Java Basics

Get Started (Java Basics)

1. What is Java?

Java is a **high-level, object-oriented, platform-independent programming language**.
Key features:

- **Write Once, Run Anywhere (WORA)** — Java code runs on any machine that has JVM.
- **OOP (Object-Oriented Programming)** — Everything is organized around classes & objects.
- **Secure & robust** — Strong memory management and error handling.
- **Used in:** Android apps, financial trading systems, backend systems, games, cloud applications, etc.

2. How Java Programs Run (Compilation Flow)

Java program flow:

1. You write a .java file → **Java source code**
2. The Java compiler (javac) converts it to .class → **Bytecode**
3. JVM (Java Virtual Machine) runs the bytecode

```
Java File (.java) → Compiler → Bytecode (.class) → JVM → Output on screen
```

3. Your First Java Program

Example: Basic "Hello, World!"

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Understanding the program:

- **public class Main**
  A class named Main. Every Java program must have at least one class.
- **public static void main(String[] args)**
  Entry point of the program. JVM starts executing from here.
- **System.out.println()**
  Used to print output on the screen.

4. How to Run Java Program

Using command line (if needed)

1. Save your file as **Main.java**

2. Compile:

```
javac Main.java
```

1. Run:

```
java Main
```

Numbers, Strings & Characters in Java

1. Numbers in Java

Java has two main types of numbers:

A) Integer Types (whole numbers)

Type	Size	Example
byte	1 byte	-128 to 127
short	2 bytes	-32768 to 32767
int	4 bytes	-2 billion to 2 billion
long	8 bytes	very large values

Most commonly used = **int**

Example:

```
int age = 25;
long population = 1400000000L; // L is required for long
```

B) Floating-Point Types (decimal numbers)

Type	Size	Example
float	4 bytes	3.14f (ends with f)
double	8 bytes	3.141592653

Most commonly used = **double**

Example:

```
double price = 99.99;
float temperature = 36.5f;
```

2. Characters (char)

- char stores a **single character**
- Must be inside **single quotes ' '**

Example:

```
char grade = 'A';
char symbol = '#';
char digit = '5'; // still a character, not number
```

✔ A char can also store **Unicode characters** (emojis, Hindi, symbols)

```
char smile = '😊';
char hindi = 'क';
```

3. Strings (text)

- A **String** stores multiple characters

- Always enclosed inside **double quotes " "**
- String is a *class* in Java (not primitive type)

Example:

```
String name = "Sagar";
String message = "Welcome to Java!";
```

String + Number (Concatenation)

You can combine strings and numbers using +:

```
String result = "Age: " + 25;
System.out.println(result);
```

4. Using Numbers & Strings Together

Example Program:

```
public class Main {
    public static void main(String[] args) {
        String name = "Sagar";
        int age = 24;
        double height = 5.9;

        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Height: " + height);
    }
}
```

Here is the **short and crisp version** of the topic **Comments in Java**:

Java Comments (Short Notes)

1. Single-line Comment

Used for short notes.

```
// This is a single-line comment
```

2. Multi-line Comment

Used for longer explanations.

```
/*
This is a multi-line
comment
*/
```

3. Documentation Comment

Used to create Java documentation (advanced).

```
/**
 * This is a documentation comment
 */
```

Why comments?

- Explain code

- Improve readability
- Disable code during testing
- Help other developers understand logic

Great! Let's cover **Variables** and **Variables (II)** together in a clear and simple way.

Variables in Java — Short & Clear Notes

1. What is a Variable?

A **variable** is a container used to store data.
Example:

```
int age = 25;
```

Here:

- int → data type
- age → variable name
- 25 → value stored

2. Rules for Variable Names

You **can** use:

- letters (a–z, A–Z)
- numbers (0–9) (not at the start)
- underscore _
- dollar $

You **cannot**:

- start with numbers (1age ❌)
- use spaces (my age ❌)
- use keywords (int, class, etc.)

✔ Good variables:
age, totalMarks, price_of_item

3. Declaration & Initialization

Declaration (create variable):

```
int number;
```

Initialization (give value):

```
number = 10;
```

Both together:

```
int number = 10;
```

4. Changing Variable Values

You can change values later:

```
int x = 10;
x = 20;  // new value
```

5. Different Data Types (Quick Guide)

✔ Integer types
```
int age = 24;
long population = 1000000L;
```
✔ Decimal numbers
```
double price = 99.99;
float speed = 45.5f;
```
✔ Characters
```
char grade = 'A';
```
✔ Strings
```
String name = "Sagar";
```

6. Final Variables (Constants)
If you want a variable that **cannot be changed**, use final:
```
final int MONTHS = 12;
```
Trying to change it → error.

7. Multiple Variable Declaration
You can declare multiple variables of same type:
```
int a = 10, b = 20, c = 30;
```
Or:
```
int x, y, z;
x = 5;
y = 10;
z = 15;
```

8. Concatenation (joining variables)
```
String name = "Sagar";
int age = 24;

System.out.println(name + " is " + age + " years old");
```
Output:
```
Sagar is 24 years old
```

You're right — **memory management for variables** is an important concept, and it normally belongs with “Variables (II)”.
Let me complete that part now in a clear and simple way.

Memory Management in Java (for Variables)
Java stores different types of variables in **different memory areas**:

1. Stack Memory (for Primitive Variables + References)
Stack stores:
- **primitive type values**
  (int, double, char, boolean, etc.)
- **references to objects** (addresses pointing to heap)

Stack is:

- Fast
- Small
- Auto-managed (memory freed when the method ends)

✔ Example:

```
int a = 10;
double price = 99.9;
```

All these values (10, 99.9) are stored in **stack**.

2. Heap Memory (for Objects & Strings)

Heap stores:

- **Objects**
- **Arrays**
- **Strings**
- **Class instances**

Heap is:

- Larger
- Shared across application
- Garbage-collected

✔ Example:

```
String name = "Sagar";
```

Memory:

- name (reference) → stored in **stack**
- Actual text "Sagar" → stored in **heap**

3. How Memory Works in a Program

Example:

```
int age = 25;
String city = "Pune";
```

Memory placement:

Variable	Memory	Notes
age	Stack	stores 25 directly
city	Stack → Heap	stack stores reference, heap stores actual text

4. Variables Inside Methods

Whenever you call a method, Java creates a **new stack frame**.

Example:

```
public void test() {
    int x = 10;       // stored in stack
    String msg = "Hi"; // reference in stack, object in heap
}
```

After method finishes:

- x is removed
- msg reference removed
- "Hi" in heap may be removed later by **Garbage Collector**

5. Garbage Collection (GC)
Java automatically deletes unused objects.
Example:
```
String name = "Sagar";
name = "Rohan";  // old "Sagar" becomes garbage
```
Old object "Sagar" loses reference → garbage collector will clean it later.

Perfect — we will cover each topic **one by one**, in detail, but **without assignments** (as you requested).
Let's begin.

Printing Output in Java
Java provides multiple methods for printing output to the console.
The most commonly used are:

A) System.out.println()
- Prints text
- Moves cursor to **next line**

Example:
```
System.out.println("Hello Java");
System.out.println("Welcome!");
```
Output:
```
Hello Java
Welcome!
```

B) System.out.print()
- Prints text
- **Does NOT move to next line**

```
System.out.print("Hello ");
System.out.print("Java");
```
Output:
```
Hello Java
```

C) System.out.printf() (formatted printing)
Used for formatted output (similar to C's printf).
```
System.out.printf("My age is %d", 25);
```
Format specifiers:
- %d → integers
- %f → floating point
- %s → string
- %c → character

Example:
```
System.out.printf("Price: %.2f", 99.456);
```
Output:
```
Price: 99.46
```

D) Concatenating output

You can join variables and strings using +.

```
String name = "Sagar";
int age = 24;

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

E) Escape Sequences

These allow special formatting inside strings.

Escape	Meaning
\n	New line
\t	Tab (spaces)
\"	Double quote
\\	Backslash

Example:

```
System.out.println("Line1\nLine2");
System.out.println("She said \"Hello\"");
```

F) Print using variables

```
int a = 10, b = 20;
System×out×println("Sum = " + (a + b));
```

Great! Let’s move to the next topic.

Arithmetic Operators in Java

Arithmetic operators are used to perform basic mathematical operations on numbers.

1. Basic Arithmetic Operators

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

2. Integer Division vs Floating Division

✔ Integer division (both operands int)

```
System.out.println(7 / 2);   // Output: 3
```

Because 7 and 2 are integers, result is also an integer (decimal part removed).

✔ Floating-point division

```
System.out.println(7.0 / 2); // Output: 3.5
```

3. Modulus % Operator

Gives the **remainder** of division.

Examples:

```
10 % 3 → 1
9 % 2 → 1
15 % 5 → 0
```

Usefulness:

- To check **even or odd**
- To rotate values in loops
- Circular indexing

4. Unary Operators (Increment & Decrement)

✔ Increment ++

```
int x = 5;
x++; // 6
```

✔ Decrement --

```
int x = 5;
x--; // 4
```

5. Prefix vs Postfix

Postfix (x++, x--)

Uses value first, then changes it.

```
int a = 5;
System.out.println(a++); // prints 5, then a becomes 6
```

Prefix (++x, --x)

Changes value first, then uses it.

```
int a = 5;
System.out.println(++a); // prints 6
```

6. Arithmetic with Variables

```
int a = 10, b = 20;

int sum = a + b;
int diff = a - b;
int prod = a * b;
int div = b / a;
int rem = b % a;
```

7. Operator Precedence

Order in which operations occur:

1. *, /, %
2. +, -
3. Left to right evaluation

Example:

```
System.out.println(10 + 5 * 2); // Output: 20
```
Why?
```
5 * 2 = 10
10 + 10 = 20
```

Great! Let's continue.

Data Conversion in Java (Type Casting)

Data conversion means converting one data type into another.
Java provides **automatic** and **manual** ways to do this.

1. Widening Conversion (Automatic)

Also called **implicit casting**.
Happens when you convert a **smaller** data type → **larger** data type.
Java does it **automatically**.

Order (small → large):

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

Example:

```
int a = 10;
double b = a;   // int to double (automatic)
```
Output:
```
10.0
```
✔ No data loss
✔ Automatically handled by Java

2. Narrowing Conversion (Manual)

Also called **explicit casting**.
Happens when you convert a **larger** data type → **smaller** data type.
You must explicitly specify the cast.

Example:

```
double x = 9.7;
int y = (int) x;  // manual cast
```
Output:
```
9
```
✔ Decimal part is **lost**
✔ Must be written manually → (int)

3. Converting Numbers to Strings

Using String.valueOf()
```
int age = 25;
String s = String.valueOf(age);
```
Using concatenation:
```
String s = age + "";
```

4. Converting Strings to Numbers

✔ Convert to integer:

```
String s = "100";
int num = Integer×parseInt(s);
```
✔ Convert to double:
```
String price = "59.99";
double d = Double.parseDouble(price);
```
Be careful:

- If the string is not numeric, program throws **NumberFormatException**.

Example:
```
int n = Integer.parseInt("abc"); // ERROR
```

5. Char ↔ Int Conversion
A char actually stores a Unicode number.
✔ char → int:
```
char c = 'A';
int n = c;   // gives 65
```
✔ int → char:
```
int n = 66;
char c = (char)n;  // 'B'
```

6. Automatic Promotion in Expressions
Java promotes small types to **int** during mathematical operations.
Example:
```
byte a = 10;
byte b = 20;
byte c = (byte)(a + b);  // must cast back
```
Because a + b becomes int automatically.

7. Casting with Overflow
When casting exceeds the range of the target type:
```
int x = 130;
byte b = (byte)x;
```
Output:
```
−126   // overflow
```

Great! Let’s move to the next major topic.

Object-Oriented Programming (OOP) — Clear & Simple Notes
OOP (Object-Oriented Programming) is the **heart of Java**.
It organizes programs using **classes** and **objects**.

1. What is a Class?
A **class** is a blueprint/template for creating objects.
It contains:

- **Variables** (attributes)
- **Methods** (behaviors)

Example:

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

    void drive() {
        System.out.println("Car is driving");
    }
}
```

2. What is an Object?

An **object** is an instance of a class (created from a class).
Example:

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

You can access variables and methods:

```
myCar.color = "Red";
myCar.drive();
```

3. Four Pillars of OOP

Java's OOP is built on 4 main principles:

A) Encapsulation (Data Protection)

Wrap data + methods together and hide internal details using **private** fields and **public** methods.
Example:

```
class Person {
    private int age;

    public void setAge(int a) {
        age = a;
    }

    public int getAge() {
        return age;
    }
}
```

Benefits:

- Data security
- Controlled access

B) Inheritance (Code Reuse)

One class inherits properties of another class using **extends**.

```
class Animal {
```

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

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

Now Dog has:

- eat() (from Animal)
- bark() (its own)

C) Polymorphism (Many Forms)

Same method name → different behavior.

✔ Method Overriding (runtime polymorphism)

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

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

✔ Method Overloading (compile-time polymorphism)

Methods with same name but different parameters.

```
void sum(int a, int b) { }
void sum(double a, double b) { }
```

D) Abstraction (Hide Complexity)

Show essential details, hide unnecessary ones.

Using:

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

Example interface:

```
interface Vehicle {
    void start();
}
```

4. Constructors

Special method used to **create objects**.

Default constructor:

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

Parameterized constructor:

```
class Student {
```

```
    int age;
    Student(int a) {
        age = a;
    }
}
```

5. this Keyword

Used to refer to the current object.

```
class Person {
    int age;
    Person(int age) {
        this×age = age;   // current object's age
    }
}
```

6. Static Members

static belongs to class, not object.

```
class Test {
    static int count = 0;
}
```

Access directly using class name:

```
Test.count;
```

7. Packages

Packages organize classes.

Example:

```
package bank;
```

Importing:

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

8. Real-Life Example (Short)

Class = Blueprint (like Car design)

Object = Actual Car

Attributes:

- color
- model
- speed

Methods:

- start()
- stop()
- brake()

Taking Input in Java

Java provides multiple ways to take input, but the **most commonly used** and easiest is:

Using Scanner Class

To use Scanner, you must import it:
`import java.util.Scanner;`
Then create a Scanner object:
`Scanner sc = new Scanner(System.in);`
Now you can read different types of input using methods of Scanner.

1. Input a String

✔ next() → reads one word

`String name = sc.next();`
If input is:
`Sagar Mali`
Output stored → "Sagar"

✔ nextLine() → reads entire line

`String fullName = sc.nextLine();`
This will store "Sagar Mali".

2. Input an Integer

`int age = sc.nextInt();`

3. Input a Double

`double price = sc.nextDouble();`

4. Input a Character

There is no direct method.
We use string → char:
`char ch = sc.next().charAt(0);`
This reads the **first character**.

5. Input Multiple Values

Example:

```
int a = sc.nextInt();
int b = sc.nextInt();
```

User enters:
`10 20`

6. Scanner Complete Example

```
import java.util.Scanner;

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

        System.out.print("Enter your name: ");
        String name = sc.nextLine();
```

```
        System.out.print("Enter your age: ");
        int age = sc.nextInt();

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

7. Closing Scanner

`sc.close();`
Best practice: Close when you no longer need it.

8. Important Issue: nextLine() + nextInt()

nextInt() does NOT consume the newline.
Example:

```
int age = sc.nextInt();
String name = sc.nextLine(); // this will read empty line!
```

Fix:

```
int age = sc.nextInt();
sc.nextLine();       // consume leftover newline
String name = sc×nextLine();
```

If you use nextInt(), nextDouble(), nextFloat(), etc. → insert a nextLine() before using nextLine().

Java Booleans

A **boolean** in Java can store only two values:

- true
- false

We use booleans in decision-making (if/else, loops, comparisons).

Where do booleans come from?

1) Direct values

```
boolean isJavaFun = true;
boolean isRainy = false;
```

2) Comparison results

Any comparison gives a boolean:

```
boolean result = 5 > 2;     // true
boolean check = (10 == 20); // false
```

3) Methods returning boolean

Many built-in methods return boolean:

```
String s = "Java";
boolean empty = s×isEmpty();  // false
boolean starts = s×startsWith("J"); // true
```

Boolean operators

Booleans work with logical operators (we'll go deeper later):

- && → AND
- || → OR
- ! → NOT

Example:

```
boolean a = true;
boolean b = false;

boolean result1 = a && b; // false
boolean result2 = a || b; // true
boolean result3 = !a;     // false
```

Memory Tip

A boolean answers a YES/NO question.
If the answer is YES → true
If the answer is NO → false

if...else Statement

The **if...else** statement lets your program make decisions based on boolean values.

Basic Structure

```
if (condition) {
    // runs when condition is true
} else {
    // runs when condition is false
}
```

The **condition must be a boolean** (true/false).

Example 1: Simple check

```
int age = 20;

if (age >= 18) {
    System.out.println("You can vote");
} else {
    System.out.println("You cannot vote");
}
```

Example 2: if–else if–else

Use this when you have multiple conditions:

```
int marks = 75;

if (marks >= 90) {
    System.out.println("A grade");
} else if (marks >= 75) {
```

```
    System.out.println("B grade");
} else if (marks >= 60) {
    System.out.println("C grade");
} else {
    System.out.println("Fail");
}
```
Only **one** block executes — the first true condition.

Example 3: Combining with booleans
```
boolean isMember = true;
int purchase = 600;

if (isMember && purchase > 500) {
    System.out.println("Extra discount applied");
} else {
    System.out.println("No additional discount");
}
```

Logical Operators (&&, ||, !)
Logical operators combine or modify boolean values.
These are essential for interview questions.

1) AND → &&
Both conditions must be **true**.
```
if (age >= 18 && hasID) {
    System.out.println("Allowed");
}
```
2) OR → ||
At least **one** must be true.
```
if (score > 90 || bonusPoints > 5) {
    System.out.println("Eligible");
}
```
3) NOT → !
Flips true ⇔ false.
```
boolean isClosed = false;
if (!isClosed) {
    System.out.println("Open now");
}
```

Real example combining them
```
boolean isMember = true;
boolean hasCoupon = false;
int total = 1200;

if ((isMember || hasCoupon) && total > 1000) {
```

```
    System.out.println("Discount applied");
} else {
    System.out.println("No discount");
}
```
Try reading it in English:
If you are a member **OR** you have a coupon
AND
your total is more than 1000...

Quick Tip (very useful in interviews)
&& has higher priority than ||
Just like multiplication has higher priority than addition.
So:
`A || B && C`
is read as:
`A || (B && C)`

while Loop
A **while loop** repeats code *as long as* a condition is true.

Basic structure
```
while (condition) {
    // repeated code
}
```
The condition must become **false** at some point, otherwise you get an **infinite loop**.

🔷 Example 1: Counting
```
int i = 1;

while (i <= 5) {
    System.out.println(i);
    i++;   // VERY important!
}
```
Output: 1 2 3 4 5

🔷 Example 2: Loop until user input
```
Scanner sc = new Scanner(System.in);
String input = "";

while (!input.equals("exit")) {
    input = sc.nextLine();
}
```
The loop continues **until** the user types "exit".

for Loop

A **for loop** is used when you know *how many times* you want to repeat something.

Basic structure

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

This is the same 3 parts as while loops — just in one line.

Example 1: Simple counting

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

Example 2: Reverse loop

```
for (int i = 5; i >= 1; i--) {
    System.out.println(i);
}
```

Example 3: Step by 2

```
for (int i = 0; i <= 10; i += 2) {
    System.out.println(i);
}
```

Example 4: Looping through an array

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

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

Quick Tips

- Use **for** when the loop has a fixed number of iterations.
- Use **while** when you loop until a condition changes (unknown count).
- All 3 parts in for-loop are optional:

```
for (;;) {
    // infinite loop
}
```

Exactly right — **2 4 6** ✓
You're moving smoothly through loops. Let's continue.

break and continue

These are used *inside loops* to control the flow.

1) break → stop the loop immediately

It **exits** the loop, no matter what the condition says.

Example:

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

Output: 1 2 3 4

2) continue → skip the current iteration

It does **NOT** stop the loop —
it just jumps to the **next cycle**.

Example:

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

Output: 1 2 4 5

Using with while loops

```
int i = 1;

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

switch Statement

A **switch statement** selects one block of code to run based on the value of a variable.
It is often cleaner than writing many if...else if chains.

Basic syntax

```
switch (variable) {
    case value1:
        // code
        break;

    case value2:
```

```
        // code
        break;

    default:
        // code if no cases match
}
```

Example 1: Day of week

```
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("Unknown day");
}
```

Why do we use break?

Without break, execution “falls through” to the next case.
Example:

```
int x = 2;

switch (x) {
    case 1:
        System.out.println("One");
    case 2:
        System.out.println("Two");
    case 3:
        System.out.println("Three");
}
```

Output:
Two
Three
Because there are no breaks.

Example 2: Using Strings in switch

Java allows Strings too:

```
String role = "admin";

switch (role) {
```

```
    case "admin":
        System.out.println("Full Access");
        break;

    case "user":
        System.out.println("Limited Access");
        break;

    default:
        System.out.println("No Access");
}
```

Example 3: Multiple cases together

```
int month = 12;

switch (month) {
    case 12:
    case 1:
    case 2:
        System.out.println("Winter");
        break;

    default:
        System.out.println("Not winter");
}
```

Array (What & Why)

An **array** stores multiple values of the **same type** in a single variable.
Example:

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

Here:

- nums is an array of **int**
- It contains 4 elements
- Indexing starts from **0**

Indexes:

Index	Value
0	10
1	20
2	30
3	40

Declaring and Creating an Array

Two steps:

```
int[] arr;       // declaration
arr = new int[5]; // creation (5 elements, all 0 initially)
```

Assigning values
```
arr[0] = 10;
arr[1] = 20;
```
If you try arr[5] → error (index out of range).

Reading values
```
System.out.println(arr[1]); // 20
```

Length of array
```
int len = arr.length;
```
Note: length is a **variable**, not a method. (No brackets.)

Correct — **6** ✓
Nice, let's continue.

Array Input (Taking values from user)
We use **Scanner** to fill an array.

Example: Input 5 numbers
```
Scanner sc = new Scanner(System.in);

int[] arr = new int[5];

for (int i = 0; i < arr.length; i++) {
    arr[i] = sc.nextInt();
}
```
Steps happening:
1. Create array of size 5
2. Loop from index 0 to 4
3. Read each value from user
4. Store it in the array

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

for-each Loop (Enhanced for loop)
A **for-each** loop is used to go through all elements of an array *without using indexes*.
It's simpler and clean — but you **cannot modify array size** or **access specific positions** using it.

Syntax
```
for (int value : arr) {
```

```
    System.out.println(value);
}
```
This means:
For every element in arr, store it in value and run the loop.

Example
```
int[] nums = {10, 20, 30};

for (int x : nums) {
    System.out.println(x);
}
```
Output:
```
10
20
30
```

Limitations (Important for interviews)
You cannot get index
You cannot modify the array's values directly
You cannot skip indices or go backwards
✔ Use it when you only want to **read** all values.

Java Methods

1. What Is a Method?
A method is a block of code with a name that performs a task.
A method has:

Part	Example
Access Modifier	public, private
Optional Modifiers	static, final
Return Type	int, void, String, List<T>
Method Name	processOrder()
Parameters	(int x, String y)
Body	{ … }

2. Basic Method Syntax
```
public returnType methodName(parameters...) {
    // method body
}
```
Example:
```
public int add(int a, int b) {
    return a + b;
}
```

3. Calling Methods
There are two categories:

A) Instance Methods
Require an object.
```
Calculator c = new Calculator();
int sum = c.add(5, 10);
```
B) Static Methods
Called using class name.
```
int sum = Calculator.add(5, 10);
```

4. Return Types
A) Returning a value
```
public boolean isEven(int x) {
    return x % 2 == 0;
}
```
B) Returning nothing (void)
```
public void log(String msg) {
    System.out.println("[LOG] " + msg);
}
```
C) Returning objects
```
public User getUser(int id) {
    return new User(id, "Sagar");
}
```

5. Parameters
Parameters are values passed *into* a method.
Primitive Parameter
```
public void setAge(int age) { }
```
Object Parameter
```
public void updateUser(User u) {
    u.setActive(true); // affects original object
}
```

6. Java Is Pass-By-Value (Important!)
Java **is NOT pass-by-reference**.
Java always passes a **copy of the variable**.
Example:
```
public void change(int x) {
    x = 100;
}

public void changeList(List<String> list) {
    list.add("new");
}
```
Usage:
```
int a = 5;
change(a);
// a is STILL 5
```

```
List<String> names = new ArrayList<>();
changeList(names);
// names now contains "new"
```
✔ Objects can be changed
✘ But object *references* cannot be reassigned from inside the method

7. Method Overloading
Same method name, different parameter types / order / count.
```
public class Printer {
    void print(int x) { System.out.println("int: " + x); }
    void print(String s) { System.out.println("string: " +
s); }
    void print(int x, int y) { System.out.println(x + ", " +
y); }
}
```
Usage:
```
Printer p = new Printer();
p.print(10);
p.print("hello");
p.print(5, 6);
```

8. Method Overriding
Subclass changes behavior of superclass method.
```
class Animal {
    void speak() { System.out.println("Some sound"); }
}

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

9. Static vs Instance Methods
Static (Utility logic)
- Does not need object
- Used for pure logic or helper functions

```
public static double square(double x) {
    return x * x;
}
```
Instance (Object behavior)
- Needs object state

```
public class Account {
    private int balance;
    public void deposit(int amount) { balance += amount; }
}
```

10. Variable Arguments (varargs)
Method that accepts unknown number of parameters.

```
public int sum(int... nums) {
    int total = 0;
    for (int n : nums) total += n;
    return total;
}
```

Usage:

```
sum(1);
sum(1, 2, 3, 4);
```

11. Access Modifiers for Methods

Modifier	Where Accessible	Common Use
public	anywhere	API methods
private	same class	helper logic
protected	subclasses	framework code
default (no modifier)	same package	internal APIs

12. Exceptions in Methods
Declaring checked exceptions:

```
public void readFile(String path) throws IOException {
    Files.readString(Path.of(path));
}
```

Throwing exceptions:

```
public int divide(int a, int b) {
    if (b == 0) throw new IllegalArgumentException("b cannot
be zero");
    return a / b;
}
```

13. Method Chaining (Fluent APIs)

```
public class Builder {
    private String name;

    public Builder setName(String name) {
        this×name = name;
        return this;   // returning 'this' enables chaining
    }
}
```

Usage:

```
new Builder().setName("Sagar");
```

14. Pure vs Impure Methods
Pure method

- No side effects
- Same output for same input

```
public int doubleValue(int x) {
    return x * 2;
}
```
Impure method

- Modifies state / interacts with system

```
public void save(User u) {
    repository.save(u);
}
```

15. Real Backend Example

A service method that:

- validates input
- processes
- returns response

```
public class OrderService {

    public OrderResponse placeOrder(OrderRequest req) {
        validate(req);
        int price = calculatePrice(req.getItems());
        String id = saveOrder(req, price);
        return new OrderResponse(id, price);
    }

    private void validate(OrderRequest req) {
        if (req.getItems().isEmpty())
            throw new IllegalArgumentException("Empty
order");
    }

    private int calculatePrice(List<Item> items) {
        int total = 0;
        for (Item i : items) total += i.getPrice();
        return total;
    }

    private String saveOrder(OrderRequest req, int price) {
        return "ORD-" + System.currentTimeMillis();
    }
}
```

This is how real-world backend methods are structured.

Great — let's move to **Subtopic 2: Standard Library Methods**.
To keep this *study-mode friendly*, I'll explain concepts clearly, give examples, and ask small check-questions so you stay actively learning. No homework problems — just learning.

Standard Library Methods

The Java Standard Library (JDK) contains thousands of methods

1. String Methods (most used in backend)

Strings are immutable. All methods return *a new string*.

Common methods:

```
String s = "  hello world  ";
s.trim();          // "hello world"
s.toUpperCase();   // "  HELLO WORLD  "
s.startsWith("he");
s.contains("world");
s.replace("hello", "hi");
s.substring(0, 5);
```

Example:

```
String email = "  Sagar@Barclays.com  ";
String cleaned = email.trim().toLowerCase();
```

2. Math Methods

Useful for calculations, rounding, min/max, random numbers.

```
Math.abs(-10);              // 10
Math.max(10, 20);           // 20
Math.pow(2, 3);             // 8
Math.sqrt(25);              // 5
Math.random();              // 0.0 to 1.0
```

Prefer ThreadLocalRandom.current().nextInt() for high-performance random values.

3. Arrays Utility Methods — Arrays class

Arrays gives helper methods for array operations.

```
int[] arr = {3, 1, 2};

Arrays.sort(arr);           // [1, 2, 3]
Arrays.toString(arr);       // "[1, 2, 3]"
Arrays.binarySearch(arr, 2);
```

Sorted binary search example:

```
int[] nums = {1,2,3,4};
int index = Arrays.binarySearch(nums, 3); // returns 2
```

4. Collections Utility Methods — Collections class

These work with Lists, Sets, Maps.

```
List<Integer> list = new ArrayList<>(List.of(3,1,2));

Collections.sort(list);         // mutates list
Collections.reverse(list);
Collections.max(list);
Collections.unmodifiableList(list);
```

Example:

```
List<String> names = new ArrayList<>();
List<String> safeCopy = Collections.unmodifiableList(names);
```

5. Objects Utility Methods

Null-safe handling:

```
Objects.equals(a, b); // avoids NullPointerException
Objects.requireNonNull(obj, "msg"); // good for validation
```

Example:

```
public void setName(String name) {
    this×name = Objects.requireNonNull(name, "name cannot be
null");
}
```

6. Optional Methods

Optional is used to prevent null returns.

```
Optional<String> name = Optional.of("Sagar");

name.isPresent();
name.ifPresent(n -> System.out.println(n));
name.orElse("default");
name.orElseThrow();
```

Example:

```
return Optional.ofNullable(userMap.get(id));
```

7. File & Path Methods

From java.nio.file.

```
Path p = Path×of("test.txt");
Files.readString(p);
Files.writeString(p, "Hello");
Files.exists(p);
```

Senior-level usage:

```
try (Stream<String> lines = Files.lines(p)) {
    lines.forEach(System.out::println);
}
```

8. Time API Methods — java.time

Modern, immutable, thread-safe.

```
LocalDate today = LocalDate×now();
LocalDate next = today×plusDays(1);

LocalDateTime dt = LocalDateTime.now();
Duration diff = Duration.between(start, end);
```

Example:

```
LocalDateTime created = LocalDateTime.now().minusHours(5);
```

9. Stream Methods (Very powerful)

Streams give you functional-style operations.

Example:
```
List<Integer> list = List×of(1, 2, 3, 4);

int sum = list.stream()
              .filter(x -> x % 2 == 0)
              .mapToInt(x -> x)
              .sum();
```
What this does:
1. filter even numbers
2. convert to primitive stream
3. sum

10. Common Standard Library Patterns
A) Creating immutable collections
```
List<Integer> numbers = List.of(1, 2, 3);
Map<String, Integer> scores = Map.of("a", 1, "b", 2);
```
B) Defensive copies
```
return List.copyOf(internalList);
```
C) Reusing shared objects
```
private static final Pattern EMAIL = Pattern.compile("\\w+@\
\w+\\.\\w+");
```
D) Try-with-resources
```
try (BufferedReader br = Files.newBufferedReader(path)) {
    return br.readLine();
}
```

Great — let's continue with **Subtopic 3: Local Variables**.
I'll teach this in a clean, step-by-step way with examples, and I'll ask small check-questions so you stay actively involved.

Local Variables

1. What Are Local Variables?
A **local variable** is a variable declared inside:
- a method
- a block { }
- a loop (for, while)
- a constructor

It exists **only within that block**.
Example:
```
void process() {
    int x = 10;  // local variable
}
```
You cannot access x outside the method.

2. Scope of Local Variables

Scope = where the variable is visible.

```
void test() {
    int a = 5;        // visible everywhere inside test()

    {
        int b = 10;   // visible only inside this block
        System.out.println(a + b);
    }

    // b is NOT visible here
}
```

Scopes can nest but never expand outward.

3. Lifetime of Local Variables

A local variable exists **while the method is executing**.

Once the method ends, all local variable references are removed from the stack.

Example:

```
public String greet() {
    String name = "Sagar";
    return "Hi " + name;
}
```

name ceases to exist after the method finishes.

4. Final & Effectively Final Variables

A senior-level concept many junior devs miss.

final local variable

Cannot be reassigned.

```
final int x = 10;
x = 20; // ERROR
```

Effectively final

A variable is "effectively final" if you **don't reassign it**, even if you didn't use the keyword.

Why it matters:

Lambdas and anonymous classes **can only use final or effectively final locals**.

Example:

```
int n = 10;  // effectively final

Runnable r = () -> System.out.println(n);
```

If you try to reassign:

```
n = 20;  // ERROR inside lambda
```

5. Primitive vs Reference Local Variables

Primitive (int, long, boolean, double)

Holds the actual value.

```
int a = 5;
```

```
int b = a;
b = 10;
// a is still 5
```
Reference variable (String, List, User)
Holds a reference (pointer) to an object.
```
List<String> list1 = new ArrayList<>();
List<String> list2 = list1;

list2.add("x");
// list1 also has "x"
```
Local variable → reference
Object → in heap

6. Shadowing (avoid this!)
Shadowing = when a local variable has the same name as a field.
```
class Test {
    int value = 10;

    void print() {
        int value = 20;  // shadows field
        System.out.println(value); // prints 20
    }
}
```
Better to avoid naming locals same as fields.

7. Local Variable Type Inference (var)
Introduced in **Java 10**.
Instead of:
```
List<String> list = new ArrayList<>();
```
You can write:
```
var list = new ArrayList<String>();
```
Rules:
- Only for local variables
- Type must be inferable from the right-hand side
- Not allowed for fields or method parameters

Good usage:
```
var map = new HashMap<String, Integer>();
```

8. Local Variable Best Practices (Senior Developer View)
✔ Keep variable scope as small as possible
Small scope = fewer bugs.
✔ Prefer meaningful names
Avoid a, b, x, y except for loops.
✔ Reuse variables carefully
Reusing a variable for multiple purposes reduces clarity.
✔ Limit mutability

Use final where possible for safety.

✔ Don’t declare all variables at top

Declare variables **right before use**, not 20 lines above.

Great — this is an important one.
Subtopic 4: Think Methods is NOT about syntax… it’s about *how senior Java backend developers think when designing methods*.
This is where your coding style becomes “clean”, “professional”, and “interview-ready”.
I’ll teach you step-by-step, with examples, patterns, and short check questions.

Think Methods

1. A Method Should Do ONE Thing (SRP)
SRP = **Single Responsibility Principle**
If your method does too many things, it becomes harder to test, understand, and maintain.
Bad:

```
void processOrder(Order o) {
    validate(o);
    calculatePrice(o);
    updateInventory(o);
    sendEmail(o);
    saveToDB(o);
}
```

This is basically a *controller* in one method.
Better:

```
void processOrder(Order o) {
    validate(o);
    int total = calculatePrice(o);
    updateInventory(o);
    saveToDB(o, total);
    notifyUser(o);
}
```

Even better: break each into private methods.

2. Keep Methods Small (Usually < 20–30 lines)
Shorter methods:

- are easier to read
- easier to test
- easier to debug
- force you to structure logic

A senior developer refactors early so methods don’t “blow up”.

3. Good Method Names > Good Comments
Bad names need comments.
Good names eliminate comments.

Bad:
```
void calc() { ... }        // what does calc mean??
```
Good:
```
int calculateInvoiceAmount(Order order)
```
Patterns senior developers use:

- get → return a field or a value
- calculate → pure computation
- find → may return null/Optional
- fetch → external call
- update/save/delete → side-effect
- validate → throws exception on bad input
- process/handle → orchestrates things

4. Limit Number of Parameters (≤ 4 is good)
If a method needs 6–8 parameters, something is wrong.
Bad:
```
void register(String name, String email, String address, int
age, boolean active, Date created)
```
Better: use a Request object.
```
void register(RegisterRequest req)
```
Or builder pattern.

5. **Design rule:**
Separate pure logic from side-effect logic.
Example:
```
int total = calculator.calculateTotal(order);
paymentService.charge(total);
```

6. Validate Inputs Early
```
public void update(User u) {
    Objects.requireNonNull(u, "User must not be null");

    if (u.getName().isBlank()) {
        throw new IllegalArgumentException("Name cannot be
blank");
    }
}
```
You never trust incoming data.

7. Return Meaningful Types
Bad:
```
int process(Order o); // what does int represent?
```
Good:
```
OrderResult process(Order o);
```
Or:
```
Optional<User> findById(String id);
```
Returning rich objects improves clarity and avoids bugs.

8. Design for Testability
A senior developer writes methods that are easy to unit test.
Bad:
```
void process() {
    System.out.println("Hi");
    Thread.sleep(1000);
    callApi();
}
```
Good:
Separate logic:
```
String createMessage() {
    return "Hi";
}

void sendMessage() {
    System.out.println(createMessage());
}
```
You can test createMessage() easily.

9. Don’t Use Static When You Need Behavior to Vary
Static is good for utilities:
```
Math.max(...)
Collections.sort(...)
```
But not for business logic:
```
static void chargeOrder()   // hard to test, hard to mock
```
Better:
```
class PaymentService {
    void chargeOrder() { ... }
}
```

10. Use Helper Methods to Avoid Repetition
If you repeat logic more than twice, extract method.
Instead of:
```
if (x == null || x.isBlank()) { ... }
if (y == null || y.isBlank()) { ... }
if (z == null || z.isBlank()) { ... }
```
Extract:
```
private void checkBlank(String s, String msg) {
    if (s == null || s.isBlank()) throw new
IllegalArgumentException(msg);
}
```
Now clean.

11. Fail Fast, Not Fail Late
Throw exception early.
Bad:

```
void save(User u) {
    // many operations...
    // finally realize u was null!
}
```
Good:
```
Objects.requireNonNull(u);
```

12. Don’t Return Null (Use Optional)
Bad:
```
User findUser(int id) {
    return null;
}
```
Good:
```
Optional<User> findUser(int id)
```

13. Use Method Chaining for Builders / Fluent APIs
Example:
```
userBuilder
    .setName("Sagar")
    .setAge(25)
    .setActive(true)
    .build();
```
Makes code more readable.

14. Concurrency-Safe Method Design
If multiple threads call the same method:
- avoid modifying shared state
- otherwise use synchronization or concurrent collections

Example safe method:
```
public int incrementAndGet() {
    return counter.incrementAndGet(); // atomic
}
```

Great — perfect connection.
In Java, a **String is exactly that: text or a sentence**, but with some important rules.

What is a String?
A **String in Java is an object** that represents a sequence of characters.
Example:
```
String name = "Sagar";
String message = "Hello World!";
```
Even though you write it like a simple value, behind the scenes it **lives as an object in memory**.

Strings in Java are immutable — once created, they cannot be changed.
Meaning:

```
String s = "abc";
s = s + "d";
```
You didn't modify "abc" — you created a *new* String "abcd".
This immutability is often asked in interviews.

Exactly!
You're thinking correctly — the original "java" is **not** modified. A new String "javarocks" is created.
Great, now let's deepen the concept.

How Strings are Created

There are **two common ways**:
1) String literal (most common)
```
String s1 = "Hello";
```
These are stored in a special memory area called the **String Constant Pool (SCP)**.
2) Using new keyword
```
String s2 = new String("Hello");
```
This *always* creates a new object in heap memory.

Important String Methods

The most commonly used methods:

- length()
- charAt()
- substring()
- toUpperCase(), toLowerCase()
- contains()
- equals() vs ==
- indexOf()
- trim()
- replace()

length()
Returns the number of characters in a string.
```
String name = "Sagar";
int len = name.length();  // 5
```
charAt(index)
Returns the character at a particular position.
```
String s = "Java";
char c = s.charAt(2);   // 'v'
```
Index starts from **0**.

substring(start) and substring(start, end)

Example 1:
```
String s = "Interview";
```

```
System.out.println(s.substring(5));
```
This prints everything **from index 5 to end** → "view"

Example 2:

```
s.substring(0, 5)
```
This prints characters from **index 0 to index 4** (end index is *exclusive*) → "Inter"

equals() vs == (VERY IMPORTANT for interviews)

- equals() → checks **text/content**
- == → checks **memory reference**

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

System.out.println(a == b);      // true (same object in
SCP)
System.out.println(a.equals(b)); // true
```
But:
```
String x = new String("Java");
String y = new String("Java");

System.out.println(x == y);      // false (different
objects)
System.out.println(x.equals(y)); // true (same content)
```
toUpperCase() / toLowerCase()

Convert the whole string to uppercase or lowercase.
```
String s = "Java";
s.toUpperCase(); // "JAVA"
s.toLowerCase(); // "java"
```
Useful in interviews for normalization.

contains()

Checks if a substring exists inside the string.
```
String s = "fullstack developer";

s.contains("stack");   // true
s.contains("java");    // false
```
indexOf()

Returns the **first index** of a character or substring.
```
String s = "developer";
s.indexOf('e');     // 1
s.indexOf("lop");   // 3
s.indexOf("java");  // -1 (not found)
```
If not found → returns **-1**
Interviewers check this.

trim()

Removes leading and trailing spaces.

```
String s = "   hello  ";
s.trim();     // "hello"
```

Useful when taking user input.