Unit 1: Java Basics and Control Structures

1.1.1 What is Java?

Java is a **high-level, class-based, object-oriented programming language** that is designed to have as few implementation dependencies as possible. It is a **general-purpose language**, designed to let application developers write once, run anywhere (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation.

1.1.2 History of Java

YEAR	EVENT
1991	Project started at Sun Microsystems by James Gosling (originally called *Oak*)
1995	Officially released as Java 1.0
2006	Sun released Java as open-source (OpenJDK)
2010	Oracle acquired Sun Microsystems
2023+	Java is one of the most widely-used languages for web, mobile, and enterprise applications

1.1.3 Key Features of Java

FEATURE	DESCRIPTION
SIMPLE	Easy syntax, inspired by C/C++, but without complex features like pointers
OBJECT-ORIENTED	Everything is treated as an object, supports OOP principles
PLATFORM INDEPENDENT	Uses the Java Virtual Machine (JVM) to execute bytecode on any OS
SECURE	No explicit memory access; includes bytecode verification
ROBUST	Strong memory management, exception handling, and type checking
MULTITHREADED	Built-in support for multithreading (parallel execution)
PORTABLE	Code written on one system can run on any other with JVM
HIGH PERFORMANCE	Just-In-Time (JIT) compiler improves performance
DISTRIBUTED	Supports networking and remote method invocation (RMI)

1.1.4 How Java Works – Execution Flow

🔄 Compilation and Execution Flow:

1. Write code in .java file.
2. Compile with javac → generates .class file (bytecode).
3. Execute with java → uses **JVM** to interpret bytecode.

```
javac HelloWorld.java         # Compile
java HelloWorld               # Run
```

1.1.5 Java Architecture

Source Code (.java)

↓

Compiler (javac)

↓

Bytecode (.class)

↓

JVM (Java Virtual Machine)

↓

Operating System

- **JDK (Java Development Kit)**: Includes compiler, JRE, and development tools.
- **JRE (Java Runtime Environment)**: Includes JVM + libraries to run Java programs.
- **JVM (Java Virtual Machine)**: Runs the bytecode on the system.

1.1.6 Setting Up Java Development Environment (with IDE)

Step 1: Install Java JDK

1. **Download JDK:**
   - Visit: https://www.oracle.com/java/technologies/javase-downloads.html
   - Choose the correct version for your OS (Windows, Mac, Linux).
2. **Install JDK:**
   - Follow on-screen instructions.
   - After installation, set up environment variables:
     - Add path to JDK's bin directory in PATH
     - Set JAVA_HOME environment variable
3. **Verify Installation:**
   Open Command Prompt or Terminal:

```
java -version

javac -version
```

Step 2: Choose and Install an IDE

An **IDE** provides features like code suggestion, debugging, and project management. Recommended IDEs:

IDE	FEATURES	DOWNLOAD LINK
INTELLIJ IDEA (COMMUNITY EDITION)	Smart code assistance, debugging, refactoring	https://www.jetbrains.com/idea/download/
ECLIPSE IDE	Highly customizable, plugin-based, suitable for enterprise	https://www.eclipse.org/downloads/
NETBEANS IDE	Simple and official Apache IDE with built-in GUI designer	https://netbeans.apache.org/download/index.html
VS CODE **(WITH JAVA EXTENSIONS)**	Lightweight editor with powerful extensions	https://code.visualstudio.com/

Sample Setup: IntelliJ IDEA

1. Install IntelliJ IDEA Community Edition.
2. Open IntelliJ → Create New Project → Choose **Java** SDK.
3. Write your Java program (e.g., HelloWorld).
4. Click **Run** (green triangle icon) or right-click the file → Run.

Advantages of Using an IDE:

- Auto-completion of code
- Real-time error detection
- Integrated debugger
- Easy project and file management
- One-click compile and run

1.1.7 First Java Program

```
// HelloWorld.java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, Java!");
    }
}
```

Explanation:

LINE	DESCRIPTION
`public class helloworld`	Declares a class
`public static void main(string[] args)`	Main method — entry point
`system.out.println()`	Prints output to console

1.1.9 Java Editions

EDITION	USE
JAVA SE (STANDARD EDITION)	Core language, desktop apps
JAVA EE (ENTERPRISE EDITION)	Web, distributed enterprise applications
JAVA ME (MICRO EDITION)	Embedded and mobile devices
JAVAFX	Rich GUI applications

1.1.10 Platform Independence

What is Platform Independence?

Platform independence means that the **same Java program** can run on **any operating system (OS)** without modification.

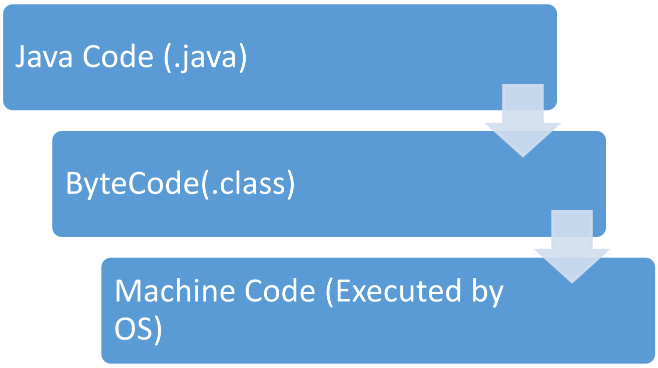
"Write Once, Run Anywhere" (WORA) — Java's core principle.

How is Java Platform Independent?

The Role of the Java Virtual Machine (JVM)

- Java source code (.java) is **compiled into bytecode** (.class file) by the **Java Compiler** (javac).
- This bytecode is **not tied to any specific OS**.
- Instead, it runs on the **JVM**, which is available for all major platforms (Windows, Linux, Mac, etc.).

Flow:



The **JVM** acts as an **interpreter** between your program and the operating system.

Analogy: Universal Charger Adapter

Imagine:

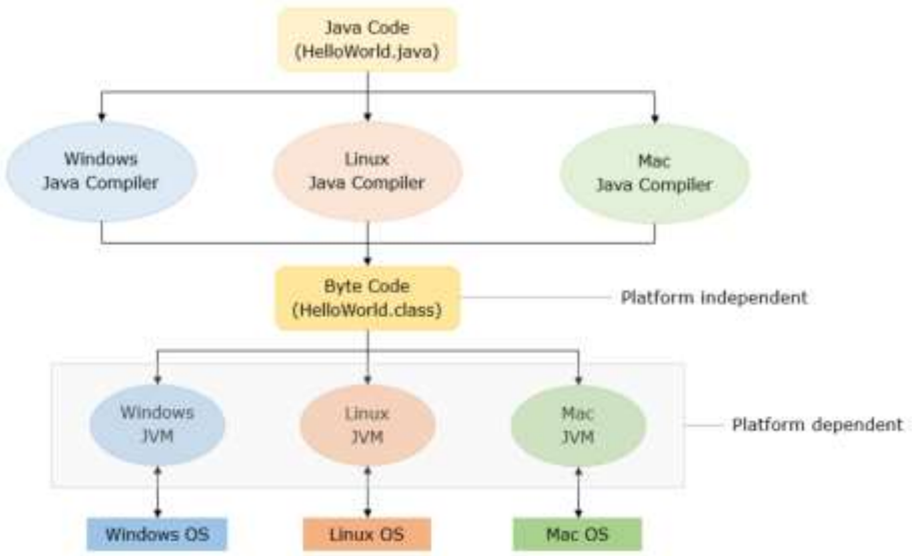
- Your **Java program** is like a **universal device charger**.
- The **bytecode** is like the standard **USB cable**.
- The **JVM** is like a **plug adapter** for each country's socket (India, US, UK).
- You can use the same charger (code) in any country (OS) — you just need the correct adapter (JVM).

Why Platform Independence is Important

- Saves time and effort in developing software for multiple platforms.
- Makes Java suitable for distributed systems, web applications, and mobile development (like Android).

Summary

FEATURE	DESCRIPTION
COMPILATION	Java code → Bytecode using `javac`
EXECUTION	Bytecode → Machine code using JVM
PLATFORM INDEPENDENCE	Same `.class` file works on any OS with appropriate JVM
CORE ENABLER	Java Virtual Machine (JVM)
BENEFIT	"Write Once, Run Anywhere" philosophy



1.1.11 Practice Activities – Java Console Output

Basic Output with `System.out.println()`

1. Print **"Hello, World!"**

   ➤ *Expected output:* `Hello, World!`

2. Print your **name and age** on separate lines

   ➤ *Expected output:*

   ```
   Name: John
   Age: 21
   ```

3. Print 5 lines, each saying:

   ➤ `This is line X` (Replace X with 1 to 5)

Understanding `System.out.print()`

4. Print `Java` and `Programming` on the **same line** using two print statements.

```
System.out.print("Java ");
System.out.print("Programming");
```

   ➤ *Expected output:* `Java Programming`

5. Use a mix of `print()` and println() to print:

```
Welcome
to Java
Programming!
```

Using `System.out.printf()`

6. Print your **name** and a **percentage** in formatted style:

```
String name = "Alice";
double percentage = 92.75;
System.out.printf("Name: %s, Score: %.2f%%\n", name, percentage);
```

➤ *Expected output:* `Name: Alice, Score: 92.75%`

7. Display formatted table-like output for 3 students:

```
System.out.printf("%-10s %-5s %-5s\n", "Name", "Age", "Grade");
System.out.printf("%-10s %-5d %-5c\n", "Ravi", 20, 'A');
System.out.printf("%-10s %-5d %-5c\n", "Meera", 19, 'B');
System.out.printf("%-10s %-5d %-5c\n", "John", 21, 'A');
```

➤ *Expected output:*

```
Name       Age   Grade
Ravi       20    A
Meera      19    B
John       21    A
```

Challenging Output Tasks

8. Print a box shape using `*` symbol:

```
*****
*   *
*   *
*****
```

9. Print a triangle using numbers:

```
1
12
123
1234
```

1.2.1 What is a Variable?

A **variable** is a name given to a memory location that stores data. It acts as a container for storing values during the execution of a program.

Analogy: Think of a variable as a labeled jar that holds a value.

```
int age = 25;  // 'age' is a variable storing an integer value
```

1.2.2 Java Data Types

Java has ***two*** broad categories of data types:

1. Primitive Data Types

DATA TYPE	SIZE	DESCRIPTION
BYTE	1 byte	Small integer (-128 to 127)
SHORT	2 bytes	Larger than byte
INT	4 bytes	Default integer type
LONG	8 bytes	Very large integers
FLOAT	4 bytes	Decimal numbers (7 digits precision)
DOUBLE	8 bytes	Decimal (15 digits precision)
CHAR	2 bytes	Single character (e.g., 'A')
BOOLEAN	1 bit	true or false

Note: **int**, **double**, and **boolean** are most commonly used.

2. Non-Primitive (Reference) Data Types

- Examples: String, Array, Class, Interface, Object

```
String name = "Alice";
int[] marks = {85, 90, 75};
```

1.2.3 Declaring and Initializing Variables

Syntax:

<data_type> <variable_name> = <value>;

Examples:

```
int age = 21;
float percentage = 89.5f;
char grade = 'A';
boolean passed = true;
String name = "Ravi";
```

Use `f` or `F` with float and `L` with long.

```
float height = 5.8f;
long population = 7000000000L;
```

1.2.4 Default Values (for class-level variables only)

DATA TYPE	DEFAULT VALUE
BYTE	0
INT	0
FLOAT	0.0
BOOLEAN	false
CHAR	'\u0000'
OBJECT	null

Local variables **must be initialized** before use.

1.2.5 Rules for Naming Variables

Valid names:

- Must start with a letter (A-Z or a-z), $, or _
- Can contain digits after the first character
- Cannot use **keywords** (like `int`, `class`, `public`)
- Case-sensitive

Invalid:

```
int 1value;     // starts with number ✕
int class;      // keyword ✕
```

Valid:

```
int value1;
int $amount;
int _count;
```

1.2.6 Type Inference (from Java 10+ using `var`)

You can let the compiler infer the data type:

```
var number = 10;        // int
var name = "Java";      // String
var marks = 88.5;       // double
```

Not allowed without initialization:

```
// var x; ✕ Invalid
```

Code Examples

```
public class VariableExample {
    public static void main(String[] args) {
        int age = 22;
        double salary = 55000.75;
        char grade = 'A';
        boolean isEmployee = true;

        System.out.println("Age: " + age);
        System.out.println("Salary: ₹" + salary);
        System.out.println("Grade: " + grade);
        System.out.println("Employee? " + isEmployee);
    }
}
```

1.2.7 Practice Exercises

1. Declare variables of all primitive types and print their values.
2. Create a program to store and display student details: name, age, grade, and marks.
3. Try using invalid variable names and fix the errors.
4. Use `var` to declare different types and print their values.
5. Print the size ranges of `byte`, `short`, `int`, and `long` using constants from `Byte.MIN_VALUE`, etc.

1.3.1 What is Type Casting?

Type casting is the process of converting a variable from one data type to another.

- **Implicit Casting (Widening):** Smaller to larger type – done automatically
- **Explicit Casting (Narrowing):** Larger to smaller type – done manually

Implicit Type Casting (Widening Conversion)

Done **automatically** when there is **no risk of data loss**.

Example:

```
int a = 100;
long b = a;         // int → long
float c = b;        // long → float
System.out.println(c);
```

From → To	Allowed
byte → short → int → long → float → double	Yes

Explicit Type Casting (Narrowing Conversion)

Done **manually**, might cause **data loss** or **precision loss**.

Syntax:

<dataType> variableName = (dataType) value;

Example:

```
double d = 9.8;
int i = (int) d;    // Decimal part will be truncated
System.out.println(i);  // Output: 9
```

Type Casting Between Numeric Types

FROM TYPE	TO TYPE	SAFE?	METHOD
INT TO FLOAT		Implicit	
FLOAT TO INT	⚠	Explicit (possible precision loss)	
LONG TO INT	⚠	Explicit (possible data loss)	
CHAR TO INT		Implicit	
INT TO CHAR	/⚠	Depends on range	

Example:

```
char ch = 'A';        // Unicode = 65
int num = ch;         // Implicit
System.out.println(num);  // 65

int x = 66;
char c = (char) x;     // Explicit
System.out.println(c);  // B
```

Type Conversion Rules

1. Only **compatible types** can be converted.
2. Automatic conversion happens only **when no data is lost**.
3. Casting between boolean and other types is **not allowed**.
4. You must cast explicitly when:
    - Going from larger to smaller type
    - Converting floating point to integer

Precision Loss Example

```
double pi = 3.14159;
int approx = (int) pi;
System.out.println("Pi as int: " + approx);  // Output: 3
```

Decimal part lost: 0.14159

Type Casting in Non-Primitive Types (briefly)

- Only allowed when there's a parent-child relationship between classes.

```
Animal a = new Dog();  // Upcasting (automatic)
Dog d = (Dog) a;       // Downcasting (must be done explicitly)
```

This is covered in detail under **OOP - Inheritance and Polymorphism**

Practice Activities

1. Write a program to demonstrate **implicit** type casting from int → float → double.
2. Convert a double salary to an int and display both.
3. Print ASCII value of a character using casting.
4. Convert an integer to a char and print the result.
5. Try converting boolean to int and explain the error.
6. Show the difference in output between:

```
System.out.println((int) 7.9);
System.out.println((float) 7);
System.out.println((double) 7/2);
```

1.4.1 What is an Operator?

An **operator** is a symbol that performs an operation on one or more operands (values/variables).

```
int a = 5 + 3;  // '+' is an operator
```

1.4.2 Types of Operators

CATEGORY	OPERATORS
ARITHMETIC	+, -, *, /, %
RELATIONAL	==, !=, >, <, >=, <=
LOGICAL	&&, `
ASSIGNMENT	=, +=, -=, *=, /=, %=
UNARY	+, -, ++, --, !
BITWISE	&, `
TERNARY	condition ? true : false

Arithmetic Operators

OPERATOR	MEANING	EXAMPLE	RESULT
+	Addition	5 + 2	7
-	Subtraction	5 - 2	3
*****	Multiplication	5 * 2	10
/	Division	5 / 2	2 (int)
%	Modulus	5 % 2	1

Note: Integer division discards decimal part.

Relational Operators

Used to compare two values (returns true or false)

OPERATOR	MEANING	EXAMPLE
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater or equal	a >= b
<=	Less or equal	a <= b

Logical Operators

Used to combine **boolean** expressions.

OPERATOR	MEANING	EXAMPLE
&&	Logical AND	a > 5 && b < 10
!	Logical NOT	!(a > 5)

Assignment Operators

OPERATOR	MEANING	EXAMPLE
=	Assign	x = 5
+=	Add and assign	x += 3 → x = x + 3
-=	Subtract and assign	x -= 2
*=	Multiply and assign	x *= 2
/=	Divide and assign	x /= 3
%=	Modulo and assign	x %= 2

Unary Operators

OPERATOR	MEANING	EXAMPLE
+	Unary plus	+a
-	Unary minus	-a
++	Increment	a++, ++a
--	Decrement	a--, --a
!	Logical complement	!true → false

Prefix vs Postfix:

```
int a = 5;
System.out.println(++a);  // 6 (prefix: increment before use)
System.out.println(a++);  // 6 (postfix: use before increment)
System.out.println(a);    // 7
```

Ternary Operator

A **shortcut** for if-else:

```
String result = (marks >= 50) ? "Pass" : "Fail";
```

Bitwise Operators (for integers)

OPERATOR	**MEANING**	**EXAMPLE**
&	AND	a & b
|	|	OR
^	XOR	a ^ b
~	NOT	~a
<<	Left shift	a << 2
>>	Right shift	a >> 2

Operator Precedence (Simplified)

PRIORITY	OPERATORS
HIGH	(), ++, --, !
MEDIUM	*, /, %
LOWER	+, -
LOWER	>, <, >=, <=
LOWER	==, !=
LOWER	&&, `
LOWEST	=, +=, -= (assignment)

1.4.3 Practice Activities

1. Write a program that performs **all arithmetic operations** on two integers.
2. Create a calculator that accepts two values and an operator (+, -, *, /) using if or switch.
3. Use logical operators to check if a number is between 10 and 100.
4. Use a ternary operator to determine if a number is even or odd.
5. Demonstrate the difference between a++ and ++a.

1.5.1 What are Control Flow Statements?

- Control flow statements **control the order of execution** of statements in a program.
- They allow you to make **decisions** (branching) or **repeat actions** (loops).

1.5.2 if Statement

Used when you want to execute a block **only if a condition is true**.

```
if (condition) {
    // code runs if condition is true
}
```

Example:

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

1.5.3 if-else Statement

Executes one block if condition is true, another if false.

```
if (condition) {
  // if true
} else {
  // if false
}
```

Example:

```
int number = 7;
if (number % 2 == 0) {
    System.out.println("Even");
} else {
    System.out.println("Odd");
}
```

1.5.4 if-else-if Ladder

Used when checking multiple conditions:

```
if (condition1) {
    // code
} else if (condition2) {
    // code
} else {
    // default
}
```

Example:

```
int marks = 85;
if (marks >= 90) {
    System.out.println("Grade A");
} else if (marks >= 75) {
    System.out.println("Grade B");
} else {
    System.out.println("Grade C");
}
```

1.5.5 Nested if Statements

Placing an if inside another if.

```
if (condition1) {
    if (condition2) {
        // code
    }
}
```

Example:

```
int age = 25;
boolean hasID = true;

if (age >= 18) {
    if (hasID) {
        System.out.println("Access granted");
    } else {
        System.out.println("ID required");
    }
}
```

1.5.6 switch Statement

A cleaner alternative to multiple if-else for discrete values.

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

Example:

```
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");
}
```

- break is used to stop further case execution.
- default is optional but recommended.

1.5.7 Enhanced switch (Java 14+)

```
String day = "TUESDAY";

switch (day) {
    case "MONDAY"    -> System.out.println("Start of week");
    case "TUESDAY"   -> System.out.println("Work day");
    default          -> System.out.println("Another day");
}
```

1.5.8 Practice Activities

1. Write a program to check if a number is **positive, negative, or zero** using if-else-if.
2. Use a switch statement to print the day of the week for a number (1–7).
3. Accept marks from the user and assign grade using if-else-if.
4. Write a nested if program to check if a user can register for a service (age ≥ 18, has ID).
5. Create a switch that prints month names based on user input (1 to 12).

1.6.1 What are Loops?

Loops allow us to **repeat a block of code** multiple times.

1.6.2 for Loop

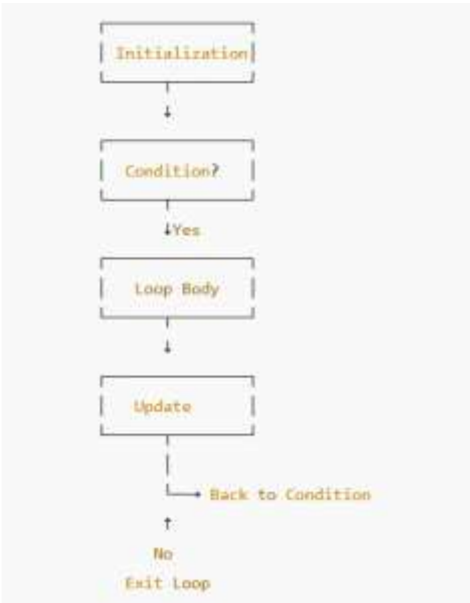
Used when you know **exactly how many times** to loop.

```
for (initialization; condition; update) {
    // code to be repeated
}
```

Example:

```
for (int i = 1; i <= 5; i++) {
    System.out.println("Hello " + i);
}
```

Flow Chart



1.6.3 while Loop

Used when the **number of iterations is not known** beforehand.

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

Example:

```
int i = 1;
while (i <= 5) {
    System.out.println("Hi");
    i++;
}
```

Flow Chart

```
┌────────────────┐
│  Condition?    │
└───────┬────────┘
        ↓Yes
┌────────────────┐
│   Loop Body    │
└───────┬────────┘
        ↓
└──→ Back to Condition
        ↑
       No
   Exit Loop
```

1.6.4 do-while Loop

Similar to while, but it **executes at least once**, even if condition is false.

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

Example:

```
int i = 1;
do {
    System.out.println("Welcome");
    i++;
} while (i <= 3);
```

Flow Chart

```
┌────────────────┐
│   Loop Body    │
└──────┬─────────┘
       ↓
┌────────────────┐
│  Condition?    │
└──────┬─────────┘
       ↓Yes
└──→ Back to Loop Body
       ↑
      No
     Exit Loop
```

1.6.5 Loop Comparison

FEATURE	FOR	WHILE	DO-WHILE
INITIALIZATION	Inside loop line	Outside loop	Outside loop
CONDITION	Checked first	Checked first	Checked after loop
EXECUTION	0 or more times	0 or more times	1 or more times
USE CASE	Known counts	Unknown counts	At least once

1.6.6 `break` and `continue`

- `break`: Exit the loop entirely.
- `continue`: Skip the current iteration and go to the next one.

Example (break):

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) break;
    System.out.println(i);  // prints 1, 2
}
```

Example (continue):

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) continue;
    System.out.println(i);  // prints 1, 2, 4, 5
}
```

1.6.7 Nested Loops

Loops inside loops.

Example:

```
for (int i = 1; i <= 5; i++) {
    for (int j = 1; j <= 5; j++) {
        System.out.println("i=" + i + ", j=" + j);
    }
}
```

1.6.8 Practice Activities

1. Print numbers from 1 to 10 using all three loops.
2. Print even numbers between 1 and 50 using `for` loop.
3. Print the multiplication table of a number (e.g., 5).
4. Create a `do-while` loop that runs at least once even if condition is false.
5. Use nested `for` loops to print a pattern:

```
*
* *
* * *
```

6. Write a loop that breaks when a user inputs `0`.
7. Use `continue` to skip printing multiples of 3 in a loop from 1 to 20.

1.7.1 What is a Method?

A **method** is a block of code that performs a specific task.
It helps in:

- Code reuse
- Modularity
- Better structure and readability

Defining a Method

```
returnType methodName(parameters) {
    // method body
    return value; // optional
}
```

Example:

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

Calling a Method

You call a method using its name followed by parentheses.

```
int result = add(10, 20);
System.out.println(result); // Output: 30
```

`void` vs Return Type

- `void`: Method does **not return** anything.
- Return types (`int`, `String`, etc.): Method **returns a value**.

Example (void):

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

Example (return type):

```
String getName() {
    return "Java";
}
```

Method with Parameters

You can pass input values using parameters.

```
void sayHello(String name) {
    System.out.println("Hello, " + name);
}
```

Calling:

```
sayHello("Alice"); // Output: Hello, Alice
```

Local Variables in Java Methods

What is a Local Variable?

A **local variable** is a variable **declared inside a method**, **constructor**, or **block**, and it **exists only within that block**.

Once the method finishes execution, the local variable is **destroyed**.

Key Features of Local Variables

FEATURE	DESCRIPTION
SCOPE	Only within the method or block where it's declared
LIFETIME	Exists only while the method is executing
INITIALIZATION REQUIREMENT	Must be initialized **before use**
MEMORY LOCATION	Stored in the stack memory

Example

```
public class LocalVariableExample {

    public void displaySum() {
        int a = 10;  // local variable
        int b = 20;  // local variable
        int sum = a + b;  // local variable
        System.out.println("Sum: " + sum);
    }

    public void showName() {
        String name = "Java"; // local variable
```

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

Variables `a`, `b`, `sum`, and `name` are **only usable inside their respective methods**.

Best Practices

- Use **local variables** for temporary storage or calculations.
- Always **initialize** them before use.
- Prefer **local scope** to reduce memory usage and increase readability.

Unit 2: Object-Oriented Programming – I

2.1.1 Classes and Objects

What is a Class?

A **class** is a user-defined data type that serves as a **blueprint** for creating objects. It contains:

- **Fields** (also called attributes or properties) → store data
- **Methods** → define behavior (functions inside a class)

Think of a class as a **template**, and an object as a **real-world instance** of that template.

Real-Life Analogy:

- **Class** → `Car` (definition of what a car is)
- **Object** → `car1`, `car2` (specific cars like Honda City, BMW)

Syntax of a Class in Java:

```
class ClassName {
    // Fields (data)
    dataType variableName;

    // Methods (behavior)
    returnType methodName(parameters) {
        // code
    }
}
```

Example: Defining a Class

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

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

This class:

- Has two fields: `name` and `age`
- Has one method: `displayInfo()`

2.1.1 What is an Object?

An **object** is a runtime entity created from a class using the `new` keyword.

Each object has:

- Its own copy of the class fields
- The ability to use methods defined in the class

Mini Activity 1:

Think of 3 real-world entities you can model using classes and objects (e.g., `Book`, `Employee`, `MobilePhone`). What fields and methods would you include?

2.2.1 Defining and Instantiating Classes

Instantiating (Creating) an Object:

```
ClassName obj = new ClassName();
```

Full Example with Object:

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

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

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();  // Object creation
        s1.name = "Alice";
        s1.age = 21;
        s1.displayInfo();  // Method call
    }
}
```

Output:

Name: Alice
Age: 21

2.2.2 Object vs Class (Summary):

CONCEPT	CLASS	OBJECT
TYPE	Blueprint / definition	Instance of a class
CREATED	At compile time	At runtime using `new`
MEMORY	No memory unless instantiated	Takes memory (fields & refs)

2.2.3 Practice Activities:

1. Create a class `Book` with fields `title`, `author`, `price`. Write a method `displayBook()`.
2. Write a program to create 2 objects of class `Car` and assign different values.
3. Create a class `Employee` and display their `name` and `salary`.
4. Try creating a `Person` object without setting values. What is the default value of `String` and `int`?

2.3.1 Constructors in Java

Analogy:

Imagine you're assembling a **new phone** in a factory:

- When the phone (object) is created, the factory sets its initial configuration (model, battery, screen).
- This "setup process" is like a **constructor**—it **initializes the object** with meaningful default or passed values.

What is a Constructor?

A **constructor** is a special method in Java that:

- **Has the same name** as the class.
- **Has no return type** (not even `void`).
- **Is automatically called** when an object is created using `new`.

Why Use Constructors?

- To **initialize** objects at the time of creation.
- To **avoid calling a separate method** after object creation just to assign values.

Syntax of a Constructor

```
class ClassName {
    ClassName() {
        // initialization code
    }
}
```

2.3.2 Default Constructor

```
class Student {
    Student() {
        System.out.println("Constructor called");
    }
}
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student(); // Constructor called automatically
    }
}
```

Output:

Constructor called

This is called a **default (no-argument) constructor**.

2.3.3 Parameterized Constructor

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

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

    void display() {
        System.out.println(name + " - " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("John", 20);
        s1.display();
    }
}
```

Output:

John - 20

2.3.4 Constructor Overloading

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

Example:

```
class Rectangle {
    int length, width;

    Rectangle() {
        length = 0;
        width = 0;
    }

    Rectangle(int l, int w) {
        length = l;
        width = w;
    }

    void display() {
        System.out.println("Area: " + (length * width));
    }
}
```

Usage:

```
Rectangle r1 = new Rectangle();       // Area: 0
Rectangle r2 = new Rectangle(5, 3);   // Area: 15
```

Real-world Analogy for Constructor Overloading

Think of **pizza ordering**:

- You can order a **default pizza** (no toppings, small size).
- Or, you can **customize** it (extra cheese, medium, mushrooms).

In code, you have:

- Pizza() → default constructor
- Pizza(String size, String toppings) → parameterized constructor

2.3.5 Summary Table

FEATURE	DESCRIPTION
CONSTRUCTOR NAME	Same as class name
RETURN TYPE	None (not even `void`)
WHEN CALLED	Automatically at object creation
CAN BE OVERLOADED	Yes
CAN TAKE PARAMETERS	Yes (Parameterized Constructor)
CAN INITIALIZE VARIABLES	Yes

2.3.6 Practice Activities

1. Create a `Car` class with fields `brand`, `price`. Use a constructor to initialize and a method to display.
2. Overload the constructor in a `Box` class: one with no dimensions, one with `length`, `breadth`, `height`.
3. Write a class `Account` with `name` and `balance`, and use constructor overloading for different account types.
4. Create a class `Laptop` and use the constructor to auto-set `brand` and `RAM` size.

2.4.1 Method Overloading in Java

What is Method Overloading?

Method Overloading is a feature in Java that allows a class to have **more than one method with the same name** but **different parameters** (type, number, or order).

It's a way of performing **polymorphism** (compile-time / static polymorphism).

Analogy: Multiple Contact Numbers for the Same Person

Imagine you save a contact as **"Mom"** in your phone:

- "Mom (Mobile)"
- "Mom (Work)"
- "Mom (Home)"

All entries have the **same name**, but different **numbers** (parameters).
When you call "Mom", your phone chooses the correct number based on the **context**.

Similarly, in Java:

- Same method name
- Different parameter list
- Java decides **which version** to run based on the arguments passed.

Example of Method Overloading

```
public class Calculator {

    // Method 1: Add two integers
    public int add(int a, int b) {
        return a + b;
    }

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

    // Method 3: Add two doubles
    public double add(double a, double b) {
        return a + b;
    }
}
```

Usage:

```
Calculator calc = new Calculator();


System.out.println(calc.add(10, 20));          // Output: 30
System.out.println(calc.add(10, 20, 30));      // Output: 60
System.out.println(calc.add(5.5, 3.3));        // Output: 8.8
```

➞ All methods are named `add`, but **Java differentiates them** based on **number and type of arguments**.

How Java Resolves Overloaded Methods

Java compiler looks at:

1. **Number of parameters**
2. **Type of parameters**
3. **Order of parameters**

Rules of Method Overloading

RULE	DESCRIPTION
✓	Must have **same method name**
✓	Must have **different parameter list**
✗	Changing only **return type** does NOT count as overloading
✗	Changing only **access modifiers or static/non-static** is NOT valid

Invalid Overload Example:

```
public int show() { return 1; }
public double show() { return 1.0; } // ✗ Compile-time error
```

Use Cases

- Creating flexible APIs
- Performing similar operations with different input types
- Simplifying code readability and structure

Quick Summary Table

FEATURE	DESCRIPTION
WHAT	Same method name, different signatures
TYPE	Compile-time polymorphism
PARAMETER VARIATIONS	Number, type, or order
RETURN TYPE ALONE	Cannot be used to overload
USE CASE	Code readability and method flexibility

2.5.1 static Keyword in Java

The static keyword in Java is used for **memory management**. It is applied to:

- Variables (static variables)
- Methods (static methods)
- Blocks (static blocks)
- Nested classes (static nested classes)

The static keyword tells Java: **"This belongs to the class, not to a specific object."**

Analogy: Common Notice Board in a School

In a school:

- Every **student (object)** has their **own notebook (instance variable)**.
- But the **common notice board (static variable)** is **shared by all students**.

Anything marked static in Java is like that **notice board** — **shared and common** to all.

2.5.2. Static Variable (Class Variable)

Definition:

A variable declared with static inside a class is **shared among all instances** of that class.

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

Usage:

```
Student s1 = new Student();
Student s2 = new Student();

System.out.println(s1.college); // ABC College
System.out.println(s2.college); // ABC College
```

Change it in one place → reflects for all:

```
Student.college = "XYZ College";
```

2.5.3 Static Method

Definition:

A method marked static can be **called without creating an object** of the class.

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

Usage:

```
int result = MathUtil.square(5); // No object needed
System.out.println(result);      // 25
```

Rules of Static Methods:

RULE	EXPLANATION
✓	Can access **only static data** directly
✓	Can call **only static methods** directly
✗	Cannot use this or super
✗	Cannot access instance variables/methods directly

2.5.4. Static Block

Definition:

A static block is used to **initialize static variables**. It executes **once** when the class is loaded.

```
class Config {
    static int version;
    static {
        version = 1;
        System.out.println("Static block executed");
    }
}
```

Usage:

```
public class Test {
    public static void main(String[] args) {
        System.out.println(Config.version); // Triggers static block once
    }
}
```

2.5.5 Static Nested Class

You can declare a **class inside another class** using static.

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

Usage:

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

Unlike non-static inner classes, static nested classes **don't require an object** of the outer class.

2.5.6 Why Use static?

- Memory-efficient: one copy for all instances.
- Easy access: no object needed.
- Useful for **constants**, **utility methods**, **configuration data**, etc.

2.5.7 Quick Recap Table

ELEMENT	STATIC USE & MEANING
STATIC VARIABLE	One copy shared by all instances
STATIC METHOD	Called without object; can't access instance members
STATIC BLOCK	Initializes static members; runs once when class is loaded
STATIC CLASS	Inner class that doesn't depend on outer class instance

Example Combining All

```
public class Example {
    static int count;

    static {
        count = 10;
        System.out.println("Static block run");
    }

    static void showCount() {
        System.out.println("Count: " + count);
    }

    static class Helper {
        void help() {
            System.out.println("Helping...");
        }
    }

    public static void main(String[] args) {
        Example.showCount();
        Example.Helper h = new Example.Helper();
        h.help();
    }
}
```

2.6.1 this Keyword in Java

What is `this` in Java?

`this` is a **reference variable** in Java that refers to the **current object** — the object on which a method or constructor is being called.

Think of `this` as **"myself"** for an object.

Analogy: Self-Introduction

In a classroom:

- A student says: "**Hi, I am Rahul**."
- Here, "**I**" refers to **Rahul himself**.

In Java:

- An object can refer to itself using `this`.

2.6.2 Why Use `this` Keyword?

1. To **refer to current class instance variables**
2. To **invoke current class methods or constructors**
3. To **pass the current object as a parameter**
4. To **return the current class object**

Differentiate Between Instance and Local Variables

Problem:

When constructor parameters **have the same name** as instance variables.

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

    Student(int id, String name) {
        id = id;        // ✕ does NOT assign to instance variable
        name = name;    // ✕ same here
    }
}
```

Solution:

Use this to refer to instance variables.

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

    Student(int id, String name) {
        this.id = id;         // now it's clear
        this.name = name;
    }
}
```

this.id refers to the **instance variable**,
id refers to the **parameter**.

To Invoke Current Class Method

```
class Demo {
    void display() {
        System.out.println("Display method called");
    }

    void show() {
        this.display(); // Calls display() of this object
    }
}
```

To Invoke Constructor from Another Constructor

This is called **constructor chaining** using this().

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

    Car() {
        this("Unknown", 0);  // calling parameterized constructor
    }

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

Must be the **first statement** in the constructor.

To Pass Current Object as a Parameter

```
class Printer {
    void print(Student s) {
        System.out.println("Printing student: " + s.name);
    }
}

class Student {
    String name = "Amit";

    void show() {
        Printer p = new Printer();
        p.print(this); // passing current object
    }
}
```

To Return Current Object

Useful for **method chaining**.

```
class Person {
    Person getObject() {
        return this;
    }
}
```

2.6.3 Key Points About this

USE CASE	DESCRIPTION
REFERRING INSTANCE VARIABLE	When local and instance variable names clash
INVOKING INSTANCE METHOD	Call another method of the same class
CALLING ANOTHER CONSTRUCTOR	To chain constructors
PASS CURRENT OBJECT	Pass this to another class or method
RETURN CURRENT OBJECT	Enable method chaining

2.6.4 Summary Table

CONTEXT	EXAMPLE	MEANING
THIS.VARIABLE	this.name = name;	Refers to current object's variable
THIS.METHOD()	this.display();	Calls method on current object
THIS()	this("car", 2020);	Calls another constructor in same class
AS PARAMETER	p.print(this);	Passes current object
AS RETURN	return this;	Returns the current object

2.6.4 Real-Life Example: Method Chaining

```
class Builder {
    Builder start() {
        System.out.println("Started");
        return this;
    }

    Builder build() {
        System.out.println("Building...");
        return this;
    }

    Builder end() {
        System.out.println("Finished");
        return this;
    }
}

public class Main {
    public static void main(String[] args) {
        new Builder().start().build().end();  // Chaining using this
    }
}
```

2.7.1 Access Modifiers in Java

What Are Access Modifiers?

Access Modifiers in Java **define the visibility/scope** of:

- Classes
- Variables
- Methods
- Constructors

They control **who can access what** in your code.

Analogy: House Rooms and Keys

Imagine a house:

- **Private Room** – only you can enter.
- **Default Room** – only people inside the house (package) can enter.
- **Protected Room** – family (subclass) and housemates (package) can enter.
- **Public Room** – anyone can enter.

In Java, these levels are:

private Access Modifier

- Accessible **only within the same class**
- Not accessible outside the class, not even in subclasses

```
class Account {
    private double balance = 5000;

    private void showBalance() {
        System.out.println("Balance: " + balance);
    }
}
```

❌ Cannot access balance from another class directly.

(default) — No Modifier

- Also called **package-private**
- Accessible **within the same package only**

```
class Employee {
    int empId = 101; // default
    void show() {
        System.out.println("Employee ID: " + empId);
    }
}
```

❌ Cannot access from a class in a **different package**.

protected Access Modifier

- Accessible:
  - Within the **same package**
  - In **subclasses**, even if they are in a **different package**

```
class Person {
    protected String name = "John";
}

class Student extends Person {
    void display() {
        System.out.println("Name: " + name); // Accessible in subclass
    }
}
```

public Access Modifier

- Accessible **from anywhere** — any class, any package

```
public class Calculator {
    public void add(int a, int b) {
        System.out.println("Sum: " + (a + b));
    }
}
```

Can be accessed from other classes or packages.

Comparison Table

MODIFIER	SAME CLASS	SAME PACKAGE	SUBCLASS (DIFFERENT PACKAGE)	OTHER PACKAGES
PRIVATE	✓	✗	✗	✗
(DEFAULT)	✓	✓	✗	✗
PROTECTED	✓	✓	✓	✗
PUBLIC	✓	✓	✓	✓

Access Modifiers with Classes

- **Top-level classes** can only be:
    - public
    - *(default)* (no modifier)

You **cannot declare a top-level class as private or protected.**

2.7.2 Best Practices

USE CASE	SUGGESTED MODIFIER
INTERNAL HELPER METHODS	private
FIELDS (ENCAPSULATION)	private + getters/setters
PUBLIC API METHODS	public
INHERITANCE SUPPORT METHODS	protected
PACKAGE-ONLY UTILITY CLASSES	*(default)*

2.7.3 Example Combining All

```
public class Example {

    private int secret = 123;          // Only this class
    int packageValue = 100;            // Same package
    protected String name = "Java";    // Same package + subclasses
    public void show() {               // Accessible everywhere
        System.out.println("Public method");
    }

    private void privateMethod() {
        System.out.println("Private method");
    }
}
```

2.7.4 Summary Table

ACCESS LEVEL	USE FOR	KEYWORD
PRIVATE	Sensitive data, internal logic	private
DEFAULT	Package-level classes	*(no keyword)*
PROTECTED	Inheritance support	protected
PUBLIC	Public APIs or core features	public

2.8.1 Encapsulation in Java

Definition:

Encapsulation is the **hinding of data (variables)** and the **code (methods)** that operate on the data **into a single unit**, while **restricting direct access** to some components.

It's one of the **four pillars of OOP** (along with inheritance, polymorphism, and abstraction).

Real-world Analogy: Medicine Bottle

Imagine a **medicine bottle**:

- The **medicine (data)** is inside the bottle.
- The **bottle (class)** protects the medicine.
- The **cap (access control)** prevents direct access — you can only get the medicine **through a prescription or controlled dose**.
- You can't reach in and change the ingredients directly.

Like a medicine bottle, **encapsulation controls how data is accessed and modified**, keeping it safe from misuse.

This is **encapsulation** — hiding the internal complexity and providing a clean interface.

How to Achieve Encapsulation in Java

1. **Make variables `private`**
2. **Provide `public` getter and setter methods**

Example:

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

    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
        if (amount > 0)
            balance += amount;
    }

    public void withdraw(double amount) {
```

```
        if (amount > 0 && amount <= balance)
            balance -= amount;
    }
}
```

Usage:

```
BankAccount account = new BankAccount();
account.deposit(1000);
account.withdraw(500);
System.out.println(account.getBalance()); // 500
```

Direct access to balance is not allowed.
Only controlled access via getBalance(), deposit(), withdraw().

Why Use Encapsulation?

REASON	BENEFIT
HIDE IMPLEMENTATION DETAILS	Reduces complexity
SECURE DATA	Prevents unauthorized access
EASY TO MAINTAIN	Internal code changes don't affect external classes
ADDS CONTROL	You can add logic in setters/getters

2.8.2 Best Practices

- Always make fields private
- Provide public getter/setter methods **only if needed**
- Add validation logic inside setters
- Avoid public setters if the field should be **read-only**

2.8.3 Quick Recap Table

TERM	DESCRIPTION
ENCAPSULATION	Binding data and code, hiding data
PRIVATE	Used to restrict direct access
GETTER	Method to read private variable
SETTER	Method to write/update private variable

2.9.1 Introduction to Packages in Java

What is a Package in Java?

A **package** is a **namespace** that groups **related classes and interfaces** together.

Think of a package as a **folder** or **directory** that helps organize your Java files in a logical way.

Real-world Analogy: File Cabinet

Imagine a **file cabinet** in an office:

- Each **drawer** is like a **package**.
- Inside each drawer, you keep **files of a specific type** — e.g., invoices, resumes, reports.

Packages in Java work the same way — they organize your classes so you don't lose track and avoid name conflicts.

Why Use Packages?

BENEFIT	🔍 DESCRIPTION
BETTER ORGANIZATION	Group similar classes (e.g., all GUI classes in one package)
AVOID CLASS NAME CONFLICTS	Two classes with the same name can exist in different packages
CONTROLLED ACCESS	Use access modifiers (public, protected, etc.)
REUSABILITY	Code can be shared and reused across multiple projects

2.9.2 Creating a Package

Step 1: Declare package at the top of the Java file

```
package mypackage;

public class MyClass {
    public void display() {
        System.out.println("Hello from MyClass");
    }
}
```

package must be the **first statement** in the file.

Step 2: Compile with directory structure

javac -d . MyClass.java

- -d . tells the compiler to create the folder structure based on the package name.
- This creates a folder mypackage/ containing MyClass.class.

Step 3: Import and Use the Package

```
import mypackage.MyClass;


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

2.9.3 Types of Packages

TYPE	DESCRIPTION	EXAMPLE
BUILT-IN	Predefined packages in Java API	java.util, java.io
USER-DEFINED	Packages created by the programmer	mypackage, school.student

2.9.4 Common Built-in Packages

PACKAGE	USE
JAVA.LANG	Basic classes (String, Math, etc.)
JAVA.UTIL	Collections, Date, Scanner, etc.
JAVA.IO	File handling
JAVA.SQL	Database connectivity
JAVAX.SWING	GUI programming

2.9.5 Package Structure (Hierarchy)

You can create **sub-packages** using dot (.) notation:

```
package com.company.department;

public class Employee {
    // code here
}
```

This would create a folder path: com/company/department/Employee.class

2.9.6 Access Modifiers in Packages

MODIFIER	ACCESSIBLE WITHIN SAME PACKAGE?	ACCESSIBLE FROM OUTSIDE PACKAGE?
PRIVATE	✗	✗
(DEFAULT)	✓	✗
PROTECTED	✓	(only through subclass)
PUBLIC	✓	✓

Best Practices

- Use company/domain-style package names (e.g., com.techacademy.utils)
- Keep related classes together (e.g., DAO classes in com.app.dao)
- Avoid using default package (no package declaration)
- Use meaningful names (e.g., student.records, billing.invoice)

Example Summary

```
// File: com/example/Hello.java
package com.example;

public class Hello {
    public void greet() {
        System.out.println("Hello from package!");
    }
}

// File: Main.java
import com.example.Hello;

public class Main {
    public static void main(String[] args) {
        Hello h = new Hello();
        h.greet();
    }
}
```

2.10.1 Array in Java

An **array** is a **collection of elements** of the **same data type** stored in **contiguous memory locations**. It is used to store multiple values under a single variable name.

Think of an array as a row of lockers (indexed), where each locker holds a value of the same type.

Syntax

1. Declaration:

```
int[] numbers;       // Recommended
// OR
int numbers[];       // Also valid
```

2. Instantiation:

```
numbers = new int[5];  // Array of 5 integers (default values: 0)
```

3. Initialization:

```
numbers[0] = 10;
numbers[1] = 20;
```

4. Combined:

```
int[] numbers = new int[] {10, 20, 30, 40, 50};
```

OR

```
int[] numbers = {10, 20, 30, 40, 50};
```

Real-world Analogy

Imagine you run a delivery service and assign lockers for packages. Each locker is numbered (indexed). You can access a specific locker (array index) to retrieve the package (value).

Example:

```
public class ArrayDemo {
    public static void main(String[] args) {
        String[] products = {"Laptop", "Tablet", "Mobile"};

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

Output:

Laptop
Tablet
Mobile

Types of Arrays in Java

TYPE	DESCRIPTION	EXAMPLE
SINGLE-DIMENSIONAL	Linear list of elements	`int[] marks = new int[5];`
MULTI-DIMENSIONAL	Array of arrays (matrix-style)	`int[][] matrix = new int[3][3];`
JAGGED ARRAY	Array of arrays with different lengths	`int[][] arr = new int[3][];`

Array Properties

- Fixed size (declared at the time of creation)
- Index starts from 0
- Can store **primitive** or **reference types**
- Default values:
  - `0` for int
  - `false` for boolean
  - `null` for objects

Common Use Cases

- Storing student marks
- Holding a list of products
- Representing 2D data (like a matrix or a chessboard)

Advantages

- Easy to use
- Memory-efficient for fixed-size data
- Fast data access using index

Limitations

- Fixed size – can't grow dynamically (use `ArrayList` instead for dynamic needs)
- All elements must be of the same type
- Insertion/deletion in the middle is expensive (shifting needed)

Best Practices

- Always check `array.length` before iterating to avoid `ArrayIndexOutOfBoundsException`
- For unknown sizes, use **collections** like `ArrayList`
- Use **enhanced for loop** for readability:

```
for (String item : products) {
    System.out.println(item);
}
```

Summary

FEATURE	DESCRIPTION
DEFINITION	Fixed-size container for same-type elements
ACCESS	Via index (starting at 0)
TYPE	Single or multi-dimensional
BETTER ALTERNATIVE (DYNAMIC)	Use ArrayList or other collections

2.10.2 Array Input/Output in Java

- **Array Input**: Taking values from the user and storing them in an array.
- **Array Output**: Displaying the values stored in the array.

Both input and output can be performed using loops like for or for-each.

Real-World Analogy

Think of a row of exam paper slots for students:

- Input: Each student drops their paper into a numbered slot (array index).
- Output: The examiner reads papers from the slots one by one.

Example: Taking Input and Printing Output of an Integer Array

```
import java.util.Scanner;

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

        System.out.print("Enter size of the array: ");
        int size = scanner.nextInt();

        int[] numbers = new int[size];

        // Input
        System.out.println("Enter " + size + " numbers:");
        for (int i = 0; i < size; i++) {
            numbers[i] = scanner.nextInt();
        }
```

```
        // Output
        System.out.println("You entered:");
        for (int i = 0; i < size; i++) {
            System.out.println("Element at index " + i + ": " + numbers[i]);
        }

        scanner.close();
    }
}
```

Sample Output

Enter size of the array: 4
Enter 4 numbers:
10
20
30
40
You entered:
Element at index 0: 10
Element at index 1: 20
Element at index 2: 30
Element at index 3: 40

Using Enhanced For Loop for Output

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

For String Array Input/Output Example

```
String[] names = new String[3];
Scanner sc = new Scanner(System.in);

System.out.println("Enter 3 names:");
for (int i = 0; i < names.length; i++) {
    names[i] = sc.nextLine();
}

System.out.println("Names entered:");
for (String name : names) {
    System.out.println(name);
}
```

Common Mistakes to Avoid

MISTAKE	FIX
arrayindexoutofboundsexception	Always use array.length for loops
forgetting to close scanner	Use scanner.close() at the end
mixing nextint() and nextline()	Use scanner.nextLine() after nextInt() to consume leftover \n

Summary Table

OPERATION	CODE SNIPPET
DECLARE ARRAY	int[] arr = new int[5];
INPUT VALUES	arr[i] = sc.nextInt(); in loop
OUTPUT VALUES	System.out.println(arr[i]); or for-each
DYNAMIC SIZE	int size = sc.nextInt(); then create array

2.10.3 2D Array in Java

A **2D array** is an array of arrays. It stores data in **rows and columns**, like a **matrix or table**.

Definition: A 2D array in Java is declared as: dataType[][] arrayName;

Real-World Analogy

Think of a **spreadsheet** or **chessboard**:

- Rows and columns hold values.
- Each cell is accessed by its row and column number (like [i][j]).

Syntax of 2D Arrays

Declaration:

```
int[][] matrix;          // Recommended
int matrix[][];          // Also valid
```

Instantiation:

```
matrix = new int[3][4];  // 3 rows and 4 columns
```

Initialization:

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

Accessing Elements

```
matrix[0][1];  // Access element at 1st row, 2nd column (value: 2)
```

Taking Input and Printing a 2D Array

```
import java.util.Scanner;

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

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

        // Input
        System.out.println("Enter elements (2 rows, 3 columns):");
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 3; j++) {
                matrix[i][j] = sc.nextInt();
            }
        }

        // Output
        System.out.println("Matrix:");
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 3; j++) {
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();
        }

        sc.close();
    }
}
```

Sample Output:

Enter elements (2 rows, 3 columns):
1 2 3
4 5 6
Matrix:
1 2 3
4 5 6

Enhanced For Loop for Output

```
for (int[] row : matrix) {
    for (int val : row) {
        System.out.print(val + " ");
    }
    System.out.println();
}
```

Common Use Cases of 2D Arrays

USE CASE	EXAMPLE
MATRIX OPERATIONS	Addition, multiplication
TABLES OR GRIDS	Marks of students (rows: students, columns: subjects)
BOARD GAMES	Chess, Sudoku, Tic Tac Toe
IMAGE REPRESENTATION	Pixel values in grayscale

Important Points

- Indexing starts at 0 → [0][0] is first row, first column.
- Default values:
    - 0 for int/float
    - false for boolean
    - null for objects
- matrix.length gives **number of rows**
  matrix[0].length gives **number of columns**

Summary

FEATURE	DESCRIPTION
DECLARATION	int[][] arr = new int[3][4];
INPUT	Nested for loop
OUTPUT	Nested for or for-each loop
USE CASES	Tables, matrices, grids, boards

2.11.1 What is a String in Java

A **String** in Java is a **sequence of characters**, treated as an object of the String class in the java.lang package.

Java Strings are **immutable**, meaning once created, their values **cannot be changed**.

Real-World Analogy

Think of a **string** as a **word written in ink** on paper — once written (created), you can read, compare, or copy it, but **can't change the ink directly** (immutable). To change it, you create a new paper (new String).

String Declaration and Initialization

```
// Using string literal (stored in string pool)
String s1 = "Hello";

// Using new keyword (stored in heap)
String s2 = new String("World");
```

Common String Methods

METHOD	DESCRIPTION	EXAMPLE
LENGTH()	Returns the number of characters	s.length()
CHARAT(INT INDEX)	Returns character at a specific index	s.charAt(1)
TOUPPERCASE()	Converts to uppercase	s.toUpperCase()
TOLOWERCASE()	Converts to lowercase	s.toLowerCase()
EQUALS()	Compares content (case-sensitive)	s1.equals(s2)
EQUALSIGNORECASE()	Compares ignoring case	s1.equalsIgnoreCase(s2)
CONTAINS()	Checks if string contains substring	s.contains("text")
SUBSTRING(START, END)	Extracts substring	s.substring(1, 4)
REPLACE(A, B)	Replaces characters	s.replace("a", "b")
SPLIT(" ")	Splits string into array	s.split(" ")
TRIM()	Removes leading/trailing spaces	s.trim()

String Immutability Explained

```
String s = "Hello";
s.concat(" World");  // does NOT change original string
System.out.println(s);  // Output: Hello
```

To reflect the change:

```
s = s.concat(" World");
System.out.println(s);  // Output: Hello World
```

String Comparison

```
String s1 = "Hello";
String s2 = "Hello";
String s3 = new String("Hello");

System.out.println(s1 == s2);      // true (same object in pool)
System.out.println(s1 == s3);      // false (different object)
System.out.println(s1.equals(s3)); // true (same content)
```

Example Program

```
public class StringExample {
    public static void main(String[] args) {
        String name = "Java Programming";

        System.out.println("Length: " + name.length());
        System.out.println("Upper: " + name.toUpperCase());
        System.out.println("First char: " + name.charAt(0));
        System.out.println("Contains 'Java': " + name.contains("Java"));
    }
}
```

Best Practices

- Prefer string **literals** for memory efficiency.
- Use equals() for comparison, **not** ==.
- Use StringBuilder for heavy string modifications (e.g., in loops).
- Avoid unnecessary string concatenations — it's memory-expensive.

Summary

TOPIC	KEY POINT
IMMUTABLE	Once created, cannot be changed
STORAGE	String Pool (literal), Heap (new)
METHODS	Powerful built-in methods for processing
COMPARISON	equals() for content, == for reference check
USE CASE	Widely used in file I/O, user input, APIs, etc.

2.11.2 String Literal Vs String Object

What is a String Literal?

A **String literal** is any sequence of characters enclosed in double quotes, e.g.:

```
String s1 = "Java";
```

- Stored **in the String Constant Pool (SCP)** inside the **Method Area** of JVM memory.
- If "Java" already exists in the SCP, it **does not create a new object** — it just returns a reference to the existing one.

What is a String Object?

You can also create a String using the `new` keyword:

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

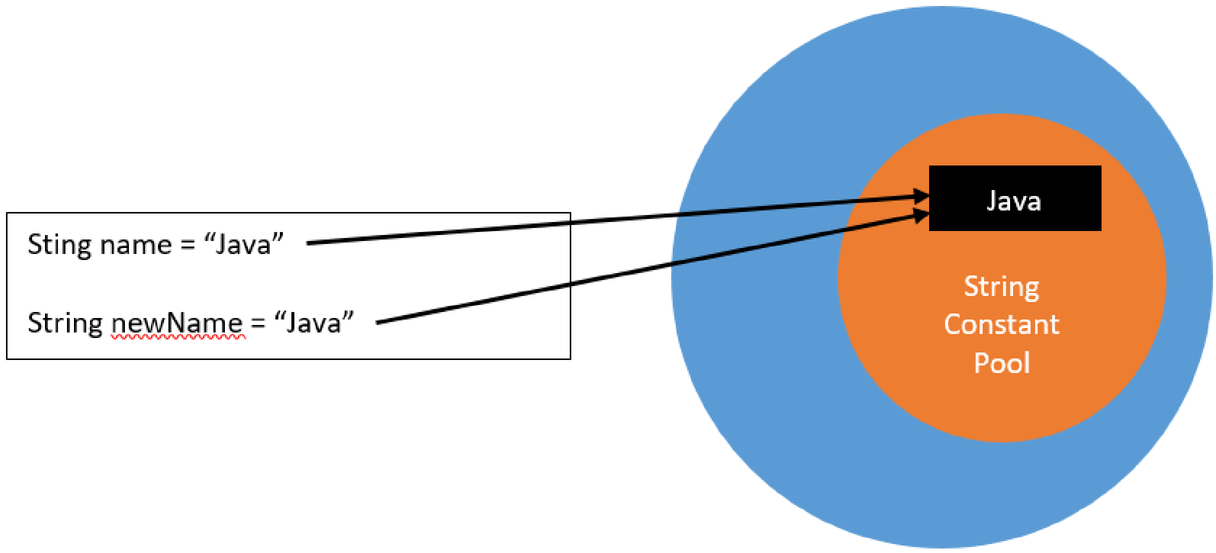
- Creates a **new object in the Heap** memory.
- Also refers to "Java" in the **SCP** (for internal character storage).
- So this creates **two objects**:
  → One in Heap (via `new`)
  → One in SCP (if not already present)

Key Differences: Literal vs Object

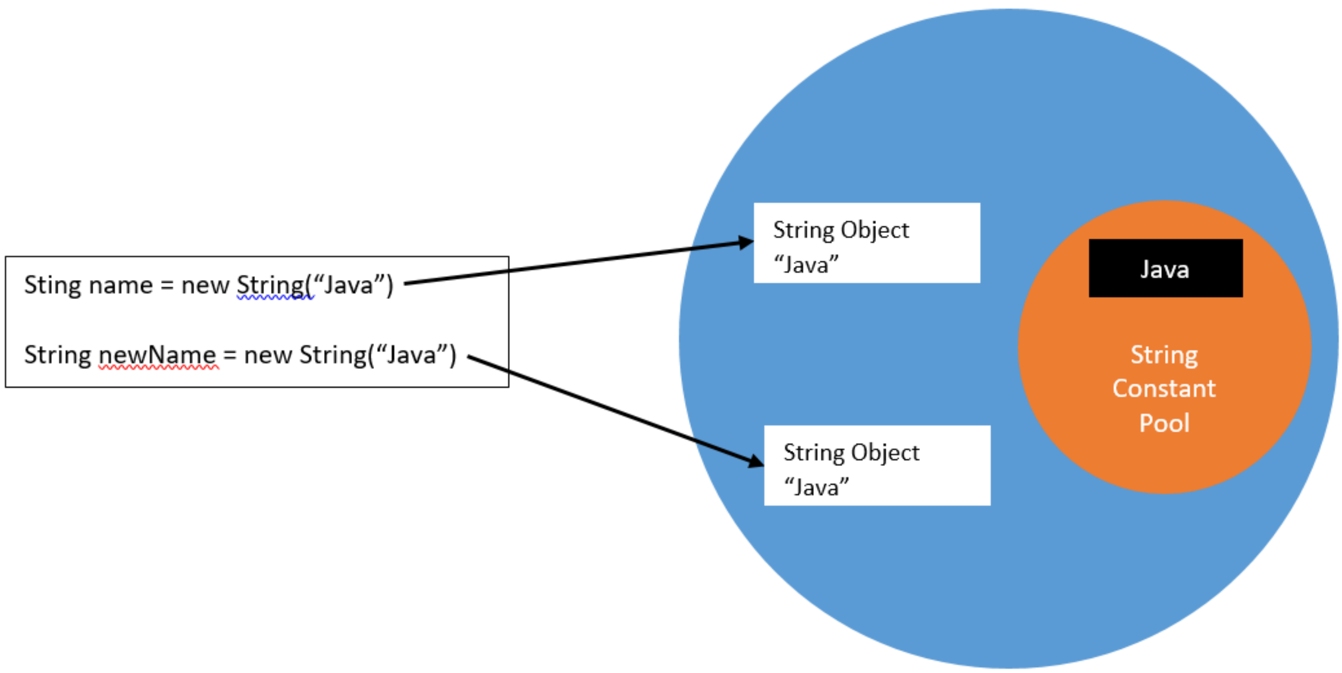
FEATURE	STRING LITERAL ("JAVA")	STRING OBJECT (NEW STRING("JAVA"))
MEMORY LOCATION	String Constant Pool (SCP)	Heap + reference to SCP
REUSE	Reused if already exists	Always a new object
EFFICIENCY	Memory efficient	Less efficient (creates duplicate)
COMPARISON USING ==	Might return `true`	Always returns `false`
EXAMPLE	`String s = "Java";`	`String s = new String("Java");`

Memory Diagram

String Literal



String Object



Comparison Example

```
public class StringMemoryDemo {
    public static void main(String[] args) {
        String s1 = "Java";
        String s2 = "Java";
        String s3 = new String("Java");

        System.out.println(s1 == s2);      // true (same SCP object)
        System.out.println(s1 == s3);      // false (heap vs SCP)
        System.out.println(s1.equals(s3)); // true (content comparison)
    }
}
```

Output:

```
true
false
true
```

Real-World Analogy

Imagine the **SCP as a library**:

- When you ask for a book titled "Java":
  - If it already exists on the shelf, you're **given the same copy** (literal).
  - If you say, "I want a brand new one" (using new), then the library **prints a fresh copy** and gives it to you.

Best Practices

- Use string **literals** when possible to save memory.
- ❌ Avoid unnecessary use of new String() unless you need a **separate object**.
- Always use .equals() to compare strings (not ==).

Bonus: intern() Method

You can force a string object to refer to SCP:

```
String s4 = new String("Java").intern();
```

Now s4 will point to "Java" in SCP — just like a literal.

Summary

TERM	MEANING
SCP	String Constant Pool — stores unique string literals
HEAP	General object storage area in memory
NEW STRING()	Always creates a new object in Heap
INTERN()	Moves or refers a string to the SCP
==	Compares reference (address)
EQUALS()	Compares content

2.11.3 StringBuffer in Java

1. Overview / Explanation

- `StringBuffer` is a **mutable** sequence of characters (unlike `String`, which is immutable).
- Part of `java.lang` package.
- Used when you need to **modify strings frequently** (e.g., appending, inserting, deleting).
- **Thread-safe** – methods are **synchronized**, so safe to use in multi-threaded environments.

Use Case: When building dynamic strings in a loop or multithreaded app – e.g., processing input, generating reports.

2. Declaration and Instantiation

```
StringBuffer sb1 = new StringBuffer();              // Empty buffer
StringBuffer sb2 = new StringBuffer("Hello");       // Initialized
StringBuffer sb3 = new StringBuffer(50);            // With capacity
```

3. Common Methods with Examples

append()

Adds text at the end.

```
sb1.append("Java");
System.out.println(sb1);  // Java
```

insert()

Inserts text at a specific index.

```
sb1.insert(4, " Programming");
System.out.println(sb1);  // Java Programming
```

replace()

Replaces part of the string between start and end index.

```
sb1.replace(0, 4, "Python");
System.out.println(sb1);  // Python Programming
```

delete()

Deletes characters between start and end index.

```
sb1.delete(0, 7);
System.out.println(sb1);  // Programming
```

reverse()

Reverses the entire content.

```
sb1.reverse();
System.out.println(sb1);  // gnimmargorP
```

length() and capacity()

```
System.out.println(sb1.length());   // No. of characters
System.out.println(sb1.capacity()); // Buffer capacity
```

charAt() and setCharAt()

```
char ch = sb1.charAt(0);
sb1.setCharAt(0, 'X');
```

4. Why Use StringBuffer Over String?

OPERATION	STRING	STRINGBUFFER
MUTABILITY	Immutable	Mutable
THREAD-SAFE	Not thread-safe	Yes
PERFORMANCE	Slower in loops	Faster in loops

5. StringBuffer vs StringBuilder

FEATURE	STRINGBUFFER	STRINGBUILDER
THREAD-SAFETY	Yes (synchronized)	No
PERFORMANCE	Slower	Faster (in single-thread)
USE CASE	Multithreaded apps	Single-thread apps

2.11.4 StringBuilder in Java

1. Overview / Explanation

- StringBuilder is a **mutable** sequence of characters, just like StringBuffer.
- **Not thread-safe**, but **faster** than StringBuffer in single-threaded applications.
- Part of java.lang package.
- Ideal when you're performing **lots of modifications to strings** in a **single-threaded** context.

Use Case: Building or modifying strings inside loops, parsing files, generating HTML reports, etc.

2. Declaration and Instantiation

```
StringBuilder sb1 = new StringBuilder();              // Empty buffer
StringBuilder sb2 = new StringBuilder("Hello");       // With initial value
StringBuilder sb3 = new StringBuilder(50);            // With specific capacity
```

3. Common Methods with Examples

append()

```
sb1.append("Java");
System.out.println(sb1);  // Java
```

insert()

```
sb1.insert(4, " World");
System.out.println(sb1);  // Java World
```

replace()

```
sb1.replace(0, 4, "Hello");
System.out.println(sb1);  // Hello World
```

delete()

```
sb1.delete(5, 11);
System.out.println(sb1);  // Hello
```

reverse()

```
sb1.reverse();
System.out.println(sb1);  // olleH
```

length() and capacity()

```
System.out.println(sb1.length());    // Number of characters
System.out.println(sb1.capacity());  // Total buffer size (default is 16 + initial content length)
```

charAt() and setCharAt()

```
char ch = sb1.charAt(0);
sb1.setCharAt(0, 'M');
System.out.println(sb1);  // Ml...
```

4. StringBuilder vs String vs StringBuffer

FEATURE	STRING	STRINGBUILDER	STRINGBUFFER
MUTABILITY	❌ Immutable	Mutable	Mutable
THREAD-SAFE	❌ No	❌ No	Yes
PERFORMANCE	❌ Slower	Fastest	⚠ Slower (sync)
BEST FOR	Constant text	Fast updates (1 thread)	Multithreading

2.11.5 String vs StringBuffer vs StringBuilder

FEATURE	STRING	STRINGBUFFER	STRINGBUILDER
MUTABILITY	❌ Immutable	Mutable	Mutable
THREAD-SAFE	❌ No	Yes (all methods are synchronized)	❌ No
PERFORMANCE	❌ Slowest (new object per change)	⚠ Slower (due to thread-safety overhead)	Fastest (no sync overhead)
SYNCHRONIZATION	❌ Not applicable	Synchronized	❌ Not synchronized
USE CASE	Constant/fixed string content	Multi-threaded environment	Single-threaded environment
PACKAGE	java.lang	java.lang	java.lang
INTRODUCED IN	JDK 1.0	JDK 1.0	JDK 1.5
METHODS FOR CHANGE	N/A (strings can't be modified)	append(), insert(), delete(), replace()	append(), insert(), delete(), replace()
MEMORY EFFICIENT?	❌ No (creates many objects)	Yes	Yes

Example Comparison

```
// String (immutable)
String s = "Hello";
s = s + " World";  // Creates a new String object

// StringBuffer (mutable, thread-safe)
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World");  // Modifies original object

// StringBuilder (mutable, not thread-safe)
StringBuilder sb2 = new StringBuilder("Hello");
sb2.append(" World");  // Modifies original object
```

When to Use What?

SITUATION	RECOMMENDED TYPE
SIMPLE, UNCHANGING TEXT	String
MANY STRING CHANGES IN MULTITHREADED **CODE**	StringBuffer
MANY STRING CHANGES IN SINGLE-THREADED **CODE**	StringBuilder

2.12.1 Reusability (in Java)

Definition:

Reusability is the ability to write code once and **use it multiple times** without rewriting it.

Real-life Analogy:

Like using the **same key for multiple locks** in your house — one key, many doors.

In Java:

- Achieved using:
    - **Methods**
    - **Classes**
    - **Inheritance**
    - **Packages**

Benefits:

- Reduces code duplication
- Saves development time
- Improves code quality and consistency
- Easier to debug and test

2.12.2 Modularity (in Java)

Definition:

Modularity is the process of dividing a large program into **independent, interchangeable modules**.

Real-life Analogy:

Like assembling a **car from different parts** (engine, wheels, seats) — each part (module) works independently but together they form a complete system.

In Java:

- Achieved using:
  - **Classes and methods**
  - **Packages**
  - **Modules (Java 9+)**

Benefits:

- Easier to understand and manage code
- Improves maintainability
- Encourages separation of concerns
- Simplifies teamwork and parallel development

Quick Comparison:

FEATURE	REUSABILITY	MODULARITY
PURPOSE	Use the same code again	Divide code into logical parts
REDUCES	Duplication	Complexity
ACHIEVED BY	Inheritance, methods, packages	Classes, packages, Java modules

Unit 3: Object-Oriented Programming – II

3.1.1 Aggregation in Java

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

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

Syntax Example

```
class Address {
    String city, state;

    Address(String city, String state) {
        this.city = city;
        this.state = state;
    }
}

class Customer {
    String name;
    Address address; // Aggregation: Customer has an Address

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

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

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

Output

Ravi lives in Chennai, Tamil Nadu

Real-World Analogy

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

When to Use Aggregation

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

Best Practices

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

3.1.2 Composition in Java?

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

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

Key Characteristics of Composition

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

Syntax Example

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

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

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

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

    void drive() {
        engine.start();
        System.out.println("Car is moving");
    }
}

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

Output

Engine started
Car is moving

Real-World Analogy

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

When to Use Composition

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

Composition vs Aggregation

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

3.2.1 Association in Java

What is Association in Java?

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

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

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

Basic Syntax Example

```
class Customer {
    String name;

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

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

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

Output:

Rahul placed an order.

Types of Association in Java

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

1. One-Way Association (Unidirectional)

Only one class is aware of the relationship.

Example: `Order` knows `Customer`, but `Customer` doesn’t know `Order`.

2. Two-Way Association (Bidirectional)

Both classes are aware of each other.

Example:

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

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

class Order {
    String id;
    Customer customer;

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

Now both classes **reference each other**.

Association vs Aggregation vs Composition

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

Real-World Analogy

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

Best Practices

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

Summary

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

Would you like this turned into a:

- **Visual diagram or chart**?
- **MCQ quiz** for practice?
- Or added to your OOPs module notes with real industry case examples?

Let me know!

3.3.1 Inheritance in Java

Definition:

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

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

Real-world Analogy:

Imagine:

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

Similarly in Java:

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

3.3.2 Syntax with Customer Example

Code Explanation

We have:

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

Code

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

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

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

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

Usage

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

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

        // Own property
        pc.rewardPoints = 1500;

        // Inherited method
        pc.showDetails();

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

Output

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

What's Happening Here

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

Key Takeaways

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

PremiumCustomer is-a Customer

3.3.3 Real-world Example:

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

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

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

Inheritance Hierarchy:

```
public class InheritanceDemo {

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

}

class Person {
    static {
        System.out.println("Person Static Block");
    }

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

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

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

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

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

    Customer(String name, String email) {
        System.out.println("Customer Parameterized Constructor: " + name + ", " + email + "\n");
    }
}

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

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

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

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

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

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

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

Output Explanation

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

Static blocks

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

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

Instance Initializers and Constructors

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

```
Person Instance Initializer
Person Constructor

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

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

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

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

```
Person Instance Initializer
Person Constructor

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

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

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

Parameterized Constructor Flow

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

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

Order of Execution

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

3.3.4 Types of Inheritance in Java:

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

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

1. Single Inheritance

One child class inherits from one parent class.

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

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

PremiumCustomer inherits from Customer.

2. Multilevel Inheritance

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

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

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

```
    }
}

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

LoyalCustomer inherits from Customer, and Customer inherits from Person.

3. Hierarchical Inheritance

☞ Multiple child classes inherit from a single parent class.

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

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

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

PremiumCustomer and RegularCustomer both inherit from Customer.

4. Multiple Inheritance (Not Supported via Classes)

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

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

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

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

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

How Java Supports Multiple Inheritance: Using Interfaces

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

interface B {
    void msg();
}

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

No conflict if the method is implemented in C.

5. Hybrid Inheritance (Combination — Only via Interfaces)

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

Summary Table

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

Things to Remember:

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

3.4.1 What is `super` in Java?

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

Real-world Analogy

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

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

Use Cases of super

It is mainly used for:

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

1. Calling Parent Class Constructor

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

Syntax:

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

Example:

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

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

Output:
Customer Constructor
PremiumCustomer Constructor

2. Accessing Parent Class Methods

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

Example:

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

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

Output:

Customer details
Premium customer details

3. Accessing Parent Class Variables

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

Example:

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

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

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

Output:

Child name: Premium Customer
Parent name: General Customer

Difference between `this` and `super`

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

Important Rules

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

Best Practices

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

3.5.1 Method Overriding

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

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

Real-World Analogy

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

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

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

Syntax Example

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

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

Usage Example

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

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

Rules for Overriding

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

Overriding with Access Modifiers

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

class B extends A {
    public void show() { } // Valid (more accessible)
}
```

Illegal Override Example

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

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

Real-World Example: Customer

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

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

```
}

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

Best Practices

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

Summary

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

Here's a **complete guide to Polymorphism in Java** — covering types, real-world analogies, syntax, examples, and how it connects with method overriding/overloading. This is ideal for lectures, student notes, or interview prep.

3.6.1 Polymorphism

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

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

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

Real-World Analogy

Imagine the word **"print"**:

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

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

Types of Polymorphism

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

Compile-Time Polymorphism (Method Overloading)

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

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

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

Runtime Polymorphism (Method Overriding)

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

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

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

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

Why Use Polymorphism?

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

Real-World Corporate Example: Customer Notifications

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

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

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

Key Concepts to Remember

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

Bonus: Polymorphism with Interfaces

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

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

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

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

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

Summary

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

3.7.1 Abstraction

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

Why? To reduce complexity and increase efficiency.

Real-World Analogy

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

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

This is **abstraction**.

What is an Abstract Class?

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

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

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

Example: Real-World Corporate — Customer Notification System

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

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

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

Usage:

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

Key Points About Abstract Classes

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

Example with Constructor:

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

    abstract void accessDashboard();
}

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

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

Summary: Abstract Class vs Interface (Quick Look)

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

Best Practices

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

3.8.1 Interface

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

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

Real-World Analogy

Imagine an **ATM Machine Interface**:

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

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

Syntax

```
interface Payment {
    void pay();  // implicitly public and abstract
}
```

Implementation:

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

Example: Customer Payment Interface

```
interface PaymentService {
    void processPayment(double amount);
}

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

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

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

```
        p1.processPayment(1000);

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

Features of Interface

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

Interface Features by Java Version

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

Java 8+ Interface Example (with default method)

```
interface Printer {
    void print();

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

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

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

public class InterfaceDemo {
```

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

Interface vs Abstract Class

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

Best Practices

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

3.8.2 Functional Interface

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

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

Key Points:

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

Real-World Analogy

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

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

Syntax

```
@FunctionalInterface
interface MessageService {
    void sendMessage(String message);  // Single abstract method
}
```

You can then implement it like this using a lambda:

```
public class Main {
    public static void main(String[] args) {
        MessageService service = (msg) -> System.out.println("Sending: " + msg);
        service.sendMessage("Welcome!");
    }
}
```

Why Use @FunctionalInterface Annotation?

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

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

Built-in Functional Interfaces (Java 8+)

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

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

Example: Using Built-in Functional Interface

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

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

Functional Interface vs Abstract Class

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

Best Practices

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

Here's a **complete guide to Lambda Expressions in Java**, ideal for teaching, interviews, or practical use in real-world coding.

3.8.3 Lambda Expression

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

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

Syntax

(parameters) -> { body }

Or simplified:

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

Example 1: Without Lambda (Traditional)

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

With Lambda:

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

Real-World Analogy

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

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

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

Real Example: Functional Interface + Lambda

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

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

Example 2: Lambda with Parameters

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

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

Common Uses (Java 8+)

Lambda expressions are **widely used** in:

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

Lambda with Java Built-in Functional Interfaces

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

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

Advantages of Lambda Expressions

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

Rules / Restrictions

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

3.8.4 Binding in Java

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

There are two types:

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

1. STATIC BINDING (Early Binding)

Definition:

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

Key Characteristics:

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

Example:

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

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

Real-World Analogy:

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

2. DYNAMIC BINDING (Late Binding)

Definition:

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

Key Characteristics:

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

3.8.5 Dynamic Method Dispatch

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

Syntax Example:

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

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

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

Real World Example: Customer Support System

We have:

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

Code Example

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

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

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

```
}

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

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

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

How It Works: Dynamic Dispatch in Action

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

Analogy

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

Multiple Inheritance Highlight

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

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

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

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

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

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

```
}
```

Why This Is the Best Example

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

Comparison Table

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

Overloading vs Overriding (Quick Reference)

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

Summary

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

3.8.6 What is Casting in Java?

Casting in Java is converting one type to another.

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

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

1. Upcasting (Widening Reference)

Definition:

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

Safe and implicit — no explicit cast needed.

Syntax:

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

Example:

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

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

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

Analogy:

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

2. Downcasting (Narrowing Reference)

Definition:

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

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

Syntax:

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

Example:

```
Customer c = new PremiumCustomer();  // Upcasting
PremiumCustomer pc = (PremiumCustomer) c;  // Downcasting
pc.accessPremiumSupport();  // Allowed
```

Wrong Downcasting Example (Leads to `ClassCastException`):

```
Customer c = new Customer();  // Not actually a PremiumCustomer
PremiumCustomer pc = (PremiumCustomer) c;  // Runtime error!
```

Real-World Analogy

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

How JVM handles it:

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

Best Practice

- Always check the actual object type before downcasting:

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

3.9.1 final Keyword?

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

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

1. final Variable — Value Cannot Change

Use:

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

Example:

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

Real-World Analogy:

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

2. final Method — Cannot Be Overridden

Use:

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

Example:

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

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

Real-World Analogy:

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

3. final Class — Cannot Be Extended

Use:

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

Example:

```
final class PaymentGateway {
    void processPayment() {
        System.out.println("Payment done");
    }
}

// class CustomGateway extends PaymentGateway { }  // ❌ Error
```

Real-World Analogy:

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

Additional Notes

Final Reference (for Objects):

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

Final with Blank Initialization:

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

```
class Student {
    final int id;

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

final VS finally VS finalize()

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

Best Practices

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

3.10.1 `Object` Class and Its Methods in Java

Analogy:

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

What is the `Object` class?

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

So, when you write:

```
class MyClass { }
```

Behind the scenes, it's like:

```
class MyClass extends Object { }
```

Why is it important?

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

Common Methods of `Object` Class

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

1. toString() Method

By default:

MyClass@15db9742 // ClassName@hexadecimal hash

Customizing:

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

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

Output:

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

2. equals() Method

Default: Compares memory address (same as ==)

Override to check logical equality

```
class Student {
    String name;

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

3. hashCode() Method

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

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

Example:

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

4. getClass() Method

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

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

5. clone() Method

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

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

```
class Student implements Cloneable {
    String name;

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

6. finalize() Method (Deprecated)

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

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

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

Summary Table

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

Unit 4: Exceptional Handling & File Handling

4.1.1 What is Exception Handling?

Exception Handling in Java is a powerful mechanism to **handle runtime errors** so the normal flow of the application can be maintained.

What is an Exception?

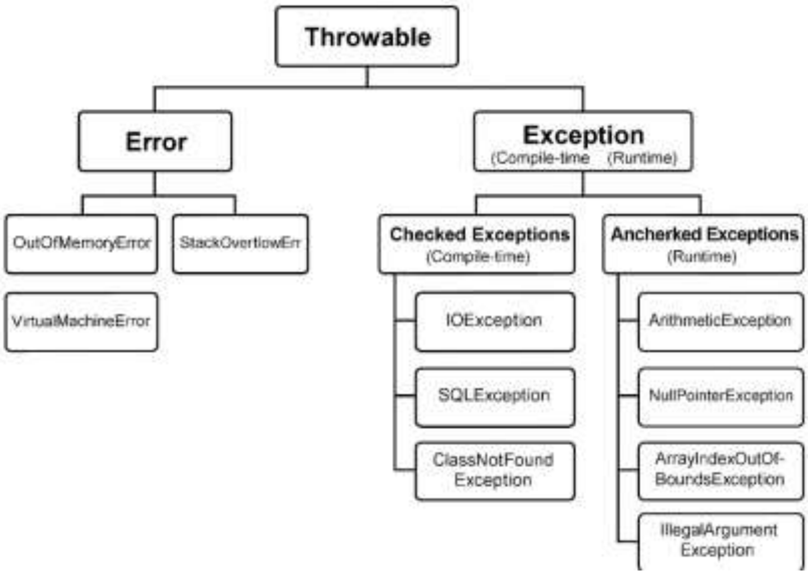
An **exception** is an **event** that occurs during the execution of a program that **disrupts the normal flow** of instructions.

Example: Dividing by zero, accessing an array out of bounds, file not found, etc.

Why Use Exception Handling?

- To **catch and handle errors** gracefully
- Prevent application **crashes**
- Log and fix **unpredictable conditions**
- Keep the **code clean and safe**

Exception Hierarchy



4.1.2 Keywords in Java Exception Handling

KEYWORD	DESCRIPTION
try	Code that might throw an exception
catch	Handles the exception
finally	Executes regardless of exception (cleanup code)
throw	Manually throw an exception
throws	Declare exceptions a method might throw

Syntax:

```
try {
    // risky code
} catch (ExceptionType e) {
    // handling code
} finally {
    // cleanup code
}
```

Example:

```
public class Example {
    public static void main(String[] args) {
        try {
            int a = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        } finally {
            System.out.println("Always executed");
        }
    }
}
```

Output:

Cannot divide by zero!
Always executed

throw vs throws

throw: Used to manually throw an exception.

```
throw new ArithmeticException("Divide by zero");
```

throws: Used to declare exceptions a method might throw.

```
void readFile() throws IOException {
    // code that may throw IOException
}
```

Real-World Analogy

Imagine:

- **try block** = Entering a risky zone (e.g., ATM transaction)
- **catch block** = Security camera catches and reports issue
- **finally block** = Clean-up like removing card from ATM

Best Practices

- Catch **specific exceptions** first, then general
- Avoid empty catch blocks
- Always clean resources in finally or use **try-with-resources**
- Don't overuse exceptions for control flow

4.1.3 Types of Exceptions in Java

Java broadly divides exceptions into **two main types**:

CATEGORY	DESCRIPTION	HANDLED AT?
CHECKED EXCEPTION	Known at **compile-time**	Must be handled
UNCHECKED EXCEPTION	Known at **runtime**	Optional to handle

There's also a third category:

CATEGORY	DESCRIPTION	HANDLED AT?
ERROR	Serious issues, not meant to be caught	System-level issue

1. Checked Exceptions (Compile-Time Exceptions)

These exceptions must be either:

- **caught using try-catch**, or
- **declared using throws**

Examples:

- IOException
- SQLException
- FileNotFoundException
- ClassNotFoundException

Example:

```
import java.io.*;

public class CheckedEx {
  public static void main(String[] args) throws IOException {
    FileReader fr = new FileReader("file.txt");  // Checked exception
  }
}
```

2. Unchecked Exceptions (Runtime Exceptions)

These are exceptions that:

- **Do not need to be declared**
- Usually caused by **programming errors** (e.g., logic flaws)

All **subclasses of RuntimeException** are unchecked.

Examples:

- ArithmeticException
- NullPointerException
- ArrayIndexOutOfBoundsException
- NumberFormatException

Example:

```
public class UncheckedEx {
  public static void main(String[] args) {
    int a = 10 / 0;  // ArithmeticException
  }
}
```

Errors (Not Exceptions)

Errors are serious problems that **cannot be handled by the application**. They indicate **system failure** or **resource issues**.

Examples:

- OutOfMemoryError
- StackOverflowError
- VirtualMachineError

Note:

You should **never try to catch Errors** — they're meant to crash the program or JVM.

Summary Table

TYPE	PARENT CLASS	MUST HANDLE?	EXAMPLES
CHECKED EXCEPTION	Exception	Yes	IOException, SQLException
UNCHECKED EXCEPTION	RuntimeException	✕ No	ArithmeticException, NPE
ERROR	Error	✕ No	OutOfMemoryError, StackOverflow

Best Practices

- Always handle **checked exceptions** properly using `try-catch` or `throws`.
- Avoid catching **generic Exception** or **Error** unless absolutely necessary.
- Handle **unchecked exceptions** through proper validation and safe coding.

4.1.4 Multiple `catch` Block

A **multiple `catch` block** allows you to handle **different types of exceptions separately** using different handlers for different exception classes — all associated with a single `try` block.

Syntax:

```
try {
    // Code that might throw multiple exceptions
} catch (ExceptionType1 e1) {
    // Handle ExceptionType1
} catch (ExceptionType2 e2) {
    // Handle ExceptionType2
} catch (Exception e) {
    // General exception handler (optional, must be last)
}
```

Real-World Analogy

Imagine you're trying to withdraw money from an ATM:

- If the card is invalid → `CardException`
- If there's no network → `NetworkException`
- If ATM is out of cash → `CashUnavailableException`

Each error needs its own solution.

Example:

```
public class MultipleCatchExample {
    public static void main(String[] args) {
        try {
            int[] arr = new int[5];
            arr[5] = 100 / 0;  // Causes ArithmeticException first
        } catch (ArithmeticException ae) {
            System.out.println("Cannot divide by zero.");
        } catch (ArrayIndexOutOfBoundsException ai) {
            System.out.println("Array index is out of bounds.");
        } catch (Exception e) {
            System.out.println("General exception occurred.");
        }
    }
}
```

Output:

Cannot divide by zero.

Only the **first matching catch block** is executed.

Catch Order Rule

- Catch blocks must go from **most specific to most general**.
- If a **superclass exception** (like Exception) is caught before a **subclass** (like ArithmeticException), you'll get a **compile-time error**.

```
// ✕ Compile-time error
catch (Exception e) { ... }
catch (ArithmeticException ae) { ... }

Correct way:

catch (ArithmeticException ae) { ... }
catch (Exception e) { ... }
```

Java 7+ Feature: Multi-Catch (Single Catch for Multiple Exceptions)

```
try {
    // code
} catch (IOException | SQLException e) {
    System.out.println("IO or SQL error: " + e.getMessage());
}
```

- Use **| (pipe)** to combine exceptions.
- All exceptions **must not be in a parent-child relationship**.

Best Practices

- Catch **specific exceptions** first.
- Use **multi-catch** to reduce redundancy when exceptions need same handling.
- Log or handle each exception meaningfully.

- Don’t swallow exceptions silently (`catch (Exception e) {}` with empty body is bad).

Summary

FEATURE	PURPOSE
MULTIPLE CATCH	Handle different exceptions differently
ORDER MATTERS	Specific → General
MULTI-CATCH (JAVA 7+)	Handle multiple exceptions together

4.1.5 `throw` and `throws` in Java

Both `throw` and `throws` are used in **exception handling**, but they serve **different purposes**:

KEYWORD	PURPOSE
THROW	To **explicitly throw an exception**
THROWS	To **declare an exception** in method signature

1. `throw` Keyword

Definition:

The `throw` keyword is used to **manually throw an exception** (either checked or unchecked).

Syntax:

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

Example:

```
public class ThrowExample {
    public static void main(String[] args) {
        int age = 15;
        if (age < 18) {
            throw new ArithmeticException("Not eligible to vote");
        }
        System.out.println("You can vote!");
    }
}
```

Output:

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

2. throws Keyword

Definition:

The throws keyword is used in the method signature to **declare** one or more exceptions that the method might throw. This shifts the responsibility to the method caller.

EXCEPTION TYPE	THROWS REQUIRED?	EXAMPLE
checked exception	Yes	IOException, SQLException
unchecked exception	✗ No	NullPointerException, ArithmeticException

Syntax:

```
returnType methodName() throws ExceptionType1, ExceptionType2 {
    // method code
}
```

Example:

```
import java.io.*;

public class ThrowsExample {
    static void readFile() throws IOException {
        FileReader fr = new FileReader("file.txt"); // May throw IOException
    }

    public static void main(String[] args) {
        try {
            readFile();
        } catch (IOException e) {
            System.out.println("File not found!");
        }
    }
}
```

throw vs throws – Comparison Table

FEATURE	THROW	THROWS
PURPOSE	Actually throws an exception	Declares exceptions a method may throw
PLACEMENT	Inside method body	In method signature
NUMBER OF EXCEPTIONS	One at a time	Can declare multiple, comma-separated
USED FOR	Instantiating and throwing exception	Forwarding responsibility to calling method
FOLLOWS BY	Instance of Throwable subclass	List of exception classes

Real-World Analogy

- throw = You manually **raise a red flag** (you throw the error).
- throws = You **warn others** that this method **might throw a red flag**.

Summary

- Use throw to **actually throw** the exception.
- Use throws to **declare** that a method might throw an exception.
- Always **handle checked exceptions** either using try-catch or throws.

4.1.7 Exception Chaining in Java — Complete Explanation

Exception chaining is a powerful concept in Java that allows you to associate one exception with another — making it easier to **track the root cause** of a problem across multiple layers of code.

What Is Exception Chaining?

Exception chaining means **wrapping one exception inside another** so that you can propagate the **original cause** while throwing a **higher-level exception**.

This helps preserve the actual root problem even when re-throwing a new exception.

Why Use Exception Chaining?

- To preserve the **original exception context**
- To **abstract internal details** while providing user-friendly messages
- For better **debugging and logging**
- To maintain **clean exception architecture** in multi-layered applications

Syntax

```
Throwable getCause();
Throwable initCause(Throwable cause);
```

Or use constructors directly:

```
public NewException(String message, Throwable cause);
```

Example: Without Chaining

```
public class NoChaining {
    public static void main(String[] args) {
        try {
            parseNumber("abc");
        } catch (NumberFormatException e) {
            throw new RuntimeException("Failed to parse input.");
        }
    }

    static void parseNumber(String s) {
        Integer.parseInt(s); // Throws NumberFormatException
    }
}
```

Output:

```
Exception in thread "main" java.lang.RuntimeException: Failed to parse input
```

The actual **cause** (NumberFormatException) is lost.

Example: With Exception Chaining

```
public class ChainedExceptionDemo {
    public static void main(String[] args) {
        try {
            parseNumber("abc");
        } catch (NumberFormatException e) {
            throw new RuntimeException("Failed to parse input", e);  // Chaining
        }
    }

    static void parseNumber(String s) {
        Integer.parseInt(s); // Throws NumberFormatException
    }
}
```

Output:

```
Exception in thread "main" java.lang.RuntimeException: Failed to parse input
Caused by: java.lang.NumberFormatException: For input string: "abc"
```

You can now **trace the root cause**!

Real-World Analogy

Imagine a customer order fails because:

1. Payment service failed.
2. Payment failed because database access failed.

Each layer throws a new exception with a user-friendly message but **chains the original cause**, so developers can debug the actual root — **DB failure**.

How to Create a Custom Exception with Chaining

```
class MyCustomException extends Exception {
    public MyCustomException(String message, Throwable cause) {
        super(message, cause);  // Proper chaining
    }
}
```

Usage:

```
try {
    throw new IOException("Disk failure");
} catch (IOException e) {
    throw new MyCustomException("System error occurred", e);
}
```

Best Practices

- Always **chain exceptions** if you're re-throwing at a higher level.
- Avoid hiding the root cause.
- Use getCause() when logging or debugging.
- Custom exceptions should include a constructor that accepts a cause.

Summary Table

FEATURE	DESCRIPTION
PURPOSE	Preserve root cause when rethrowing exceptions
METHOD	Use constructors with Throwable cause
BENEFITS	Easier debugging, cleaner error propagation
COMMON USAGE	Service layers, API abstraction, frameworks

4.2.1 File Handling in Java

File handling allows you to **create**, **read**, **write**, **update**, and **delete** files using the Java I/O (java.io) and NIO (java.nio) packages.

Common Classes in File Handling

CLASS	PURPOSE
File	Represent file/directory
FileReader	Read character files
FileWriter	Write character files
BufferedReader	Efficient reading of text
BufferedWriter	Efficient writing of text
PrintWriter	Convenient file writing with print methods
Scanner	Read text using regex/token patterns

1. Creating and Checking Files with File Class

```
import java.io.File;
import java.io.IOException;

public class CreateFileDemo {
    public static void main(String[] args) {
        try {
            File myFile = new File("example.txt");

            if (myFile.createNewFile()) {
                System.out.println("File created: " + myFile.getName());
            } else {
                System.out.println("File already exists.");
            }

        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

2. Writing to a File

```
import java.io.FileWriter;
import java.io.IOException;
```

```java
public class WriteToFile {
    public static void main(String[] args) {
        try {
            FileWriter writer = new FileWriter("example.txt");
            writer.write("Hello, this is a file write example.");
            writer.close();
            System.out.println("Successfully written to the file.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

3. Reading from a File

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class ReadFile {
    public static void main(String[] args) {
        try {
            File file = new File("example.txt");
            Scanner reader = new Scanner(file);
            while (reader.hasNextLine()) {
                String data = reader.nextLine();
                System.out.println(data);
            }
            reader.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

4. Deleting a File

```java
import java.io.File;

public class DeleteFile {
    public static void main(String[] args) {
        File file = new File("example.txt");
        if (file.delete()) {
            System.out.println("Deleted the file: " + file.getName());
        } else {
            System.out.println("Failed to delete the file.");
        }
    }
}
```

4.2.2 BufferedReader (Efficient Reading)

```
import java.io.*;

public class BufferedReaderExample {
    public static void main(String[] args) {
        try {
            BufferedReader br = new BufferedReader(new FileReader("example.txt"));
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
            br.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

4.2.3 PrintWriter (Efficient Writing)

```
import java.io.*;

public class PrintWriterExample {
    public static void main(String[] args) throws IOException {
        PrintWriter pw = new PrintWriter("example.txt");
        pw.println("Line 1");
        pw.println("Line 2");
        pw.close();
        System.out.println("Data written using PrintWriter");
    }
}
```

Best Practices

- Always close streams (close() or try-with-resources).
- Prefer BufferedReader/Writer for large files.
- Use try-with-resources for auto-closing streams (Java 7+).
- Always handle IOException.

Summary Table

OPERATION	CLASS USED
CREATE	File
READ	FileReader, Scanner, BufferedReader
WRITE	FileWriter, PrintWriter, BufferedWriter
DELETE	File

4.2.4 File Handling with .csv file

1. Create and Write a .csv File (Manual – Without Libraries)

```
import java.io.FileWriter;
import java.io.IOException;

public class CsvWrite {
    public static void main(String[] args) {
        String filePath = "data.csv";
        try (FileWriter writer = new FileWriter(filePath)) {
            writer.append("ID,Name,Email\n");
            writer.append("1,John Doe,john@example.com\n");
            writer.append("2,Jane Smith,jane@example.com\n");
            System.out.println("CSV file created and written successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2. Read a .csv File (Manual Read)

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

public class CsvRead {
    public static void main(String[] args) {
        String filePath = "data.csv";
        try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
            String line;
            while ((line = br.readLine()) != null) {
                String[] values = line.split(",");
                for (String v : values) {
                    System.out.print(v + "\t");
                }
                System.out.println();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

3. Delete a .csv File

```
import java.io.File;

public class CsvDelete {
    public static void main(String[] args) {
        File file = new File("data.csv");
        if (file.delete()) {
```

```java
            System.out.println("CSV file deleted successfully.");
        } else {
            System.out.println("Failed to delete the file.");
        }
    }
}
```

Optional: Use OpenCSV (Simplified & Cleaner)

Add Dependency

If you're using Maven:

```xml
<dependency>
    <groupId>com.opencsv</groupId>
    <artifactId>opencsv</artifactId>
    <version>5.7.1</version>
</dependency>
```

Write with OpenCSV

```java
import com.opencsv.CSVWriter;
import java.io.FileWriter;
import java.io.IOException;

public class OpenCsvWrite {
    public static void main(String[] args) {
        try (CSVWriter writer = new CSVWriter(new FileWriter("data.csv"))) {
            String[] header = { "ID", "Name", "Email" };
            String[] record1 = { "1", "John", "john@example.com" };
            String[] record2 = { "2", "Jane", "jane@example.com" };

            writer.writeNext(header);
            writer.writeNext(record1);
            writer.writeNext(record2);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Read with OpenCSV

```java
import com.opencsv.CSVReader;
import java.io.FileReader;
import java.io.IOException;

public class OpenCsvRead {
    public static void main(String[] args) {
        try (CSVReader reader = new CSVReader(new FileReader("data.csv"))) {
            String[] line;
            while ((line = reader.readNext()) != null) {
                for (String cell : line) {
```

```
                    System.out.print(cell + "\t");
                }
                System.out.println();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Summary Table

TASK	JAVA I/O CLASSES	LIBRARY (OPTIONAL)
CREATE	FileWriter	OpenCSV CSVWriter
READ	BufferedReader	OpenCSV CSVReader
DELETE	File	—

4.2.5 Exception handling in file handling

Common Exceptions in File Handling

EXCEPTION	CAUSE
FILENOTFOUNDEXCEPTION	File doesn't exist (when reading)
IOEXCEPTION	General I/O error (read/write/close fails)
SECURITYEXCEPTION	Access denied due to JVM SecurityManager policy
NULLPOINTEREXCEPTION	Stream used without being properly initialized
EOFEXCEPTION	Reached end of file unexpectedly during reading

Note: Best Practice: Always Use `try-catch-finally` *or* `try-with-resources`

Example: Handling Exception While Reading a File

```
import java.io.*;

public class ReadFileWithExceptionHandling {
    public static void main(String[] args) {
        BufferedReader br = null;

        try {
            br = new BufferedReader(new FileReader("input.txt"));
            String line;
```

```
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }

        } catch (FileNotFoundException e) {
            System.out.println("⚠ File not found. Please check the file path.");
        } catch (IOException e) {
            System.out.println("⚠ An error occurred while reading the file.");
        } finally {
            try {
                if (br != null)
                    br.close();
            } catch (IOException e) {
                System.out.println("⚠ Failed to close the file properly.");
            }
        }
    }
}
```

Better Way: Try-With-Resources (Java 7+)

```
import java.io.*;

public class TryWithResourcesExample {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new FileReader("input.txt"))) {
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (FileNotFoundException e) {
            System.out.println("⚠ File not found!");
        } catch (IOException e) {
            System.out.println("⚠ Error reading the file.");
        }
    }
}
```

Automatically closes file
Cleaner code
No need for finally

Example: Writing File with Exception Handling

```
import java.io.FileWriter;
import java.io.IOException;

public class WriteFileWithExceptionHandling {
    public static void main(String[] args) {
        try (FileWriter fw = new FileWriter("output.txt")) {
            fw.write("File written successfully.");
        } catch (IOException e) {
```

```
            System.out.println("⚠ Cannot write to file: " + e.getMessage());
        }
    }
}
```

When to Use `throws`?

In **method definitions** for file operations in modular programs:

```
public void readFile(String path) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    // ...
}
```

The calling method must handle it.

Unit 5: Exceptional Handling & File Handling

5.1.1. Basics of Multithreading

Analogy:

Imagine you're cooking while listening to music and downloading a file — all happening simultaneously. That's **multitasking**, and in Java, it's called **multithreading**.

What is Multithreading?

- **Multithreading** is the ability of a program to execute **multiple threads concurrently**.
- A **thread** is a **lightweight subprocess**—the smallest unit of processing.
- Java supports multithreading via the **java.lang.Thread** class and the **Runnable** interface.

Why Use Multithreading?

BENEFIT	EXPLANATION
BETTER CPU UTILIZATION	Makes full use of processor cores
FASTER EXECUTION	Tasks run in parallel (e.g., download + UI update)
RESOURCE SHARING	Threads share memory space, making communication easier
ASYNCHRONOUS BEHAVIOR	Improves performance and user experience (e.g., in UI apps)

Single-threaded vs Multi-threaded

SINGLE-THREADED APP	MULTI-THREADED APP
ONE TASK AT A TIME	Multiple tasks at the same time
SLOWER AND LESS RESPONSIVE	Faster, more responsive

Thread vs Process

TERM	THREAD	PROCESS
DEFINITION	Smallest unit of a program	Independent program in memory
MEMORY	Shares memory with other threads	Has separate memory
OVERHEAD	Low	High

Real-life Examples of Multithreading:

- Web browsers: Render page + load resources + run JS
- Games: Background music + physics + rendering
- Text editor: Typing + spell check + autosave

Example Use Cases in Java:

- **Banking App**: One thread processes transactions, another logs them.
- **Video Player**: One thread decodes video, another handles audio.

5.1.2 Creating and Managing Threads in Java

Ways to Create a Thread in Java

There are **two main approaches**:

APPROACH	DESCRIPTION	USE WHEN...
1. EXTENDING THREAD	Create a subclass of Thread and override run()	You don't need to extend another class
2. IMPLEMENTING RUNNABLE	Create a class that implements Runnable and pass it to a Thread object	You need to extend another class

Method 1: Extending Thread Class

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

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

Method 2: Implementing Runnable Interface

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

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

```
        Thread t1 = new Thread(myRunnable); // Pass Runnable to Thread
        t1.start();
    }
}
```

Which One Should You Use?

- Prefer **Runnable** if your class already extends another class (since Java supports only single inheritance).
- Use **Thread** when you want to override `Thread` methods or don't need to extend any other class.

Creating Multiple Threads Example

```
class MyTask extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(getName() + ": " + i);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        MyTask t1 = new MyTask();
        MyTask t2 = new MyTask();

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

Practice Activities

1. Create a thread using `Thread` and `Runnable`—print your name 5 times in each.
2. Run 2 threads: One prints even numbers, the other prints odd numbers.
3. Modify the class to accept thread names and print them in the output.

5.1.3 Thread Lifecycle in Java

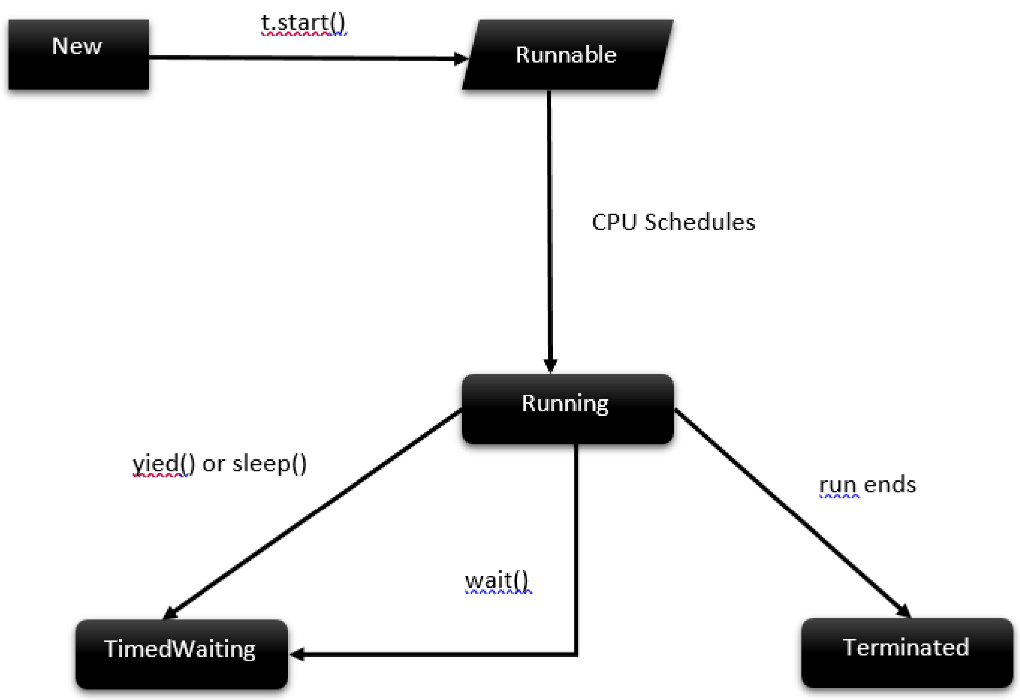
Analogy:

Think of a thread as a **task** performed by a **worker**.
Just like a worker goes through different stages—**hired**, **ready**, **working**, **waiting**, **done**—so does a thread.

Thread Lifecycle Stages

STATE	DESCRIPTION
NEW	Thread is created but not started
RUNNABLE	Thread is ready to run, waiting for CPU
RUNNING	Thread is executing
BLOCKED/WAITING	Thread is paused, waiting for a resource or another thread
TERMINATED	Thread has finished executing

Lifecycle Diagram



Description of Each State:

1. New

- Thread is created using new Thread() or by extending Thread.

```
Thread t = new Thread();  // NEW
```

2. Runnable

- You called start(). The thread is **ready**, waiting to be picked by CPU.

```
t.start();  // Moves to RUNNABLE
```

3. Running

- JVM scheduler has selected the thread to run.
- The thread's run() method is now executing.

4. Blocked/Waiting

- Thread is **waiting** due to:
    - sleep()
    - join()
    - wait()
- It resumes only when the condition is met (time ends, other thread completes, notify is called).

5. Terminated (Dead)

- run() method completes or an exception is thrown.

```
System.out.println("Thread done");
```

Lifecycle Demo in Code:

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

public class Main {
    public static void main(String[] args) {
        MyThread t = new MyThread();  // NEW
        System.out.println(t.getState()); // NEW
        t.start();                    // RUNNABLE -> RUNNING
        System.out.println(t.getState()); // Might print RUNNABLE or TERMINATED
    }
}
```

5.1.4 Important Thread Methods in Java

These methods help you control how threads behave—when to pause, wait, yield control, or coordinate with others.

Commonly Used Thread Methods

METHOD	DESCRIPTION
START()	Starts a new thread (calls run() internally)
RUN()	Contains the task the thread will perform
SLEEP(MS)	Pauses the thread for a specific time
JOIN()	Waits for another thread to finish
YIELD()	Suggests that the current thread pause and let others run
ISALIVE()	Checks if a thread is still running
SETNAME() / GETNAME()	Sets or gets thread name

1. start() vs run()

```
Thread t = new Thread();
t.start(); // Executes in a new thread
t.run();   // Just a method call, no new thread
```

Always use start() to begin multithreaded execution.

2. sleep()

Pauses the current thread temporarily.

```
Thread.sleep(1000); // 1 second
```

InterruptedException must be handled using try-catch.

```
try {
    Thread.sleep(2000);
} catch (InterruptedException e) {
    System.out.println("Interrupted!");
}
```

Use case: Delaying animations, retry mechanisms, simulating time.

3. join()

Waits for another thread to complete.

```
t1.join(); // Main thread waits for t1 to finish
```

Example:

```
Thread t1 = new Thread(() -> {
    for (int i = 0; i < 3; i++) {
        System.out.println("Child thread");
    }
});

t1.start();
t1.join(); // Main waits
System.out.println("Main thread runs after t1");
```

4. yield()

Temporarily pauses the current thread and allows other threads of the same priority to execute.

```
Thread.yield();
```

Not guaranteed to pause—it just **suggests** the CPU.

5. isAlive(), setName(), getName()

```
Thread t = new Thread();
t.setName("Worker-1");
System.out.println(t.getName());
System.out.println(t.isAlive()); // true if started and not finished
```

5.1.5 Synchronization in Java

Analogy:

Imagine two people trying to **withdraw money from the same ATM** at the same time. If they access the same account without taking turns, they might withdraw more than what's available — that's a **race condition**.

🔑 Solution? One person must wait — this is **synchronization**.

What is Synchronization?

Synchronization ensures that **only one thread can access a shared resource at a time**, preventing inconsistent or corrupt data.

The Problem Without Synchronization

```
class Counter {
    int count = 0;
    void increment() {
        count++;
    }
}

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

        Thread t1 = new Thread(() -> {
            for(int i = 0; i < 1000; i++) counter.increment();
        });
        Thread t2 = new Thread(() -> {
            for(int i = 0; i < 1000; i++) counter.increment();
        });

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

        System.out.println("Final count: " + counter.count); // Expected: 2000
    }
}
```

Output may be less than 2000!
Why? Both threads try to update `count` at the same time.

Solution: Use `synchronized`

1. Synchronized Method

```
class Counter {
    int count = 0;

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

2. Synchronized Block

```
synchronized(counter) {
    counter.increment();
}
```

You can synchronize **only the critical section** (the part that modifies shared data), which is more efficient.

Use Cases

- Bank account operations
- Online booking systems
- Shared counters or lists

Locks Behind the Scenes

Every object in Java has a **monitor lock**. When a thread enters a `synchronized` method/block, it acquires the lock. Other threads trying to access it must wait.

5.1.6 Inter-thread Communication in Java

Analogy:

Imagine a **producer** (chef) preparing food and a **consumer** (waiter) serving it.

- If the chef is too fast, the waiter can't keep up.
- If the waiter is too fast, he may find nothing to serve.

They need to **coordinate**.

That's **inter-thread communication** — threads cooperating instead of competing.

Why It's Needed

Java threads can **pause and notify each other** using three main methods (defined in `Object` class):

METHOD	DESCRIPTION
WAIT()	Pauses the current thread
NOTIFY()	Wakes up a single waiting thread
NOTIFYALL()	Wakes up all waiting threads

Rules:

1. These methods must be called **within a synchronized block/method**
2. They must be called **on the same object** used for locking

Producer-Consumer Example (Simplified)

```
class Store {
    int item;
    boolean available = false;

    synchronized void produce(int value) {
        while (available) {
            try { wait(); } catch (InterruptedException e) {}
        }
        item = value;
        available = true;
        System.out.println("Produced: " + item);
        notify();  // Notify consumer
    }

    synchronized void consume() {
        while (!available) {
            try { wait(); } catch (InterruptedException e) {}
        }
        System.out.println("Consumed: " + item);
        available = false;
        notify();  // Notify producer
    }
}
public class Main {
    public static void main(String[] args) {
        Store store = new Store();

        Thread producer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                store.produce(i);
            }
        });

        Thread consumer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                store.consume();
            }
        });

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

Breakdown:

- **Producer waits** if item is already available.
- **Consumer waits** if there's nothing to consume.
- They **notify** each other after completing their action.

Key Notes:

- Use `while` (not `if`) to avoid **spurious wakeups**.
- `wait()` releases the lock; `sleep()` does not.
- Ideal for **resource sharing** situations.

5.2.1 Java Collections Framework Overview

What is the Java Collections Framework?

It's a **set of classes and interfaces** in Java that provides **ready-to-use data structures** (like lists, sets, maps) and algorithms (like sorting and searching).

Think of it as Java's built-in **toolbox** for managing groups of objects.

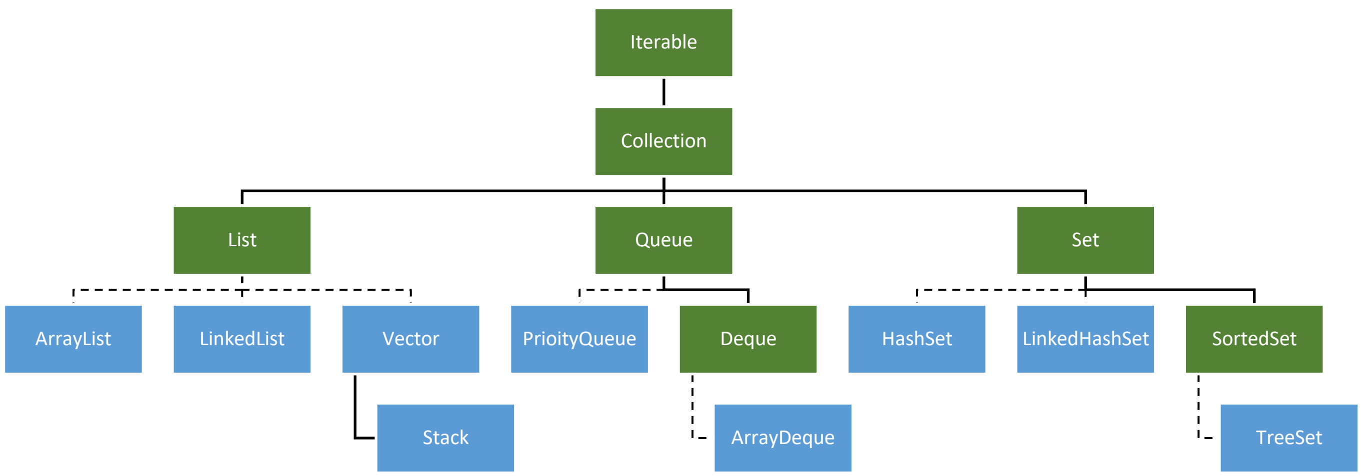
Why Use Collections?

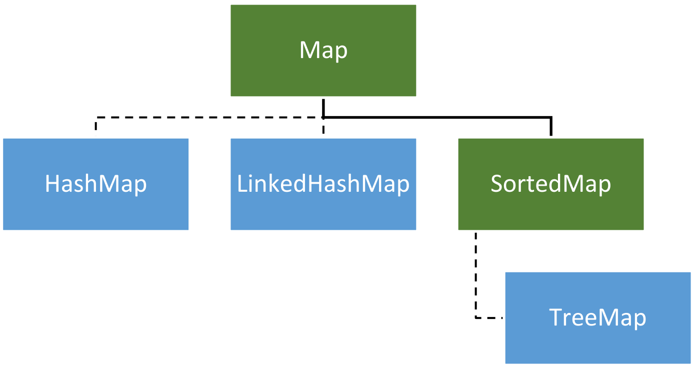
- No need to write your own data structures.
- Provides **efficient**, **scalable**, and **thread-safe** options.
- Interfaces allow **flexibility** and **interchangeability**.

Core Interfaces of Collections

INTERFACE	DESCRIPTION	COMMON IMPLEMENTATIONS
LIST	Ordered, duplicates allowed	ArrayList, LinkedList, Vector
SET	Unordered, no duplicates	HashSet, LinkedHashSet, TreeSet
MAP	Key-value pairs	HashMap, TreeMap, LinkedHashMap
QUEUE	FIFO structure	PriorityQueue, ArrayDeque

Collection Hierarchy





List vs Set vs Map

FEATURE	LIST	SET	MAP
ALLOWS DUPLICATES	Yes	✗ No	Keys no, Values yes
MAINTAINS ORDER	Yes (List)	Some (LinkedHashSet)	Yes (LinkedHashMap)
KEY ACCESS	✗	✗	Yes (via keys)

Key Classes at a Glance

1. ArrayList

- Resizable array
- Fast access, slow insertion/deletion in middle

```
ArrayList<String> list = new ArrayList<>();
list.add("Apple");
list.add("Banana");
System.out.println(list);
```

2. LinkedList

- Nodes connected by links
- Good for frequent insertions/deletions

```
LinkedList<Integer> nums = new LinkedList<>();
nums.add(10);
nums.addFirst(5);
```

3. HashSet

- No duplicates, no ordering
- Uses hash table

```java
HashSet<String> names = new HashSet<>();
names.add("John");
names.add("John"); // Ignored
```

4. HashMap

- Stores key-value pairs
- Keys must be unique

```java
HashMap<Integer, String> map = new HashMap<>();
map.put(1, "Apple");
map.put(2, "Mango");
System.out.println(map.get(1)); // Apple
```

Iterating over Collections

For List:

```java
for (String item : list) {
    System.out.println(item);
}
```

For Map:

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

5.2.2 ArrayList in Java

1. Overview / Explanation

- ArrayList is a **resizable array** in Java (part of java.util).
- Maintains **insertion order**.
- Allows **duplicate elements**.
- Elements are indexed (like arrays).
- Ideal for **random access** and **frequent read** operations.

Use Case: Managing a dynamic list of items like names, scores, tasks.

2. Declaration

```
ArrayList<String> list;             // Generic declaration
List<Integer> numbers;              // Using interface type
```

3. Instantiation

```
list = new ArrayList<>();           // No initial size
numbers = new ArrayList<>(10);      // With initial capacity
```

Full example:

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

4. Adding Elements

```
fruits.add("Apple");
fruits.add("Banana");
fruits.add("Mango");
fruits.add("Apple");  // Allows duplicates
```

5. Accessing Elements / Iteration

a) Index-based Access:

```
System.out.println(fruits.get(0)); // Apple
```

b) For Loop:

```
for (int i = 0; i < fruits.size(); i++) {
    System.out.println(fruits.get(i));
}
```

c) Enhanced For Loop:

```
for (String fruit : fruits) {
    System.out.println(fruit);
```

```
}
```

d) forEach + Lambda (Java 8+):

```
fruits.forEach(f -> System.out.println(f));
```

6. Updating Elements

```
fruits.set(1, "Grapes"); // Replace Banana with Grapes
```

7. Deleting / Removing Elements

a) By Value:

```
fruits.remove("Apple"); // Removes first occurrence
```

b) By Index:

```
fruits.remove(0);       // Removes item at index 0
```

c) Remove All / Clear:

```
fruits.clear();         // Removes all elements
```

8. Searching / Contains Check

```
boolean hasMango = fruits.contains("Mango");
int index = fruits.indexOf("Apple");
```

9. Sorting

```
Collections.sort(fruits); // Sorts in ascending order
```

Descending:

```
fruits.sort(Collections.reverseOrder());
```

10. Other Useful Methods

```
int size = fruits.size();
boolean empty = fruits.isEmpty();
Object[] array = fruits.toArray();
```

5.2.3 LinkedList in Java

1. Overview / Explanation

- LinkedList is a **doubly-linked list** implementation of the List and Deque interfaces.
- Allows **duplicates** and **maintains insertion order**.
- Ideal for **frequent insertions and deletions** (especially in the middle or start).
- Slower than ArrayList for **random access** because there's no index-based storage internally.

Use Case: Implementing queues, playlists, undo functionality.

2. Declaration

```
LinkedList<String> list;              // Using class type
List<String> names;                   // Using interface type
```

3. Instantiation

```
list = new LinkedList<>();
names = new LinkedList<>();
```

Full example:

```
LinkedList<String> cities = new LinkedList<>();
```

4. Adding Elements

```
cities.add("Chennai");
cities.add("Mumbai");
cities.add("Delhi");

cities.addFirst("Kolkata");    // Adds at the beginning
cities.addLast("Bangalore");   // Adds at the end
```

5. Accessing Elements / Iteration

a) Index-based Access:

```
System.out.println(cities.get(2));
```

b) For Loop:

```
for (int i = 0; i < cities.size(); i++) {
    System.out.println(cities.get(i));
}
```

c) Enhanced For Loop:

```
for (String city : cities) {
```

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

d) Lambda with forEach:

```
cities.forEach(city -> System.out.println(city));
```

6. Updating Elements

```
cities.set(1, "Hyderabad");  // Replace Mumbai with Hyderabad
```

7. Deleting / Removing Elements

a) By Index:

```
cities.remove(3);
```

b) By Value:

```
cities.remove("Delhi");
```

c) Remove First/Last:

```
cities.removeFirst();
cities.removeLast();
```

d) Remove All:

```
cities.clear();
```

8. Searching / Contains Check

```
boolean found = cities.contains("Bangalore");
int index = cities.indexOf("Mumbai");
```

9. Sorting

```
Collections.sort(cities);
```

Descending:

```
cities.sort(Collections.reverseOrder());
```

10. Other Useful Methods

```
String first = cities.getFirst();
String last = cities.getLast();
int size = cities.size();
boolean empty = cities.isEmpty()
```

5.2.4 Stack in Java

1. Overview / Explanation

- Stack is a **Last-In-First-Out (LIFO)** data structure.
- Java provides Stack as a **class** in java.util (it extends Vector).
- You can also implement a stack using Deque (ArrayDeque is preferred in modern Java for stack operations due to performance).

Use Case: Undo operations, expression evaluation, backtracking, function calls.

2. Declaration

```
Stack<Integer> stack;
```

3. Instantiation

```
stack = new Stack<>();
```

Full example:

```
Stack<String> books = new Stack<>();
```

4. Adding Elements (Push)

```
books.push("Java");
books.push("Python");
books.push("C++");
```

5. Accessing Elements / Iteration

a) Iterating using loop:

```
for (String book : books) {
    System.out.println(book);
}
```

b) Access Top Element without Removing:

```
System.out.println(books.peek()); // Returns "C++"
```

6. Updating Elements

Stack doesn't offer direct updating methods (like set(index, value)), but if needed:

```
books.set(1, "C#"); // Replace at index 1
```

(Use with caution — this breaks typical stack usage)

7. Deleting / Removing Elements (Pop)

```
books.pop(); // Removes "C++"
```

You can also remove using:

```
books.remove("Python");
books.remove(0); // Removes by index
```

8. Searching / Contains Check

a) Contains:

```
books.contains("Java");
```

b) Search position from top (1-based index):

```
int pos = books.search("Java"); // 2
```

9. Sorting

Stacks are not meant to be sorted, but it can be done using:

```
Collections.sort(books);
```

(Note: this violates LIFO nature — use only if needed for special cases.)

10. Other Useful Methods

```
boolean empty = books.isEmpty();
int size = books.size();
books.clear();
```

5.2.5 PriorityQueue in Java

1. Overview / Explanation

- PriorityQueue is a **queue** that retrieves elements based on their **priority** rather than the order they were added.
- By default, it behaves like a **Min-Heap** (smallest element has the highest priority).
- It does **not allow null** elements.
- Not thread-safe (use PriorityBlockingQueue for concurrency).

Use Case: Task schedulers, Dijkstra's algorithm, bandwidth management, etc.

2. Declaration

```
PriorityQueue<Integer> pq;
Queue<String> taskQueue;
```

3. Instantiation

```
pq = new PriorityQueue<>();
taskQueue = new PriorityQueue<>();
```

Custom comparator (e.g., for Max-Heap):

```
PriorityQueue<Integer> maxPQ = new PriorityQueue<>(Collections.reverseOrder());
```

4. Adding Elements

```
pq.add(10);
pq.add(5);
pq.add(15);
pq.add(1);
```

Note: Elements are reordered internally to maintain heap property, not insertion order.

5. Accessing Elements / Iteration

a) Peek (retrieve head without removal):

```
System.out.println(pq.peek()); // Will show the smallest element
```

b) Iteration (order not guaranteed):

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

6. Updating Elements

There is **no direct update** method. Remove the element and re-add the updated value.

```
pq.remove(10);
pq.add(12);
```

7. Deleting / Removing Elements

a) Remove head:

```
pq.poll(); // Removes smallest element
```

b) Remove specific element:

```
pq.remove(15);
```

c) Clear all:

```
pq.clear();
```

8. Searching / Contains Check

```
boolean hasFive = pq.contains(5);
```

9. Sorting

Not applicable directly as PriorityQueue manages its internal order based on priority.

To sort, extract elements into a list:

```
List<Integer> sortedList = new ArrayList<>();
while (!pq.isEmpty()) {
    sortedList.add(pq.poll());
}
```

10. Other Useful Methods

```
int size = pq.size();
boolean empty = pq.isEmpty();
Object[] arr = pq.toArray();
```

5.2.6 ArrayDeque in Java

1. Overview / Explanation

- ArrayDeque (Array Double-Ended Queue) is a **resizable array-based implementation** of the Deque interface.
- Allows insertion and deletion from **both ends** (head and tail).
- **Faster than Stack and LinkedList** for stack/queue operations.
- Does **not allow null** elements.
- Can function as:
    - **Queue** (FIFO)
    - **Stack** (LIFO)

Use Case: Undo-redo stack, browser forward/back navigation, queue of tasks.

2. Declaration

```
Deque<String> deque;
ArrayDeque<Integer> intDeque;
```

3. Instantiation

```
deque = new ArrayDeque<>();
intDeque = new ArrayDeque<>(10); // Optional initial capacity
```

4. Adding Elements

a) As Queue:

```
deque.addLast("A");
deque.addLast("B");
deque.addLast("C");
```

b) As Stack:

```
deque.addFirst("X");
deque.addFirst("Y");
```

Other methods:

```
deque.offer("Z");           // Add to tail
deque.offerFirst("Start"); // Add to head
deque.offerLast("End");     // Add to tail
```

5. Accessing Elements / Iteration

a) Peek First and Last:

```
System.out.println(deque.peekFirst());
System.out.println(deque.peekLast());
```

b) Iterate:

```
for (String item : deque) {
    System.out.println(item);
}
```

6. Updating Elements

Like PriorityQueue, **no direct update**; remove and re-insert.

```
deque.remove("A");
deque.add("A_updated");
```

7. Deleting / Removing Elements

```
deque.removeFirst(); // Removes head
deque.removeLast();  // Removes tail
deque.poll();        // Removes head, returns null if empty
deque.clear();       // Clears entire deque
```

8. Searching / Contains Check

```
boolean exists = deque.contains("B");
```

9. Sorting

You can convert to a list and sort:

```
List<String> list = new ArrayList<>(deque);
Collections.sort(list);
```

Then rebuild the deque if needed:

```
deque = new ArrayDeque<>(list);
```

10. Other Useful Methods

```
int size = deque.size();
boolean empty = deque.isEmpty();
```

5.2.7 HashSet in Java

1. Overview / Explanation

- HashSet is a part of Java's Collection Framework that implements the **Set** interface.
- It stores **unique elements only** — no duplicates allowed.
- **No guaranteed order** (not insertion order or sorted).
- Backed by a **hash table**.
- Allows **null** (only one null element).

Use Case: Removing duplicates, membership testing, set operations like union/intersection.

2. Declaration

```
Set<String> names;
HashSet<Integer> numbers;
```

3. Instantiation

```
names = new HashSet<>();
numbers = new HashSet<>(20);  // with initial capacity
```

Full example:

```
HashSet<String> fruits = new HashSet<>();
```

4. Adding Elements

```
fruits.add("Apple");
fruits.add("Banana");
fruits.add("Orange");
fruits.add("Apple"); // Duplicate - will be ignored
```

5. Accessing Elements / Iteration

a) Enhanced for-loop:

```
for (String fruit : fruits) {
    System.out.println(fruit);
}
```

b) Iterator:

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

6. Updating Elements

There is **no direct update** in a set. You need to remove the old value and add a new one:

```
fruits.remove("Orange");
fruits.add("Mango");
```

7. Deleting / Removing Elements

```
fruits.remove("Banana");
fruits.clear(); // removes all elements
```

8. Searching / Contains Check

```
boolean hasApple = fruits.contains("Apple");
```

9. Sorting

Since `HashSet` is **unordered**, you must convert it to a list to sort:

```
List<String> sortedFruits = new ArrayList<>(fruits);
Collections.sort(sortedFruits);
System.out.println(sortedFruits);
```

10. Other Useful Methods

```
int size = fruits.size();
boolean empty = fruits.isEmpty();
```

5.2.8 LinkedHashSet in Java

1. Overview / Explanation

- LinkedHashSet is a **HashSet** with a **predictable iteration order**.
- It **maintains insertion order** using a **doubly-linked list** internally.
- Like HashSet, it:
    - Stores **unique elements only** (no duplicates)
    - Allows **one null**
    - Is **not synchronized**

Use Case: When you want a set with **no duplicates** but also need to **preserve the insertion order**.

2. Declaration

```
Set<String> set;
LinkedHashSet<Integer> numbers;
```

3. Instantiation

```
set = new LinkedHashSet<>();
numbers = new LinkedHashSet<>(20); // with initial capacity
```

Example:

```
LinkedHashSet<String> colors = new LinkedHashSet<>();
```

4. Adding Elements

```
colors.add("Red");
colors.add("Green");
colors.add("Blue");
colors.add("Red"); // Duplicate, will be ignored
```

5. Accessing Elements / Iteration

Maintains the **order in which elements were added**.

```
for (String color : colors) {
    System.out.println(color);
}
```

Or using iterator:

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

6. Updating Elements

Like `HashSet`, there's **no direct update**. You must remove and re-add the element.

```
colors.remove("Blue");
colors.add("Cyan");
```

7. Deleting / Removing Elements

```
colors.remove("Green");
colors.clear(); // removes all elements
```

8. Searching / Contains Check

```
boolean hasRed = colors.contains("Red");
```

9. Sorting

To sort, convert it to a list:

```
List<String> sortedColors = new ArrayList<>(colors);
Collections.sort(sortedColors);
System.out.println(sortedColors);
```

10. Other Useful Methods

```
int size = colors.size();
boolean empty = colors.isEmpty();
```

5.2.9 TreeSet in Java

1. Overview / Explanation

- TreeSet is a **SortedSet** implementation that stores elements in **ascending order** by default.
- Uses a **Red-Black Tree** internally.
- **No duplicates allowed**
- Does **not allow null** elements (unlike HashSet).
- Provides methods to access elements based on sorting order.

Use Case: When you need a **unique set of elements that are automatically sorted**.

2. Declaration

```
Set<Integer> set;
TreeSet<String> cities;
```

3. Instantiation

```
set = new TreeSet<>();
cities = new TreeSet<>();
```

For custom sorting (e.g., descending order):

```
TreeSet<Integer> descSet = new TreeSet<>(Collections.reverseOrder());
```

4. Adding Elements

```
cities.add("Delhi");
cities.add("Mumbai");
cities.add("Chennai");
cities.add("Delhi"); // Duplicate - will be ignored
```

5. Accessing Elements / Iteration

Elements will be returned in **sorted order**:

```
for (String city : cities) {
    System.out.println(city);
}
```

Or using iterator:

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

6. Updating Elements

No direct update; remove the old one and add the updated:

```
cities.remove("Chennai");
cities.add("Hyderabad");
```

7. Deleting / Removing Elements

```
cities.remove("Mumbai");
cities.clear(); // remove all elements
```

8. Searching / Contains Check

```
boolean found = cities.contains("Delhi");
```

9. Sorting

Not needed – TreeSet is **always sorted**.

Custom sorting (e.g., reverse alphabetical):

```
TreeSet<String> reverseCities = new TreeSet<>(Collections.reverseOrder());
reverseCities.addAll(cities);
```

10. Other Useful Methods

```
int size = cities.size();
boolean empty = cities.isEmpty();
```

Additional Navigational Methods:

```
System.out.println(cities.first());   // Smallest
System.out.println(cities.last());    // Largest
System.out.println(cities.higher("Delhi")); // Next greater element
System.out.println(cities.lower("Delhi"));  // Previous smaller element
```

5.2.10 Conversions Involving Java Collections

Array ➞ List

```
String[] fruits = {"Apple", "Banana", "Mango"};
List<String> fruitList = Arrays.asList(fruits);
```

Note: Arrays.asList() returns a **fixed-size list** backed by the array. To make it resizable:

```
List<String> resizableList = new ArrayList<>(Arrays.asList(fruits));
```

2. List ➞ Array

```
List<String> list = new ArrayList<>();
list.add("A");
list.add("B");

String[] array = list.toArray(new String[0]);
```

3. List ➞ Set

```
List<String> names = Arrays.asList("Ravi", "Ravi", "Kiran");
Set<String> uniqueNames = new HashSet<>(names); // removes duplicates
```

4. Set ➞ List

```
Set<String> colors = new HashSet<>();
colors.add("Red");
colors.add("Blue");

List<String> colorList = new ArrayList<>(colors);
```

5. Set ➞ Array

```
Set<Integer> numbers = new HashSet<>();
numbers.add(1);
numbers.add(2);

Integer[] numberArray = numbers.toArray(new Integer[0]);
```

6. Array ➞ Set

```
String[] names = {"Ravi", "Ravi", "Anil"};
Set<String> nameSet = new HashSet<>(Arrays.asList(names));
```

7. Map ➞ Set (of Keys / Values / Entries)

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "A");
map.put(2, "B");

Set<Integer> keys = map.keySet();
Collection<String> values = map.values();
Set<Map.Entry<Integer, String>> entries = map.entrySet();
```

8. Set (of entries) ➞ Map

```
Set<Map.Entry<Integer, String>> entries = map.entrySet();
Map<Integer, String> newMap = new HashMap<>();

for (Map.Entry<Integer, String> entry : entries) {
```

```
    newMap.put(entry.getKey(), entry.getValue());
}
```

9. List ➞ Map (with unique keys)

```
List<String> students = Arrays.asList("A", "B", "C");
```

```
Map<Integer, String> studentMap = new HashMap<>();
for (int i = 0; i < students.size(); i++) {
    studentMap.put(i + 1, students.get(i)); // RollNo => Name
}
```

5.2.11 HashMap in Java

1. Overview / Explanation

- HashMap is a part of the Java Collection Framework.
- Stores **key-value pairs**.
- **No duplicate keys** (keys must be unique, but values can repeat).
- Allows **one null key** and multiple null values.
- **Unordered** – does not maintain insertion or sorted order.
- Internally uses a **hash table** for fast access.

Use Case: Storing mappings like studentID → studentName, productCode → price, etc.

2. Declaration

```
Map<Integer, String> studentMap;
HashMap<String, Integer> ageMap;
```

3. Instantiation

```
studentMap = new HashMap<>();
ageMap = new HashMap<>();
```

4. Adding Elements (put)

```
studentMap.put(101, "Ravi");
studentMap.put(102, "Anu");
studentMap.put(103, "Kiran");
studentMap.put(101, "Raj");  // Overwrites value for key 101
```

5. Accessing Elements (get & iteration)

```
System.out.println(studentMap.get(102));  // Output: Anu
```

Iteration over entries:

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

Iteration using keySet:

```
for (Integer key : studentMap.keySet()) {
    System.out.println("Key: " + key + ", Value: " + studentMap.get(key));
}
```

6. Updating Elements

```
studentMap.put(103, "Karthik"); // replaces "Kiran"
```

7. Deleting Elements

```
studentMap.remove(102);
studentMap.clear(); // removes all entries
```

8. Searching / Contains Check

```
studentMap.containsKey(101);   // true
studentMap.containsValue("Anu"); // true or false
```

9. Sorting

Since HashMap is **unordered**, to sort:

a) By keys (ascending):

```
Map<Integer, String> sortedByKey = new TreeMap<>(studentMap);
```

b) By values:

```
studentMap.entrySet()
    .stream()
    .sorted(Map.Entry.comparingByValue())
    .forEach(System.out::println);
```

10. Other Useful Methods

```
studentMap.size();
studentMap.isEmpty();
```

5.2.12 LinkedHashMap in Java

1. Overview / Explanation

- LinkedHashMap is a **Map** implementation that **preserves the insertion order**.
- Inherits from HashMap, but uses a **doubly-linked list** to maintain order.
- Allows **one null key** and multiple null values.
- **Faster iteration** compared to HashMap because of predictable order.

Use Case: When you need a key-value mapping **with predictable insertion order**.

2. Declaration

```
Map<Integer, String> studentMap;
LinkedHashMap<String, Integer> ageMap;
```

3. Instantiation

```
studentMap = new LinkedHashMap<>();
ageMap = new LinkedHashMap<>();
```

Optional: Create with **access-order** (for LRU cache-like behavior):

```
LinkedHashMap<Integer, String> lruMap = new LinkedHashMap<>(16, 0.75f, true);
```

4. Adding Elements (put)

```
studentMap.put(101, "Ravi");
studentMap.put(102, "Anu");
studentMap.put(103, "Kiran");
studentMap.put(101, "Raj");  // Overwrites value for key 101
```

Insertion order is maintained:

```
101=Raj, 102=Anu, 103=Kiran
```

5. Accessing Elements

```
System.out.println(studentMap.get(102));  // Output: Anu
```

Iteration (in insertion order):

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

6. Updating Elements

```
studentMap.put(103, "Karthik"); // updates "Kiran"
```

7. Deleting Elements

```
studentMap.remove(101);
studentMap.clear(); // removes all entries
```

8. Searching / Contains Check

```
studentMap.containsKey(102);      // true
studentMap.containsValue("Ravi"); // false
```

9. Sorting

If you need to sort:

By keys:

```
Map<Integer, String> sorted = new TreeMap<>(studentMap);
```

By values (using stream):

```
studentMap.entrySet()
    .stream()
    .sorted(Map.Entry.comparingByValue())
    .forEach(System.out::println);
```

10. Other Useful Methods

```
studentMap.size();
studentMap.isEmpty();
studentMap.keySet();
studentMap.values();
```

5.2.13 TreeMap in Java

1. Overview / Explanation

- `TreeMap` is a **Map** implementation that keeps **keys sorted** in **natural order** (or by a custom comparator).
- Uses a **Red-Black Tree** internally.
- **No duplicate keys allowed**.
- **Does not allow `null` keys** (unlike `HashMap`), but allows multiple null values.
- Slower than `HashMap`, but useful when **sorted keys** are required.

Use Case: Whenever you need a **sorted key-value mapping** (e.g., student marks by roll number, product catalog sorted by code).

2. Declaration

```
Map<Integer, String> treeMap;
TreeMap<String, Integer> marksMap;
```

3. Instantiation

```
treeMap = new TreeMap<>();
marksMap = new TreeMap<>();
```

For **custom sorting** (e.g., reverse order):

```
TreeMap<Integer, String> reverseMap = new TreeMap<>(Collections.reverseOrder());
```

4. Adding Elements (put)

```
treeMap.put(103, "Ravi");
treeMap.put(101, "Anu");
treeMap.put(102, "Kiran");
treeMap.put(104, "Raj");
```

🔁 **Automatically sorted by keys**:

```
101=Anu, 102=Kiran, 103=Ravi, 104=Raj
```

5. Accessing Elements

```
System.out.println(treeMap.get(102));  // Output: Kiran
```

Iteration (Sorted Order):

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

6. Updating Elements

```
treeMap.put(102, "Karthik"); // updates "Kiran"
```

7. Deleting Elements

```
treeMap.remove(103);
treeMap.clear(); // remove all entries
```

8. Searching / Contains Check

```
treeMap.containsKey(101);       // true
treeMap.containsValue("Anu");   // true
```

9. Sorting

Already **sorted by keys**. For **custom sorting**, you can use:

```
TreeMap<String, Integer> customMap = new TreeMap<>(Comparator.reverseOrder());
customMap.putAll(marksMap);
```

To sort **by values**, use streams:

```
treeMap.entrySet()
       .stream()
       .sorted(Map.Entry.comparingByValue())
       .forEach(System.out::println);
```

10. Other Useful Methods

```
treeMap.firstKey();     // smallest key
treeMap.lastKey();      // largest key
treeMap.higherKey(102); // next greater key
treeMap.lowerKey(102);  // previous smaller key

treeMap.keySet();
treeMap.values();
```

5.2.14 Comparable vs Comparator in Java

Both are used to compare and sort objects, but they differ in how and where the sorting logic is defined.

1. `Comparable` Interface (Natural Ordering)

Key Points:

- Found in `java.lang`
- Must override `compareTo()`
- Sorting logic is part of the class itself
- Used when a class has a **natural/default ordering**

Syntax:

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

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

    @Override
    public int compareTo(Student s) {
        return this.id - s.id;  // Ascending order by ID
    }
}
```

Example:

```
List<Student> list = new ArrayList<>();
list.add(new Student(102, "Ravi"));
list.add(new Student(101, "Amit"));

Collections.sort(list);  // uses compareTo()
```

2. Comparator Interface (Custom Ordering)

Key Points:

- Found in java.util
- Used for **external or multiple sort strategies**
- Override compare()
- Often used with lambda expressions

Syntax:

```
class NameComparator implements Comparator<Student> {
    public int compare(Student a, Student b) {
        return a.name.compareTo(b.name);
    }
}
```

Example:

```
Collections.sort(list, new NameComparator());
```

Java 8+ Lambda version:

Collections.sort(list, (a, b) -> a.name.compareTo(b.name));

Difference Table

FEATURE	COMPARABLE	COMPARATOR
PACKAGE	java.lang	java.util
METHOD	compareTo(T o)	compare(T o1, T o2)
DEFINES IN	Same class	Separate class or lambda
SORTING TYPE	Natural (default)	Custom (flexible)
AFFECTS	One default sorting logic	Multiple sorting criteria possible
USAGE	Collections.sort(list)	Collections.sort(list, comparator)

Real-Life Analogy

Think of a Student class:

- **Comparable**: “Sort by student roll number” — default logic inside the class.
- **Comparator**: “Sort by name, then by marks, then by DOB” — various strategies based on situation.

Example: Sort by ID, then by Name

```
Collections.sort(list, Comparator
    .comparing(Student::getId)
    .thenComparing(Student::getName));
```

Unit 6: Java 8 Features

6.1.1 Java 8 Features

Java 8 introduced powerful features that made Java more **concise**, **functional**, and **efficient**, especially for working with data and behavior.

1. Lambda Expressions

- Enables writing **anonymous functions** in a concise way.
- Makes code **shorter and cleaner**, especially with collections and threads.

2. Functional Interfaces

- Interfaces with a **single abstract method**, used with lambdas.
- Examples: Runnable, Callable, Comparator.

3. Streams API

- Processes collections using **functional-style operations**.
- Supports methods like filter(), map(), collect(), reduce().

4. Default and Static Methods in Interfaces

- Interfaces can now have **method bodies** (default or static).
- Enables backward compatibility with interface enhancements.

5. Method References

- A **shorthand** for calling methods via lambdas.
- Example: System.out::println.

6. Optional Class

- Helps handle **null values** safely.
- Avoids NullPointerException using isPresent(), orElse(), etc.

7. New Date and Time API

- java.time package provides **modern, immutable**, and **thread-safe** classes like LocalDate, LocalTime, and Period.

8. Collectors

- Used with Streams to **gather results**.
- Example: collect(Collectors.toList()).

6.2.1 Lambda Expressions in Java 8

1. What is a Lambda Expression?

A **lambda expression** is a concise way to represent an **anonymous function** (i.e., a function without a name) that can be passed as an argument or used to implement a **functional interface**.

It provides a **clear and simple syntax** for writing inline behavior.

2. Syntax

(parameters) -> { body }

Variants:

TYPE	EXAMPLE
NO PARAMETER	() -> System.out.println("Hello")
ONE PARAMETER	x -> x * x
MULTIPLE PARAMETERS	(a, b) -> a + b
WITH DATA TYPE (OPTIONAL)	(int a, int b) -> a * b
WITH BLOCK AND RETURN STATEMENT	(a, b) -> { return a + b; }

3. When to Use Lambda?

- To implement **functional interfaces** (interfaces with a single abstract method).
- Common with APIs like:
  - **Runnable**
  - **Comparator**
  - **ActionListener**
  - **Stream operations**

4. Example: Using Lambda with Runnable

Without Lambda:

```
Runnable r1 = new Runnable() {
    public void run() {
        System.out.println("Thread running");
    }
};
new Thread(r1).start();
```

With Lambda:

```
Runnable r2 = () -> System.out.println("Thread running");
new Thread(r2).start();
```

5. Example: Custom Functional Interface

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

public class LambdaDemo {
    public static void main(String[] args) {
        Calculator add = (a, b) -> a + b;
        System.out.println(add.operate(5, 3));  // Output: 8
    }
}
```

6. Lambda with Collections (Streams)

```
List<String> list = Arrays.asList("Java", "Python", "C++");

list.forEach(language -> System.out.println(language));
```

Or use method reference:

```
list.forEach(System.out::println);
```

7. Benefits of Lambda Expressions

FEATURE	BENEFIT
CONCISE	Less boilerplate code
READABLE	Clear and focused on business logic
REUSABLE	Easily pass behavior as parameters
EFFICIENT	Encourages functional programming

6.2.2 Functional Interface in Java 8

1. What is a Functional Interface?

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

- It can have **default** or **static methods** (with implementation), but only **one abstract method**.
- Functional interfaces can be used as the **target types** for **lambda expressions** or **method references**.

2. @FunctionalInterface Annotation

This annotation is optional but recommended.
It helps the compiler **enforce the rule** that the interface should only have **one abstract method**.

```
@FunctionalInterface
interface MyInterface {
    void show();
}
```

3. Example: Custom Functional Interface with Lambda

```
@FunctionalInterface
interface Greetable {
    void greet(String name);
}

public class Test {
    public static void main(String[] args) {
        Greetable g = name -> System.out.println("Hello, " + name);
        g.greet("Alice");
    }
}
```

4. Built-in Functional Interfaces (from java.util.function)

INTERFACE	METHOD	DESCRIPTION
PREDICATE<T>	boolean test(T)	Tests a condition and returns boolean
FUNCTION<T,R>	R apply(T)	Converts input of type T to R
CONSUMER<T>	void accept(T)	Performs action on an object
SUPPLIER<T>	T get()	Supplies a value of type T

5. Examples of Built-in Functional Interfaces

Predicate

```
Predicate<Integer> isEven = x -> x % 2 == 0;
System.out.println(isEven.test(4));  // true
```

Function

```
Function<String, Integer> strLength = s -> s.length();
System.out.println(strLength.apply("Java"));  // 4
```

Consumer

```
Consumer<String> display = s -> System.out.println(s);
display.accept("Hello World");
```

Supplier

```
Supplier<Double> random = () -> Math.random();
System.out.println(random.get());
```

6. Functional Interface with Thread

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

7. Why Use Functional Interfaces?

- Enables **functional programming** in Java.
- Required to use **Lambda Expressions**.
- Promotes **cleaner and more concise code**.
- Encourages **reusability of behavior**.

6.2.3 Java 8 Built-in Functional Interfaces (Deep Dive)

1. Predicate<T>

Purpose:

Represents a **boolean-valued function** of one argument.

Functional Method:

```
boolean test(T t);
```

Common Use:

Used for **filtering** and **conditional logic**.

Example:

```
Predicate<String> startsWithA = s -> s.startsWith("A");
System.out.println(startsWithA.test("Apple"));  // true
```

Chaining with and(), or(), negate():

```
Predicate<String> lengthCheck = s -> s.length() > 3;
Predicate<String> combined = startsWithA.and(lengthCheck);
System.out.println(combined.test("Ace"));  // false
```

2. Function<T, R>

Purpose:

Takes a value of type `T` and returns a value of type `R`.

Functional Method:

```
R apply(T t);
```

Common Use:

Used for **data transformation**.

Example:

```
Function<String, Integer> strToLength = s -> s.length();
System.out.println(strToLength.apply("Java"));  // 4
```

Chaining:

- `andThen()`: executes after current
- `compose()`: executes before current

```
Function<String, String> addPrefix = s -> "Hello " + s;
Function<String, String> toUpper = s -> s.toUpperCase();

System.out.println(addPrefix.andThen(toUpper).apply("john")); // HELLO JOHN
```

3. Consumer<T>

Purpose:

Accepts a value of type `T` and returns nothing (void).

Functional Method:

```
void accept(T t);
```

Common Use:

Used for **printing, logging, or saving** without returning a result.

Example:

```
Consumer<String> printUpper = s -> System.out.println(s.toUpperCase());
printUpper.accept("hello");  // HELLO
```

Chaining with andThen():

```
Consumer<String> printLength = s -> System.out.println(s.length());
```

```
printUpper.andThen(printLength).accept("Java");
```

4. Supplier<T>

Purpose:

Takes **no input** but **returns** a result of type T.

Functional Method:

```
T get();
```

Common Use:

Used for **generating values** like random numbers, timestamps, etc.

Example:

```
Supplier<Double> randomValue = () -> Math.random();
System.out.println(randomValue.get());
```

5. BiFunction<T, U, R>

Purpose:

Takes **two inputs** of types T and U and returns a result of type R.

Functional Method:

```
R apply(T t, U u);
```

Example:

```
BiFunction<Integer, Integer, Integer> multiply = (a, b) -> a * b;
System.out.println(multiply.apply(5, 4));  // 20
```

6. BinaryOperator<T>

- A **special case** of BiFunction<T, T, T>, returns the same type as input.

```
BinaryOperator<Integer> add = (a, b) -> a + b;
System.out.println(add.apply(2, 3)); // 5
```

7. UnaryOperator<T>

- A **special case** of Function<T, T> — one input, one output of same type.

```
UnaryOperator<String> toUpper = s -> s.toUpperCase();
System.out.println(toUpper.apply("hello")); // HELLO
```

6.2.4 Practice Problems with Solutions – Java 8 Functional Interfaces

1. Predicate<T>

Problem:

Filter out all even numbers from a list of integers.

Solution:

```
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

public class PredicateExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(5, 2, 8, 3, 7, 6);
        Predicate<Integer> isEven = n -> n % 2 == 0;

        List<Integer> evenNumbers = numbers.stream()
                                           .filter(isEven)
                                           .collect(Collectors.toList());

        System.out.println(evenNumbers);  // Output: [2, 8, 6]
    }
}
```

2. Function<T, R>

Problem:

Convert a list of strings into their lengths.

Solution:

```
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

public class FunctionExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Java", "Python", "Go");

        Function<String, Integer> strLength = s -> s.length();

        List<Integer> lengths = names.stream()
                                     .map(strLength)
                                     .collect(Collectors.toList());

        System.out.println(lengths);  // Output: [4, 6, 2]
    }
}
```

3. Consumer<T>

Problem:

Print each string in uppercase.

Solution:

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

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

        Consumer<String> printUpper = s -> System.out.println(s.toUpperCase());

        fruits.forEach(printUpper);
        // Output: APPLE BANANA MANGO
    }
}
```

4. Supplier<T>

Problem:

Generate and print 5 random double values.

Solution:

```
import java.util.function.*;
import java.util.stream.*;

public class SupplierExample {
    public static void main(String[] args) {
        Supplier<Double> randomSupplier = () -> Math.random();

        List<Double> randomNumbers = Stream.generate(randomSupplier)
                                           .limit(5)
                                           .collect(Collectors.toList());

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

5. BiFunction<T, U, R>

Problem:

Create a full name from first and last name.

Solution:

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

public class BiFunctionExample {
    public static void main(String[] args) {
        BiFunction<String, String, String> fullName =
            (first, last) -> first + " " + last;

        System.out.println(fullName.apply("John", "Doe"));  // John Doe
    }
}
```

6. BinaryOperator<T>

Problem:

Add two integers.

Solution:

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

public class BinaryOperatorExample {
    public static void main(String[] args) {
        BinaryOperator<Integer> add = (a, b) -> a + b;
        System.out.println(add.apply(10, 15));  // Output: 25
    }
}
```

7. UnaryOperator<T>

Problem:

Add 10 to each number in a list.

Solution:

```
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

public class UnaryOperatorExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4);
        UnaryOperator<Integer> addTen = n -> n + 10;

        List<Integer> updated = numbers.stream()
                                       .map(addTen)
                                       .collect(Collectors.toList());

        System.out.println(updated);  // Output: [11, 12, 13, 14]
    }
}
```

6.2.5 Java 8 Stream API – Complete Guide

1. What is the Stream API?

The **Stream API** allows you to process **collections** (like List, Set, etc.) in a **functional style**.
It provides a high-level abstraction for processing sequences of elements with operations like filtering, mapping, sorting, and collecting.

Think of it like a **conveyor belt** — elements flow through a pipeline of operations.

2. Key Features

- Works with **Collections** and **arrays**
- Supports **lazy** and **parallel** operations
- Allows **pipelining** of multiple operations
- Promotes **declarative programming**

3. Stream Pipeline Structure

Collection -> Stream -> Intermediate Operations -> Terminal Operation -> Result

Example:

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

names.stream()
     .filter(s -> s.startsWith("A"))
     .map(String::toUpperCase)
     .forEach(System.out::println);  // Output: ALICE
```

4. Stream Creation

```
List<String> list = Arrays.asList("a", "b", "c");
Stream<String> stream = list.stream();
Stream<String> stream2 = Stream.of("x", "y", "z");
```

5. Intermediate Operations (returns Stream)

METHOD	DESCRIPTION
.FILTER()	Filters elements using a predicate
.MAP()	Transforms each element
.SORTED()	Sorts the elements
.DISTINCT()	Removes duplicates
.LIMIT(N)	Limits the result to n elements
.SKIP(N)	Skips the first n elements
.PEEK()	Debug stream content during processing

6. Terminal Operations (returns a result or side-effect)

METHOD	DESCRIPTION
.FOREACH()	Performs an action for each element
.COLLECT()	Converts to List, Set, Map, etc.
.COUNT()	Counts number of elements
.REDUCE()	Reduces elements to a single value
.ANYMATCH()	Checks if any element matches criteria
.ALLMATCH()	Checks if all elements match
.NONEMATCH()	Checks if no elements match
.FINDFIRST()	Gets the first element (Optional)
.FINDANY()	Gets any one element (Optional)

7. Common Use Cases with Examples

Filter and Print Names Starting with 'A'

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

Convert List of Strings to Uppercase

```
List<String> upper = names.stream()
                         .map(String::toUpperCase)
                         .collect(Collectors.toList());
```

Find Length of Each String

```
List<Integer> lengths = names.stream()
                            .map(String::length)
                            .collect(Collectors.toList());
```

Sum of Integers using reduce

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4);
int sum = nums.stream()
              .reduce(0, (a, b) -> a + b);
System.out.println(sum); // 10
```

Count Unique Elements

```
long count = nums.stream()
                 .distinct()
                 .count();
```

Sort Strings by Length

```
List<String> sorted = names.stream()
                           .sorted(Comparator.comparing(String::length))
                           .collect(Collectors.toList());
```

8. Collecting Results

Use Collectors utility class for collecting stream data:

```
import java.util.stream.Collectors;

List<String> result = names.stream()
                           .filter(n -> n.length() > 3)
                           .collect(Collectors.toList());
```

Great! Let’s go through **default and static methods in interfaces** — one of the key enhancements in Java 8 to support better interface design.

6.2.6 Java 8: Default and Static Methods in Interfaces

Why Were These Introduced?

Before Java 8, interfaces could only have **abstract methods** — meaning all implementing classes had to provide their own definitions.

This made it hard to:

- **Add new methods** to interfaces without breaking existing code.
- Provide **shared behavior** among multiple classes.

Java 8 introduced default and static methods to solve these problems.

1. Default Methods

What is a Default Method?

A method in an interface that has a **default implementation** — introduced using the default keyword.

Syntax:

```
interface Vehicle {
    default void start() {
        System.out.println("Vehicle is starting...");
    }
}
```

Example:

```
interface Vehicle {
    default void start() {
        System.out.println("Vehicle starting...");
    }
}

class Car implements Vehicle {
    // Inherits default method unless overridden
}

public class Test {
    public static void main(String[] args) {
        Car c = new Car();
        c.start();  // Output: Vehicle starting...
    }
}
```

2. Static Methods

What is a Static Method in Interface?

Static methods belong to the **interface itself**, not to instances. They are called using the **interface name**.

Syntax:

```
interface Utility {
    static void log(String msg) {
        System.out.println("LOG: " + msg);
    }
}
```

Example:

```
public class Test {
    public static void main(String[] args) {
        Utility.log("Running app..."); // Output: LOG: Running app...
    }
}
```

Default vs Static

FEATURE	DEFAULT METHOD	STATIC METHOD
ACCESSED VIA	Object/instance	Interface name
INHERITED?	Yes (can be overridden)	No (not inherited)
USE CASE	Provide shared implementation	Provide utility/helper methods

Handling Multiple Inheritance Conflicts

If a class implements **two interfaces** with the same default method, the class **must override** that method to avoid conflict.

Example:

```
interface A {
    default void show() {
        System.out.println("From A");
    }
}

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

class C implements A, B {
    public void show() {
        A.super.show();  // or B.super.show();
    }
}
```

Real-World Analogy

- default method → Like a **default setting** in a mobile app — you can override it if needed.
- static method → Like a **utility tool** available in the app's settings — always there, same for all users.

Best Practices

- Use default methods **only** when adding new behavior to existing interfaces to maintain backward compatibility.
- Keep interfaces **clean and focused** — avoid overusing default/static for logic-heavy methods.

Sure! Let's dive into **Method References** — one of the most concise and elegant features introduced in Java 8.

6.2.7 Java 8: Method Reference

What is a Method Reference?

A **method reference** is a **shorthand syntax** for a **lambda expression** that simply calls an existing method.

In other words:

If a lambda looks like this:

```
s -> System.out.println(s)
```

You can simplify it using a method reference:

```
System.out::println
```

Syntax Forms

TYPE	SYNTAX EXAMPLE	USED FOR
STATIC METHOD	ClassName::staticMethod	Math::max, Integer::parseInt
INSTANCE METHOD (OF OBJECT)	obj::instanceMethod	System.out::println
INSTANCE METHOD (OF TYPE)	ClassName::instanceMethod	String::length, String::toUpperCase
CONSTRUCTOR REFERENCE	ClassName::new	ArrayList::new, Employee::new

1. Reference to a Static Method

Lambda:

```
Function<Integer, String> func = n -> String.valueOf(n);
```

Method Reference:

```
Function<Integer, String> func = String::valueOf;
```

2. Reference to an Instance Method of a Particular Object

Lambda:

```
Consumer<String> printer = s -> System.out.println(s);
```

Method Reference:

```
Consumer<String> printer = System.out::println;
```

3. Reference to an Instance Method of an Arbitrary Object of a Particular Type

Lambda:

```
Function<String, Integer> lengthFunc = s -> s.length();
```

Method Reference:

```
Function<String, Integer> lengthFunc = String::length;
```

This is **very common** when working with streams:

```
List<String> names = Arrays.asList("Java", "Python", "C");
```

```
names.stream().map(String::toUpperCase).forEach(System.out::println);
```

4. Reference to a Constructor

Lambda:

```
Supplier<List<String>> listSupplier = () -> new ArrayList<>();
```

Method Reference:

```
Supplier<List<String>> listSupplier = ArrayList::new;
```

Also supports parameterized constructors:

```
Function<String, StringBuilder> sbFunc = StringBuilder::new;
System.out.println(sbFunc.apply("Hello"));  // Output: Hello
```

Real-World Example

Problem:

Convert a list of strings to uppercase and print them.

Using Lambda:

```
names.stream()
     .map(s -> s.toUpperCase())
     .forEach(s -> System.out.println(s));
```

Using Method References:

```
names.stream()
     .map(String::toUpperCase)
     .forEach(System.out::println);
```

When to Use Method Reference?

Use it **only when**:

- The lambda calls a method directly
- It improves **readability** and **brevity**

Dont

This won't work:

```
Function<String, String> f = s -> s.concat("!");
```

You **cannot** write:

```
Function<String, String> f = String::concat;
```

Unless you already know that the second argument will be provided later. So be mindful about method **signatures**.

6.2.8 Java 8: `Optional` Class – A Guide to Avoid NullPointerException

What is Optional<T>?

Optional is a **container object** which may or may not contain a non-null value.

Think of it as a **box**:

- It may contain a value.
- ❌ It may be empty.

It forces you to **explicitly check** whether a value is present — helping you write **null-safe code**.

Why Use Optional?

Before Java 8:

```
String name = getName();
if (name != null) {
   System.out.println(name.length());
}
```

With Optional:

```
Optional<String> name = getName();
name.ifPresent(n -> System.out.println(n.length()));
```

Creating Optionals

METHOD	DESCRIPTION
OPTIONAL.OF(VALUE)	Creates Optional with a **non-null** value
OPTIONAL.OFNULLABLE(V)	Allows **null** or non-null
OPTIONAL.EMPTY()	Creates an **empty** Optional

Examples:

```
Optional<String> a = Optional.of("Java");         // Valid
Optional<String> b = Optional.ofNullable(null);   // Empty Optional
Optional<String> c = Optional.empty();            // Explicitly empty
```

Common Methods

METHOD	DESCRIPTION
ISPRESENT()	Returns true if value is present
IFPRESENT(CONSUMER)	Executes if value exists
GET()	Returns value, throws NoSuchElementException if empty (⚠ risky)
ORELSE(DEFAULT)	Returns value or default if empty
ORELSEGET(SUPPLIER)	Returns value or uses Supplier to compute default
ORELSETHROW()	Throws exception if value is empty
MAP(FUNCTION)	Transforms the value inside Optional
FILTER(PREDICATE)	Returns Optional if value passes filter
FLATMAP()	For nested Optionals

Examples

1. Using of and get

```
Optional<String> name = Optional.of("Alice");
System.out.println(name.get()); // Alice
```

2. Avoid null

```
Optional<String> name = Optional.ofNullable(null);
System.out.println(name.orElse("Default")); // Output: Default
```

3. ifPresent

```
name.ifPresent(n -> System.out.println("Hello " + n));
```

4. map and filter

```
Optional<String> name = Optional.of("Alice");

name.filter(n -> n.startsWith("A"))
    .map(String::toUpperCase)
    .ifPresent(System.out::println);  // Output: ALICE
```

Real-World Use Case: Avoiding Null in Service/DAO Return

```
public Optional<User> findUserById(int id) {
    User user = dao.find(id);
    return Optional.ofNullable(user);
}
```

In client code:

```
Optional<User> userOpt = service.findUserById(1);
userOpt.ifPresent(user -> System.out.println(user.getName()));
```

Don't Misuse Optional

✕ WRONG USAGE	BETTER ALTERNATIVE
AS METHOD PARAMETER	Use regular object (nullable)
IN CLASS FIELDS	Avoid — makes code noisy
FOR EVERY VALUE BLINDLY	Use only when null is expected

Summary

TASK	TRADITIONAL	WITH OPTIONAL
CHECK FOR NULL	if (obj != null)	optional.isPresent()
SAFE USE OF VALUE	if != null then	optional.ifPresent()
DEFAULT FALLBACK	if == null ? x	optional.orElse(x)
CHAINING OPERATIONS	Complex checks	optional.map().filter()

6.2.9 Java 8 Date and Time API (`java.time`)

Why a New API?

Old APIs like Date, Calendar, and SimpleDateFormat:

- Were **not thread-safe**
- Had **poor API design**
- Mixed **mutability** and **confusing behavior**
- Lacked **timezone support**

Java 8 solved this with a **cleaner, immutable, and thread-safe** Date-Time API inspired by Joda-Time.

Core Classes in java.time

Class	Purpose
LocalDate	Date without time (e.g., 2025-06-05)
LocalTime	Time without date (e.g., 10:15:30)
LocalDateTime	Date + Time (no timezone)
ZonedDateTime	Date + Time + Timezone
Period	Difference between dates (in years, months, days)
Duration	Difference between times (in seconds, nanos)
DateTimeFormatter	Formatting and parsing dates and times

1. LocalDate, LocalTime, LocalDateTime

LocalDate

```
LocalDate date = LocalDate.now(); // today's date
LocalDate dob = LocalDate.of(1995, 12, 15);
System.out.println(dob.getYear());  // 1995
System.out.println(dob.plusDays(5)); // 1995-12-20
```

LocalTime

```
LocalTime time = LocalTime.now();     // e.g., 14:23:45
LocalTime specific = LocalTime.of(9, 30);
System.out.println(specific.plusHours(2)); // 11:30
```

LocalDateTime

```
LocalDateTime dateTime = LocalDateTime.now();
System.out.println(dateTime);  // e.g., 2025-06-05T14:23:45
```

2. ZonedDateTime and ZoneId

Handle timezones accurately:

```
ZonedDateTime zoned = ZonedDateTime.now();
System.out.println(zoned);  // Includes offset and zone

ZoneId zone = ZoneId.of("Asia/Kolkata");
ZonedDateTime istTime = ZonedDateTime.now(zone);
System.out.println(istTime);
```

3. Period and Duration

Period (for LocalDate)

```
LocalDate start = LocalDate.of(2020, 1, 1);
LocalDate end = LocalDate.now();
Period p = Period.between(start, end);
System.out.println(p.getYears() + " years " + p.getMonths() + " months");
```

Duration (for LocalTime)

```
LocalTime t1 = LocalTime.of(10, 0);
LocalTime t2 = LocalTime.of(12, 30);
Duration d = Duration.between(t1, t2);
System.out.println(d.toMinutes());  // 150
```

4. Formatting and Parsing

Use DateTimeFormatter to convert date/time to/from Strings.

```
LocalDateTime now = LocalDateTime.now();
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm");

String formatted = now.format(formatter);
System.out.println(formatted);  // e.g., 05-06-2025 14:25

LocalDateTime parsed = LocalDateTime.parse("01-01-2020 10:00", formatter);
System.out.println(parsed);
```

5. Conversion with Old API

```
Date date = new Date();
Instant instant = date.toInstant();
LocalDateTime ldt = LocalDateTime.ofInstant(instant, ZoneId.systemDefault());
```

Summary of Improvements

Feature	Old API	New API (java.time)
Thread-safe	❌	✅
Immutable	❌	✅
Readable	❌ Complex API	Clear, fluent API
Timezones	Limited, error-prone	Full ZoneId, ZonedDateTime
Formatting	Confusing SimpleDateFormat	Powerful DateTimeFormatter

Real-World Use Case

Imagine a booking system:

- LocalDate for event date
- LocalTime for time slots
- ZonedDateTime for global user support
- Period to calculate membership duration
- DateTimeFormatter for displaying times cleanly

Absolutely! Let's dive into `Collectors` — a powerful utility class in the **Java Stream API** that helps collect the result of a stream into a collection or summary.

6.2.9 Java 8 Collectors (from `java.util.stream.Collectors`)

What Are Collectors?

Collectors are utility methods that transform a stream's elements into:

- **Collections** (List, Set, Map)
- **Summarized values** (sum, avg, count)
- **Grouped data**
- **Joined Strings**

They work with the Stream.collect() method.

```
List<String> names = list.stream().collect(Collectors.toList());
```

Commonly Used Collectors

COLLECTOR	DESCRIPTION
TOLIST()	Collect elements into a List
TOSET()	Collect elements into a Set
TOMAP(KEYMAPPER, VALUEMAPPER)	Collect elements into a Map
JOINING()	Concatenate strings
COUNTING()	Count number of elements
SUMMARIZINGINT() / SUMMARIZINGDOUBLE()	Returns count, sum, min, avg, max
GROUPINGBY(CLASSIFIER)	Group elements based on a property
PARTITIONINGBY(PREDICATE)	Split into true/false lists
MAPPING()	Map + Collect in nested collectors

Examples

1. toList()

```
List<String> names = Stream.of("A", "B", "C")
    .collect(Collectors.toList());
```

2. toSet()

```
Set<Integer> nums = Stream.of(1, 2, 2, 3)
    .collect(Collectors.toSet());  // Removes duplicates
```

3. toMap()

```
List<String> words = Arrays.asList("Java", "Python");

Map<String, Integer> wordLengths = words.stream()
    .collect(Collectors.toMap(w -> w, w -> w.length()));

If keys may duplicate, use merge function:

.collect(Collectors.toMap(w -> w, w -> 1, Integer::sum));
```

4. joining()

```
List<String> list = Arrays.asList("One", "Two", "Three");

String result = list.stream()
    .collect(Collectors.joining(", "));  // One, Two, Three
```

5. counting()

```
long count = list.stream()
    .collect(Collectors.counting());
```

6. summarizingInt()

```
IntSummaryStatistics stats = Stream.of(1, 2, 3, 4)
    .collect(Collectors.summarizingInt(i -> i));

System.out.println(stats.getAverage());  // 2.5
```

7. groupingBy()

```
class Student {
    String name;
    String dept;

    Student(String name, String dept) {
```

```java
        this.name = name;
        this.dept = dept;
    }
}

List<Student> students = Arrays.asList(
    new Student("A", "CSE"),
    new Student("B", "ECE"),
    new Student("C", "CSE")
);

Map<String, List<Student>> grouped = students.stream()
    .collect(Collectors.groupingBy(s -> s.dept));
```

8. partitioningBy()

```java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

Map<Boolean, List<Integer>> evenOdd = numbers.stream()
    .collect(Collectors.partitioningBy(n -> n % 2 == 0));
```

9. mapping() (Nested collection transform)

```java
Map<String, List<String>> deptNames = students.stream()
    .collect(Collectors.groupingBy(
        s -> s.dept,
        Collectors.mapping(s -> s.name, Collectors.toList())
    ));
```

Summary

TASK	COLLECTOR
CONVERT TO LIST	toList()
CONVERT TO SET	toSet()
CONVERT TO MAP	toMap()
CONCATENATE STRINGS	joining()
COUNT ELEMENTS	counting()
GET STATS (AVG, MIN, MAX)	summarizingInt()
GROUP BY FIELD	groupingBy()
PARTITION BY TRUE/FALSE	partitioningBy()
COLLECT & TRANSFORM	mapping()