Advance Educational Activities Pvt. Ltd.

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. Java is a versatile, general-purpose programming language designed so developers can build applications once and deploy them on any platform that supports Java, without modifying the compiled code.

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

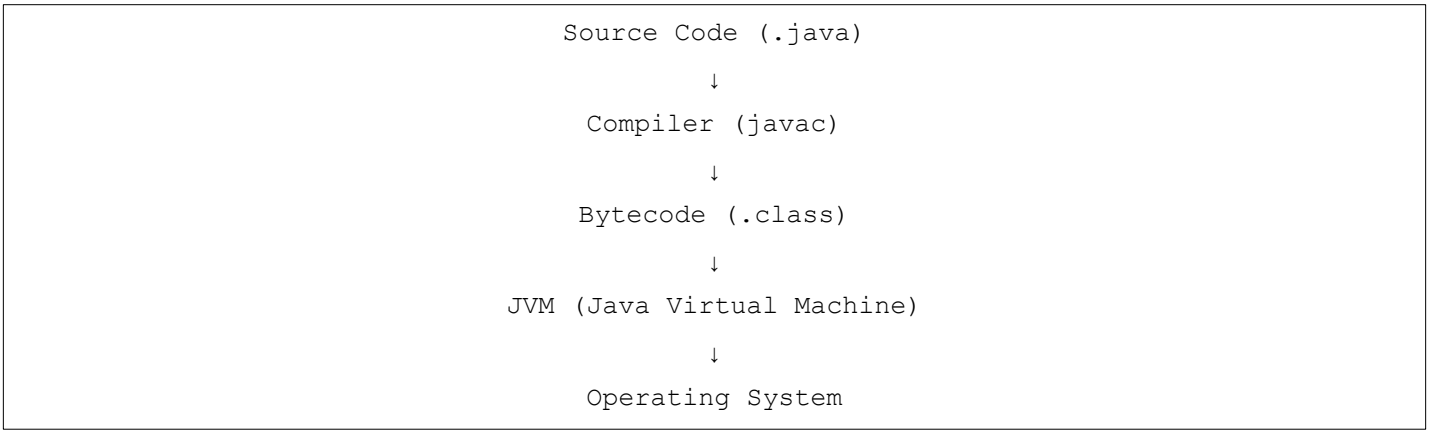
Compilation and Execution Flow:

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
```

© Advanced Educational Activities Pvt. Ltd. 95B, Siddamsetty Complex, Park Lane, Secunderabad – 500003. All rights reserved. No part of this material may be reproduced, in any form or by any means, without permission in writing. This course material is only for the use of bonafide students of **Advanced Educational Activities Pvt. Ltd. (T.I.M.E.)** and its licensees/franchisees and is not for sale. (20 pages) (cael/caem)

1.1.5 Java Architecture



- **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.

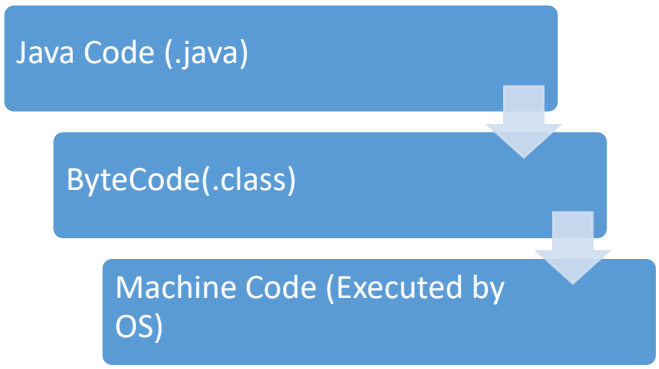
"Write Once, Run Anywhere" (WORA) — Java's core principle.

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:

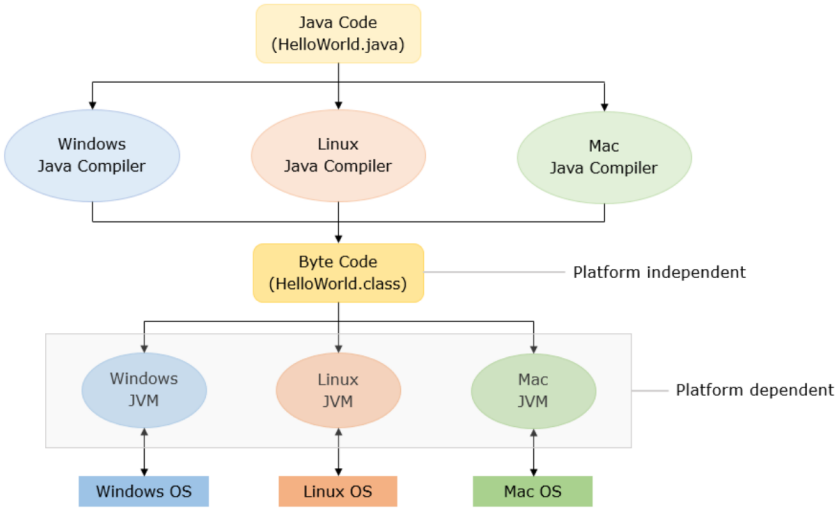
- Your **Java program** is like a **universal device charger**.
- The **bytecode** is like the standard **USB cable**.
- The **JVM** is like a **plug adapter** for each country's socket (India, US, UK).
- You can use the same charger (code) in any country (OS) — you just need the correct adapter (JVM).

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	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

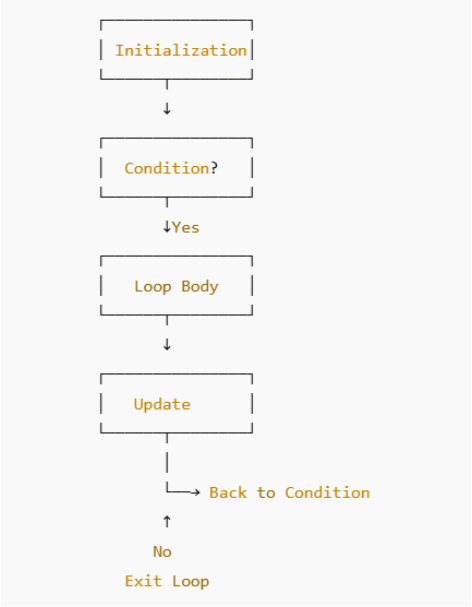
Used when you know **exactly how many times** to loop.

```
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

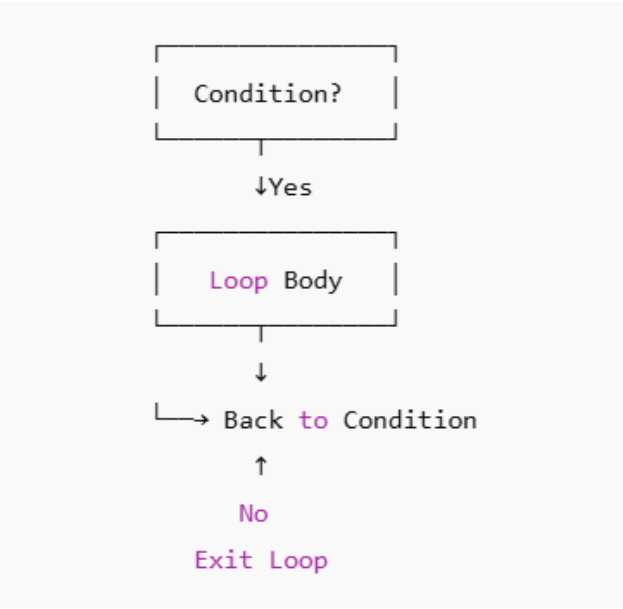
Used when the **number of iterations is not known** beforehand.

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

Example:

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

Flow Chart



1.6.4 do-while Loop

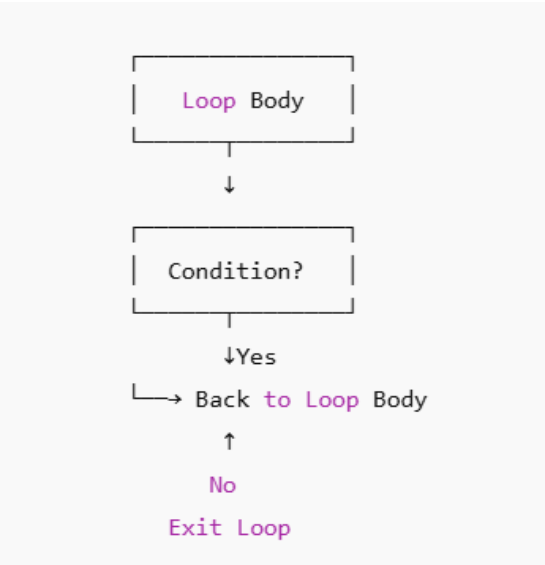
Similar to while, but it **executes at least once**, even if condition is false.

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

Example:

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

Flow Chart



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.