You Want to Fetch Results Later (Lazy/Efficient Execution)

Use `Future` to **submit a task now** and **get the result later**.

🧪 Example:

```java
Future<String> future = executor.submit(task);
// ... do other work
String result = future.get();  // wait only when you need the result
```

📌 5. When You Want to Cancel a Long-Running Task

With `Future.cancel(true)`, you can cancel tasks.

🧪 Example:

```java
Future<?> future = executor.submit(longTask);
Thread.sleep(1000);
future.cancel(true);  // interrupt the thread
```

📌 6. When You Want Timeout Control on a Task

Avoid hanging forever! Use `future.get(timeout)`.

🧪 Example:

```java
Future<String> future = executor.submit(someTask);
String result = future.get(2, TimeUnit.SECONDS);  // Timeout
```

📂 Java I/O – File Handling

Java provides powerful classes in the `java.io` package to handle files. The most commonly used classes for file reading/writing are:

Class	Purpose
`FileWriter`	Write text data to file
`FileReader`	Read character data from file
`BufferedWriter`	Write text efficiently (with buffer)
`BufferedReader`	Read text line by line (efficiently)

📊 Comparison

Feature	FileWriter/Reader	BufferedWriter/Reader
Speed	Normal	Faster (buffered)
Reads line by line	❌ No	✅ Yes
Suitable for	Small files	Large files

✍️ 1. Writing to a File using FileWriter

java

```java
import java.io.FileWriter;
import java.io.IOException;

public class FileWriteExample {
    public static void main(String[] args) throws IOException {
        FileWriter fw = new FileWriter("data.txt");
        fw.write("Hello from Java!\n");
        fw.write("This is the second line.");
        fw.close();  // Always close the writer
        System.out.println("✅ Data written successfully.");
    }
}
```

✅ **Output** in `data.txt`:

pgsql

```
Hello from Java!
This is the second line.
```

📖 2. Reading a File using FileReader

java

```java
import java.io.FileReader;
import java.io.IOException;

public class FileReadExample {
    public static void main(String[] args) throws IOException {
        FileReader fr = new FileReader("data.txt");
        int i;
        while ((i = fr.read()) != -1) {
            System.out.print((char) i); // prints char by char
        }
        fr.close();
    }
}
```

✅ **Output**:

pgsql

```
Hello from Java!
This is the second line.
```

📄 3. Reading Line-by-Line using BufferedReader

```java
import java.io.*;

public class BufferedReadExample {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new FileReader("data.txt"));
        String line;
        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }
        br.close();
    }
}
```

✅ **Output**:

```pgsql
Hello from Java!
This is the second line.
```

✍️ 4. BufferedWriter Example

```java
import java.io.*;

public class BufferedWriteExample {
    public static void main(String[] args) throws IOException {
        BufferedWriter bw = new BufferedWriter(new FileWriter("data.txt"));
        bw.write("BufferedWriter Line 1");
        bw.newLine();  // adds new line
        bw.write("Line 2");
        bw.close();
        System.out.println("✅ Written using BufferedWriter.");
    }
}
```

✅ Summary

Class	Use
FileWriter	Write text to file
FileReader	Read characters from file
BufferedWriter	Efficient writing with buffering
BufferedReader	Efficient reading line by line

🔍 Byte Stream vs Character Stream in Java

Java I/O is divided into two major stream types:

Stream Type	Package	Base Classes	Used For
Byte Stream	`java.io`	`InputStream`, `OutputStream`	Binary data (PDF, images, videos)
Character Stream	`java.io`	`Reader`, `Writer`	Text data (letters, characters)

1️⃣ Byte Stream

✅ Reads/Writes data in bytes (8 bits)

- Handles any type of data (binary or text)
- Not character-aware (no encoding handled)
- Base classes: InputStream and OutputStream

📌 Common Classes:

- FileInputStream
- FileOutputStream
- BufferedInputStream
- BufferedOutputStream

🔧 Example:

java Copy Edit

```java
import java.io.*;

public class ByteStreamExample {
    public static void main(String[] args) throws IOException {
        FileOutputStream fos = new FileOutputStream("bytefile.dat");
        fos.write(65); // Writes byte value (ASCII of 'A')
        fos.close();

        FileInputStream fis = new FileInputStream("bytefile.dat");
        int data = fis.read();  // Reads 1 byte
        System.out.println("Read byte: " + data + " = " + (char) data);  // 65 = A
        fis.close();
    }
}
```

✅ Output:

arduino Copy Edit

```
Read byte: 65 = A
```

2️⃣ Character Stream

✅ Reads/Writes data in characters (16 bits)

- Specifically designed for text data
- Handles encoding (like UTF-8, UTF-16) automatically
- Base classes: Reader and Writer

📌 Common Classes:

- FileReader
- FileWriter
- BufferedReader
- BufferedWriter

🔧 Example:

```java
import java.io.*;

public class CharStreamExample {
    public static void main(String[] args) throws IOException {
        FileWriter fw = new FileWriter("charfile.txt");
        fw.write("Java");
        fw.close();

        FileReader fr = new FileReader("charfile.txt");
        int ch;
        while ((ch = fr.read()) != -1) {
            System.out.print((char) ch);  // prints J a v a
        }
        fr.close();
    }
}
```

✅ Output:

```nginx
Java
```

⚖️ Byte vs Character Streams – Comparison Table

Feature	Byte Stream	Character Stream
Data Unit	8-bit byte	16-bit character
Handles text encoding	❌ No	✅ Yes
Base Classes	`InputStream` / `OutputStream`	`Reader` / `Writer`
For binary files (images)	✅ Yes	❌ No
For text files (txt, csv)	❌ Not preferred	✅ Preferred

🧠 When to Use What?

Use Case	Stream Type
Reading an image, video, PDF	Byte Stream
Reading plain text (txt, XML)	Character Stream
Transferring network bytes	Byte Stream
Reading config files (text)	Character Stream

📦 Serialization vs Deserialization in Java

📄 Definition

Term	Meaning
Serialization	Converting a Java object into a **byte stream** so it can be saved or transferred
Deserialization	Converting the **byte stream back into the original Java object**

✅ Why Use It?

- To **save** an object's state to a file
- To **transfer** objects between systems (e.g., over a network)
- To **store object** in a database or cache

Classes & Interfaces Involved

Component	Purpose
`Serializable` interface	Marker interface (no methods) required to serialize an object
`ObjectOutputStream`	Writes serialized object to a stream
`ObjectInputStream`	Reads and restores the object

Serialization Example

java

```java
import java.io.*;

// Must implement Serializable
class Student implements Serializable {
    int id;
    String name;

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

```java
public class SerializeExample {
    public static void main(String[] args) throws Exception {
        Student s = new Student(101, "Subbiah");

        // Write object to file
        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("student.ser"));
        oos.writeObject(s);
        oos.close();

        System.out.println("✅ Object serialized.");
    }
}
```

📥 Deserialization Example

java Copy Edit

```java
import java.io.*;

public class DeserializeExample {
    public static void main(String[] args) throws Exception {
        // Read object from file
        ObjectInputStream ois = new ObjectInputStream(new FileInputStream("student.ser"));
        Student s = (Student) ois.readObject();
        ois.close();

        System.out.println("✅ Object deserialized.");
        System.out.println("ID: " + s.id + ", Name: " + s.name);
    }
}
```

🟢 Output:

vbnet

```
✅ Object serialized.
✅ Object deserialized.
ID: 101, Name: Subbiah
```

❗ Important Notes

- The class **must implement** `Serializable`
- If a field is marked `transient`, it won't be saved during serialization
- Always provide the **same class definition** while deserializing

⚖️ Serialization vs Deserialization – Comparison Table

Feature	Serialization	Deserialization
Direction	Object → Byte Stream	Byte Stream → Object
Class Used	`ObjectOutputStream`	`ObjectInputStream`
Common Use	Save to file, send over network	Restore from file/network
Required Interface	`Serializable`	Same class must be available
Can skip fields	Yes (using `transient`)	No – expects matching data

🧠 Real Use Cases

- **Storing session state** (e.g., in web apps)
- **Messaging**: Send Java objects through network (like RMI, sockets)
- **Caching**: Save object states in serialized form
- **Saving game progress** in desktop apps

✅ JDBC (Java Database Connectivity)

JDBC allows Java applications to connect and interact with databases like MySQL, Oracle, PostgreSQL, etc.

🔁 JDBC Workflow Steps

1. Load the Driver
2. Establish the Connection
3. Create a Statement
4. Execute the Query
5. Process the ResultSet
6. Close the Resources

1️⃣ Load the Driver

```java
Class.forName("com.mysql.cj.jdbc.Driver");
```

- Loads the MySQL JDBC driver class.
- This step registers the driver with the DriverManager.

⚠️ From JDBC 4.0+, this step is optional if driver is in classpath.

2 Create a Connection

```java
Connection con = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/testdb", "root", "password");
```

- Connects to MySQL at port `3306`, to database `testdb`, with username and password.

3 Create a Statement

```java
Statement stmt = con.createStatement();
```

- Used to send SQL queries like SELECT, INSERT, etc.

4 Execute a Query

```java
ResultSet rs = stmt.executeQuery("SELECT * FROM students");
```

- For `SELECT` queries, use `executeQuery()`.
- For `INSERT`, `UPDATE`, or `DELETE`, use `executeUpdate()`.

5 Process the ResultSet

```java
while (rs.next()) {
    int id = rs.getInt("id");
    String name = rs.getString("name");
    System.out.println("ID: " + id + ", Name: " + name);
}
```

- Reads each row of result from the database.

6 Close Resources

```java
rs.close();
stmt.close();
con.close();
```

- Always close all resources to prevent memory leaks.

🧪 Full Working Example

java

```java
import java.sql.*;

public class JDBCDemo {
    public static void main(String[] args) {
        try {
            // 1. Load Driver
            Class.forName("com.mysql.cj.jdbc.Driver");

            // 2. Establish Connection
            Connection con = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/testdb", "root", "password");

            // 3. Create Statement
            Statement stmt = con.createStatement();

            // 4. Execute SELECT Query
            ResultSet rs = stmt.executeQuery("SELECT * FROM students");

            // 5. Process ResultSet
            while (rs.next()) {
                System.out.println("ID: " + rs.getInt("id") +
                                   ", Name: " + rs.getString("name"));
            }

            // 6. Close all
            rs.close();
            stmt.close();
            con.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

🔍 Interfaces Used in JDBC

Interface	Role
`Driver`	Connects Java with DB
`Connection`	Represents DB connection session
`Statement`	Sends SQL query
`PreparedStatement`	Optimized query with placeholders
`ResultSet`	Holds data from SELECT queries

✅ Best Practices

- Use `PreparedStatement` to avoid SQL injection.
- Always close all resources (`ResultSet` , `Statement` , `Connection`) inside `finally` block or use **try-with-resources**.

🥊 `Statement` vs `PreparedStatement`

Feature	`Statement`	`PreparedStatement`
Query Type	Static SQL (no parameters)	Dynamic SQL with **? parameters**
Query Compiled Every Time	✅ Yes	❌ No (compiled once, reused)
Performance	Slower (on repetitive queries)	Faster (precompiled and reusable)
SQL Injection Risk	❌ High	✅ Low (auto escapes input)
Readability	Hard with dynamic values	Clean and clear
Use Case	Simple queries, one-time queries	Repeated queries, user input, secure input

✅ When to Use Which?

Use Case	Use
Hardcoded SQL without input	`Statement`
Dynamic SQL with user input	`PreparedStatement` ✅ (recommended)
Repetitive queries in loop	`PreparedStatement` ✅

📜 `Statement` Example

```java
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM users WHERE name = 'John'");
```

⚠️ **Risk**: If user input is injected:

```java
String name = "' OR '1'='1";
String query = "SELECT * FROM users WHERE name = '" + name + "'";
```

It becomes:

```sql
SELECT * FROM users WHERE name = '' OR '1'='1'
```

➡️ **All rows will be selected** → SQL Injection vulnerability!

✅ `PreparedStatement` Example (Safe)

```java
PreparedStatement ps = con.prepareStatement("SELECT * FROM users WHERE name = ?");
ps.setString(1, "John"); // 1 → first '?'
ResultSet rs = ps.executeQuery();
```

- ✔ Cleaner
- ✔ Faster
- ✔ Safe from injection

🔁 Reusability Example

```java
PreparedStatement ps = con.prepareStatement("INSERT INTO employees(name, age) VALUES (?, ?)");

ps.setString(1, "Subbiah");
ps.setInt(2, 26);
ps.executeUpdate();

ps.setString(1, "Ram");
ps.setInt(2, 30);
ps.executeUpdate();
```

✅ Same query reused by changing parameters → High performance

📌 Summary

Criteria	`Statement`	`PreparedStatement`
Supports parameters	❌ No	✅ Yes (`?` placeholders)
Compiled once and reused	❌ No	✅ Yes
Safe from SQL Injection	❌ No	✅ Yes
Best For	Simple SQL	Dynamic/user input-based SQL

🔄 JDBC Transactions (`commit`, `rollback`)

🔍 What is a Transaction?

A **transaction** is a group of SQL operations executed as a single unit of work.
It must follow **ACID** properties (Atomicity, Consistency, Isolation, Durability).

✅ Example:

```sql
1. Deduct ₹500 from Account A
2. Add ₹500 to Account B
```

If one fails, both should fail — that's a transaction.

🧪 Full Transaction Example

java Copy Edit

```java
import java.sql.*;

public class TransactionExample {
    public static void main(String[] args) {
        try {
            Connection conn = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/testdb", "root", "password");

            // 1. Disable auto-commit
            conn.setAutoCommit(false);

            // 2. Create statements
            Statement stmt = conn.createStatement();
            stmt.executeUpdate("UPDATE accounts SET balance = balance - 500 WHERE id = 1");  // A
            stmt.executeUpdate("UPDATE accounts SET balance = balance + 500 WHERE id = 2");  // B

            // 3. Commit transaction
            conn.commit();
            System.out.println("✅ Transaction successful. Amount transferred.");

        } catch (Exception e) {
            e.printStackTrace();
            try {
                // 4. Rollback if error
                conn.rollback();
                System.out.println("❌ Transaction failed. Rolled back.");
            } catch (SQLException ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

💡 When to Use Transactions?

- Money transfers
- Order placement (inserting order + updating stock)
- Multiple queries that depend on each other

⚠️ Notes:

Point	Description
`setAutoCommit(false)`	Start transaction
`commit()`	End transaction (success)
`rollback()`	Cancel transaction (on error)
Always close resources	Use try-with-resources or finally block

🧠 Summary

Step	Description
1️⃣	Turn off auto-commit
2️⃣	Execute multiple SQL queries
3️⃣	Commit if all succeed
4️⃣	Rollback if any fails

🏊‍♂️ What is Connection Pooling in Java?

✅ Simple Definition:

Connection Pooling is a technique to **reuse database connections** instead of creating a new one every time.

🔍 Without Pooling

Every time your code runs:

1. Load Driver
2. Open a new connection
3. Use it
4. Close it

🔁 Repeating this for each request is **slow and expensive**, especially in large applications.

🚀 With Connection Pooling

- A **pool of pre-created connections** is maintained.
- When you need a connection, you **borrow** it from the pool.
- After use, you **return** it (not close).
- The pool reuses it for the next request.

🎯 Benefit:

- 🔄 Reuse existing connections
- ⚡ Improves performance
- 🔐 Efficient resource management

🧪 Real-Life Analogy

🔌 Think of it like **renting a power bank** from a station:

- You don’t buy a new one every time.
- You borrow, use, return.
- Someone else can use it again.

📦 Popular Connection Pool Libraries

Library	Description
HikariCP	Fastest and most used (Spring Boot default)
Apache DBCP	Apache’s connection pool
C3P0	Older, but reliable

📃 Basic Example using HikariCP

java

```
HikariConfig config = new HikariConfig();
config.setJdbcUrl("jdbc:mysql://localhost:3306/testdb");
config.setUsername("root");
config.setPassword("password");

HikariDataSource ds = new HikariDataSource(config);
Connection con = ds.getConnection();
```

You now **get connections from the pool** (not new ones every time).

⚙️ Behind the Scenes:

1. On startup → pool creates N connections.
2. On request → gives a free connection.
3. On return → adds back to pool.
4. On shutdown → closes all connections.

🧠 Summary

Feature	Without Pooling	With Pooling
Performance	❌ Slow	✅ Fast
Resource usage	🔴 High	🟢 Efficient
Connection reuse	❌ No	✅ Yes

✅ Java Annotations & Reflection

Java **annotations** are metadata that can be added to code to give information to the compiler or runtime.

📦 1. Built-in Annotations

✅ `@Override`

Tells the compiler that a method is being overridden from a superclass or interface.

java Copy Edit

```java
@Override
public String toString() {
    return "Hello";
}
```

If the method signature is wrong, the compiler gives an error.

✅ `@Deprecated`

Marks the method or class as **not recommended** for use. Compiler shows a warning.

java

```java
@Deprecated
public void oldMethod() {
    System.out.println("Use newMethod() instead!");
}
```

✅ `@SuppressWarnings`

Tells the compiler to **ignore certain warnings**.

java

```java
@SuppressWarnings("unchecked")
List list = new ArrayList();
```

You can suppress:

- **"unchecked"**
- **"deprecation"**
- **"rawtypes"**

🛠️ 2. Custom Annotations

You can define your own annotation using `@interface`.

✅ Define:

```java
@interface Info {
    String author();
    int version() default 1;
}
```

✅ Use:

```java
@Info(author = "Subbiah", version = 2)
public class MyClass {
    // some logic
}
```

🔍 3. Reflection Basics

Reflection is a feature that allows Java code to inspect and modify classes, methods, fields, and annotations at **runtime**.

Use Cases:

- Frameworks (like Spring, Hibernate)
- Tools that scan classes and methods
- Unit testing frameworks

🧪 Example: Reading Annotation with Reflection

```java
import java.lang.annotation.*;
import java.lang.reflect.*;

// Define annotation
@Retention(RetentionPolicy.RUNTIME)
@interface Info {
    String author();
}

// Apply annotation
@Info(author = "Subbiah")
class Book {
}

// Reflection to read annotation
public class ReflectionDemo {
    public static void main(String[] args) {
        Class<Book> cls = Book.class;
        Annotation[] annotations = cls.getAnnotations();

        for (Annotation a : annotations) {
            if (a instanceof Info) {
                Info info = (Info) a;
                System.out.println("Author: " + info.author());
            }
        }
    }
}
```

✅ Output:

```makefile
Author: Subbiah
```

📘 Summary Table

Feature	Example
`@Override`	Overrides a method
`@Deprecated`	Marks something as outdated
`@SuppressWarnings`	Suppress compile warnings
Custom Annotation	You define using `@interface`
Reflection	Inspect classes at runtime

🧠 What are Design Patterns?

Design patterns are **proven solutions** to common problems in software design. They improve code **reusability**, **scalability**, and **maintainability**.

✅ 1. Singleton Pattern

🎯 Purpose:

Ensure that a **class has only one instance** and provides a global point of access to it.

📄 Real-World Example:

- One **Database Connection Manager**
- One **Logger class**
- One **Configuration Reader**

🧪 Singleton Example (Lazy Initialization)

java

```java
class Singleton {
    private static Singleton instance;

    // Private constructor
    private Singleton() {
        System.out.println("🔒 Singleton constructor called");
    }

    // Public method to return the instance
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();  // creates only once
        }
        return instance;
    }
}
public class Main {
    public static void main(String[] args) {
        Singleton obj1 = Singleton.getInstance();
        Singleton obj2 = Singleton.getInstance();

        System.out.println(obj1 == obj2);  // true → same object
    }
}
```

✅ Output:

kotlin

```kotlin
🔒 Singleton constructor called
true
```

🔐 **Key Rules:**

- Constructor is **private**
- Static method gives access
- Static variable stores the instance

✅ 2. Factory Pattern

🎯 **Purpose:**

To **create objects** without exposing the object creation logic to the client and to refer to the newly created object using a common interface.

📄 **Real-World Example:**

- **ShapeFactory** creates objects like Circle, Square
- JDBC uses DriverManager:
  Connection con = DriverManager.getConnection(...)

🧱 Structure

java

```java
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");
    }
}
```

```java
class ShapeFactory {
    public Shape getShape(String type) {
        if (type.equalsIgnoreCase("circle")) {
            return new Circle();
        } else if (type.equalsIgnoreCase("square")) {
            return new Square();
        }
        return null;
    }
}

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

        Shape shape1 = factory.getShape("circle");
        shape1.draw();  // Output: Drawing Circle

        Shape shape2 = factory.getShape("square");
        shape2.draw();  // Output: Drawing Square
    }
}
```

✅ **Output:**

```
scss

Drawing Circle
Drawing Square
```

🧱 3. Builder Pattern

🎯 **Purpose:**

To build complex objects step-by-step, especially when an object has many optional fields.

✅ Example:

```java
class Computer {
    private String cpu;
    private String ram;

    // Private constructor
    private Computer(Builder builder) {
        this.cpu = builder.cpu;
        this.ram = builder.ram;
    }

    public static class Builder {
        private String cpu;
        private String ram;

        public Builder setCpu(String cpu) {
            this.cpu = cpu;
            return this;
        }
        public Builder setRam(String ram) {
            this.ram = ram;
            return this;
        }

        public Computer build() {
            return new Computer(this);
        }
    }

    public String toString() {
        return "CPU: " + cpu + ", RAM: " + ram;
    }
}

public class Main {
    public static void main(String[] args) {
        Computer c = new Computer.Builder()
                        .setCpu("i5")
                        .setRam("16GB")
                        .build();

        System.out.println(c);
    }
}
```

✅ Output:

```yaml
CPU: i5, RAM: 16GB
```

🔁 4. Strategy Pattern

🎯 Purpose:

Encapsulate different algorithms (or strategies) and make them interchangeable at runtime.

✅ Example:

```java
interface PaymentStrategy {
    void pay(int amount);
}

class CardPayment implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid Rs." + amount + " using Card");
    }
}

class UPIPayment implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid Rs." + amount + " using UPI");
    }
}

class PaymentContext {
    private PaymentStrategy strategy;

    public void setStrategy(PaymentStrategy strategy) {
        this.strategy = strategy;
    }

    public void executePayment(int amount) {
        strategy.pay(amount);
    }
}

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

        context.setStrategy(new CardPayment());
        context.executePayment(1000);

        context.setStrategy(new UPIPayment());
        context.executePayment(500);
    }
}
```

✅ Output:

```cpp
Paid Rs.1000 using Card
Paid Rs.500 using UPI
```

👥 5. Observer Pattern

🎯 Purpose:

When one object changes, all its dependents (observers) are automatically notified.

✅ Example:

```java
import java.util.*;

interface Observer {
    void update(String msg);
}

class Subscriber implements Observer {
    String name;

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

    public void update(String msg) {
        System.out.println(name + " received: " + msg);
    }
}

class Channel {
    List<Observer> subscribers = new ArrayList<>();

    public void subscribe(Observer o) {
        subscribers.add(o);
    }

    public void notifyAll(String msg) {
        for (Observer o : subscribers) {
            o.update(msg);
        }
    }
}

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

        Subscriber a = new Subscriber("UserA");
        Subscriber b = new Subscriber("UserB");

        yt.subscribe(a);
        yt.subscribe(b);

        yt.notifyAll("New video uploaded!");
    }
}
```

✅ Output:

```less
UserA received: New video uploaded!
UserB received: New video uploaded!
```

Pattern	Purpose	Key Concepts	Common Use Cases
Singleton	Ensure only one instance exists	- Private constructor - Static instance - `getInstance()`	Logger, Config reader, DB connection
Factory	Create objects without exposing creation logic	- Common interface - Factory class creates objects	UI elements, Notification system, JDBC
Builder	Construct complex objects step-by-step	- Chain methods - Nested `Builder` class - Immutable	`Computer`, `Pizza`, `User` objects
Strategy	Select algorithm/behavior at runtime	- Interface for strategies - Set strategy at runtime	Payment methods, Sorting, File compression
Observer	Notify multiple objects when one changes state	- Subject + Observers - `subscribe()`, `notifyAll()`	Event handling, UI updates, Messaging systems

---