

Basics Of Java

Why We Need Java Programming Language?

Java is one of the most widely used programming languages in the world. It was developed by **James Gosling** at **Sun Microsystems** in 1995. Java provides a robust, secure, and platform-independent environment for software development.

Key Reasons Why Java Is Needed

1. Platform Independence (Write Once, Run Anywhere - WORA)

- Java programs are compiled into **bytecode**, which runs on any device with the **Java Virtual Machine (JVM)**.
- No need to rewrite code for different OSes like Windows, Linux, or macOS.

2. Object-Oriented Programming (OOP)

- Java follows the OOP paradigm, which helps in organizing complex programs using real-world modeling.
- Key OOP features in Java: **Class, Object, Inheritance, Polymorphism, Encapsulation, Abstraction.**

3. Rich API (Application Programming Interface)

- Java provides a large number of **built-in classes** and libraries.
- Supports data structures, networking, file handling, GUI (AWT/Swing/JavaFX), multithreading, etc.

4. Security

- Java provides a **secure execution environment**.
- Features like bytecode verification, sandboxing, and secure class loading reduce the risk of malware or memory corruption.

5. Automatic Memory Management (Garbage Collection)

- Java handles memory allocation and deallocation automatically using **Garbage Collector**, preventing memory leaks.

6. Multithreading Support

- Java provides in-built support for writing **multithreaded programs**, enabling faster execution and better CPU utilization.

7. Robust and Reliable

- Java emphasizes **early error checking, strong memory management, and exception handling**, making it a robust language.

8. Community and Support

- Huge community support and rich documentation.
- Continuous updates and frameworks like Spring, Hibernate, etc., make it suitable for enterprise-level applications.

9. Enterprise and Mobile Applications

- Used in **Android app development, web applications, banking systems, e-commerce platforms, and cloud-based applications**.

Where Java Is Used?

Domain	Application Examples
Web Development	JSP, Servlets, Spring Framework
Android Development	Android Studio using Java/Kotlin
Enterprise Applications	ERP, CRM systems
Desktop GUI Applications	Swing, JavaFX
Scientific Applications	MATLAB-based tools, simulations
Cloud Computing	Backend systems, cloud services
Game Development	2D/3D Java games using LibGDX, jMonkey

Java History

What is Java?

Java is a high-level, class-based, object-oriented programming language designed to have as few implementation dependencies as possible. It was initially designed for embedded systems but quickly evolved into one of the most powerful general-purpose programming languages.

Timeline of Java History

Year	Milestone/Event
1991	Project Green initiated by James Gosling, Patrick Naughton, and Mike Sheridan at Sun Microsystems
1995	Official release of Java 1.0 by Sun Microsystems
1996	Java Development Kit (JDK) 1.0 released; 250,000 downloads in a month
1998	Release of Java 2 Platform, Standard Edition (J2SE)
2006	Sun released Java as open-source under GNU GPL license
2009	Oracle Corporation acquired Sun Microsystems
2011	Java 7 released by Oracle
2014	Java 8 released, introducing Lambda expressions, Stream API
2017	Oracle moved Java to a 6-month release cycle
2021	Java 17 released (LTS version)
2023	Java 20 released with more enhancements

Who Created Java?

- **James Gosling** is known as the *Father of Java*.
 - Java was developed as part of the **Green Project** at Sun Microsystems in 1991.
 - Initially, it was called **Oak**, named after an oak tree outside Gosling's office.
-

Why Was Java Created?

- Designed for **interactive television**, but it was too advanced for cable TV at the time.
 - Later, it became ideal for **Internet-based, network-centric applications**.
 - Main goals:
 - Platform independence
 - Simplicity and reliability
 - Security
 - Portability
-

□ Evolution of Java Versions

Java Version	Notable Features
Java 1.0	Initial release with core classes
Java 1.2	J2SE branding, Swing API, Collections
Java 5	Generics, Enhanced for loop, Annotations
Java 8	Lambda expressions, Stream API, Optional class
Java 11	LTS version, local variable syntax in lambda
Java 17	Latest LTS version with enhanced switch expressions, sealed classes

How Java Works Internally

Java uses a **two-step compilation and execution process** to ensure **platform independence, performance, and security**. Let's break it down step-by-step:

Internal Working of Java

Step 1: Writing Java Code

You write your code in a .java file using any text editor or IDE like IntelliJ, Eclipse, or NetBeans.

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

Step 2: Compilation by Java Compiler (javac)

- The Java compiler converts the .java file into a **bytecode** file (.class).
- This bytecode is not specific to any one machine — it's **platform-independent**.

javac HelloWorld.java → HelloWorld.class

Tool used: javac

Output: Platform-independent **Bytecode**

Step 3: Execution by JVM (Java Virtual Machine)

- JVM loads the .class bytecode and interprets or compiles it **at runtime** into machine code using **JIT (Just-In-Time) Compiler**.
- The machine code is then executed by the **CPU**.

□ JVM Responsibilities:

- Class loading (using ClassLoader)
 - Bytecode verification
 - Memory management (Heap & Stack)
 - Exception handling
 - Garbage collection
-

Internal Architecture of Java

1. Java Development Kit (JDK)

- Contains tools like compiler (javac), debugger, documentation tool, etc.
- Used by developers to **write, compile, and debug** Java programs.

2. Java Runtime Environment (JRE)

- Provides environment to **run** Java programs.
- Contains JVM + core libraries (but no compiler).

3. Java Virtual Machine (JVM)

- **Heart of Java** — responsible for converting bytecode to machine code.
- Platform-specific — different JVMs for Windows, Linux, etc.
- Core components of JVM:
 - **ClassLoader**: Loads .class files
 - **Bytecode Verifier**: Ensures code safety
 - **Interpreter + JIT Compiler**: Executes bytecode
 - **Memory Manager**: Handles heap, stack, and garbage collection

Memory Areas in JVM

Memory Area	Purpose
Method Area	Stores class structure, static variables
Heap	Stores objects and instance variables
Stack	Stores method calls, local variables
PC Register	Keeps address of currently executing code
Native Method Stack	Invokes native (non-Java) methods

Flow Diagram:

Java Source Code (.java)



Java Compiler (javac)



Bytecode (.class)



ClassLoader (JVM)



Bytecode Verifier → JIT Compiler → Machine Code



Executed by CPU

JVM vs JRE vs JDK vs JIT in Java

These four components are **core parts** of the Java programming environment. Understanding them is essential for writing, compiling, and running Java programs.

1. JVM (Java Virtual Machine)

Definition:

JVM is an engine that provides a runtime environment to execute Java bytecode.

□ Key Features:

- Converts **bytecode to machine code**.
- Platform-dependent (different JVMs for Windows, Linux, etc.).
- Manages memory (Heap, Stack), security, and garbage collection.

□ **Responsibilities:**

- **Loads** .class files (using ClassLoader)
 - **Verifies** bytecode for safety
 - **Executes** code via **Interpreter + JIT**
 - **Manages memory** and **exceptions**
-

2. JRE (Java Runtime Environment)

Definition:

JRE is the software package that provides everything required to **run** a Java program.

□ **Includes:**

- **JVM**
- **Core libraries** (rt.jar)
- Supporting files (properties, resources)

Note:

- You can **run** Java programs with JRE, but **cannot compile** them.
 - Suitable for **end users** who only need to run Java apps.
-

3. JDK (Java Development Kit)

Definition:

JDK is a full-featured software development kit used to **develop**, **compile**, and **run** Java programs.

□ **Includes:**

- **JRE** (JVM + Libraries)
- **Development tools:**
 - javac (compiler)
 - java (launcher)

- javadoc, jdb, jar, etc.

Note:

- **Must-have for developers.**
 - Multiple JDK versions exist (JDK 8, 11, 17, etc.).
-

4. JIT (Just-In-Time Compiler)

Definition:

JIT is a part of JVM that **improves performance** by compiling bytecode to native machine code **at runtime**.

How It Works:

- JVM first interprets bytecode.
- JIT identifies **frequently used (hotspot)** methods.
- It compiles those into **native machine code** for faster execution.

Benefits:

- Increases **speed and efficiency**
 - Reduces repeated interpretation
-

Summary Table

Component	Full Form	Purpose	Includes
JVM	Java Virtual Machine	Runs Java bytecode	Core runtime engine
JRE	Java Runtime Environment	Runs Java applications	JVM + Libraries
JDK	Java Development Kit	Develops & runs Java apps	JRE + Tools (javac)
JIT	Just-In-Time Compiler	Speeds up execution inside JVM	Part of JVM

Real-World Analogy

Concept	Analogy
JDK	Like a toolbox to build things
JRE	Like the furnace to run things
JVM	Like the engine of the furnace
JIT	Like a turbocharger in the engine that speeds things up

Difference Between Java and Other Programming Languages

Understanding how **Java compares with other languages** like **C, C++, Python, and JavaScript** helps you choose the right language for the right task and understand Java's unique strengths.

Java vs C

Feature	Java	C
Platform Independence	Yes (via JVM)	No (compiled to platform-specific code)
Programming Paradigm	Object-Oriented	Procedural
Memory Management	Automatic (Garbage Collector)	Manual (malloc/free)
Pointers	No direct use	Fully supported
Security	More secure (runtime checks)	Less secure

Feature	Java	C
Compilation	Bytecode compiled and run on JVM	Compiled to machine code

Java vs C++

Feature	Java	C++
Platform Independence	Yes	No
Multiple Inheritance	No (uses interfaces)	Yes
Memory Management	Automatic	Manual
Operator Overloading	Not supported	Supported
Pointers	Not supported	Supported
Compilation	Compiles to bytecode	Compiles to native code
Execution Speed	Slower (due to JVM)	Faster (compiled to machine code)

Java vs Python

Feature	Java	Python
Syntax	Verbose	Simple, concise
Compilation	Compiled + Interpreted (via JVM)	Interpreted
Speed	Faster than Python	Slower than Java

Feature	Java	Python
Typing	Static (type-safe)	Dynamic (more flexible)
Code Length	Longer	Shorter
Popular Use Cases	Android, Web, Enterprise Apps	AI, ML, Data Science, Automation

Java vs JavaScript

Feature	Java	JavaScript
Type	OOP Language	Scripting Language
Runs On	JVM	Web Browser (via engine like V8)
Use Case	Backend, Android, Enterprise Apps	Frontend Web Development
Typing	Static	Dynamic
Compilation	Compiled to bytecode	Interpreted (Just-in-time compiled)
Inheritance	Class-based	Prototype-based

Difference Between Platform Dependent and Platform Independent Languages

Understanding platform dependency is essential in appreciating one of Java's biggest strengths — **"Write Once, Run Anywhere."**

Definitions

Platform Dependent Language:

A language in which the **compiled code** runs **only on the same platform** (OS and hardware) where it was compiled.

Example: C, C++

Platform Independent Language:

A language in which the **compiled code** can run on **any platform**, regardless of the underlying OS or hardware.

Example: Java

Key Differences

Feature	Platform Dependent	Platform Independent
Compilation Output	Machine Code (Platform Specific)	Bytecode (Universal for JVM)
Portability	Not portable (needs recompilation)	Highly portable (runs on any JVM)
Recompilation Needed?	Yes, for each platform	No
Execution Environment	OS-specific	Java Virtual Machine (JVM)
Example Languages	C, C++	Java, Python (with interpreter), .NET (CLR)

Feature	Platform Dependent	Platform Independent
Flexibility	Less flexible	More flexible

What is a Class in Java?

And Why Is Java Code Written Inside a Class?

What is a Class?

A **class** in Java is a **blueprint** or **template** for creating objects.
It defines **data (attributes/variables)** and **behaviors (methods/functions)**.

Syntax of a Class:

```
class ClassName {  
    // Variables (Attributes / Data Members)  
    // Methods (Functions / Behavior)  
}
```

Example:

```
class Car {  
    String brand;  
    int speed;  
  
    void drive() {  
        System.out.println("The car is driving...");  
    }  
}
```

□ Here, Car is a class with:

- **Attributes:** brand, speed
- **Method:** drive()

Why Java Code Must Be Written Inside a Class?

Java is a **pure Object-Oriented Programming (OOP)** language. To enforce **OOP principles** like abstraction, encapsulation, inheritance, and polymorphism, Java requires all code to reside inside **classes**.

Key Reasons:

Reason	Explanation
Encapsulation	Data and methods are bundled together in a class
OOP Compliance	Promotes design based on real-world objects
Reusability	Classes can be reused across programs and projects
Modularity	Easier to organize and maintain code in class-based units
Object Creation	Objects (real instances) are created from classes
Platform Independence	Java classes compile to bytecode which runs on JVM anywhere

Variables in Java

What is a Variable?

A **variable** is a **named memory location** used to store data that can be **changed during program execution**.

Variable Declaration Syntax

dataType variableName = value;

Example:

```
int age = 20;  
String name = "Prathamesh";
```

Types of Variables in Java

Java has **three types** of variables based on where they are declared:

Type	Declared In	Scope	Lifetime
Local	Inside a method/block	Only within that method/block	Until method/block finishes execution
Instance	Inside class, outside any method (no static)	Available in objects	As long as object exists
Static	Inside class, using static keyword	Shared among all objects of class	Exists as long as the program runs

Local Variable Example:

```
void show() {  
    int x = 10; // local variable  
    System.out.println(x);  
}
```

Instance Variable Example:

```
class Student {  
    String name; // instance variable  
}
```

Static Variable Example:

```
class College {  
    static String collegeName = "MIT"; // static variable
```

}

Key Notes:

- **Local variables** must be initialized before use.
 - **Instance variables** are unique for each object.
 - **Static variables** are common for all instances (class-level).
-

Variable Naming Rules

Valid variable names:

- Must begin with a letter, _, or \$
- Can contain letters, digits, _, \$
- Cannot be a **Java keyword**
- Are **case-sensitive**

Data Types in Java

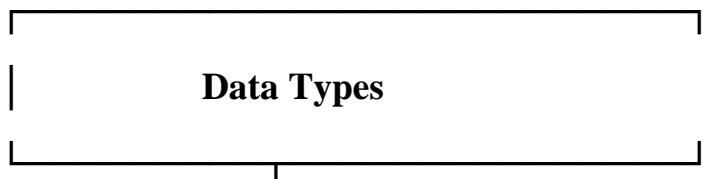
What are Data Types?

A **data type** in Java specifies the **type of data** a variable can store — such as integer, float, character, boolean, etc.

Java is a **statically-typed language**, meaning you **must declare the data type of a variable before using it**.

Types of Data Types in Java

Java data types are broadly categorized into:



Data Types

Primitive Data Types
(8 types, fixed size)

Non-Primitive Data Types
(Classes, Arrays, etc.)

1. Primitive Data Types

Type	Size	Description	Example
byte	1 byte	Smallest integer (-128 to 127)	byte a = 10;
short	2 bytes	Short integer (-32K to +32K)	short s = 1000;
int	4 bytes	Default integer type	int x = 5000;
long	8 bytes	Large integer values	long l = 100000L;
float	4 bytes	Decimal with single precision	float f = 5.75f;
double	8 bytes	Decimal with double precision	double d = 19.99;
char	2 bytes	Single character (Unicode)	char c = 'A';
boolean	1 bit	True/False	boolean isJava = true;

Default Types:

- int for integers
- double for decimal values
- char for characters
- boolean for logical values

2. Non-Primitive (Reference) Data Types

These refer to objects and classes.

They do **not store the value directly**, but reference the memory location.

Examples:

Type	Description	Example
String	Sequence of characters	String name = "Java";
Array	Collection of similar data types	int[] arr = { 1,2,3};
Class	Blueprint for objects	Student s = new Student();
Interface, Enum	Other user-defined types	interface Drawable {}

Notes on Type Conversion

- **Implicit Conversion (Widening):**
Automatically done by Java (e.g., int → long)
- **Explicit Conversion (Narrowing):**
Done manually using casting

```
double d = 9.7;  
int i = (int) d; // i = 9
```

Identifiers in Java

What is an Identifier?

An **identifier** is the **name used to identify** a variable, class, method, object, array, interface, etc., in Java.

It is basically any **user-defined name** for elements in a program.

Examples of Identifiers

```
int age = 25;          // "age" is an identifier  
String name = "Java"; // "name" is an identifier  
  
class Student {       // "Student" is an identifier
```

```
void displayInfo() { // "displayInfo" is an identifier
    System.out.println("Hello");
}
```

Rules for Writing Identifiers in Java

Rule No.	Rule Description	Example
1	Can contain letters (A-Z, a-z), digits (0-9), underscore (_), and dollar sign (\$)	my_name, \$_value
2	Must begin with a letter, underscore (_), or dollar sign (\$)	student1, _id
3	Cannot start with a digit	1name (Invalid)
4	Cannot use Java keywords	int, class
5	Case-sensitive (e.g., name and Name are different)	name ≠ Name
6	No spaces or special characters like @, #, !, etc.	user name

Invalid Identifiers

```
int 1value; // starts with a digit
int class; // Java keyword
int full name; // contains space
int @data; // special character
```

Valid Identifiers

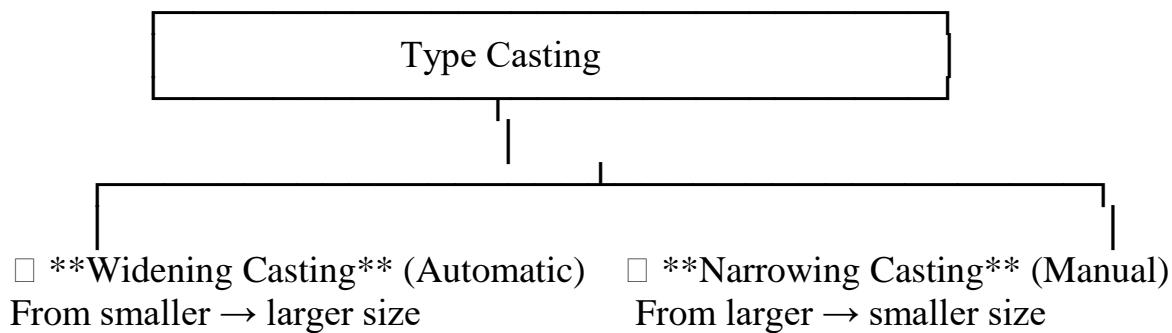
```
int studentAge;  
String _firstName;  
double $salary;  
boolean isJavaFun;
```

Type Casting in Java

What is Type Casting?

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

Java supports two types of type casting:



1. Widening Casting (Implicit Conversion)

Automatically done by the compiler when **no loss of data** is expected.

Small → Large

byte → short → int → long → float → double

Example:

```
int a = 10;
```

```
double b = a; // automatically converted  
System.out.println(b); // Output: 10.0
```

2. Narrowing Casting (Explicit Conversion)

Done **manually** when there's a chance of **data loss** (larger to smaller type).

□ **Syntax:**

```
dataType variableName = (dataType) value;
```

□ **Example:**

```
double d = 9.8;  
int x = (int) d; // explicitly cast  
System.out.println(x); // Output: 9
```

□ Decimal part is **truncated**, not rounded.

Java Operators – Complete Notes

□ **What is an Operator in Java?**

Operators are **symbols** that perform operations on variables and values. Java has a rich set of operators to manipulate data.

1. Arithmetic Operators

Used for **basic mathematical operations**.

Operator	Meaning	Example (a=10, b=3)	Output
+	Addition	a + b	13
-	Subtraction	a - b	7

Operator	Meaning	Example (a=10, b=3)	Output
*	Multiplication	a * b	30
/	Division	a / b	3
%	Modulus	a % b	1

```
int a = 10, b = 3;
System.out.println(a + b); // 13
System.out.println(a % b); // 1
```

2. Relational Operators (Comparison)

Used to **compare two values**. Result is true or false.

Operator	Description	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

```
int a = 10, b = 5;
System.out.println(a > b); // true
```

3. Conditional / Logical Operators

Used to **combine multiple conditions.**

Operator	Meaning	Example
&&	Logical AND	a > 5 && b < 10
'		'
!	Logical NOT	!(a > b)

```
int a = 10, b = 5;
System.out.println(a > 5 && b < 10); // true
System.out.println(!(a < b)); // true
```

4. Bitwise Operators

Work on **bits** and perform bit-by-bit operation.

Operator	Description	Example (a=5, b=3)	Binary	Result
&	AND	a & b	0101 & 0011	0001
'	OR	'a	b'	b'
^	XOR	a ^ b	0101 ^ 0011	0110
~	NOT	~a	~0101	1010
<<	Left shift	a << 1	0101 << 1	1010
>>	Right shift	a >> 1	0101 >> 1	0010

```
int a = 5, b = 3;
System.out.println(a & b); // 1
System.out.println(a | b); // 7
System.out.println(a << 1); // 10
```

5. Ternary Operator (?:)

Short-hand for if-else.

```
int a = 10, b = 20;  
int max = (a > b) ? a : b;  
System.out.println("Max = " + max); // 20
```

Syntax:

```
condition ? expression_if_true : expression_if_false;
```

6. Assignment Operators

Used to **assign values to variables**.

Operator	Example	Equivalent To
=	a = 10	Assign value
+=	a += 5	a = a + 5
-=	a -= 3	a = a - 3
*=	a *= 2	a = a * 2
/=	a /= 4	a = a / 4
%=	a %= 3	a = a % 3

```
int a = 10;  
a += 5;  
System.out.println(a); // 15
```

Java Decision Making – Conditional Statements

What are Decision Making Statements?

Decision-making statements in Java allow you to **execute certain blocks of code based on conditions** (true/false).

Java supports several conditional constructs:

- if
 - if-else
 - if-else-if ladder
 - nested if-else
-

1. if Statement

Executes the block **only if** the condition is true.

Syntax:

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

Example:

```
int age = 20;  
if (age >= 18) {  
    System.out.println("You are eligible to vote.");  
}
```

2. if-else Statement

Executes one block if the condition is true, otherwise another block.

Syntax:

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

```
} else {  
    // code if condition is false  
}
```

□ **Example:**

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

3. if-else-if Ladder

□ Used when there are **multiple conditions** to test one after another.

□ **Syntax:**

```
if (condition1) {  
    // block 1  
} else if (condition2) {  
    // block 2  
} else if (condition3) {  
    // block 3  
} else {  
    // default block  
}
```

□ **Example:**

```
int marks = 75;  
if (marks >= 90) {  
    System.out.println("Grade A");  
} else if (marks >= 75) {  
    System.out.println("Grade B");  
} else if (marks >= 60) {  
    System.out.println("Grade C");  
} else {  
    System.out.println("Fail");  
}
```

}

4. Nested if-else

- You can place an if or if-else **inside another if or else** block.

- **Syntax:**

```
if (condition1) {  
    if (condition2) {  
        // inner if block  
    } else {  
        // inner else block  
    }  
} else {  
    // outer else block  
}
```

- **Example:**

```
int num = 15;  
if (num > 0) {  
    if (num % 2 == 0) {  
        System.out.println("Positive Even Number");  
    } else {  
        System.out.println("Positive Odd Number");  
    }  
} else {  
    System.out.println("Number is not positive");  
}
```

Java Notes – User Input with Scanner Class

- **What is Scanner in Java?**

The Scanner class is part of the `java.util` package and is used to **read input from the user** through the keyboard.

1. Importing Scanner Class

You must import the Scanner class before using it.

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

2. Creating Scanner Object

You can create a Scanner object using:

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

Here, System.in refers to **standard input (keyboard)**.

3. Methods of Scanner Class

Method	Description	Example Input Type
next()	Reads a single word (until space)	Hello
nextLine()	Reads a full line	Hello World
nextInt()	Reads an integer	25
nextDouble()	Reads a double	3.14
nextFloat()	Reads a float	7.5f
nextBoolean()	Reads a boolean value	true/false
nextLong()	Reads a long integer	123456789

5. Note: Common Pitfall with nextLine() After nextInt() or nextDouble()

If you use `nextInt()` or `nextDouble()` and then `nextLine()`, it may skip the line. This is because `nextInt()` leaves a **newline character** in the input buffer.

□ **Fix:**

Use an **extra `nextLine()`** to consume the leftover newline.

```
sc.nextLine(); // consume leftover newline
```

6. Close the Scanner

It's good practice to close the scanner when done:

```
sc.close();
```

Java Notes – Loops in Java

□ **What is a Loop?**

A **loop** in Java is used to **execute a block of code repeatedly** until a specific condition is met.

1. for Loop

Use Case:

When the **number of iterations is known**.

□ **Syntax:**

```
for (initialization; condition; increment/decrement) {  
    // code to execute  
}
```

□ **Example:**

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

}

□ How it Works:

1. Initialization: Run once at the start.
 2. Condition: Checked before every iteration.
 3. Increment/Decrement: Runs after each loop cycle.
-

2.while Loop

Use Case:

When the **number of iterations is not known** in advance.

□ Syntax:

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

□ Example:

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

□ Important:

- If the condition is **false initially**, the loop will **never execute**.
-

3.do-while Loop

Use Case:

When the block **must run at least once**, even if the condition is false.

□ **Syntax:**

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

□ **Example:**

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

□ **Key Difference:**

- Executes the **loop body first**, then checks the condition.

Java Notes – Control Statements: break, continue, return

Java provides control statements that allow you to **alter the flow** of execution within loops or methods.

1. break Statement

Use Case:

- **Terminates** the loop or switch-case immediately.
- Used when an **exit condition** is met before normal loop completion.

□ **Syntax:**

```
break;
```

□ **Example: Break in a loop**

```
for (int i = 1; i <= 10; i++) {  
    if (i == 5) {
```

```
        break; // exits the loop when i is 5
    }
    System.out.println(i);
}
```

□ **Output:**

```
1
2
3
4
```

2. continue Statement

Use Case:

- **Skips** the current iteration and proceeds to the **next iteration**.
- Useful when **some conditions** need to be skipped in a loop.

□ **Syntax:**

```
continue;
```

□ **Example:**

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

□ **Output:**

```
1
2
4
5
```

3.return Statement

Use Case:

- Used to **exit from a method** and optionally **return a value** to the caller.

Syntax (void method):

```
return;
```

Syntax (returning value):

```
return value;
```

Example 1: Returning a value

```
public int add(int a, int b) {  
    return a + b; // returns the sum  
}
```

Java Notes – Types of Variables in Java

In Java, variables are classified based on their **scope**, **lifetime**, and **where they are declared**.

1.Instance Variables

Definition:

- Declared **inside a class** but **outside any method or constructor**.
- Each object of the class **gets its own copy**.
- Also known as **non-static** variables.

Characteristics:

- Belong to the **object (instance)** of the class.
- Memory is allocated **when the object is created**.
- Accessed using the object reference.

□ **Example:**

```
public class Student {  
    // instance variable  
    String name;  
  
    void setName(String n) {  
        name = n;  
    }  
  
    void display() {  
        System.out.println("Name: " + name);  
    }  
}
```

□ **Usage:**

```
Student s1 = new Student();  
s1.setName("Prathamesh");  
s1.display(); // Output: Name: Prathamesh
```

2. Local Variables

Definition:

- Declared **inside a method, constructor, or block.**
- Scope is **limited to that method or block.**
- Must be **initialized before use.**

□ **Characteristics:**

- No default values – must be explicitly initialized.
- Stored in the **stack** memory.
- Cannot use public, private, or static modifiers.

□ **Example:**

```
public class Demo {  
    void show() {  
        int x = 10; // local variable
```

```
        System.out.println("x = " + x);
    }
}
```

3.Static Variables (Class Variables)

Definition:

- Declared with the **static keyword** inside a class.
- Shared among **all instances of the class**.

□ Characteristics:

- Belong to the **class**, not to any specific object.
- Loaded when the class is **first loaded** into memory.
- Accessed using **class name** or object.

□ Example:

```
public class Counter {
    static int count = 0; // static variable

    Counter() {
        count++;
        System.out.println("Count: " + count);
    }
}
```

□ Output on creating 3 objects:

```
Counter c1 = new Counter(); // Count: 1
Counter c2 = new Counter(); // Count: 2
Counter c3 = new Counter(); // Count: 3
```

□ Summary Table

Variable Type	Scope	Stored In	Default Value	Declared Where?
Instance	Per object	Heap	Yes	Inside class, outside method
Local	Inside method/block	Stack	(Must initialize)	Inside method or block
Static	Shared by class	Method Area	Yes	Inside class with static

□ Difference Between Instance, Local, and Static Variables in Java

Feature / Aspect	Instance Variable	Local Variable	Static Variable
Declared in	Inside a class but outside any method or block	Inside a method, constructor, or block	Inside a class with the static keyword
Belongs to	An object	Only inside method/block	Class itself (shared among all objects)
Memory Allocation	At the time of object creation	When method/block is called	When class is loaded into memory
Default Value	Yes (e.g., 0, null, false)	Must be initialized explicitly	Yes (e.g., 0, null, false)
Access Modifier	Can have access modifiers (public, private)	Cannot use access modifiers	Can have access modifiers (public, private)
Used for	Object-level properties	Temporary/short-term calculations/logic	Shared values like counters, configuration, etc.

Feature / Aspect	Instance Variable	Local Variable	Static Variable
Accessed using	Object reference (e.g., obj.name)	Directly inside method	Class name or object (e.g., ClassName.count)
Stored in	Heap memory	Stack memory	Method Area / PermGen (class-level memory)
Scope	Throughout the class (via object)	Only within the method/block it's declared in	Throughout the class, independent of objects

Java Notes – Methods in Java

□ What is a Method?

A **method** in Java is a block of code that performs a specific task. It helps in **code reusability, modularity**, and improves **readability**.

□ Syntax of a Method

```
returnType methodName(parameterList) {  
    // method body  
}
```

1.No Parameter, No Return

Use Case:

Method just performs an action. No input or output.

□ Example:

```
public class Demo {
```

```
void greet() {  
    System.out.println("Hello, Prathamesh!");  
}  
  
public static void main(String[] args) {  
    Demo d = new Demo();  
    d.greet();  
}  
}
```

2. With Parameter, No Return

Use Case:

Takes input (parameter), performs action, but **does not return** result.

□ Example:

```
public class Demo {  
    void printSquare(int num) {  
        System.out.println("Square: " + (num * num));  
    }  
  
    public static void main(String[] args) {  
        Demo d = new Demo();  
        d.printSquare(5); // Output: Square: 25  
    }  
}
```

3. No Parameter, With Return

Use Case:

Does not take input, but **returns** some result.

□ Example:

```
public class Demo {  
    int getRandomNumber() {
```

```
        return 42;
    }

public static void main(String[] args) {
    Demo d = new Demo();
    int result = d.getRandomNumber();
    System.out.println("Number: " + result);
}
```

4. With Parameter, With Return

Use Case:

Takes input, performs operation, and **returns** result.

□ Example:

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

    public static void main(String[] args) {
        Demo d = new Demo();
        int sum = d.add(10, 20);
        System.out.println("Sum: " + sum);
    }
}
```

Java Notes – Arrays in Java + For-each Loop

□ What is an Array?

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

Why Use Arrays?

- Store multiple values in a single variable.
 - Reduce redundancy.
 - Easier data manipulation using loops.
-

Array Declaration & Initialization

□ Syntax:

```
dataType[] arrayName = new dataType[size];
```

□ Example 1: Declaration + Initialization

```
int[] marks = new int[5]; // declaration  
marks[0] = 90; // initialization
```

□ Example 2: Declaration + Initialization in One Line

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

□ Accessing Array Elements

```
System.out.println(numbers[0]); // Output: 10  
System.out.println(numbers[2]); // Output: 30
```

Note: Array indices start from 0 and go up to length - 1.

□ Length of an Array

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

□ Iterating an Array using for Loop

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

}

For-Each Loop in Java

What is For-each Loop?

- Introduced in Java 5
- Used to **iterate over arrays or collections** easily.
- **No index access, read-only** loop.

□ Syntax:

```
for (dataType var : arrayName) {  
    // use var  
}
```

□ Example:

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

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

□ This will print each element one by one from the array.

Comparison: for vs for-each

Feature	for loop	for-each loop
Access	Index-based	Element-based
Read/Write	Can read & write elements	Can only read (no index access)
Simplicity	Slightly more complex	Very simple
Use case	When index is needed (e.g., skip)	When you just want to read elements

